The Compositional Security Checker: A Tool for the Verification of Information Flow Security Properties *

Riccardo Focardi
Dipartimento di Matematica Applicata ed Informatica
Università Ca' Foscari di Venezia
Via Torino 155,
I – 30173 Mestre (Ve), Italy
e_mail: focardi@dsi.unive.it

Roberto Gorrieri
Dipartimento di Scienze dell'Informazione
Università di Bologna
Mura Anteo Zamboni 7,
I – 40127 Bologna, Italy
e_mail: gorrieri@cs.unibo.it

Abstract

The Compositional Security Checker (CoSeC for short) is a semantic-based tool for the automatic verification of some compositional information flow properties. The specifications given as inputs to CoSeC are terms of the Security Process Algebra, a language suited for the specification of concurrent systems where actions belong to two different levels of confidentiality. The information flow security properties which can be verified by CoSeC are some of those classified in [8]. They derive from some classic notions, e.g. Non Interference [11]. The tool is based on the same architecture as the Concurrency Workbench [5], from which some modules have been imported unchanged. The usefulness of the tool is tested with the significant case-study of an access-monitor, presented in several versions in order to illustrate the relative merits of the various information flow properties that CoSeC can check. Finally, we present an application in the area of network security: we show that the theory (and the tool) can be reasonably applied also for singling out security flaws in a simple, yet paradigmatic, communication protocol.

1 Introduction

Security is a crucial property of system behaviour, requiring a strict control over the information flow among different parts of the system. In classic computer security, the main problem is to limit, and possibly to avoid, damage produced by malicious programs, called *Trojan Horses*, which try to leak secret information. There are several approaches to solve this problem.

In the *Discretionary Access Control* security (DAC for short), every *subject* (i.e., an active agent such as a user), decides the access properties of its *objects* (i.e., passive agents such as files). An example of DAC is the file management in Unix that allows a user to decide the access rights on her/his files. So, if a user A executes a Trojan Horse program, this can modify the security properties of A's objects. A solution to this problem is *Mandatory Access Control* (MAC for short), where access rules are imposed by the system. An example of MAC is *Multilevel Security* [2]: every object is bound to a security level, and so is every subject; information can flow from a certain object to a certain subject only if the level of the subject is greater than the level of the object. So a Trojan Horse, which operates at a certain level, has no way to *downgrade* information, and its action is restricted to that level. There are two access rules: *No Read Up* (a subject cannot read data from an upper level object) and *No Write Down* (a subject cannot write data in a lower level object). However, these access rules are not sufficient. It could be possible to transmit information indirectly using system side effects. For example, if two levels – 'high' and 'low' – share some finite storage resource (e.g., a hard disk), it is possible to transmit data from level 'high' to level 'low' by using the 'resource full' error message. For a high level transmitter, it is sufficient to alternatively fill or empty the resource in order to transmit a '1' or a '0' datum. Simultaneously, the low level receiver tries to write on the resource, decoding every error message as a '1' and every successful write as a '0'. It is clear

*Work partially supported by CNR, MURST and ESPRIT project LOMAPS n. 8130.

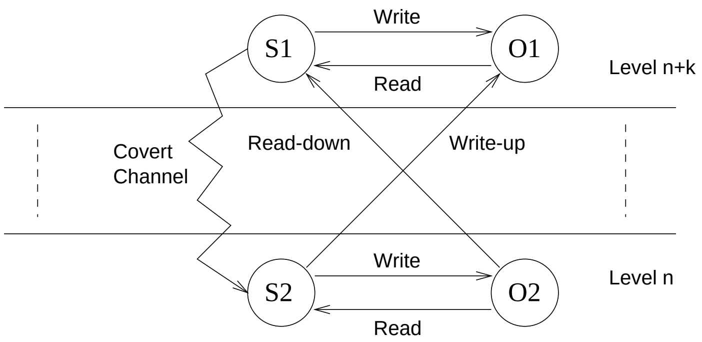

Figure 1: Information flows in multilevel security.

that such indirect transmissions, called *covert channels*, do not violate the two multilevel access rules (see Figure 1). Therefore it is necessary to integrate a MAC discipline with a covert channel analysis (see e.g. [28]).

An alternative, more general approach to computer security directly controls the whole flow of information, rather than the accesses of subjects to objects. To make this, it is necessary to choose a formal model of system behaviour and to define the information flow on such a model. By imposing some information flow rules, we can control both the direct and indirect transmissions. This is the approach we will follow in this paper.

In the literature, there are many different security definitions reminiscent of the information flow idea, each based on some system model (see e.g. [11, 29]). In [8] we have uniformly re-phrased them in the setting of *Security Process Algebra* (SPA, for short), then compared and classified them, leading to our proposed notion of *Bisimulation Non Deducibility on Compositions* (*BNDC*, for short).

SPA is an extension of CCS [18] – a language proposed to specify concurrent systems, whose programs are terms of an algebra comprising some operators for building systems bottom-up from smaller subsystems. The basic building blocks are the atomic activities, simply called *actions*; unlike CCS, in SPA actions belong to two different levels of confidentiality, thus allowing the specification of multilevel (actually, two-level) systems. As for CCS, the model used to describe the operational semantics of SPA is the *labelled transition system* model [17], where the states are the terms of the algebra. The *operational* semantics for a SPA term E is the portion of the SPA labelled transition system (LTS, for short) reachable from the initial state E. Then, in order to express that certain states are indistinguishable for an external observer, a semantic equivalence over terms/states is defined such that two terms are observationally indistinguishable iff they are equivalent.

For some of the information flow properties we investigated, we have provided useful algebraic characterizations. They are all of the following form. Let E be an SPA process term, let X be a security property, let $\approx$ be a semantic equivalence among process terms and let $\mathcal{C}_X$ and $\mathcal{D}_X$ be two SPA contexts [1] for property X. Then, we can say:

$$E \text{ is } X\text{-secure if and only if } \mathcal{C}_X[E] \approx \mathcal{D}_X[E].$$

Hence, checking the X-security of E is reduced to the "standard" problem of checking semantic equivalence between two terms having E as a subterm. In recent years a certain number of tools for checking semantic equivalence have been presented; among them, the Concurrency Workbench (CW for short) [5] is a well-known one.

The first aim of this work is to present a tool called *Compositional Security Checker* which can be used to check automatically (finite state) SPA specifications against some information flow security properties. The tool has the same modular architecture as CW (Version 6.1), from which some modules have been imported whole, and others modified. The tool is equipped with a parser, which transforms an SPA specification into

[1] An SPA context $\mathcal{G}$ is an SPA term "with a hole". E.g., $\mathcal{G}[-] = F + -$. The insertion of E in the context $\mathcal{G}$, written as $\mathcal{G}[E]$, has the effect of filling the hole with E. In the example, $\mathcal{G}[E] = F + E$. Subscript X simply means that $\mathcal{C}_X$ and $\mathcal{D}_X$ are two particular contexts for property X, i.e., it is used to give a name to contexts for property X.

a parse-tree; then, for the parsed specification CoSeC builds the labelled transition system following the operational rules defined in Plotkin' SOS style [23]. When a user wants to check if an SPA process E is X-secure, CoSeC first provides operational semantic descriptions to the terms $\mathcal{C}_X[E]$ and $\mathcal{D}_X[E]$ in the form of two LTSs; then verifies the semantic equivalence of $\mathcal{C}_X[E]$ and $\mathcal{D}_X[E]$ using their LTS representations. An interesting feature of CoSeC is the exploitation of the compositionality of some security properties in order to avoid, in some cases, the exponential state explosion due to the parallel composition operator.

The second aim of the work is to assess the merits of the various information flow properties presented in this paper. They are parametric w.r.t. the chosen notion of equivalence: if an equivalence $\approx_1$ is finer than $\approx_2$ then each security property based on $\approx_1$ is satisfied by a subset of the processes satisfying the corresponding security property based on $\approx_2$. We make this comparison by showing, through the running example of an access monitor, which kinds of flows are detectable by the various properties. In particular, we want to show that certain properties are not appropriate to deal with some kinds of information flows and so it is necessary to strengthen them by choosing a finer equivalence notion or, if this is not enough, to follow a different approach.

The basic ideas behind information flow security are general and potentially applicable to other related security areas. Indeed, the third aim of the paper is precisely to show that the theory of information flow security we have developed is applicable to typical problems of network security. For instance, the verification of a security protocol can be expressed as the verification of absence of information flow between any potential attacker and the partners of the communication. We study a simple, yet paradigmatic example (session keys distribution protocol), to illustrate the main features of such formal verification, made possible in practice only because of the availability of the tool. Indeed, often even small examples, such as those reported in this paper, may generate a large state space that cannot be checked reasonably without the help of an automatic tool.

The paper is organized as follows. In Section 2 we present SPA, together with the first version of our running example: an access monitor. In Section 3 we recall from [8] some of the security properties which are verified by CoSeC, giving some variations of the access monitor example. Section 4.1 reports the input-output behaviour of CoSeC, while Section 4.2 describes the architecture of the tool. The implementation of the security predicates is the subject of Section 5. Then, a sample session with the interactive tool is described in Section 6.1 and Section 6.2 is devoted to the formal verification of the access monitor case-study. Section 7 is about the state explosion problem and the compositional algorithm. Then we present the application of our theory and tool to the formal verification of a communication protocol in Section 8. In Section 9, we give a detailed comparison with related work in the area. In the final Section 10 we give some concluding remarks about future research.

2 SPA, Semantic Equivalences and Value-Passing

2.1 The Model

The model used to represent systems is the labelled transition system model [17]. A labelled transition system is essentially an automaton although it may have infinitely many states. "A significant advantage of having the model based on a standard notion like the automaton is that extensive literature and well-developed intuition become immediately applicable to the problem domain" (quoted from the DoD-NCSC Integrity report [22]). In addition, labelled transition systems constitute the preferred semantic domain for the operational description of many concurrent languages.

Definition 2.1 *A* labelled transition system (LTS) *is a triple* $(S, T, \rightarrow)$ *such that:*

- *S is a set of states,*
- *T is a set of labels (actions),*
- *$\rightarrow \subseteq S \times T \times S$ is a set of labelled transitions.* ∎

$(S_1, \alpha, S_2) \in \rightarrow$ (or equivalently $S_1 \stackrel{\alpha}{\rightarrow} S_2$) means that the system can move from the state S_1 to the state S_2 through the action α. A LTS is *finite* if it has a finite number of states and transitions.

2.2 The Language

The *Security Process Algebra* (SPA for short) is a slight extension of Milner's CCS [18], where the set of visible actions is partitioned into high level actions and low level ones in order to specify multilevel systems. [2]

SPA syntax is based on the following elements:

- A set $I = \{a, b, \ldots\}$ of *input* actions, a set $O = \{\bar{a}, \bar{b}, \ldots\}$ of *output* actions; $\mathcal{L} = I \cup O$ is called the set of *visible* actions, ranged over by α.
- A function $\bar{\cdot} : (I \to O) \cup (O \to I)$, extended to sets as follows: $\bar{L} \stackrel{\text{def}}{=} \{\bar{a} : a \in L\}$, with $L \subseteq \mathcal{L}$.
- Two sets Act_H and Act_L of high and low level actions which are closed with respect to function $\bar{\cdot}$ (i.e., $\overline{Act_H} = Act_H$, $\overline{Act_L} = Act_L$); moreover they form a covering of $\mathcal{L}$ and they are disjoint (i.e., $Act_H \cup Act_L = \mathcal{L}$, $Act_H \cap Act_L = \emptyset$).
- A set $Act = \mathcal{L} \cup \{\tau\}$ of actions (τ is the internal, invisible action), ranged over by μ.
- A set K of constants, ranged over by Z.

The syntax of SPA *agents* (or *processes*) is defined (as for CCS) as follows:

$$E ::= \underline{0} \;\big|\; \mu.E \;\big|\; E + E \;\big|\; E|E \;\big|\; E \setminus\!\setminus L \;\big|\; E[f] \;\big|\; Z$$

where $L \subseteq \mathcal{L}$ and $f : Act \to Act$ is such that $f(\bar{\alpha}) = \overline{f(\alpha)}$, $f(\tau) = \tau$. For every constant Z there must be the corresponding definition: $Z \stackrel{\text{def}}{=} E$. Moreover every E must be *guarded* on constants. This means that the recursive substitution of all the non prefixed (i.e., not appearing in a context $\mu.E'$) constants in E with their definitions terminates after a finite number of steps. This means that there exists a term obtainable by constant substitutions from E where all the possible initial actions are explicitly represented (through the prefix operator $\mu.E$). For instance, agent $A \stackrel{\text{def}}{=} B$ with $B \stackrel{\text{def}}{=} A$ is not guarded on constants. On the contrary, if B is defined as $a.A$, then A is guarded on constants.

The only difference from CCS is the restriction operator $\setminus\!\setminus$; intuitively $E \setminus\!\setminus L$ can execute all the actions E is able to do, provided that they do not belong to L. In CCS the corresponding operator $\setminus$ requires that the actions do not belong to $L \cup \bar{L}$. We need this slight modification in order to define, below in this Section, an additional input restriction operator. Concerning the other operators, we have that $\underline{0}$ is the empty process, which cannot do any action; $\mu.E$ can do an action μ and then behaves like E; $E_1 + E_2$ can alternatively choose [3] to behave like E_1 or E_2 ; $E_1|E_2$ is the parallel composition of E_1 and E_2, where the executions of the two systems are interleaved, possibly synchronized on complementary input/output actions, producing an internal τ; if E can execute action μ, then $E[f]$ performs $f(\mu)$.

Now, we define the standard restriction operator of CCS and two auxiliary operators, namely the hiding operator of CSP [14] and the input restriction operator, which are useful in characterizing some security properties in an algebraic style. We will have no further use for the $\setminus\!\setminus$ operator.

$$\begin{aligned} E \setminus L &\stackrel{\text{def}}{=} E \setminus\!\setminus L \cup \bar{L} \\ E/L &\stackrel{\text{def}}{=} E[f_L] \text{ where } f_L(x) = \begin{cases} x & \text{if } x \notin L \\ \tau & \text{if } x \in L \end{cases} \\ E \setminus_I L &\stackrel{\text{def}}{=} E \setminus\!\setminus L \cap I \end{aligned}$$

Intuitively, $E \setminus L$ can execute all the actions E is able to do, provided that they do not belong to $L \cup \bar{L}$, while $E \setminus_I L$ requires that the actions of E do not belong to $L \cap I$; E/L turns all the actions in L into internal τ's.

[2] Actually, only two-level systems can be specified; note that this is not a real limitation because it is always possible to deal with the multilevel case by grouping – in several ways – the various levels in two clusters.

[3] For notational convenience, we use sometimes the $\sum$ operator (indexed on a set) to represent a general n-ary (or even infinitary) sum operator.

Prefix	$\dfrac{-}{\mu.E \stackrel{\mu}{\rightarrow} E}$
Sum	$\dfrac{E_1 \stackrel{\mu}{\rightarrow} E_1'}{E_1 + E_2 \stackrel{\mu}{\rightarrow} E_1'}$ $\quad$ $\dfrac{E_2 \stackrel{\mu}{\rightarrow} E_2'}{E_1 + E_2 \stackrel{\mu}{\rightarrow} E_2'}$
Parallel	$\dfrac{E_1 \stackrel{\mu}{\rightarrow} E_1'}{E_1 \vert E_2 \stackrel{\mu}{\rightarrow} E_1' \vert E_2}$ $\quad$ $\dfrac{E_2 \stackrel{\mu}{\rightarrow} E_2'}{E_1 \vert E_2 \stackrel{\mu}{\rightarrow} E_1 \vert E_2'}$ $\quad$ $\dfrac{E_1 \stackrel{\alpha}{\rightarrow} E_1' \quad E_2 \stackrel{\bar{\alpha}}{\rightarrow} E_2'}{E_1 \vert E_2 \stackrel{\tau}{\rightarrow} E_1' \vert E_2'}$
Restriction	$\dfrac{E \stackrel{\mu}{\rightarrow} E'}{E \setminus\!\setminus L \stackrel{\mu}{\rightarrow} E' \setminus\!\setminus L}$ if $\mu \notin L$
Relabelling	$\dfrac{E \stackrel{\mu}{\rightarrow} E'}{E[f] \stackrel{f(\mu)}{\rightarrow} E'[f]}$
Constant	$\dfrac{E \stackrel{\mu}{\rightarrow} E'}{A \stackrel{\mu}{\rightarrow} E'}$ if $A \stackrel{\mathrm{def}}{=} E$

Table 1: The operational rules for SPA.

2.3 Operational Semantics and Equivalences

Let $\mathcal{E}$ be the set of SPA agents, ranged over by E and F. Let $\mathcal{L}(E)$ denote the *sort* of E, i.e., the set of the (possibly executable) actions occurring syntactically in E. The sets of high level agents and low level ones are defined as $\mathcal{E}_H \stackrel{\mathrm{def}}{=} \{E \in \mathcal{E} \mid \mathcal{L}(E) \subseteq Act_H \cup \{\tau\}\}$ and $\mathcal{E}_L \stackrel{\mathrm{def}}{=} \{E \in \mathcal{E} \mid \mathcal{L}(E) \subseteq Act_L \cup \{\tau\}\}$, respectively. Note that $\mathcal{E}_H \cup \mathcal{E}_L \subset \mathcal{E}$, i.e., there exist systems that execute both high and low level actions allowing communications between the two levels.

The operational semantics of SPA is the LTS $(\mathcal{E}, Act, \rightarrow)$, where the states are the terms of the algebra and the transition relation $\rightarrow \subseteq \mathcal{E} \times Act \times \mathcal{E}$ is defined by structural induction as the least relation generated by the axioms and inference rules reported in Table 1. The operational semantics for an agent E is the subpart of the SPA LTS reachable from the initial state E. We denote with $\mathcal{E}_{FS}$ the set of all the SPA agents with a finite LTS as operational semantics.

Now we introduce the idea of observable behaviour: two systems should have the same semantics if and only if they cannot be distinguished by an external observer. To obtain this we define an equivalence relation over states/terms of the SPA LTS, equating two processes when they are indistinguishable. In this way the semantics of a term becomes an equivalence class of terms.

It is possible to define various equivalences of this kind. We recall two of them. The first one is the classic definition of trace equivalence, according to which two agents are equivalent if they have the same execution traces. The second one discriminates agents also according to the nondeterministic structure of their LTSs. This equivalence is based on the concept of bisimulation [18].

Definition 2.2 *The expression* $E \stackrel{\alpha}{\Longrightarrow} E'$ *is a shorthand for* $E(\stackrel{\tau}{\rightarrow})^* E_1 \stackrel{\alpha}{\rightarrow} E_2(\stackrel{\tau}{\rightarrow})^* E'$, *where* $(\stackrel{\tau}{\rightarrow})^*$ *denotes a (possibly empty) sequence of* τ *labelled transitions. Let* $\gamma = \alpha_1 \ldots \alpha_n \in \mathcal{L}^*$ *be a sequence of actions; then* $E \stackrel{\gamma}{\Longrightarrow} E'$ *if and only if there exist* $E_1, E_2, \ldots, E_{n-1} \in \mathcal{E}$ *such that* $E \stackrel{\alpha_1}{\Longrightarrow} E_1 \stackrel{\alpha_2}{\Longrightarrow} \cdots \stackrel{\alpha_{n-1}}{\Longrightarrow} E_{n-1} \stackrel{\alpha_n}{\Longrightarrow} E'$. *We say that* E' *is* reachable *from* E *when* $\exists \gamma : E \stackrel{\gamma}{\Longrightarrow} E'$ *and we write* $E \Longrightarrow E'$. ■

Definition 2.3 *For any* $E \in \mathcal{E}$ *the set* $T(E)$ *of* traces *associated with* E *is defined as follows:* $T(E) = \{\gamma \in \mathcal{L}^* \mid \exists E' : E \stackrel{\gamma}{\Longrightarrow} E'\}$. E *and* F *are* trace equivalent *(notation* $E \approx_T F$*) if and only if* $T(E) = T(F)$. ■

We recall also the definition of weak bisimulation [18] over SPA agents. Bisimulation is based on an idea of mutual step-by-step simulation, i.e., when E executes a certain action moving to E', F must be able to

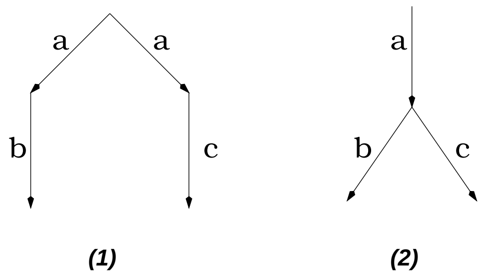

Figure 2: Systems not observationally equivalent

simulate this single step by executing the same action and moving to an agent F' which is again bisimilar to E' (this is because it must be able to simulate every successive step of E'), and vice-versa.

A weak bisimulation is a bisimulation which does not care about internal τ actions. So, when F simulates an action of E, it can also execute some τ actions before or after that action.

In the following, $E \stackrel{\hat{\mu}}{\Longrightarrow} E'$ stands for $E \stackrel{\mu}{\Longrightarrow} E'$ if $\mu \in \mathcal{L}$, and for $E \, (\stackrel{\tau}{\rightarrow})^* \, E'$ if $\mu = \tau$ (note that $(\stackrel{\tau}{\rightarrow})^*$ means "zero or more τ labelled transitions" while $\stackrel{\tau}{\Longrightarrow}$ requires at least one τ labelled transition).

Definition 2.4 *A relation $R \subseteq \mathcal{E} \times \mathcal{E}$ is a* weak bisimulation *if $(E, F) \in R$ implies, for all $\mu \in Act$,*

- *whenever $E \stackrel{\mu}{\longrightarrow} E'$, then there exists $F' \in \mathcal{E}$ such that $F \stackrel{\hat{\mu}}{\Longrightarrow} F'$ and $(E', F') \in R$;*
- *conversely, whenever $F \stackrel{\mu}{\longrightarrow} F'$, then there exists $E' \in \mathcal{E}$ such that $E \stackrel{\hat{\mu}}{\Longrightarrow} E'$ and $(E', F') \in R$.*

Two SPA agents $E, F \in \mathcal{E}$ are observationally equivalent, *notation $E \approx_B F$, if there exists a weak bisimulation containing the pair (E, F).* ■

In [18] it is proved that $\approx_B$ is an equivalence relation. Moreover, it is easy to see that $E \approx_B F$ implies $E \approx_T F$; indeed, if $E \approx_B F$ then F must be able to simulate every sequence of visible actions executed by E, i.e., every trace of E; since the simulation corresponds to the execution of the actions interleaved with some τ's, then every trace of E is also a trace for F. Symmetrically, E must be able to simulate every sequence of F. So $E \approx_T F$.

In Figure 2 there is an example of two trace-equivalent systems which are not observationally equivalent. In fact both (1) and (2) can execute the two traces ab and ac. However, it is not possible for (1) to simulate step-by-step process (2). In particular, (2) executes a and moves to a state where it can execute both b and c. Process (1) can simulate this first step of (2) but, after this, it cannot simulate (2) anymore since it is in a state where it can execute only b or c.

We want to conclude this section presenting a class of finite state agents. This is useful since CoSeC will eventually terminate its verification for all agents in this class. This class of agents consists of the so-called *nets of automata*:

$$\begin{array}{rcl} p & ::= & \underline{0} \;\big|\; \mu.p \;\big|\; p+p \;\big|\; Z \\ E & ::= & p \;\big|\; E|E \;\big|\; E \setminus L \;\big|\; E \setminus_I L \;\big|\; E/L \;\big|\; E[f] \end{array}$$

where for every constant Z there must be the corresponding definition $Z \stackrel{\rm def}{=} p$ (with p guarded on constants). It is easy to prove that every agent of this form is finite state. Nonetheless, this condition is not necessary, in the sense that other agents not belonging to the class of nets of automata are finite state as well. For instance, consider $B \stackrel{\rm def}{=} a.\underline{0} + D \setminus \{i\}$ with $D \stackrel{\rm def}{=} i.(o.\underline{0}|D)$. It can execute only an action a and then it stops, so it is clearly finite state. However note that it does not conform to the syntax of nets of automata since there is a parallel operator underneath a sum.

2.4 Value-Passing SPA

In this section we briefly present a value-passing extension of "pure" SPA (VSPA, for short). All the examples contained in this paper will use this value passing calculus, because it leads to more readable specifications than those written in pure SPA. Here we present a very simple example of a value-passing agent showing how it can be translated into a pure SPA agent. Then we define the VSPA syntax and we sketch the semantics by translating a generic VSPA agent into its corresponding SPA agent.

As an example, consider the following buffer cell [18]:

$$\begin{aligned} C &\stackrel{\mathrm{def}}{=} in(x).C'(x) \\ C'(x) &\stackrel{\mathrm{def}}{=} \overline{out}(x).C \end{aligned}$$

where x is a variable that can assume values in $\mathbb{N}$ (we usually write $x \in \mathbb{N}$). C reads a natural number n through action in and stores it in variable x. Then this value is passed to agent C' which can give n as output through action out moving again to C. So C represents a buffer which may hold a single data item.

Now we show how C can be translated into an SPA agent. The parametrized constant C' becomes a family of constants C'_v one for each value $v \in \mathbb{N}$. Similarly $\overline{out}(x)$ becomes a family $\overline{out}_v$ of prefixes. So the single definition for C' becomes the following family of definitions:

$$C'_v \stackrel{\mathrm{def}}{=} \overline{out}_v.C \quad (v \in \mathbb{N})$$

Now consider the prefix $in(x)$. To reflect the fact that it can accept any input value, binding variable x, we translate it into $\sum_{v \in \mathbb{N}} in_v$. So the definition becomes:

$$C \stackrel{\mathrm{def}}{=} \sum_{v \in \mathbb{N}} in_v.C'_v$$

VSPA is very similar to the value-passing CCS introduced in [18]. The main difference is that in VSPA we can have more than one parameter for actions and parameters are multi-sorted.

The syntax of VSPA agents is defined as follows:

$$\begin{aligned} E \ ::= \ & \underline{0} \ \big| \ a(x_1,\ldots,x_n).E \ \big| \ \bar{a}(e_1,\ldots,e_n).E \ \big| \ \tau.E \ \big| \ E+E \ \big| \ E|E \ \big| \ E \setminus L \ \big| \ E \setminus_I L \ \big| \ E/L \ \big| \\ & \big| \ E[f] \ \big| \ A(e'_1,\ldots,e'_n) \ \big| \ \textbf{if } b \textbf{ then } E \ \big| \ \textbf{if } b \textbf{ then } E \textbf{ else } E \end{aligned}$$

where the variables $x_1, \ldots, x_n$, the value expressions $e_1, \ldots, e_n$ and $e'_1, \ldots, e'_n$ must be consistent with the arity of the action a and constant A, respectively (the arity specifies the sorts of the parameters), and b is a boolean expression.

It is also necessary to define constants as follows: $A(x_1, \ldots, x_m) \stackrel{\mathrm{def}}{=} E$ where E is a VSPA agent which may contain no free variables except $x_1, \ldots, x_m$, which must be distinct. As in [18] the semantics of the value-passing calculus is given as a translation into the pure calculus. The translations rests upon the idea that a single label a of VSPA, with n parameters with sorts $S_1 \ldots S_n$, becomes the set of labels $\{a_{v_1 \ldots v_n} : v_i \in S_i, \forall i \in [1,n]\}$ in SPA. We consider only agents without free variables because if an agent has a free variable then it becomes a family of agents, one for each value of the variable. The translation can be given recursively on the structure of agents (see [18] for more details about this kind of translation).

Note that we do not partition the set of actions into two levels; we directly refer to the partition in the pure calculus. In this way it is possible for a certain action in VSPA to correspond, in the translation, to actions at different levels in SPA. This can be useful if we want a parameter representing the level of a certain action. As an example consider an action $access\_r(l,x)$ with $l \in \{high, low\}$ and $x \in [1,n]$, representing a read request from a user at level l to an object x; we can assign the high level to the actions with $l = high$ and the low level to the others in this way: $access\_r(high,x) \in Act_H$ and $access\_r(low,x) \in Act_L$ for all $x \in [1,n]$. A VSPA agent is finite state if its corresponding SPA agent is so. Hence, in general, a necessary condition is that every variable can assume values over a finite set only.

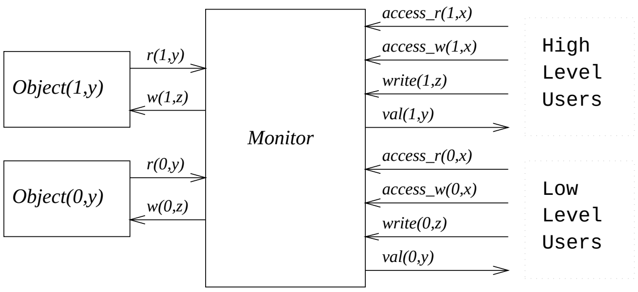

Figure 3: The Access Monitor for Example 2.5.

2.5 The Access Monitor

Here we give an example of a VSPA agent specification. It is an access monitor which handles read and write requests on two binary variables enforcing the multilevel security policy. We will analyze and modify this example in the next sections in order to assess the merits of the various information flow properties that we will propose.

Example 2.5 Consider the following system:

$$
\begin{array}{rcl}
Access\_Monitor\_1 & \stackrel{\mathrm{def}}{=} & (Monitor \mid Object(1,0) \mid Object(0,0)) \setminus L \\
Monitor & \stackrel{\mathrm{def}}{=} & access\_r(l,x). \\
& & (\textbf{if } x \leq l \textbf{ then} \\
& & \quad r(x,y).\overline{val}(l,y).Monitor \\
& & \ \textbf{else} \\
& & \quad \overline{val}(l,err).Monitor) \\
& & + \\
& & access\_w(l,x).write(l,z). \\
& & (\textbf{if } x \geq l \textbf{ then} \\
& & \quad \overline{w}(x,z).Monitor \\
& & \ \textbf{else} \\
& & \quad Monitor) \\
Object(x,y) & \stackrel{\mathrm{def}}{=} & \overline{r}(x,y).Object(x,y) + w(x,z).Object(x,z)
\end{array}
$$

where $x, y, z, l \in \{0,1\}$, $L = \{r, w\}$ and $\forall i \in \{0,1\}$ we have $r(1,i)$, $w(1,i)$, $access\_r(1,i)$, $val(1,i)$, $val(1,err)$, $access\_w(1,i)$, $write(1,i) \in Act_H$ and all the other actions are low level ones. Note that in $Access\_Monitor\_1$ every variable can assume values over a finite set only. When we translate it into SPA, we obtain a net of automata, hence $Access\_Monitor\_1$ is a finite state agent.

Figure 3 represents process $Access\_Monitor\_1$ that handles read and write requests from high and low level users on two binary objects: a high level variable and a low level one. It achieves *no read up* and *no write down* access control rules allowing a high level user to read from both objects and write only on the high one; conversely, a low level user is allowed to write on both objects and read only from the low one. Users interact with the monitor through the following access actions: $access\_r(l,x)$, $access\_w(l,x)$, $write(l,z)$ where l is the user level ($l = 0$ low, $l = 1$ high), x is the object ($x = 0$ low, $x = 1$ high) and z is the binary value to be written.

As an example, consider $access\_r(0,1)$ which represents a low level user ($l = 0$) read request from the high level object ($x = 1$), and $access\_w(1,0)$ followed by $write(1,0)$ which represents a high level user ($l = 1$) write request of value 0 ($z = 0$) on the low object ($x = 0$). Read results are returned to users through the output actions $val(l,y)$. This can be also an error in case of a read-up request. Note that if a high level user tries to write on the low object – through $access\_w(1,0)$ followed by $write(1,z)$ – such a request is not executed and no error message is returned.

In order to understand how the system works, let us consider the following transitions sequence representing the writing of value 1 in the low level object by the low level user:

$$\begin{array}{ll} & (Monitor \mid Object(1,0) \mid Object(0,0)) \setminus L \\ \stackrel{access\_w(0,0)}{\longrightarrow} & (write(0,z).\overline{w}(0,z).Monitor \mid Object(1,0) \mid Object(0,0)) \setminus L \\ \stackrel{write(0,1)}{\longrightarrow} & (\overline{w}(0,1).Monitor \mid Object(1,0) \mid Object(0,0)) \setminus L \\ \stackrel{\tau}{\longrightarrow} & (Monitor \mid Object(1,0) \mid Object(0,1)) \setminus L \end{array}$$

The trace corresponding to this sequence of transitions is $\gamma = access\_w(0,0).write(0,1)$ and so we can write:

$$(Monitor \mid Object(1,0) \mid Object(0,0)) \setminus L \stackrel{access\_w(0,0).write(0,1)}{\Longrightarrow} (Monitor \mid Object(1,0) \mid Object(0,1)) \setminus L$$

Note that, after the execution of the trace, the low level object contains value 1.

Access_Monitor_1 is a value passing specification of an access monitor. Its translation into CoSeC syntax for pure SPA is reported in Table 3 of Section 6.2. As an example, here we provide the translation of $Object(x,y)$ into the pure calculus by means of the following four constant definitions:

$$\begin{array}{lcl} Object_{00} & \stackrel{\mathrm{def}}{=} & \overline{r}_{00}.Object_{00} + w_{00}.Object_{00} + w_{01}.Object_{01} \\ Object_{01} & \stackrel{\mathrm{def}}{=} & \overline{r}_{01}.Object_{01} + w_{00}.Object_{00} + w_{01}.Object_{01} \\ Object_{10} & \stackrel{\mathrm{def}}{=} & \overline{r}_{10}.Object_{10} + w_{10}.Object_{10} + w_{11}.Object_{11} \\ Object_{11} & \stackrel{\mathrm{def}}{=} & \overline{r}_{11}.Object_{11} + w_{10}.Object_{10} + w_{11}.Object_{11} \end{array}$$

■

3 Some Information Flow Properties

In this section we present some of the security properties which can be verified using CoSeC. The common intuition behind all these properties is that the low level users should not be able to deduce anything about high level users' activity. The section is composed of four subsections. In the first three we present properties based on different equivalence notions. We start with the weakest and most intuitive one: trace equivalence. Then, we see that it is necessary to move to finer equivalence notions in order to detect possible high level deadlocks that can compromise the security of the system. For this reason, in the second subsection we briefly recall failure and testing equivalences [4, 21], which have been designed specifically to detect deadlocks, and we base our security properties on these notions. However, we discover that the failure/testing setting is not ideal for our purposes because of the way it deals with (potential) divergences. This leads us, in the third subsection, to prefer to base our security properties on the notion of weak bisimulation. In the last subsection we give further arguments which make us prefer weak bisimulation instead of failure/testing setting.

As this section reports some material already appeared in [8], it is useful to explain now what are the differences. In [8] it is possible to find further details regarding (most of) the properties based on trace and weak bisimulation equivalences, but there is no mention of failure/testing problems. The last bisimulation based property we present (called *SBSNNI*) is an approximation of a property (called *BNDC*) that is difficult to check automatically. This approximation improves the one we presented in [8]. The running example, which could not be checked easily without CoSeC, is new as well as Theorem 3.13; this Theorem is an interesting theoretical result of closure with respect to equivalence which can be exploited for increasing the verification efficiency of the tool.

3.1 Properties Based on Trace Equivalence

We start with *Non-deterministic Non Interference* (*NNI*, for short), which is a natural generalization (assuming two user groups only) to non-deterministic systems of the classic notion of *Non Interference*, proposed for deterministic systems in [11]. Intuitively, the high level does not interfere with the low level if the effects of high level inputs are not visible by a low level user. In order to check this, we consider every trace γ of the system containing high level inputs. Then, we look if there exists another trace γ' with the same subsequence of low level actions and without high inputs. A low level user, which can only observe low level actions, is not able to distinguish γ and γ'. As both γ and γ' are legal traces, we can conclude that the possible execution of the high level inputs in γ has no effect on the low level view of the system. This may be expressed algebraically as:

Definition 3.1 $E \in \mathit{NNI} \iff (E \setminus_I Act_H)/Act_H \approx_T E/Act_H$. ■

As a matter of fact, in E/Act_H all the high level actions are hidden, hence giving the low level view of the system; $E \setminus_I Act_H$ instead prevents traces from containing high level inputs. So, if the two terms are equivalent, then for every trace with high level inputs we can find another trace without such actions but with the same subsequence of low level actions.

It is possible to prove that *Access_Monitor_1* of Example 2.5 is *NNI*. In fact, the next example shows that *NNI* is able to detect whether the multilevel access control rules are implemented correctly in the monitor.

Example 3.2 Consider the following modified monitor [4] which does not control write accesses:

$$\begin{array}{rcl}
Access_Monitor_2 & \stackrel{\text{def}}{=} & (Monitor_2 \mid Object(1,0) \mid Object(0,0)) \setminus L \\
Monitor_2 & \stackrel{\text{def}}{=} & access_r(l,x). \\
& & (\textbf{if } x \le l \textbf{ then} \\
& & \quad r(x,y).\overline{val}(l,y).Monitor_2 \\
& & \ \textbf{else} \\
& & \quad \overline{val}(l,err).Monitor_2) \\
& & \ + \\
& & access_w(l,x).write(l,z).\overline{w}(x,z).Monitor_2
\end{array}$$

Now it is possible for a high level user to write in the low level object (action $access\_w(1,0)$ followed by $write(1,z)$) so the system is not secure. We have that *NNI* is able to detect this kind of direct flow. *Access_Monitor_2* can execute the following trace:

$$\gamma = access\_w(1,0).write(1,1).access\_r(0,0).\overline{val}(0,1)$$

In γ we have two accesses to the monitor: first a high level user modifies the value of the low object writing down value 1, and then the low user reads value 1 from the object. If we purge γ of high level actions, we obtain the sequence

$$\gamma' = access\_r(0,0).\overline{val}(0,1)$$

representing the reading by a low level user of value 1 from the low object. This trace is not a legal trace for *Access_Monitor_2* since the low object is initialized to value 0. Moreover, it is not possible to obtain a trace for *Access_Monitor_2* by adding to γ' only high level outputs, because all the high level outputs in *Access_Monitor_2* are prefixed by high level inputs. Hence γ' is not even a trace for $(Access\_Monitor\_2 \setminus_I Act_H)/Act_H$. In other words, it is not possible to find a trace γ'' of *Access_Monitor_2* with the same low level actions of γ and without high level inputs.

Since γ' is a trace for agent $Access\_Monitor\_2/Act_H$ but not for agent $(Access\_Monitor\_2 \setminus_I Act_H)/Act_H$, we conclude that *Access_Monitor_2* is not *NNI*. ■

The next example shows that *NNI* is not adequate to deal with synchronous communications and, consequently, it is too weak for SPA agents.

[4] In the following, if an agent is not specified (e.g. $Object(x,y)$) we mean that it has not been modified with respect to previous versions of the Access Monitor.

Example 3.3 Consider *Access_Monitor_1*, and suppose that we want to add a high level output signal which informs high level users about write operations of low level users in the high level object. This could be useful to know the integrity of high level information. We obtain the following VSPA agent with the new *written_up* action:

$$
\begin{array}{rcl}
Access\_Monitor\_3 & \stackrel{\mathrm{def}}{=} & (Monitor\_3 \mid Object(1,0) \mid Object(0,0)) \setminus L \\
Monitor\_3 & \stackrel{\mathrm{def}}{=} & access\_r(l,x). \\
& & (\ \textbf{if}\ x \leq l\ \textbf{then} \\
& & \quad r(x,y).\overline{val}(l,y).Monitor\_3 \\
& & \ \textbf{else} \\
& & \quad \overline{val}(l,err).Monitor\_3) \\
& & \ + \\
& & access\_w(l,x).write(l,z). \\
& & (\ \textbf{if}\ x = l\ \textbf{then} \\
& & \quad \overline{w}(x,z).Monitor\_3 \\
& & \ \textbf{else} \\
& & \quad\ \textbf{if}\ x > l\ \textbf{then} \\
& & \quad\quad \overline{w}(x,z).\overline{written\_up}(z).Monitor\_3 \\
& & \quad\ \textbf{else} \\
& & \quad\quad Monitor\_3)
\end{array}
$$

where $\forall i \in \{0,1\}$, $written\_up(i) \in Act_H$. It is possible to prove that $Access\_Monitor\_3$ is NNI. However, consider the following trace of $Access\_Monitor\_3$:

$$\gamma = access\_w(0,1).write(0,0).\overline{written\_up}(0).access\_w(0,1).write(0,0)$$

where a low level user writes two times value 0 to the high level object. If we purge γ of high level actions (i.e. of *written_up*) we obtain the following sequence:

$$\gamma' = access\_w(0,1).write(0,0).access\_w(0,1).write(0,0)$$

that cannot be a trace for $Access\_Monitor\_3$, because after every low level write operation there must be an action *written_up*.

So, if a low level user succeeds in executing the two write requests, then (s)he will know that some high level user has accepted the high level output $\overline{written\_up}$ (because of synchronous communications). In other words, a high level user can interfere with a low level one accepting or not the high level output $\overline{written\_up}$. NNI is not able to detect this, because it verifies only the high level input interferences over low level actions. In fact, since γ is a trace of $Access\_Monitor\_3$, then γ' is a trace for both $Access\_Monitor\_3/Act_H$ and $(Access\_Monitor\_3 \setminus_I Act_H)/Act_H$.

■

The example above shows that synchronous communications induce a symmetry over inputs and outputs. So we define a symmetric form of NNI. It requires that, for every trace γ, the sequence γ', obtained from γ by deleting all the high level actions, is still a trace. This property is called *Strong NNI* ($SNNI$ for short).

Definition 3.4 $E \in SNNI \Leftrightarrow E/Act_H \approx_T E \setminus Act_H$. ■

We have that $Access\_Monitor\_3$ is not $SNNI$ since γ' is a trace for $Access\_Monitor\_3/Act_H$ and γ' is not a trace for $Access\_Monitor\_3 \setminus Act_H$. We have the following result proved in [8]:

Proposition 3.5 $SNNI \subset NNI$. ■

It is possible to prove that $Access\_Monitor\_1$ of Example 2.5 is also $SNNI$.

The next example shows that trace equivalence – as the basic equivalence for security properties – is too weak; in particular, it is not able to detect deadlocks due to high level activities, that influence the security of a system.

Example 3.6 Suppose we have a high level action h_stop which explicitly stops the monitor. Obviously, in such a case there is a possible deadlock caused by a high level activity. Consider the following system:

$$
\begin{array}{rcl}
Access\_Monitor\_4 & \stackrel{\mathrm{def}}{=} & (Monitor\_4 \mid Object(1,0) \mid Object(0,0)) \setminus L \\
Monitor\_4 & \stackrel{\mathrm{def}}{=} & access\_r(l,x). \\
& & (\ \mathbf{if}\ x \leq l\ \mathbf{then} \\
& & \quad r(x,y).\overline{val}(l,y).Monitor\_4 \\
& & \ \mathbf{else} \\
& & \quad \overline{val}(l,err).Monitor\_4) \\
& & \ + \\
& & access\_w(l,x).write(l,z). \\
& & (\ \mathbf{if}\ x \geq l\ \mathbf{then} \\
& & \quad \overline{w}(x,z).Monitor\_4 \\
& & \ \mathbf{else} \\
& & \quad Monitor\_4) \\
& & \ + \\
& & h\_stop.\underline{0}
\end{array}
$$

where $h\_stop \in Act_H$. It is possible to prove that it is $SNNI$. This is because trace equivalence is not able to detect deadlocks and h_stop does not modify the low traces of the system. It could seem that a deadlock caused by a high level activity is not really interfering with any low level users, since a low level user, trying to make an access to the monitor, is not able to conclude that the monitor is blocked. However such a user can obviously deduce that the system is not blocked every time it accepts some access requests or gives some outputs. In the case of $Access\_Monitor\_4$, a low level user will never be able to conclude that h_stop has been executed; nonetheless, at every interaction with the system, the user will know that $Access\_Monitor\_4$ is not blocked and so that h_stop has not been executed yet. In [10] we show how this subtle information flow can be exploited in order to construct an information channel from high level to low level. ■

In order to detect this kind of flows, it is necessary to use some notion of equivalence which is able to detect deadlocks. One obvious possibility could be the failure/testing setting [4, 21], that has been designed for this purpose.

3.2 Detecting High Level Deadlocks Through Failure/Testing Equivalences

The failure semantics [4], introduced for the CSP language, is a refinement of the trace semantics where it is possible to observe which actions are not executable after a certain trace. In particular, a system is characterized by the so-called failure set, i.e., a set of pairs (s, X) where s is a trace and X is a set of actions. For each pair (s, X), the system must be able to reach a state executing trace s, where every action in X cannot be executed [5] . For instance, consider the agent $A \stackrel{\mathrm{def}}{=} a.\underline{0} + a.b.\underline{0}$ which can stop after the execution of a. Note that system A after a can refuse to execute action b. So A has the pair $(a, \{b\})$ in its failure set. Now, consider system $B \stackrel{\mathrm{def}}{=} a.b.\underline{0}$. It is trace equivalent to A, but after the execution of a it is always able to execute b. So B does not have $(a, \{b\})$ in the failure set, hence it is not failure equivalent to A. We conclude that failure semantics is able to detect the possible deadlock after a in system A.

[5]Indeed, there is a condition on the traces s in couples (s, X). It is required that during any execution of s no infinite internal computation sequences are possible. We will analyze this aspect more in details in the following.

A different characterization of failure equivalence has been given in [21]. It is based on the idea of tests. We can see a test as a process which can execute a particular *success* action $\omega \notin \mathcal{L}$. A test T is applied to a system E using the parallel composition operator $|$. A test T *may be satisfied* by system E if and only if system $(E|T) \setminus \mathcal{L}$ can execute ω:

$$E \ may \ T \text{ if and only if } (E|T) \setminus \mathcal{L} \stackrel{\omega}{\Longrightarrow} (E'|T') \setminus \mathcal{L}$$

A maximal computation of $(E|T) \setminus \mathcal{L} = ET_0$ is a sequence $ET_0 \stackrel{\tau}{\rightarrow} ET_1 \stackrel{\tau}{\rightarrow} \ldots \stackrel{\tau}{\rightarrow} ET_n \stackrel{\tau}{\rightarrow} \ldots$ which can be finite or infinite; if it is finite the last term must have no outgoing transitions. Now, we say that T *must be satisfied* by E if and only if every maximal computation of $(E|T) \setminus \mathcal{L} = ET_0$ contains a state ET_i which can execute ω:

$$E \ must \ T \text{ if and only if } \forall \text{ maximal computation of } ET_0 = (E|T) \setminus \mathcal{L}, \exists i \text{ such that } ET_i \stackrel{\omega}{\Longrightarrow} ET_i'$$

Two systems E and F are testing equivalent, $E \approx_{test} F$, if and only if

i) $E \ may \ T \Leftrightarrow F \ may \ T$

ii) $E \ must \ T \Leftrightarrow F \ must \ T$

It is easy to see that the first condition holds if and only if $E \approx_T F$. In fact, if E may satisfy T, then E is able to execute the trace which moves T to the state where it can execute ω [6]. It is not difficult to see that the second condition corresponds to failure equivalence. The basic idea is that if E fails to execute a certain action a after a trace γ, we can detect this with a test T which executes the complementary actions of γ followed by $\bar{a}$, and then executes ω. In fact, for that T we have that $E \ \not{must} \ T$.

Consider the version of *SNNI* based on testing equivalence:

Definition 3.7 *(testing SNNI)* $E \in TSNNI \iff E/Act_H \approx_{test} E \setminus Act_H$ ■

We have that $Access\_Monitor\_4 \notin TSNNI$. In fact it is sufficient to consider the test $T \stackrel{\text{def}}{=} \overline{access\_r}(0,0).\omega.\underline{0}$ which is able to detect the deadlock introduced by action *h_stop*. In particular, we have that:

$$Access\_Monitor\_4 \setminus Act_H \ must \ T$$

$$Access\_Monitor\_4/Act_H \ \not{must} \ T$$

Intuitively, when we hide *h_stop* in $Access\_Monitor\_4/Act_H$ we obtain an internal transition to a deadlock state. If the system moves to that state before the execution of $access\_r(0,0)$ the test will not be satisfied.

So, it seems that *TSNNI* is exactly the deadlock-sensitive extension of *SNNI* we were looking for. However, in the following we want to show that for systems with some high level loops or with τ loops, *TSNNI* is not interesting and is not able to detect security flaws. This is caused by the way testing equivalence deals with divergences (infinite sequences of τ actions). In fact, it is possible to prove that if E is divergent, then the (failure) condition (*ii*) of $\approx_{test}$ is verified if and only if also F is divergent. In the failure setting, when we have a divergent state we can observe the behaviour of a process only before reaching that state. All the actions after a divergence cannot be observed. For instance, we have that $A \stackrel{\text{def}}{=} a.b.0$ is not testing equivalent to $B \stackrel{\text{def}}{=} a.B'$ with $B' \stackrel{\text{def}}{=} \tau.B' + b.0$. The only difference between A and B is the divergence after the execution of a in B (see Figure 4).

Now, consider a system E with some high level loops and without divergences. When we hide high level actions in *TSNNI* with operation E/Act_H, we obtain some τ loops and so E/Act_H is divergent. As we said above, this will be failure equivalent to $E \setminus Act_H$ only if $E \setminus Act_H$ is divergent as well. However since E is without divergences, $E \setminus Act_H$ cannot have divergences too. We conclude that this kind of systems cannot be *TSNNI* no matter what the interactions between high and low level actions are. Note that every recursive high level system (with no low level actions and so secure by definition) is of this kind. All the access monitors we have seen so far have this feature as well.

[6] We could have more than one trace or more than one ω. However if a system may satisfy a test T, then it may satisfy also a test T' with only one ω and only one trace to it. T' corresponds to the trace executed by the system in order to satisfy the test T.

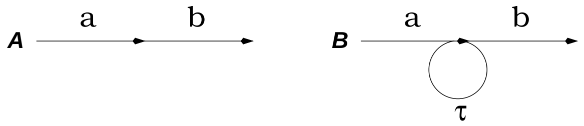

Figure 4: Divergence makes A and B non testing equivalent.

Moreover we have an even worse situation if we consider a divergent process D, as also $D \setminus Act_H$ and D/Act_H are divergent. So for these two processes (ii) is verified. This means that for divergent processes $\approx_{test}$ becomes equal to $\approx_T$. Hence *TSNNI* becomes equal to *SNNI* and, in general, all the properties based on testing equivalence become equal to the corresponding ones based on trace equivalence. In Section 3.4, we present an example of a divergent *TSNNI* process which, nonetheless, contains some high level deadlocks.

In the next section we move to weak bisimulation and observational equivalence. These notions are finer than failure/testing and so are able to detect deadlocks as well [7]. Moreover they give a *fair* interpretation of divergence, i.e., they assume that a τ loop will be executed an arbitrary, yet finite, number of times. In this way they can observe system behaviour after divergences. For instance, systems A and B above are observationally equivalent. More details and further reasons for preferring weak bisimulation to failure/testing will be given in Section 3.4.

3.3 Properties Based on Observational Equivalence

We introduce the bisimulation-based security properties *BNNI* and *BSNNI*, by substituting $\approx_B$ for $\approx_T$ in their SPA-based definitions. As expected, it can be proved [8] that each of these new properties is properly finer than its corresponding trace-based one. E.g., $BSNNI \subset SNNI$.

Definition 3.8 *(Bisimulation NNI, SNNI)*

(i) $E \in BNNI \iff E/Act_H \approx_B (E \setminus_I Act_H)/Act_H$

(ii) $E \in BSNNI \iff E/Act_H \approx_B E \setminus Act_H$ ■

Consider again *Access_Monitor_4* containing the *h_stop* event. It is neither *BSNNI* nor *BNNI*, as observational equivalence is able to detect deadlocks. In particular, $Access\_Monitor\_4/Act_H$ can move to $\underline{0}$ through an internal action τ, while $Access\_Monitor\_4 \setminus Act_H$ is not able to reach (in zero or more τ steps) a state equivalent to $\underline{0}$.

Now we want to show that *BSNNI* and *BNNI* are still not able to detect some potential deadlocks due to high level activities. This will induce us to propose another property based on a different intuition. Let us consider *Access_Monitor_1*. We can prove that such a system is *BSNNI* as well as *BNNI*. However, the following two dangerous situations are possible: (i) a high level user makes a read request without accepting the corresponding output from the monitor (remember that communications in SPA are synchronous) and (ii) a high level user makes a write request and does not send the value to be written. In both cases we have a deadlock due to a high level activity that *BNNI* and *BSNNI* are not able to reveal. To solve this problem, we are going to present a stronger property, called *Bisimulation-based Non Deducibility on Compositions* (*BNDC*, for short), we proposed in [8]. It is simply based on the idea of checking the system against all high level potential interactions. A system E is *BNDC* if for every high level process Π a low level user cannot distinguish E from $(E|\Pi) \setminus Act_H$. In other words, a system E is *BNDC* if what a low level user sees of the system is not modified by composing any high level process Π to E (see Figure 5).

Definition 3.9 *E is BNDC if and only if* $\forall \Pi \in \mathcal{E}_H,\ E/Act_H \approx_B (E \mid \Pi) \setminus Act_H$. ■

Figure 6 summarizes the relations among the bisimulation-based security properties (including another property we are going to present later). So we have that *BNDC* is stronger than *BNNI* and *BSNNI*. Note that $BSNNI \not\subseteq BNNI$ while for trace equivalence we had that $SNNI \subset NNI$. It is also interesting to observe that the trace based *BNDC* (*NDC*, for short) is exactly equal to *SNNI* ($NDC = SNNI$) [8]. So in the trace equivalence setting we gain nothing by introducing this new property.

[7] Indeed, only the strong (considering action τ as observable) bisimulation is finer than the strong failure/testing. The weak versions are incomparable because of the problem with divergences mentioned above.

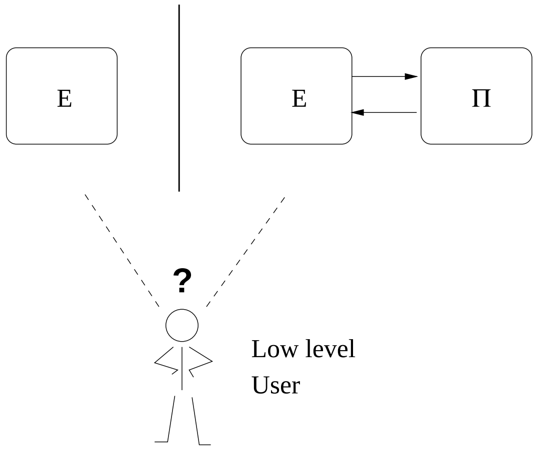

Figure 5: BNDC intuition

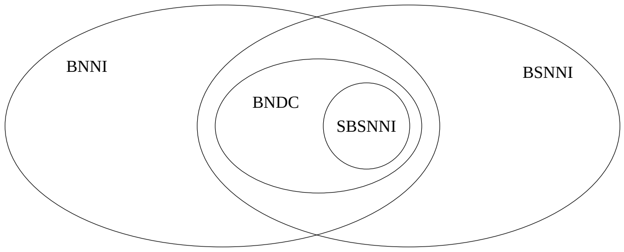

Figure 6: The inclusion diagram for bisimulation-based properties

Example 3.10 We want to show that $Access\_Monitor\_1$ is not $BNDC$. Consider $\Pi = \overline{access\_r}(1,1).\underline{0}$. System $(Access\_Monitor\_1 \mid \Pi) \setminus Act_H$ will be blocked immediately after the execution of the read request by Π, moving to the following deadlock state:

$$((\overline{val}(1,0).Monitor \mid Object(0,0) \mid Object(1,0)) \setminus L \mid 0) \setminus Act_H$$

This happens because Π executes a read request and does not wait for the corresponding return value (action val). We conclude that Π can interfere with low level users. Since there are no possible deadlocks in $Access\_Monitor\_1/Act_H$, we find out that $(Access\_Monitor\_1|\Pi) \setminus Act_H \not\approx_B Access\_Monitor\_1/Act_H$ and so $Access\_Monitor\_1$ is not $BNDC$.

Moreover, there is another potential source of deadlock when a high level user makes a write request and does not send the value to be written. In particular, the high level user $\Pi' = \overline{access\_w}(1,0).\underline{0}$ will block system $(Access\_Monitor\_1|\Pi') \setminus Act_H$ immediately after the execution of the write request by Π', moving the system to the following deadlock state:

$$(((write(1,0).Monitor + write(1,1).Monitor) \mid Object(0,0) \mid Object(1,0)) \setminus L \mid 0) \setminus Act_H$$

Again, we have that $(Access\_Monitor\_1|\Pi') \setminus Act_H \not\approx_B Access\_Monitor\_1/Act_H$. In order to obtain a $BNDC$ access monitor, we modify the monitor by adding an interface for each level which temporarily stores the output value of the monitor (passing it later to the users and thus making communication asynchronous) and that guarantees mutual exclusion within the same level; moreover, we use an atomic action for write request and value sending. Note that, because of the interface, actions $access\_r, access\_w$ and val become a_r, a_w and put, respectively. The resulting system follows (see Figure 7):

$$Access\_Monitor\_5 \stackrel{\text{def}}{=} (AM \mid Interf) \setminus N$$

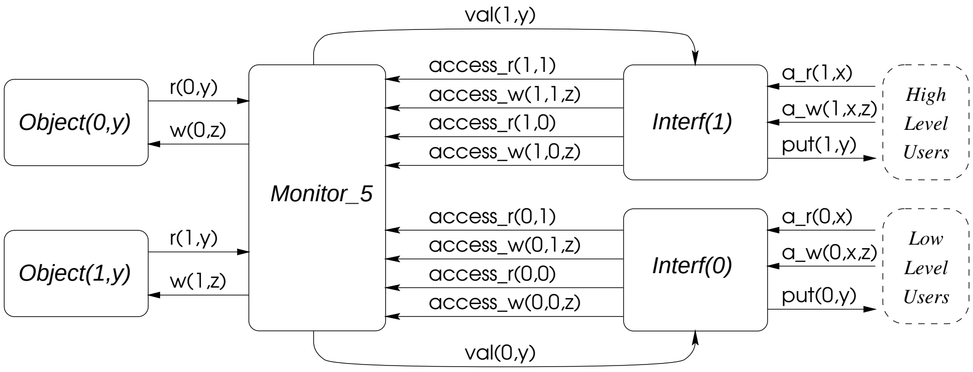

Figure 7: The $BNDC$ $Access\_Monitor\_5$

Version	Description	NNI	$SNNI$	$BNNI$	$BSNNI$	$BNDC$
1	Multilevel rules	X	X	X	X	
2	Without write control					
3	With high signal for integrity	X				
4	With explicit high level deadlock	X	X			
5	With buffers and atomic write	X	X	X	X	X

Table 2: Properties satisfied by the different versions of the access monitor.

$$
\begin{array}{rcl}
AM & \stackrel{\rm def}{=} & (Monitor\_5 \mid Object(1,0) \mid Object(0,0)) \setminus L \\
Monitor\_5 & \stackrel{\rm def}{=} & access\_r(l,x). \\
& & (\ \textbf{if } x \leq l \textbf{ then} \\
& & \quad r(x,y).\overline{val}(l,y).Monitor\_5 \\
& & \ \textbf{else} \\
& & \quad \overline{val}(l,err).Monitor\_5) \\
& & + \\
& & access\_w(l,x,z). \\
& & (\ \textbf{if } x \geq l \textbf{ then} \\
& & \quad \overline{w}(x,z).Monitor\_5 \\
& & \ \textbf{else} \\
& & \quad Monitor\_5) \\
Interf & \stackrel{\rm def}{=} & Interf(0) \mid Interf(1) \\
Interf(l) & \stackrel{\rm def}{=} & a\_r(l,x).\overline{access\_r}(l,x).val(l,k).\overline{put}(l,k).Interf(l) \\
& & + \\
& & a\_w(l,x,z).\overline{access\_w}(l,x,z).Interf(l)
\end{array}
$$

where $k \in \{0,1,err\}$, $L = \{r, w\}$, $N = \{val,\ access\_r,\ access\_w\}$ and $a\_r(1,x)$, $a\_w(1,x)$, $put(1,y) \in Act_H$ $\forall x \in \{0,1\}$ and $\forall y \in \{0,1,err\}$, while the same actions with 0 as first parameter belong to Act_L. It is possible to verify that $Access\_Monitor\_5$ is $BNDC$ using the automatically checkable property we are going to present. ■

Table 2 summerizes the properties satisfied by the different versions of the access monitor.

The above definition of *BNDC* is difficult to use in practice, because of the universal quantification on high level processes. It would be desirable to have an alternative formulation of *BNDC* which avoids universal quantification, exploiting local information only; however, a solution to this problem is still to be found. Nonetheless, here we propose a sufficient condition for *BNDC*, namely the *SBSNNI* property, which exploits local information only and moreover is compositional (i.e., if two systems are *SBSNNI* their composition is *SBSNNI* too).

Definition 3.11 *(SBSNNI: Strong BSNNI)*
A system $E \in SBSNNI$ if and only if for all E' reachable from E we have $E' \in BSNNI$. ■

SBSNNI is easily verifiable, as *BSNNI* is so; moreover, we can use it in order to check that a system is *BNDC* because of the following result (see Appendix A for the proof).

Proposition 3.12 $SBSNNI \subset BNDC$ ■

We end this section with a remark. In the automatic verification of properties, it can be very useful to work on a *reduced* system, i.e., a system equivalent to the original one, but with a smaller number of states. In fact, the Concurrency Workbench provides a procedure that minimizes the number of states, thus reducing a lot the time spent in the verification. In order to see if this proof strategy can be used also in our case, we need to prove that if a system E is *BNDC*, then any other observationally equivalent system F is *BNDC* too. Indeed, the theorem below (see Appendix A for the proof) shows that this is the case, also for all the other security properties we have discussed in this section. For this reason, we imported in CoSeC the minimization procedure of the Concurrency Workbench.

Theorem 3.13 *If $E \approx_B F$, then $E \in X \Leftrightarrow F \in X$, where X can be NNI, SNNI, NDC, BNNI, BSNNI, BNDC, SBSNNI.* ■

3.4 Testing vs. Bisimulation

In this Section we give three further arguments which make us prefer weak bisimulation instead of failure/testing setting. The first one is an example of a divergent system which can be normally analyzed by bisimulation based properties, but is not checkable using testing semantics; the second is related to complexity issues; the last one is about the class of LTSs for which the two equivalence notions are decidable.

Example 3.14 Consider once more *Access_Monitor_1*. We want to add a backup feature which is able to make a copy of the values stored in objects periodically. Obviously, there should be also a recovery procedure, but we do not model this in order to simplify as much as possible the example. We have a first process which represents the backup timer and sends periodically a signal in order to obtain a backup. It is an abstraction of a clock, since in SPA it is not possible to handle time.

$$Backup\_timer \stackrel{\rm def}{=} \overline{backup}.Backup\_timer$$

Then we slightly modify the *Monitor* process by inserting two actions which suspend its execution until the backup is finished.

$$\begin{array}{rcl} Monitor\_B & \stackrel{\rm def}{=} & \ldots \\ & & \text{the same as in } Access\_Monitor\_1 \\ & & \ldots \\ & & +\ start\_backup.end\_backup.Monitor \end{array}$$

The backup process is enabled by the timer, then it stops the monitor, reads the values of variables, stores them into two additional objects ($Object(2, y)$ and $Object(3, y)$) and resumes the monitor:

$$\begin{array}{rcl} Backup & \stackrel{\rm def}{=} & backup. \\ & & \overline{start\_backup}. \end{array}$$

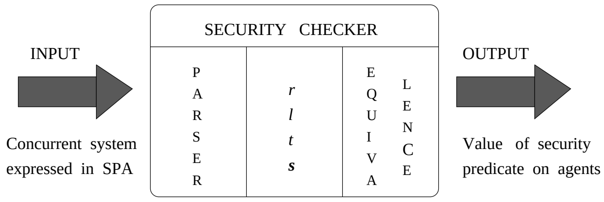

Figure 8: Structure of the CoSeC

$$\begin{array}{l} r(0,y).r(1,z). \\ \overline{w}(2,y).\overline{w}(3,z). \\ \overline{end\_backup}. \\ Backup \end{array}$$

The access monitor with backup is given by the following system:

$$\begin{aligned} Access\_Monitor\_B \overset{\text{def}}{=} & (Monitor\_B \mid Backup\_timer \mid Backup \mid Object(0,0) \mid Object(1,0) \mid \\ & \mid Object(2,0) \mid Object(3,0)) \setminus L \end{aligned}$$

where $L = \{r, w, start\_backup, end\_backup, backup\}$. As a result, the backup procedure of the system is something internal, i.e., an external user can see nothing of the backup task. This makes the system divergent. In fact, if the variable values are unchanged, then the backup procedure is a τ loop that moves the system to the same state where it started the backup. For weak bisimulation this is not a problem and we can analyze this new system as well. In particular, we can check with the tool that $Access\_Monitor\_B$ is observationally equivalent to $Access\_Monitor\_1$. So, because of Theorem 3.13 we can conclude that every security analysis made on $Access\_Monitor\_1$ is valid also for $Access\_Monitor\_B$. In particular $Access\_Monitor\_B$ is not $BNDC$.

On the other hand, if we try to analyze this system with some testing equivalence based property, we have an inaccurate result. Indeed $Access\_Monitor\_B$ is $TNDC$ and so this property is not able to detect the potential high level deadlocks revealed by $BNDC$. This happens because $Access\_Monitor\_B$ is divergent and so $Access\_Monitor\_B/Act_H$ and $(Access\_Monitor\_B|\Pi) \setminus Act_H$ $(\forall \Pi \in \mathcal{E}_H)$ are divergent as well. Thus they are failure equivalent and, since $Access\_Monitor\_B$ is $SNNI$ they are also trace (and so testing) equivalent. ■

There is also an interesting, practical difference between bisimulation and failure/testing. As we said above, it is possible to check bisimulation (observational equivalence) in polynomial time [16]. On the other side, we have that the problem of establishing language equivalence of nondeterministic finite automata is reducible in polynomial time to the problem of checking any of the testing, failure, and trace equivalences. Such a problem has been proved to be PSPACE-complete [27] The consequence is that all these problems are PSPACE-complete as well.

Moreover it is interesting to observe that failure/testing equivalences are not decidable on infinite state systems. On the other hand, there are some interesting results on the decidability of weak bisimulation over some classes of infinite state systems, e.g. totally normed Basic Process Algebras (BPA) [13] [8] . For instance it is possible to define a BPA agent representing an unbounded queue.

4 What is the Compositional Security Checker

In this Section we briefly describe the CoSeC structure and architecture. Before giving this description, we want to informally justify why the theory developed so far should be equipped with a tool. First of all, we need a tool to practically check the examples; even for small problems, the state space grows quite

[8]BPA [3] are basically the transition systems associated with Greibach normal form (GNF) context-free grammars in which only left-most derivations are permitted. In order to obtain an LTS an action is associated with every rule.

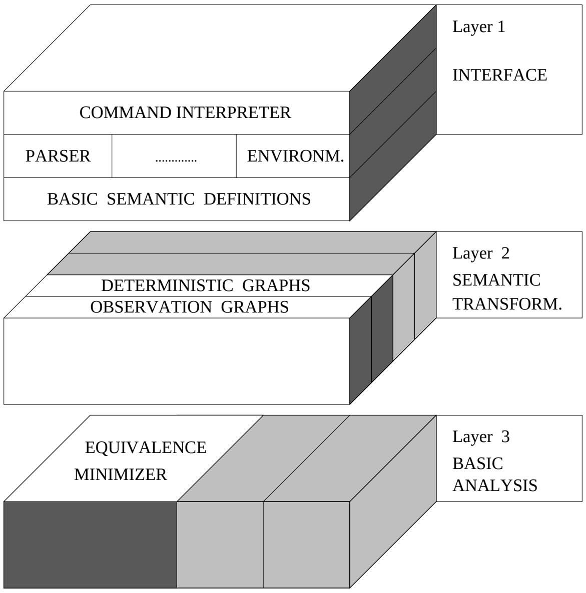

Figure 9: CoSeC Architecture.

beyond our human ability to manage it. A tool is also useful to help intuition on what flaws these security properties really capture; only through the tool we can analyze easily a large number of examples, and refine our understanding of the security properties and of the tractability of validating them. Finally, a tool gives an idea of which kind of verification techniques engineers should become acquainted with in the future to certify their products (e.g., security protocols).

4.1 Input-Output

The inputs of CoSeC are concurrent systems expressed as SPA agents. The outputs are answers to questions like: "does this system satisfy that specific security property ?". The structure of CoSeC is described in Figure 8. In detail, the tool is able:

- to parse SPA agents, saving them in suitable environments as parse trees;
- to give a semantics to these parse trees, building the corresponding rooted labelled transition systems (RLTS for short); this phase terminates only if the SPA term generates a finite LTS.
- to check if an agent satisfies a certain security property; the routine implemented for this purpose verifies the equivalence of two particular agents modeled as RLTS. In this way, future changes to the language will not compromise the validity of the core of the tool.

4.2 Architecture

The CoSeC has the same general architecture of the CW. In its implementation we have decided to exploit the characteristic of versatility and extensibility of CW. In particular CoSeC maintains the strongly modular architecture of CW.

Figure 9 shows the architecture of CoSeC. The modules of the system have been partitioned in three main layers: interface layer, semantic layer, analysis layer. In the interface layer we have the command interpreter. It allows us to define the agents and the set of high level actions; it also allows to invoke the security predicates and the utility functions on the behaviour of an agent. Then we have a parser which recognizes the SPA syntax of agents and stores them as parse trees in appropriate environments. The partition of the set of visible actions in the sets of high and low level actions has been obtained by defining the set of high level actions; by default, all the other possible actions are considered at low level. Then we have defined a function that, according to the operational semantic rule of SPA, provides all possible transitions for an agent. This function allows the construction of the LTS associated with an agent.

In the semantic layer, CoSeC uses two transformation routines to translate LTSs into deterministic and observation graphs [9] respectively. Since they both refer to processes modeled as LTSs, they have been imported from CW in CoSeC without any modification.

In the analysis layer, CoSeC uses a routine of equivalence and one of minimization that belong to the analysis layer of CW. These are a slight modification of the algorithm by Kanellakis and Smolka [16] which finds a bisimulation between the roots of two finite state LTSs by partitioning their states. It is interesting to note that a simple modification of this algorithm can be used to obtain the minimization of a finite state LTS.

5 Checking the Security Properties

In this Section we briefly explain how the system works in evaluating security predicates *NNI, SNNI, NDC, BNNI, BSNNI, SBSNNI*, and we discuss about their computational complexity. CoSeC computes the value of these predicates over finite state agents (i.e. agents possessing a finite state LTS), based on the definitions given in Section 3 that we report below in CoSeC syntax (for ease of parsing, in CoSeC the hiding and input restriction operators are represented by ! and ?, respectively.):

$$\begin{array}{rcl} E \in NNI & \Leftrightarrow & E!Act_H \approx_T (E?Act_H)!Act_H \\ E \in SNNI \equiv NDC & \Leftrightarrow & E!Act_H \approx_T E \setminus Act_H \\ E \in BNNI & \Leftrightarrow & E!Act_H \approx_B (E?Act_H)!Act_H \\ E \in BSNNI & \Leftrightarrow & E!Act_H \approx_B E \setminus Act_H \\ E \in SBSNNI & \Leftrightarrow & E' \in BSNNI,\ \forall E' \textit{ reachable from } E \end{array}$$

As for CW, the inner computation of the CoSeC follows three main phases.

Phase a) (the same for all predicates) CoSeC builds the RLTSs of the two agents of which it wants to compute the equivalence. For example in the case of NNI, CoSeC computes the transition graph for $(E?Act_H)!Act_H$ and $E!Act_H$. In this phase we do not have any particular problem with complexity, except for the intrinsic exponential explosion in the number of states of the RLTS due to parallel composition.

Phase b) (This is split into two depending on the semantics requested by the security predicate)

- **b1:** (for predicates *NNI, SNNI, NDC*) The two RLTSs obtained in Phase a) are transformed into deterministic graphs following the classic subset construction (see e.g. [15]). This algorithm has exponential complexity since it is theoretically possible, in the deterministic graph, to have a node for every subset of nodes in the original graph. However, experience shows that very often the number of obtained nodes is less than the number of nodes of the beginning graph because of the collapsing of the τ_transitions.
- **b2:** (for predicates *BNNI* and *BSNNI*) The two RLTSs obtained in Phase a) are transformed into observation graphs using the classic algorithms for the product of two relations and the reflexive transitive closure of a relation. This transformation has a $O(n^3)$ complexity, in which n is the number of nodes in the original graph.

Phase c) (For all predicates) The general equivalence algorithm [16] is applied to the graphs obtained in Phase b). Time and space complexities of this algorithm are $O(k * l)$ and $O(k + l)$ respectively, where l is the number of nodes and k is the number of edges in the two graphs. This is not a limiting factor in the computation of the observational and trace equivalences. In particular, for observational equivalence, in most cases 80% of computation time is due to the routine for reflexive transitive closure of Phase b).

[9] An observation graphs [5] is obtained by obscuring the precise amount of internal computation. In particular the edges of the LTS are modified in order to reflect the $\rightsquigarrow$ relation defined as: $n \overset{\epsilon}{\rightsquigarrow} n'$ iff $n \overset{\tau}{\Longrightarrow} n'$ and $n \overset{a}{\rightsquigarrow} n'$ iff $n \overset{a}{\Longrightarrow} n'$.

Since *SBSNNI* is verified by testing *BSNNI* over all the n states of the original graph, the resulting complexity will be n times the *BSNNI* complexity.

It is interesting to observe that the exponential explosion of the number of nodes of the transition graphs (Phase a), due to the operator of parallel composition, influences negatively the following phases, but it can not be avoided because of its intrinsic nature. A solution to this problem for the predicates *NNI, SNNI, NDC* and *SBSNNI* could be based on the exploitation of compositional properties (see Section 7 for more details).

6 Using CoSeC

6.1 Sample session

The style used in specifying SPA agents in CoSeC is the same used for CCS agents in CW. For example the command line [10]

`Command:` **bi** $A \quad h.'l.'h.A + 'h.'l.A$

defines the agent $A \stackrel{\mathrm{def}}{=} h.\bar{l}.\bar{h}.A + \bar{h}.\bar{l}.A$. As in CW the first letter of agents must be a capital letter and output actions have to be prefixed by $'$.

We assumed that the set of visible actions $\mathcal{L}$ is partitioned in two complete subsets Act_H and Act_L of high and low level actions respectively. With the command:

`Command:` **acth** $h \quad x$

we specify that $Act_H = \{h,' h, x,' x\}$. In this way we obtain that $h,' h, x,' x$ are considered as high level actions and any other action as low level one.

Now, we can check whether agent A is NNI secure:

`Command:` **nni** A
`true`

CoSeC tells us that A is NNI secure. Now we can check if agent A is SNNI secure too:

`Command:` **snni** A
`false`

So A is *NNI* secure but is not *SNNI* secure. If we want to know why such a system is not *SNNI* we can use the *debugging* version of the *SNNI*:

`Command:` **d_snni** A

```
false
Agent A!ActH
can perform action sequence 'l
which agent A\ActH
cannot
```

The tool shows a (low level) trace which distinguishes processes A/Act_H and $A \setminus Act_H$. The trace is $\bar{l}$ which can be executed only by the first one. This can be useful to understand why a process is not secure. Finally the command **quit** causes an exit to the shell.

6.2 Checking the Access Monitor

In this Section we use CoSeC to automatically check all the versions of the access monitor discussed in Example 3.10. Since CoSeC works on SPA agents we have to translate all the VSPA specifications into SPA. Consider once more $Access\_Monitor\_1$. Table 3 reports the translation of $Access\_Monitor\_1$ specification

```
bi  Access_Monitor_1
 (Monitor  |  Object_l0  |  Object_h0)\ L

bi  Monitor
 access_r_hh.(rh0.'val_h0.Monitor + rh1.'val_h1.Monitor) + \
 access_r_lh.'val_l_err.Monitor + \
 access_r_hl.(rl0.'val_h0.Monitor + rl1.'val_h1.Monitor) + \
 access_r_ll.(rl0.'val_l0.Monitor + rl1.'val_l1.Monitor) + \
 access_w_hh.(write_h0.'wh0.Monitor + write_h1.'wh1.Monitor) + \
 access_w_lh.(write_l0.'wh0.Monitor + write_l1.'wh1.Monitor) + \
 access_w_hl.(write_h0.Monitor + write_h1.Monitor) + \
 access_w_ll.(write_l0.'wl0.Monitor + write_l1.'wl1.Monitor)

bi Object_h0
 'rh0.Object_h0 + wh0.Object_h0 + wh1.Object_h1

bi Object_h1
 'rh1.Object_h1 + wh0.Object_h0 + wh1.Object_h1

bi Object_l0
 'rl0.Object_l0 + wl0.Object_l0 + wl1.Object_l1

bi Object_l1
 'rl1.Object_l1 + wl0.Object_l0 + wl1.Object_l1

basi L
 rh0  rh1  rl0  rl1  wh0  wh1  wl0  wl1

acth
 rh0 rh1 wh0 wh1 access_r_hh access_r_hl val_h0 val_h1 val_h_err \
 access_w_hh access_w_hl write_h0 write_h1
```

Table 3: Translation of Access_Monitor_1 to CoSeC syntax for SPA.

into the CoSeC syntax for SPA [11] . It has been used a new command **basi** which binds a set of actions to an identifier. Moreover, the \ character at the end of a line does not represent the restriction operator, but is the special character that permits to break in more lines the description of long agents and long action lists.

We can write to a file the contents of Table 3 and load it, in CoSeC, with command **if** <*filename*>. Now we can check that *Access_Monitor_1* satisfies all the security properties except *SBSNNI* using the following command lines:

`Command:` **bnni** *Access_Monitor_1*
`true`
`Command:` **bsnni** *Access_Monitor_1*
`true`
`Command:` **sbsnni** *Access_Monitor_1*
`false:` $(\texttt{'val\_h1.Monitor} \,|\, \texttt{Object\_l1} \,|\, \texttt{Object\_h1}) \setminus \texttt{L}$

Note that when CoSeC fails to verify *SBSNNI* on a process E, it gives as output an agent E' which is reachable from E and is not *BSNNI*.

So we have found that $Access\_Monitor\_1 \in BSNNI, BNNI$ and $Access\_Monitor\_1 \notin SBSNNI$. Since we have that $SBSNNI \subset BNDC \subset BSNNI, BNNI$, we cannot conclude whether *Access_Monitor_1* is *BNDC* or not. However, using the output state E' of the *SBSNNI* verification, it is easy to find a high level process Π which can block the monitor. Indeed, in the state given as output by *SBSNNI*, the monitor is waiting for the high level action `'val_h1`; so, if we find a process Π which moves the system to such a state and does not execute the `val_h1` action, we will have a high level process able to block the monitor. It is sufficient to consider $\Pi = {'access\_r\_hh}.0$. Agent $(Access\_Monitor\_1|\Pi) \setminus Act_H$ will be blocked immediately after the execution of the read request by Π, moving to the following deadlock state:

$$((\texttt{'val\_h0.Monitor} \,|\, \texttt{Object\_l0} \,|\, \texttt{Object\_h0}) \setminus \texttt{L} \,|\, \texttt{0}) \setminus \texttt{Act}_{\texttt{H}}$$

[10] Here we use the typewriter style for CoSeC messages (such as the prompt "`Command:`"); the bold style for CoSeC commands and the italic style for the remaining text (such as agents and sets) inserted by users.

[11] In the translation, we use values $\{l, h\}$ in place of $\{0, 1\}$ for the levels of users and objects in order to make the SPA specification clearer. As an example $access\_r(1, 0)$ becomes `access_r_hl`.

agent	B	D	$B\|D\|B$	$B\|D\|D\|B$
state number	3	3	27	81
time spent	<1 sec.	<1 sec.	~11 sec.	~270 sec.

Table 4: Number of states and time spent on a SPARC station 5.

(this state differs from the one given as output by *SBSNNI* only for the values stored in objects). It is possible to verify that $Access\_Monitor\_1 \notin BNDC$ by checking that $(Access\_Monitor\_1|\Pi) \setminus Act_H \not\approx_B Access\_Monitor\_1/Act_H$ using the following command:

`Command:` **bi** $Pi \quad 'access\_r\_hh.0$
`Command:` **eq**
`Agent:` $(Access\_Monitor\_1 \mid Pi) \setminus acth$
`Agent:` $Access\_Monitor\_1 \; ! \; acth$
`false`

As we said in Example 3.10, such a deadlock is caused by synchronous communications in SPA. Moreover, using the CoSeC output again, we can find out that also the high level process $\Pi' = 'access\_w\_hl.0$ can block $Access\_Monitor\_1$, because it executes a write request and does not send the corresponding value. Hence, in Example 3.10 we proposed the modified system $Access\_Monitor\_5$ with an interface for each level and atomic actions for write request and value sending. We finally check that this version of the monitor is *SBSNNI*, hence *BNDC* too:

`Command:` **sbsnni** $Access\_Monitor\_5$
`true`

7 State Explosion and Compositionality

We now want to explain how the parallel composition operator can increase exponentially the number of states of the system, and then how it can slow down the execution speed of security predicate verification. Let us define in CoSeC the two agents B, D and the set Act_H of high level actions:

`Command:` **bi** $B \quad y.a.b.B + a.b.B$
`Command:` **bi** $D \quad 'a.'b.(x.D + D)$
`Command:` **acth** $x \;\; y$

Let us check now if B and D are *SBSNNI* secure:

`Command:` **sbsnni** B
`true`
`Command:` **sbsnni** D
`true`

As we will see, *SBSNNI* is a compositional property, so the two agents $B|D|B$ and $B|D|D|B$ must also be *SBSNNI* secure. Hence the verification of these two agents could be reduced to the verification of their two basic components B and D only. The time spent in verifying *SBSNNI* directly on $B|D|B$ and $B|D|D|B$ is very long. Using the **size** command of CoSeC, which computes the number of states of an agent, we can fill in Table 4, which points out the exponential increase of the number of states and the consequent increase of the computation time for verification of *SBSNNI*. The next theorem states that *SBSNNI* is preserved by the parallel and the restriction operators (see Appendix A for the proof). Similar results hold for *NNI* and *SNNI* [8].

Theorem 7.1 *The following hold:*

(i) $E, F \in SBSNNI \Longrightarrow (E|F) \in SBSNNI$

(ii) $E \in SBSNNI, L \subseteq \mathcal{L} \Longrightarrow E \setminus L \in SBSNNI$ ■

CoSeC is able to exploit the compositionality of security properties through an algorithmic scheme we are going to present. For a certain compositional property P this scheme also requires the following condition: if $Z \stackrel{\text{def}}{=} E$ and E is P-secure then also Z is P-secure. This condition is satisfied by all the above presented properties because of Theorem 3.13.

Definition 7.2 *(Compositional Algorithm) Let $P \subseteq \mathcal{E}$ be a set of SPA agents such that*

- $E, E' \in P \Longrightarrow E|E' \in P$
- $E \in P, L \subseteq \mathcal{L} \Longrightarrow E \setminus L \in P$
- $E \in P, Z \stackrel{\text{def}}{=} E \Longrightarrow Z \in P$

and let A_P be a decision algorithm which checks if a certain agent $E \in \mathcal{E}_{FS}$ belongs to P; in other words, $A_P(E) = true$ if $E \in P$, $A_P(E) = false$ otherwise. Then we can define a compositional algorithm $A'_P(E)$ in the following way:

1) *if E is of the form $E' \setminus L$, then compute $A'_P(E')$; if $A'_P(E') = true$ then return true, else return the result of $A_P(E)$;*
2) *if E is of the form $E_1|E_2$, then compute $A'_P(E_1)$ and $A'_P(E_2)$; if $A'_P(E_1) = A'_P(E_2) = true$ then return true, else return the result of $A_P(E)$;*
3) *if E is a constant Z with $Z \stackrel{\text{def}}{=} E'$, then return the result of $A'_P(E')$;*
4) *if E is not in any of the three forms above, then return $A_P(E)$.* ■

The compositional algorithm $A'_P(E)$ works as the given algorithm $A_P(E)$ when the outermost operator of E is neither the restriction operator, nor the parallel one, nor a constant definition. Otherwise, it applies componentwise to the arguments of the outermost operator; if the property does not hold for them, we cannot conclude that the whole system is not secure, and we need to check it with the given algorithm.

Note that the compositional algorithm exploits the assumption that property P is closed with respect to restriction and uses this in step 1. This could seem of little practical use, as the dimension of the state space for, let say, E is often bigger than that of $E \setminus L$. However, parallel composition is often used in the form $(A|B) \setminus L$ in order to force some synchronizations, and so if we want to check P over A and B separately, we must be granted that P is preserved by both parallel and restriction operators.

To obtain the result for $A'_P(F)$, we essentially apply – in a syntax-driven way – the four rules above recursively, obtaining a proof tree having (the value of) $A'_P(F)$ as the root and the various (values of) $A_P(E)$'s on the leaves for the subterms E of F on which the induction cannot be applied anymore. The following theorem justifies the correctness of the compositional algorithm, by proving that the evaluation strategy terminates and gives the same result as the given algorithm $A_P(F)$.

Theorem 7.3 *Let $F \in \mathcal{E}_{FS}$. If the agent E' occurring in step 1 belongs to $\mathcal{E}_{FS}$ each time the algorithm A'_P executes that step, then $A'_P(F)$ terminates and $A_P(F) = A'_P(F)$.*

PROOF. First we want to prove that, in computing $A'_P(F)$, if the evaluation of the given algorithm A_P is required on an agent E, then E belongs to $\mathcal{E}_{FS}$. The proof is by induction on the proof tree for the evaluation of $A'_P(F)$. The base case is when F can be evaluated by step 4; as – by hypothesis – agent F is finite state, the thesis follows trivially. Instead, if F is of the form $E' \setminus L$, then – by the premise of this theorem – $E' \in \mathcal{E}_{FS}$, and the inductive hypothesis can be applied. In step 2, as $F = E_1|E_2$, we have that $E_1, E_2 \in \mathcal{E}_{FS}$, and the inductive hypothesis can be applied to prove the thesis. Similarly for step 3, as constant Z is finite state if and only if the defining agent E' is so. So, when the algorithm executes $A_P(E)$ in steps 1, 2, 3 or 4, it always terminates because $E \in \mathcal{E}_{FS}$.

To complete the proof concerning termination of the compositional algorithm, we still have to prove that the inductive evaluation, in steps 1, 2 and 3, cannot loop; in other words, that the proof tree for $A'_P(F)$ is finite. While it is obvious for cases 1 and 2 (no term can be a proper subterm of itself in a finite term), the thesis follows in case 3 because of the constant guardedness condition over SPA agents. It guarantees that

```
Command: c_sbsnni  Access_Monitor_5
[\] Verifying AM | Interf
  [|] Verifying AM
    [\] Verifying Monitor_5 | Object_l0 | Object_h0
      [|] Verifying Monitor_5
      [|] Failed!
    [\] Failed! Verifying directly (Monitor_5 | Object_l0 | Object_h0)\L
  [|] Failed!
[\] Failed! Verifying directly (AM | Interf)\K
true
```

Table 5: Verification of *SBSNNI* on *Access_Monitor_5* with the compositional algorithm.

the recursive substitution of non prefixed constants with their definitions terminates. Hence the computation of $A'_P(F)$ ends.

To prove that the result of the compositional algorithm A'_P is consistent with the one obtained by the given algorithm A_P, we observe that the four rules above guarantee this, using compositionality properties for steps 1 and 2. ■

The theorem above requires that – in evaluating $A'_P(E)$ – if E is in the form $E' \setminus L$, then E' must be finite state. In fact, if we consider a finite state system $E \setminus L$ such that $E \notin \mathcal{E}_{FS}$, then $A_P(E \setminus L)$ terminates while $A'_P(E \setminus L)$ possibly do not, because it tries to compute $A_P(E)$ on the non-finite state agent E. The premise of the theorem above trivially holds for agents in the class of *nets of automata*.

The CoSeC command **c_sbsnni** checks the *SBSNNI* property exploiting compositionality. Let us now compare the compositional algorithm w.r.t. the normal one, starting from *Access_Monitor_5*. The normal verification of the *SBSNNI* property on such a system requires a lot of time (about 16 minutes [12] on a SUN5 workstation) because of the above mentioned exponential state explosion due to parallel composition. We could hope to get a better result using the compositional algorithm. Table 5 reports the output of the compositional verification of *Access_Monitor_5* where the symbols [\] and [|] represent steps 1 and 2 of the algorithm, respectively. This table shows that the algorithm fails in the verification of *SBSNNI* over *AM* and then succeeds in checking the system as a whole.[13] Hence, in this case, the compositional technique cannot help reducing the execution time.

However, we can modify *AM* in order to obtain a *SBSNNI* system by making (only!) high level communications asynchronous. This can be done adding a high level buffer between the monitor and the interface.

$$
\begin{array}{rcl}
Access\_Monitor\_6 & \stackrel{\mathrm{def}}{=} & (AM\_6 \mid Interf\_6) \setminus N \\
AM\_6 & \stackrel{\mathrm{def}}{=} & ((Monitor\_5 \mid Object(1,0) \mid Object(0,0) \mid hBuf(empty)) \setminus L)[res(0,y)/val(0,y)] \\
hBuf(j) & \stackrel{\mathrm{def}}{=} & \overline{res}(1,j).hBuf(empty) + val(1,k).hBuf(k) \\
Interf\_6 & \stackrel{\mathrm{def}}{=} & Interf\_6(0) \mid Interf\_6(1) \\
Interf\_6(l) & \stackrel{\mathrm{def}}{=} & a\_r(l,x).\overline{access\_r}(l,x).Interf\_6\_reply(l) \\
 & & + \\
 & & a\_w(l,x,z).\overline{access\_w}(l,x,z).Interf\_6(l) \\
Interf\_6\_reply(l) & \stackrel{\mathrm{def}}{=} & res(l,y). \\
 & & (\ \mathbf{if}\ y = empty\ \mathbf{then} \\
 & & \quad\quad Interf\_6\_reply(l) \\
 & & \ \ \mathbf{else} \\
 & & \quad\quad \overline{put}(l,y).Interf\_6(l))
\end{array}
$$

[12] This value and all the following are obtained exploiting also the state minimization feature of the tool.

[13] The reason why it fails on *AM* is because *AM* is essentially *Access_Monitor_1* with atomic write operations; hence, it is not *SBSNNI* because of the possible high level deadlock of Example 3.10 caused by synchronous communications. The interface was introduced just in order to make communications asynchronous.

```
Command: c_sbsnni  Access_Monitor_6
[\] Verifying AM_6 | Interf_6
  [|] Verifying AM_6
  [|] Verifying Interf_6
    [|] Verifying Interf_6_1
    [|] Verifying Interf_6_0
true
```

Table 6: Verification of *SBSNNI* on $Access\_Monitor\_6$ exploiting compositionality.

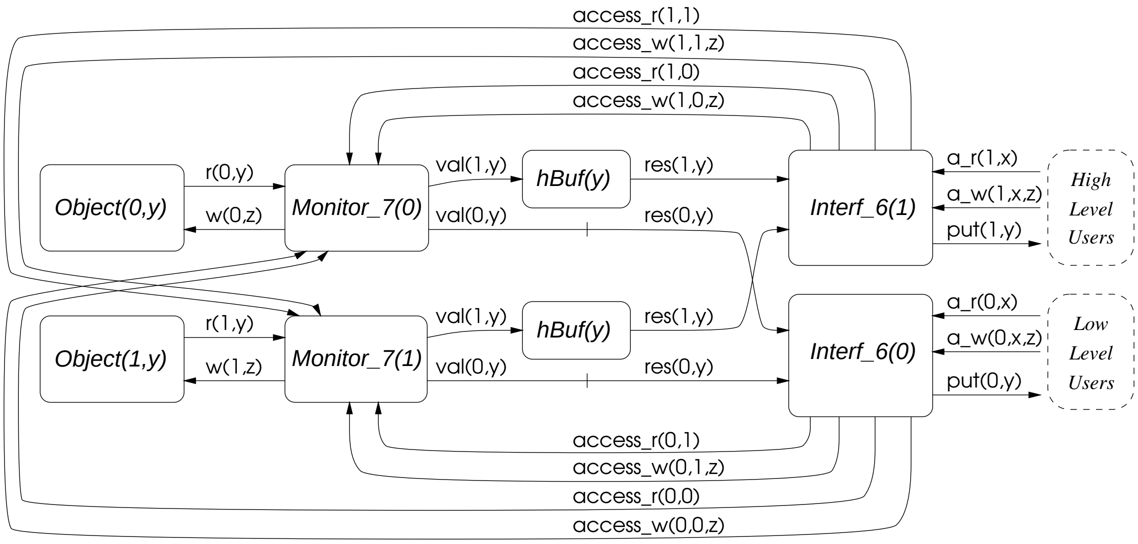

Figure 10: The Modular $Access\_Monitor\_7$

where $j \in \{0, 1, err, empty\}$, $L = \{r, w, val(1, y)\}$, $N = \{res, access\_r, access\_w\}$ and $res(1, y) \in Act_H, \forall y \in \{0, 1, err, empty\}$, while the same actions with 0 as first parameter belong to Act_L. Note that we have modified the interface so that it is now able to wait until the high buffer is filled by the monitor.

Using the compositional algorithm, system $Access\_Monitor\_6$ can be checked very efficiently; the verification of *SBSNNI* takes about 90 seconds (see Table 6). We can also check that $Access\_Monitor\_5 \approx_B Access\_Monitor\_6$; so, as expected, the introduction of the buffer does not modify the behaviour of the monitor. This verification requires about 2 minutes. Note that, by Theorem 3.13, we can conclude that also $Access\_Monitor\_5$ is *SBSNNI*, even if a direct check takes (as we said) about 16 minutes.

$Access\_Monitor\_6$ represents an example of successful application of the compositional checking; nonetheless, this does not mean that we cannot do better. Indeed, such a system is not defined in a very modular way and we hope that a more modular definition will lead to a more efficient compositional verification. In fact, suppose we want to add other objects to $Access\_Monitor\_6$; in such a case, the size of AM_6 will increase exponentially with respect to the number of added objects. Now we present rather a modular version of the access monitor. The basic idea of this new version (Figure 10) is that every object has a "private" monitor which implements the access functions for such (single) object. To make this, we have decomposed process $Monitor\_5$ into two different processes, one for each object; then we have composed such processes to their respective objects together with a high level buffer obtaining the *SBSNNI*-secure $Modh$ and $Modl$ agents. In particular, $Monitor\_7(x)$ handles the accesses to object x ($x = 0$ low, $x = 1$ high). As in $Access\_Monitor\_6$, we have an interface which guarantees the exclusive use of the monitor within the same level and is able to

```
Command: c_sbsnni Access_Monitor_7
[\] Verifying Modh | Modl | Interf_6
  [|] Verifying Modh
  [|] Verifying Modl
  [|] Verifying Interf_6
    [|] Verifying Interf_6_1
    [|] Verifying Interf_6_0
true
```

Table 7: Verification of *SBSNNI* on $Access\_Monitor\_7$ exploiting compositionality.

read values from the high buffer.

$$
\begin{array}{rcl}
Access\_Monitor\_7 & \stackrel{\mathrm{def}}{=} & (Modh \mid Modl \mid Interf\_6) \setminus L \\
Modh & \stackrel{\mathrm{def}}{=} & ((Monitor\_7(1) \mid Object(1,0) \mid hBuf(empty)) \setminus Lh)[res(0,y)/val(0,y)] \\
Modl & \stackrel{\mathrm{def}}{=} & ((Monitor\_7(0) \mid Object(0,0) \mid hBuf(empty)) \setminus Lh)[res(0,y)/val(0,y)] \\
Monitor\_7(x) & \stackrel{\mathrm{def}}{=} & access\_r(l,x). \\
 & & (\ \mathbf{if}\ x \leq l\ \mathbf{then} \\
 & & \qquad r(x,y).\overline{val}(l,y).Monitor\_7(x) \\
 & & \ \mathbf{else} \\
 & & \qquad \overline{val}(l,err).Monitor\_7(x)) \\
 & & + \\
 & & access\_w(l,x,z). \\
 & & (\ \mathbf{if}\ x \geq l\ \mathrm{then} \\
 & & \qquad \overline{w}(x,z).Monitor\_7(x) \\
 & & \ \mathbf{else} \\
 & & \qquad Monitor\_7(x))
\end{array}
$$

where $L = \{res, access\_r, access\_w\}$ and $Lh = \{r, w, val(1,y)\}$. Table 7 reports the output of the (successful) verification of the *SBSNNI* property for $Access\_Monitor\_7$. This task takes about 20 seconds on a SUN5 workstation, supporting our claim that a modular definition would help. Moreover, we can also check the new version of the monitor is functionally equivalent to the previous ones: in about 5 minutes, CoSeC checks that $Access\_Monitor\_7 \approx_B Access\_Monitor\_5$, and so also $Access\_Monitor\_7 \approx_B Access\_Monitor\_6$.

As a final remark, the compositional verification is more convenient only when building complex systems as parallel composition of simpler ones. For this reason, the tool offers to the user the choice between the normal and the compositional verification algorithms. It is up to the user to choose which one (s)he thinks could go better, or even to make them work in parallel.

8 An Application of NI to Protocol Verification

In this section we want to show how the techniques based on the idea of Non-Interference could be used to find some flaws in a simple key distribution protocol. The example is admittedly very simple and is introduced only to the aim of showing that the basic ideas presented so far can be applied in this field as well. What we do is to specify the protocol using SPA. We observe its behaviour by inserting some actions representing the start of the protocol and its correct termination. Then we specify an enemy which is able to intercept and create messages on the communication channel. The enemy behaviour can be observed through an action *fake* which is an abstraction of all the malicious activities on the channel. We can also observe if the enemy succeeds in obtaining the key which should be distributed in a secure manner by the protocol. The idea is to check if this action $fake$ can interfere with the correct termination of the protocol, i.e., if it can interfere with all the other observable actions. For instance if $fake$ can cause the interception of the key by the enemy the protocol will be clearly insecure. A feature of this method is that we do not specify particular behaviours or properties that the protocol should have to be secure, we simply say which are the actions that are interesting for the security of the protocol and we see if the enemy is able to modify their execution in some way. We think that this could be useful to discover new flaws in protocols, since we do not need to know exactly which kind of attacks on the protocol we are observing. This example is simple and we also have an action representing the interception of the key by the enemy, so we explicitly know when the enemy succeeds in the attack. However this is not required by the method and there are also cases where it is not possible to model the attacker' success explicitly with an action. An example of this is the Needham-Schroeder public-key protocol [20], studied in [7, 9]. Further details on this protocol analysis with NI based properties can be found in [7].

The problem is the following: There is a process KDC (Key Distribution Center) on a remote host which is devoted to the distribution of the public-keys. In particular, when a user A needs to send some secret information to another user B, A asks KDC for B's public key. Then, A can encode every secret message with this key and is guaranteed that only B will be able to read it. In particular, in the protocol we are going to analyze, A sends to B a session-key encoded with the public key of B. Such session key will be used for every further communication, until the two users decide to establish a new session with a new session-key.

The aim of the protocol is simply to distribute the session-key generated by A to B. Since A uses the public-key of B, (s)he is assured that only B will know the session-key and so there is a form of implicit one-side authentication of B for A. The opposite authentication is not valid since B has no guarantees about who (s)he is talking with. Anyone can encrypt a session-key with B's public-key and send it pretending to be A. So, B will presumably check the identity of A during the session (if this is required). We could imagine a situation of a remote login on a machine B from a user A. First the user establishes a session-key with the remote login server using the protocol above. Then the communication proceeds encrypted with the session-key and the server checks the identity of A using, for instance, some login/password mechanism. However A is assured to communicate with B since only B can have received the session-key.

In order to formalize the protocol we use the notation $A \stackrel{msg}{\longrightarrow} B$ representing the sending of message msg from A to B. The protocol is the following sequence of three message exchanges:

$$\begin{array}{ccc}
A & \stackrel{reqPK(A,B)}{\longrightarrow} & KDC \\
A & \stackrel{reply(A,PK_B)}{\longleftarrow} & KDC \\
A & \stackrel{exchange(A,B,encrypt(PK_B,K_{sess}))}{\longrightarrow} & B
\end{array}$$

where $reqPK(A,B)$ is the request, sent from A to KDC, of B's public key; $reply(A, PK_B)$ is the reply from KDC to A, containing the public key PK_B of B; finally, $encrypt(PK_B, K_{sess})$ represents the session key K_{sess} encoded using the public key PK_B and it is sent to B through message $exchange(A, B, encrypt(PK_B, K_{sess}))$.

We can write the protocol in VSPA as follows:

$$\begin{array}{rcl}
P & \stackrel{\text{def}}{=} & (A(K_{sess}, B) \mid B \mid KDC) \setminus \{reqPK, reply, exchange\} \\
A(K_{sess}, B) & \stackrel{\text{def}}{=} & \overline{start}(K_{sess}). \\
 & & \overline{reqPK}(A, B). \\
 & & reply(A, pk). \\
 & & \overline{exchange}(A, B, encrypt(pk, K_{sess})). \\
 & & \overline{commit}(A, B, K_{sess}).\underline{0} \\
B & \stackrel{\text{def}}{=} & exchange(user, B, encrypt(pk, K_{sess})). \\
 & & \textbf{if } pk = PK_B \textbf{ then} \\
 & & \quad \overline{commit}(B, user, K_{sess}).\underline{0} \\
KDC & \stackrel{\text{def}}{=} & reqPK(user_1, user_2). \\
 & & \overline{reply}(user_1, PK_{user_2}).
\end{array}$$

KDC

We have inserted two output actions which can be used to observe the current state of the protocol: $start(key)$ and $commit(user_1, user_2, key)$. The first one is used to see when the protocol starts up with session key key (see process A). The second one is used to observe when the protocol correctly stops for $user_1$, which will communicate with $user_2$ using key key. For A this happens when (s)he sends the session key to B encrypted with B's public key. For B it is sufficient to receive a session key from any user. The only condition is that the session key is encrypted with B's public key. This action can be seen as an abstraction of the communication session starting between $user_1$ and $user_2$.

In order to check if this protocol can be attacked, we specify an enemy process which can intercept and generate every message on the net provided that it does not decrypt and encrypt messages using passwords that it does not know. We also suppose that the enemy already knows all the public keys and so it is not necessary that it asks the KDC for them. Since the keys are public, this is not an unrealistic assumption, and it permits us to simplify the specification.

$$
\begin{aligned}
E \stackrel{\text{def}}{=} \; & \overline{fake}. \\
& \quad (\; reqPK(user_1, user_2).E \\
& \quad + reply(user_1, pk).E \\
& \quad + exchange(user_1, user_2, encrypt(pk, ks)). \\
& \qquad \textbf{if } pk = PK_E \textbf{ then } \overline{commit}(E, user_1, ks).E \\
& \qquad \textbf{else } E \\
& \quad + \sum_{\substack{u1,\, u2 \,\in\, U \\ u1 \,\neq\, u2}} \overline{reqPK}(u1, u2).E \\
& \quad + \sum_{\substack{u1 \,\in\, U \\ pk \,\in\, PK}} \overline{reply}(u1, pk).E \\
& \quad)
\end{aligned}
$$

where U is the set of users and PK is the set of public keys. In this example $U = \{A, B, E\}$ and $PK = \{PK_A, PK_B, PK_E\}$.

Process E can intercept $reqPK$, $reply$ and $exchange$ actions and can produce as output actions $reqPK$ and $reply$. We can observe if the enemy is doing something through the output action $fake$ which is executed at every enemy iteration. Moreover if the enemy intercepts an $exchange(user_1, user_2, encrypt(pk, ks))$ action containing a session key ks encrypted with its public key ($pk = PK_E$), then it can use that session key to intercept and decrypt all the future messages from $user_1$ to $user_2$. In this situation E could either pretend to be $user_2$, or forward the session key ks to $user_2$ through action $\overline{exchange}(user_1, user_2, encrypt(pk, ks))$ and then simply listen to the communications between the two users. We are not interested in what E will do with this session key, so we abstract all the malicious activities of E with $\overline{commit}(E, user_1, ks)$ which simply means that E shares ks with $user_1$.

In order to analyze the protocol we insert the enemy among A, B and KDC as follows:

$$
PE \stackrel{\text{def}}{=} (A(K_{sess}, B) \mid B \mid KDC \mid E) \setminus \{reqPK, reply, exchange\}
$$

Now PE has three external actions: $start$, $commit$ and $fake$. We want to observe how the activity of the enemy can interfere with the protocol. The trick is to consider $fake$ as a high level action and $start$ and $commit$ as low level ones. In this way we can check if the enemy activity (abstracted with $fake$) can interfere with the various *start-commit* sequences that represent the protocol sessions.

In particular we can try to check if PE is *SNNI* as a first step. The tool gives this answer:

```
false
```

```
Agent PE!fake
can perform action sequence 'start_Ksess,'commit_A_B_Ksess,'commit_E_A_Ksess
which agent PE\fake
cannot
```

so we have that when we hide $fake$ (i.e., when the enemy can try to break the protocol), the trace composed by the actions $\overline{start}(K_{sess}), \overline{commit}(A, B, K_{sess}), \overline{commit}(E, A, K_{sess})$ can be executed. This means that A starts the protocol and E succeeds in intercepting the session key. We have not a detailed trace of the enemy activity (this will be a future enhancement in the tool). However we can reason on the information we have in order to find the problem in the protocol. Since E executes $\overline{commit}(E, A, K_{sess})$ then it must have intercepted the action $\overline{exchange}(A, user_2, encrypt(PK_E, K_{sess}))$ from A. This means that A has received a $\overline{reply}(A, PK_E)$. Now we can observe that this action could be generated also by E and so we have found the attack: E intercepts the correct response $\overline{reply}(A, PK_B)$ of KDC, which is sending the public key of B to A (second message of the protocol), and replaces it with $\overline{reply}(A, PK_E)$. In this way A asks for public key of B, but the enemy substitutes it with its own public key. A cannot detect this. This is a typical example of the so-called *man-in-the-middle* attack; it can be fixed by encrypting the reply of the KDC with KDC's private key (digital signature). In this way everybody can read the KDC message, decrypting it with KDC public key, but no one can modify or substitute it since only KDC knows its own private key.

We change the second message obtaining a new version of the protocol:

$$\begin{array}{lcl} A & \xrightarrow{reqPK(A,B)} & KDC \\ A & \xleftarrow{reply(A,encrypt(SK_{KDC},PK_B)} & KDC \\ A & \xrightarrow{exchange(A,B,encrypt(PK_B,K_{sess}))} & B \end{array}$$

where SK_u represents the secret key for user u.

We can specify this in SPA and analyze it against the same enemy. What we find out is that also this second version of the protocol is not *SNNI*. The output of the tool is the same as in the previous case! So the enemy is still able to get the session key from A. We can understand this by going on with the above reasoning exactly where we stopped. Now the enemy cannot generate the message $\overline{reply}(A, encript(SK_{KDC}, PK_E))$ since it does not know SK_{KDC}. So this message must have been executed by KDC. This means that KDC has received a request $reqPK(A, E)$. But this action could have been generated by E. And so here we have the second attack: the enemy intercepts the request for A to KDC of the public key of B and substitutes it with a request $reqPK(A, E)$. In this way KDC believes that A is asking for E's public key and sends it to A. This attack is more subtle than the previous one: since the enemy cannot change the message containing the public key of B (it is signed), then it changes the request. This problem can also be fixed. One possibility is to add in the encrypted part of the reply from KDC also the identifier of the owner of the key sent. We can formalize it as follows (only the second message has been changed):

$$\begin{array}{lcl} A & \xrightarrow{reqPK(A,B)} & KDC \\ A & \xleftarrow{reply(A,encrypt(SK_{KDC},PK_B,B)} & KDC \\ A & \xrightarrow{exchange(A,B,encrypt(PK_B,K_{sess}))} & B \end{array}$$

In this way, if the enemy changes the request $reqPK(A, B)$ of A to $reqPK(A, E)$, then KDC will reply with $reply(A, encrypt(SK_{KDC}, PK_E, E))$ and A will discover that PK_E is the public key of E.

Again we can formalize the new version of the protocol in SPA, and this time we can successfully check that it is *SNNI*.

9 Related Work

The use of process algebras to formalize information flow security properties is not new. In [26] it is possible to find a definition of Non Interference given on CSP. It looks like *SNNI* with some side conditions on acceptable low level actions. This definition is recalled in [1], where a comparison with another information flow property is reported.

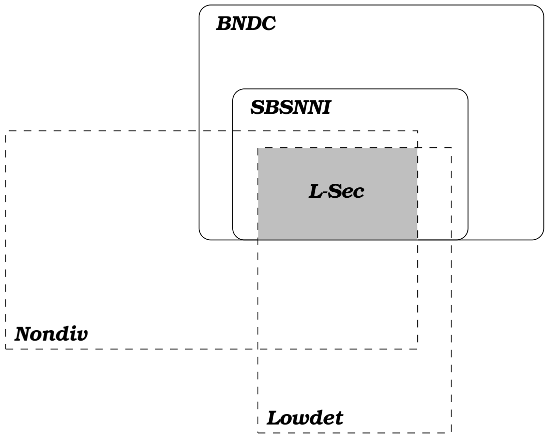

Figure 11: Relations between *L-Sec*, *SBSNNI* and *BNDC*

More recent results based on CSP model are contained in [25], where the authors introduce some information flow security properties based on the notion of *deterministic views* and show how to automatically verify them using the CSP model checker FDR [24]. The most interesting property is *lazy security* (*L-Sec*) which, however, requires the absence of non-determinism in the low view of the system (i.e., when hiding high actions through interleaving) and for this reason we think it could be too restrictive in a concurrent environment. For example, all the low non-deterministic systems – such as $E = l.\bar{l_1} + l.\bar{l_2}$ – are considered not to be secure. In [6] one of the authors has compared those properties with ours using a failure-equivalence version of *BNDC*, called *FNDC*. The main result of the paper is that *BNDC* restricted to the class of low-deterministic and non-divergent processes is equal to *L-Sec*. In order to better relate this discussion to the one in Section 3.2 and Section 3.4 about divergences and failure/testing semantics, note that since the comparison is restricted to the class of non-divergent agents we could use also a testing equivalence based *NDC* instead of *FNDC*. In fact, it is easy to see that, on non-divergent agents, failure and testing equivalences coincide.

In the following we denote with $E|||F$ the interleaving without communication between agents E and F. It can be expressed in SPA as $(E[A/\mathcal{L}(E)] \mid F[B/\mathcal{L}(F)])[\mathcal{L}(E)/A, \mathcal{L}(F)/B]$ where $A, B \subseteq \mathcal{L}, A \cap B = \emptyset$ and $A/\mathcal{L}(E)$ is a bijective function which maps all the actions executable by E into actions of A, with $\mathcal{L}(E)/A$ as inverse (the same holds for $B/\mathcal{L}(F)$ and $\mathcal{L}(F)/B$). Let $RUN_H \stackrel{\mathrm{def}}{=} \sum_{h \in Act_H} h.RUN_H$ be the agent which can execute all the high level actions. In $E|||RUN_H$ an external observer is not able to tell if a certain high level action cames from E or from RUN_H (we say that we have an "obscuring of high level actions by interleaving"). Intuitively, *L-Sec* requires that the obscuring of high level actions by interleaving does not introduce any non-determinism in the system.

To give a formal definition of *L-Sec* we will denote with *Det* the set of all deterministic agents in the failure setting. In particular an agent is in *Det* if and only if whenever it can move to two different agents E' and E'' executing a certain trace γ, such agents are failure equivalent. Moreover *Nondiv* is the set of all the agents which do not contain a divergence. *L-Sec* is re-phrased on SPA as follows:

Definition 9.1 $E \in \textit{L-Sec} \Leftrightarrow E|||RUN_H \in \textit{Det} \cap \textit{Nondiv}$. ■

Low-determinism can be seen as determinism over low level actions and can be defined as follows:

Definition 9.2 *E is low-deterministic ($E \in$ Lowdet) if and only if $E \setminus Act_H \in$ Det.* ■

The main result, proved in [6], is the following:

Theorem 9.3 $\textit{BNDC} \cap \textit{Lowdet} \cap \textit{Nondiv} = \textit{SBSNNI} \cap \textit{Lowdet} \cap \textit{Nondiv} = \textit{L-Sec}$. ■

A diagram summarizing the results is given in Figure 11.

The theorem above implies that, for low-deterministic and non-divergent processes, we can use CoSeC to verify *L-Sec*. Moreover, we can see *BNDC* applied to low-deterministic processes as a fair extension of *L-Sec* to divergent processes.

10 Conclusion

Among the possible extensions of the tool, we would like to mention the following:

(*i*) Extension to value-passing SPA: it would be useful to make CoSeC accept specification written directly in VSPA. Moreover, one could exploit interesting techniques, such as the one of [12], that could be used to verify the equivalence of finite state value-passing SPA agents which have infinite state SPA translations.

(*ii*) It could be interesting to exploit the results we have presented in Section 3.4 about decidability of weak bisimulation on BPA agents [13]. This would allow to check security properties over infinite state systems, so allowing the specification of unbounded data structures.

(*iii*) Integration with other tools: CoSeC is essentially built on top of CW, but it is not difficult to imagine how to build similar routines on top of any of the available tools for verification of systems represented as labelled transition systems.

(*iv*) Addition of new information flow security properties: Even if we think *BNDC* is the right one, any new information flow property that has an algebraic characterization of the form seen in the Introduction can be included easily in CoSeC.

(*v*) A graphic interface: it should help the user when working with CoSeC, avoiding the need of memorizing the technical names.

From a theoretical point of view, it would be nice to find a static, easily verifiable characterization of *BNDC*, instead of resorting – as we did here – to the stronger *SBSNNI*. There is also an increasing interest in security for mobile processes and in [9] we briefly discuss the possibility of changing the underlying model of the developed theory, in order to deal with them. In our opinion, the whole research about SPA-based information flow properties, that we are presently carrying on, could be easily extended to a model which includes mobility (e.g. π-calculus [19]).

Acknowledgements

We would like to thank the anonymous referees for helpful comments and suggestions.

References

[1] P. G. Allen. "A Comparison of Non-Interference and Non-Deducibility using CSP". In *Proceedings of the Fourth IEEE Computer Security Foundations Workshop*, pages 43–54, Franconia, New Hampshire, June 1991.

[2] D. E. Bell and L. J. La Padula. "Secure Computer Systems: Unified Exposition and Multics Interpretation". *ESD-TR-75-306, MITRE MTR-2997*, March 1976.

[3] J. A. Bergstra and J. W. Klop. "Algebra of Communicating Processes with Abstraction". *Theoretical Computer Science*, 37:77–121, 1985.

[4] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. "A Theory of Communicating Sequential Processes". *Journal of the Association for Computing Machinery*, 31(3):560–599, July 1984.

[5] R. Cleaveland, J. Parrow, and B. Steffen. "The Concurrency Workbench: a Semantics Based Tool for the Verification of Concurrent Systems". *ACM Transactions on Programming Languages and Systems*, Vol. 15 No. 1:36–72, January 1993.

[6] R. Focardi. "Comparing Two Information Flow Security Properties". In *Proceedings of Ninth IEEE Computer Security Foundation Workshop, (CSFW'96), (M. Merritt Ed.)*, pages 116–122. IEEE press, June 1996.

[7] R. Focardi, A. Ghelli, and R. Gorrieri. "Using Non-Interference for the Analysis of Security Protocols". Forthcoming.

[8] R. Focardi and R. Gorrieri. "A Classification of Security Properties for Process Algebras". *Journal of Computer Security*, 3(1):5–33, 1994/1995.

[9] R. Focardi and R. Gorrieri. "Non Interference: Past, Present and Future". In *Proceedings of 1997 Foundations for Secure Mobile Code Workshop*, pages 33–36. Sponsored by DARPA, March 1997.

[10] R. Focardi, R. Gorrieri, and R. Segala. "A Concrete Criterion for Information Flow Security". Forthcoming.

[11] J. A. Goguen and J. Meseguer. "Security Policy and Security Models". In *Proceedings of the 1982 Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, April 1982.

[12] M. Hennessy and H. Lin. "Symbolic Bisimulations". Technical Report CSR1/92, University of Sussex, 1992.

[13] Y. Hirshfeld. "Bisimulation Trees and the Decidability of Weak Bisimulations". Technical report, Tel Aviv University, 1996.

[14] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[15] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation.*, pages 22–24. Addison-Wesley, 1979.

[16] P. Kanellakis and S.A. Smolka. "CCS Expressions, Finite State Processes, and Three Problems of Equivalence". *Information & Computation 86*, pages 43–68, May 1990.

[17] R. Keller. "Formal Verification of Parallel Programs". *Communications of the ACM*, 19 (7):561–572, 1976.

[18] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[19] R. Milner, J. Parrow, and D. Walker. "A Calculus of Mobile Processes". *Information & Computation 100*, pages 1–77, 1992.

[20] R. Needham and M. Schroeder. "Using encryption for authentication in large networks of computers". *Communications of the ACM*, 21(12):993–999, 1978.

[21] R. De Nicola and M. Hennessy. "Testing equivalences for processes". *Theoretical Computer Science*, 34:83–133, 1984.

[22] Department of Defense. "Integrity in Automated Information Systems". Technical Report C 79-91, National Computer Security Center, September 1991.

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

[24] A. W. Roscoe. "Model Checking CSP". In A. W. Roscoe (ed) *A Classical Mind.* Prentice Hall, 1994.

[25] A. W. Roscoe, J. C. P. Woodcock, and L. Wulf. "Non-interference through Determinism". In *Proceeding of European Symposium on Research in Computer Security 1994 (ESORICS'94)*, pages 33–53. Springer-Verlag LNCS 875, 1994.

[26] P. Y. A. Ryan. "A CSP Formulation of Non-Interference". In *Proceedings of the 1990 Computer Security Foundation Workshop III*, Franconia, 1990. IEEE press.

[27] L. J. Stockmeyer and A. R. Meyer. "Word problems requiring exponential time". In *Proceedings of the 5th ACM Symposium on Theory of Computing*, pages 1–9, Austin, Texas, 1973.

[28] C. R. Tsai, V. D. Gligor, and C. S. Chandersekaran. "On the Identification of Covert Storage Channels in Secure Systems". *IEEE Transactions on Software Engineering*, pages 569–580, June 1990.

[29] J. T. Wittbold and D. M. Johnson. "Information Flow in Nondeterministic Systems". In *Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy*, pages 144–161. IEEE Computer Society Press, 1990.

A Proofs

Proposition 3.12 *SBSNNI* $\subset$ *BNDC*

PROOF. Let E be a system and Π a high level process. Let R be a relation defined as follows: $(E' \setminus Act_H, (E' \mid \Pi') \setminus Act_H) \in R$, for all E', Π' such that $E \Rightarrow E'$ and $\Pi \Rightarrow \Pi'$. We want to prove that if E is *SBSNNI* then R is a weak bisimulation up to $\approx_B$ (see [18]). There is only one non trivial case: $(E' \mid \Pi') \setminus Act_H \xrightarrow{\tau} (E'' \mid \Pi'') \setminus Act_H$. As there exists $h \in Act_H$ such that $E' \xrightarrow{h} E''$, then $E'/Act_H \xrightarrow{\tau} E''/Act_H$. By hypothesis, $E' \in BSNNI$ hence we have $E'/Act_H \approx_B E' \setminus Act_H$ and so there exists an agent E''' such that $E' \setminus Act_H \stackrel{\hat{\tau}}{\Longrightarrow} E''' \setminus Act_H$ and $E''' \setminus Act_H \approx_B E''/Act_H$. By hypothesis, $E'' \in BSNNI$ hence we also have that $E''/Act_H \approx_B E'' \setminus Act_H$ and so $E''' \setminus Act_H \approx_B E'' \setminus Act_H$. Since $(E'' \setminus Act_H, (E'' \mid \Pi'') \setminus Act_H) \in R$ and $E''' \setminus Act_H \approx_B E'' \setminus Act_H$ then R is a bisimulation up to $\approx_B$.

Now we have that $E \setminus Act_H \approx_B (E \mid \Pi) \setminus Act_H$ for all $\Pi \in \mathcal{E}_H$. Since $E \in BSNNI$, then $E/Act_H \approx_B E \setminus Act_H$. Therefore $E/Act_H \approx_B (E \mid \Pi) \setminus Act_H$. This means that $E \in BNDC$.

The inclusion is strict because agent $E = l.h.l.0 + l.0 + l.l.0$ is *BNDC* but not *SBSNNI*. ∎

Theorem 3.13 *If $E \approx_B F$, then $E \in X \Leftrightarrow F \in X$, where X can be NNI, SNNI, NDC, BNNI, BSNNI, BNDC, SBSNNI.*

PROOF. It derives from the definition of the security properties, observing that trace and bisimulation equivalences are congruences with respect to $\_ \setminus L$ and $\_|\_$ operators of CCS [18]. It is possible to prove that they are also congruence with with respect to $\_ \setminus_I L$ and $\_/L$ operators of SPA. For trace equivalence the proof is trivial. In the following we prove the closure of weak bisimulation equivalence w.r.t. $\_ \setminus_I L$ and leave to the reader the other similar case of $\_/L$. Let $E \approx_B F$; we want to prove that $E \setminus_I L \approx_B F \setminus_I L$. Consider a bisimulation R such that $(E, F) \in R$; then define the relation P as follows: $(E' \setminus_I L, F' \setminus_I L) \in P$ if and only if $(E', F') \in R$. P is a bisimulation too, in fact if $E' \setminus_I L \xrightarrow{\mu} E'' \setminus_I L$ then $\mu \notin L \cap I$ and $E' \xrightarrow{\mu} E''$. So $\exists F''$ such that $F' \stackrel{\hat{\mu}}{\Longrightarrow} F''$; since $\mu \notin L \cap I$ then $F' \setminus_I L \stackrel{\hat{\mu}}{\Longrightarrow} F'' \setminus_I L$ (and vice versa for $F' \setminus_I L \xrightarrow{\mu} F'' \setminus_I L$). ∎

Theorem 7.1 *The following implications hold:*

(*i*) $E, F \in SBSNNI \Longrightarrow (E|F) \in SBSNNI$

(*ii*) $E \in SBSNNI, L \subseteq \mathcal{L} \Longrightarrow E \setminus L \in SBSNNI$

PROOF. We need the following:

Lemma A.1 $(E|F)/Act_H \approx_B E/Act_H|F/Act_H$.

PROOF. Consider the relation defined as follows: $((E'|F')/Act_H, E'/Act_H|F'/Act_H) \in R$ where $E \Rightarrow E'$ and $F \Rightarrow F'$. It is easy to prove that R is a bisimulation. Indeed the only non trivial case is the synchronization $(E'|F')/Act_H \xrightarrow{\tau} (E''|F'')/Act_H$ with the corresponding move $E'/Act_H|F'/Act_H \stackrel{\tau\cdot\tau}{\longrightarrow} E''/Act_H|F''/Act_H$. ∎

Now we can prove the Theorem.

(*i*) Consider the relation $((E'|F') \setminus Act_H, E' \setminus Act_H|F' \setminus Act_H) \in R$ for all E', F' such that $E \Rightarrow E'$ and $F \Rightarrow F'$. If we prove that R is a weak bisimulation up to $\approx_B$ then, by hypothesis and Lemma A.1, we obtain the thesis. We consider the only non trivial case: $(E'|F') \setminus Act_H \xrightarrow{\tau} (E''|F'') \setminus Act_H$ with $E' \xrightarrow{h} E''$ and $F' \xrightarrow{\bar{h}} F''$. Since $E'/Act_H \xrightarrow{\tau} E''/Act_H$ and, by hypothesis, $E' \in BSNNI$, we have that $\exists E'''$ such that $E' \setminus Act_H \stackrel{\hat{\tau}}{\Longrightarrow} E''' \setminus Act_H$ and $E''/Act_H \approx_B E''' \setminus Act_H$; finally, by hypothesis, $E'' \in BSNNI$, hence we obtain $E''' \setminus Act_H \approx_B E'' \setminus Act_H$. Repeating the same procedure for F' we have $\exists E''', F'''$ such that $E' \setminus Act_H | F' \setminus Act_H \stackrel{\hat{\tau}}{\Longrightarrow} E''' \setminus Act_H | F''' \setminus Act_H \approx_B E'' \setminus Act_H | F'' \setminus Act_H$. Since $((E''|F'') \setminus Act_H, E'' \setminus Act_H | F'' \setminus Act_H) \in R$, then R is a bisimulation up to $\approx_B$.

(ii) Consider the following relation $((E'/Act_H)\setminus L, (E'\setminus L)/Act_H) \in R$, for all E' such that $E \Rightarrow E'$ and for all $L \subseteq \mathcal{L}$. If we prove that R is a bisimulation up to $\approx_B$ then, by applying hypothesis and observing that $(E'\setminus L)\setminus Act_H \approx_B (E'\setminus Act_H)\setminus L$, we obtain the thesis. The only non trivial case is $(E'/Act_H)\setminus L \stackrel{\tau}{\rightarrow} (E''/Act_H)\setminus L$ with $E' \stackrel{h}{\rightarrow} E''$ and $h \in Act_H$. By hypothesis, $E' \in BSNNI$ hence we have that $(E'/Act_H)\setminus L \approx_B (E'\setminus Act_H)\setminus L$ and so $\exists E'''$ such that $(E'\setminus Act_H)\setminus L \stackrel{\hat{\tau}}{\Longrightarrow} (E'''\setminus Act_H)\setminus L \approx_B (E'''/Act_H)\setminus L$ and $(E'''\setminus Act_H)\setminus L \approx_B (E''/Act_H)\setminus L$. Obviously, we also have that $(E' \setminus L) \setminus Act_H \stackrel{\hat{\tau}}{\Longrightarrow} (E''' \setminus L) \setminus Act_H$ and so $(E' \setminus L)/Act_H \stackrel{\hat{\tau}}{\Longrightarrow} (E''' \setminus L)/Act_H$. We briefly summarize the proof: we had the synchronization $(E'/Act_H) \setminus L \stackrel{\tau}{\rightarrow} (E''/Act_H) \setminus L$ and we proved that there exists E''' such that $(E' \setminus L)/Act_H \stackrel{\hat{\tau}}{\Longrightarrow} (E''' \setminus L)/Act_H$ and $(E'''/Act_H) \setminus L \approx_B (E''/Act_H) \setminus L$. Since $((E'''/Act_H) \setminus L, (E''' \setminus L)/Act_H) \in R$ then R is a weak bisimulation up to $\approx_B$. ∎