A framework for automatic generation of security controller

Fabio Martinelli, Ilaria Matteucci

SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2008; **00**:1–30 (DOI: 10.1002/000)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/000

A framework for automatic generation of security controller‡

Fabio Martinelli[1], Ilaria Matteucci[1]

[1] *Istituto di Informatica e Telematica - C.N.R., Pisa, Italy*

SUMMARY

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 interacting components.

A complex system under analysis is described as an open system, *i.e.*, a system in which an unspecified component (a component whose behaviour is not fixed in advance) interacts with the known part of the system. Within this formal approach, we propose techniques that aim to synthesize controller programs able to guarantee that, for all possible behaviours of the unspecified component, the system should work properly, *e.g.*, it should be able to satisfy a certain property.

For performing this task, we first need to identify the set of necessary and sufficient conditions that the unspecified component has to satisfy in order to ensure that the whole system is secure. Hence, by exploiting satisfiability procedures for temporal logic, we automatically synthesize an appropriate controller program that forces the unspecified component to meet these conditions. This will ensure the security of the whole system.

In particular, 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 also address the synthesis problem.

Moreover, we describe a tool for the synthesis of secure systems which is able to generate appropriate controller programs. This tool is also able to translate the synthesized controller programs into the ConSpec language. ConSpec programs can be actually deployed for enforcing security policies on mobile Java applications by using the run-time framework developed in the ambit of the European Project S3MS.

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

*Correspondence to: Istituto di Informatica e Telematica - C.N.R., Pisa, Italy
†E-mail: {Fabio.Martinelli, Ilaria.Matteucci}@iit.cnr.it
‡This work is an expanded and revised version of [36, 37, 38].
Contract/grant sponsor: Work partially supported by EU-funded project “Software Engineering for Service-Oriented Overlay Computers”(SENSORIA).; contract/grant number: FP6-016004
Contract/grant sponsor: Work partially supported by EU-funded project “Secure Software and Services for Mobile Systems”(S3MS).; contract/grant number: FP6-027004
Contract/grant sponsor: Work partially supported by EU-funded project “Emergent Connectors for Eternal Software Intensive Networked Systems” (Connect); contract/grant number: FP7-231167

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 basic reasons which stimulates the research towards *secure systems*, *i.e.*, systems that satisfy some *security properties* describing their acceptable executions.

In particular, this paper concerns the automatic generation of mechanisms for making systems secure, *i.e.*, mechanisms able to enforce security properties.

We start from the approach proposed in [31, 32, 35] for the verification of security properties. This approach is based on the notion of *open system*, *i.e.*, a system $S(X)$ with an unspecified component X, that plays the role of a potential malicious user of the system S itself. In our framework we use a very specific format[†] of open system: $S\|X$, where S and X are processes and $\|$ is the parallel composition operator of the CCS process algebra [41]. Given a security property ϕ, expressed through a logical formula (*e.g.*, [2, 3]), the main verification problem corresponds to check that $S\|X$, denoted by $\models$, satisfies the formula ϕ, regardless the behaviour of any possible malicious user X that may interact with S. Formally, the verification problem for an open system is:

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

What happens if (1) is not fulfilled, *i.e.*, if there exists X' s.t. $S\|X' \not\models \phi$?

The theory developed in this paper addresses this problem. In particular, we envisage a method aiming to constrain the behaviour of X in order to prevent it from attacking the system S. More formally, we wonder if there exists a process Y that, by controlling the behaviour 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 behaviour of X. Hereafter, we refer to this problem as the *synthesis of secure system* problem.

We show an approach for solving this problem by automatically generating an implementation Y, able to guarantee that the open system results secure. Through our framework, we are able to enforce a lot of significant security properties according to the semantics definition of the considered controller operator $\triangleright$.

Intuitively, those properties state that "nothing bad can happen". When a safety property is violated on an execution trace, then there exists a finite prefix of the trace that violates the property. This same notion of safety properties is used in security when denoting a large set of properties as confidentiality (*e.g.*, a message is never disclosed) and access control ones. To give an example of access control policy we mention the *Chinese Wall policy*: Assuming that there are two disjoint sets of resources,

[†]It is important to note that in this paper we study a particular format of open systems, however this is very suitable to model many existing real-life scenarios and the theory could be easily extended to more complex formats.

DOI: 10.1002/stvr

once one accesses a resource of one set, it is not possible to access the other set of resources any more and vice-versa.

Our approach works as follows: We first identify which are the necessary and sufficient conditions that a possible malicious component X should satisfy for altering the correct behaviour of the system S by applying the *partial model checking* function [2, 3]. This way, we evaluate the formula ϕ w.r.t. the behaviour of S and obtain a new formula $\phi' = \phi_{//_S}$ that deals only with the un-trusted part of the system, here X. Indeed, according to the partial model checking theory, the following holds:

$$S \| X \models \phi \text{ iff } X \models \phi_{//_S}$$

In order to force X to behave correctly, *i.e.*, as prescribed by ϕ', we appropriately use controller operators, denoted $Y \triangleright X$. In order to synthesize the correct process Y, we use satisfiability procedures for temporal logic.

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 (*e.g.*, [16]) 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 that constrains only un-trusted ones, *e.g.*, let us consider a smart-phone able to execute a program downloaded from the Net, for instance a java-applet. Since the developer of the applet is unknown, we do not know a priori if the program that we are going to execute is secure or not. Hence, we would like to have a method that allows us to check at run-time if the applet runs without violating the security of our device[‡].

Similarly, it would not be possible to build a monitor for a whole distributed architecture, though it could be possible to build one for some of its components.

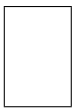
Another advantage of our approach is that it allows to automatically synthesize a controller program Y for a controller operator $Y \triangleright X$, by exploiting satisfiability procedures for temporal logic.

To summarize, in this work:

- we show a method to synthesize a controller program Y for a controller operator $\triangleright$ (provided it fulfils a mild assumption);
- we propose a general framework to enforce security properties that is able to deal with several problems. In particular, we present here a possible extension of our framework for treating parametrized systems;
- we have developed a synthesis tool in the objective language O'caml [29] that, given as an input both a system and a security property, it automatically generates a controller program. In addition, we have developed a translator from our internal representation to *ConSpec* [1], the security policy language used in the ambit of the S3MS European project [46]. There exists a run-time framework for enforcing security policies (controller programs in our context) for execution monitoring of Java applications on mobile devices. Hence, as final result, we are able to generate a controller program directly executable on a mobile phone.

[‡]This is part of the work done S3MS EU-Project. Indeed in the ambit of this project, a run-time monitoring framework has been developed in order to guarantee the security of Java applications running on mobile devices.

Prepared using stvrauth.cls
DOI: 10.1002/stvr



This paper is organized as follows: Section 2 recalls some useful background notions. Section 3 proposes our approach for the synthesis of controller program able to enforce such security properties. It also shows how the proposed framework could be exploited for dealing with parameterized systems. Section 4 describes the tool we have developed in order to automatically generate controller programs given the system, the properties and the kind of controller operator we are going to use. Section 5 compares our work with other work already appeared in literature and Section 6 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* [2, 3], a modal logic suitable for expressing, in a qualitative way, relations among events. Successively, we describe the CCS process algebra [41]. Finally, we show the compositional analysis techniques proposed by Andersen [2, 3] that deals with partially specified systems by introducing the *partial model checking* function.

2.1. Equational μ-calculus

Equational μ-calculus is based on *fix-point equations* that allow to define recursive properties of a system.

Let Act be the set of actions ranged over by $a, b, \ldots$, let 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 constants, respectively, *true* and *false*; $\wedge$ 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 into a state that satisfies ϕ. The necessity modality $[a]\phi$ expresses that after each a transition there is a state that satisfies ϕ. The minimal (maximal) fix-point equation syntax, $Z =_\mu \phi$ ($Z =_\nu \phi$), is used to define recursive formulas. It is assumed that variables appear only once on the left-hand sides of the equations of the list, the set of these variables being 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) $\langle S, Act, \rightarrow \rangle$ that consist of a set of states S, a set of labels (or *actions*)Act and a *transition relation*, $\rightarrow \subseteq S \times Act \times S$, that describes how a process passes from a state to another, *i.e.*, given two states s an s' and a label a, $s \xrightarrow{a} s'$ means that the process passes from the state s to the state s' through an arc labelled by a. Hence, the semantics of an assertion ϕ corresponds to a subset of the states in S, denoted by $[\![\phi]\!]_\rho$ where ρ is a function, called *environment* from free variables in ϕ to subsets of states in S (Figure 1). Moreover, the semantics $[\![D]\!]_\rho$ of a definition list is an environment which assigns subsets of S to variables in $Def(D)$. Let σ be in $\{\mu, \nu\}$, the semantics of the equation lists is defined by the following equations:

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

DOI: 10.1002/stvr

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

Figure 1. Denotational semantics of assertions in equational μ-calculus.

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

Informally, the semantics definition of the fix-point 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 $D \downarrow Z$ 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. Besides Z must be the first variable in the list D.

For this calculus we have the following satisfiability result.

Theorem 2.1 ([49]) *Given a formula ϕ, it is possible to decide, within exponential time in the length of ϕ, if there exists a model of ϕ. It is also possible to provide an example of such model.*

2.2. Process algebra: CCS

The *Calculus of Communicating Systems* (CCS) is a process algebra developed by Robin Milner [41] used for describing concurrent and communication processes.

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. Tracing 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 constant process equipped with its formal definition $A = P$ and eventually the relabelling function $f : Act \mapsto Act$ must be such that $f(\tau) = \tau$.

The operational semantics of CCS operators is described by a $(\mathcal{E}, Act, \rightarrow)$, 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. Indeed $\rightarrow$ defines the usual concept of derivation in one step. As a matter of fact $P \xrightarrow{a} P'$ means that process P

DOI: 10.1002/stvr

$$\text{Prefixing: } \frac{}{a.P \xrightarrow{a} P}$$

$$\text{Choice: } \frac{P \xrightarrow{a} P'}{P+Q \xrightarrow{a} P'} \quad \frac{Q \xrightarrow{a} Q'}{P+Q \xrightarrow{a} Q'}$$

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

$$\text{Restriction: } \frac{P \xrightarrow{a} P'}{P\backslash L \xrightarrow{a} P'\backslash L}$$

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

$$\text{Constant: } \frac{P \xrightarrow{a} P'}{A \xrightarrow{a} P'}$$

Figure 2. Operational semantics for CCS operators.

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 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 interfere with the capabilities of the other process. The third rule of parallel composition is characteristic of this calculus. It expresses that the communication between two processes takes place 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 between parallel processes on an action, we have to set the restriction operator in conjunction with the 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 those actions that syntactically appear in process P.

DOI: 10.1002/stvr

Figure 3. Example of two not observationally equivalent processes.

2.2.1. Behavioural relations

In general, it is interesting to compare processes by abstracting from irrelevant aspects. Here, we recall the notion of *weak-simulation* behavioural relation [41, 42] that permits to compare the behaviour of the two considered processes. Within a step-wise development strategy, this could be appealing as we would be able to substitute simple specifications with more complex ones, without having to affect the overall visible behaviour of the system. For example, we can imagine substituting 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 behaviour of a system.

Look at the following example.

Example 2.1. *The processes P and Q in Figure 3 cannot be considered equivalent, since the second performs an internal action by reaching a state where an action a is no longer possible. Thus, the non visible behaviour of the system, represented by the τ action, can modify its visible behaviour.*

■

To compare this kind of processes we recall the following definition.

Definition 2.1. *Let $(\mathcal{E}, Act, \rightarrow)$ be an LTS of concurrent processes over the set of actions Act, and let $\mathcal{R}$ be a binary relation over $\mathcal{E}$. Then $\mathcal{R}$ is called* weak simulation *denoted by $\preceq$, over $(\mathcal{E}, Act, \rightarrow)$ if and only if, whenever $(P, Q) \in \mathcal{R}$ we have:*

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

2.3. Compositional analysis: the *partial model checking*

The theory on compositionality was developed in [3]. 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?

DOI: 10.1002/stvr

$(D\downarrow Z)/\!/t = (D/\!/t)\downarrow Z_t$
$\epsilon/\!/t = \epsilon$
$(Z =_\sigma \phi D)/\!/t = ((Z_s =_\sigma \phi//s)_{s\in S})(D)/\!/t$
$Z/\!/t = Z_t$
$\phi_1 \wedge \phi_2/\!/s = (\phi_1/\!/s)\wedge(\phi_2/\!/s)$
$\phi_1 \vee \phi_2/\!/s = (\phi_1/\!/s)\vee(\phi_2/\!/s)$
$[a]\phi/\!/s = [a](\phi/\!/s)\wedge \bigwedge_{s\xrightarrow{a}s'} \phi/\!/s'$, if $a \neq \tau$
$\langle a\rangle\phi/\!/s = \langle a\rangle(\phi/\!/s)\vee \bigvee_{s\xrightarrow{a}s'} \phi/\!/s'$, if $a \neq \tau$
$[\tau]\phi/\!/s = [\tau](\phi/\!/s) \wedge \bigwedge_{s\xrightarrow{\tau}s'} \phi/\!/s' \wedge \bigwedge_{s\xrightarrow{a}s'} [\overline{a}](\phi/\!/s')$
$\langle\tau\rangle\phi/\!/s = \langle\tau\rangle(\phi/\!/s) \vee \bigvee_{s\xrightarrow{\tau}s'} \phi/\!/s' \vee \bigvee_{s\xrightarrow{a}s'} \langle\overline{a}\rangle(\phi//_{s'})$
$\mathbf{T}/\!/s = \mathbf{T}$
$\mathbf{F}/\!/s = \mathbf{F}$

Figure 4. Partial evaluation function for parallel operator, where $t = (S, Act, \rightarrow)$ is the labelled transition system representing a process while $s \in S$ is the state in which we evaluate the formula [3].

This kind of problem can be found, for instance, when a large system is developed. Since the implementation cannot be immediately built based on specification 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. [3] 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 sub-processes.

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 (Figure 4). In order to further explain how partial model checking function acts on a given equational μ-calculus formula, we show the following example.

Example 2.2. *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 specified by the conjunction $\wedge_{P\xrightarrow{\tau}P'}\phi_{//_{P'}}$, where every formula $\phi_{//_{P'}}$ takes into account the behaviour of Q in composition with a τ successor of P.*
- *the τ action is due to the performance of two complementary actions by the two processes. So for every a-successor P' of P there is a formula $[\overline{a}](\phi_{//_{P'}})$.*

■

To sum up, the meaningful result given in [3] is the following lemma.

DOI: 10.1002/stvr

Lemma 2.1. *Given a process $P\|Q$ (where P is a finite-state process) and an equational specification $D \downarrow Z$ we have:*

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

This lemma allows to partially evaluate an equational μ-calculus formula w.r.t. the behaviour of a component of the considered system (P in the lemma), in such a way that 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 malicious component.

Remarkably, this function is exploited in [3] to perform model checking efficiently, where both P and Q are specified. In our setting, the process Q is not specified. Thus, given a certain system P, it is possible to find the property that the malicious component must satisfy to avoid a successful attack on the system. It is worth noticing that partial model-checking function may be automatically derived from the semantics rules used to define a language semantics. Thus, the proposed technique is very flexible. According to [3], when 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 bound by $|\phi| \times |P|$. Referring to [2], any assertion can be transformed to an equivalent simple assertion in linear time. Hence, we can conclude that the size of $\phi_{//P}$ is polynomial in the size of ϕ and P.

It is important to notice that a lemma similar to Lemma 2.1 holds for each CCS operator [2]. This is a meaningful result because it allows us to evaluate a formula according to a part of a system, whatever structure the system may have, *i.e.*, independently from how the different components of the system are related to one another.

3. SYNTHESIS OF RUN-TIME CONTROLLER PROGRAMS

Here, we propose process algebra *controller operators* as mechanisms to enforce security properties at run time. Indeed, our framework is based on process algebra, partial model checking and *open system paradigm* suggested for the modelling and the verification of systems [31, 35], and here extended to deal with the synthesis problem.

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

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

The main idea is that, when analysing security-sensitive systems S, the malicious component's behaviour -hereafter denoted by X - should not be fixed beforehand. A system should be secure regardless of the behaviour of X, which is exactly a *verification* problem of open systems, that can be formalized as follows:

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

where ϕ is a (temporal) logic formula expressing the security property. Using the open system approach, we develop a theory to enforce security properties. Our goal consists in protecting the system against possible malicious components. Indeed, we have to find a way to guarantee that the whole system behaves correctly, regardless the behaviour of a possible intruder. For that reason, we introduce

DOI: 10.1002/stvr

a controller operator and describe a mechanism for synthesizing a controller program that, according to the semantics of the controller operator, assures that the considered system results secure.

3.1. Synthesis of process algebra controller operators

In this section, we provide a definition of *process algebra controller operators* that forces the intruder to behave correctly, *i.e.*, as prescribed by a security property expressed as an equational μ-calculus formula.

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.

Example 3.1. *There is not a unique way to control a target system in order to enforce security properties. It is possible to use a different controller operator, according to which properties the system has to satisfy and how Indeed, it is possible to define several controller operators with different behaviours.*

As an example of controller operator, let us consider a controller operator with the following semantics definition:

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

where E represents the controller program and F the target. This operator models Schneider's automaton [48] when we consider only deterministic automata. Hence, it works by monitoring each target's sequence of actions step by step. Indeed, process E describes an allowed behaviour and, if both E and F perform an action a, consequently $E \triangleright F$ performs that same a action, otherwise $E \triangleright F$ halts.

We use controller operators in such a way that the specification of the system becomes:

$$\exists Y \quad \forall X \quad \text{s.t.} \quad S \| (Y \triangleright X) \models \phi \tag{3}$$

By using partial model checking it is possible to reduce it as follows:

$$\exists Y \quad \forall X \quad Y \triangleright X \models \phi' \tag{4}$$

This way the behaviour of the safe and known part of the system is considered directly into the formula ϕ'. The problem described by Statement (4) is about the target system X and the controller program Y. The controller program has to work only on X and does not care about the rest of the system.

Furthermore, even the problem in Statement (4) might not be easy to manage because of the presence of the universal quantification on all possible behaviours of the target X.

Hereafter we present a possible solution to this problem and a method for synthesizing a controller program for the given controller operator. First of all we 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 w.r.t. parallel operator, this set of formulas is closed under the partial model checking function. Moreover, for this class of formulas, the following result holds.

DOI: 10.1002/stvr

Proposition 3.1 ([13]) *Let E and F be two processes and $\phi \in Fr_\mu$. If $F \preceq E$ then $E \models \phi \Rightarrow F \models \phi$. The same result holds also if F and E are strong similar.*

Now, let us consider the following mild assumption on the controller operators we intend to use.

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

$$Y \triangleright X \preceq Y$$

Since we consider formulas in Fr_μ, if a controller operator semantics is defined in such a way that the resulting *controlled program* $Y \triangleright X$ is simulated by the *controller program* Y itself, then we are able to abstract from the universal quantification on X. In fact, in order to solve the problem in Statement (4) it is sufficient to solve the following reduced one:

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

The formulation (5) is easier to manage than the previous one. 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 found according to the Theorem 2.1. Consequently, we are able to prove the following result

Theorem 3.1. *Under Assumption 3.1, the problem described in Formula (4) 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 Walckiewicz in [51].

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 bounded availability and Chinese Wall policies are safety properties, whom we refer to as an example of how our mechanism works.

Referring to the controller operator defined in Example 3.1, the following holds.

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

$$Y \triangleright X \preceq Y$$

3.1.1. Feasibility issues for our controllers

The introduction of a controller operator helps to guarantee a correct behaviour of the entire system. Several semantics definitions can be given to describe the behaviour 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, $\triangleright'$:

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

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

DOI: 10.1002/stvr

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

$$\frac{E \xrightarrow{a} E' \; F \xrightarrow{a} F'}{E \rhd'' F \xrightarrow{a} E' \rhd'' F'} \qquad \frac{E \xrightarrow{a} E' \qquad F \overset{a}{\not\rightarrow}}{E \rhd'' F \xrightarrow{a} E' \rhd'' F}$$

The operator $\rhd''$ cannot be implemented if we are not able to decide a priori which are the possible next steps that the target agent is likely to perform. If we do not have the code of F at our disposal (*i.e.*, this is a black box) we cannot perform it easily. 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 always allow the correct action of the target. Thus, if the code of F was known in advance (remind CCS is Turing powerful), controller $\rhd''$ would be our favourite. In fact when applying the first rule, the external agent is able to execute the correct action while controller $\rhd''$ denies the unwanted situation when applying the second rule for checking it.

3.1.2. Two examples

In this section we show two examples of properties that can be enforced by using our framework. Using our framework, we synthesize a controller program especially designed for enforcing safety properties. As we have already recalled in the background section, the class of safety properties covers many meaningful security properties that can require a considered system in order to be satisfied [§].

3.1.2.1. An example of the whole procedure. Illustrating this simple example we show the whole procedure presented in this paper. To this aim, we firstly apply the partial model checking function, pointing out the necessary and sufficient conditions the target has to satisfy to not violate the security of the system. 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 $S \xrightarrow{a} S'$ so S' is $b.\mathbf{0}$.

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

It is worth noting that the process $Y = a.Y$ is a model of $\mathcal{D}$.

[§] In the ambit of the S3MS project for the security of mobile devices, several scenarios of application of this theory are described in the project's documentation.

DOI: 10.1002/stvr

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:

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

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.

3.1.2.2. The Chinese Wall *policy* To show an elementary example of a safety property which we are able to enforce, we present the *Chinese Wall* policy. It is a very common security property: Basically, assuming we have to have two data sets, once a user has accessed one set, then he cannot access the other. The Chinese Wall policy is expressed by the formula $\phi = Z \vee W$ where

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

Let us consider $S = \mathbf{0}$, thus $S|X$ is just X. We synthesize a controller program Y for enforcing ϕ. For instance, we consider the process

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

It is possible to note that such Y initially allows any possible behaviour of the unspecified component.

Let us consider the following instance: $X = open_A.open_B.X$. In such a case, by using the controller operator $\triangleright$, we obtain that the execution is halted after the first $open_A$ action because the target X attempts to perform the action $open_B$ that is not allowed once the $open_A$ one has been performed. Indeed, the execution steps are the following:

$$Y \triangleright X \xrightarrow{open_A} Y_1 \triangleright open_B.X$$

Hence, at the beginning, both options, 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.

3.1.3. Possible extensions of our framework: Parameterized Systems

In this section we show how our technique can be also used for dealing with *parametrized systems*.

A parametrized system describes an infinite family of (typically finite-state) systems (see [9]). Instances of the family can be obtained by fixing parameters.

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

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

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

Example 3.2. *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 through synchronization on $\bar{a}$ and a actions.*

DOI: 10.1002/stvr

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

$$\forall n \quad \forall X \quad P_n \| X \models \phi \tag{6}$$

It is possible to note that in the previous equation there are two universal quantifications: The first one refers to the number of instances of the process P, n, and the second one to the possible behaviour 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 3.1. *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 3.3. *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 [9], 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 2.1, $\forall j$ 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 called a *contracting sequence*. If it is possible to find the invariant formula ψ_ω for a chain of μ-calculus formulas, also known as *limit of the sequence*, then the following identity holds:

$$\forall X \quad (X \models \psi_\omega \Leftrightarrow \forall n \geq 1 \quad P_n \| X \models \phi) \tag{7}$$

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

In some cases it was not 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 [9, 15]).

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 no diamond nor μ operators [12].

It takes as an input both system S and formula ϕ and gives as output a process Y, described as a labelled graph or in ConSpec language, that is a model for ϕ', the formula obtained by the partial evaluation of ϕ by S. According to the theory developed in the previous section a such Y guarantees $S \| (Y \triangleright X)$ satisfies ϕ whatever X is. Note that here we are referring here to generic system S. Actually we do not explicitly deal with the particular case of parametrized systems.

DOI: 10.1002/stvr

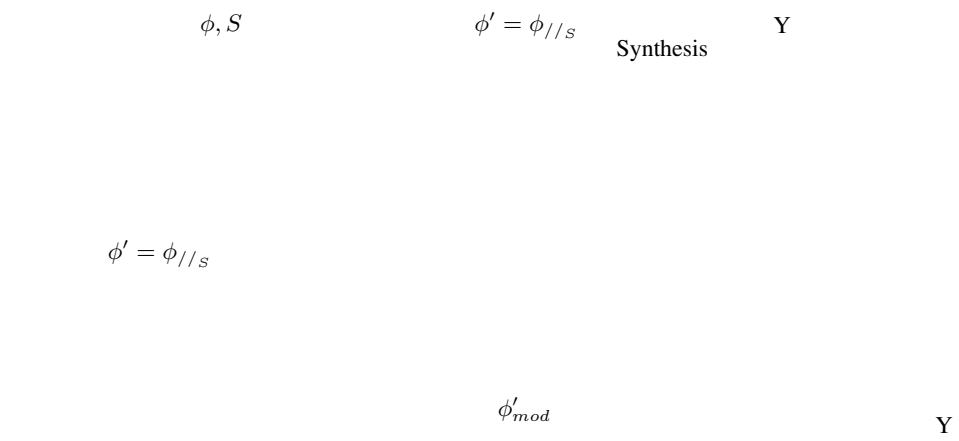


Figure 5. Architecture of the tool.

The tool is made up of two main parts (see Figure 5.a)): The first part implements the partial model checking function; the second one, by referring the satisfiability procedure, generates a process Y.

In Figure 5 there is a graphical representation of the architecture of the whole tool that we explain in more detail in the following section.

4.1. Architecture of the tool

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 [3]. 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 checking and it is presented in the standard output or written in a file.

It works by taking as an input both 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.

DOI: 10.1002/stvr

Synthesis *internal module.* The *Synthesis* module (Figure 5.b)) developed in O'Caml 3.09 (see [29]), implements a satisfiability procedure [51] for formulas expressing safety properties. Note that, according to the choice of implementing the Walukiewicz satisfiability procedure, the Synthesis module builds a model for a modal μ-calculus formula. It is made up of two components:

- the *Translator*, that takes as an input the equational μ-calculus formula generated by the MuDiv tool and "translates" it in an equivalent equational μ-calculus formula. It consists of several functions:
  - `fparser.ml` and `flexer.ml` permit to read the *MuDiv* output file and analyse it as input sequence in order to determine its grammatical structure with respect to our grammar.
  - `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 definition given in `type_for.ml`.
  - `convert.ml`, translates the equational μ-calculus formula ϕ' in the modal one ϕ'_{mod}.
- The *Synthesis* implements a satisfiability procedure for safety properties, described by Walukievicz in [51]. It consists in several functions. :
  - `model.ml` builds a graph for the considered formula ϕ'_{mod} according to the set of rules of the Walukievicz procedure. The generated graph is not yet a labelled transition system because some arcs derived from rules on logical connectives.
  - `goodgraph.ml` works in parallel to the previous function for establishing if the built graph is a model or not for the considered formula.
  - `simplify.ml` extracts the labelled transition system from the graph. This will be the model of ϕ'_{mod}.
  - Other minimal functions are `controllers.ml`, `printGraph.ml`, that print the obtained controller program and `main.ml` that creates the executable file (`.exe`).

We are also able to translate the output into the $ConSpec$ policy language [1] through a function `tr.ml` that is simply a parser from our language to ConSpec. Hence our tool can be used to effectively enforce security policies on mobile phones by using the framework proposed in [46].

4.2. 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 2.6.26.

We have tested some security properties expressed by logic formulas. In particular, we have tested the following security properties:

- *Only actions a are allowed*
- *After performing a it is not possible to perform c*
- *You cannot open a new file while another file is open*
- *It is possible to access "www.google.it" or "www.yahoo.it" but once one accesses "www.yahoo.it" it is possible to access to it only*
- *The Chinese Wall policy*

DOI: 10.1002/stvr

- *It is not allowed to perform three open actions sequentially*

Let us start by applying the partial model checking function to the set formulas expressing the properties listed above. We are interested in the *size* of the resulting formula. In this context, the size of the formula is the number of states of the corresponding graph generated by exploiting the Walukievicz procedure.

Our experiments concern the measurement of the time spent for generating a controller program w.r.t. the *size* of the formula obtained by applying partial model checking. Indeed the Synthesis module of our tool generates a controller program Y for the formula obtained as output of MuDiv tool.

Firstly, we consider $S = \mathbf{0}$. Hence the partial model checking function is the identity of the considered formula. Then, according to the property we consider, we evaluate it by partial model checking w.r.t. a selected system S. The resulting formula has a different size from the initial one. For instance, the formula expressing the property *It is possible to access "www.google.it" or "www.yahoo.it" but once one accesses "www.yahoo.it" it is possible to access to it only* evaluated by $S = \mathbf{0}$, initially has size 25. If we evaluate it by $S = yahoo.\mathbf{0}$ we obtain a new formula whose size is 54.

Several tests were performed on each formula. In the following table we have collected an average value of the results obtained from the experiments by focusing on the relation that exists among the size of formula and time.

Size	Total time	User time	System time
7	0m0.0096s	0m0.0044s	0m0.00367s
13	0m0.0103s	0m0.0046 s	0m0.00375s
15	0m0.0104s	0m0.0056s	0m0.004s
16	0m0.0112s	0m0.006s	0m0.0046s
25	0m0.0118s	0m0.0072s	0m0.0052s
54	0m0.0126s	0m0.0074s	0m0.0058s

We have observed both the *user time* and the *system time*. The *user time* is the time spent by the CPU in user mode to execute the application while the system time is the time spent by the CPU in system mode, *i.e.*, the time spent to execute system function. The *total time* of the application is approximately the sum of the user time and the system time. Indeed it is affected by the scheduling strategy of the OS platform.

In Figure 4.2 there is a graphical representation of the experimental results collected in the table.

4.3. Two case studies

Let us consider a mobile phone on which we have installed an application for the Internet connection. Let us also consider to require the following security properties on our device:

It is possible to access either "www.google.it" or "www.yahoo.it" but once one accesses "www.yahoo.it" it is possible to access it only.

This policy can be formalized as follows:

$$Z_1 =_\nu [\tau]Z_1 \wedge [\texttt{google}]Z_1 \wedge [\texttt{yahoo}]Z_2$$
$$Z_2 =_\nu [\tau]Z_2 \wedge [\texttt{yahoo}]Z_2 \wedge [\texttt{google}]F$$

DOI: 10.1002/stvr

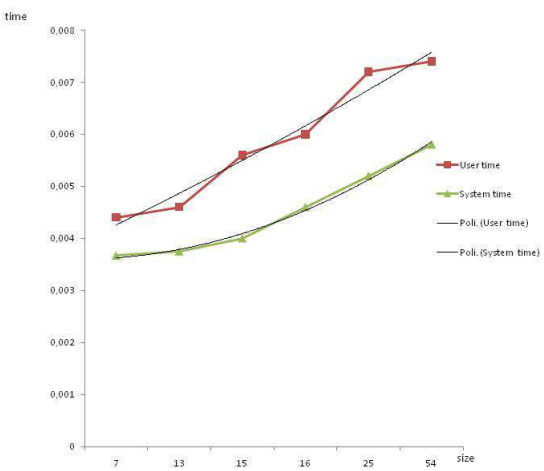


Figure 6. A graphical representation of the system time tendency.

DOI: 10.1002/stvr

By using our tool, we are able to generate a controller program for the controller operator $\triangleright$, whose description in terms of process algebra is the following:

$$Y = \texttt{google}.Y + \texttt{yahoo}.Z$$
$$Z = \texttt{yahoo}.Z$$

It is possible to note that Y is a model for the considered formula. Hence, let us consider, for instance a target $X = \texttt{yahoo.google}.0$ then

$$Y \triangleright X = \texttt{google}.Y + \texttt{yahoo}.Z \triangleright \texttt{yahoo.google}.0 \xrightarrow{\texttt{yahoo}} \texttt{yahoo}.Z \triangleright \texttt{google}.0$$

since the controller program aim to perform the action `yahoo` while X wants to performs `google` then the execution halts and the security requirement is satisfied.

Let us now consider the following policy on a smart-phone:

You cannot open a new file while another file is open.

It can be formalized by an equation system D as follows:

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

By using the $\triangleright$ operator, we halt the system whenever a user tries 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.

In order to enforce the policy on a smart-phone, by using a function of our tool, we can translate the definition of Y into ConSpec, as follows:

```
MAXINT 100 MAXLENGTH 160
SCOPE Global
SECURITY STATE
int = 1;
BEFORE open PERFORM
int == 1- > int := 2;
BEFORE close PERFORM
int == 2- > int := 3;
```

This policy can be really enforced on Java applications on mobile devices by using the S3MS run-time framework (the interested reader can find in [1] the full syntax of ConSpec).

DOI: 10.1002/stvr

5. RELATED WORK

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

In [48] 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. Starting from 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's automata. The other kinds of automata similarly allow a sort of modification in the output of the target system.

Our approach based on process algebra permits us to automatically synthesize a controller program for a chosen controller operator. We can define process algebra operators that precisely correspond to the Ligatti *et al.*'s automata (*e.g.*, see [36]). 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. [48] and [10, 11] since the synthesis problem is not addressed there.

Bartoletti, Degano and Ferrari in [7] refer to [48] 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 states 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 [8] 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 the prior work addresses the study of enforceable properties and related mechanisms. In [16] authors deal with a safety interface that permits to study if a module is safe or not in a given environment. Here the whole system is checked. Instead in our approach, through the partial model checking function, we are able to monitor only the necessary/untrusted parts of the system.

In [33] the authors provided a preliminary work in which several techniques were presented for automatically synthesizing systems enjoying a very strong security property, *i.e.*, $SBSNNI$ (see [17]). This is also called P_BNDC in [18]. In both works the authors did not deal with control theory.

The synthesis of controllers is a framework also addressed in other research areas (*e.g.*, see [6, 28, 47, 52, 21]). Our work on controller mechanisms starts from the necessity to make systems secure regardless the behaviour 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 behaviour of an unspecified components that interacts with the considered system.

Many other approaches to the controller synthesis problem are based on game theory (see [4, 27, 30, 34]). 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

DOI: 10.1002/stvr

the given properties. For instance in [4], the authors deal with the synthesis of controllers for discrete event systems by finding a winning strategies for parity games. In this framework it is possible to extend the specification of the supervised systems as well as the imposed 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 shares the interesting properties of the classical one. The method proposed in this paper for solving a control problem consists in transforming this problem into a problem of satisfiability of a μ-calculus formula. Indeed the set of models of this formula coincides with the set of controllers that solve the problem. On the other hand we synthesize controllers that work by monitoring only the possible un-trusted component of the system. Moreover they do not address 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 behaviour of unknown components. Our controllers are synthesized without any information about the process they are going to control.

In [43, 44, 45] the authors developed a theory for the synthesis of the maximally permissive controller. The authors have proposed a general synthesis procedure which always computes a maximal permissive controller when it exists. However they generate a maximal permissive controller knowing the behaviour of the process they are going to control. Contrastingly, we do not know anything a priori on the possible behaviour of an eventual malicious agent whose behaviour we want to control.

6. CONCLUSION AND FUTURE WORK

In this paper we have illustrated some results leading to 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, and also to tackle *synthesis* problems of secure systems.

Indeed, we solve the problem of finding a possible implementation of a controller program that, by monitoring the possible malicious component, makes the whole system 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.

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.

In [39], we have also given several semantics definitions for controller operators, starting from the work of Schneider [48]. 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, 48]). Schneider in [48] considered an enforcement mechanism as a program which controls that a given security property is respected. He has also given a definition of *security automaton* as an automaton that processes a sequence of input actions that has finite or infinite length. It works by monitoring the

DOI: 10.1002/stvr

target system, *i.e.*, an application whose behaviour 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 halt program execution before a security property is violated, though it cannot modify program behaviour. 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 [39], 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 behaviour of the corresponding security automaton. Moreover, it is possible to apply the theory developed in this work to such controller operators in order to synthesize controller programs able to enforce safety properties in these four different ways.

REFERENCES

1. I. Aktug, K. Naliuka: Conspec – A formal language for policy specification. *Electr. Notes Theor. Comput. Sci.* 197(1), pages 45–58, 2008.
2. H. R. Andersen. *Verification of Temporal Properties of Concurrent Systems*. PhD thesis, Department of Computer Science, Aarhus University, Denmark, 1993.
3. H. R. Andersen. Partial model checking (extended abstract). In *Proceedings of 10th Annual IEEE Symposium on Logic in Computer Science*, pages 398–407. IEEE Computer Society Press, 1995.
4. A. Arnold, A. Vincent, and I. Walukiewicz. Games for synthesis of controllers with partial observation. *Theoretical Computer Science*, 303(1), pages 7–34, 2003.
5. E. Asarin and C. Dima. Balanced timed regular expressions. *Electr. Notes Theor. Comput. Sci.*, 68(5), pages 16–33, 2002.
6. E. Badouel, B. Caillaud, and P. Darondeau. Distributing finite automata through petri net synthesis. *Journal on Formal Aspects of Computing*, 13, pages 447–470, 2002.
7. M. Bartoletti, P. Degano, and G. L. Ferrari. Enforcing secure service composition. In *CSFW*, pages 211–223. IEEE Computer Society, 2005.
8. 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.
9. 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. Springer, April 2003.
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.

DOI: 10.1002/stvr

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), pages 2–16, February 2005.
12. J. Bradfield and C. Stirling. *Modal logics and mu-calculi: an introduction*. Handbook of Process Algebra. Elsevier, 2001.
13. 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. Springer-Verlang, 1997.
14. F. Corradini, D. D'Ortenzio, and P. Inverardi. On the relationships among four timed process algebras. *Fundam. Inform.*, 38(4), pages 377–395, 1999.
15. 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, ACM Press. Los Angeles, California, 1977.
16. 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*, vol. 3688 of *Lecture Notes in Computer Science*, pages 246–260. Springer, 2005.
17. R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1), pages 5–33, 1994/1995.
18. R. Focardi and S. Rossi. Information flow security in dynamic contexts, *Journal of Computer Security*, 14(1), pages 65–110, 2006.
19. R. Focardi, R. Gorrieri, and F. Martinelli. Real-time Information Flow Analysis. *IEEE JSAC*, 21(1), pages 20–35, 2003.
20. 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), pages 168–186, 2004.
21. K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 342–356. Springer-Verlag, 2002.
22. M. Hennessy. *Algebraic Theory of Processes*. The MIT Press, 1988.
23. 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.
24. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.
25. P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1), pages 43–68, 1990.
26. D. Kozen. Results on the propositional $\mu-$calculus. *Theoretical Computer Science*, 27(3), pages 333–354, 1983.
27. O. Kupferman, P. Madhusudan, P. S. Thiagarajan, and M. Y. Vardi. Open systems in reactive environments: Control and synthesis. *Lecture Notes in Computer Science*, vol. 1877, pages 92–107, 2000.
28. 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.
29. X. Leroy, D. R. Damien Doligez, Jacques Garrigue, and J. Vouillon. The Objective Caml system release 3.09, 2004.
30. O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems (extended abstract). *Lecture notes in Computer Science* 900, pages 229–242, 1995.

DOI: 10.1002/stvr

31. F. Martinelli. *Formal Methods for the Analysis of Open Systems with Applications to Security Properties*. PhD thesis, University of Siena, December 1998.
32. 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.
33. F. Martinelli. Towards automatic synthesis of systems without information leaks. In *Proceedings of Workshop in Issues in Theory of Security (WITS)*, 2000.
34. F. Martinelli. Module checking through partial model checking. Technical Report IIT-TR06/2002, IIT-CNR, 2002.
35. F. Martinelli. Analysis of security protocols as open systems. *Theoretical Computer Science*, 290(1), pages 1057–1106, 2003.
36. F. Martinelli and I. Matteucci. Through modeling to synthesis of security automata. *Electr. Notes Theor. Comput. Sci.*, 179, pages 31–46, 2007.
37. 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.
38. I. Matteucci. Automated synthesis of enforcing mechanisms for security properties in a timed setting. *Electr. Notes Theor. Comput. Sci.*, 186, pages 101–120, 2007.
39. I. Matteucci. Synthesis of Secure Systems. PhD thesis, University of Siena (April 2008).
40. P. Merlin and G. V. Bochmann. On the construction of submodule specification and communication protocols. *ACM Transactions on Programming Languages and Systems*, vol. 5, pages 1–25, 1983.
41. 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, 1990.
42. 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.
43. 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.
44. S. Riedweg and S. Pinchinat. Maximally permissive controllers in all contexts. In *Workshop on Discrete Event Systems*, Reims, France, September 2004.
45. 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.
46. http://www.s3ms.org/index.jsp[Last visited: 14 July 2008].
47. H. Saidi. Towards automatic synthesis of security protocols. *In Logic-Based Program Synthesis Workshop, AAAI Spring Symposium*. March 2002.
48. F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1), pages 30–50, 2000.
49. R. S. Street and E. A. Emerson. An automata theoretic procedure for the propositional μ-calculus. *Information and Computation*, 81(3), pages 249–264, 1989.
50. 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*. Springer-Verlang. 1997.
51. I. Walukiewicz. *A Complete Deductive System for the μ-Calculus*. PhD thesis, Institute of Informatics, Warsaw University, June 1993.
52. 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.

DOI: 10.1002/stvr

Appendix: Technical Proof

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

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

Hence we have to prove that, if

$$E \triangleright F \xrightarrow{a} (E \triangleright F)' \; \exists E' \; E \stackrel{a}{\Longrightarrow} E' \text{ and } ((E \triangleright F)', E') \in \mathcal{S}$$

According to the semantics definition of $\triangleright$, $E \triangleright F \xrightarrow{a} (E \triangleright F)'$ if and only if $E \xrightarrow{a} E'$ and $F \xrightarrow{a} F'$. This implies that $(E \triangleright F)'$ is $E' \triangleright F'$. Moreover, it also implies that there exists E' s.t. $E \stackrel{a}{\Longrightarrow} E'$ and that $((E \triangleright F)', E') \in \mathcal{S}$ as required.

It is interesting to note, as lateral condition, that according to the semantics definition of $\triangleright$, the same relation holds also between $E \triangleright F$ and F, *i.e.*, $E \triangleright F \preceq F$.

□

Proof of Proposition 3.3: The proof comes directly from the Lemma 2.1. As a matter of fact, according to Lemma 2.1 and to Definition 3.1:

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

Reiterating this procedure n times we obtain:

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

□

DOI: 10.1002/stvr