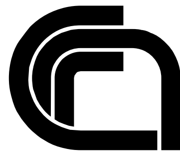


Consiglio Nazionale delle Ricerche

A framework for automatic security controller generation

F. Martinelli, I. Matteucci

IIT TR-03/2009

Technical report

marzo 2009

Istituto di Informatica e Telematica

A framework for automatic security controller generation *

Fabio Martinelli[1], Ilaria Matteucci[1,2]
[1]Istituto di Informatica e Telematica - C.N.R., Pisa, Italy
[2]Create-Net, Trento, Italy
e-mail:{Fabio.Martinelli, Ilaria.Matteucci}@iit.cnr.it

Abstract

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

A complex system under analysis is described as an *open* system, in which a certain component has an unspecified behavior (not fixed in advance). Regardless of the unspecified behavior, the system should work properly, *e.g.*, should satisfy a certain property. Within this formal approach, we propose techniques to enforce properties and synthesize controller programs able to guarantee that, for all possible behaviors of the unspecified component, the overall system results secure.

For performing this task, we use techniques able to provide us necessary and sufficient conditions on the behavior of this unspecified component to ensure the whole system is secure. Hence, we automatically synthesize the appropriate controller programs by exploiting satisfiability results for temporal logic.

We contribute within the area of the enforcement of security properties by proposing a flexible and automated framework that goes beyond the definition of how a system should behave to work properly. Indeed, while the majority of related work focuses on the definition of monitoring mechanisms, we aid in the synthesis of enforcing techniques. Moreover, we present a tool for the synthesis of secure systems able to generate a controller program directly executable on real devices as smart phones.

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

1 Introduction

The diffusion of distributed systems and networks has increased the amount of information and sensible data that circulate on the net. This is one of the important reasons that stimulates the research towards *secure systems*.

In our framework, a *secure system* is a system that satisfies some *security properties* specifying acceptable executions of programs. For example, a security property might concern either *access control*, that specifies what operations individuals can perform on objects, or *availability*, that prohibits to an entity the use of a source, as a result of execution of that source by other entities.

In particular, this paper concerns the *synthesis of secure systems*.

One remarkable *synthesis* problem has been described by Merlin and Bochman in [1]: It occurs when one deals with a system in which there are some unspecified components, *e.g.*, a not completely implemented software. When considering a partially specified system, represented by the term $S()$, one may wonder if there exists an implementation Y that can be plugged into the system by satisfying some properties of the whole system. Hence the problem that must be solved is the following one:

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

where ϕ is a logic formula representing the property to be satisfied, Y represents the actual behavior of the undefined component, $S(Y)$ represent the composed and completely specified system, and $\models$ represents the truth relation of the logical language used to express the properties. This problem is also named submodule construction. (Note that if we consider S as an empty system, then the synthesis problem amounts to classical satisfiability in logic, *i.e.*, $\exists Y\ Y \models \phi$.)

*Work partially supported by EU-funded project "Software Engineering for Service-Oriented Overlay Computers"(SENSORIA) and by EU-funded project "Secure Software and Services for Mobile Systems"(S3MS). This work is an expanded and revised version of [51, 52, 53].

The problem of the *synthesis of secure systems* we deal with in this paper is slightly different. Let us consider a system that we want to be secure. We study it by exploit the *open system paradigm* and *process algebras* (see [5]). A system is said to be *open* if it has some unspecified components. In accordance to the approach proposed in [2, 3, 4], the open system paradigm is suitable for doing security analysis by considering the unspecified components of the system as potential attackers.

Hence, given a system S and a security property expressed by a logic formula ϕ, our goal is to guarantee that S is secure, *i.e.*, S satisfies the formula ϕ, against whatever possible intruder or malicious user, hereafter denoted by X, is put in parallel execution with it, represented by $S \| X$, where $\|$ is the CCS process algebra *parallel operator* (see [5]). More formally, the security verification goal is to check that:

$$\forall X \quad S \| X \models \phi \tag{1}$$

What does it happen if the systems is not secure?

As a matter of fact, in the case the requirement 1 does not hold, we might wonder if there exists an implementation Y that, by controlling the behavior of the unspecified component X, guarantees the overall system satisfies the required security property, *i.e.*,

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

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

We show how to automatically generate an implementation Y able to guarantee that the open system results secure. This solves our problem of secure system synthesis.

Through our framework, we are able to enforce a lot of significant security property. For instance, *access-control policies*: The set of proscribed partial executions contains those partial executions ending with an unacceptable operation being attempted. There is no way to "unaccess" the resource and fix the situation afterward. since once a restricted resource has been accessed, the policy is broken. There is no way to "unaccess" the resource and fix the situation afterward. Also some *bounded availability* properties may be characterized as safety ones. An example is "one principal cannot be denied the use of a resource for more then D steps as a results of the use of that resource by other principals". Here, the defining set of partial executions contains intervals that exceed D steps and during which a principal is denied use of a resource. Another example of a global security property is the *Chinese Wall policy*. It says that, let A and B two sets of elements. Once one accesses to an element in A, he cannot access to B and viceversa. Here we consider that A and B are sets of files and we consider the action *open*.

The automatic generation procedure, presented in this paper, starts with the application of the *partial model checking* function (see [6, 7]) to the above equation, in order to evaluate the formula ϕ by the behavior of S. In this way we obtain a new formula $\phi' = \phi_{//S}$ that deals only with the un-trusted part of the system, here X. Thus, we study whether a potential attacker exists and, in particular, which are the necessary and sufficient conditions that this enemy should satisfy for altering the correct behavior of the system. Hence, in order to force X to behave correctly, *i.e.*, as prescribed by ϕ', we appropriately use controller operators, denoted $Y \triangleright X$.

The application of partial model checking represents an advantage of our approach for enforcing security properties. Indeed, in our framework, we are able to control only the possible un-trusted component of a given system yet ensuring the overall security. Other approaches deal with the problem of monitoring the possible un-trusted component to enjoy a given property, by treating it as the whole system of interest. However, it is frequent the case where not the entire 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.

Another advantage of our approach is that it allows to automatically synthesize a controller program Y for a controller operator $Y \triangleright X$, by exploiting satisfiability procedures for temporal logic.

To sum up, in this work we show a method to synthesize a controller program Y for a controller operator $\triangleright$ (provided it holds a mild assumption).

Moreover, here, we propose a general framework to enforce security properties that is able to deal with several problems, as parameterized systems and composition of security properties.

Finally, we have developed a tool that permits to generate a controller program for a specified controller operator. As a matter of fact, we have implemented a synthesis tool in the objective language O'caml [8] that, given a system, a security property, and a controller operator enforcing that property, is able to generate the respective controller program. We have also a translator from our internal representation to a policy language called *ConSpec*, developed in the ambit of the S3MS European project (see [46]). For this policy language we developed a running execution monitoring environment for mobile phones with Java applications. The results shown in this paper can be directly used in that framework.

This paper is organized as follows: Section 2 compare our work with other work already appeared in literature. Section 3 recalls some useful background notions. Section 4 proposes our mechanism for the synthesis of controller program able to enforce such security properties. Section 5 describes the tool we have developed in order to automatically generate controller programs, given the system, the properties and which controller operator we are going to use. Section 6 presents some further results on composition of properties and parameterized systems and Section 7 concludes the paper and proposes some future work.

2 Logic, Process Algebra and Partial Model Checking

We start by recalling some basic notions about the *equational μ-calculus* (see [6, 7]) a modal logic suitable for expressing in a qualitative ways relations among events.

Successively, we describe the CCS process algebra (see [5]), that belongs to formalisms for the description of concurrent communicating processes (see also [30, 31]). We present the CCS operational semantics in the style proposed by Plotkin (see [29]), namely *Structured Operational Semantics* (SOS).

Finally, we show the compositional analysis techniques proposed by Andersen (see [6, 7]) that deals with partially specified systems by introducing the *partial model checking* function.

2.1 Equational μ-calculus

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

Let a be in Act, Z be a variable ranging over a set of variables V. The syntax of the assertions (ϕ) and of the lists of equations (D) is given by the following grammar:

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

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

where $\mathbf{T}$ and $\mathbf{F}$ are the logical constant *true* and *false*, respectively; $\wedge$ and $\vee$ are the classical symbols for conjunction and disjunction. The possibility modality $\langle a \rangle \phi$ expresses the ability to have an a transition to a state that satisfies ϕ. The necessity modality $[a]\phi$ expresses that after each a transition there is a state that satisfies ϕ. The fixpoint equation 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 are denoted as $Def(D)$. A list of equations is closed if every variable that appears in the assertions of the list is in $Def(D)$.

The semantics of equational μ-calculus is given by means of *labelled transition systems* (LTSs) where transitions are labelled to describe the actions which cause a change in the state.

LTSs consist of a set of states, a set of labels (or *actions*) and a *transition relation*, $\rightarrow$, that describes how a process passes from a state to another, *i.e.*, given two states s an s' and a label a, $s \stackrel{a}{\longrightarrow} s'$ means that the process passes from the state s to the state s' with an arc labeled a.

Definition 2.1 *A triple $\langle S, Act, \rightarrow \rangle$ is called* Labelled Transition System *(LTS), where S is a set of states, Act is a set of actions (or labels) and $\rightarrow \subseteq S \times Act \times S$ is a ternary relation, known as a* transition relation*.*

Let $M = \langle S, Act, \rightarrow \rangle$ be an LTS, where $\rightarrow$ is the transition relation, ρ be an environment that assigns subsets of S to variables that appear in all assertions of D, but which are not in $Def(D)$. Then, the semantics $[\![\phi]\!]_\rho$ of an assertion ϕ is the same as for μ-calculus assertions and the semantics $[\![D]\!]_\rho$ of a definition list is an environment which assigns subsets of S to variables in $Def(D)$. The semantics of the equational μ-calculus is in Figure 1. Informally, the semantics definition of the fixpoint says that the solution to $(Z =_\sigma \phi)D$ is the σ 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 ρ 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.

$$
\begin{array}{ll}
[\![\mathbf{T}]\!]_\rho = S & [\![\mathbf{F}]\!]_\rho = \emptyset \\
[\![\phi_1 \wedge \phi_2]\!]_\rho = [\![\phi_1]\!]_\rho \cap [\![\phi_2]\!]_\rho & [\![\phi_1 \vee \phi_2]\!]_\rho = [\![\phi_1]\!]_\rho \cup [\![\phi_2]\!]_\rho \\
[\![\langle\alpha\rangle\phi]\!]_\rho = \{s | \exists s' : s \xrightarrow{\alpha} s' \text{ and } s' \in [\![\phi]\!]_\rho\} & [\![[\alpha]\phi]\!]_\rho = \{s | \forall s' : s \xrightarrow{\alpha} s' \text{ implies } s' \in [\![\phi]\!]_\rho\} \\
[\![\epsilon]\!]_\rho = [] & [\![X]\!]_\rho = \rho(X) \\
[\![X =_\sigma \phi D']\!]_\rho = [\![D']\!]_{(\rho \sqcup [U'/X])} \sqcup [U'/X] &
\end{array}
$$

where $U' = \sigma U.[\![\phi]\!]_{(\rho \sqcup [U/X] \sqcup \rho'(U))}$ and $\rho'(U) =[\![\text{D'}]\!]_{(\rho \sqcup [U/X])}$. $\sqcup$ represents union of disjoint environments.

Figure 1: Denotational semantics of equational μ-calculus.

2.1.1 Examples and facts

A lot of properties can be defined by using equational μ-calculus. In particular it is useful to express several security properties, *e.g.*, a safety property that expresses the possibility to open a new file only if the previous one is closed:

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

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

Another interesting formula is $\nu Z.[K]Z \wedge [Act \backslash K]\mathbf{F}$[1] which expresses the fact that only actions in K can be performed by a process in any reachable state.

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

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

Moreover the *Chinese Wall* policy. This policy says that, let A and B two sets of elements. Once one accesses to an element in A, he cannot access to B and viceversa. Here we consider that A and B are sets of files and we consider the action *open*. This can be expressed by the formula $\phi = \phi_1 \vee \phi_2$ where ϕ_1 and ϕ_2 are the following two formulas respectively:

$$
\begin{array}{lll}
\phi_1 = \text{LET MAX} & W = & [open_A]W \wedge [open_B]\mathbf{F}\text{IN}W \\
\phi_2 = \text{LET MAX} & V = & [open_B]V \wedge [open_A]\mathbf{F}\text{IN}V
\end{array}
$$

As a matter of fact ϕ is a disjunction between two different formulas ϕ_1 and ϕ_2 that cannot be both true at the same time. Indeed ϕ_1 permits to open only file in A, on the other hand ϕ_2 allows the access to elements in B.

The same consideration can be done for the specification of a *liveness property*. For instance, the property "a state satisfying ϕ can be reached" is expressed by $Z =_\mu \langle _ \rangle Z \vee \phi$, where $\langle _ \rangle$ means $\langle Act \rangle$, *i.e.*, $\bigvee_{a \in Act} \langle a \rangle$ [2].

For this calculus we have the following satisfiability result.

Theorem 2.1 ([32]) *Given a formula ϕ, it is possible to decide within exponential time in the length of ϕ if there exists a model of ϕ and it is also possible to give an example of such model.*

2.2 Process algebra: CCS

Process algebras (see [30, 31]) are approaches to formally model concurrent systems. They provide a method for the high-level description of interactions, communications, and synchronization between independent entities.

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

[2] In writing properties, here and in the rest of the paper, we use the shortcut notations $[_]$ means $[Act]$, *i.e.*, $\bigwedge_{a \in Act}[a]$

Prefixing: $\frac{}{a.P \xrightarrow{a} P}$ Choice: $\frac{P \xrightarrow{a} P'}{P+Q \xrightarrow{a} P'}$ $\frac{Q \xrightarrow{a} Q'}{P+Q \xrightarrow{a} Q'}$

Parallel: $\frac{P \xrightarrow{a} P'}{P\|Q \xrightarrow{a} P'\|Q}$ $\frac{Q \xrightarrow{a} Q'}{P\|Q \xrightarrow{a} P\|Q'}$ $\frac{P \xrightarrow{l} P' \quad Q \xrightarrow{\bar{l}} Q'}{P\|Q \xrightarrow{\tau} P'\|Q'}$ Restriction: $\frac{P \xrightarrow{a} P'}{P \backslash L \xrightarrow{a} P' \backslash L}$

Relabelling: $\frac{P \xrightarrow{a} P'}{P[f] \xrightarrow{f(a)} P'[f]}$ Constant: $\frac{P \xrightarrow{a} P'}{A \xrightarrow{a} P'}$

Figure 2: *SOS* system for CCS.

A process calculus of our interest is the CCS (or *Calculus of Communicating Systems*) developed by Robin Milner. The main notion of CCS is the communication between processes, that is a synchronous one. Both the processes must agree on performing the communication at the same time, and communication is modelled by a simultaneous performing of complementary actions (*e.g.*, send-receive actions). This event is represented by a synchronization action (or internal action) τ.

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

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

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

where $L \subseteq Act$; A denotes a process constant equipped with its formal definition $A = P$ and eventually the relabelling function $f : Act \mapsto Act$ must be such that $f(\tau) = \tau$

In order to give (operational) semantics of CCS terms (see [5]), we use the *Structural Operational Semantics* (SOS, for short) proposed by Plotkin (see [29]). The operational semantics explicitly describes how programs compute in a stepwise fashion, and the possible state-transformations they can perform. Moreover, this method has a logical flavor and permits to compositionally reason about the behavior of programs.

It is described by a labelled transition system $(\mathcal{E}, Act, \rightarrow)$, where $\mathcal{E}$ is the set of all CCS terms and $\rightarrow \subseteq \mathcal{E} \times Act \times \mathcal{E}$ is a transition relation defined by structural induction as the least relation generated by the set of the structural operational semantics rules of Figure 2. The transition relation $\rightarrow$ defines the usual concept of derivation in one step. As a matter of fact $P \xrightarrow{a} P'$ means that process P evolves in one step into process P' by executing action $a \in Act$. The transitive and reflexive closure of $\bigcup_{a \in Act} \xrightarrow{a}$ is written $\rightarrow^*$.

Informally, the meaning of CCS operators is the following:

0: is the process that does nothing.

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

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

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

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

Relabelling: the process $P[f]$ behaves like P, but its actions are renamed through relabelling function f.

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

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

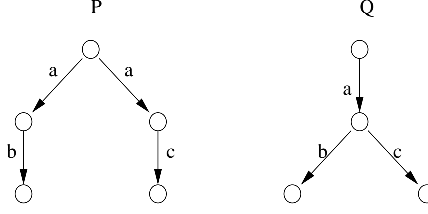


Figure 3: Example of two processes in which "Q *simulates* P".

2.2.1 Behavioral equivalence

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

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

Strong bisimulation We recall the *simulation* pre-order and the related bisimulation congruence (see [5, 39]).

Let us consider the following example.

Example 2.1 *Consider two vendor machines P and Q which behaviors can be represented by their $LTSs$ (see Figure 3). The first process is forced to perform a b or c action at the beginning of its computation,* i.e.*, when it decides which a action to follow, while the latter after performing the a action; if this action can influence the choice of the following behavior of the process then it is reasonable to consider that the second process has a more decisional power.*

■

As we can see from the graphical representation in Figure 3 are not equal. To compare their behavior we introduce the notion of *simulation*. According to it, Q can simulate P, but the contrary does not hold. Informally, saying that "Q simulates P" means that Q's behavior pattern is at least as rich as that of P.

More formally, we can define the notion of *strong simulation/bisimulation* by following Park's definition [39] as follows.

Definition 2.2 *Let $(\mathcal{E}, Act, \rightarrow)$ be an LTS of concurrent processes over the set of actions Act, and let $\mathcal{R}$ be a binary relation over $\mathcal{E}$. Then $\mathcal{R}$ is called* strong simulation*, denoted by $\prec$, over $(\mathcal{E}, Act, \rightarrow)$ if and only if, whenever $(P, Q) \in \mathcal{R}$ we have:*

$$\textit{if } P \stackrel{a}{\longrightarrow} P' \textit{ then } \exists\, Q' \textit{ s.t. } Q \stackrel{a}{\longrightarrow} Q' \textit{ and } (P', Q') \in \mathcal{R}$$

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

Definition 2.3 *A* strong bisimulation *is a relation $\mathcal{R}$ such that both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are strong simulations,* i.e.*, if for each $(P, Q) \in \mathcal{R}$ and for each $a \in Act$:*

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

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

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

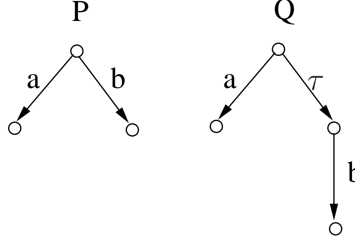


Figure 4: Example of two not observationally bisimilar processes.

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

Look at the following example.

Example 2.2 *The processes P and Q in Figure 4 cannot be consider equivalent, since the second performs an internal action by reaching a state where an action a is no longer possible. Thus, the non visible behavior of the system, represented by the τ action, can modify its visible behavior.*

■

To compare this kind of processes, Milner, in [5], proposed the notion of *observational equivalence*, or *weak bisimulation*.

Let us consider $a \neq \tau$, $\hat{a} = a$, and $\hat{\tau} = \epsilon$. Then, we use the notation $P \stackrel{\tau}{\Longrightarrow} P'$ ($P \stackrel{\epsilon}{\Longrightarrow} P'$) in order to denote that P and P' belongs to the reflexive and transitive closure of τ. Also, $P \stackrel{\hat{a}}{\Rightarrow} P'$ if $P \stackrel{\epsilon}{\Rightarrow} P_\epsilon \stackrel{\hat{a}}{\rightarrow} P'_\epsilon \stackrel{\epsilon}{\Rightarrow} P'$ where P_ϵ and P'_ϵ denote intermediate states[3].

We can give the following definition:

Definition 2.4 *Let $(\mathcal{E}, Act, \rightarrow)$ be an LTS of concurrent processes over the set of actions Act, and let $\mathcal{R}$ be a binary relation over $\mathcal{E}$. Then $\mathcal{R}$ is called* weak simulation*, denoted by $\preceq$, over $(\mathcal{E}, Act, \rightarrow)$ if and only if, whenever $(P, Q) \in \mathcal{R}$ we have:*

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

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

Definition 2.5 *A* weak bisimulation *is a relation $\mathcal{R}$ such that both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are weak simulations,* i.e.*, if for each $(P, Q) \in \mathcal{R}$ and for each $a \in Act$:*

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

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

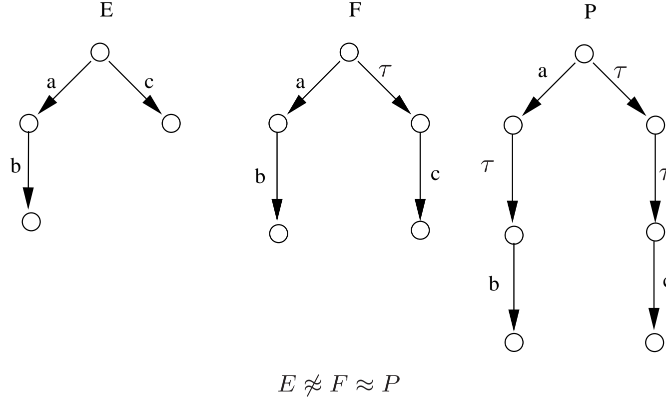
Two processes P and Q are weakly bisimilar if there exists a bisimulation $\mathcal{R}$ such that $(P, Q) \in \mathcal{R}$. The maximal weak bisimulation is $\approx$ which is the union of every weak bisimulation. It is easy to check that this relation is still a weak bisimulation and moreover is reflexive,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

An important result proved by Milner is the following.

Proposition 2.1 ([5]) *Every strong simulation is also a weak one.*

Example 2.3 *Let us consider the processes E, F and P of Fig. 5. F and P are weakly bisimilar, while E and F (P) are not.*

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



$E \not\approx F \approx P$

Figure 5: Example of observational equivalence between different processes.

■

Bisimulation is a very interesting equivalence. It is decidable in polynomial time for finite-state processes, [40]. Moreover, proving that two processes P and Q are bisimilar can be done by quite elegant proof techniques. Actually, it is sufficient to provide a bisimulation $\mathcal{R}$ such that $(P, Q) \in \mathcal{R}$.

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

A *characteristic formula* is a formula in equational μ-calculus that completely characterizes the behavior of a state-transition graph or of a state in a graph modulo a chosen notion of behavioral relation. Here, we recall the definition of the characteristic formula of a finite-state process, by following the approach studied in [41].

Definition 2.6 *Given a finite-state process P, its* characteristic formula *with respect to strong bisimulation is given by the closed list $D_P \downarrow Z_P$ where for every $P' \in Der(P)$, $a \in Act$:*

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

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

Let $\langle\langle a \rangle\rangle$ be a weak version of the modality $\langle a \rangle$, introduced as abbreviation and defined as follows (see [41]):

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

Now we are able to give the following definition.

Definition 2.7 *Given a finite-state process P, its characteristic formula with respect to weak bisimulation is given by the closed list $D_P \downarrow Z_P$ where for every $P' \in Der(P)$, $a \in Act$:*

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

The following lemma characterizes the power of these formulas.

Lemma 2.1 ([35]) *Let P_1 and P_2 be two different finite-state processes. If ϕ_{P_2} is characteristic for P_2 then:*

1. *If $P_1 \approx P_2$ then $P_1 \models \phi_{P_2}$;*

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

Following a similar reasoning to the one in [41] for defining characteristic formulas with respect to bisimulation, it is possible to define a characteristic formula for a given process also with respect to strong and weak simulation.

Definition 2.8 *Given a finite-state process P, its characteristic formula with respect to strong and weak simulation is given by the closed list $D_P \downarrow Z_P$ where, respectively:*

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

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

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

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

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

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

Following a similar reasoning made in [41], it is possible to prove that the following proposition holds because of the definition of the characteristic formula does not depend on Q.

Lemma 2.2 *Let P and Q be a finite-state process and let $\phi_{P,\prec}$ be the characteristic formula of the process P with respect to strong simulation then:*

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

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

2.3 Compositional analysis: the *partial model checking*

In this section we recall the theory on compositionality developed in [7]. The problem under consideration is the following:

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

This kind of problem can be found, for instance, when a large system is developed. Since the implementation cannot be immediately extracted from the specification, the implementation phase consists of a large number of small refinements of the initial specification until, eventually, the implementation can be clearly identified. In [7], Andersen has proposed the *partial model checking* mechanism in order to give a compositional method for proving properties of concurrent systems, *i.e.*, the task of verifying an assertion for a composite process is decomposed into verification tasks for the subprocesses.

The intuitive idea underlying the partial model checking is the following: proving that $P \| Q$ satisfies an equational μ-calculus formula ϕ is equivalent to prove that Q satisfies a modified specification $\phi_{//_P}$, where $//_P$ is the partial evaluation function for the parallel composition operator (see [7] or Table 6). The formula ϕ is specified by use the equational μ-calculus.

Hence, the behavior of a component is partially evaluated and the requirements are changed in order to respect this evaluation. The partial model checking function (also called *partial evaluation function*) for the parallel operator is given in Figure 6. In order to explain better how partial model checking function acts on a given equational μ-calculus formula, we show the following example.

Example 2.4 *Let $[\tau]\phi$ be the given formula and let $P \| Q$ be a process. We want to evaluate the formula $[\tau]\phi$ w.r.t. the $\|$ operator and the process P. The formula $[\tau]\phi_{//_P}$ is satisfied by Q if the following three conditions hold at the same time:*

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

$(D\downarrow Z)/\!\!/t = (D/\!\!/t)\downarrow Z_t$ $\quad\quad$ $\epsilon/\!\!/t = \epsilon$

$(Z =_\sigma \phi D)/\!\!/t = ((Z_s =_\sigma \phi//s)_{s\in Der(E)})(D)/\!\!/t$ $\quad\quad$ $Z/\!\!/t = Z_t$

$\phi_1 \wedge \phi_2/\!\!/s = (\phi_1/\!\!/s)\wedge(\phi_2/\!\!/s)$ $\quad\quad$ $\phi_1 \vee \phi_2/\!\!/s = (\phi_1/\!\!/s)\vee(\phi_2/\!\!/s)$

$[a]\phi/\!\!/s = [a](\phi/\!\!/s)\wedge \bigwedge_{s\xrightarrow{a}s'} \phi/\!\!/s'$, if $a \neq \tau$ $\quad\quad$ $\langle a\rangle\phi/\!\!/s = \langle a\rangle(\phi/\!\!/s)\vee \bigvee_{s\xrightarrow{a}s'} \phi/\!\!/s'$, if $a \neq \tau$

$[\tau]\phi/\!\!/s = [\tau](\phi/\!\!/s) \wedge \bigwedge_{s\xrightarrow{\tau}s'} \phi/\!\!/s' \wedge \bigwedge_{s\xrightarrow{a}s'} [\overline{a}](\phi/\!\!/s')$ $\quad\quad$ $\mathbf{T}/\!\!/s = \mathbf{T}$

$\langle\tau\rangle\phi/\!\!/s = \langle\tau\rangle(\phi/\!\!/s) \vee \bigvee_{s\xrightarrow{\tau}s'} \phi/\!\!/s' \vee \bigvee_{s\xrightarrow{a}s'} \langle\overrightarrow{a}\rangle(\phi//_{s'})$ $\quad\quad$ $\mathbf{F}/\!\!/s = \mathbf{F}$

where t is the rest of the consider process and s is the state in which we do the reduction.

Figure 6: Partial evaluation function for parallel operator.

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

■

In [7], the following lemma is given.

Lemma 2.3 *Given a process $P\|Q$ (where P is a finite-state process) and an equational specification $D\downarrow Z$ we have:*

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

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

Remarkably, this function is exploited in [7] to perform model checking efficiently, where both P and Q are specified. In our setting, the process Q is not specified. Thus, given a certain system P, it is possible to find the property that the enemy must satisfy to avoid a successful attack on the system. It is worth noticing that partial model checking function may be automatically derived from the semantics rules used to define a language semantics. Thus, the proposed technique is very flexible. According to [7], when ϕ is *simple*, *i.e.*, it is of the form X, $\mathbf{T}$, $\mathbf{F}$, $X_1 \wedge \ldots \wedge X_k \wedge [a_1]Y_1 \wedge \ldots \wedge [a_l]Y_l$, $X_1 \vee \ldots \vee X_k \vee \langle a_1\rangle Y_1 \vee \ldots \vee \langle a_l\rangle Y_l$, then the size of $\phi_{//P}$ is bounded by $|\phi| \times |P|$. Referring to [6], any assertion can be transformed to an equivalent simple assertion in linear time. Hence, we can conclude that the size of $\phi_{//P}$ is polynomial in the size of ϕ and P.

It is important to note that a lemma similar to Lemma 2.3 holds for each CCS operators.

3 Synthesis of Run-Time Controller Programs

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

The technical proofs of the results in this section are in the Appendix.

3.1 Specification and verification of secure systems

Following the approach proposed in [2, 3], we describe a methodology for the formal analysis of secure systems based on the concept of open systems and partial model checking techniques.

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

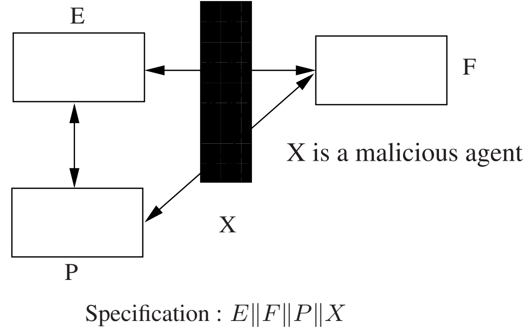


Figure 7: Graphical representation of a possible open system scenario.

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

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

Example 3.1 *We suppose to have a system S in which three processes E, F and P work in parallel. In order to be sure that S works as we expected, we consider that a possible malicious agent X works in parallel with E, F and P as we can see in Fig. 7. In this case the possible intruder, here denoted by X, is able to interact with the other components in order to make the system unsafe. For that reason, instead to consider and analyze the system $S = E\|F\|P$, we study $S = E\|F\|P\|X$ and we require that S is safe whatever the behavior of X is.*

■

The main idea is that, when analyzing security-sensitive systems, neither the enemy's behavior nor the malicious users' behavior should be fixed beforehand. A system should be secure regardless of the behavior that the malicious users or intruders may have, which is exactly a *verification* problem of open systems. According to [2, 3], 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 ϕ is a (temporal) logic formula expressing the security property. It roughly states that the property ϕ holds for the system S, regardless of the unspecified component which may possibly interact with it. By using partial model checking it is possible to reduce such a verification problem as in Statement (2) to a validity checking problem as follows:

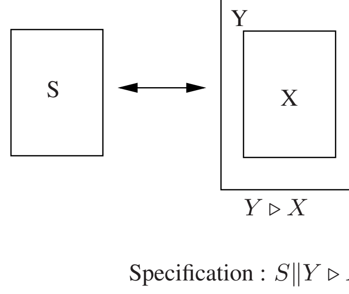
$$\forall X \quad S\|X \models \phi \quad \text{iff} \quad X \models \phi_{//_S} \tag{3}$$

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

3.2 Synthesis of process algebra controller operators

According to the Statement (3), 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 Statement (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. Hence, we use controller operators in such a way the specification of the system becomes:

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



Specification : $S \| Y \triangleright X$

Figure 8: A graphical representation of how a controller program Y works.

By partially evaluating ϕ with respect to S the Statement (4) is reduced as follows:

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

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

In this way the behavior of the safe and known part of the system is considered directly into the formula ϕ'. The problem described by the Statement (5) is about the target system X and the controller program Y. The controller program has to work only on X and does not care about the rest of the system.

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

Hence, given a system, that we want to be secure, a security property that we want to enforce and a controller operator we are going to use in order to do that, we show how synthesize a controller program for the given controller operator (by assuming it satisfies a mild assumption).

The equation in Statement (5) might not be easy to manage because of the presence of the universal quantification on all possible behaviors of the target X. For that reason, firstly, we underline that, by $\triangleright$ operators, we are goring to enforce safety properties. Hence we restrict ourselves to consider a subclass of equational μ-calculus formulas that we call Fr_μ. It consists of equational μ-calculus formulas without $\langle _ \rangle$ operator. It is easy to prove that, according to the rule of the partial evaluation function with respect to parallel operator, this set of formulas is closed under the partial model checking function.

Now, let us consider again the problem in Statement (5). We have the universal quantification on all possible target behavior. In order to manage it we make the following assumption:

Assumption 3.1 *For every X and Y, we have:*

$$Y \triangleright X \preceq Y$$

In the following we present a method to synthesize controller program Y for controller operator whose semantics definition satisfies the Assumption 3.1.

Firstly we give the following result.

Proposition 3.1 ([42]) *Let E and F be two processes and $\phi \in Fr_\mu$. If $F \preceq E$ then $E \models \phi \Rightarrow F \models \phi$. The same result holds also if F and E are strong similar.*

Hence, for safety properties expressed by a formula in Fr_μ, whether it is used a controller operators that satisfies the Assumption 3.1 in order to enforce such safety properties, the problem in the Statement (5) can be equivalently reduced as follows:

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

according to the fact that the set Fr_μ is closed for partial model checking.

The formulation (6) is easier to be managed. In particular, in this way, we have reduced a validity problem to a satisfiability one in μ-calculus. Hence a possible model Y for ϕ' can be find according to the Theorem 2.1. So we are able to prove the following result

Theorem 3.1 *The problem described in Formula (5) is decidable.*

Note that the trivial solution exists. As a matter of fact the process **0** is a model for all possible formulas in Fr_μ, *i.e.*, for every process P, $0 \preceq P$, hence, according to the Proposition 3.1, $0 \models \phi$. Obviously this is the easiest solution, however, it is possible to find more complex model for ϕ by exploiting satisfiability procedure, *e.g.*, the one developed by Waluckiewicz in [43].

By using formulas in Fr_μ we are able to express several safety properties that are meaningful in the security scenario. For instance, access control policies, that we have already described before, are safety properties, as also, bounded availability and Chinese Wall policies, whose we refer as an example of how our mechanism works.

In order to understand how our method works, let us consider, for instance, a controller operator with the following semantics definition:

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

We are able to prove that

Proposition 3.2 *For the controller operator $\triangleright$ the Assumption 3.1 holds,* i.e.*,*

$$Y \triangleright X \preceq Y$$

This provides that, whatever the controller operator $\triangleright$ is chosen to enforce a given safety properties, it is possible to find a solution for the problem in Formula (5) by finding a controller program Y such that:

$$Y \models \phi'$$

where ϕ' is a formula in Fr_μ.

3.2.1 Feasibility issues for our controllers

The introduction of a controller operator helps to guarantee a correct behavior of the entire system.

Several semantics definitions can be given to describe the behavior of possible controller operators. According to the semantics it is possible to establish if a controller operator is implementable or not. For instance, consider the following controller operator, $\triangleright'$:

$$\frac{E \xrightarrow{a} E' \; F \xrightarrow{a} F'}{E \triangleright' F \xrightarrow{a} E' \triangleright' F'} \quad \frac{E \xrightarrow{a} E'}{E \triangleright' F \xrightarrow{a} E' \triangleright' F}$$

we can note that E may in any moment neglect the external agent F behavior. The behavior of the system may simply follow the behavior of the controller process. This behavior is easy to implement. Indeed, before every target execution there is performed a check between the action is going to be performed by the target and the one of the controller operator. If they match then the action is allowed, otherwise it follows the behavior of E.

As an example of a controller operator whose behavior cannot be always implemented, we consider a controller operator defined as follows:

$$\frac{E \xrightarrow{a} E' \; F \xrightarrow{a} F'}{E \triangleright'' F \xrightarrow{a} E' \triangleright'' F'} \quad \frac{E \xrightarrow{a} E' \qquad F \not\xrightarrow{a} F'}{E \triangleright'' F \xrightarrow{a} E' \triangleright'' F}$$

The operator $\triangleright''$ cannot be implemented if we are not able to decide a priori which are possible next steps that the target agent is going to perform. If we do not have the code of F at our disposal (*i.e.*, this is a black box) we cannot simply do it. On the contrary, if it is possible to know a priori which is the set of possible next steps the target is going to perform, and a is not among these, then it would be possible to give priority to the first rule in order to allow always the correct action of the target. Thus, controller $\triangleright''$ would be our favorite, because it leaves the external agent to execute the correct action, if the first rule can be applied, and denies the unwanted situation checking them by the second rule. Unfortunately, also having at our disposal the code of F this is not always the case (remind CCS is Turing powerful).

3.3 Some examples

In this section we show two example of properties that can be enforced by using our framework.

3.3.1 The *Chinese Wall* policy

Now we present how we enforce the *Chinese Wall* policy, that is a very common security property. The Chinese Wall policy is expressed by the formula $\phi = Z \vee W$ where

$$\begin{array}{lcl} Z & =_\nu & [open_A]Z \wedge [open_B]\mathbf{F} \\ W & =_\nu & [open_B]W \wedge [open_A]\mathbf{F} \end{array}$$

Let us consider $S = X$. We synthesize a controller program Y for enforcing ϕ. For instance, we consider the process

$$\begin{array}{lcl} Y & = & Y_1 + Y_2 \\ Y_1 & = & open_A.Y_1 \\ Y_2 & = & open_B Y_2 \end{array}$$

It is possible to note that such Y, at the beginning, permits whatever possible behavior of the unspecified component.

Let us consider, for instance, $X = open_A.open_B.X$. In this case, by using the controller operator $\triangleright$, we have that , after the first $open_A$ action the execution is halt because the target X try to perform the action $open_B$ that is not allowed after the $open_A$ one. Indeed, the execution steps are the following:

$$Y \triangleright X \stackrel{open_A}{\longrightarrow} Y_1 \triangleright open_B.X$$

Hence, at the beginning, both the possibility, executing the action $open_A$ as well as executing the action $open_B$, are allowed. Since the first step is performed by X, the controller program chooses the component Y_1 and the action $open_B$ becomes forbidden from now on. However, after the transition, the target system tries to perform an action $open_A$ so the system halts.

3.3.2 Other simple example

Let us consider the process $S = a.b.\mathbf{0}$ and the following equational definition $\phi = Z$ where $Z =_\nu [\tau]Z \wedge [a]W$ and $W =_\nu [\tau]W \wedge [c]\mathbf{F}$. It asserts that, after every action a, an action c cannot be performed. Let $Act = \{a, b, c, \tau, \bar{a}, \bar{b}, \bar{c}\}$ be the set of actions. Applying the partial evaluation for the parallel operator we obtain, after some simplifications, the following system of equation, that we denoted with $\mathcal{D}$.

$$\begin{array}{l} Z_{//S} =_\nu [\tau]Z_{//S} \wedge [\bar{a}]Z_{//S'} \wedge [a]W_{//S} \wedge W_{//S'} \\ W_{//S'} =_\nu [\tau]W_{//S'} \wedge [\bar{b}]W_{//\mathbf{0}} \wedge [c]\mathbf{F} \\ Z_{//S'} =_\nu [\tau]Z_{//S'} \wedge [\bar{b}]Z_{//\mathbf{0}} \wedge [a]W_{//S'} \\ W_{//S} =_\nu [\tau]W_{//S} \wedge [\bar{a}]W_{//S'} \wedge [c]\mathbf{F} \\ Z_{//\mathbf{0}} = \mathbf{T} \\ W_{//\mathbf{0}} = \mathbf{T} \end{array}$$

where $S \stackrel{a}{\longrightarrow} S'$ so S' is $b.\mathbf{0}$.

The information obtained through partial model checking can be used to enforce a security policy.

We can note the process $Y = a.Y$ is a model of $\mathcal{D}$. Then, for any component X, we have $S \| (Y \triangleright X)$ satisfies ϕ. For instance, consider $X = a.c.\mathbf{0}$. Looking at the first rule of $\triangleright$, we have:

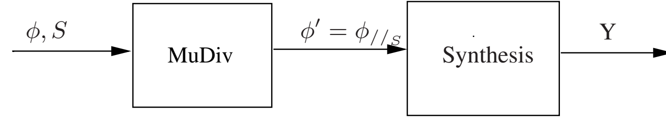
$$(S \| (Y \triangleright X)) = (a.b.\mathbf{0} \| (a.Y \triangleright a.c.\mathbf{0})) \stackrel{a}{\longrightarrow} (a.b.\mathbf{0} \| (Y \triangleright c.\mathbf{0}))$$

Since Y is going to perform a and the target X is going to perform the action c, the execution halts and so the system still preserves its security .

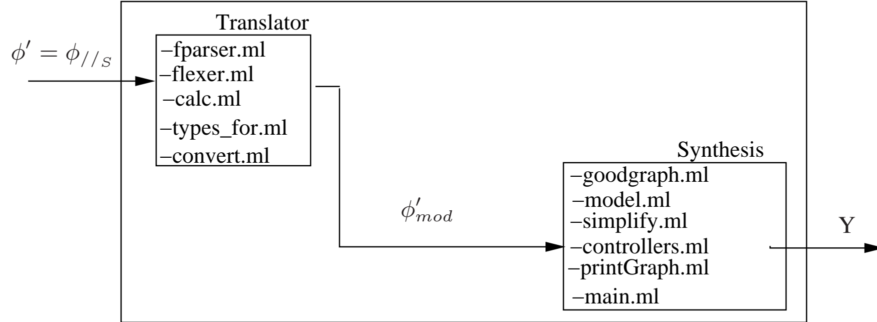
4 A Tool for the Synthesis

We have implemented a tool in order to automatically generate controller programs for enforcing safety properties, *i.e.*, properties that are expressed by equational μ-calculus formulas in which there is not diamond neither μ operators.

The tool consists of two main modules, going after the theoretical approach described before. The first one is the *MuDiv* tool developed by Nielsen and Andersen, that implements the partial model checking function for process algebra operators (see [6, 7]) and a second one is the *Synthesis* module implemented in O'Caml 3.09 (see [8]).



a) The architecture of the whole tool



b) A zoom of the Synthesis module

Figure 9: Architecture of the tool.

The *MuDiv* tool takes in input a system S and a formula of equational μ-calculus, ϕ, and calculates $\phi' = \phi_{//S}$ that is the partial evaluation of ϕ with respect to the system S.

The second part implements the satisfiability procedure developed by Walukiewicz in [43] for the modal μ-calculus formulas of our interest in this paper (*i.e.*, not presenting neither $\langle\rangle$ nor μ operators).

[4](see [44]). In particular it generates a model for ϕ'.

4.1 Synthesis tool

Now we describe more in detail the architecture of our implementation. As we have already said, it takes in input a system S and a formula ϕ and gives in output a process Y, described as a labelled graph, that is a model for ϕ', the formula obtained by the partial evaluation of ϕ by S. According to the theory developed in previous section a such Y guarantees $S\|(Y \triangleright X)$ satisfies ϕ whatever X is.

The tool is made up of two main parts (see Figure 9.a)): The first part implements the partial model checking function; the second one, by referring the satisfiability procedure, generates a process Y.

In Figure 9 there is a graphical representation of the architecture of the whole tool that we explain in more detail in the following section.

4.1.1 Architecture of the tool

As we have already said, the tool is made up of two main parts: The *MuDiv* module and the *Synthesis* module.

***MuDiv* tool** The first module of our tool consists of the *MuDiv* module. It is a tool for verifying concurrent systems. It is based on the technique of partial model checking described in [7]. The technique uses the equational μ-calculus to express the modal requirements and parallel composition of finite labelled transition systems to construct the model.

It has been developed in C++ by J.B. Nielsen and H.R. Andersen. The result is a non interactive batch program, where the input is provided as one or more input files, describing the model and the requirements. The output is the result of the model check and it is presented on the standard output or written to a file.

[4]We have implemented the Synthesis module for formulas of the modal μ-calculus because we have chosen to implemented the Walukiewicz procedure that is given for modal formulas.

Policy	Size	User time	System time
Only action a are allowed	7	0m0.005s	0m0.001s
It isn't allowed open a new file while another file is open	16	0m0.007s	0m0.004s
Chinese Wall	16	0m0.006s	0m0.001s
It isn't allowed performing three open action sequentially	25	0m0.021s	0m0.008s

Figure 10: Synthesis module experiments results.

***Synthesis* internal module.** The second module of our tool is the *Synthesis* one. It is able to build a model for a given modal μ-calculus formula by exploiting the satisfiability procedure. It is developed in O'caml 3.09 (see [8]) and it is described better in Figure 9.b) in which we can see that it consists of two submodules: the *Translator* and the *Synthesis*.

The Translator module. It manages the formula ϕ', output of the *MuDiv* module in order to obtain a formula that can be read from the Synthesis module. It "translates" ϕ' from an equational to a modal μ-calculus formula. This translation is necessary because the Walukiewicz's satisfiability procedure was developed for modal μ-calculus formulas instead the partial model checking was developed for equational μ-calculus ones. It is important to underline that we implemented the satisfiability procedure described by Walukievicz only for the μ-calculus formulas of our interest.

The Translator module consists in several functions: `fparser.ml` and `flexer.ml` 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 μ-calculus formula ϕ' according to the type that we have defined in `type_for.ml`. The last function, `convert.ml`, translates the equational μ-calculus formula ϕ' in the modal one ϕ'_{mod}.

The Synthesis submodule. It implements a satisfiability procedure for safety properties, described by Walukievicz in [43]. It is basically a tableaux construction. Each node in the graph representing the tableaux is characterized by the set of formulas that it satisfies. In `model.ml` we build the entire graph for the given formula ϕ'_{mod} in a recursive way by checking if the graph that we have generated is effectively a model or a refutation for ϕ'_{mod} by `goodgraph.ml`. Initially, the input is a node labelled by ϕ and `Empty_Graph`, that represents the empty graph. Then, in a recursive way, we build the graph as we have explained before.

It is important to note that the graph that we generate has some transitions that are labelled by an action and some transitions 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 ϕ'_{mod}.

In order to synthesize a process Y that is a model of ϕ'_{mod} according to the semantics of the considered controller operator, we have implemented the function `controllers. ml` that enforces the property according to the semantics of the controller.

Other minimal functions as `printGraph.ml` and `main.ml`, permit to print the graph and to create the executable file (`.exe`) respectively.

We are also able to translate the output into the $ConSpec$ policy language (see [45]). Hence our tool can be used to effectively enforce security policies on mobile phones by using the framework proposed in [46].

4.1.2 Performance

For our experiments we have used a PC with a CPU Intel core duo T2600 2.16GHz, 1GB RAM and an operative system linux Fedora Core 6 kernel 2.6.19.1.

We have observed the behavior of the Synthesis module, *i.e.*, we have analyzed the performance only of this module because it is the part of the tool that effectively generates the controller program.

We have tested several formulas with different size (i.e. the number of nodes in the graph). The results we have observed are summarized in Figure 10.

4.1.3 A case study

In order to explain better how our tool works we present an example in which a system must satisfy a safety property. We consider the controller operator that we have considered along all paper, *i.e.*,

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

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:

$$\begin{array}{l} Z_1 =_\nu [\tau]Z_1 \wedge [\texttt{open}]Z_2 \\ Z_2 =_\nu [\tau]Z_2 \wedge [\texttt{close}]Z_1 \wedge [\texttt{open}]F \end{array}$$

By using the $\triangleright$ operator, we halt the system if an user try to open a file while another is already open. In this case we generate a controller program Y for $Y \triangleright X$ and we obtain:

$$Y = \texttt{open.close}.Y$$

Y is a model for D.

In order to show how it works as controller program for $Y \triangleright X$ we suppose to have a possible user X that tries to open two different files. Hence $X = \texttt{open.open}.\mathbf{0}$. Applying $Y \triangleright X$ we obtain:

$$Y \triangleright X = \texttt{open.close}.Y \triangleright \texttt{open.open}.\mathbf{0} \xrightarrow{\texttt{open}} \texttt{close}.Y \triangleright \texttt{open}.\mathbf{0}$$

Since Y and X are going to perform a different action, *i.e.*, Y is going to perform `close` while X is going to perform `open`, the whole system halts.

5 Further Results

In this section we show how our technique can be used also for synthesizing controller programs able to enforce *composition of properties* and to deal with *parameterized systems*.

5.1 Synthesis of controller programs for composition of properties

We wonder if there exists a way easier than the method described before, to enforce a property described by a formula ϕ that can be written as conjunction of sub-formulas ϕ_i *simpler* than itself, *i.e.*, the size of each ϕ_i is minor of the size of ϕ.

Thus, we present a method to enforce this kind of properties by using the controller operator defined in Section 3.2. In particular we prove that the composition of controller programs for such operator enforce the conjunction of the properties they enforce. It is important to note that, since we consider the truncation operator, we are working under the additional assumption that the the properties that we investigate are safety properties, *i.e.*, we consider formulas in Fr_μ (see Section 3).

Hence we would like that the following relation holds:

$$\forall X \quad S \| X \models \phi \equiv \phi_1 \wedge \ldots \wedge \phi_n \tag{7}$$

where $\phi_1, \ldots, \phi_n$ are safety properties simpler than ϕ. In order to guarantee that the whole system satisfy ϕ we have to find a controller program Y that forces ϕ to be satisfied *i.e.*, :

$$\exists Y \quad \forall X \quad S \| Y \triangleright X \models \phi_1 \wedge \ldots \wedge \phi_n \tag{8}$$

According to Theorem 2.1, the cost of the satisfiability procedure is exponential in the size of the formula.

Here we present a method to find a controller program Y for ϕ starting from controller operators of its sub-formulas ϕ_i. As a matter of fact, let $\phi = \bigwedge_{i=1}^n \phi_i$ be the given formula, then by exploiting he Theorem 2.1, we synthesize a controller program Y_i for each of ϕ_i formula. Finally, by composing Y_i one to each other we obtain Y.

This method is less expensive than synthesize directly Y. Indeed, according to the Theorem 2.1, finding a model for a μ-calculus formula ϕ has a cost exponential in the size of ϕ, *i.e.*, let us consider that all the ϕ_i have the same size m, then the size of ϕ is $m \times n$. Hence synthesize directly Y costs $\mathcal{O}(c^{m\times n})$.

On the other hand, the cost of our method is $n\mathcal{O}(c^m)$ because the cost for synthesizing n models, one for each formula ϕ_i, is $n\mathcal{O}(c^m)$ and the cost of the composition through the $\triangleright$ operator is constant in the size of the formula.

In order to describe our method, first of all, we rewrite Formula (7), by exploiting the semantics definition of the logical conjunction, as follows:

$$\begin{array}{l} \forall X \quad S\|X \models \phi_1 \text{ and} \\ \forall X \quad S\|X \models \phi_2 \text{ and} \\ \dots \\ \forall X \quad S\|X \models \phi_n \end{array}$$

By partial model checking we obtain:

$$\begin{array}{l} \forall X \quad X \models \phi'_1 \text{ and} \\ \forall X \quad X \models \phi'_2 \text{ and} \\ \dots \\ \forall X \quad X \models \phi'_n \end{array}$$

where for each i from 1 to n, $\phi'_i = (\phi_i)_{//S}$.

Let $Y_1, \dots, Y_n$ be n processes such that:

$$\begin{array}{l} \forall X \quad Y_1 \triangleright X \models \phi'_1 \text{ and} \\ \forall X \quad Y_2 \triangleright X \models \phi'_2 \text{ and} \\ \dots \\ \forall X \quad Y_n \triangleright X \models \phi'_n \end{array}$$

It is possible to prove the following result.

Lemma 5.1 *Let ϕ be a safety property, conjunction of n safety properties,* i.e., $\phi = \phi_1 \wedge \phi_2 \wedge \dots \wedge \phi_n$ *where $\phi_1, \dots \phi_n$ are safety properties. Let $Y_1, \dots, Y_n$ be n controller programs such that $\forall i$ such that $1 \leq i \leq n$ $Y_i \models \phi_i$. We have*

$$\forall X \quad Y_n \triangleright (Y_{n-1} \triangleright (\dots \triangleright (Y_2 \triangleright (Y_1 \triangleright X)))) \models \phi$$

This means that, once we have synthesized controller programs for enforcing several safety properties, we are able to enforce also the conjunction of them simply applying them successively.

However, in this way, we apply the procedure for enforcing n times. Instead we want apply it only one time to force the conjunction of formulas. For that reason we prove the following proposition.

Proposition 5.1 *Let us consider the controller operator $\triangleright$ defined in Section 3.2. It is possible to find $Y_1, \dots, Y_n$ controller programs such that. if $Y_1 \triangleright X \models \phi'_1, \dots, Y_n \triangleright X \models \phi_n$ then $(Y_1 \triangleright \dots \triangleright Y_n) \triangleright X \models \phi_1 \wedge \dots \wedge \phi_n$.*

Hence, referring to the Formula (8), in order to find Y we find $Y_1, \dots, Y_n$ that enforce $\phi'_1, \dots, \phi'_n$ respectively and we compose them as in Proposition 5.1. In this way we find Y that force $\phi' = \phi'_1 \wedge \dots \wedge \phi'_n$. According to Lemma 2.3 we have:

$$\forall X \quad Y \triangleright X \models \phi' \Leftrightarrow \forall X \quad S\|Y \triangleright X \models \phi$$

Hence we obtain a controller program Y for ϕ.

5.2 Synthesis of controller programs for parameterized systems

A parameterized system describes an infinite family of (typically finite-state) systems (see [47]). Instances of the family can be obtained by fixing parameters.

Let us consider a parameterized system $S = P_n$ defined by parallel composition of processes P, *e.g.*,

$$\underbrace{P\|P\|\dots\|P}_{n}$$

The parameter n represents the number of processes P present in the system S.

Example 5.1 *Consider a system with one consumer process C and several producer processes P. Each process P is defined $P \stackrel{def}{=} a.P$ where $a \in Act$, and the process C is $\bar{a}.C$. Let us suppose that the system consists of n producer and one consumer, then the entire system is $(P_n \| C)\backslash\{a\}$ and the processes communicate by synchronization on $\bar{a}$ and a actions.*

Referring to the Formula (2) we may wish to have:

$$\forall n \quad \forall X \quad P_n \| X \models \phi \tag{9}$$

It is possible to note that in the previous equation there are two universal quantifications: The first one is on the number of instances of the process P, n, and the second one is on the possible behavior of the unknown agent.

In order to eliminate the universal quantification on the number of processes, firstly we define the concept of *invariant formula with respect to partial model checking for parallel operator* as follows.

Definition 5.1 *A formula ϕ is said an* invariant with respect to partial model checking *for the system $P \| X$ if and only if $\phi \Leftrightarrow \phi_{//P}$.*

It is possible to prove the following result.

Proposition 5.2 *Given the system $P_k \| X$. If ϕ is an invariant formula for the system $P \| X$ then*

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

In order to apply our theory, we show a method to find the invariant formula. According to [47], let ψ_i be defined as follows:

$$\psi_i = \begin{cases} \phi_1 & \text{if } i = 1 \\ \psi_{i-1} \wedge \phi_i & \text{if } i > 1 \end{cases}$$

where for each i, $\phi_i = \phi_{//P_i}$.

By definition of ψ_i and by Lemma 2.3, $\forall j$ such that $1 \leq j \leq i$ $(X \models \phi'_j) \Leftrightarrow X \models \psi_i$. Hence $X \models \psi_i$ means that $\forall j$ such that $1 \leq j \leq i$ $P_j \| X \models \phi$. We say that ψ_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 ψ_ω for a chain of μ-calculus formulas, that is also said *limit of the sequence*, then the following identity holds:

$$\forall X \quad (X \models \psi_\omega \Leftrightarrow \forall n \geq 1 \quad P_n \| X \models \phi) \tag{10}$$

Now we have a problem equivalent to the problem expressed in Statement 2. Then we apply the theory developed in previous section to synthesize a controller program for a controller operator.

In some cases it could not be possible to find the limit of the chain. However there are some techniques that can be useful in order to find an approximation of this limit (see [47, 48]).

6 Related work

In this section we present some of the related work on controller theory and security.

In [9] security automata for enforcing security properties were introduced by Schneider. These automata pick each action produced by the target system and check if this action is allowed in a given state. A security property that can be enforced in this way corresponds to a *safety property* (according to [9], 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 [10, 11] have defined four different kinds of security automata which deal with finite sequences of actions: *truncation automaton, suppression automaton, insertion automaton* and *edit automaton*. The truncation automata basically correspond to Schneider automata. The other kinds of automata allow also a sort of modification of the output of the target system to make it correct.

Our approach based on process algebra permits us to automatically synthesize a controller program for a chosen controller operator. We can definite process algebra operators that precisely corresponds to the Ligatti's *et al.* automata (*e.g.*, see [51]). Hence, it is possible to synthesize edit automata for enforcing a specific security policy expressed in temporal logic. The resulting automata are finite state. This is an advantage of our approach w.r.t. [9] and [10, 11] since the synthesis problem is not addressed there.

Also Bartoletti, Degano and Ferrari in [12] refer to [9] by saying that while safety properties can be enforced by an execution monitor (that does not alter program execution), *liveness properties* cannot (a liveness property said that "something good happens"). 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. In [13] they have proposed a mixed approach to access control, that efficiently combines static analysis and run-time checking. They compile a program with policy framings into an equivalent one without framings, but instrumented with local checks. The static analysis determines which checks are needed and where they must be inserted to obtain a program respecting the given security requirements. The execution monitor is essentially a finite-state automaton associated with the relevant security policies. Their work is not focussed on synthesis as ours. In our work we isolate the possible un-trusted components by partial model checking then we checks at run-time the target.

Much of prior work are about the study of enforceable properties and related mechanisms. In [14] authors deal with a safety interface that permits to study if a module is safe or not in a given environment. Here all system is checked, instead in our approach, through the partial model checking function, we are able to monitor only the necessary/untrusted part of the system.

In [15] the authors provided a preliminary work in which there are presented several techniques for automatically synthesizing systems enjoying a very strong security property, *i.e.*, $SBSNNI$ (see [16]). This is also called P_BNDC in [17]. In both these work the authors did not deal with control theory.

The synthesis of controllers is a framework addressed also in other research areas (*e.g.*, see [18, 19, 20, 21]). Our work on controller mechanisms starts from the necessity to make systems secure regardless the behavior of possible intruders. Indeed, in our work we do not generate a controller for a specified problem, on the contrary, we synthesize a controller that is able to make the system secure against every possible malicious behavior of an unspecified components that interacts with the considered system.

Many other approaches to the controller synthesis problem are based on game theory (see [22, 23, 24, 25]). As a matter of fact, different kinds of automata are used to model properties that must be enforced. Games are defined on the automata in order to find the structure able to satisfy the given properties. For instance in [22], the authors deal with the synthesis of controllers for discrete event systems by finding a winning strategies for a parity games. In this framework it is possible to extend the specification of the supervised systems as well as constraints on the controllers by expressing them in the modal μ-calculus. In order to express un-observability constraints, they propose an extension of the modal mu-calculus in which one can specify whether an edge of a graph is a loop. This extended μ-calculus still has the interesting properties of the classical one. The method proposed in this paper to solve a control problem consists in transforming this problem into a problem of satisfiability of a μ-calculus formula so that the set of models of this formula is exactly the set of controllers that solve the problem. On the contrary, we synthesize controllers that work by monitoring only the possible un-trusted component of the system. Moreover they do not addressed any security analysis, *i.e.*, they synthesize controllers for a given process that must be controlled. On the contrary we synthesize controllers that make the system secure for whatever behavior of unknown components. Our controllers are synthesized without any information about the process they are going to control.

In [26, 27, 28] the authors developed a theory for the synthesis of the maximally permissive controller. The authors have proposed general synthesis procedure which always computes a maximal permissive controller when it exists. However they generate a maximal permissive controller knowing the behavior of the process they are going to control. On the contrary, we do not know anything a priori on the possible behavior of a possible malicious agent whose behavior we want to control.

7 Conclusion and Future Work

In this paper we have illustrated some results towards a uniform theory for enforcing security properties. In particular, we have extended a framework based on process calculi and logical techniques, that have been shown to be suitable to *model* and *verify* several security properties, to tackle also *synthesis* problems of secure systems.

Indeed, we solve the problem of finding a possible implementation of a controller program, that, by monitoring the target, replaces the unknown component, in such a way that the whole system is secure. Hence, according to the security property we are considering, we reduce our original problem to a satisfiability problem. In this way, by applying a satisfiability procedure, we obtain a controller program Y that is a model for the formula we want to enforce and a controller program for the controller operator provided it holds a mild assumption. However, the satisfiability problem for μ-calculus formulas is decidable in exponential time in the dimension of the formula.

We present a tool for the synthesis of a controller program. The tool merges our implementation of a satisfiability procedure based on the Walukiewicz's algorithm and the partial model checking technique. In particular, starting from a system and

a formula describing a security property, the tool generates a process that, by monitoring a possible un-trusted component, guarantees that the system with an unspecified component satisfies the required formula whatever the target is.

An extended version of this work is [49], in which we have also given several semantics definition for controller operators, starting from the work of Schneider [9]. As a matter of fact, recently the interest on developing techniques to study how to make a system secure by enforcing *security policy* has been growing (*e.g.*, see [10, 11, 9]). Schneider in [9] considered an enforcement mechanism as a program which controls 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 Schneider's definition, Ligatti *et al.* described four different ways to enforce safety policies (see [10, 11]). 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 [49], we model the security automata of [10, 11] through process algebra by defining *controller operators* $Y \triangleright_{\mathbf{K}} X$, with $\mathbf{K} \in \{T, S, I, E\}$ where T, S, I and E represent Truncation, Suppression, Insertion and Edit automaton, respectively, Y is the controller program and X the target system. We give the semantics definition of each of controller operator and prove that they have the same behavior of the respective security automaton. Moreover, it is possible to apply the theory developed in this work to such controller operators in order to synthesis controller programs able to enforce safety properties in these four different ways.

References

[1] P. Merlin and G. V. Bochmann. On the construction of submodule specification and communication protocols. *ACM Transactions on Programming Languages and Systems*, 5:1–25, 1983.

[2] F. Martinelli. *Formal Methods for the Analysis of Open Systems with Applications to Security Properties*. PhD thesis, University of Siena, December 1998.

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

[4] F. Martinelli. Partial model checking and theorem proving for ensuring security properties. In *CSFW '98: Proceedings of the 11th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 1998.

[5] R. Milner. Operational and algebraic semantics of concurrent processes. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 19, pages 1201–1242. The MIT Press, New York, N.Y., 1990.

[6] H. R. Andersen. *Verification of Temporal Properties of Concurrent Systems*. PhD thesis, Department of Computer Science, Aarhus University, Denmark, 1993.

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

[8] X. Leroy, D. R. Damien Doligez, Jacques Garrigue, and J. Vouillon. The Objective Caml system release 3.09, 2004.

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

[10] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In I. Cervesato, editor, *Foundations of Computer Security: proceedings of the FLoC'02 workshop on Foundations of Computer Security*, pages 95–104, Copenhagen, Denmark, 25–26 July 2002. DIKU Technical Report.

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

[12] M. Bartoletti, P. Degano, and G. L. Ferrari. Enforcing secure service composition. In *CSFW*, pages 211–223. IEEE Computer Society, 2005.

[13] M. Bartoletti, P. Degano, and G. L. Ferrari. Checking risky events is enough for local policies. In M. Coppo, E. Lodi, and G. M. Pinna, editors, *ICTCS*, volume 3701 of *Lecture Notes in Computer Science*, pages 97–112. Springer, 2005.

[14] J. Elmqvist, S. Nadjm-Tehrani, and M. Minea. Safety interfaces for component-based systems. In R. Winther, B. A. Gran, and G. Dahll, editors, *SAFECOMP*, volume 3688 of *Lecture Notes in Computer Science*, pages 246–260. Springer, 2005.

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

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

[17] R. Focardi and S. Rossi. Information flow security in dynamic contexts, 2002.

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

[19] O. Kupferman and M. Vardi. μ-calculus synthesis. In *Proc. 25th International Symposium on Mathematical Foundations of Computer Science*, volume 1893 of *Lecture Notes in Computer Science*, pages 497–507. Springer-Verlag, 2000.

[20] H. Saidi. Towards automatic synthesis of security protocols. March 2002.

[21] H. Wong-Toi and D. L. Dill. Synthesizing processes and schedulers from temporal specifications. In E. M. Clarke and R. P. Kurshan, editors, *CAV*, volume 531 of *Lecture Notes in Computer Science*, pages 272–281. Springer, 1990.

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

[23] O. Kupferman, P. Madhusudan, P. S. Thiagarajan, and M. Y. Vardi. Open systems in reactive environments: Control and synthesis. *Lecture Notes in Computer Science*, 1877:92+, 2000.

[24] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems (extended abstract).

[25] F. Martinelli. Module checking through patrial model checking. Technical Report IIT-TR06/2002, IIT-CNR, 2002.

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

[27] S. Riedweg and S. Pinchinat. Maximally permissive controllers in all contexts. In *Workshop on Discrete Event Systems*, Reims, France, September 2004.

[28] S. Riedweg and S. Pinchinat. You can always compute maximally permissive controllers under partial observation when they exist. In *Proc. 2005 American Control Conference.*, Portland, Oregon, June 2005.

[29] G. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI-FN-19, Aarhus University, 1981.

[30] M. Hennessy. *Algebraic Theory of Processes*. The MIT Press, Cambridge, Mass., 1988.

[31] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

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

[33] E. Asarin and C. Dima. Balanced timed regular expressions. *Electr. Notes Theor. Comput. Sci.*, 68(5), 2002.

[34] R. Gorrieri, R. Lanotte, A. Maggiolo-Schettini, F. Martinelli, S. Tini, and E. Tronci. Automated analysis of timed security: a case study on web privacy. *Int. J. Inf. Sec.*, 2(3-4):168–186, 2004.

[35] R. Focardi, R. Gorrieri, and F. Martinelli. Real-time Information Flow Analysis. *IEEE JSAC*, 2003.

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

[37] M. Hennessy and T. Regan. A temporal process algebra. In *FORTE '90: Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*, pages 33–48. North-Holland, 1991.

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

[39] D. Park. Concurrency and automata on infinite sequences. In *Proceedings 5th GI Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183, 1981.

[40] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, 1990.

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

[42] G. Bruns and I. Sutherland. Model checking and fault tolerance. In *AMAST '97: Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology*, pages 45–59, London, UK, 1997. Springer-Verlag.

[43] I. Walukiewicz. *A Complete Deductive System for the μ-Calculus*. PhD thesis, Institute of Informatics, Warsaw University, June 1993.

[44] D. Kozen. Results on the propositional $\mu-$calculus. *Theoretical Computer Science*, 27(3):333–354, 1983.

[45] I. Aktung, K. Naliuka: Conspec – A formal language for policy specification. Electr. Notes Theor. Comput. Sci. **197**(1) (2008) 45–58

[46] http://www.s3ms.org/index.jsp[Last visited: 14 July 2008].

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

[48] P. Cousot and R. Cousot. 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*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[49] Matteucci, I.: Synthesis of Secure Systems. PhD thesis, University of Siena (April 2008)

[50] J. Bradfield and C. Stirling. *Modal logics and mu-calculi: an introduction*. Handbook of Process Algebra. Elsevier, 2001.

[51] F. Martinelli and I. Matteucci. Through modeling to synthesis of security automata. *Electr. Notes Theor. Comput. Sci.*, 179:31–46, 2007.

[52] I. Matteucci. A tool for the synthesis of controller programs. In T. Dimitrakos, F. Martinelli, P. Y. A. Ryan, and S. A. Schneider, editors, *Formal Aspects in Security and Trust*, volume 4691 of *Lecture Notes in Computer Science*, pages 112–126. Springer, 2006.

[53] I. Matteucci. Automated synthesis of enforcing mechanisms for security properties in a timed setting. *Electr. Notes Theor. Comput. Sci.*, 186:101–120, 2007.

Appendix: Technical Proof

Proof of Proposition 3.2: We prove that the following relation is a weak simulation:

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

Assume that $E \triangleright F \xrightarrow{a} E' \triangleright F'$. For the semantics definition of $\triangleright$, we have that $F \xrightarrow{a} F'$ and $E \xrightarrow{a} E'$. Hence, there exists E' s.t. $E \stackrel{a}{\Longrightarrow} E'$ and $(E' \triangleright F', E') \in \mathcal{S}$.

It is not difficult to note that, following a similar reasoning it is also possible to prove that $Y \triangleright X \preceq X$.

$\square$

Proof of Lemma 5.1: For induction on the number of the formulas in the conjunction n:

$n = 1$**:** In this case $\phi = \phi_1$. Hence $Y = Y_1$ that is the controller program such that $Y \triangleright X \models \phi$.

$n \Rightarrow n+1$**:** Let ϕ be a formula such that $\phi = \phi_1 \wedge \ldots \wedge \phi_{n+1}$ and Y_{n+1} be a controller program such that for all possible X, $Y_{n+1} \triangleright X \models \phi_{n+1}$. For inductive hypothesis we know that for all possible X, $Y_n \triangleright (Y_{n-1} \triangleright (\ldots \triangleright (Y_2 \triangleright (Y_1 \triangleright X)))) \models \phi_1 \wedge \ldots \wedge \phi_n$. We have to prove that

$$\forall X \quad Y_{n+1} \triangleright (Y_n \triangleright (Y_{n-1} \triangleright (\ldots \triangleright (Y_2 \triangleright (Y_1 \triangleright X))))) \models \phi$$

For sake of simplicity, we denote by Y^n the process $Y_n \triangleright (Y_{n-1} \triangleright (\ldots \triangleright (Y_2 \triangleright (Y_1 \triangleright X))))$. We know that for all possible X, $Y_{n+1} \triangleright X \models \phi_{n+1}$, so $Y_{n+1} \triangleright Y^n \models \phi_{n+1}$. For Proposition 3.1 and Lemma 3.2, $Y_{n+1} \triangleright Y^n \models \phi_1 \wedge \ldots \wedge \phi_n$. Hence, for the definition of conjunction $Y_{n+1} \triangleright Y^n \models \phi$.

□

In order to prove the Proposition 5.1 we prove the following lemma from which the proof of the proposition follows directly.

Lemma 7.1 *Let ϕ, $Y_1, \ldots, Y_n$ be as in Lemma 5.1. We have that $\forall X$*

$$Y_n \triangleright (Y_{n-1} \triangleright (\ldots \triangleright (Y_2 \triangleright (Y_1 \triangleright X)))) \models \phi \Rightarrow (Y_n \triangleright \ldots \triangleright Y_1) \triangleright X \models \phi$$

holds.

Proof: For induction on the number of controller programs n:

$n = 1$**:** Trivial.

$n \Rightarrow n+1$**:** For hypothesis we have that

1. $\forall 1 \leq i \leq n+1, \forall X \; Y_i \triangleright X \models \phi_i$;
2. $\forall X \; Y_n \triangleright (Y_{n-1} \triangleright (\ldots \triangleright (Y_2 \triangleright (Y_1 \triangleright X)))) \models \phi \Rightarrow \forall X \; (Y_n \triangleright \ldots \triangleright Y_1) \triangleright X \models \phi$

We want to prove that

$$\forall X \; Y_{n+1} \triangleright (Y_n \triangleright (\ldots \triangleright (Y_2 \triangleright (Y_1 \triangleright X)))) \models \phi \Rightarrow \forall X \; (Y_{n+1} \triangleright \ldots \triangleright Y_1) \triangleright X \models \phi$$

For sake of simplicity we denote by $Y^n_\triangleright$ the process $(Y_n \triangleright \ldots \triangleright Y_1)$. For hypothesis 1 we can consider Y^n as X so, $Y_{n+1} \triangleright Y^n_\triangleright \models \phi_{n+1}$. For Lemma 5.1 and hypothesis 2 $Y^n_\triangleright \triangleright Y_{n+1} \models \phi_1 \wedge \ldots \wedge \phi_n$. Since $Y^n_\triangleright \triangleright Y_{n+1}$ and $Y_{n+1} \triangleright Y^n_\triangleright$ are bisimilar so they satisfy the same formulas (see [50]). In particular $Y_{n+1} \triangleright Y^n_\triangleright \models \phi_1 \wedge \ldots \wedge \phi_n$. Hence $Y_{n+1} \triangleright Y^n_\triangleright \models \phi$. For Proposition 3.2, we conclude that

$$\forall X \; (Y_{n+1} \triangleright \ldots \triangleright Y_1) \triangleright X \models \phi$$

□

Proof of Proposition 5.2: The proof comes directly from the Lemma 2.3. As a matter of fact, according to Lemma 2.3 and to Definition 5.1:

$$P_n \| X \models \phi \text{ if and only if } P_{n-1} \| X \models \phi_{//P} \equiv \phi$$

Reiterating this procedure n times we obtain:

$$\forall X \quad (\forall n \quad P_n \| X \models \phi \quad \text{if and only if} \quad X \models \phi)$$

□