Abstract Interpretation and Model Checking for Checking Secure Information Flow in Concurrent Systems

Nicoletta De Francesco
Dipartimento di Ingegneria della Informazione, Università di Pisa, Italy
e-mail: n.defrancesco@iet.unipi.it

Antonella Santone
RCOST - Research Centre on Software Technology, University of Sannio, Benevento, Italy
e-mail: santone@unisannio.it

Luca Tesei
Dipartimento di Informatica, Università di Pisa - Italy
e-mail: tesei@di.unipi.it

Abstract. We propose a method to check secure information flow in concurrent programs with synchronization. The method is based on the combination of *abstract interpretation* and *model checking*: by abstract interpretation we build a finite representation (transition system) of the behavior of the program. Then we model check the the abstract transition system with respect to the security properties, expressed by a set of temporal logic formulae. The approach allows certifying more programs than previous methods do. The main point is that we are able to check more carefully the scope of indirect information flows.

1 Introduction and Overview

The *secure information flow* property [11] requires that information at a given security level does not flow to lower levels. A program in which every variable is assigned a security level has secure information flow if, when the program terminates, the value of each variable does not depend on the initial value of the variables with higher security levels. Let us suppose to have two security levels, h and l, such that h is higher than l. Suppose that variable `x` has security level l and variable `y` has security level h. Examples of violation of secure information flow are:

```
x:=y,
if  y=0  then  x:=1  else  x:=0
if  y=0  then  x:=1  else  skip.
```

In the first case, there is a *direct* information flow from level h to level l, while, in the second and third cases there is an *indirect* information flow: in both cases, checking the final value of `x` reveals information on the value of the higher security variable `y`. While direct information flow occurs with assignments, indirect information flows are generated by branching commands, which in high level languages are `if` and `while` commands. Other violations of secure information flow may occur when the behavior of the program depends on the high information flow [24]. Consider the program: `while y>0 do skip` which terminates only if `y` $\leq$ `0`: observing its termination reveals information on the value of `y`. Many methods have been defined to cope with secure information flow in sequential programs. Some of them are based on a static analysis [1, 3, 12, 23], while others are semantics based [5, 16–18].

Concurrent programs are considered in [19, 21, 22], which present type systems for certifying these programs. When considering parallel programs composed of a set of synchronously communicating processes, handling indirect flows is more difficult, due to the possible blocking of processes on synchronization points. In fact in sequential languages, the scope of the indirect flow caused by a branching command can be statically derived, since it coincides with the scope of the command itself. When the command has been completely executed, the indirect flow terminates, since the successive command is however executed, whatever branch has been chosen before. For example the program (with `y,w` high variables and `x` low variable):

```
if y=0 then w:=5 else skip; x:=4;
```

is correct, since the assignment to `x` occurs outside the high indirect flow. When considering a concurrent language with synchronization, the calculation of the scope of indirect flows is complicated by the fact that some synchronization may not occur, and thus a process can be blocked or delayed, altering the control flow of the program. Consider for example the following process, where `x` is the only low variable and `SYNC` is a synchronization point:

`if y=0 then SYNC else skip; x:=4;` $\|$ `...` (a)

There is a high indirect flow from the value of `y` to the final value of `x`: in fact, if `y` $\neq$ `0`, then the final value of `x` is 4, while, if `y = 0`, the final value of `x` may be equal to the initial value, if the synchronization cannot occur since no parallel process is willing to communicate. Thus the indirect flow generated by the `if` command extends till outside the command itself. Another problem to cope with is that, due to synchronizations, an command i may belong to an indirect flow caused by a branching command belonging to another process, as occurs for the assignment `x:=4;` in the following example:

`if y=0 then SYNC else ...` $\|$ `... SYNC; x:=4;`

A solution to ensure secure information flow is to impose that no synchronization can occur within any branching command with high guard [19], to be sure that the command always terminates, and thus the scope of the indirect flow coincides with that of the branching command. But this constraint is in many cases too restrictive: for example, if the sequential process in (a) is put in parallel with a process always willing to synchronize, then the final value of `x` is not affected by the high indirect flow: in this case the indirect flow terminates with the conditional command, like as in a sequential language. The point is that the scope of indirect flows depends on the dynamic behavior of the program.

In this paper we propose a method to check secure flow in concurrent programs based on the combination of *abstract interpretation* and *model checking* [20]: by abstract interpretation we build a finite representation (transition system) of the behavior of the program, representing all possible executions. Then we model check the abstract transition system against a set of temporal logic formulae expressing the security properties. In particular, we inspect the behavior of indirect flows by checking if and when indirect flows terminate.

2 The security properties

We consider concurrent programs consisting of a number of independently executing sequential processes with private memory. Processes may communicate each other by sending/receiving messages on a set of one-way channels. Each channel is private to two processes. The communications are synchronous: processes synchronize by explicit sending and receiving primitives, which are defined in the CSP [13] style, as follows:

$a!e$: to send the expression e over the channel a;

$a?x$: to receive a message from the channel a and save it into the variable x.

Let us now define the syntax and semantics of our concurrent language with synchronization. The language is a usual high level language.

We assume a set of constant numbers, ranged over by k, a countable set *Var* of variables ranged over by $x, y, \ldots$, and a countable set of *channel names*, ranged over by a, $b, \ldots$. We assume that all simple commands (assignments, sending/receiving of messages, `skip` command) are labeled by a set $\mathcal{L} \subseteq \mathbb{N}$ of labels, ranged over by i.

$$
\begin{array}{lcl}
exp & ::= & k \mid x \mid exp \;\; op \;\; exp \\
simple\_com & ::= & \texttt{skip} \mid x := exp \mid a?x \mid a!exp \\
com & ::= & i : simple\_com \mid \texttt{if}\ exp\ \texttt{then}\ com\ \texttt{else}\ com \mid \\
 & & \texttt{while}\ exp\ \texttt{do}\ com \mid com; com \mid \{com\} \\
proc & ::= & com \mid proc \| proc
\end{array}
$$

where *op* stands for the usual arithmetic and logic operations. We denote by $\mathcal{P}$, ranged over by p, the set of *processes* generated by *proc*.

The *concrete semantics* of the language is given by means of an operational semantics, defining a labeled transition system. $\mathcal{V}$ is the domain of values, $\mathcal{M} = Var \longrightarrow \mathcal{V}$ is the domain of memories, ranged over by m. The states are a subset of $\mathcal{Q} = \mathcal{P} \times \mathcal{M}$: each state is a pair $\langle p, m \rangle$ of a process and a memory. p can be also λ and in that case $\langle \lambda, m \rangle$ represents a state in which no action can be executed. The actions are the domain $\mathcal{A} = \mathcal{L} \cup (\mathcal{L} \times \mathcal{L}) \cup \{\tau\}$, ranged over by α: each action is either i) a label of a simple command different from a sending and receiving command, or ii) an ordered pair of labels labeling two matching sending and receiving command (defined on the same channel), or iii) the evaluation of a condition (action τ).

$$\textbf{Assign}\ \frac{\langle e,m\rangle \longrightarrow_{expr} k}{\langle i: x := e, m\rangle \xrightarrow{i} \langle \lambda, m[k/x]\rangle}$$

$$\textbf{If}_{true}\ \frac{\langle e,m\rangle \longrightarrow_{expr} \texttt{true}}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m\rangle \xrightarrow{\tau} \langle c_1, m\rangle}$$

$$\textbf{If}_{false}\ \frac{\langle e,m\rangle \longrightarrow_{expr} \texttt{false}}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m\rangle \xrightarrow{\tau} \langle c_2, m\rangle}$$

$$\textbf{Skip}\ \frac{}{\langle i: \texttt{skip}, m\rangle \xrightarrow{i} \langle \lambda, m\rangle}$$

$$\textbf{While}_{true}\ \frac{\langle e,m\rangle \longrightarrow_{expr} \texttt{true}}{\langle \texttt{while}\ e\ \texttt{do}\ c\ , m\rangle \xrightarrow{\tau} \langle c; \texttt{while}\ e\ \textbf{do}\ c\ , m\rangle}$$

$$\textbf{While}_{false}\ \frac{\langle e,m\rangle \longrightarrow_{expr} \texttt{false}}{\langle \texttt{while}\ e\ \texttt{do}\ c\ , m\rangle \xrightarrow{\tau} \langle \lambda, m\rangle}$$

$$\textbf{Seq}_1\ \frac{\langle c_1, m\rangle \xrightarrow{\alpha} \langle \lambda, m'\rangle}{\langle c_1; c_2, m\rangle \xrightarrow{\alpha} \langle c_2, m'\rangle} \qquad \textbf{Seq}_2\ \frac{\langle c_1, m\rangle \xrightarrow{\alpha} \langle c_1', m'\rangle}{\langle c_1; c_2, m\rangle \xrightarrow{\alpha} \langle c_1'; c_2, m'\rangle}$$

$$\textbf{Com}\ \frac{\langle e,m\rangle \longrightarrow_{expr} k}{\begin{array}{l}\langle c_1 \| \cdots \| r: a?x; c_i \| \cdots \| t: a!e; c_j \| \cdots \| c_n, m\rangle \xrightarrow{r,t} \\ \qquad \langle c_1 \| \cdots \| c_i \| \cdots \| c_j \| \cdots \| c_n, m[k/x]\rangle\end{array}}$$

$$\textbf{Par}\ \frac{\langle c_i, m\rangle \xrightarrow{\alpha} \langle c_i', m'\rangle}{\begin{array}{l}\langle c_1 \| \cdots \| c_i \| \cdots \| c_n, m\rangle \xrightarrow{\alpha} \\ \qquad \langle c_1 \| \cdots \| c_i' \| \cdots \| c_n, m'\rangle\end{array}} \quad c_i \neq r: a!e,\ c_i \neq r: a?x$$

Table 1. Concrete Semantics Rules

The semantics is given by the rules in Table 1, which define a relation $\longrightarrow \subseteq \mathcal{Q} \times \mathcal{A} \times \mathcal{Q}$. If $(q_1, \alpha, q_2) \in \longrightarrow$ we write $q_1 \xrightarrow{\alpha} q_2$. The relation $\longrightarrow_{expr} \subseteq (exp \times \mathcal{Q}) \times \mathcal{V}$ is used to compute the value of the expressions. With $m[k/x]$ we denote the memory m' which agrees with m on all variables, except for x, for which $m'(x) = k$. In the following, if $\delta \in \mathcal{A}^*$ and $\delta = \alpha_1, \ldots, \alpha_n$, $n \geq 1$, we write $p \xrightarrow{\delta} p'$ to mean $p \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} p'$. For the empty sequence λ of actions, we have that $p \xrightarrow{\lambda} p$, for every $p \in \mathcal{P}$. Moreover, $\langle p, m\rangle \not\longrightarrow$ means that no p', m', α exist such that $\langle p, m\rangle \xrightarrow{\alpha} \langle p', m'\rangle$.

For the sake of simplicity, in this paper procedures and other language features are not included. Moreover we assume that each sequential process is deterministic. This means that in each state of the concrete semantics at most one transition involving a sequential process is enabled.

We assume, for simplicity, to have a set $\mathcal{S} = \{l, h\}$ of two security levels, ordered by $l \sqsubset h$. Given a process p, the set of variables occurring in p is partitioned into low level an high level ones. More precisely, a program P is a triple $\langle p, L, H\rangle$ where p is a process, and L and H are the high and low variables of p, respectively ($H \cap L = \emptyset$). Given $\sigma, \tau \in \mathcal{S}$, $\sigma \sqcup \tau$ denotes the least upper bound of σ and τ.

We now define the secure information flow property, which describes the fact that information with high security level is kept secret. This notion is elsewhere called non-interference [23]. First, we need the following definition, which states when two memories agree on all low variables.

Definition 1. *Let $L \subseteq Var$ and $m_1, m_2 \in \mathcal{M}$.*

$$m_1 =_L m_2 \text{ if and only if } \forall x \in L,\ m_1(x) = m_2(x)$$

Definition 2 (secure information flow). *Let $P = \langle p, L, H \rangle$ be a program. P is secure if and only if $\forall m_1, m_2 \in \mathcal{M}$ such that $m_1 =_L m_2$*

$$\text{if } \langle p, m_1 \rangle \xrightarrow{\delta} \langle p_1, m_1' \rangle \not\rightarrow \text{ and } \langle p, m_2 \rangle \xrightarrow{\gamma} \langle p_2, m_2' \rangle \not\rightarrow \text{ then } m_1' =_L m_2',$$

where $\delta, \gamma \in \mathcal{A}^*$.

A secure program has the property that, if it terminates starting with two memories which agree on the value of the low variables, then the resulting memories also agree on the value of the low variables, regardless the initial value of the high variables. This means that the initial value of the high variables does not influence the final value of the low variables in all executions. Note that this definition considers only terminating executions: even if a program is secure, its termination may be influenced by the high information flow. For example, the programs (with `y` high variable) `if y=1 then (while true do skip)` and `while y=1 do skip` have an insecure information flow: observing the termination of the program reveals that `y` was not 1. This information flow is called *covert flow* [24]. The following property expresses this fact.

Definition 3 (secure termination).
A program $P = \langle p, L, H \rangle$ has secure termination if and only if $\forall m_1, m_2 \in \mathcal{M}$ such that $m_1 =_L m_2$:

$$\langle p, m_1 \rangle \xrightarrow{\delta} \langle p_1, m_1' \rangle \not\rightarrow \text{ implies } \langle p, m_2 \rangle \xrightarrow{\gamma} \langle p_2, m_2' \rangle \not\rightarrow \quad \text{where } \delta, \gamma \in \mathcal{A}^*.$$

3 Abstract interpretation

Abstract interpretation [9, 14] is a method for analyzing programs in order to collect approximate information about their run-time behavior. It is based on a non-standard semantics, that is a semantic definition in which simpler (abstract) domains replace the standard (concrete) ones, and the operations are interpreted on the new domains. The purpose of abstract interpretation is to correctly approximate the concrete semantics of all executions in a finite way, keeping only the information concerned with the analysis of a given property. Here we consider abstract interpretation of operational semantics. The abstract semantics of a program is a finite abstraction of the concrete one: the labels of the transitions are the same, but each concrete state is approximated by its command part, i.e. both values and memories are forgotten, since we are interested in the control flow of the program. The labels of the transitions are the same as in the concrete semantics, since we need to know what commands may be executed and their order, and the beginning and termination of the branching commands. Note that, when dealing with conditional or iterative commands, the abstract transition system has multiple execution paths due to the loss of precision of abstract data. The abstract domains are $\mathcal{L}^\natural = \mathcal{L}, \mathcal{A}^\natural = \mathcal{A}, \mathcal{M}^\natural = \{\cdot\}, \mathcal{Q}^\natural = \mathcal{P}^\natural \times \mathcal{M}^\natural = \mathcal{P} \times \{\cdot\} = \mathcal{P}$ Every constant value $k \in \mathcal{V}$ and every memory is abstracted to "$\cdot$", and each state $\langle p, m \rangle$ is abstracted in p. The abstract semantics is shown in Table 2: its rules are obtained by simplifying the corresponding rules of the concrete semantics, used on the abstract domains. ($\cdot\ op\ \cdot = \cdot$ and $\cdot[k/x] = \cdot$). The transition relation of the abstract semantics is denoted by $\longrightarrow^\natural \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}$. Note that, for `if` commands both rules $\mathbf{If}_{true}$ and $\mathbf{If}_{false}$ apply, since $true$ and $false$ are both abstracted to "$\cdot$". The same occurs for `while` commands. Given a program P, we denote by $A(P)$ the abstract transition system defined by the abstract rules.

The following theorem states that the abstract semantics includes all possible execution paths of a program.

Theorem 1. *Let $P = \langle p, L, H \rangle$ be a program and $m_0 \in \mathcal{M}$. For each path $\langle p, m_0 \rangle \xrightarrow{\alpha_1} \langle p_1, m_1 \rangle \xrightarrow{\alpha_2} \ldots$ there exists a path $p \xrightarrow{\alpha_1}{}^\natural\ p_1 \xrightarrow{\alpha_2}{}^\natural \ldots$ in $A(p)$.*

Note that $A(p)$ is finite for each program $P = \langle p, L, H \rangle$. Since loop nesting levels are finite and in a sequence of commands only the leftmost command is acted on, thus the number of different commands which may occur in the abstract states is finite.

$$\mathbf{Assign}\ \frac{}{i : x := e \xrightarrow{i}{}^\natural\ \lambda} \qquad \mathbf{Skip}\ \frac{}{i : skip \xrightarrow{i}{}^\natural\ \lambda}$$

$$\mathbf{If}_{true}\ \frac{}{\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \xrightarrow{\tau}{}^\natural\ c_1} \qquad \mathbf{If}_{false}\ \frac{}{\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \xrightarrow{\tau}{}^\natural\ c_2}$$

$$\mathbf{While}_{true}\ \frac{}{\texttt{while}\ e\ \texttt{do}\ c \xrightarrow{\tau}{}^\natural\ c; \texttt{while}\ e\ \mathbf{do}\ c} \qquad \mathbf{While}_{false}\ \frac{}{\texttt{while}\ e\ \texttt{do}\ c \xrightarrow{\tau}{}^\natural\ \lambda}$$

$$\mathbf{Seq}_1\ \frac{c_1 \xrightarrow{\alpha}{}^\natural\ \lambda}{c_1; c_2 \xrightarrow{\alpha}{}^\natural\ c_2} \qquad \mathbf{Seq}_2\ \frac{c_1 \xrightarrow{\alpha}{}^\natural\ c_1'}{c_1; c_2 \xrightarrow{\alpha}{}^\natural\ c_1'; c_2}$$

$$\mathbf{Com}\ \frac{}{c_1 \| \cdots \| r : a?x; c_i \| \cdots \| t : a!e; c_j \| \cdots \| c_n \xrightarrow{r,t}{}^\natural\ c_1 \| \cdots \| c_i \| \cdots \| c_j \| \cdots \| c_n}$$

$$\mathbf{Par}\ \frac{c_i \xrightarrow{\alpha}{}^\natural\ c_i'}{c_1 \| \cdots \| c_i \| \cdots \| c_n \xrightarrow{\alpha}{}^\natural\ c_1 \| \cdots \| c_i' \| \cdots \| c_n} \quad c_i \neq r : a!e,\ c_i \neq r : a?x$$

Table 2. Abstract Semantics Rules

4 Model checking the abstract semantics

In the model checking framework, systems are modeled as transition systems and requirements are expressed as formulae in temporal logic. A model checker then accepts two inputs, a transition system and a temporal formula, and returns "true" if the system satisfies the formula and "false" otherwise. In this work we use the *selective mu-calculus* logic [4], which is a variant of mu-calculus [15] and differs from it in the definition of the modal operators. The syntax of the selective mu-calculus is the following, where $K, R \subseteq \mathcal{A}$, while Z ranges over a set of variables:

$$\phi ::= \mathtt{tt} \mid \mathtt{ff} \mid Z \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi \Rightarrow \phi \mid \ [K]_R\, \phi \mid \ \langle K \rangle_R\, \phi \mid \nu Z.\phi \mid \mu Z.\phi$$

The satisfaction of a formula ϕ by a state s of a transition system, written $s \models \phi$, is defined in the usual way for $\vee, \wedge$ and $\Rightarrow$ operators. $[K]_R\, \phi$ and $\langle K \rangle_R\, \phi$ are the selective modal operators and their meaning is the following:

$[K]_R\, \phi$ is satisfied by a state which, for every performance of a sequence of actions not belonging to $R \cup K$, followed by an action in K, evolves in a state obeying ϕ.

$\langle K \rangle_R\, \phi$ is satisfied by a state which can evolve to a state obeying ϕ by performing a sequence of actions not belonging to $R \cup K$ followed by an action in K.

As in standard mu-calculus, a fixed point formula has the form $\mu Z.\phi$ $(\nu Z.\phi)$ $\mu Z.\phi$ is the least fix-point of the recursive equation $Z = \phi$, while $\nu Z.\phi$ is the greatest one. A transition system T satisfies a formula ϕ, written $T \models \phi$, if and only if $s \models \phi$, where s is the initial state of T. A program P satisfies ϕ if $A(P) \models \phi$.

In the sequel we use the following formula which expresses the property that an action α eventually occurs.

$$\mathit{eventually}(\alpha) = \mu Z = \langle - \rangle\, \mathtt{tt} \wedge [-\alpha]\, Z$$

We need some notations and definitions. The *control flow graph*, $G(P)$, of the program represents the dependencies among the simple commands of the program: $G(P)$ is the directed graph having as nodes the labels of the commands, and where the edges are defined as follows: $G(P)$ contains the edge (i, j) if and only if a) i and j belong to the same sequential process and the command labeled by j can be immediately executed after the command labeled by i; or b) i and j are two matching sending/receiving or receiving/sending commands.

The following function $\mathit{level} : \mathit{exp} \longrightarrow \mathcal{S}$ statically computes the security level of an expression occurring in P.

$$\begin{aligned}
\mathit{level}(k) &= l \\
\mathit{level}(x) &= \begin{cases} l \ \text{ if } x \in L \\ h \text{ if } x \in H \end{cases} \\
\mathit{level}(e_1\ \mathit{op}\ e_2) &= \mathit{level}(e_1) \sqcup \mathit{level}(e_2)
\end{aligned}$$

To handle indirect flows, we need to represent in the abstract transition system where an implicit flow begins and where it ends. To clearly emphasize these points, we suppose that each `if` and each `while` command is preceded and followed by a `skip` command. Given a program P, we define the following actions sets which are subsets of $\mathcal{A}$. Let $\sigma \in \mathcal{S}$.

$$L\_ASSIGN^P(\sigma) = \left\{ i \mid \boxed{\text{i: x:=e}} \text{ occurs in } P,\ x \in L,\ level(e) = \sigma \right\}$$

$$L\_COMM^P(\sigma) = \left\{ (i,j) \mid \boxed{\text{i: a?x}} \text{ and } \boxed{\text{j: a!e}} \text{ occur in } P,\ x \in L,\ level(e) = \sigma \right\}$$

The set $L\_ASSIGN^P(\sigma)$ contains the labels of all assignments of an expression with level σ to a low variable, while $L\_COMM^P(\sigma)$ contains the actions corresponding to all synchronizations in which the value of an expression with level σ is stored into a low variable.
We use also the set

$$L\_ASSIGN^P = L\_ASSIGN^P(l) \cup L\_ASSIGN^P(h)$$

of all assignments to a low variable and

$$L\_COMM^P = L\_COMM^P(l) \cup L\_COMM^P(h)$$

of all communications involving a low variable.

To discover illegal direct flows we check the following selective mu-calculus formula:

$$\boxed{direct = \bigwedge_{\alpha \in L\_ASSIGN^P(h) \cup L\_COMM^P(h)} [\alpha]_{\emptyset}\, \mathtt{ff}}$$

The formula *direct* means that *no assignment of a high expression to a low variable can occur.* Now let us consider indirect flows. We use the following sets of actions.

$$IF^P(\sigma) = \left\{ (i,j) \,\middle|\, \begin{array}{l} \boxed{i : \mathtt{skip}; \mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2;\ j : \mathtt{skip}} \text{ occurs in } P, \\ level(e) = \sigma \end{array} \right\}$$

$$WHILE^P(\sigma) = \left\{ (i,j) \,\middle|\, \begin{array}{l} \boxed{i : \mathtt{skip}; \mathtt{while}\ e\ \mathtt{do}\ c;\ j : \mathtt{skip}} \text{ occurs in P}, \\ level(e) = \sigma \end{array} \right\}$$

For each `if` (`while`) command with a condition with level σ, $IF^P(\sigma)$ ($WHILE^P(\sigma)$) contains the pair of the labels of the `skip` commands immediately preceding and following the `if` (`while`) command.

$$\begin{aligned} DEP^P(i) = & \left\{ j \mid j \in L\_ASSIGN^P \text{ and there is a path from } i \text{ to } j \text{ in } G(P) \right\} \\ & \cup \left\{ (j,r) \,\middle|\, \begin{array}{l} (j,r) \in L\_COMM^P \text{ and there is a path} \\ \text{from } i \text{ to } j \text{ and from } i \text{ to } r \text{ in } G(P) \end{array} \right\} \end{aligned}$$

Given a label i, $DEP^P(i)$ contains the labels of all assignments or communication actions involving a low variable that follow the command labeled by i in every execution. If i is the action immediately preceding a conditional or iterative command, the execution of the actions in $DEP^P(i)$ could be influenced by the indirect information flow beginning at i. An action k in $DEP^P(i)$ does not belong to the indirect flow beginning at i only if a) it does never occur before the termination of the conditional command; and b) if it occurs, the conditional command must always terminate in all executions. An action not in $DEP^P(i)$, instead, is performed asynchronously with respect to the conditional command and thus is not influenced by the indirect flow. The following formula expresses this fact.

$$\boxed{\begin{array}{l} \mathit{indirect\_if} = \\ \bigwedge_{(i,j)\in IF^P(h)} \quad \bigwedge_{\alpha\in DEP^P(i)} \nu Z.\,[i]_\emptyset \left([\alpha]_{\{j\}}\,\mathtt{ff} \wedge (\langle\alpha\rangle_\emptyset\,\mathtt{tt} \Rightarrow \mathit{eventually}(j)) \wedge Z\right) \end{array}}$$

The meaning of the formula $\mathit{indirect\_if}$ that, whenever an `if` command with an high guard begins, ($[i]_\emptyset$ with $(i,j) \in IF^P(h)$), a) no assignment $\alpha \in DEP^P(i)$ to a low variable depending on that `if` command is performed until the end of the `if` is reached ($[\alpha]_j\,\mathtt{ff}$); and b) if such an action is performed ($\langle\alpha\rangle_\emptyset\,\mathtt{tt}$), then the command must terminate in whatever execution ($\mathit{eventually}(j)$). Note that for sequential programs condition a) would be sufficient to guarantee that α is not influenced by the indirect flow. Instead, in parallel programs, the scope of the indirect flow caused by a conditional command propagates beyond the syntactic scope of the command, unless the command itself always terminates. Analogously, the indirect flow caused by `while` commands is checked by the following formula:

$$\boxed{\mathit{indirect\_while} = \bigwedge_{(i,j)\in WHILE^P(h)} \quad \bigwedge_{\alpha\in DEP^P(i)} \nu Z.\,[i]_\emptyset \left([\alpha]_{\{j\}}\,\mathtt{ff} \;\wedge\; Z\right)}$$

The meaning of the formula $\mathit{indirect\_while}$ is: whenever a `while` command with high guard begins ($[i]_\emptyset$), no assignment to a low variable depending on that `while` command can be performed, until the `while` command terminates ($[\alpha]_{\{j\}}\,\mathtt{ff}$).

Let us now consider covert flows that arise from nontermination. A covert flow of this kind can occur when the termination/nontermination of the program depends on some high information flow generated by some `while` command. This occurs when a `while` command with an high condition is executed: the termination/nontermination of the program depends on an high value. Another case is when a `while` command (with a guard of whatever level) belongs to the scope of the information flow caused of an `if` command with high guard. In this case the the `while` command (that can cause nontermination) may be or may be not executed, depending on the high guard of the `if` command. The two situations are checked by the following two formulae.

$no\_while = \bigwedge_{(i,j)\in WHILE^P(h)} [i]_\emptyset\, \mathtt{ff}$
$no\_while\_in\_if = \bigwedge_{(i,j)\in IF^P(h)} \quad \bigwedge_{(k,t)\in WHILE^P(l)\cup WHILE^P(h), k\in DEP^P(i)} \varphi$
where
$\varphi = \nu Z.\, [i]_\emptyset \left([k]_{\{j\}}\, \mathtt{ff} \wedge (\langle k\rangle_\emptyset\, \mathtt{tt} \Rightarrow \mathit{eventually}(j)) \wedge Z \right)$

no_while guarantees that the guards of all `while` loops have low level, while *no_while_in_if* says that in the scope of all `if` commands with high guard no `while` commands is allowed.

4.1 Examples

Consider $P_1 = \langle p_1, \{x\}, \{y, w\}\rangle$ and $P_2 = \langle p_2, \{x\}, \{y, w\}\rangle$ where p_1 and p_2 are defined as follows:

$p_1 = (1 : \mathtt{skip}; \mathtt{if}\ y = 0\ \mathtt{then}\ 2 : \mathtt{a?w}\ \mathtt{else}\ 3 : \mathtt{skip};\ 4 : \mathtt{skip};\ 5 : x := 4) \parallel 6 : b!2$
$p_2 = (1 : \mathtt{skip}; \mathtt{if}\ y = 0\ \mathtt{then}\ 2 : \mathtt{a?w}\ \mathtt{else}\ 3 : \mathtt{skip};\ 4 : \mathtt{skip};\ 5 : x := 4) \parallel 6 : a!2$

The program P_1 is not secure: checking the final value of the low variable x reveals information about the high variable y. The abstract transition system of p_1 (shown in Figure 1) does not satisfy the "*indirect_if*" property, which is expressed by the following formula:

$$\nu Z.\Big([1]_\emptyset \left([5]_{\{4\}}\, \mathtt{ff} \wedge (\langle 5\rangle_\emptyset\, \mathtt{tt} \Rightarrow\ \mathit{eventually}(4))\right)\ \wedge\ Z\Big)$$

In $A(p_1)$ the end of the `if` command (labeled with 4) occurs before the assignment to the low level variable x (labeled with 5 and belonging to $DEP^{P_1}(1)$), but it does not occur in each execution. We instead accept as correct the second program, since the conditional command always terminates. It holds that $A(p_2)$ (shown in Figure 1) satisfies the security formulae.

Consider now $P_3 = \langle p_3, \{\}, \{y\}\rangle$, where p_3 is defined as follows:

$1 : \mathtt{skip}; \mathtt{if}\ y = 1\ \mathtt{then}\ \{2 : \mathtt{skip};\ \mathtt{while}\ \mathit{true}\ \mathtt{do}\ 3 : \mathtt{skip}; 4 : \mathtt{skip}\}$
$\quad\quad \mathtt{else}\ 5 : \mathtt{skip};\ 6 : \mathtt{skip}$

This program has a covert flow: observing the termination of the program reveals that `y` was not 1. The abstract transition system of p_3, shown in Figure 1, does not satisfies the "*no_while_in_if*" property, which is expressed by the following formula:

$$\varphi = \nu Z.\, [1]_\emptyset \left([2]_{\{6\}}\, \mathtt{ff} \wedge (\langle 2\rangle_\emptyset\, \mathtt{tt} \Rightarrow \mathit{eventually}(6)) \wedge Z \right)$$

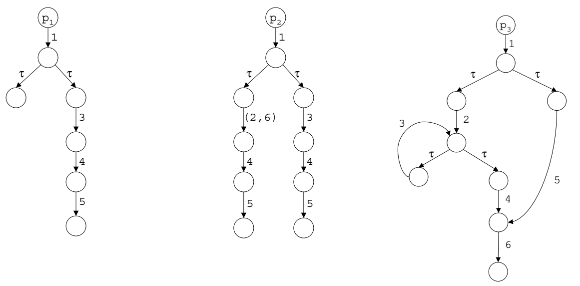

Fig. 1. The abstract transition systems of p_1, p_2 and p_3

4.2 Correctness

We now sketch the proof of the correctness of our method. The following lemma ensures that, if the abstract transition system of a program satisfies the property "*direct*", then no assignment to a low variable of an high expression is performed.

Lemma 1. *Let* $P = \langle p, L, H\rangle$ *be a program. Suppose that* $A(p) \models direct$. *For each path:* $p \xrightarrow{\delta}{}^{\natural} p', \quad \delta \in \mathcal{A}^*$, *it holds that:*

$$\forall \alpha \textit{ occurring in } \delta,\ \alpha \notin (L\_ASSIGN^P(h) \cup L\_COMM^P(h)).$$

The following lemma ensures that, if the abstract transition system of a program satisfies the properties "*indirect_if*" and "*indirect_while*", then inside the scope of each indirect high flow no assignment to a low variable is performed.

Lemma 2. *Let* $P = \langle p, L, H\rangle$ *be a program. Suppose that* $A(p) \models indirect\_if \wedge indirect\_while$. *For each path:* $p \xrightarrow{\delta i \gamma j}{}^{\natural} p'$, *such that* $(i,j) \in \mathcal{A}, \delta, \gamma \in \mathcal{A}^*$ *and* $(i,j) \in IF^P(h)$ *or* $(i,j) \in WHILE^P(h)$ *it holds that:*

for each α *occurring in* γ, $\alpha \notin DEP^P(i)$.

The next lemma states that, if the abstract transition system of a program satisfies the property $direct \wedge indirect\_if \wedge indirect\_while$, then two executions, starting from two memories which agree on the value of the low variables, produce the same sequence of assignments to each low variable.

Lemma 3. *Let* $P = \langle p, L, H\rangle$ *be a program. Suppose that* $A(p) \models direct \wedge indirect\_if \wedge indirect\_while$. *Consider* $m_1, m_2 \in \mathcal{M}$ *such that* $m_1 =_L m_2$. $\forall x \in L$ *and for all two paths:*

$$\langle p, m_1\rangle \xrightarrow{\delta} \langle q, m_1'\rangle \not\longrightarrow \quad \textit{and} \quad \langle p, m_2\rangle \xrightarrow{\gamma} \langle \overline{q}, m_2'\rangle \not\longrightarrow$$

it holds that: $\delta \downarrow_x = \gamma \downarrow_x$, *where, given* $\delta \in \mathcal{A}^*$, $\delta \downarrow_x$ *is the projection of* δ *on the actions* α *such that either* $\alpha = i$ *and* $i : x := e$ *occurs in* P *or* $\alpha = i, j$ *and* $i : a?x$ *occurs in* P.

The following theorems hold:

Theorem 2. *Let $P = \langle p, L, H \rangle$ be a program.*

$$A(p) \models direct \ \wedge \ indirect\_if \ \wedge \ indirect\_while \quad \Rightarrow \quad P \ secure.$$

Proof Sketch. *Consider $m_1, m_2 \in \mathcal{M}$ such that:*

$$m_1 =_L m_2, \ \langle p, m_1 \rangle \xrightarrow{\delta} \langle q, m_1' \rangle \not\rightarrow \ and \ \langle p, m_2 \rangle \xrightarrow{\gamma} \langle \overline{q}, m_2' \rangle \not\rightarrow .$$

We have to prove that $m_1' =_L m_2'$. By Theorem 1, $A(p)$ contains the paths $p \xrightarrow{\delta}^{\natural} q$ and $p \xrightarrow{\gamma}^{\natural} \overline{q}$. By Lemma 3, for each $x \in L$ it is $\delta \downarrow_x = \gamma \downarrow_x$. The proof then follows by Lemma 1.

Theorem 3. *Let $P = \langle p, L, H \rangle$ be a program.*

$$A(p) \models no\_while \ \wedge \ no\_while\_in\_if \quad \Rightarrow \quad P \ has \ secure \ termination.$$

Proof Sketch. *From Theorem 1 and the above lemmas.*

5 Implementation

To evaluate the feasibility of the proposed method, we implemented it using the LOTOS Reduction tool (LR tool for short) developed by the authors [10]. The LR tool integrates the Concurrency Workbench of North Carolina (CWB-NC) [7,8] with the selective mu-calculus logic. The Concurrency Workbench of North Carolina (CWB-NC) is a verification environment including several different specification languages. The specifications can be checked for different equivalences, and different logics can be used to express properties. In the CWB-NC the verification of temporal logic formulae is based on model checking.

The LR tool allows verifying selective mu-calculus formulae on specifications defined using the Basic LOTOS specification language [6]. Basic LOTOS is a process algebra by means of which it is possible to describe the behavior of concurrent processes, concentrating on communications between processes. It includes commands for synchronous communication and parallelism. CWB-NC performs LOTOS programs by building a transition system representing the behavior of the program that is all possible execution paths.

The implementation of our methodology is as follows:

1. The program P to be checked against security is translated into a LOTOS program $LOTOS(P)$ representing only the skeleton of the program: values are substituted by security levels and only branch points and communication points are considered. The translation is straightforward. It ensures that the semantics of $LOTOS(P)$, as defined by the CWB-NC, is isomorphic to the abstract semantics of P as defined in Section 3.

2. The sets $ASSIGN, DEP, IF, \dots$ defined in Section 4 are built by statically examining the program P.
3. The security mu-calculus formulae defined in Section 4 are specialized for $LOTOS(P)$ with the actual action names.
4. Finally, the formulae are checked on $LOTOS(P)$.

For example, consider the program P_2 in Section 4.1, $LOTOS(P_2)$ is the following:

```
process P2:= Q0 |[(2_6)]| Q1
   where
   Q0 := 1; (tau; 2_6; exit [] tau; 3; exit) >> 4; 5; stop)
   Q1 := 2_6; stop
endproc
```

The actions of $LOTOS$(P2) coincides with the actions of P_2: there is an action for each label, if the label does not correspond to any communication primitive, and an action for each pair of labels corresponding to two matching sending/receiving events. They are composed by means of the sequentialization operator "`;`" and the choice operator "`[]`". The program specifies a process `P2` which is the parallel composition of two processes: `Q0` and `Q1`. The process `Q0` composes, with the sequentialization operator "`;`", the simple command labeled by 1, the `if` command and the simple commands labeled by 4 and 5. The the translation of the `if` command is

```
   (tau; 2_6; exit [] tau; 3; exit)
```

which means that, after a nondeterministic internal choice (`tau` action) obtained by using the choice operator "`[]`", the process can perform the actions `2_6`, or the action `3`. The action `2_6` is a communication between `Q0` and `Q1`. The process `exit` represents successful termination; it is used by the enabling (`>>`) operator: `T1 >> T2` represents the sequentialization between the two processes `T1` and `T2`. The process `stop` cannot perform any move. The process `Q1` can perform only the communication action `2_6` and then becomes the process `stop`.

The "*indirect_if*" property is expressed by the following selective mu-formula:

$$\nu Z.\Big([1]_\emptyset \Big([5]_{\{4\}}\, \mathtt{ff} \wedge (\langle 5\rangle_\emptyset\, \mathtt{tt} \Rightarrow \ \mathit{eventually}(4))\Big) \ \wedge \ Z\Big)$$

Figure 2 shows the user interface of the LR tool. We write the LOTOS specification of `P2` in the `LOTOS Program` box and the selective mu-calculus formula in the `Selective mu-calculus` box. The CWB-NC environment can be called by clicking the `CWB-NC` button. The translation of the formula to mu-calculus, which is the logic used in CWB-NC, is automatically performed by clicking the `SMC->MC` button).

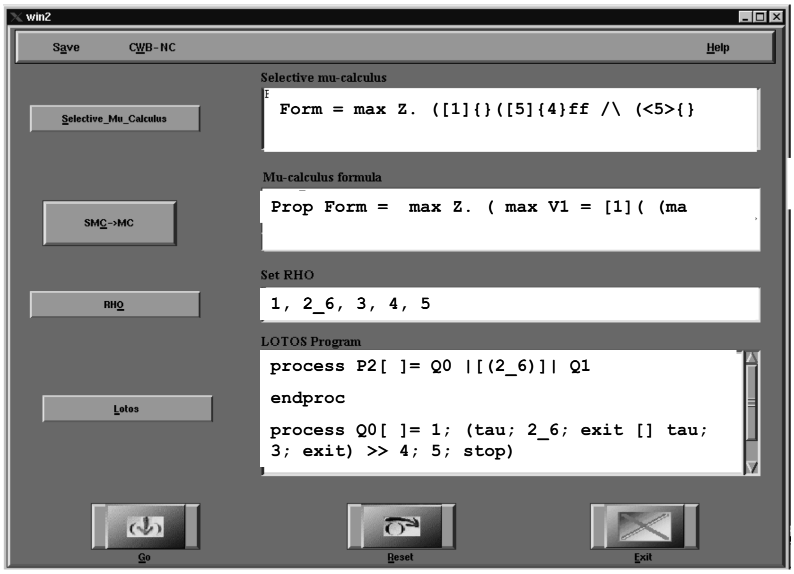

Fig. 2. The user interface of LR tool

Note that the translation of the formulae is in the format of the CWB-NC tool: a formula has the form `prop` $id = \phi$, and recursive formulae are given with the least (`min`) and greatest (`max`) fix-point operators.

The CWB-NC tool builds the transition system for `P2` which is isomorphic to the abstract transition system shown in Figure 1; this transition system can be used to check the property. It holds that `P2` satisfies φ (i.e., the "*indirect_if*" property).

6 Related work and Discussion

In [19] a comparative overview is given of the existing works on security in multi-threaded programs. The same work concentrates on the problems arising when synchronization is of concern. It proposes a secure type system for multi-threaded shared memory programs with synchronization. This type system certifies more programs than the previous works [2]. The method is compositional. With our approach we are able to certify more programs (for example the program P_2 of Section 4.1), at the price of a greater complexity. In fact model checking suffers of the state explosion problem, especially with parallel programs. The use of the selective mu-calculus [4] helps in partially reducing this problem: in fact this logic was defined to simplify the automaton to be model checked, where the simplification is driven by the formula to be checked.

In [21] a type system is presented which resolves similar problems for a multi-threaded language with shared memory.

The restriction to deterministic processes has been made to point out more clearly the main argument of the work, i.e. checking indirect flows in concurrent programs. Adding non determinism does not introduce new problems. As a future work we intend to remove the restriction and adapt the model to handle local and global nondeterminism.

An advantage of the abstract interpretation + model checking approach is flexibility: once defined a suitable abstraction, several properties can be checked using the corresponding formulae. As a future work, we are going to describe in selective mu-calculus the other security properties defined in the literature.

References

1. M. Abadi, A. Banerjee, N. Heintze, J.G. Riecke. A Core Calculus of dependency. Proceedings 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages Conference, San Antonio, Texas, USA, 1999, pp. 147-160.
2. G. R. Andrews, R. P. Reitman. An axiomatic approach to information flow in programs. ACM Transactions on programming languages and systems, 2(1), 1980, pp. 56-76.
3. J. Banatre, C. Bryce, D. L. Métayer. Compile-time detection of information flow in sequential programs. Proceedings European Symposium on Research in Computer Security, LNCS 875, Springer Verlag, 1994, pp. 55-73.
4. R. Barbuti, N. De Francesco, A. Santone, G. Vaglini. Selective mu-calculus and Formula-Based Equivalence of Transition Systems. Journal of Computer and System Sciences, 59(3), 1999. pp. 537-556.
5. R. Barbuti, C. Bernardeschi, N. De Francesco. Abstract Interpretation of Operational Semantics for Secure Information Flow. Information Processing Letters, 2002.
6. T. Bolognesi, E. Brinksma. Introduction to ISO Specification Language LOTOS. Comp. Networks and ISDN Systems, 14, 1987. 25-59.
7. R. Cleaveland, S. Sims. The NCSU Concurrency Workbench. In Proceedings of the Eighth International Conference on Computer-Aided Verification (CAV'96), Lecture Notes in Computer Science 1102, 1996. 394–397.
8. The Concurrency Workbench of North Carolina home page. URL http://www4.ncsu.edu/eos/users/r/rance/WWW/ncsu-cw.html.
9. P. Cousot, R. Cousot. Abstract interpretation frameworks. Journal of Logic and Computation, 2, 1992, pp. 511-547.
10. N. De Francesco, A. Santone. A Tool Supporting Efficient Model Checking of Concurrent Specifications. Microprocessors and Microsystems, 25(9-10), 2002. pp. 401-407.

11. D. E. Denning, P. J. Denning. Certification of programs for secure information flow. Communications of the ACM, 20(7), 1977, pp. 504-513.
12. N. Heintze, J.G. Riecke. The Slam Calculus: programming with Secrecy and Integrity. Proceedings 25th ACM Principles of Programming Languages Conference, San Diego, USA, 1998, pp. 365-377.
13. C.A.R. Hoare. Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs, NJ. 1985.
14. N. D. Jones, F. Nielson. Abstract interpretation: a semantic based tool for program analysis. in S. Abramsky, D.M. Gabbay, T.S.E. Maibaum(Eds.), Handbook of Logic in Computer Science , Vol. 4, Oxford University Press, Oxford, 1995, 527-636.
15. D. Kozen. Results on the propositional mu-calculus. Theoretical Computer Science, 27, 1983. pp. 333–354.
16. K.R.M. Leino, R. Joshi A semantic approach to secure information flow. Science of Computer Programming, 37(1), 2000.
17. M. Mizuno, D. A. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. Formal Aspects of Computing 4, 1992, pp. 727-754.
18. A. Sabelfeld, D. Sands. A PER model of secure information flow in sequential programs. Proceedings 8th European Symposium on Programming, ESOP'99, LNCS 1576, Springer-Verlag, 1999, pp. 40-58.
19. A. Sabelfeld, D. Sands. The impact of synchronization on secure information flow in concurrent programs. Proceedings Andrei Ershov 4th International Conference on Perspective of System Informatics, Novosibirsk, LNCS, Springer-Verlag, July 2001.
20. D. A. Schmidt. Data-flow analysis is model checking of abstract interpretations. Proc. 25th ACM Symp. Principles of Programming Languages, San Diego, 1998.
21. G. Smith. A New Type System for Secure Information Flow. Proc. 14th IEEE Computer Security Foundations Workshop (CSFW'01), pp. 115-125, Cape Breton, Nova Scotia, June 2001.
22. G. Smith, D. Volpano. Secure information flow in a multi-threaded imperative language. Proceedings 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages, San Diego California, 1998, pp. 1-10.
23. D. Volpano, G. Smith, C. Irvine. A sound type system for secure flow analysis. Journal of Computer Security, 4(3), 1996, pp. 167-187.
24. D. Volpano, G. Smith. Eliminating covert flows with minimum typing. Proceedings 10th IEEE Computer Security Security Foundation Workshop, June 1997, pp. 156-168.