Automatic Compositional Verification of Some Security Properties

R. Focardi*
R. Gorrieri*

ABSTRACT [1] The Compositional Security Checker (CSC for short) is a semantic tool for the automatic verification of some compositional information flow properties. The specifications given as inputs to CSC are terms of the Security Process Algebra, a language suited for the specification of systems where actions belong to two different levels of confidentiality. The information flow security properties which can be verified by CSC are some of those classified in [4]. They are derivations of some classic notions, e.g. Non Interference [6]. The tool is based on the same architecture of the Concurrency Workbench [2], from which some modules have been integrally imported. The usefulness of the tool is tested with the significative case-study of an access-monitor.

1 Introduction

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

In the *Discretionary Access Control* security (DAC for short), every *subject* decides the access properties of its *objects*. An example of DAC is the file management in Unix where a user can decide the access possibilities of her/his files. So, if a user executes a Trojan Horse program, this can modify the security properties of user's objects.

A solution to this problem is the *Mandatory Access Control* (MAC for short), where access rules are imposed by the system. An example of MAC is *Multilevel Security* [1]: every object is bounded to a security level, and so every subject is; information can flow from a certain object to a certain

*Dipartimento di Scienze dell'Informazione, Università di Bologna, Piazza di Porta San Donato 5, I – 40127 Bologna (Italy). e_mail:{focardi,gorrieri}@cs.unibo.it
[1]Research supported in part by CNR and MURST.

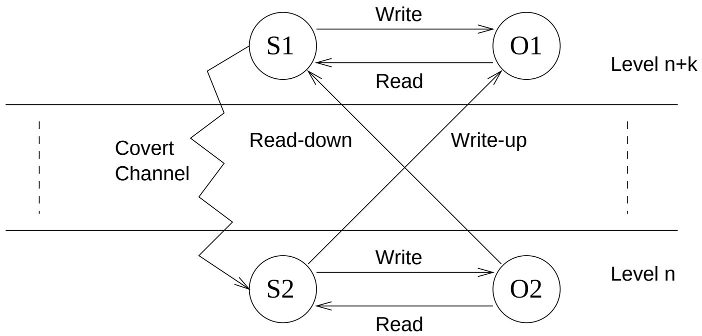

FIGURE 1. Information flows in multilevel security.

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 inside such a 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 to a lower level object).

However, these access rules are not enough. It could be possible to indirectly transmit information using some system *side effect*. For example, if two levels – 'high' and 'low' – share some finite storage resource, 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 that such indirect ways of transmission, 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. [10]).

An alternative, more general approach to security requires to control directly 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 rule, we can control any kind of transmission, be it direct or indirect.

In the literature, there are many different definitions of this kind based on several system models (see e.g. [6, 11]). In [4] we have rephrased them in the uniform setting of *Security Process Algebra* (SPA, for short), then compared and classified. SPA is an extension of CCS [9] which permits to describe systems where actions belong to two different levels of confidentiality.

For some of the investigated information flow properties, we 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 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) [2] is one of the most famous.

The 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 of CW (Version 6.1), from which some modules have been integrally imported and some others only modified.

The tool is equipped with a parser, which transforms an SPA specification into a parse-tree; then, for the parsed specification CSC builds the labelled transition system following the operational rules defined in Plotkin' SOS style. When a user wants to check if an SPA process E is X-secure, CSC first provides operational semantic descriptions to the terms $\mathcal{C}_X[E]$ and $\mathcal{D}_X[E]$ in the form of two lts's; then verifies the semantic equivalence of $\mathcal{C}_X[E]$ and $\mathcal{D}_X[E]$ using their lts representations.

An interesting feature of CSC is the exploitation of the compositionality of some security property. The main problem in the verification of security properties is the exponential state explosion due to parallel composition. As an example consider two agents E_1 and E_2; the number of states of their parallel composition $E_1|E_2$ is equal to the product of the states of E_1 and E_2. Now if we have a compositional security property X, i.e. such that $F_1|F_2$ is X-secure whenever F_1 and F_2 are X-secure, then we can apply the following strategy: check the X-security of E_1 and E_2; if it is satisfied conclude that $E_1|E_2$ is X-secure, otherwise check the X-security of the whole agent $E_1|E_2$. Using this strategy for compositional security properties, CSC is able to avoid, in some cases, the exponential state explosion due to parallel composition operator.

The paper is organized as follows. In Section 2 we present SPA. In Section 3 we recall from [4] some of the security properties which are verified by CSC giving some examples. Section 4.1 reports the input-output behavior of CSC, while in Section 4.2 we describe 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 a case-study (access-monitor). Finally, Section 7 is about the state explosion problem and the compositional algorithm.

2 SPA and Semantic Equivalences

In the following, systems will be specified using the *Security Process Algebra* (SPA for short), a slight extension of Milner's CCS [9]. SPA includes two more operators, namely the hiding operator E/L of CSP [7] and the (new) input restriction operator $E \setminus_I L$, which are useful in characterizing some security properties in an algebraic style. Intuitively $E \setminus_I L$ can execute all the actions process E is able to do, provided that they are not inputs in L. Moreover 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, a set $\mathcal{L} = I \cup O$ of *visible* actions, (ranged over by α) and the usual function $\bar{\cdot} : \mathcal{L} \to \mathcal{L}$ such that $a \in I \Longrightarrow \bar{a} \in O$ and $\bar{a} \in O \Longrightarrow \bar{\bar{a}} = a \in I$; two sets Act_H and Act_L of high and low level actions such that $\overline{Act_H} = Act_H$, $\overline{Act_L} = Act_L$, $Act_H \cup Act_L = \mathcal{L}$ and $Act_H \cap Act_L = \emptyset$ where $\bar{L} \stackrel{\text{def}}{=} \{\bar{a} : a \in L\}$; 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* is defined as follows:

$$E ::= \underline{0} \mid \mu.E \mid E + E \mid E|E \mid E \setminus L \mid E \setminus_I L \mid E/L \mid E[f] \mid Z$$

where $L \subseteq \mathcal{L}$ and $f : Act \to Act$ is such that $f(\bar{\alpha}) = \overline{f(\alpha)}$, $f(\tau) = \tau$. Moreover, for every constant Z there must be the corresponding definition: $Z \stackrel{\text{def}}{=} E$. The meaning of $\underline{0}$, $\mu.E$, $E + E$, $E|E$, $E \setminus L$, $E[f]$ and $Z \stackrel{\text{def}}{=} E$ is as for CCS [9].

Let $\mathcal{E}$ be the set of *closed* and *guarded* [9] SPA agents, ranged over by E, 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 $\mathcal{E}_H \stackrel{\text{def}}{=} \{E \in \mathcal{E} \mid \mathcal{L}(E) \subseteq Act_H \cup \{\tau\}\}$ and $\mathcal{E}_L \stackrel{\text{def}}{=} \{E \in \mathcal{E} \mid \mathcal{L}(E) \subseteq Act_L \cup \{\tau\}\}$, respectively. The operational semantics of SPA is given (as usual) associating to each agent a particular state of the labelled transition system $(\mathcal{E}, Act, \to)$ where $\to \subseteq \mathcal{E} \times Act \times \mathcal{E}$ and, intuitively, $E \xrightarrow{\mu} E'$ means that agent E can execute μ moving to E'.

We recall here the definition of weak bisimulation [9] over SPA agents. In the following the expression $E \stackrel{\alpha}{\Longrightarrow} E'$ is a shorthand for $E(\xrightarrow{\tau})^* E_1 \xrightarrow{\alpha} E_2(\xrightarrow{\tau})^* E'$, where $(\xrightarrow{\tau})^*$ denotes a (possibly empty) sequence of τ labelled transitions. Moreover $E \stackrel{\hat{\mu}}{\Longrightarrow} E'$ stands for $E \stackrel{\mu}{\Longrightarrow} E'$ if $\mu \in \mathcal{L}$, and for $E\ (\xrightarrow{\tau})^*\ E'$ if $\mu = \tau$ (note that $(\xrightarrow{\tau})^*$ means "zero or more τ labelled transitions" while $\stackrel{\tau}{\Longrightarrow}$ requires at least one τ labelled transition).

---

[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.

Definition 2.1 *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* $E \xrightarrow{\mu} E'$ *then there exists* $F' \in \mathcal{E}$ *such that* $F \stackrel{\hat{\mu}}{\Longrightarrow} F'$ *and* $(E', F') \in R$;
- *conversely, whenever* $F \xrightarrow{\mu} F'$ *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 observational equivalent, notation* $E \approx_B F$, *if there exists a weak bisimulation containing the pair* (E, F). *Note that* $\approx_B$ *is an equivalence relation.* ■

Now we present a value-passing extension of SPA (VSPA, for short). All the examples contained in this paper will be done using such value passing calculus, because it originates more readable specifications than those written in pure SPA. As in [9], the semantics of the value-passing calculus can be given via translation into the pure calculus [3].

The syntax of VSPA agents is defined as follows:

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

where the variables $x_1, \ldots, x_n$ and 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. An example of VSPA agent specification follows.

Example 2.2 Consider the following system specified using VSPA:

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

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 others actions are low level ones. The process $Access\_Monitor\_1$ (Figure 2) handles read and write requests from high and low level users on two binary variables: a high level 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

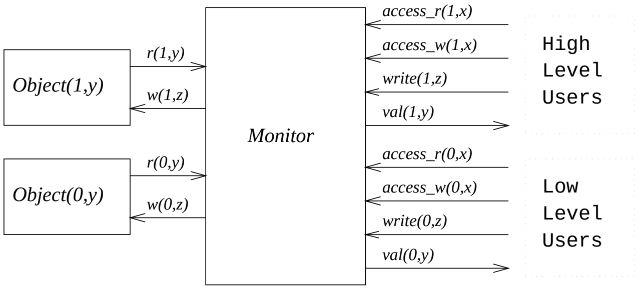

FIGURE 2. The Access Monitor for Example 2.2.

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 $val(l,y)$. ■

3 Some Information Flow Properties

In this section we present some of the security properties (see [3, 4] for more details) which can be verified using CSC.

Bisimulation Non-deterministic Non Interference (*BNNI*, for short) is a generalization of *Non Interference* [6]. Intuitively, the high level does not interfere with the low level if and only if a low level user cannot distinguish between processes E and $E \setminus_I Act_H$. In other words a system is *BNNI* if what a low level user sees of the system cannot be modified by any high level input.

Definition 3.1 $E \in BNNI \Leftrightarrow (E \setminus_I Act_H)/Act_H \approx_B E/Act_H$ ■

Example 3.2 Consider the following modified monitor [3] which does not control write accesses:

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

[3]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 examples.

Now it is possible for a high level user to write down (actions $access\_w(1,0)$ and $access\_w(1,1)$) so the system is not secure. In fact, $Access\_Monitor\_2$ is not *BNNI* as it 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 2 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

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

that, clearly, can not be a trace for $Access\_Monitor\_2$. In fact, in γ', we have that a low level user reads 1 from the low level object without other interactions between the monitor and the environment (note that the initial values of the objects is 0). Moreover it is not possible to obtain a trace for $Access\_Monitor\_2$ 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 a trace for $(Access\_Monitor\_2 \setminus_I Act_H)/Act_H$ too. In other words, it is not possible to find a trace γ'' 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 *BNNI*.

Hence, *BNNI* is able to detect if high level inputs interfere with low level executions, i.e. if a low level user can deduce something about high level inputs by observing only low level actions. ■

In [4] we proposed a more intuitive notion of information flow security: *Bisimulation Non Deducibility on Compositions* (*BNDC*, for short). A system E is *BNDC* if for every high level process Π a low level user cannot distinguish between processes E and $(E|\Pi) \setminus Act_H$. 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.

Definition 3.3 $E \in BNDC \Leftrightarrow \forall \Pi \in \mathcal{E}_H,\ E/Act_H \approx_B (E \mid \Pi) \setminus Act_H$. ■

A static characterization of *BNDC* – which does not involve composition with every processes Π – is not immediate. As a matter of fact, this problem is still open. In [5] we proposed the *SBSNNI* property which is static, compositional (i.e. if two systems are *SBSNNI* their composition is *SBSNNI*, too) and strictly stronger than *BNDC*. We first define *Bisimulation Strong Non-deterministic Non Interference* (*BSNNI*, for short), a property which differs from *BNNI* only because it restricts system E over all the high level actions rather than only over high level inputs.

Definition 3.4 $E \in BSNNI \Leftrightarrow E/Act_H \approx_B E \setminus Act_H$. ■

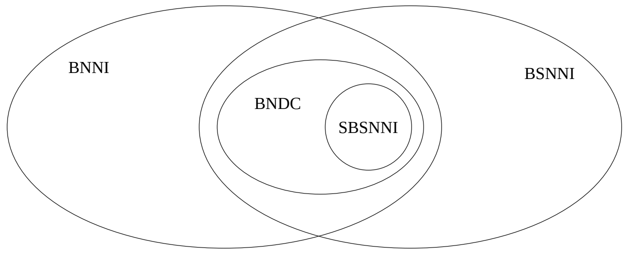

FIGURE 3. The inclusion diagram for bisimulation-based properties

Now we can define *Strong BSNNI* (*SBSNNI*, for short).

Definition 3.5 *A system $E \in SBSNNI$ if and only if for all E' reachable from E we have $E' \in BSNNI$.* ■

The following holds [5]:

Proposition 3.6 *$SBSNNI \subset BNDC$* ■

In the automatic verification of security properties it can be very useful to work on a *reduced* system, i.e. a system equivalent to the original one, but with a minimum number of states. The Concurrency Workbench provides a procedure to this aim and we imported it in CSC. This is very useful because we can prove that if a system E is *BNDC*, then any other observational equivalent system F is *BNDC*. This also holds for all the other security properties.

Theorem 3.7 *If $E \approx_B F$, then $E \in X \Leftrightarrow F \in X$, where X can be BNNI, BSNNI, BNDC, SBSNNI.* ■

Figure 3 summarizes the relations among the security properties defined above.

The following example shows that *BSNNI* and *BNNI* are not able to detect some deadlocks due to high level activities which, on the contrary, are revealed by *BNDC* (this because they do not check the system against all the possible high level interactions, as *BNDC* does).

Example 3.8 Consider the first version of the monitor $Access\_Monitor\_1$. Using CSC we can verify that such system is *BSNNI* and *BNNI*, but it is not *BNDC*. This happens because a high level user can make a read request without accepting the corresponding output from the monitor (remember that communications in SPA are synchronous). In particular, consider $\Pi = \overline{access\_r}(1,1).\underline{0}$. System $(Access\_Monitor\_1|\Pi) \setminus Act_H$ will be deadlocked immediately after the execution of the read request by Π, blocking in the following 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 process $Access\_Monitor\_1/Act_H$, we find out that $(Access\_Monitor\_1|\Pi) \setminus Act_H \not\approx_B Access\_Monitor\_1/Act_H$, so $Access\_Monitor\_1$ is not *BNDC*.

Moreover, there is another possible deadlock due to high level activity; this happens if a high level user makes a write request and do not send the value to be written. In particular, if we consider the high level user $\Pi' = \overline{access\_w}(1,0).\underline{0}$, it will deadlock system $(Access\_Monitor\_1|\Pi') \setminus Act_H$ immediately after the execution of the write request by Π', blocking in the following state:

$$\begin{array}{l}(((write(1,0).Monitor + write(1,1).Monitor) \mid Object(0,0) \mid \\ \mid Object(1,0)) \setminus L \mid 0) \setminus Act_H\end{array}$$

This happens because Π' executes a write request and does not send the corresponding value through action $write(1,0)$ or $write(1,1)$. 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* system, we modify the monitor by adding an output buffer for each level (this makes communications asynchronous) and using an atomic action for write request and value sending. The resulting system follows:

$$\begin{array}{rcl}Access\_Monitor\_3 & \stackrel{\text{def}}{=} & (Monitor|Object(1,0)|Object(0,0)|Buf(1,empty)| \\ & & |Buf(0,empty)) \setminus L \\ Monitor & \stackrel{\text{def}}{=} & access\_r(l,x).(\text{if } x \leq l \text{ then } r(x,y).\overline{val}(l,y).Monitor \\ & & \ \text{else } \overline{val}(l,err).Monitor) + \\ & & \ + access\_w(l,x,z).(\text{ f } x \geq l \text{ then } \overline{w}(x,z).Monitor \\ & & \ \text{else } Monitor) \\ Buf(x,j) & \stackrel{\text{def}}{=} & \overline{res}(x,j).Buf(x,empty) + val(x,k).Buf(x,k)\end{array}$$

where $k \in \{0,1,err\}$ and $j \in \{0,1,err,empty\}$; $L = \{r,\ w,\ val\}$. Moreover output actions $res(x,j)$ of buffer substitute output actions $val(x,k)$ in the interactions with users, with $res(1,i) \in Act_H, \forall i \in \{0,1,err,empty\}$.

Using CSC it is possible to automatically verify that $Access\_Monitor\_3$ is *SBSNNI* and so *BNDC*. ■

4 What is the Compositional Security Checker

4.1 Input-Output

The inputs of CSC are concurrent systems expressed as SPA agents. The outputs are answers to questions like: "does this system satisfy that specific

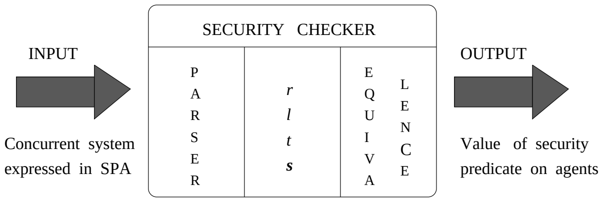

FIGURE 4. Structure of the CSC

security property ?". The structure of CSC is described in Figure 4. In detail, the tool is able:

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

4.2 Architecture

The CSC 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 CSC maintains the strongly modular characteristic of CW. The modules of the system are 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 transition function that, according to the operational semantic rule of SPA, provides all possible transitions for an agent. This function allows the construction of the transition graph associated to an agent.

In the semantic layer, CSC uses a transformation routine to translate transition graphs into observational graphs [2]. Since it refers to processes modeled as transition graphs, it has been imported from CW in CSC without any modification.

In the analysis layer, CSC 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 [8] which finds a bisimulation between the roots of two graphs by partitioning their nodes.

5 Security Predicates

Now, we want to explain briefly how the system works in evaluating security predicates *BNNI, BSNNI, SBSNNI,* discussing, at the same time, about their computational complexity. CSC computes the value of these predicates over *finite state agents* (i.e. agents possessing a finite state transition graph), based on the definitions given in Section 2 that we report below in CSC syntax: [4]

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

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

Phase a) CSC builds the transition graphs of the two agents of which it wants to compute the equivalence. For example in the case of *BNNI*, CSC 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 nodes of the graphs due to parallel composition.

Phase b) The two transition graphs obtained in Phase a) are transformed into observational 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) The general equivalence algorithm [8] 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 equivalence. In particular we have that in most cases 80% of computation time is due to the routine for reflexive transitive closure of Phase b).

[4] In the CSC the hiding and input restriction operators are respectively represented by ! and ?, for easy of parsing.

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 predicate *SBSNNI* could be based on the exploitation of compositional properties proved in [4] and recalled in Section 7.

6 Using CSC

6.1 Sample session

The style used in specifying SPA agents in CSC is the same used for CCS agents in CW. For example the command line [5]

`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 subsets Act_H and Act_L of high and low level actions respectively. With the command

`Command:` **acth** h

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

Now, we can check whether agent A is *BNNI* secure:

`Command:` **bnni** A
`true`

CSC tells us that A is *BNNI* secure. Now we can check if agent A is *BSNNI* secure too:

`Command:` **bsnni** A
`false`

So A is *BNNI* secure but is not *BSNNI* secure. Finally the command **quit** causes an exit to the shell.

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

```
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
 Object_h0[rl0/rh0,rl1/rh1,wl0/wh0,wl1/wh1]

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 .1. Translation of Access_Monitor_1 into CSC syntax.

6.2 Checking the Access Monitor

In this Section we use CSC to analyze the systems of Example 3.8. Since CSC works on SPA agents we have to translate all the VSPA specifications into SPA. Consider system *Access_Monitor_1*. Table.1 reports the translation of *Access_Monitor_1* specification into the CSC syntax. [6] The new command **basi** has been used to bind 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.1 and load it, in CSC, 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*

[6] In the translation, we use $\{l,h\}$ in place of $\{0,1\}$ for the levels of users and objects in order to make the SPA specification more clear. Formally we make the translation considering variables l and x ranging in $\{l,h\}$. As an example $access\_r(1,0)$ becomes `access_r_hl`

```
true
```
`Command:` **sbsnni** *Access_Monitor_1*
`false:` $(\texttt{'val\_h1.Monitor} \mid \texttt{Object\_l1} \mid \texttt{Object\_h1}) \setminus \texttt{L}$ [7]

Note that when *SBSNNI* fails for a process E, CSC gives as output an agent E' which is reachable from E and is not *BSNNI*. In the following we will show that this can be useful to decide if E is *BNDC*. So we have found that $Access\_Monitor\_1 \in BSNNI, BNNI$ and $Access\_Monitor\_1 \notin SBSNNI$ Since $SBSNNI \subset BNDC \subset BSNNI, BNNI$ (see Proposition 3.6), we cannot conclude whether *Access_Monitor_1* is *BNDC* or not. However using the output of *SBSNNI* it is easy to find a high level process Π which can deadlock the monitor. In fact, in the state given as output by *SB-SNNI*, the monitor is waiting for the high level action `'val_h1`; so, if we find a process Π which leads the system to such a state and does not execute the `val_h1` action, we will have a high level process able to deadlock the monitor. It is sufficient to consider $\Pi = {}'access\_r\_hh.0$. System $(Access\_Monitor\_1|\Pi) \setminus Act_H$ will be deadlocked immediately after the execution of the read request by Π, blocking in the following state

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

(this state differs from the one given as output by *SBSNNI* only for the values stored in objects). We verify that *Access_Monitor_1* is not *BNDC* by checking that $(Access\_Monitor\_1|\Pi) \setminus Act_H \not\approx_B Access\_Monitor\_1/Act_H$ using the following commands:

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

As we said in Example 3.8, such a deadlock is caused by synchronous communications in SPA. Moreover, using the CSC output again, we can find out that also the high level process $\Pi' = {}'access\_w\_hl.0$ can deadlock *Access_Monitor_1*, this because it executes a write request and does not send the corresponding value. Hence, in Example 3.8 we proposed the modified system *Access_Monitor_3* with an output buffer 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_3*
```
true
```

[7] We write `Object_l1` instead of `Object_h1[rl0/rh0,rl1/rh1,wl0/wh0,wl1/wh1]`

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 .2. Number of states and time spent on a SPARC station 5.

7 State Explosion and Compositionality

We now want to plain out 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 CSC the two agents B, D and the set Act_H of high level actions:

```
Command: bi  B  y.a.b.B + a.b.B
Command: bi  D  '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 that *SBSNNI* is preserved by system composition, the two agents $B|D|B$ and $B|D|D|B$ must also be *SBSNNI* secure. Hence the verification of these two agents can 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 CSC, which computes the number of states of an agent, we can fill in Table.2, which points out the exponential increase of the number of states and the consequent increase of the computation time for verification of *SBSNNI*.

Theorem 7.1 [5] *The following hold:*

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

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

In the following $\mathcal{E}_{FS} \subseteq \mathcal{E}$ will denote the set of closed and guarded SPA agents with a finite lts. CSC is able to exploit the compositionality of security properties through the following algorithm:

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

- $E, E' \in P \Longrightarrow E|E' \in P$
- $E \in P, L \subseteq \mathcal{L} \Longrightarrow E \setminus L \in P$

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

1) *if E is in the form $E' \setminus L$ (recursively) calculate $A'_P(E')$; if $A'_P(E') = true$ then return true else return the result of $A_P(E)$;*
2) *if E is in the form $E_1|E_2$ (recursively) calculate $A'_P(E_1)$ 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) *otherwise return $A_P(E)$.* ■

Note that the algorithm requires that property P is closed with respect to restriction and uses this in step 1. This could seem useless; however, the parallel composition is often in the following form: $(A|B) \setminus L$ (in order to force some synchronization) 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. We have the following correctness result for the compositionality algorithm:

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

Proof. First, note that in step 1 of A'_P it is $E' \in \mathcal{E}_{FS}$ (by hypothesis) and in step 2 if $E \in \mathcal{E}_{FS}$ then $E_1, E_2 \in \mathcal{E}_{FS}$. As $F \in \mathcal{E}_{FS}$, then we recursively obtain that every E, E' and E_1, E_2 of steps 1, 2 and 3 belong to $\mathcal{E}_{FS}$. So, when the algorithm executes $A_P(E)$ in steps 1,2 or 3, it terminates because $E \in \mathcal{E}_{FS}$.

We still have to prove that, in steps 1 and 2, $A'_P(E')$ and $A'_P(E_1), A'_P(E_2)$ terminate. In particular we must prove that for every $F' \in \mathcal{E}_{FS}$ the evaluation of $A'_P(F')$ never needs to evaluate $A'_P(F')$ itself (going into an infinite loop). This holds because agents in $\mathcal{E}_{FS}$ are guarded; so the evaluation of $A'_P(F')$ could at most need to evaluate $A'_P(\mu.F')$ where μ is the guard for F'. Hence $A'_P(F)$ terminates.

When the algorithm calculates $A_P(E)$ in steps 1, 2 and 3 it is always $E \in \mathcal{E}_{FS}$, so $A_P(E) = true$ if $E \in P$ and $A_P(E) = false$ if $E \notin P$. Hence, by (partial) structural induction and using compositionality properties, we obtain that $A_P(F) = A'_P(F)$. ■

The theorem above requires that, every time the algorithm executes step 1, E' belongs to $\mathcal{E}_{FS}$; i.e. in $A'_P(E)$, if E is in the form $E' \setminus L$ then E' must be finite state. As an example, 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 calculate $A_P(E)$ and E is not finite state.

Note that such a condition trivially holds if we specify systems as composition and restriction of finite state subsystems. In particular we can use

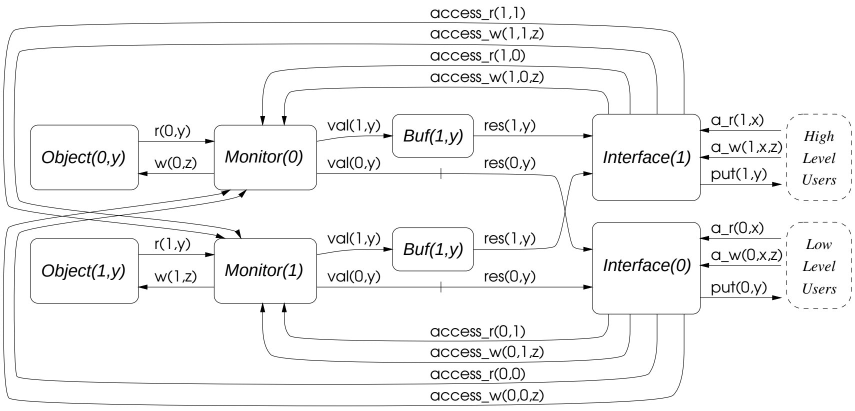

FIGURE 5. The compositional Access_Monitor

the following syntax which defines the so called *nets of automata*:

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

where for every constant Z there must be the corresponding definition $Z \stackrel{\text{def}}{=} p$.

It not necessary to use such a syntax in order to satisfy theorem hypotheses. As an example consider the following agent $B \stackrel{\text{def}}{=} a.\underline{0} + D \setminus \{i\}$ with $D \stackrel{\text{def}}{=} i.(o.\underline{0}|D)$ which is finite state but it is not a net of automata. Since the top operator is a $+$ then $A'_P(B)$ behaves just like $A_P(B)$ and so it terminates (theorem hypotheses trivially hold in this case).

Example 7.4 Consider again *Access_Monitor_3*. The verification of the *SBSNNI* property on such a system requires a lot of time (more than 1 hour on a SUN5 workstation) because of the above mentioned exponential state explosion due to parallel composition. We can try to verify *SBSNNI* using the compositional algorithm. Unfortunately we have that *Monitor* is not *SBSNNI* and so, in this case, the compositional technique cannot help us to reduce the execution time. This happens because the *BNDC*-security of *Access_Monitor_3* depends on both monitor and objects; i.e. process *Monitor* is not able to guarantee multilevel security for every possible object connected to it. As an example, consider the following modified objects:

$$Object(x,y) \stackrel{\text{def}}{=} \overline{r}(x,y).Object(x,0) + w(x,z).Object(x,z)$$

which reset (to zero) their value every time they are read. Using these objects, we obtain a system which is not *BSNNI* and so is not *BNDC*. In such a system, a high level user can change (to zero) the value of the low level variable by simply reading it. This is generally called "half-bit" covert channel because the high level user can set the low level variable only to one of the two possible values (in this case 0) and so can transmit only a half-bit information to low level. In [3] we show how to construct a 1-bit channel using some half-bit ones.

Finally we present a version of the Access Monitor (Figure 5) which can be verified very efficiently by exploiting the compositionality of *SBSNNI*. Here every object has a "private" monitor which implements the access functions for such (single) object. To make this, we have decomposed process $Monitor$ (which is not *BNDC*) into two different processes, one for each object; then we have composed such processes to respective objects together with a high level buffer obtaining the *BNDC*-secure $Modh$ and $Modl$ agents. In particular, $Monitor(x)$ handles the accesses to object x ($x = 0$ low, $x = 1$ high). We also have an interface which guarantees the exclusive use of the monitor within the same level. Moreover the new interface actions $a\_r(l, x)$, $a\_w(l, x, z)$ and $put(l, y)$ substitute actions $access\_r(l, x)$, $access\_w(l, x, z)$ and $res(l, y)$ in the interaction between the users and the monitor.

$$
\begin{aligned}
Access\_Monitor\_4 &\stackrel{\text{def}}{=} (Modh|Modl|Interf) \setminus L \\
Modh &\stackrel{\text{def}}{=} ((Monitor(1)|Object(1,0)|Buf(1, empty)) \setminus Lh) \\
&\qquad [res(0, y)/val(0, y)] \\
Modl &\stackrel{\text{def}}{=} ((Monitor(0)|Object(0,0)|Buf(1, empty)) \setminus Ll) \\
&\qquad [res(0, y)/val(0, y)] \\
Interf &\stackrel{\text{def}}{=} Interf(0)|Interf(1) \\
Interf(l) &\stackrel{\text{def}}{=} a\_r(l, x).\overline{access\_r}(l, x).res(l, y).\overline{put}(l, y).Interf(l) + \\
&\qquad + a\_w(l, x, z).\overline{access\_w}(l, x, z).Interf(l) \\
Monitor(x) &\stackrel{\text{def}}{=} access\_r(l, x).(\text{ if } x \leq l \text{ then } r(x, y).\overline{val}(l, y). \\
&\qquad Monitor(x) \text{ else } \overline{val}(l, err).Monitor(x)) + \\
&\qquad + access\_w(l, x, z).(\text{ if } x \geq l \text{ then } \overline{w}(x, z).Monitor(x) \\
&\qquad \text{ else } Monitor(x)) \\
Object(x, y) &\stackrel{\text{def}}{=} \overline{r}(x, y).Object(x, y) + w(x, z).Object(x, z) \\
Buf(x, j) &\stackrel{\text{def}}{=} \overline{res}(x, j).Buf(x, empty) + val(x, k).Buf(x, k)
\end{aligned}
$$

where $L = \{res, access\_r, access\_w\}$, $Lh = \{r, w, val(1, y)\}$ and $Ll = \{r, w, val(1, y)\}$. Table.3 reports the output of the (successful) verification of *SBSNNI* on $Access\_Monitor\_4$. This task requires about 1 minute on a SUN5 workstation. We can also check that $Access\_Monitor\_4 \approx_B (Access\_Monitor\_3|Interf) \setminus L$ with $L = \{res, access\_r, access\_w\}$; i.e. this Access Monitor version is equivalent to the $Access\_Monitor\_3$ with the interface. Such equivalence verification requires about 10 minutes. Note that, by Theorem 3.7, we can conclude that also $(Access\_Monitor\_3|Interf) \setminus L$ is *BNDC*, even if a direct (non compositional) check would take about

```
Verifying Modh | Modl | Interf
  Verifying Modh
  Verifying Modl
  Verifying Interf
    Verifying Interf_h
    Verifying Interf_l
true
```

TABLE .3. Verification of *SBSNNI* on *Access_Monitor_4* exploiting compositionality.

20 minutes (about 20 times longer than checking the equivalent process *Access_Monitor_4*). Note that checking $(Access\_Monitor\_3|Interf) \setminus L$ requires less time than checking *Access_Monitor_3* alone. So for this agent the compositional algorithm takes more time with respect to direct checking. This happens because $(Access\_Monitor\_3|Interf) \setminus L$ has less states than *Access_Monitor_3*; in fact, the interface reduces the internal parallelism in system *Access_Monitor_3* (in particular the parallelism given by the action of the buffers). Hence it is useful to adopt the compositional technique when building complex systems as parallel composition of simpler ones, i.e. when the number of states increases (e.g. as in *Access_Monitor_4*). ■

8 References

[1] 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.

[2] 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.

[3] R. Focardi and R. Gorrieri. "The Compositional Security Checker: A Tool for the Automatic Compositional Verification of Security Properties". Forthcoming.

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

[5] R. Focardi, R. Gorrieri, and V. Panini. "The Security Checker: a Semantics-based Tool for the Verification of Security Properties". In *Proceedings Eight IEEE Computer Security Foundation Workshop, (CSFW'95) (Li Gong Ed.)*, pages 60–69, Kenmare (Ireland), June 1995. IEEE Press.

[6] 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.

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

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

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

[10] 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.

[11] 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.