

Automaton-based Non-interference Monitoring of Concurrent Programs ¹

Gurvan Le Guernic

April 2007

¹Technical Report Nr. 2007-1 (version of April 2007). Department of Computing and Information Sciences, College of Engineering, Kansas State University (234 Nichols Hall, Manhattan, KS 66506, USA).

Contents

1	Introduction	2
2	Outline	2
2.1	The Language: Syntax	2
2.2	The Language: Standard Semantics	3
2.3	Monitoring Principles	6
3	The Monitoring Automaton	10
4	The Monitoring Semantics	17
5	Example of monitored execution	20
6	Properties of the Monitoring Mechanism	22
6.1	Soundness	22
6.2	Partial Transparency	23
7	Conclusion	24
A	Proofs	26
A.1	Soundness	26
A.1.1	Definitions	26
A.1.2	Lemmas	28
A.2	Partial Transparency	45
A.2.1	Definitions	45
A.2.2	Lemmas	46
	References	51

Abstract

Earlier work [LGBJS06] presents an automaton-based non-interference monitoring mechanism for sequential programs. This technical report extends this work to a concurrent setting. Monitored programs are constituted of a set of threads running in parallel. Those threads run programs equivalent to those of [LGBJS06] except for the inclusion of a synchronization command. The monitoring mechanism is still based on a security automaton and on a combination of dynamic and static analyses. As in [LGBJS06], the monitoring semantics sends abstractions of program events to the automaton, which uses the abstractions to track information flows and to control the execution by forbidding or editing dangerous actions. All monitored executions are proved to be non-interfering (soundness).

1 Introduction

Similarly to [LGBJS06], the monitoring mechanism aims at forbidding the value of the private inputs to influence the sequence of public outputs generated during an execution. In [LGBJS06], programs are deterministic and therefore always output the same sequence of public values. This is not anymore the case in the multi-threaded setting. Hence, the monitor will not ensure that any execution started with the same public inputs generates the same public output sequence — as this is the case in [LGBJS06]. Instead, it ensures that, for any execution started with the same public inputs, there exists a thread interleaving which generates a sequence of public output having, as prefix, the sequence of public outputs generated so far by the currently monitored execution.

The language used in this report is equivalent the previous one except for the addition of a synchronization command similar to the one used in [O’H04]. The main difference between those two works lies in the semantics used. [LGBJS06] uses a big-step (or natural [Kah87]) semantics. Therefore it only takes into account terminating executions. Use of a big-step semantics is not possible in the concurrent settings. Hence, this report uses a small-step semantics. This modification allows the work presented in this report to take into account non-terminating executions.

Section 2 starts by presenting the language used: its syntax and semantics. It introduces the principles used by the monitoring mechanism presented in this report. Section 3 defines formally the security automaton which is at the heart of the monitoring mechanism. The following section characterizes the monitoring semantics which links the monitoring automaton previously presented with the standard semantics given in Section 2.2. Then, comes an example in Section 5 which examines the evolution of the monitoring automaton during an example execution. Finally, before concluding in Section 7, soundness is proved in Section 6.

2 Outline

2.1 The Language: Syntax

A concurrent program is a pool of threads (Θ); each of which contains a sequential program. Such a pool is formally defined as a partial function from integers to sequential programs. $\Theta(i)$ is the i^{th} sequential program of the pool. The grammar of the sequential programs is given in Figure 1. In order to gain in simplicity while describing the semantics, the grammar is split into four different blocks.

A sequential program ($\langle prog \rangle$) is either a sequence of sequential programs or a command ($\langle com \rangle$). Actions ($\langle action \rangle$) and control commands ($\langle control \rangle$) are the only two types of commands. An action is either an assignment of the value of an expression ($\langle expr \rangle$) to a variable ($\langle ident \rangle$), an output of the value of an expression, or a skip (which does nothing). A control command is either: a conditional executing one program — out of two — depending on the value of an expression, a loop executing a program as long as the value of a given expression remains `true`, or a synchronization command (**with** $\langle ident-set \rangle$ **when** $\langle expr \rangle$ **do** $\langle prog \rangle$ **done**). The latter is executed — i.e. the program it encompasses is executed — only if no other thread owns one of the locks of the given set of variables ($\langle ident-set \rangle$) and the value of the expression ($\langle expr \rangle$) is `true`. Otherwise, the thread executing this synchronization command is blocked — i.e. it can not execute anything as long as the conditions are not fulfilled.

The syntax of the synchronization command comes from [Hoa72, O’H04, Bro04]. It has been chosen for its ability to encode easily lots of different synchronization constructions. It allows the monitoring mechanism to deal with only one synchronization construction while keeping the language expressive

```

<action> ::= <ident> := <expr>
          | output <expr>
          | skip
<control> ::= if <expr> then <prog> else <prog> end
           | while <expr> do <prog> done
           | with <ident-set> when <expr> do <prog> done
<com> ::= <action> | <control>
<prog> ::= <prog> ; <prog> | <com>

```

Figure 1: Grammar of sequential programs in the concurrent setting

enough with regard to synchronization. Table 1 shows how to encode the Java’s “synchronized” command and standard semaphores using the synchronization command used in this report. The synchro-

```

synchronized(x){P} ≡ with x when true do P done
P(s) ≡ with s when s > 0 do s := s - 1 done
V(s) ≡ with s when true do s := s + 1 done

```

Table 1: Encoding of different synchronization commands

nization command complexifies the work of the monitor. However, when the expression enclosed in the synchronization command is “true”, the monitor can allow it to appear at some program points it would not allow otherwise. One important thing to be noticed in table 1 is that it is not infrequent for the expression in a synchronization command to simply be “true”. This explains why this special case is taken into account in the monitoring mechanism exposed in this report.

2.2 The Language: Standard Semantics

During the execution, a thread does not necessarily contain a program as defined in Figure 1. In fact, it is executing an *execution statement* ($\langle exec-stat \rangle$) which is either empty (\emptyset), a usual sequential program ($\langle prog \rangle$), an *execution statement* followed by a sequential program ($\langle exec-stat \rangle ; \langle prog \rangle$) or a *locked statement* ($\odot \langle ident-set \rangle [\langle exec-stat \rangle]$) carrying additional information about the locks owned by this particular thread. The grammar of *execution statements* is given in Figure 2 and is based on the grammar of sequential programs (Figure 1). A *locked statement* ($\odot \bar{x} [P]$) is obtained after the execution of a synchronization command (**with** \bar{x} **when** e **do** P **done**). When a thread $\Theta(i)$ executes a synchronization command, it acquires the locks of the variables given in parameter (\bar{x}). This information is registered somewhere in the program state in order to be able to forbid other threads to acquire those locks as long as the thread $\Theta(i)$ has not released them. However, the thread $\Theta(i)$ is still allowed to execute synchronization commands on those locks — i.e. acquire again the same locks. The information needed to allow it is not registered in the program state. The locks a given thread is allowed to acquire again are registered in its sequential

$$\begin{aligned}
\langle exec-stat \rangle & ::= \emptyset \\
& | \langle prog \rangle \\
& | \langle exec-stat \rangle ; \langle prog \rangle \\
& | \odot \langle ident-set \rangle [\langle exec-stat \rangle]
\end{aligned}$$

Figure 2: Grammar of sequential programs under execution

program itself by the substitution of the synchronization command just evaluated by a locked statement. This locked statement contains the variables whose locks have just been acquired and the program to be executed while holding those locks.

Due to the standard semantics used, a statement under execution (an *execution statement*) is a set — potentially empty — of interwove *locked statements* followed by a usual sequential program. Formally, an *execution statement* belongs to the language defined by the following expression:

$$\left(\odot \langle ident-set \rangle [\] \right)^n \left(\langle prog \rangle + \emptyset \right) \left(\left(; \langle prog \rangle \right)^? 1 \right)^n \left(; \langle prog \rangle \right)^?$$

Concurrent program semantics. A concurrent program under execution is a pool of execution statements. One step of execution of a concurrent program consists in taking one execution step for one of the execution statements of the thread pool. There is no constraint on which thread has to be evaluated. The scheduler used for the thread interleaving is a purely random one. After each step of execution, the next thread to execute is selected randomly among those which can take an execution step. Such a scheduler avoids timing covert channels which are based on the hypothesis that the attacker is able to guess accurately in which order the scheduler will execute the different threads. When dealing with schedulers which are not purely random, the solutions proposed in [RHNS06] should be applicable.

The semantics of unmonitored executions of concurrent programs (pool of threads) is given by the following rule:

$$\frac{t \in \text{dom}(\Theta) \quad \langle \zeta \vdash \Theta(t) \rangle \xrightarrow{o}_s \langle \zeta' \vdash S'_t \rangle}{\langle \zeta \vdash \Theta \rangle \xrightarrow{o}_s \langle \zeta' \vdash \Theta[t \mapsto S'_t] \rangle} \quad (\text{E}_s\text{-CONCUR})$$

Execution statement semantics. The semantics of concurrent programs is based on the semantics of *execution statements* which is described in Figure 3. This latter semantics is based on rules written:

$$\langle \zeta \vdash S \rangle \xrightarrow{o}_s \langle \zeta' \vdash S' \rangle$$

Those rules mean that in the execution state ζ statement S evaluates to statement S' yielding state ζ' and output sequence o . Let \mathbb{X} be the domain of variable identifiers and \mathbb{D} be the semantic domain of values. An execution state ζ is a pair (σ, λ) composed of a value store $\sigma (\mathbb{X} \rightarrow \mathbb{D})$ and a lock set $\lambda (2^{\mathbb{X}})$. σ maps variable identifiers — or names — to their respective current value. The definition of value stores is extended to expressions, so that $\sigma(e)$ is the value of the expression e in a program state whose value store is σ . λ is a set of variable identifiers. It corresponds to the set of variables whose lock is currently

$\langle\langle \sigma, \lambda \vdash x := e \rangle\rangle \xrightarrow{\epsilon}_s \langle\langle \sigma[x \mapsto \sigma(e)], \lambda \vdash \emptyset \rangle\rangle$	(E _s -ASSIGN)
$\langle\langle \zeta \vdash \mathbf{output} \ e \rangle\rangle \xrightarrow{\sigma(e)}_s \langle\langle \zeta \vdash \emptyset \rangle\rangle$	(E _s -OUTPUT)
$\langle\langle \zeta \vdash \mathbf{skip} \rangle\rangle \xrightarrow{\epsilon}_s \langle\langle \zeta \vdash \emptyset \rangle\rangle$	(E _s -SKIP)
$\frac{\sigma(e) = v}{\langle\langle \sigma, \lambda \vdash \mathbf{if} \ e \ \mathbf{then} \ S^{\mathbf{true}} \ \mathbf{else} \ S^{\mathbf{false}} \ \mathbf{end} \rangle\rangle \xrightarrow{\epsilon}_s \langle\langle \sigma, \lambda \vdash S^v \rangle\rangle}$	(E _s -IF)
$\frac{\sigma(e) = \mathbf{true}}{\langle\langle \sigma, \lambda \vdash \mathbf{while} \ e \ \mathbf{do} \ S^l \ \mathbf{done} \rangle\rangle \xrightarrow{\epsilon}_s \langle\langle \sigma, \lambda \vdash S^l ; \mathbf{while} \ e \ \mathbf{do} \ S^l \ \mathbf{done} \rangle\rangle}$	(E _s -WHILE _{true})
$\frac{\sigma(e) = \mathbf{false}}{\langle\langle \sigma, \lambda \vdash \mathbf{while} \ e \ \mathbf{do} \ S^l \ \mathbf{done} \rangle\rangle \xrightarrow{\epsilon}_s \langle\langle \sigma, \lambda \vdash \mathbf{skip} \rangle\rangle}$	(E _s -WHILE _{false})
$\frac{\bar{x} \cap \lambda = \emptyset \quad \sigma(e) = \mathbf{true}}{\langle\langle \sigma, \lambda \vdash \mathbf{with} \ \bar{x} \ \mathbf{when} \ e \ \mathbf{do} \ S^s \ \mathbf{done} \rangle\rangle \xrightarrow{\epsilon}_s \langle\langle \sigma, \lambda \cup \bar{x} \vdash \odot \bar{x}[S^s] \rangle\rangle}$	(E _s -WITH)
$\frac{}{\langle\langle \zeta \vdash \emptyset ; S^t \rangle\rangle \xrightarrow{\epsilon}_s \langle\langle \zeta \vdash S^t \rangle\rangle}$	(E _s -SEQ _{action})
$\frac{\langle\langle \zeta \vdash S^h \rangle\rangle \xrightarrow{o}_s \langle\langle \zeta' \vdash S^{h'} \rangle\rangle}{\langle\langle \zeta \vdash S^h ; S^t \rangle\rangle \xrightarrow{o}_s \langle\langle \zeta' \vdash S^{h'} ; S^t \rangle\rangle}$	(E _s -SEQ _{branch})
<hr style="border: 0.5px solid black;"/>	
$\frac{}{\langle\langle \sigma, \lambda \vdash \odot \bar{x}[\emptyset] \rangle\rangle \xrightarrow{\epsilon}_s \langle\langle \sigma, \lambda - \bar{x} \vdash \emptyset \rangle\rangle}$	(E _s -LOCKED _{action})
$\frac{\langle\langle \sigma, \lambda - \bar{x} \vdash S^s \rangle\rangle \xrightarrow{o}_s \langle\langle \sigma', \lambda' \vdash S^{s'} \rangle\rangle}{\langle\langle \sigma, \lambda \vdash \odot \bar{x}[S^s] \rangle\rangle \xrightarrow{o}_s \langle\langle \sigma', \lambda' \cup \bar{x} \vdash \odot \bar{x}[S^{s'}] \rangle\rangle}$	(E _s -LOCKED _{seq})

Figure 3: Output semantics of sequential programs in the concurrent setting

owned by a thread. An output sequence is a word in \mathbb{D}^* . It is either an empty sequence (written ϵ), a single value (for example, $\sigma(e)$) or the concatenation of two other sequences (written $o_1 o_2$).

As shown by the rule (E_s-WITH), if the prerequisites allow it, the execution of a synchronization command yields an *execution statement* $\odot \bar{x}[P]$ where \bar{x} is a set of variable identifiers. This rule states that if the conditions hold, a synchronization command is replaced by the program P it includes. The current thread will now own the locks of the variables belonging to \bar{x} .

2.3 Monitoring Principles

The mechanism developed in this report is a monitor aiming at enforcing the confidentiality of private data manipulated by any concurrent program. In order to achieve this goal, the monitoring mechanism is designed to enforce a stronger property based on the notion of non-interference. The informal definition of this notion [GM82] states that a program is “safe” if its private inputs have no influence on the observable behavior of the program. This definition is stated at the level of the whole program. Unfortunately, a monitor acts at the level of an execution and not of a whole program.

The remaining of the current section starts by giving some definitions which are then used to define formally the notion of non-interference for executions. Subsequently, it provides a short description of the way the monitor works. Finally, by commenting upon two examples of concurrent programs, it illustrates some of the difficulties of monitoring non-interference and the solutions adopted.

A processing spans over a set of executions. In a sequential program, an execution is entirely defined by a program and an initial state. In a concurrent setting, this is not the case if the term “execution” designates an entity defined by a program, an initial state and a precise output. In other terms, an execution is considered to always generate the same output. In this report, an entity defined by a program and an initial state is called a “processing”. A processing can generate different outputs depending on the thread interleaving.

Definition 2.1 (Execution). An *execution* is the evaluation of precise program in a precise initial state and following a precise thread interleaving. An execution always generates the same output.

Definition 2.2 (Processing). A *processing* is the evaluation of a precise program in a precise initial state. A given processing can generate different outputs depending on the thread interleaving.

What is a non-interfering concurrent execution? An analysis is said to be sound with regard to a given property. In this work, this property is non-interference. In order to prove soundness in section 6.1 and state formally the effect of the monitor, it is then required to first define precisely the non-interference property in our concurrent setting. A program P is said to be non-interfering if and only if its private inputs have no influence on the observable behavior of the program. This is usually characterized by the following property: any two *processing* of P started with the same public inputs — but potentially different private inputs — have the exact same observable behavior. It is then required to compare the observable behavior of different processings. In this report, the observable behavior is the output sequence generated. The output sequence generated by the evaluation of a given sequential program started in a given state is unique. However, the output sequence generated by the evaluation of a given concurrent program — a pool of sequential programs — started in a given state is not. For multi-threaded programs, the output sequence generated by the processing of a given program started in a given state depend on the thread interleaving which occurred during the evaluation. The definition of the observable behavior

of a processing must then take into account all those possible output sequences. Additionally, as it is also desired to take into account non-terminating executions, our definition of observable behavior must also take into consideration the observable behavior of a processing at any step of its evaluation, not only the final one. Definition 2.3 states that the observable behavior — or output sequences — of the processing of a given concurrent program started in a given state is the set of all prefixes of any output sequence which can be obtained by any thread interleaving.

Definition 2.3 (Observable behavior: $O[[\Theta]]_s X$).

For all semantics s , concurrent programs Θ , and states X :

$$O[[\Theta]]_s X = \{ o \mid \exists X', \Theta' : \langle X \vdash \Theta \rangle \xrightarrow{o}_s^* \langle X' \vdash \Theta' \rangle \}$$

The monitoring mechanism does not determine if a program is or is not non-interfering, but if a precise execution is or is not non-interfering. It has then to determine if, by looking at the current output sequence, it is possible to differentiate the current *execution* from *processings* of the same program started with the same public inputs but potentially different private inputs. This is possible only if the output sequence, the observable behavior, of the current execution — which follows a precise thread interleaving — of the program P can not be produced, for any thread interleaving, with an other initial state X_2 having the same public inputs. In other words, only if there exists an initial state X_2 such that the output sequence of the current execution does not belongs to $O[[\Theta]]_s X_2$. Definition 2.5 characterizes a non-interfering execution as an execution whose output sequence belongs to the observable behavior of any processing of the same program and an initial state low equivalent (Definition 2.4) to the initial state of the current execution.

Definition 2.4 (Low Equivalent States).

Two states X_1 , respectively X_2 , containing the value stores σ_1 , respectively σ_2 , are low equivalent with regard to a set of variables V , written $X_1 \stackrel{V}{=} X_2$, if and only if the value of any variable belonging to V is the same in σ_1 and σ_2 . This is formally stated as follows:

$$X_1 \stackrel{V}{=} X_2 \iff \forall x \in V : \sigma_1(x) = \sigma_2(x)$$

Definition 2.5 (Non-interfering Execution).

The execution, with a semantics s , of a concurrent program Θ started in the initial state X_1 and whose output sequence is o is said to be non-interfering, written $ni(\Theta, s, X_1, o)$, if and only if:

$$\forall X_2 : X_1 \stackrel{S(\Theta)^c}{=} X_2 \Rightarrow o \in O[[\Theta]]_s X_2$$

How does the monitor enforce non-interference? As in [LGBJS06], the monitoring mechanism is separated into two parts. The first element, described in Section 4, is a special semantics which delegates the main job to a security automaton described in Section 3. The purpose of the special semantics is to select and abstract the important events, with regard to non-interference, which occur during the execution. The abstractions of those events are then sent to the security automaton. For each input received, the automaton sends back to the semantics an output. Depending on those outputs received, the special semantics modifies the normal execution of the program. The security automaton is in charge of keeping track of the variables whose value carries *variety* — i.e. their value is influenced by the values of the private inputs — and keeping track of *variety* in the context of execution — i.e. the program counter — of each thread independently.

With regard to synchronization commands, the solution adopted follows the standard solution found in the literature [Sab01, ?]. Synchronizations inside the branch of a conditional whose branching condition is influenced by the private inputs are forbidden. However, it is impossible to stop the execution when encountering a synchronization command inside a conditional whose branching condition carries variety. Doing so would create a new covert channel leaking information to low-level users if a public output was supposed to occur later in the execution. One solution would consist in ignoring synchronization commands in a context carrying variety. The drawback of this solution is to suppress synchronizations that the programmer deemed wise to include. The solution adopted in this monitor is to force any synchronization to occur outside any conditional whose branching expression carries variety. Whenever the monitoring mechanism has to evaluate such a conditional, it executes the locking part of any synchronization command appearing in any branch of the conditional.

The monitoring mechanism by the example. As with the monitor described in [LGBJS06], the security automaton is unaware of the precise values of the variables. Consequently, it sometimes concludes that variety is carried to a variable when it is not the case. A typical example is when a variable is assigned the same value in all branches of a conditional whose condition carries variety. In such a case, the value of the assigned variable is always the same at the end of the conditional. However, this is out of the scope of the proposed monitor and of the majority of works on non-interference.

Synchronization commands are inference prone. Figure 4 contains the code of a concurrent program. It is composed of two threads, has a single private input (h) and uses an internal variable v which is never output. The first thread (Figure 4(a)) takes the lock of the variable v and then output “a” followed by “b” before releasing the lock. The second thread (Figure 4(b)) outputs “c”; then, if the private input h is true, it tries to acquire the lock of v and release it immediately; finally, it outputs “d”. What can

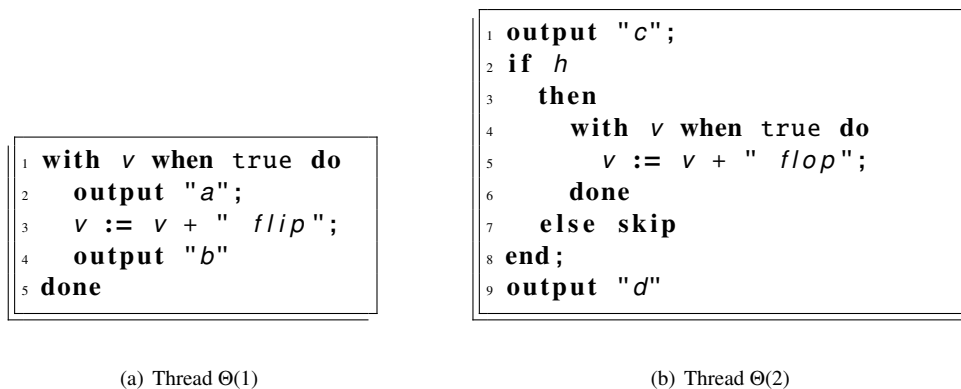


Figure 4: Example of interference due to a lock

be deduced by a low-level user if it sees the output sequence “a c d b”? From the code of thread $\Theta(1)$, it is possible to deduce that the first thread owns the lock of v at least from the time the program outputs “a” until it outputs “b”. Consequently, as “c” and “d” are output between “a” and “b”, thread $\Theta(2)$ has evaluated every command between the two outputs of “c” and “d” without needing to acquire the lock of v . It is then easy, by looking at the code of $\Theta(2)$, to deduce that the private input h is false. The “bad” flow from h to the output sequence is due to the synchronization commands on v .

For some output sequences, as “a c d b”, it is possible to deduce that the synchronization command in $\Theta(2)$ has not been executed, which in turn allows to deduce the value of h . It is worthwhile to notice that, if not evaluating the synchronization command in $\Theta(2)$ can lead to the creation of a “bad” flow, this is not the case for the evaluation of the synchronization command in $\Theta(2)$. If h is `true`, it is impossible to deduce it from the output sequence. Indeed, whatever output sequence is generated when h is `true`, there exists a thread interleaving when h is `false` generating the same output sequence.

To prevent the “bad” flow presented above, it is not a good idea to simply stop the execution whenever a synchronization command has to be executed inside a conditional whose condition carries variety. In the example presented, the synchronization command in $\Theta(2)$ is *not* executed. Stopping the program each time the program has to evaluate a conditional, whose condition carries variety and which contains a synchronization command, is not really appealing either. It does not either seem really wise to simply ignore any synchronization command inside a conditional whose condition carries variety. There may be a good reason for having one there: in the previous example, avoiding a concurrent write access to the variable v . The solution adopted by the monitoring mechanism presented in this report consists in acquiring all the locks of any synchronization command appearing in any branch of a conditional whose condition carries variety before evaluating such a conditional. This mechanism prevents the program to output “a c d b” even if h is `false`.

Be cagey with newsmongers. Figure 5(a) contains the code of a Very Important Program (VIP) which has a single private input (h) and is run concurrently with the program in Figure 5(b). This latter program is a newsmonger which outputs indefinitely and as fast as it can some of the variables manipulated by the VIP (x and y). By itself, the VIP is not leaking any secret information. The main reason being that it does not output anything. This program only sets x to 0, then y to 0 and finally resets both to 1. However, depending on the value of its private input, it resets first x or y to 1. The interference

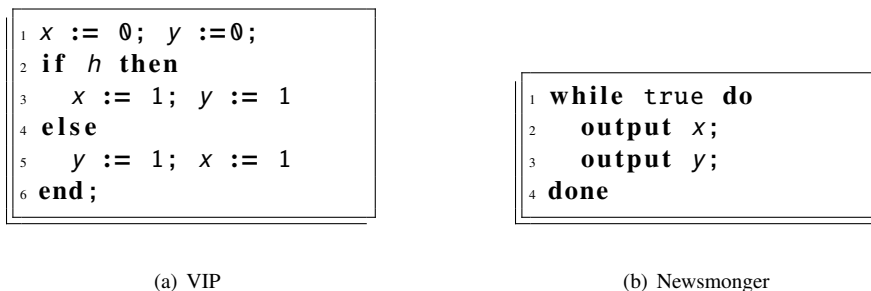


Figure 5: Example of interference due to concurrent access

comes from the newsmonger. If it is lucky enough, and the VIP unlucky enough, the scheduler will let the newsmonger take at least two steps, and so outputting x and y , while the VIP is in the middle of resetting those variables to 1. It is then easy, depending on which one of x and y has been reset to 1 first, to deduce the value of the private input h . The newsmonger is able to filch the value of the private input of an unmonitored execution of the VIP. Less intuitively, it is also possible, if the scheduler gives a high priority to the newsmonger, to deduce the value of h if the execution is supervised by a monitor which takes into account indirect flows created by an assignment only when this assignment is evaluated. The reason is that, in between the two assignments resetting x and y to 1, those two variables do not have the

same security level. Consequently, such a “bad” monitor does not act the same way for the commands “**output** x ” and “**output** y ” if only one of x and y has been reset to 1.

The monitoring mechanism proposed in this report takes into account an indirect flow as soon as the conditional which is the source of this flow is executed. For any execution of the VIP, this means that x and y get a high security level at the same time — when the conditional branching on h is first evaluated. Therefore, the monitor has the same behavior for the two commands “**output** x ” and “**output** y ”. The news monger is then unable to filch the VIP’s secret.

Section 5 follows precisely the behavior of the monitoring mechanism on an example. Before that, the next two sections give a formal definition of the monitoring mechanism: first of the security automaton used, and then of the special semantics communicating with this automaton.

3 The Monitoring Automaton

This section describes the security automaton used in the monitoring mechanism. It is in charge of tracking the information flows and controlling the execution in order to enforce non-interference. The semantics described in section 4 sends abstractions of the execution events to this automaton. In turn, the monitoring automaton sends back directions to the semantics to control the execution, thus *enforcing* non-interference.

Notations. For any set \mathbb{S} , let $2^{\mathbb{S}}$ be the power set of \mathbb{S} . For any set \mathbb{A} , let \mathbb{A}^* be the set of all strings over the alphabet \mathbb{A} . For any sequential program P whose variables belongs to \mathbb{X} , let the set of private input variables be $\mathcal{S}(P)$ ($\mathcal{S}(P) \subseteq \mathbb{X}$).

The monitoring automaton is formally defined using multisets among others. The following defines notations and operations over multisets as used in this report¹.

Within set theory, a multiset can be formally defined as a pair (A, m) where A is some set and $m : A \rightarrow \mathbb{N}$ is a function from A to the set \mathbb{N} of [...] natural numbers. The set A is called the *underlying set of elements*. For each a in A the *multiplicity* (that is, number of occurrences) of a is the number $m(a)$.

In the work proposed here, multisets are only represented by there *underlying function*. In the following, if m denotes a multiset — which is known from the context — it is the multiset (\mathbb{M}, m) where \mathbb{M} is the domain of m — written $\text{dom}(m)$. In consequence, m must be total over the *underlying set of elements* of the bag. However, for simplicity reasons, even if x does not belongs to the domain of the underlying function m , $m(x)$ is considered to be defined and equal to 0. The empty multiset, written \emptyset , is the multiset whose underlying function has an empty domain.

¹The definitions are based on those that can be found on <http://en.wikipedia.org/wiki/Multiset>.

Operations on multisets are defined as follows. For any multisets m, n and f :

$$\begin{aligned}
m \uplus n &= f \text{ where } \text{dom}(f) = \text{dom}(m) \cup \text{dom}(n) \text{ and } f(x) = m(x) + n(x) \\
m \cup n &= f \text{ where } \text{dom}(f) = \text{dom}(m) \cup \text{dom}(n) \text{ and } f(x) = \max(m(x), n(x)) \\
m \cap n &= f \text{ where } \text{dom}(f) = \text{dom}(m) \cap \text{dom}(n) \text{ and } f(x) = \min(m(x), n(x)) \\
m \setminus n &= f \text{ where } f(x) = \max(0, m(x) - n(x)) \\
m = n &\equiv \forall x \in (\text{dom}(m) \cup \text{dom}(n)) : m(x) = n(x) \\
m \subseteq n &\equiv \forall x \in \text{dom}(m) : m(x) \leq n(x)
\end{aligned}$$

If one of those operations involve a multiset and a set \mathbb{S} , this set is considered as the multiset whose *underlying set of elements* is \mathbb{S} and whose elements have a multiplicity of 1.

Formal definition of the monitoring automaton for sequential programs. Preceding sections present the monitoring mechanism as if it is using a single monitoring automaton. This is not exactly true, neither false. The monitoring automaton of concurrent programs is defined by first defining a monitoring automaton for sequential programs. A concurrent program is a pool of threads each one of which contains a single sequential program. Each time the “automaton” of the concurrent program Θ receives an input ϕ , generated by an evaluation step in a sequential program $\Theta(i)$, a state for the monitoring automaton of $\Theta(i)$ is extracted from the state of the monitoring automaton of Θ . Then, the monitoring automaton for sequential programs take a transition on input ϕ generating output ψ . Finally, the state of the monitoring automaton of Θ is updated using the new state returned by the transition of the monitoring automaton for sequential programs and the output ψ is sent back to the monitoring semantics.

The automaton enforcing non-interference for the sequential program P is the tuple $\mathcal{A}(P) = (Q, \Phi, \Psi, \delta, q_0)$ where:

- Q is a set of states ($Q = 2^{\mathbb{X}} \times (\mathbb{X} \rightarrow \mathbb{N}) \times 2^{\mathbb{X}} \times \{\top, \perp\}^*$);
- Φ is the input alphabet, constituted of abstractions of a subset of program events, specified below;
- Ψ is the output alphabet, constituted of execution controlling commands, specified below;
- δ is a transition function ($Q \times \Phi \rightarrow \Psi \times Q$);
- q_0 , an element of Q , is the start state ($q_0 = (\mathcal{S}(P), \emptyset, \emptyset, \epsilon)$).

The automaton states. An automaton state is a quadruplet (V, W, L, w) composed of two sets (V and L) of variables belonging to \mathbb{X} , a multiset (W) of variables belonging to \mathbb{X} and a word (w) belonging to a language whose alphabet is $\{\top, \perp\}$. At any step of the execution, V contains all the variables which *may* carry variety — i.e. whose values *may* have been influenced by the initial values of the secret input variables ($\mathcal{S}(P)$). L contains the variables whose lock *may* be owned by a thread at an “equivalent” step of an execution of the same program started with the same public inputs but potentially different private inputs. L is the part of the automaton state which protects secret data against attacks similar to the one described in the example about synchronization commands on page 8. W contains at least once each pair of variables whose difference in the order of assignment may leak secret information. W is the part of the automaton state which protects secret data against attacks similar to the one described in the newsmonger’s example on page 9. w , called a *branching context*, tracks variety in the context of the

execution with regard to previous branching commands. The context consists in the level of *variety* the conditions of the previous, but still active, branching commands. If w contains \top , then the statement executed was initially a branch of a conditional whose test may have been influenced by the initial values of $S(P)$. Hence this statement may not be executed with a different choice of initial values for the private input variables ($S(P)$).

Automaton inputs. The input alphabet of the automaton (Φ) contains abstractions of the events, interesting for non-interference monitoring, that can occur during an execution. The alphabet Φ consists of the following:

“**branch**(λ, e, P^e, P^u)” is generated each time a conditional is evaluated. λ is the set of variables whose lock is currently owned by a thread, e is the expression whose value determines the branch which is executed, P^e is the branch which will be executed, and P^u is the branch which will not be executed. For example, before evaluation of “**if** $x > 10$ **then** S_1 **else** S_2 **end**” with $((\sigma, \lambda), q)$ being the monitored execution state, the input “**branch**($\lambda, x > 10, S_1, S_2$)” is sent to the monitoring automaton if x is greater than 10.

“**merge**(P^e, P^u)” is generated each time a conditional has been fully evaluated. P^e is the branch which has been executed, and P^u is the branch which has not been executed. For example, after evaluation of “**if** $x > 10$ **then** S_1 **else** S_2 **end**”, the input “**merge**(S_1, S_2)” is sent to the monitoring automaton.

“**sync**(\bar{x}, e)” is generated before any synchronization command. For example, **with** x, b **when** b **do** S^s **done** generates the input “**sync**($\{x, b\}, b$)”.

any atomic action of the language (assignment, skip or output statement). Any action which has to be evaluated is first sent to the automaton for validation before its execution.

Automaton outputs. The automaton outputs control the behavior of the monitoring semantics. They are sent back from the automaton to the monitoring semantics in order to control the execution. The output alphabet (Ψ) is composed of the following:

“**OK**” is used whenever the monitoring automaton allows an execution step that it could have altered or denied.

“**NO**” is used whenever the monitoring automaton forbids the execution of an action. This action is simply skipped by the monitored execution.

any atomic action of the language. This is the answer of the monitoring automaton whenever another action than the current one has to be executed.

Compared to [LGBJS06], the output “**ACK**” has disappeared. The reason is that the automaton can block the execution on any non-action commands. In [LGBJS06], some inputs do not require intervention of the automaton on the execution. Their only goal is to help the automaton track useful information. There is not anymore such automaton inputs for this report.

Automaton transitions. Figure 6 specifies the transition function of the automaton. A transition rule is written:

$$(q, \phi) \xrightarrow{\psi} q'$$

It reads as follows: in the state q , on reception of the input ϕ , the automaton moves to state q' and outputs ψ . Let $FV(e)$ be the set of variables occurring in e . For example, $FV(x + y)$ returns the set $\{x, y\}$.

Let $modify(S)$ be the set of all variables whose value may be modified by an execution of S . This function is used in order to take into account the implicit indirect flows created when a conditional whose condition carries variety is evaluated. A formal definition of this function follows:

$$\begin{aligned} modify(x := e) &= \{x\} \\ modify(\mathbf{output} \ e) &= \emptyset \\ modify(\mathbf{skip}) &= \emptyset \\ modify(S_1 ; S_2) &= modify(S_1) \cup modify(S_2) \\ modify(\mathbf{if} \ e \ \mathbf{then} \ S_1 \ \mathbf{else} \ S_2 \ \mathbf{end}) &= modify(S_1) \cup modify(S_2) \\ modify(\mathbf{while} \ e \ \mathbf{do} \ S \ \mathbf{done}) &= modify(S) \\ modify(\mathbf{with} \ \bar{x} \ \mathbf{when} \ e \ \mathbf{do} \ S \ \mathbf{done}) &= modify(S) \end{aligned}$$

Let $needs(S)$ be the set of all variables whose lock may required in order to execute S . This function is used by the monitoring automaton in order to “virtually” acquire all the locks which may be needed for the complete evaluation of a conditional whose condition carries variety. Those locks are not acquired by the current thread. They will be only when their corresponding synchronization command will be executed. However, the monitoring automaton registers them in his state. Before allowing the evaluation of a conditional whose condition carries variety, the automaton checks that there is no needed lock which is already registered in the automaton state for an other thread. This is done in order to make sure that the evaluation of the branch designated by the condition — which carries variety — can not be stopped because of a needed lock which would be owned by an other thread. A formal definition of $needs(S)$ follows:

$$\begin{aligned} needs(x := e) &= \emptyset \\ needs(\mathbf{output} \ e) &= \emptyset \\ needs(\mathbf{skip}) &= \emptyset \\ needs(S_1 ; S_2) &= needs(S_1) \cup needs(S_2) \\ needs(\mathbf{if} \ e \ \mathbf{then} \ S_1 \ \mathbf{else} \ S_2 \ \mathbf{end}) &= needs(S_1) \cup needs(S_2) \\ needs(\mathbf{while} \ e \ \mathbf{do} \ S \ \mathbf{done}) &= needs(S) \\ needs(\mathbf{with} \ \bar{x} \ \mathbf{when} \ e \ \mathbf{do} \ S \ \mathbf{done}) &= \bar{x} \cup needs(S) \end{aligned}$$

Finally, let $stops(S)$ be true if the execution of S may be stopped: either by looping or waiting on a synchronization command. This function is used by the automaton when finishing the complete evaluation of some conditionals whose condition carries variety. This is done in order to prevent an attacker to learn some secret information by observing from the source code that, in under some conditions, one

of the branches of a conditional, whose condition carries variety, can not terminate. $stops(S)$ is formally defined as follows:

$$\begin{aligned}
stops(x := e) &= false \\
stops(\mathbf{output} \ e) &= false \\
stops(\mathbf{skip}) &= false \\
stops(S_1 ; S_2) &= stops(S_1) \vee stops(S_2) \\
stops(\mathbf{if} \ e \ \mathbf{then} \ S_1 \ \mathbf{else} \ S_2 \ \mathbf{end}) &= stops(S_1) \vee stops(S_2) \\
stops(\mathbf{while} \ e \ \mathbf{do} \ S \ \mathbf{done}) &\equiv e \neq false \\
stops(\mathbf{with} \ \bar{x} \ \mathbf{when} \ e \ \mathbf{do} \ S \ \mathbf{done}) &\equiv e \neq true
\end{aligned}$$

Figure 6 shows that the automaton forbids (*NO*) or edits (**output** θ) only executions of output statements. For other inputs, it either allows the evaluation of the statement — a transition exists for the current input in the current state — or forces the monitoring semantics to take an execution step for another thread — there is no transition for the current input in the current state. Whenever a transition occurs, the automaton tracks, in the set V , the variables that may contain secret information — have *variety* —; it registers, in W , the variables whose variety may evolve differently due to a conditional whose condition carries variety; it tracks, in L , the variables whose lock status may carry variety; and it tracks, in w , the *variety* of the branching conditions.

Inputs “**branch**(λ, e, P^c, P^u)” are generated at entry point of conditionals. On reception of such inputs in state $((V, W, L, w))$, if the conditional to be evaluated belongs to a branch of a conditional whose condition carries variety — w does not belong to $\{\perp\}^*$, the automaton simply pushes \perp at the end of w . Otherwise, the automaton checks if the branching condition (e) carries variety — i.e. if its value might be influenced by the initial values of $S(P)$. To do so, it computes the intersection of the variables appearing in e with the set V . If the intersection is empty, then the condition of the branching statement does not carry variety. In that case, the automaton simply pushes \perp at the end of w . If the intersection is not empty, then the value of e may be influenced by the initial values of $S(P)$. If this is the case then the automaton checks if the execution can proceed without being stopped by a lock that it can not acquire. To do so, it verifies that there is no needed lock which is already owned by another thread or has already been booked previously for the execution of a conditional of another thread ($\bar{l} \cap (\lambda \cup L) = \emptyset$). If all needed locks are available, the automaton pushes \top at the end of w , registers the needed locks in the new automaton state and deals with indirect flows. This is done by adding all variables which may be assigned to in one of the branches of the conditional into the set of variables potentially carrying variety (V) and into the multiset of variables whose order of assignment may carry variety (W).

Whenever execution exits a branch — which is a member of a conditional c — the input “**merge**(P^c, P^u)” is sent to the automaton. On reception of such input, the automaton checks if other private inputs may have induced the execution of the conditional to be stopped. This is the case if the condition of the conditional c carries variety — w ends with \top — and the function $stops()$ concludes that at least one of the branches of c can get stuck. If that is not the case, the automaton allows the evaluation step and the last letter of w is removed. As any such input matches a previous input “**branch**(λ, e, P^c, P^u)” — generated by the same conditional c — which adds a letter to the end of w , the effect of “**merge**(P^c, P^u)” restores the context to its state before the conditional was processed. Additionally, if c is the first conditional whose condition carries variety in this sequential program, the variables added into the multiset W for protection while executing c are removed from the multiset.

Before executing any synchronization command, the monitoring semantics sends to the automaton

$((V, W, L, w), \text{branch}(\lambda, e, P^c, P^u)) \xrightarrow{OK} (V, W, L, w \perp)$	$\text{iff} \left\{ \begin{array}{l} FV(e) \cap V = \emptyset \\ w \notin \{\perp\}^* \end{array} \right.$	(T-BRANCH-low)
$((V, W, L, w), \text{branch}(\lambda, e, P^c, P^u)) \xrightarrow{OK} (V \cup \tilde{v}, W \uplus \tilde{v}, L \cup \tilde{I}, w \top)$ with $\left\{ \begin{array}{l} \tilde{v} = \text{modify}(P^c) \cup \text{modify}(P^u) \\ \tilde{I} = \text{needs}(P^c) \cup \text{needs}(P^u) \end{array} \right.$	$\text{iff} \left\{ \begin{array}{l} FV(e) \cap V \neq \emptyset \\ w \in \{\perp\}^* \\ \tilde{I} \cap (\lambda \cup L) = \emptyset \end{array} \right.$	(T-BRANCH-high)
$((V, W, L, w \perp), \text{merge}(P^c, P^u)) \xrightarrow{OK} (V, W, L, w)$		(T-MERGE-low)
$((V, W, L, w \top), \text{merge}(P^c, P^u)) \xrightarrow{OK} (V, W \setminus \tilde{v}, L \setminus \tilde{I}, w)$ with $\left\{ \begin{array}{l} \tilde{v} = \text{modify}(P^c) \cup \text{modify}(P^u) \\ \tilde{I} = \text{needs}(P^c) \cup \text{needs}(P^u) \end{array} \right.$	$\text{iff} \left\{ \begin{array}{l} \neg \text{stops}(P^c) \\ \neg \text{stops}(P^u) \end{array} \right.$	(T-MERGE-high)
$((V, W, L, w), \text{sync}(\bar{x}, e)) \xrightarrow{OK} (V, W, L, w)$	$\text{iff} \left\{ \begin{array}{l} FV(e) \cap V = \emptyset \\ (w \notin \{\perp\}^* \wedge \bar{x} \cap L = \emptyset) \vee x \in W \end{array} \right.$	(T-SYNC)
$(q, \text{skip}) \xrightarrow{OK} q$		(T-SKIP)
$((V, W, L, w), x := e) \xrightarrow{OK} (V \cup \{x\}, W, L, w)$	$\text{iff} \left\{ \begin{array}{l} FV(e) \cap V \neq \emptyset \\ x \in W \end{array} \right.$	(T-ASSIGN-sec)
$((V, W, L, w), x := e) \xrightarrow{OK} (V \setminus \{x\}, W, L, w)$	$\text{iff} \left\{ \begin{array}{l} FV(e) \cap V = \emptyset \\ x \notin W \end{array} \right.$	(T-ASSIGN-pub)
$((V, W, L, w), \text{output } e) \xrightarrow{OK} (V, W, L, w)$	$\text{iff } w \in \{\perp\}^* \text{ and } FV(e) \cap V = \emptyset$	(T-OUTPUT-ok)
$((V, W, L, w), \text{output } e) \xrightarrow{\text{output.}\delta} (V, W, L, w)$	$\text{iff } w \in \{\perp\}^* \text{ and } FV(e) \cap V \neq \emptyset$	(T-OUTPUT-def)
$((V, W, L, w), \text{output } e) \xrightarrow{NO} (V, W, L, w)$	$\text{iff } w \notin \{\perp\}^*$	(T-OUTPUT-no)

Figure 6: Transition function of the monitoring automata

an input of type “`sync(\bar{x}, e)`”. The automaton allows the evaluation step only if the expression in the synchronization command does not carry variety and either the needed locks are not reserved ($\bar{x} \cap L = \emptyset$) or the automaton has already booked those locks for the current thread.

Atomic actions (assignment, skip or output) are sent to the automaton for validation before their execution. The atomic action **skip** is considered safe because the non-interference definition considered in this work is not time sensitive. Hence the automaton always authorizes its execution by outputting *OK*.

When executing an assignment ($x := e$), two types of flows are created. The first is a direct flow from e to x . The second flow is an explicit indirect flow from the context of execution to x . For example, the execution of the assignment in “**if b then $x := y$ else skip end**” creates such a flow from b to x . Both forms of flows are *always* created when an assignment is executed. Explicit indirect flows are indirect flows and so are already taken care for by the processing of inputs of type “**branch(λ, e, P^e, P^u)**”. Therefore, on such input, the automaton deals only with direct flows. Hence, on input $x := e$, the automaton checks if the value of e carries variety. This is the case only if $FV(e)$ and V are not disjoint. If the value of e is influenced by the initial values of $S(P)$ then at least one of the variables appearing in e has been influenced by $S(P)$. Such variables are members of V . Let (V', W', L', w') be the new automaton state after the transition. If the origin of the direct flow is influenced by the initial values of $S(P)$, then x (the variable modified) is added to V : $V' = V \cup \{x\}$. Otherwise x receives a new value which is not influenced by $S(P)$. In that case, V' equals $V \setminus \{x\}$. This makes the mechanism flow-sensitive. Other parts of the automaton state do not change: $W' = W$, $L' = L$ and $w' = w$. There is an exception to this rule. If there exists an indirect flow from the condition carrying variety of a still active conditional then the variable assigned to must not be removed from the set of variables carrying variety. If such an indirect flow exists then the assigned variable is still under the protection acquired when the corresponding “**branch(λ, e, P^e, P^u)**” input has been processed ($x \in W$). In that case, the variable is left in the set V .

The rules for the automata input, “**output e ,**” prevent bad flows through two different channels. The first one is the actual content of what is output. In a public context, ($w \in \{\perp\}^*$), if the program tries to output a secret (i.e., the intersection of V and the variables in e is not empty), then the value of the output is replaced by a default value. This value can be a message informing the user that, for security reasons, the output has been denied. To do so, the automaton outputs a new output statement to execute in place of the current one. The second channel is the generation of an output by itself. This channel exists because, depending on the path followed, some outputs may or may not be executed. Hence, if the automaton detects that this output may not be executed with different values for $S(P)$ (the context carries *variety*) then *any* output must be forbidden; and the automaton outputs *NO*.

The automaton states for concurrent programs. As explained above, the monitoring automaton for concurrent program is based on the monitoring automaton for sequential programs. Monitoring automaton states for concurrent programs are similar to the automaton states for sequential programs. An automaton state is a quadruplet $(Q = (V, W, L, \bar{w}))$ which belongs to the following set:

$$2^{\mathbb{X}} \times (\mathbb{X} \rightarrow \mathbb{N}) \times 2^{\mathbb{X}} \times (\mathbb{N} \rightarrow \{\top, \perp\}^*)$$

V , W and L are directly inherited from automaton states for sequential programs. \bar{w} is a function mapping thread identifiers to their respective branching context (as defined for sequential programs).

The functions used to extract, respectively update, the automaton state for a given thread from, respectively into, the automaton state of the concurrent program are defined below.

$$\text{extractStates}((V, W, L, \bar{w}), t) = (V, W, L, \bar{w}(t)) \quad (1)$$

$$\text{updateStates}((V, W, L, \bar{w}), t, (V', W', L', w')) = (V', W', L', \bar{w}[t \mapsto w']) \quad (2)$$

The current section defines formally the monitoring automaton. Among others, it defines an input alphabet and an output alphabet which serve to communicate with the monitoring semantics. This semantics, described in the next section, is in charge of the concrete evaluation of the concurrent program under the supervision of the monitoring automaton.

4 The Monitoring Semantics

During a monitored execution, a thread contains a *monitored statement* ($\langle \text{monit-stat} \rangle$) which is highly similar to *execution statements* described in Figure 2. The only difference is the addition of *branched statements* ($\otimes(\langle \text{prog} \rangle, \langle \text{prog} \rangle)[\langle \text{monit-stat} \rangle]$). The statement $\otimes(P^e, P^a)[S]$ states that statement S is a partial execution of P^e — i.e. S is the result of the application of some execution steps to P^e — and that P^e is the executed branch of a conditional whose un-executed branch is P^a . The rule (E_M-IF) of Figure 8 may give a good intuition of the meaning of this statement. The grammar of *monitored statements* is given in Figure 7 and is based on the grammar of sequential programs (Figure 1).

$$\begin{aligned} \langle \text{monit-stat} \rangle & ::= \emptyset \\ & | \langle \text{prog} \rangle \\ & | \langle \text{monit-stat} \rangle ; \langle \text{prog} \rangle \\ & | \odot \langle \text{ident-set} \rangle [\langle \text{monit-stat} \rangle] \\ & | \otimes(\langle \text{prog} \rangle, \langle \text{prog} \rangle)[\langle \text{monit-stat} \rangle] \end{aligned}$$

Figure 7: Grammar of sequential programs under monitored execution

A statement under monitored execution is a set — potentially empty — of interwoven *locked statements* and *branched statements* followed by a usual sequential program. Formally, a *monitored statement* belongs to the language defined by the following expression:

$$\left(\odot \langle \text{ident-set} \rangle [+ \otimes(\langle \text{prog} \rangle, \langle \text{prog} \rangle)[]^n \left(\langle \text{prog} \rangle + \emptyset \right) \left((; \langle \text{prog} \rangle)^?]^n \left(; \langle \text{prog} \rangle \right)^? \right)$$

Thread semantics. The monitoring semantics of concurrent programs is based on the semantics of *monitored statements* which is described in Figure 8. This semantics is based on rules written:

$$\langle \zeta \Vdash S \rangle \xrightarrow{o}_M \langle \zeta' \Vdash S' \rangle$$

The previous rule reads as follows: in *monitored execution state* ζ , *monitored statements* S evaluates to S' yielding state ζ' and output sequence o . A monitored execution state ζ is a pair (ς, q) composed of an execution state (ς) of the standard semantics and a monitoring automaton state (q).

Executions of *monitored statements* interact with the security automaton by sending it an input and receiving back an output. For simplicity, we assume that the automaton inputs are generated dynamically by the interpreter that implements the semantics in Figure 3 (But a compiler can easily embed such events

into a program's compiled code). In the rules of the semantics, this is done using automaton transitions written:

$$(q, \phi) \xrightarrow{\psi} q'$$

This transition states that, in state q , if the automaton receives the input ϕ then it yields output ψ and the new automaton state is q' . The semantics of *monitored statements* also makes use of the semantics of *execution statements* — the standard semantics — for the evaluation of actions (assignments, outputs, or skip statements). Those semantics can be distinguished by their name appearing on the bottom right of the arrow.

There are three rules for atomic actions: **skip**, $x := e$ and **output** e . There is one rule for each possible automaton answer to the action executed. Either the automaton authorizes the execution (*OK*), denies the execution (*NO*), or replaces the action by another one. The rules use the standard semantics (Figure 3) when an action must be executed. In the case where the execution is denied, the evaluation omits the current action (as if the action was "skip"). For the case where, on reception of input A , the monitoring automaton returns A' , the monitoring semantics executes A' instead of A . Note from Figure 6, that A' can only be the action that outputs the default value (**output** θ).

For conditionals, the evaluation begins by sending to the automaton the input "branch(λ, e, P^e, P^u)" where λ is the set of variables whose lock is currently owned by a thread, e is the test of the conditional, P^e is the "branch" which will be executed and P^u is the "branch" which will not be executed. After the evaluation step, the monitored execution state is updated with the automaton state returned by the monitoring automaton and the conditional evaluated is replaced by the branch to execute.

A synchronization command "**with** \bar{x} **when** e **do** P^s **done**" can be evaluated only if the needed locks \bar{x} are not currently owned by another thread, the condition e is true and the monitoring automaton allows the evaluation on reception of the input "sync(\bar{x}, e)". After the evaluation step, the needed locks are added to the set of locks currently owned by a thread, the automaton state is updated and the command is replaced by a *locked statement* indicating that the locks \bar{x} are owned by the current thread for the evaluation of the program P^s .

Evaluating a *branched statements* ($\otimes(P^e, P^u)[S^b]$) is equivalent to evaluating the enclosed statement (S^b) as long as this statement is not completely evaluated. If the evaluation of S^b is completed then the input "merge(P^e, P^u)" is sent to the automaton and the evaluation of the thread can proceed only if the automaton allows it.

In order to take one evaluation step of a *locked statement* ($\odot\bar{x}[S^s]$), the set of locks \bar{x} is temporarily removed from the set of owned locks in the execution state. Then, one evaluation step is taken for S^s and the locks \bar{x} are put back in the set of owned locks in the execution state if the evaluation of S^s is not completed.

Concurrent program semantics. As explained in Section 3, the automaton states for monitoring the execution of a single thread are different from the states for monitoring concurrent programs. The description of automaton states for monitoring concurrent programs is given in Section 3. Let Q stands for the state of the security automaton for concurrent programs. The monitoring semantics uses two functions for converting between automaton states for thread and automaton states for concurrent programs. Those functions are described in Section 3. Succinctly, $\text{extractState}(Q, i)$ is the automaton state for monitoring the i^{th} thread of the concurrent program whose automaton state is Q . $\text{updateStates}(Q, i, q)$ is an automaton state for concurrent program equivalent to Q except for the i^{th} thread. Information concerning this thread has been updated using the information contained in the automaton state for thread monitoring q . The semantics of monitored executions of concurrent programs is given by the following rule:

$\frac{(q, A) \xrightarrow{OK} q' \quad \langle \zeta \vdash A \rangle \xrightarrow{o}_S \langle \zeta' \vdash \emptyset \rangle}{\langle \zeta, q \rangle \Vdash A \xrightarrow{o}_M \langle \zeta', q' \rangle \Vdash \emptyset}$	(E _M -OK)
$\frac{(q, A) \xrightarrow{A'} q' \quad \langle \zeta \vdash A' \rangle \xrightarrow{o}_S \langle \zeta' \vdash \emptyset \rangle}{\langle \zeta, q \rangle \Vdash A \xrightarrow{o}_M \langle \zeta', q' \rangle \Vdash \emptyset}$	(E _M -EDIT)
$\frac{(q, A) \xrightarrow{NO} q'}{\langle \zeta, q \rangle \Vdash A \xrightarrow{\epsilon}_M \langle \zeta, q' \rangle \Vdash \emptyset}$	(E _M -NO)
$\frac{\sigma(e) = v \quad (q, \text{branch}(\lambda, e, P^v, P^{-v})) \xrightarrow{OK} q'}{\langle \zeta, q \rangle \Vdash \text{if } e \text{ then } P^{\text{true}} \text{ else } P^{\text{false}} \text{ end} \xrightarrow{\epsilon}_M \langle \zeta, q' \rangle \Vdash \otimes(P^v, P^{-v})[P^v]}$	(E _M -IF)
$\frac{\begin{array}{l} P^{\text{true}} = P^l ; \text{ while } e \text{ do } P^l \text{ done} \quad P^{\text{false}} = \text{skip} \\ \sigma(e) = v \quad (q, \text{branch}(\lambda, e, P^v, P^{-v})) \xrightarrow{OK} q' \end{array}}{\langle \zeta, q \rangle \Vdash \text{while } e \text{ do } P^l \text{ done} \xrightarrow{\epsilon}_M \langle \zeta, q' \rangle \Vdash \otimes(P^v, P^{-v})[P^v]}$	(E _M -WHILE)
$\frac{\bar{x} \cap \lambda = \emptyset \quad \sigma(e) = \text{true} \quad (q, \text{sync}(\bar{x}, e)) \xrightarrow{OK} q'}{\langle ((\sigma, \lambda), q) \Vdash \text{with } \bar{x} \text{ when } e \text{ do } P^s \text{ done} \rangle \xrightarrow{\epsilon}_M \langle ((\sigma, \lambda \cup \bar{x}), q') \Vdash \odot \bar{x}[P^s] \rangle}$	(E _M -WITH)
$\langle \zeta \Vdash \emptyset ; S^t \rangle \xrightarrow{\epsilon}_M \langle \zeta \Vdash S^t \rangle$	(E _M -SEQ _{exit})
$\frac{\langle \zeta \Vdash S^h \rangle \xrightarrow{o}_M \langle \zeta' \Vdash S^{h'} \rangle}{\langle \zeta \Vdash S^h ; S^t \rangle \xrightarrow{o}_M \langle \zeta' \Vdash S^{h'} ; S^t \rangle}$	(E _M -SEQ _{step})
$\frac{(q, \text{merge}(P^e, P^u)) \xrightarrow{OK} q'}{\langle \zeta, q \rangle \Vdash \otimes(P^e, P^u)[\emptyset] \xrightarrow{\epsilon}_M \langle \zeta, q' \rangle \Vdash \emptyset}$	(E _M -BRANCH _{exit})
$\frac{S^b \neq \emptyset \quad \langle \zeta \Vdash S^b \rangle \xrightarrow{o}_M \langle \zeta' \Vdash S^{b'} \rangle}{\langle \zeta \Vdash \otimes(P^e, P^u)[S^b] \rangle \xrightarrow{o}_M \langle \zeta' \Vdash \otimes(P^e, P^u)[S^{b'}] \rangle}$	(E _M -BRANCH _{step})
$\langle ((\sigma, \lambda), q) \Vdash \odot \bar{x}[\emptyset] \rangle \xrightarrow{\epsilon}_M \langle ((\sigma, \lambda \setminus \bar{x}), q) \Vdash \emptyset \rangle$	(E _M -LOCKED _{exit})
$\frac{S^s \neq \emptyset \quad \langle ((\sigma, \lambda \setminus \bar{x}), q) \Vdash S^s \rangle \xrightarrow{o}_M \langle ((\sigma', \lambda'), q') \Vdash S^{s'} \rangle}{\langle ((\sigma, \lambda), q) \Vdash \odot \bar{x}[S^s] \rangle \xrightarrow{o}_M \langle ((\sigma', \lambda' \cup \bar{x}), q') \Vdash \odot \bar{x}[S^{s'}] \rangle}$	(E _M -LOCKED _{step})

Figure 8: Semantics combining standard small step semantics and the monitoring automaton

$$\frac{i \in \text{dom}(\Theta) \quad \langle\langle \zeta, \text{extractState}(Q, i) \rangle \Vdash \Theta(i) \rangle \xrightarrow{o}_M \langle\langle \zeta', q \rangle \Vdash S'_i \rangle}{\langle\langle \zeta, Q \rangle \Vdash \Theta \rangle \xrightarrow{o}_M \langle\langle \zeta', \text{updateStates}(Q, i, q) \rangle \Vdash \Theta[i \mapsto S'_i] \rangle} \quad (\text{E}_M\text{-CONCUR})$$

Initial state of a monitored concurrent execution. There is a unique initial state for monitoring the execution of a given concurrent program with a given initial value store. Definition 4.1 states that the initial state of the monitored execution of a concurrent program is the monitoring execution state whose initial value store is the given one, set of owned lock is empty and automaton state designates the value of the private inputs as the only elements carrying variety.

Definition 4.1 (Initial State of Monitored Executions).

For all concurrent programs Θ and value stores σ , let $\zeta_{\Theta, \sigma}^I$ be the initial state of the monitored execution of Θ with initial value store σ . $\zeta_{\Theta, \sigma}^I$ is equal to $((\sigma, \emptyset), (\mathcal{S}(\Theta), \emptyset, \emptyset, \{i \mapsto \epsilon \mid i \in \text{dom}(\Theta)\}))$.

5 Example of monitored execution

Figure 9 is an example of a concurrent program having two threads. Figure 9(a) contains the code of the sequential program of the first thread and Figure 9(b) contains the code of the second thread. This program has only one private input (h) and no public input. It uses three internal variables (v , b and x). Both threads attempt to write in the variable v . Both assignments to v are protected by a synchronization on the lock of v . Additionally, before assigning a value to v , the second thread waits for b to be true. After assigning a value to v , the second thread outputs twice the value of x . The first thread, depending on the value of the private input h , either assigns a value to x and outputs “a”, or assign a value to v . The last command evaluated by the first thread resets the value of x to 0.

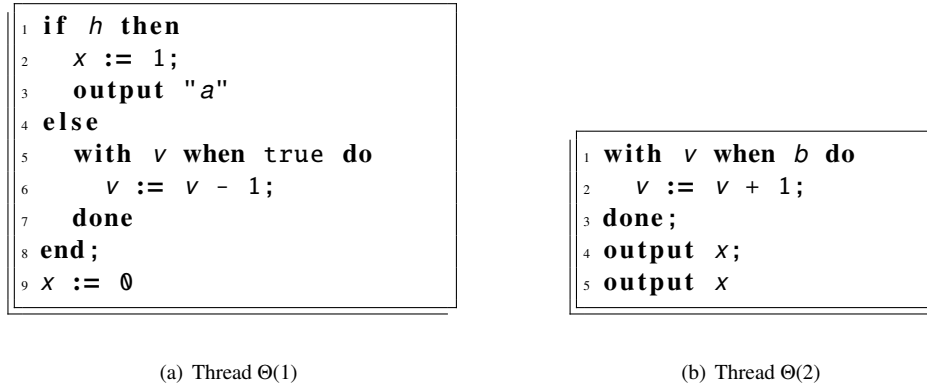


Figure 9: Another concurrent program

Table 2 shows the evolution of the monitoring automaton for an execution of the program of Figure 9. This execution starts in an initial state where the private input h is true. As the program has only one private input (h) and two threads, the initial state of the monitoring automaton is the following one:

$$(\{h\}, \emptyset, \emptyset, [1 \mapsto \epsilon, 2 \mapsto \epsilon])$$

Execution's steps are numbered on the left. The first column of the table shows the value of the program counters of the two threads. “ $i \triangleright j$ ” and “ $i \blacktriangleright j$ ” indicates that the program counter of the i^{th} thread maps to the line j . 0 is used for j when the execution of the thread is completed. \blacktriangleright is used to designate the thread which will be evaluated at this execution step. The following column contains the event abstractions which are sent by the semantics to the automaton. In this column, P^l stands for the lines 2 to 3 of the first thread and P^f stands for the lines 5 to 7 of the first thread. Then comes the answer of the automaton letting the semantics know what to do. The 4th column gives the new state of the automaton after the transition triggered by the automaton input shown on the same line. Consequently, the automaton state before the transition is the one of the preceding line. Finally, the last column lists the actions which are really evaluated by this monitored execution of the program.

	Program counters	Automaton:			Actions executed
		input	output	new state	
1	1 \triangleright 1 2 \blacktriangleright 1	<code>sync($\{v\}, b$)</code>	<i>OK</i>	$(\{h\}, \emptyset, \{v\}, [1>\epsilon, 2>\epsilon])$	
2	1 \triangleright 1 2 \blacktriangleright 2	<code>$v := v + 1$</code>	<i>OK</i>	$(\{h\}, \emptyset, \emptyset, [1>\epsilon, 2>\epsilon])$	$v := v + 1$
3	1 \blacktriangleright 1 2 \triangleright 4	<code>branch(\emptyset, h, P^l, P^f)</code>	<i>OK</i>	$(\{h, x, v\}, \{x, v\}, \{v\}, [1>\top, 2>\epsilon])$	
4	1 \triangleright 2 2 \blacktriangleright 4	<code>output x</code>	output θ	$(\{h, x, v\}, \{x, v\}, \{v\}, [1>\top, 2>\epsilon])$	output θ
5	1 \blacktriangleright 2 2 \triangleright 5	<code>$x := 1$</code>	<i>OK</i>	$(\{h, x, v\}, \{x, v\}, \{v\}, [1>\top, 2>\epsilon])$	$x := 1$
6	1 \blacktriangleright 3 2 \triangleright 5	<code>output "a"</code>	<i>NO</i>	$(\{h, x, v\}, \{x, v\}, \{v\}, [1>\top, 2>\epsilon])$	
7	1 \blacktriangleright 8 2 \triangleright 5	<code>merge(P^l, P^f)</code>	<i>OK</i>	$(\{h, x, v\}, \emptyset, \emptyset, [1>\epsilon, 2>\epsilon])$	
8	1 \blacktriangleright 9 2 \triangleright 5	<code>$x := 0$</code>	<i>OK</i>	$(\{h, v\}, \emptyset, \emptyset, [1>\epsilon, 2>\epsilon])$	$x := 0$
9	1 \triangleright 0 2 \blacktriangleright 5	<code>output x</code>	<i>OK</i>	$(\{h, v\}, \emptyset, \emptyset, [1>\epsilon, 2>\epsilon])$	output x

Table 2: Example of the automaton evolution during an execution.

At the execution step 2, the only thread which can be executed is the second one. The reason is that at the preceding step, the second thread took the lock of v . The first command to be executed by the first thread is a conditional containing a synchronization command on v . Therefore, to execute the first thread, the monitor requires this thread to be able to acquire the lock of v . This is impossible as this lock is owned by the second thread. Hence, even if the branch to be executed is not the one containing the synchronization command, the program can not evaluate the conditional of the first thread. An example of the reason of this rule has been given on page 8.

Execution step 3 evaluates the conditional of the first thread. This conditional, whose condition carries variety, contains an assignment to x in one branch and an assignment to v in the other one. In order to prevent an attack similar to the one exposed on page 9, the monitor considers them as carrying variety straight away, even if their respective assignments have not been evaluated yet. This is done by adding them to the first element of the new automaton state. A clever newsmonger — that special type of attacker thread — may try to dupe the monitor into believing that those variables do not carry variety anymore by resetting their value. To prevent it, the monitor also registers that those variables carry variety because of the processing of a conditional. This is done by adding them into the multiset — or bag — which is the second element of the new state. Variables x and v will appear at least once in this multiset as long as the processing of the conditional which added them into it is not over. Furthermore, as stated in the explanation of step 2, the first thread acquires the lock of v because the conditional processed at this step is conditioned by an expression carrying variety and contains a synchronization command on v in one of its branches. Finally, the monitor registers the new context of execution — reflecting the fact that the branch that will be executed depend on an expression carrying variety — by adding \top to the word

associated with the identifier of the first thread in the fourth element of the new state.

In step 4 of the execution, the second thread attempts to output the value of x . This variable belongs to the first element of the automaton state before the transition. This means that the value of x may carry variety. Therefore, the monitor replaces the value of x by a default value. It allows the user to know that an output has been prevented for security reasons. Five steps later, at step 9, the second thread tries again to output the value of x . However, in the meanwhile, the first thread has reset the value of x to a new value. Doing so, it removed the variety in this variable. This is reflected by the fact that x does not belong anymore to the first element of the automaton state. Therefore, the monitoring automaton lets the output occur.

In step 6 of the execution, the first thread attempts to output a constant while still in one of the branches of a conditional whose condition carries variety. This is an unsafe behavior that the monitor forbids. Consequently, the execution simply skips this command.

This section shows on an example that the monitoring mechanism is able to ensure the confidentiality of private inputs. The next section proves this fact for any monitored execution.

6 Properties of the Monitoring Mechanism

After formally defining the monitoring mechanism, this section collates the monitor's characteristics to the standard properties of *soundness* and, specific to monitors, *transparency*. Section 6.1 proves the soundness of the monitoring mechanism with regard to the notion of non-interference. If the monitor fulfills the soundness property, it is of course impossible for it to be complete, as the non-interference problem is undecidable, or transparent, as with an interfering execution the monitor can be either transparent or sound but not both. However, section 6.2 still proves that the monitor is transparent for a non-trivial set of executions. .

6.1 Soundness

The monitoring mechanism is proved to be sound with regard to the non-interference property for executions. This means that, for any output sequences generated during the monitored execution of any program Θ started in the initial state $\zeta_{\Theta, \sigma}^I$ and value stores σ' low equivalent to σ , there exists a thread interleaving such that during the monitored execution of Θ started in the initial state $\zeta_{\Theta, \sigma'}^I$, the same output sequence is generated. This property is formally stated by theorem 6.1.

Theorem 6.1 (Monitored Executions are Non-interfering).

For all programs Θ , value stores σ , any monitored execution of Θ started in the initial state $\zeta_{\Theta, \sigma}^I$ is non-interfering. This is formally stated as follows:

$$\forall \Theta, \forall \sigma \in (\mathbb{X} \rightarrow \mathbb{D}) : o \in O[[\Theta]]_{M, \zeta_{\Theta, \sigma}^I} \Rightarrow ni(\Theta, M, \zeta_{\Theta, \sigma}^I, o)$$

Proof sketch. The proof — which can be found in Appendix A.1 — goes by induction on the derivation tree of the current execution (e_c). First, it relies on the fact that, because of the monitoring mechanism, for any evaluation started with the same public inputs there exists an execution (e_2) having a thread interleaving “similar” to the one of the current execution. Additionally, it is demonstrated that, if a thread of e_2 follows a path in its sequential program different from the path followed by the equivalent thread

in e_c , nothing will be output by those threads until they reach an “equivalent state”. It is then possible to show that e_c and e_2 have the same output sequence. \square

6.2 Partial Transparency

A common property stated for monitors is *transparency*. A transparent monitor is one that does not alter the behavior of the monitored program. An interfering execution has, with regard to confidentiality, a faulty observable behavior. Therefore, a sound monitor enforcing non-interference has to alter the observable behavior of such an execution. Consequently, it is impossible for a sound monitor enforcing non-interference to also be transparent. However, for a precise set of programs, it is possible to prove that the monitoring mechanism presented in this report achieves transparency — i.e. it does not alter the observable behavior of any execution of those programs. This set of programs is the set of all programs which are well typed under a type system similar to the one of Smith and Volpano [SV98].

This type system is described in Figure 10. The language used in this report includes two structures which do not appear in the language used in [SV98]. The two typing rules added for those structures are the only salient differences with the type system of Smith and Volpano [SV98]. The lattice of types used has only two elements and is defined using the reflexive relation $\leq (L \leq H)$. L is the type for public data and H the type for private data. The typing environment, γ , prescribes types for identifiers and is extended to handle expressions. $\gamma(e)$ is the type of the expression e in the typing environment γ . It is equal to the least upper bound of the types of the free variables appearing in e , or L if there is no free variable in e . It is formally defined as follows:

$$\gamma(e) = \bigsqcup_{x \in FV(e)} \gamma(x)$$

For any sequential program P , if “ $\gamma \vdash_S P : \tau$ cmd” for some τ and γ in which every secret input is typed secret — i.e. $\forall x \in \mathcal{S}(P), \gamma(x) = H$ — then P is said to be well typed under the typing environment γ . For any concurrent program Θ , if all its threads contain a program well typed under the same typing environment γ , then Θ is said to be well typed under this typing environment γ . This is written “ $\gamma \vdash_S \Theta$ ”.

Theorem 6.2 states that the monitoring mechanism does not alter executions of well typed programs. In other words, the monitoring mechanism is transparent for any well typed program. To show that the monitor is transparent, and still sound, for a bigger set of programs, consider the concurrent program composed only of this sequential program : $x := h; x := \mathbf{0}; \mathbf{output} \ x$. h is the only secret input. Every execution is non-interfering. But as the type system is flow insensitive, this program is ill-typed. However, the monitoring mechanism does not interfere with the outputs of this program while still guaranteeing that any monitored execution is non-interfering.

Theorem 6.2 (Partial-transparency: monitoring preserves *type-safe* programs).

For all programs Θ with secret inputs $\mathcal{S}(\Theta)$, typing environments γ with variables belonging to $\mathcal{S}(\Theta)$ typed secret, and value stores σ , if Θ is well typed under γ then the observable behavior of any monitored execution of Θ started in the initial state $\zeta_{\Theta, \sigma}^I$ is equal to the observable behavior of the unmonitored execution of Θ started in the initial state (σ, \emptyset) . This is formally stated as follows:

$$\gamma \vdash_S \Theta \quad \Rightarrow \quad O[[\Theta]]_m \zeta_{\Theta, \sigma}^I = O[[\Theta]]_s(\sigma, \emptyset)$$

Proof sketch. As “ $\gamma \vdash_S \Theta$ ” implies “ $\gamma \vdash_E \Theta$ ”, the proof follows from lemmas A.10 and A.11. \square

$\frac{}{\gamma \vdash_S \mathbf{skip} : \tau \text{ cmd}}$	(T _S -SKIP)
$\frac{\gamma(e) \leq \gamma(x) \quad \tau \leq \gamma(x)}{\gamma \vdash_S x := e : \tau \text{ cmd}}$	(T _S -ASSIGN)
$\frac{\gamma(e) \leq L}{\gamma \vdash_S \mathbf{output} e : L \text{ cmd}}$	(T _S -OUTPUT)
$\frac{\gamma \vdash_S S^h : \tau \text{ cmd} \quad \gamma \vdash_S S^t : \tau \text{ cmd}}{\gamma \vdash_S S^h ; S^t : \tau \text{ cmd}}$	(T _S -SEQ)
$\frac{\gamma(e) \leq \tau' \quad \gamma \vdash_S S^{\text{true}} : \tau' \text{ cmd} \quad \gamma \vdash_S S^{\text{false}} : \tau' \text{ cmd} \quad \tau \leq \tau'}{\gamma \vdash_S \mathbf{if} e \mathbf{then} S^{\text{true}} \mathbf{else} S^{\text{false}} \mathbf{end} : \tau \text{ cmd}}$	(T _S -IF)
$\frac{\gamma(e) \leq L \quad \gamma \vdash_S S^1 : L \text{ cmd}}{\gamma \vdash_S \mathbf{while} e \mathbf{do} S^1 \mathbf{done} : L \text{ cmd}}$	(T _S -WHILE)
$\frac{\gamma(e) \leq L \quad \gamma \vdash_S S^s : L \text{ cmd}}{\gamma \vdash_S \mathbf{with} \bar{x} \mathbf{when} e \mathbf{do} S^s \mathbf{done} : L \text{ cmd}}$	(T _S -SYNC)

Figure 10: The type system used for comparison

7 Conclusion

This report addresses the problem of the respect of the secret data confidentiality by concurrent programs. The problem is formalized using the non-interference property which states that a program is safe if and only if its publicly observable behavior is not influenced by the values of its private inputs.

This problem is usually addressed using static analyses. To the author's knowledge, the solution proposed in this report is the only dynamic analysis dealing with non-interference in a concurrent setting. It consists in a monitoring mechanism enforcing the respect of the confidentiality of secret inputs. It is defined as a special semantics communicating with a security automaton. The role of the semantics is to send abstractions of the events occurring during the execution to the automaton. Then, it executes the program in accordance to the answers sent back by the automaton. This latter tracks the information flows and controls — allows, modifies or denies — the execution of output statements and synchronization commands.

As the proposed mechanism deals with non-interference — which is not a property of an execution trace —, it significantly differs from usual monitors. The dynamic analysis proposed is required to take into account the content of branches which are not executed. Additionally, due to the fact that the

monitor enforces non-interference on the fly, it is also required to change the position of synchronization commands while still enforcing the thread interleaving constraints induced by the original position of those commands.

Section 6 not only proves that this monitoring mechanism is sound — in the sense that it enforces non-interference for any execution —, but also that it is transparent for any program well typed under a type system similar to the one of Smith and Volpano [SV98].

A Proofs

The following proofs have been written before minor modifications of the notations used and has not been rewritten yet. Consequently, some of the notations found in the proofs may be different from those used in the previous sections. However the modifications are straightforward.

A.1 Soundness

A.1.1 Definitions

Notations. For any sup and any i , let $\zeta_i^{sup} = (\varsigma_i^{sup}, q_i^{sup}) = ((\sigma_i^{sup}, \lambda_i^{sup}), (V_i^{sup}, W_i^{sup}, L_i^{sup}, w_i^{sup}))$.

Owned locks. $owns(S)$ is the set of all locks currently owned by the thread executing S .

$$\begin{aligned}
 owns(x := e) &= \emptyset \\
 owns(\mathbf{output} \ e) &= \emptyset \\
 owns(\mathbf{skip}) &= \emptyset \\
 owns(\emptyset) &= \emptyset \\
 owns(S_1 ; S_2) &= owns(S_1) \cup owns(S_2) \\
 owns(\mathbf{if} \ e \ \mathbf{then} \ S_1 \ \mathbf{else} \ S_2 \ \mathbf{end}) &= owns(S_1) \cup owns(S_2) \\
 owns(\mathbf{while} \ e \ \mathbf{do} \ S \ \mathbf{done}) &= owns(S) \\
 owns(\mathbf{with} \ \bar{x} \ \mathbf{when} \ e \ \mathbf{do} \ S \ \mathbf{done}) &= owns(S) \\
 owns(\otimes(P_1, P_2)[S]) &= owns(S) \\
 owns(\odot \bar{x}[S]) &= \bar{x} \cup owns(S)
 \end{aligned}$$

Underlying low equivalence. The following is a definition of low equivalence between states of monitored executions.

Definition A.1 (Low Equivalent Thread Monitoring States).

Two monitored execution states $((\sigma_1, \lambda_1), q_1)$ and $((\sigma_2, \lambda_2), q_2)$ are low equivalent, written $((\sigma_1, \lambda_1), q_1) \cong ((\sigma_2, \lambda_2), q_2)$, if and only if:

- $q_1 = q_2 = (V, W, L, w)$ and $w \in \mathcal{L}(\perp^*)$,
- $\forall x \notin V : \sigma_1(x) = \sigma_2(x)$,
- $\forall x \notin L : x \in \lambda_1 \Leftrightarrow x \in \lambda_2$.

By extension, two execution states — (ς_1, Q_1) and (ς_2, Q_2) — of a pool of threads Θ are considered low equivalent, written $(\varsigma_1, Q_1) \cong (\varsigma_2, Q_2)$, if and only if, for all ι in the domain of Θ , $(\varsigma_1, \text{extractState}(Q_1, \iota)) \cong (\varsigma_2, \text{extractState}(Q_2, \iota))$.

State coherency.

Definition A.2 (State coherency).

A monitored execution state ζ — equals to $((\sigma, \lambda), (V, W, L, w))$ — is coherent with an monitored statement S , written $\zeta \vdash S$, if and only if:

- $\forall x : x \in W \Rightarrow x \in V$,
- $S = \otimes(P^e, P^u)[S^b] \Rightarrow \text{modify}(S^b) \subseteq \text{modify}(P^e)$,
- $S = \otimes(P^e, P^u)[S^b] \Rightarrow (\text{needs}(S^b) \cup \text{owns}(S^b)) \subseteq \text{needs}(P^e)$,
- $S = \otimes(P^e, P^u)[S^b] \Rightarrow (\neg \text{stops}(P^e) \Rightarrow \neg \text{stops}(S^b))$,
- $(S = \otimes(P^e, P^u)[S^b] \wedge \exists w' \text{ s.t. } w = \top w') \Rightarrow \text{modify}(P^e) \cup \text{modify}(P^u) \subseteq W$,
- $(S = \otimes(P^e, P^u)[S^b] \wedge \exists w' \text{ s.t. } w = \top w') \Rightarrow \text{needs}(P^e) \cup \text{needs}(P^u) \subseteq L$,
- $(S = \otimes(P^e, P^u)[S^b] \wedge \exists w' \text{ s.t. } w = \top w') \Rightarrow (\forall l \in (\text{needs}(S^b) \setminus \text{owns}(S^b)) : l \notin \lambda)$,
- $S = \odot \bar{x}[S'] \Rightarrow ((\sigma, \lambda \setminus \bar{x}), (V, W, L, w)) \vdash S'$,
- $(S = \otimes(P^e, P^u)[S'] \wedge \exists w' \text{ s.t. } w = aw') \Rightarrow (\varsigma, (V, W, L, w')) \vdash S'$,

Evaluation context. The proof will make use of *evaluation contexts* [NRNH99, Sect. 5.5.1].

Definition A.3 (Evaluation context).

An evaluation context is a monitored statement containing a *unique hole* which is written \square . Evaluation contexts are described by the following grammar:

$$\begin{aligned} \langle \text{eval-context} \rangle ::= & \square \mid \langle \text{eval-context} \rangle ; \langle \text{prog} \rangle \\ & \mid \odot \langle \text{ident-set} \rangle [\langle \text{eval-context} \rangle] \\ & \mid \otimes (\langle \text{prog} \rangle, \langle \text{prog} \rangle) [\langle \text{eval-context} \rangle] \end{aligned}$$

Definition A.4 (Evaluation context as a function).

Every evaluation context E is overloaded such that it is also a function mapping monitored statements to monitored statements. This function returns the monitored statements corresponding to E where \square is replaced by the function's argument. Formally, this function is defined as follows:

$$E(S) = \begin{cases} S & \text{if and only if } E = \square \\ E'(S) ; P & \text{if and only if } E = E' ; P \\ \odot \bar{x}[E'(S)] & \text{if and only if } E = \odot \bar{x}[E'] \\ \otimes (P_1, P_2)[E'(S)] & \text{if and only if } E = \otimes (P_1, P_2)[E'] \end{cases}$$

Definition A.5 (Context word of an evaluation context).

During any evaluation, a context word w evolves with the monitored statement S under evaluation. This statement can be expressed using an evaluation context E and another monitored statement S' — $S = E(S')$. $(w \downarrow_E)$ is the prefix of w which has been created in the same time as the evaluation context E . Formally, this function is defined as follows:

$$(w \downarrow_E) = \begin{cases} \epsilon & \text{if } E = \square \\ (w \downarrow_{E'}) & \text{if } E = E' ; P \\ (w \downarrow_{E'}) & \text{if } E = \odot \bar{x}[E'] \\ a (w' \downarrow_{E'}) & \text{if } E = \otimes (P^e, P^u)[E'] \text{ and } w = aw' \end{cases}$$

Bi-similarity

Definition A.6 (Bi-similarity of execution stages).

An execution stage is a pair of a monitored execution state and a monitored statement. The execution stage $(\zeta_1 \vdash S_1)$ is bi-similar to the execution stage $(\zeta_2 \vdash S_2)$, written $(\zeta_1 \vdash S_1) \sim (\zeta_2 \vdash S_2)$, if and only if they are coherent — $\zeta_1 \vdash S_1$ and $\zeta_2 \vdash S_2$ — and one, at least, of the following is true:

$$A.6_{\triangleleft_1} \quad S_1 = S_2 \text{ and } \zeta_1 \cong \zeta_2,$$

$A.6_{\triangleleft_2}$ there exist an evaluation context E , four programs $P_1^c, P_1^u, P_2^c, P_2^u$ and two monitored statements S_1^b and S_2^b such that all the following is true:

- (a) $S_1 = E(\otimes(P_1^c, P_1^u)[S_1^b])$ and $S_2 = E(\otimes(P_2^c, P_2^u)[S_2^b])$,
- (b) $V_1 = V_2, W_1 = W_2, L_1 = L_2$ and $(w_1 \downarrow_E) = (w_2 \downarrow_E)$,
- (c) $\forall i \in \{1, 2\} : \exists w_i^s \text{ s.t. } w_i = (w_i \downarrow_E) \top w_i^s$,
- (d) $\forall x \notin V_1 : \sigma_1(x) = \sigma_2(x)$,
- (e) $\forall l \notin L_1 : l \in \lambda_1 \Leftrightarrow l \in \lambda_2$,
- (f) $(\neg \text{stops}(P_1^c) \wedge \neg \text{stops}(P_1^u)) \Leftrightarrow (\neg \text{stops}(P_2^c) \wedge \neg \text{stops}(P_2^u))$.

By extension, two execution states — (ς_1, Q_1) and (ς_2, Q_2) — and two pool of threads — Θ_1 and Θ_2 — are considered bi-similar, written $((\varsigma_1, Q_1) \vdash \Theta_1) \sim ((\varsigma_2, Q_2) \vdash \Theta_2)$, if and only if, for all t in the domain of Θ , $((\varsigma_1, \text{extractState}(Q_1, t)) \vdash \Theta_1(t)) \sim ((\varsigma_2, \text{extractState}(Q_2, t)) \vdash \Theta_2(t))$.

A.1.2 Lemmas

Lemma A.1 (Evaluation Context Generic Evaluation Rule). *For all evaluation contexts E , monitored statements S and S' , and monitored execution states $((\sigma, \lambda), q)$ and $((\sigma', \lambda'), q')$, $E(S)$ in state $((\sigma, \lambda), q)$ evaluates to $E(S')$ yielding state $((\sigma', \lambda' \cup \text{owns}(E(\emptyset))), q')$ and output o in n steps if and only if S in state $((\sigma, \lambda \setminus \text{owns}(E(\emptyset))), q)$ evaluates to S' yielding state $((\sigma', \lambda'), q')$ and output o in n steps.*

Proof. It follows directly from the definitions of evaluation context (Definition A.3 and A.4) and the semantics rules applying to $E(S)$. $E'(S) ; P$ takes one evaluation step if and only if $E'(S)$ takes one evaluation step in the same state. From the rule $(E_S\text{-LOCKED}_{\text{step}}, \odot \bar{x}[E'(S)])$ takes one evaluation step in state $((\sigma, \lambda), q)$ yielding state $((\sigma', \lambda' \cup \text{owns}(E(\emptyset))), q')$ if and only if $E'(S)$ takes one evaluation step in state $((\sigma, \lambda \setminus \text{owns}(\odot \bar{x}[\emptyset])), q)$ yielding state $((\sigma', \lambda'), q')$. $\otimes(P_1, P_2)[E'(S)]$ takes one evaluation step if and only if $E'(S)$ takes one evaluation step in the same state. \square

Lemma A.2 (Branching Context of Evaluation Context Preservation). *For all evaluation contexts E , monitored statements S and S' , and monitored execution states ζ and ζ' :*

$$\star_1 \quad \langle \zeta \vdash E(S) \rangle \xrightarrow{o} \mathbf{M}(\mathbf{O}) \quad \langle \zeta' \vdash E(S') \rangle$$

implies that $(w \downarrow_E) = (w' \downarrow_E)$.

Proof. The proof follows directly from Definitions A.4 and A.5 and the semantics rules of Figure 8 going by structural induction on E . \square

Lemma A.3 (Coherency preservation). *For all monitored statements S and S' , and monitored execution states ζ and ζ' such that:*

- ★₁ $\zeta \vdash S$,
- ★₂ $\langle \zeta \vdash S \rangle \xrightarrow{\circ} \mathbf{M}(\mathbf{O}) \langle \zeta' \vdash S' \rangle$,

it is true that:

- $\zeta' \vdash S'$.

This lemma states that coherency of monitored execution states is an execution invariant.

Proof. The proof of this lemma goes by structural induction on S . If S is an atomic action, only the first property ($\forall x : x \in W \Rightarrow x \in V$) can apply. It is then obvious from the semantics rules of Figure 8 and the transition function of Figure 6 that $\zeta' \vdash S'$. For *branched statements*, the properties are obvious after deducing from the semantics rules that initially such a statement is always of the form $\otimes(P^e, P^e)[P^e]$. For *locked statements*, it follows from the semantics rule for such statements which temporarily releases the locks owned by a thread before executing one step for this thread. \square

Lemma A.4 (Bi-similarity preservation). *For all monitored statements S_1 and S_2 , and monitored execution states $\zeta_1, \zeta'_1, \zeta_2$ and ζ'_2 such that:*

- ★₁ $(\zeta_1 \vdash S_1) \sim (\zeta_2 \vdash S_2)$,
- ★₂ $V'_1 = V'_2, W'_1 = W'_2, L'_1 = L'_2, w'_1 = w_1$ and $w'_2 = w_2$,
- ★₃ $\forall x \notin V'_1 : \sigma'_1(x) = \sigma'_2(x)$,
- ★₄ $\forall l \notin L'_1 : l \in \lambda'_1 \Leftrightarrow l \in \lambda'_2$,

it is true that $(\zeta'_1 \vdash S_1) \sim (\zeta'_2 \vdash S_2)$.

Proof. This theorem is straightforward. It follows directly from Definition A.6. Every property in this definition involving the execution state is either one of ★₂, ★₃, ★₄ or follows directly from one of them. \square

Lemma A.5 (High Context Evaluation Step). *For all evaluation contexts E , monitored statements S and S' , output sequences o and monitored execution states $((\sigma, \lambda), (V, W, L, w))$ and $((\sigma', \lambda'), (V', W', L', w'))$ such that:*

- ★₁ $((\sigma, \lambda), (V, W, L, w)) \vdash S$,
- ★₂ $\langle ((\sigma, \lambda), (V, W, L, w)) \vdash E(S) \rangle \xrightarrow{\circ} \mathbf{M}(\mathbf{O}) \langle ((\sigma', \lambda'), (V', W', L', w')) \vdash E(S') \rangle$,
- ★₃ $(w \downarrow_E) \notin \mathcal{L}(\perp^*)$,

it is true that:

- o is equal to ϵ ,
- $V' = V, W' = W, L' = L$ and $(w' \downarrow_E) = (w \downarrow_E)$,

- $\forall x \notin \text{modify}(S) : \sigma'(x) = \sigma(x)$,
- $\forall l \notin \text{needs}(S) \cup \text{owns}(S) : l \in \lambda' \Leftrightarrow l \in \lambda$.

Proof. This is trivial and follows directly from the transition function of Figure 6. Because of $\star_{A.1.2}$ any output is forbidden and any possible transition does not modifies V , W , L and $(w \downarrow_E)$; except for assignments. However, in the case of an assignment to x , because of \star_1 and \star_2 , x belongs to W and V so the transition has no effect on the automaton state and the property on σ and σ' holds. Finally, for any lock which is not needed or already owned for the execution of S , an evaluation step of S will not modify the fact that l belongs or not to λ . \square

Lemma A.6 (High Branch Traversal). *For all evaluation contexts E , monitored statements S and monitored execution states $((\sigma, \lambda), (V, W, L, w))$ such that:*

- $\star_1 ((\sigma, \lambda), (V, W, L, w)) \vdash E(S)$,
- $\star_2 \lambda \cap (\text{needs}(S) \setminus \text{owns}(S)) = \emptyset$,
- $\star_3 \neg \text{stops}(S)$,
- $\star_4 (w \downarrow_E) \notin \mathcal{L}(\perp^*)$,

it is true that, there exists a monitored execution state $((\sigma', \lambda'), (V', W', L', w'))$ such that:

- $\Downarrow((\sigma, \lambda), (V, W, L, w)) \vdash E(S) \Downarrow \xrightarrow{\xi} \mathbf{M}(\mathbf{O}) \Downarrow((\sigma', \lambda'), (V', W', L', w')) \vdash E(\emptyset) \Downarrow$.

Proof. The proof of this lemma goes by structural induction on S . Such induction goes by reducing the “structural size” of S at any induction step. Usually this type of induction does not work well with while-statements. For this proof, the hypothesis \star_3 solves the problem of while-statements. Synchronization commands can also be a problem. This one is solved by the hypotheses \star_3 and \star_2 . Additionally, in order for the proof to go smoothly, it is required to give a smaller “structural size” to the branched statement structure than to the structure of if-statements and while-statements; similarly for locked statements and synchronization commands. The small details of the proof follows directly from Lemma A.5. \square

Lemma A.7 (One Step Soundness). *For all monitored statements S and S' , and monitored execution states ζ_1 , ζ'_1 and ζ_2 such that:*

$$\star_1 \zeta_1 \cong \zeta_2,$$

$$\star_2 \zeta_1 \vdash S,$$

$$\star_3 \langle \zeta_1 \vdash S \rangle \xrightarrow{o} \mathbf{M}(\mathbf{O}) \langle \zeta'_1 \vdash S' \rangle,$$

it is true that, there exists a monitored execution state ζ'_2 such that $\zeta'_1 \cong \zeta'_2$ and one of the following is true:

$$\triangleleft_1 \langle \zeta_2 \vdash S \rangle \xrightarrow{o} \mathbf{M}(\mathbf{O}) \langle \zeta'_2 \vdash S' \rangle,$$

\triangleleft_2 *all the following is true:*

(a) *there exist an evaluation context E and three monitored statements S^b , P^e and P^u such that S is equal to $E(S^b)$, S' is equal to $E(\otimes(P^e, P^u)[P^e])$ and $w_1 = (w_1 \downarrow_E)$,*

$$(b) w'_2 = w_2 \top,$$

$$(c) o = \epsilon,$$

$$(d) \langle \zeta_2 \vdash S \rangle \xrightarrow{\epsilon} \mathbf{M}(\mathbf{O}) \langle \zeta'_2 \vdash E(\otimes(P^u, P^e)[P^u]) \rangle.$$

Proof sketch. The idea behind this lemma is to state that two executions of the same statement started in low-equivalent states either take the same execution step or diverge because it was a branching statement and both execution take different branch with the implication that the branching condition was influenced by some secret value. \square

Formal proof. The proof goes by induction on the length of the derivation tree of the global hypothesis \star_3 and by cases on the first rule used for this derivation tree. If the first rule used for the derivation tree of the global hypothesis \star_3 is:

(E_M-OK) then we can conclude that :

(1) S is an action (i.e. $S = \text{“skip”}$ or $S = \text{“}y := e\text{”}$ or $S = \text{“output } e\text{”}$), $(q_1, S) \xrightarrow{\text{OK}} q'_1$ and $\langle \zeta_1 \vdash S \rangle \xrightarrow{o} \langle \zeta'_1 \vdash \emptyset \rangle$.

It follows directly from the definition of the rule (E_M-OK) and the grammar of the language. It can also be deduced from the rule (E_M-OK), the transition function of the monitoring automaton, and the semantics (E_S).

(2) There exists q'_2 such that $(q_2, S) \xrightarrow{\text{OK}} q'_2$ and $q'_1 = q'_2$.

It follows directly from the local conclusion (1) and the global hypothesis \star_1 — as well as the definition of low equivalence (Definition A.1).

(•) There exists ζ'_2 such that $\langle \zeta_2 \vdash S \rangle \xrightarrow{o} \mathbf{M}(\mathbf{O}) \langle \zeta'_2 \vdash S' \rangle$ and $\zeta'_1 \cong \zeta'_2$.

Case 1: $S = \text{skip}$

- (a) There exists ζ'_2 such that $\langle \zeta_2 \vdash A \rangle \xrightarrow{o}_s \langle \zeta'_2 \vdash \emptyset \rangle$ and $\zeta'_1 = \zeta'_2$.
It follows directly from the case hypothesis, the rule (E_s-SKIP), the local conclusion (1) and the global hypothesis \star_1 .
- (•) There exists ζ'_2 such that $\langle \zeta_2 \vdash S \rangle \xrightarrow{o}_{M(O)} \langle \zeta'_2 \vdash S' \rangle$ and $\zeta'_1 \cong \zeta'_2$.
It follows directly from the the rule (E_M-OK) and the local conclusions (2) and (a).

Case 2: S = output e

- (a) $\mathcal{FV}(e) \cap V_1 = \emptyset$.
It follows from the case hypothesis, the fact that “ $(q_1, S) \xrightarrow{OK} q'_1$ ” (from the local conclusion (1)), and the only possible automaton transition (T-OUTPUT-ok).
- (b) $\sigma_1(e) = \sigma_2(e)$.
It follows directly from the local conclusion (a), the global hypothesis \star_1 and the definition of low equivalence (Definition A.1).
- (c) There exists ζ'_2 such that $\langle \zeta_2 \vdash A \rangle \xrightarrow{o}_s \langle \zeta'_2 \vdash \emptyset \rangle$ and $\zeta'_1 = \zeta'_2$.
It follows directly from the case hypothesis, the rule (E_s-OUTPUT), the global hypothesis \star_1 and the local conclusions (1) and (b).
- (•) There exists ζ'_2 such that $\langle \zeta_2 \vdash S \rangle \xrightarrow{o}_{M(O)} \langle \zeta'_2 \vdash S' \rangle$ and $\zeta'_1 \cong \zeta'_2$.
It follows directly from the the rule (E_M-OK) and the local conclusions (2) and (c).

Case 3: S = $y := e$

- (a) $\sigma'_1 = \sigma_1[y \mapsto \sigma_1(e)]$ and $\lambda_1 = \lambda'_1$.
It follows directly from the case hypothesis, the local conclusion (1) and the only possible rule (E_s-ASSIGN).
- (b) There exist σ'_2 and λ'_2 such that $\langle \zeta_2 \vdash S \rangle \xrightarrow{o}_s \langle \zeta'_2 \vdash \emptyset \rangle$, $\sigma'_2 = \sigma_2[y \mapsto \sigma_2(e)]$ and $\lambda_2 = \lambda'_2$.
It follows directly from the case hypothesis and the only possible rule (E_s-ASSIGN).
- (c) There exists ζ'_2 such that $\langle \zeta_2 \vdash S \rangle \xrightarrow{o}_{M(O)} \langle \zeta'_2 \vdash S' \rangle$.
It follows from the local conclusions (2) and (b).
- (d) $\forall x \notin V'_1 : \sigma'_1(x) = \sigma'_2(x)$.

Sub-case 2.a: $FV(e) \cap V_1 \neq \emptyset$ or $y \in W_1$.

- (α) $V'_1 = V_1 \cup \{y\}$.
From the sub-case hypothesis, the automaton transition of the local conclusion (1) is (T-ASSIGN-sec). From which comes the desired result.
- (•) $\forall x \notin V'_1 : \sigma'_1(x) = \sigma'_2(x)$.
It follows directly from the global hypothesis \star_1 and local conclusions (a), (b) and (α).

Sub-case 2.b: $FV(e) \cap V_1 = \emptyset$ and $y \notin W_1$.

- (α) $V'_1 = V_1 \setminus \{y\}$.
From the sub-case hypothesis, the automaton transition of the local conclusion (1) is (T-ASSIGN-pub). From which comes the desired result.
- (β) $\sigma_1(e) = \sigma_2(e)$.
It follows directly from the global hypothesis \star_1 and the sub-case hypothesis $FV(e) \cap V_1 = \emptyset$.

- $\forall x \notin V'_1 : \sigma'_1(x) = \sigma'_2(x)$.

It follows directly from the global hypothesis \star_1 and the local conclusions (α), (a), (b) and (β).

- There exists ζ'_2 such that $\langle \zeta_2 \vdash S \rangle \xrightarrow{\circ} \mathbb{M}(\mathcal{O}) \langle \zeta'_2 \vdash S' \rangle$ and $\zeta'_1 \cong \zeta'_2$.

The first part of the above result comes directly from the local conclusion (c). The second part of the result — $\zeta'_1 \cong \zeta'_2$ — comes from the definition A.1, the local conclusions (2) for the first part of the definition, the local conclusions (d) for the second part of the definition, and the global hypothesis \star_1 and the local conclusions (a) and (b) for the last part.

(E_M-EDIT) then we can conclude that :

(1) the following holds:

- $(q_1, S) \xrightarrow{\text{output } \delta} q'_1$,
- S is “**output e**”,
- $w_1 \in \mathcal{L}(\perp^*)$ and $\mathcal{FV}(e) \cap V_1 \neq \emptyset$,
- $q'_1 = q_1$,
- $o = \delta$,
- $s'_1 = s_1$.

It follows directly from the definition of the rule (E_M-EDIT), the only automaton transition returning an action as output (T-OUTPUT-def) and the rule (E_S-OUTPUT).

- (2) There exists q'_2 such that $(q_2, S) \xrightarrow{\text{output } \delta} q'_2$ and $q'_1 = q'_2$.

From the local conclusion (1), $w_1 \in \mathcal{L}(\perp^*)$ and $\mathcal{FV}(e) \cap V_1 \neq \emptyset$. Hence, from the global hypothesis \star_1 , $w_2 \in \mathcal{L}(\perp^*)$ and $\mathcal{FV}(e) \cap V_2 \neq \emptyset$. Then the automaton transition (T-OUTPUT-def) applies to q_2 ; with the consequences that $q'_2 = q_2$. Hence, from the global hypothesis \star_1 , $q'_1 = q'_2$.

- There exists ζ'_2 such that $\langle \zeta_2 \vdash S \rangle \xrightarrow{\circ} \mathbb{M}(\mathcal{O}) \langle \zeta'_2 \vdash S' \rangle$ and $\zeta'_1 \cong \zeta'_2$.

The rule (E_S-OUTPUT) implies that there exists ζ'_2 such that $\langle \zeta_2 \vdash \text{output } \delta \rangle \xrightarrow{\delta}_s \langle \zeta'_2 \vdash \emptyset \rangle$ with $\zeta'_2 = \zeta_2$. Hence the rule (E_M-EDIT) and the local conclusion (2) imply that $\langle \zeta_2 \vdash S \rangle \xrightarrow{\circ} \mathbb{M}(\mathcal{O}) \langle \zeta'_2 \vdash S' \rangle$. From the global hypothesis \star_1 and the local conclusions (1) and (2), $q'_1 = q'_2$. Hence from the global hypothesis \star_1 , the fact that $s'_1 = s_1$ (from the local conclusion (1)) and the previous conclusion $\zeta'_2 = \zeta_2$, $\zeta'_1 \cong \zeta'_2$.

(E_M-NO) then we can conclude that :

- (1) $w_1 \notin \mathcal{L}(\perp^*)$. It follows directly from the definition of the rule (E_M-NO) and the only automaton transition denying an action (T-OUTPUT-no).
- The first rule used can not be (E_M-NO). It follows from the fact that the local conclusion (1) is in contradiction with the global hypothesis \star_1 .

(E_M-IF) then we can conclude that :

- (1) there exists v belonging to the set true, false such that:

- S is “if e then P^{true} else P^{false} end”,
- $\sigma_1(e) = v$,
- $(q_1, \text{branch}(\lambda_1, e, P^v, P^{-v})) \xrightarrow{OK} q'_1$,
- $S'_1 = S_1$,
- S' is “ $\otimes(P^v, P^{-v})[P^v]$ ”.

It follows directly from the rule (E_M-IF).

- the property holds.

Case 1: $\sigma_2(e) = \sigma_1(e)$

- (a) there exists q'_2 such that $(q_2, \text{branch}(\lambda_2, e, P^v, P^{-v})) \xrightarrow{OK} q'_2$ and $q'_2 = q'_1$.
The conditions of the two possible transitions for the automaton input $\text{branch}(\lambda_2, e, P^v, P^{-v})$ are based on values which, because of the global hypothesis \star_1 , are equal for q_2 and q_1 . Therefore, the same transition can be taken by the automaton.
- There exists ζ'_2 such that $\langle \zeta_2 \vdash S \rangle \xrightarrow{M(O)} \langle \zeta'_2 \vdash S' \rangle$ and $\zeta'_1 \cong \zeta'_2$.
It follows from the rule (E_M-IF), the global hypothesis \star_1 , the case hypothesis and the local conclusions (1) and (a).

Case 2: $\sigma_2(e) \neq \sigma_1(e)$

- (a) there exist an evaluation context E and a monitored statement S^b such that S is equal to $E(S^b)$ and S' is equal to $E(\otimes(P^v, P^{-v})[P^v])$.
It follows directly from the rule (E_M-IF). E is \square and S^b is S .
- (b) $FV(e) \cap V_1 \neq \emptyset$.
It follows by contradiction using the global hypothesis \star_1 and the case hypothesis.
- (c) there exists w^p such that $w'_1 = w^p \top$.
The local conclusion (b) implies that the transition used for the automaton input $\text{branch}(\lambda, e, P^v, P^{-v})$ in state q_1 is (T-BRANCH-high). Hence, the desired result holds.
- (d) there exists q'_2 such that $(q_2, \text{branch}(\lambda, e,),) \xrightarrow{OK} q'_2$ and $q'_2 = q'_1$.
The conditions of the two possible transitions for the automaton input $\text{branch}(\lambda_2, e, P^v, P^{-v})$ are based on values which, because of the global hypothesis \star_1 , are equal for q_2 and q_1 . Therefore, the same transition can be taken by the automaton.
- (e) there exists ζ'_2 such that $\langle \zeta_2 \vdash S \rangle \xrightarrow{M(O)} \langle \zeta'_2 \vdash E(\otimes(P^{-v}, P^v)[P^{-v}]) \rangle$ and $\zeta'_1 \cong \zeta'_2$.
The rule (E_M-IF) and the local conclusions (1) and (d) imply the desired result.
- the second possible global conclusion holds.
It follows directly from the local conclusions (1), (a), (c), (d) and (e).

(E_M-WHILE) then we can conclude that :

- the property holds.
The proof is similar to the one for (E_M-IF).

(E_M-WITH) then we can conclude that :

- (1) there exists q_1^l such that:
- S is **with \bar{x} when e do S^w done**,
 - $\bar{x} \cap \lambda_1 = \emptyset$,
 - $\sigma_1(e) = \mathbf{true}$,
 - $(q_1, \mathbf{sync}(\bar{x}, e)) \xrightarrow{OK} q_1'$,
 - $FV(e) \cap V_1 = \emptyset$,
 - $L_1 \cap \bar{x} = \emptyset$,
 - $\zeta_1' = ((\sigma_1, \lambda_1 \cup \bar{x}), q_1')$,
 - $S' = \odot \bar{x}[S^w]$.

It follows directly from the global hypothesis \star_1 , the semantics rule (E_M-WITH) and from the only automaton transition applying to the automaton input.

- (2) $\bar{x} \cap \lambda_2 = \emptyset$ and $\sigma_2(e) = \mathbf{true}$.
Both equalities follow from the global hypothesis \star_1 , the definition of low equivalence for states of monitored executions (definition A.1) and the local conclusion (1) — $L_1 \cap \bar{x} = \emptyset$ and $FV(e) \cap V_1 = \emptyset$.
- (3) There exists q_2' such that $(q_2, \mathbf{sync}(\bar{x}, e)) \xrightarrow{OK} q_2'$ and $q_2' = q_1'$.
The global hypothesis \star_1 and the local conclusion (1) imply that $L_2 \cap \bar{x} = \emptyset$ and $FV(e) \cap V_2 = \emptyset$. Therefore, the desired results hold.
- (4) There exists ζ_2' such that $\langle \zeta_2 \vdash S \rangle \xrightarrow{\circ} \mathbf{M}(\mathbf{O}) \langle \zeta_2' \vdash S' \rangle$ and $\zeta_2' = ((\sigma_2, \lambda_2 \cup \bar{x}), q_2')$.
It follows from the rule (E_M-WITH) and the local conclusions (2) and (3).
- (5) $\forall x \notin V_1' : \sigma_1'(x) = \sigma_2'(x)$.
The local conclusion (1) and the automaton transition (T-SYNC) imply that V_1' is equal to V_1 . In addition, the evaluation rule (E_M-WITH), the global hypothesis \star_3 and the local conclusion (4) imply that σ_1' is equal to σ_1 and σ_2' is equal to σ_2 . Therefore, the desired result follows from the global hypothesis \star_1 .
- (6) $\forall x \notin L_1' : x \in \lambda_1' \Leftrightarrow x \in \lambda_2'$.
The local conclusion (1) and the automaton transition (T-SYNC) imply that $L_1' = L_1$. In addition, the evaluation rule (E_M-WITH), the global hypothesis \star_3 and the local conclusion (4) imply that $\lambda_1' = \lambda_1 \cup \bar{x}$ and $\lambda_2' = \lambda_2 \cup \bar{x}$. So the desired result is implied by the global hypothesis \star_1 .
- (•) There exists ζ_2' such that $\langle \zeta_2 \vdash S \rangle \xrightarrow{\circ} \mathbf{M}(\mathbf{O}) \langle \zeta_2' \vdash S' \rangle$ and $\zeta_1' \cong \zeta_2'$.
It follows directly from the local conclusions (4), (5) and (6).

(E_M-SEQ_{exit}) then we can conclude that :

- (1) S is \emptyset ; $S^r, o = \epsilon, \zeta_1' = \zeta_1$ and $S' = S^r$.
It follows directly from (E_M-SEQ_{exit}).
- (•) There exists ζ_2' such that $\langle \zeta_2 \vdash S \rangle \xrightarrow{\circ} \mathbf{M}(\mathbf{O}) \langle \zeta_2' \vdash S' \rangle$ and $\zeta_1' \cong \zeta_2'$.
It follows directly from (E_M-SEQ_{exit}) and the global hypothesis \star_1 .

(E_M-SEQ_{step}) then we can conclude that :

- (1) the following holds:

- S is $S^i ; S^r$,
- $\langle \zeta_1 \Vdash S^i \rangle \xrightarrow{o}_M \langle \zeta'_1 \Vdash S^j \rangle$ is a sub-derivation tree of the derivation tree of S ,
- $S' = S^j ; S^r$.

It follows directly from the rule (E_M -SEQ_{step}).

- (2) There exists ζ'_2 such that $\zeta'_1 \cong \zeta'_2$ and one of the possible global conclusions holds for the monitored statement S^i .

It follows directly from the inductive hypothesis.

- (•) The property holds.

Case 1: the first global conclusion holds for S^i .

- (a) $\langle \zeta_2 \vdash S^i \rangle \xrightarrow{o}_M(O) \langle \zeta'_2 \vdash S^j \rangle$

It follows directly from the local conclusion (2) and the case hypothesis.

- (•) $\langle \zeta_2 \vdash S \rangle \xrightarrow{o}_M(O) \langle \zeta'_2 \vdash S' \rangle$

It follows directly from the local conclusion (a) and the definition of the evaluation rule (E_M -SEQ_{step}).

Case 2: the second global conclusion holds for S^i .

- (a) there exist an evaluation context E and three monitored statements S^b , P^e and P^u such that:

- S^i is equal to $E(S^b)$ and S^j is equal to $E(\otimes(P^e, P^u)[P^e])$,
- $w'_2 = w_2 \top$,
- $o = \epsilon$,
- $\langle \zeta_2 \vdash S^i \rangle \xrightarrow{\epsilon}_M(O) \langle \zeta'_2 \vdash E(\otimes(P^u, P^e)[P^u]) \rangle$.

It follows directly from the local conclusion (2) and the case hypothesis.

- (b) $\langle \zeta_2 \vdash S^i ; S^r \rangle \xrightarrow{\epsilon}_M(O) \langle \zeta'_2 \vdash E(\otimes(P^u, P^e)[P^u]) ; S^r \rangle$

It follows directly from the local conclusion (a) and the evaluation rule (E_M -SEQ_{step}).

- (c) there exist E' , equal to “ $E ; S^r$ ”, such that S is equal to $E'(S^b)$ and S' is equal to $E(\otimes(P^e, P^u)[P^e])$.

This result is implied by the local conclusions (1) and (a).

- (d) $\langle \zeta_2 \vdash S \rangle \xrightarrow{\epsilon}_M(O) \langle \zeta'_2 \vdash E'(\otimes(P^u, P^e)[P^u]) \rangle$.

From the local conclusion (c), $E'(\otimes(P^u, P^e)[P^u])$ is equal to “ $E(\otimes(P^u, P^e)[P^u]) ; S^r$ ”. Hence, the local conclusions (1) and (b) imply the desired result.

- (•) the second possible global conclusion holds.

It follows directly from the local conclusions (c), (a) and (d).

(E_M -BRANCH_{exit}) then we can conclude that :

- (1) there exists q_1^l such that:

- S is $\otimes(P^e, P^u)[\emptyset]$,
- $(q_1, \text{merge}(P^e, P^u)) \xrightarrow{OK} q_1$,
- $S'_1 = S_1$
- S' is \emptyset .

It follows directly from the rule (E_M -BRANCH_{exit}) and the global hypothesis \star_1 .

- (2) There exists q'_2 such that $(q_2, \text{merge}(P^e, P^u)) \xrightarrow{OK} q_2$.
It follows directly from the global hypothesis \star_1 and the definition of the transition (T-MERGE-low).
- (•) There exists ζ'_2 such that $\langle \zeta_2 \vdash S \rangle \xrightarrow{O} \mathbf{M}(\mathbf{O}) \langle \zeta'_2 \vdash S' \rangle$ and $\zeta'_1 \cong \zeta'_2$.
This follows directly from the global hypothesis \star_1 and local conclusions (1) and (2).

(\mathbf{E}_M -BRANCH_{step}) then we can conclude that :

- (1) there exist three monitoring statements $S^b, S^{b'}, P^e$ and P^u such that:
- $S^b \neq \emptyset$,
 - S is equal to $\otimes(P^e, P^u)[S^b]$,
 - S' is equal to $\otimes(P^e, P^u)[S^{b'}]$,
 - $\langle \zeta_1 \Vdash S^b \rangle \xrightarrow{o} \mathbf{M} \langle \zeta'_1 \Vdash S^{b'} \rangle$ is a sub-derivation tree of the global hypothesis \star_3 .
- It follows directly from the definition of (\mathbf{E}_M -BRANCH_{step}).
- (2) There exists ζ'_2 such that $\zeta'_1 \cong \zeta'_2$ and one of the possible global conclusions holds for the monitored statement S^b .
It follows directly from the inductive hypothesis and the local conclusion (1).
- (•) The property holds.

Case 1: the first global conclusion holds for S^b .

- (a) $\langle \zeta_2 \Vdash S^b \rangle \xrightarrow{o} \mathbf{M} \langle \zeta'_2 \Vdash S^{b'} \rangle$
It follows directly from the local conclusion (2) and the case hypothesis.
- (•) There exists ζ'_2 such that $\langle \zeta_2 \Vdash S \rangle \xrightarrow{O} \mathbf{M}(\mathbf{O}) \langle \zeta'_2 \Vdash S' \rangle$ and $\zeta'_1 \cong \zeta'_2$.
This is direct using the definition of the rule (\mathbf{E}_M -BRANCH_{step}) and the local conclusion (a).

Case 2: the second global conclusion holds for S^b .

- (a) there exist an evaluation context E and three monitored statements S^{b_2}, P^{e_2} and P^{u_2} such that:
- S^b is equal to $E(S^{b_2})$ and $S^{b'}$ is equal to $E(\otimes(P^{e_2}, P^{u_2})[P^{e_2}])$,
 - $w'_2 = w_2 \top$,
 - $o = \epsilon$,
 - $\langle \zeta_2 \vdash S^b \rangle \xrightarrow{\epsilon} \mathbf{M}(\mathbf{O}) \langle \zeta'_2 \vdash E(\otimes(P^{u_2}, P^{e_2})[P^{u_2}]) \rangle$.
- It follows directly from the local conclusion (2) and the case hypothesis.
- (b) $\langle \zeta_2 \vdash \otimes(P^e, P^u)[S^b] \rangle \xrightarrow{\epsilon} \mathbf{M}(\mathbf{O}) \langle \zeta'_2 \vdash \otimes(P^e, P^u)[E(\otimes(P^{u_2}, P^{e_2})[P^{u_2}])] \rangle$
It follows directly from the local conclusion (a) and the evaluation rule (\mathbf{E}_M -BRANCH_{step}).
- (c) there exist E' , equal to “ $\otimes(P^e, P^u)[E]$ ”, such that S is equal to $E'(S^{b_2})$ and S' is equal to $E(\otimes(P^{e_2}, P^{u_2})[P^{e_2}])$.
This result is implied by the local conclusions (1) and (a).
- (d) $\langle \zeta_2 \vdash S \rangle \xrightarrow{\epsilon} \mathbf{M}(\mathbf{O}) \langle \zeta'_2 \vdash E'(\otimes(P^{u_2}, P^{e_2})[P^{u_2}]) \rangle$.
From the local conclusion (c), $E'(\otimes(P^{u_2}, P^{e_2})[P^{u_2}])$ is equal to “ $\otimes(P^e, P^u)[E(\otimes(P^{u_2}, P^{e_2})[P^{u_2}])]$ ”. Hence, the local conclusions (1) and (b) imply the desired result.

- the second possible global conclusion holds.
It follows directly from the local conclusions (c), (a) and (d).

(E_M-LOCKED_{exit}) then we can conclude that :

(1) there exists a set of identifiers \bar{x} such that:

- S is $\odot\bar{x}[\emptyset]$,
- S' is \emptyset ,
- $\zeta'_1 = ((\sigma_1, \lambda_1 \setminus \bar{x}), q_1)$.

It follows directly from the rule (E_M-LOCKED_{exit}).

- There exists ζ'_2 such that $\langle \zeta_2 \vdash S \rangle \xrightarrow{\circ} \mathbf{M}(\mathbf{O}) \langle \zeta'_2 \vdash S' \rangle$ and $\zeta'_1 \cong \zeta'_2$.
This follows directly from the local conclusions (1), the global hypothesis \star_1 and the rule (E_M-LOCKED_{exit}).

(E_M-LOCKED_{step}) then we can conclude that :

(1) there exist a set of identifiers \bar{x} , a set of locks λ_1^l and two monitored statements S^s and $S^{s'}$, such that:

- S is equal to $\odot\bar{x}[S^s]$,
- S' is equal to $\odot\bar{x}[S^{s'}]$,
- $S^s \neq \emptyset$,
- $\langle ((\sigma_1, \lambda_1 \setminus \bar{x}), q_1) \Vdash S^s \rangle \xrightarrow{\circ} \mathbf{M} \langle ((\sigma'_1, \lambda_1^l), q'_1) \Vdash S^{s'} \rangle$,
- $\lambda_1^l = \lambda_1^l \cup \bar{x}$.

It follows directly from the definition of (E_M-LOCKED_{step}).

(2) $\zeta_1 \cong \zeta_2 \Rightarrow ((\sigma_1, \lambda_1 \setminus \bar{x}), q_1) \cong ((\sigma_2, \lambda_2 \setminus \bar{x}), q_2)$.

This is obvious from the Definition A.1.

(3) $\zeta_1 \vdash \odot\bar{x}[S^s] \Rightarrow ((\sigma_1, \lambda_1 \setminus \bar{x}), q_1) \vdash S^s$.

This is obvious from the Definition A.2.

(4) There exists $((\sigma'_2, \lambda_2^l), q'_2)$ such that $((\sigma'_1, \lambda_1^l), q'_1) \cong ((\sigma'_2, \lambda_2^l), q'_2)$ and one of the possible global conclusions holds for the monitored statement S^s .

It follows directly from the inductive hypothesis and the local conclusions (1), (2) and (3).

• The property holds.

Case 1: the first global conclusion holds for S^s .

(a) $\langle ((\sigma_2, \lambda_2 \setminus \bar{x}), q_2) \Vdash S^s \rangle \xrightarrow{\circ} \mathbf{M} \langle ((\sigma'_2, \lambda_2^l), q'_2) \Vdash S^{s'} \rangle$

It follows directly from the local conclusion (2) and the case hypothesis.

• There exists ζ'_2 , equals to $((\sigma'_2, \lambda_2^l \cup \bar{x}), q'_2)$, such that $\langle \zeta_2 \Vdash S \rangle \xrightarrow{\circ} \mathbf{M}(\mathbf{O}) \langle \zeta'_2 \Vdash S' \rangle$ and $\zeta'_1 \cong \zeta'_2$.

This is direct using the definition of the rule (E_M-BRANCH_{step}) and the local conclusions (a), (4) and (1).

Case 2: the second global conclusion holds for S^s .

(a) there exist an evaluation context E and three monitored statements S^b , P^e and P^u such that:

- S^s is equal to $E(S^b)$ and $S^{s'}$ is equal to $E(\otimes(P^e, P^u)[P^e])$,
- $w'_2 = w_2 \top$,
- $o = \epsilon$,
- $\Downarrow((\sigma_2, \lambda_2 \setminus \bar{x}), q_2) \vdash S^s \Downarrow \xrightarrow{\epsilon} \mathbf{M}(\mathbf{O}) \Downarrow((\sigma'_2, \lambda'_2), q'_2) \vdash E(\otimes(P^u, P^e)[P^u]) \Downarrow$.

It follows directly from the local conclusion (4) and the case hypothesis.

- (b) there exists E' , equal to “ $\odot \bar{x}[E]$ ”, such that S is equal to $E'(S^b)$ and S' is equal to $E'(\otimes(P^e, P^u)[P^e])$.

This result is implied by the local conclusions (1) and (a).

- (c) $\Downarrow((\sigma_2, \lambda_2), q_2) \vdash \odot \bar{x}[S^s] \Downarrow \xrightarrow{\epsilon} \mathbf{M}(\mathbf{O}) \Downarrow((\sigma'_2, \lambda'_2 \cup \bar{x}), q'_2) \vdash \odot \bar{x}[E(\otimes(P^u, P^e)[P^u])] \Downarrow$.
It follows directly from the evaluation rule (E_M -LOCKED_{step}) and the local conclusions (1) and (a).

- (d) $\Downarrow \zeta_2 \vdash S \Downarrow \xrightarrow{\epsilon} \mathbf{M}(\mathbf{O}) \Downarrow \zeta'_2 \vdash E'(\otimes(P^u, P^e)[P^u]) \Downarrow$ with $\lambda'_2 = \lambda_2 \cup \bar{x}$.
This result is implied by the local conclusions (c), (1), and (b).

- (e) $\zeta'_1 \cong \zeta'_2$.
The local conclusion (4) and the definition A.1 imply $\forall x \notin L'_1 : x \in \lambda_1^t \Leftrightarrow x \in \lambda_2^t$.
Combined with the local conclusions (1) and (d), it implies $\forall x \notin L'_1 : x \in \lambda'_1 \Leftrightarrow x \in \lambda'_2$. This property and the local conclusion (4) imply the desired result.

- (•) the second possible global conclusion holds.
It follows directly from the local conclusions (b), (a), (d) and (e).

□

Lemma A.8 (General Soundness). *For all thread pools Θ_1 , Θ_2 and Θ_1^f , concurrent execution states ϑ_1 , ϑ_2 and ϑ_1^f , and output sequences o such that:*

$$\star_1 (\vartheta_1 \vdash \Theta_1) \sim (\vartheta_2 \vdash \Theta_2),$$

$$\star_2 \langle \vartheta_1 \vdash \Theta_1 \rangle \xrightarrow{\mathcal{M}(O)} \langle \vartheta_1^f \vdash \Theta_1^f \rangle,$$

it is true that, there exists a concurrent execution state ϑ_2^f and a thread pool Θ_2^f such that:

$$\bullet \langle \vartheta_2 \vdash \Theta_2 \rangle \xrightarrow{\mathcal{M}(O)} \langle \vartheta_2^f \vdash \Theta_2^f \rangle.$$

Proof. The proof goes by induction on the length of the derivation of the global hypothesis \star_2 .

Obviously, the lemma holds if the length of the derivation is equal to 0. In that case o is equal to ϵ and the conclusion holds with $\Theta_2^f = \Theta_2$ and $\vartheta_2^f = \vartheta_2$.

Otherwise, the global hypothesis \star_2 implies that there exist a concurrent execution state ϑ_1' and a thread pool Θ_1' such that:

$$\langle \vartheta_1 \vdash \Theta_1 \rangle \xrightarrow{o^p}_{\mathcal{M}} \langle \vartheta_1' \vdash \Theta_1' \rangle \quad (3)$$

$$\langle \vartheta_1' \vdash \Theta_1' \rangle \xrightarrow{o^s}_{\mathcal{M}} \langle \vartheta_1^f \vdash \Theta_1^f \rangle \quad (4)$$

$$o = o^p o^s \quad (5)$$

The remaining of the proof demonstrates the existence of a concurrent execution state ϑ_2' and a thread pool Θ_2' such that:

$$\begin{aligned} \langle \vartheta_2 \vdash \Theta_2 \rangle &\xrightarrow{o^p}_{\mathcal{M}} \langle \vartheta_2' \vdash \Theta_2' \rangle \\ (\vartheta_1' \vdash \Theta_1') &\sim (\vartheta_2' \vdash \Theta_2') \end{aligned}$$

The desired property will then follow by induction. As $(\vartheta_1' \vdash \Theta_1') \sim (\vartheta_2' \vdash \Theta_2')$, using the inductive hypothesis on (4), we get that there exists an execution state ϑ_2^f and a thread pool Θ_2^f such that:

$$\langle \vartheta_2' \vdash \Theta_2' \rangle \xrightarrow{o^s}_{\mathcal{M}} \langle \vartheta_2^f \vdash \Theta_2^f \rangle$$

Consequently, $\langle \vartheta_2 \vdash \Theta_2 \rangle \xrightarrow{o}_{\mathcal{M}} \langle \vartheta_2^f \vdash \Theta_2^f \rangle$.

Let ϑ_1 be equal to (ς_1, Q_1) . The only possible rule for the previous one step evaluation — (3) — is (E_M-CONCUR). It implies that there exists ι belonging to the domain of Θ_1 such that:

$$\langle (\varsigma_1, \text{extractState}(Q_1, \iota)) \vdash \Theta_1(\iota) \rangle \xrightarrow{o^p}_{\mathcal{M}} \langle (\varsigma_1', q_1') \vdash S_1' \rangle \quad (6)$$

$$\vartheta_1' = (\varsigma_1', \text{updateStates}(Q_1, \iota, q_1')) \quad (7)$$

$$\Theta_1' = \Theta_1[\iota \mapsto S_1'] \quad (8)$$

Let ζ_1 be $(\varsigma_1, \text{extractState}(Q_1, \iota))$, ζ_2 be $(\varsigma_2, \text{extractState}(Q_2, \iota))$ — with $\vartheta_2 = (\varsigma_2, Q_2)$ —, S_1 be $\Theta_1(\iota)$ and S_2 be $\Theta_2(\iota)$. Because of the global hypothesis \star_1 and the definition of bi-simulation for thread pools, the following is true: $(\zeta_1 \vdash S_1) \sim (\zeta_2 \vdash S_2)$. Therefore, one of the conclusions of definition A.6 has to hold.
 def:execution-stages-bisimilarity

Case 1: The conclusion \triangleleft_1 of definition A.6 holds.

This conclusion states that “ $\zeta_1 \cong \zeta_2$ ”, “ $\zeta_1 \vdash S_1$ ” and S_1 is equal to S_2 . Hence, as from (6) the monitored statement S_1 in the monitored execution state ζ_1 evaluates to S'_1 yielding the output sequence o^p and the monitored execution state ζ'_1 , lemma A.7 applies and one of its global conclusions has to hold.

Sub-case 1.a: The global conclusion \triangleleft_1 of lemma A.7 holds.

This conclusion states that there exists a monitored execution state ζ'_2 such that:

$$\zeta'_1 \cong \zeta'_2 \quad (9)$$

$$\langle\!\langle \zeta_2 \vdash S_2 \rangle\!\rangle \xrightarrow{op}_M \langle\!\langle \zeta'_2 \vdash S'_1 \rangle\!\rangle \quad (10)$$

Let ϑ'_2 be equal to $(\varsigma'_2, \text{updateStates}(Q_2, \iota, q'_2))$ — with $\zeta'_2 = (\varsigma'_2, q'_2)$ — and Θ'_2 be equal to $\Theta_2[\iota \mapsto S'_1]$. The previous evaluation of S_2 — (10) — and the rule (E_M-CONCUR) implies:

$$\langle\!\langle \vartheta_2 \Vdash \Theta_2 \rangle\!\rangle \xrightarrow{op}_M \langle\!\langle \vartheta'_2 \Vdash \Theta'_2 \rangle\!\rangle$$

The global hypothesis \star_1 implies that $\zeta_1 \vdash S_1$ and $\zeta_2 \vdash S_2$. Hence, lemma A.3, (6) and (10) are sufficient to conclude that $\zeta'_1 \vdash S_1$ and $\zeta'_2 \vdash S_2$. Therefore, as $\zeta'_1 \cong \zeta'_2$ and $\Theta'_1(\iota) = \Theta'_2(\iota)$, the following is true:

$$(\zeta'_1 \vdash \Theta'_1(\iota)) \sim (\zeta'_2 \vdash \Theta'_2(\iota))$$

and, with $\vartheta'_1 = (\varsigma'_1, Q'_1)$ and $\vartheta'_2 = (\varsigma'_2, Q'_2)$,

$$((\varsigma'_1, \text{extractState}(Q'_1, \iota)) \vdash \Theta'_1(\iota)) \sim ((\varsigma'_2, \text{extractState}(Q'_2, \iota)) \vdash \Theta'_2(\iota))$$

Equation (9) implies:

- $V'_1 = V'_2$, $W'_1 = W'_2$ and $L'_1 = L'_2$,
- $\forall x \notin V'_1 : \sigma'_1(x) = \sigma'_2(x)$,
- $\forall l \notin L'_1 : x \in \lambda'_1 \Leftrightarrow x \in \lambda'_2$.

For every j , let w_1^j be the forth element of $\text{extractState}(Q_1, j)$. $\text{extractState}(Q_1, j)$ is equal to (V_1, W_1, L_1, w_1^j) . Let w_2^j be similarly defined. For every j — different from ι — in the domain of Θ'_1 , (8) implies $\Theta'_1(j) = \Theta_1(j)$; and similarly for $\Theta'_2(j)$. As well, (7) implies that $\text{extractState}(Q'_1, j)$ is equal to $(V'_1, W'_1, L'_1, w_1^j)$; and similarly with $\text{extractState}(Q'_2, j)$. Therefore, lemma A.4 applies and, for every j in the domain of Θ_1 and different from ι :

$$((\varsigma'_1, \text{extractState}(Q'_1, j)) \vdash \Theta'_1(j)) \sim ((\varsigma'_2, \text{extractState}(Q'_2, j)) \vdash \Theta'_2(j))$$

Consequently,

$$(\vartheta'_1 \vdash \Theta'_1) \sim (\vartheta'_2 \vdash \Theta'_2)$$

Sub-case 1.b: The global conclusion \triangleleft_2 of lemma A.7 holds.

This conclusion states that there exists a monitored execution state ζ'_2 such that $\zeta'_1 \cong \zeta'_2$ and:

1. there exist an evaluation context E , a value v belonging to the set $\{\text{true}, \text{false}\}$ and three monitored statements S^b , P^{true} and P^{false} such that S_1 — and S_2 because $S_1 = S_2$ — is equal to $E(S^b)$, S'_1 is equal to $E(\otimes(S_v, S_{-v})[S_v])$ and $w_1 = (w_1 \downarrow_E)$,

2. $w'_2 = w_2 \top$,
3. $o^p = \epsilon$,
4. $\langle \zeta_2 \vdash S_2 \rangle \xrightarrow{\epsilon}_M \langle \zeta'_2 \vdash E(\otimes(S_{-v}, S_v)[S_{-v}]) \rangle$.

Let ϑ'_2 be equal to $(\zeta'_2, \text{updateStates}(Q_2, \iota, q'_2))$ and Θ'_2 be equal to $\Theta_2[\iota \mapsto E(\otimes(S_{-v}, S_v)[S_{-v}])]$. The previous evaluation of S_2 with the monitoring execution state ζ_2 , the rule (E_M-CONCUR) and the equality between o^p and ϵ imply that $\langle \vartheta_2 \Vdash \Theta_2 \rangle \xrightarrow{o^p}_M \langle \vartheta'_2 \Vdash \Theta'_2 \rangle$.

Lemma A.3, the global hypothesis \star_1 and the evaluations of S_1 and S_2 imply that $\zeta'_1 \vdash S'_1$ and $\zeta'_2 \vdash S'_2$.

With $P_1^e = S_1^b = P_2^u = S_v$ and $P_2^e = S_2^b = P_1^u = S_{-v}$, S'_1 is equal to $E(\otimes(P_1^e, P_1^u)[S_1^b])$ and S'_2 is equal to $E(\otimes(P_2^e, P_2^u)[S_2^b])$. This is the point (a) of the second definition \triangleleft_2 of bi-similarity.

The fact that $\zeta'_1 \cong \zeta'_2$ implies that $V'_1 = V'_2$, $W'_1 = W'_2$, $L'_1 = L'_2$ and $(w'_1 \downarrow_E) = (w'_2 \downarrow_E)$. This is the point (b) of the second definition \triangleleft_2 of bi-similarity.

From the global conclusion \triangleleft_2 of lemma A.7, $w_1 = (w_1 \downarrow_E)$ and $w'_2 = w_2 \top$. The facts that $\zeta_1 \cong \zeta_2$ and $\zeta'_1 \cong \zeta'_2$ imply that $w_1 = w_2$ and $w'_1 = w'_2$. Therefore, $w'_1 = w_1 \top$ and $w'_2 = w_2 \top$. This is the point (c) of the second definition \triangleleft_2 of bi-similarity.

As $\zeta'_1 \cong \zeta'_2$, we have that $\forall x \notin V'_1 : \sigma'_1(x) = \sigma'_2(x)$ and $\forall l \notin L'_1 : l \in \lambda'_1 \Leftrightarrow l \in \lambda'_2$. Those properties are points (d) and (e) of the second definition \triangleleft_2 of bi-similarity.

As $P_1^e = P_2^u$ and $P_2^e = P_1^u$, it's trivial to show that point (f) of the second definition \triangleleft_2 of bi-similarity holds.

Therefore,

$$(\zeta'_1 \vdash \Theta'_1(\iota)) \sim (\zeta'_2 \vdash \Theta'_2(\iota))$$

and, following a proof similar to the one used in the sub-case 1.a,

$$(\vartheta'_1 \vdash \Theta'_1) \sim (\vartheta'_2 \vdash \Theta'_2)$$

Case 2: The conclusion \triangleleft_2 of definition A.6 holds.

This conclusion states that there exist an evaluation context E , four programs $P_1^e, P_1^u, P_2^e, P_2^u$ and two monitored statements S_1^b and S_2^b such that all the following is true:

1. $S_1 = E(\otimes(P_1^e, P_1^u)[S_1^b])$ and $S_2 = E(\otimes(P_2^e, P_2^u)[S_2^b])$,
2. $V_1 = V_2$, $W_1 = W_2$, $L_1 = L_2$ and $(w_1 \downarrow_E) = (w_2 \downarrow_E)$,
3. $\forall i \in \{1, 2\} : \exists w_i^s$ s.t. $w_i = (w_i \downarrow_E) \top w_i^s$,
4. $\forall x \notin V_1 : \sigma_1(x) = \sigma_2(x)$,
5. $\forall l \notin L_1 : l \in \lambda_1 \Leftrightarrow l \in \lambda_2$,
6. $\forall L : \neg \text{stops}(P_1^e) \wedge \neg \text{stops}(P_1^u) \Leftrightarrow \neg \text{stops}(P_2^e) \wedge \neg \text{stops}(P_2^u)$.

The point 3 stated above and lemma A.5 imply that o^p equals ϵ .

Sub-case 2.a: $S_1^b \neq \emptyset$.

Equation (6), the conclusion \triangleleft_2 of definition A.6 and lemma A.1 imply that

$$\langle \zeta_1 \Vdash \otimes(P_1^e, P_1^u)[S_1^b] \rangle \xrightarrow{\epsilon}_M \langle (\zeta'_1, q'_1) \Vdash E^{-1}(S'_1) \rangle$$

Because of the sub-case hypothesis, the only possible rule for this evaluation step is $(E_{M(O)}\text{-BRANCH}_{\text{step}})$. Therefore,

$$\langle \zeta_1 \Vdash S_1^b \rangle \xrightarrow{\epsilon}_M \langle (\zeta'_1, q'_1) \Vdash S_1^n \rangle \quad \text{and} \quad S'_1 = E(\otimes(P_1^e, P_1^u)[S_1^n])$$

The following steps consist in proving that $(\zeta'_1 \vdash S'_1) \sim (\zeta_2 \vdash S_2)$.

The case hypothesis, definition A.2 and lemma A.5 imply:

$$V'_1 = V_1, W'_1 = W_1 \text{ and } L'_1 = L_1 \quad (11)$$

$$\forall x \notin \text{modify}(S_1) : \sigma'_1(x) = \sigma_1(x) \quad (12)$$

$$\forall l \notin \text{needs}(S_1) \cup \text{owns}(S_1) : l \in \lambda'_1 \Leftrightarrow l \in \lambda_1 \quad (13)$$

The first point of the definition of bi-similarity is trivial as $S'_1 = E(\otimes(P_1^e, P_1^u)[S_1^n])$ and $S_2 = E(\otimes(P_2^e, P_2^u)[S_2^b])$.

The case hypothesis combined with (11) imply

$$V'_1 = V_2, W'_1 = W_2 \text{ and } L'_1 = L_2$$

Lemma A.2 imply $(w'_1 \downarrow_E) = (w_1 \downarrow_E)$. Therefore, the case 2 hypothesis “ $(w_1 \downarrow_E) = (w_2 \downarrow_E)$ ” implies $(w'_1 \downarrow_E) = (w_2 \downarrow_E)$. This is the point (b) of the bi-similarity definition.

Let E' be $E(\otimes(P_1^e, P_1^u)[\square])$. S_1 is equal to $E'(S_1^b)$. As the case hypothesis imply $w_1 = (w_1 \downarrow_E) \sqcup w_1^s$, by definition $(w_1 \downarrow_{E'}) = (w_1 \downarrow_E) \sqcup$. Lemma A.2 imply $(w_1 \downarrow_{E'}) = (w'_1 \downarrow_{E'})$. By definition of branching context of evaluation context, there exists $w_1^{s'}$ such that $w'_1 = (w'_1 \downarrow_{E'}) \sqcup w_1^{s'}$. Therefore, there exists $w_1^{s'}$ such that $w'_1 = (w'_1 \downarrow_E) \sqcup w_1^{s'}$. This is sufficient to trivially prove the point (c) of the bi-similarity definition.

Equation (11) implies that all x not in V'_1 is not in V_1 . Definition A.2 implies that all x not belonging to V_1 does not belongs to $\text{modify}(S_1)$. Therefore, (12) implies the point (d) of the bi-similarity definition.

Similarly, (13) implies the point (e) of the bi-similarity definition.

Point (f) of the bi-similarity definition follows directly from the case hypothesis.

Therefore,

$$(\zeta'_1 \vdash \Theta'_1(i)) \sim (\zeta_2 \vdash \Theta_2(i))$$

and, following a proof similar to the one used in the sub-case 1.a,

$$(\vartheta'_1 \vdash \Theta'_1) \sim (\vartheta'_2 \vdash \Theta'_2)$$

Sub-case 2.b: $S_1^b = \emptyset$.

Equation (6), the conclusion \triangleleft_2 of definition A.6 and lemma A.1 imply that

$$\langle \zeta_1 \Vdash \otimes(P_1^e, P_1^u)[S_1^b] \rangle \xrightarrow{\epsilon}_M \langle (\zeta'_1, q'_1) \Vdash E^{-1}(S'_1) \rangle$$

Because of the sub-case hypothesis, the only possible rule for this evaluation step is (E_{M(O)}-BRANCH_{exit}). Therefore, with $\tilde{v} = \text{modify}(P_1^e) \cup \text{modify}(P_1^u)$ and $\tilde{l} = \text{needs}(P_1^e) \cup \text{needs}(P_1^u)$,

$$S'_1 = E(\emptyset) \quad \text{and} \quad \zeta'_1 = \zeta_1 \quad (14)$$

$$\neg \text{stops}(P_1^e) \quad \text{and} \quad \neg \text{stops}(P_1^u) \quad (15)$$

$$V'_1 = V_1, \quad W'_1 = W_1 \setminus \tilde{v}, \quad L'_1 = L_1 \setminus \tilde{l} \quad \text{and} \quad w'_1 = (w_1 \downarrow_E) \quad (16)$$

Equation (15), the case hypothesis and definition A.2 imply $\neg \text{stops}(S_2^b)$. The case hypothesis and definition A.2 imply $\forall l \in (\text{needs}(S_2^b) \setminus \text{owns}(S_2^b)) : l \notin \lambda_2$. Therefore, lemma A.6 implies that there exists a monitored execution state ζ_2^t such that:

$$\langle \zeta_2 \vdash S_2 \rangle \xrightarrow{\epsilon}_M^* \langle \zeta_2^t \vdash E(\otimes(P_2^e, P_2^u)[\emptyset]) \rangle$$

Let E' be $E(\otimes(P_2^e, P_2^u)[\square])$. The previous derivation, the case hypothesis and lemma A.5 imply

$$V_2^t = V_2, \quad W_2^t = W_2, \quad L_2^t = L_2 \quad \text{and} \quad (w_2^t \downarrow_{E'}) = (w_2 \downarrow_{E'}),$$

$$\forall x \notin \text{modify}(S_2) : \sigma_2^t(x) = \sigma_2(x),$$

$$\forall l \notin \text{needs}(S_2) \cup \text{owns}(S_2) : l \in \lambda_2^t \Leftrightarrow l \in \lambda_2.$$

Therefore, the case hypothesis implies:

$$V_1^t = V_2^t, \quad W_1^t = W_2^t, \quad L_1^t = L_2^t \quad \text{and} \quad (w_1^t \downarrow_E) = (w_2^t \downarrow_E)$$

$$\forall i \in \{1, 2\} : \exists w_i^t \text{ s.t. } w_i^t = (w_i^t \downarrow_E) \top w_i^t$$

$$\forall x \notin V_1^t : \sigma_1^t(x) = \sigma_2^t(x)$$

$$\forall l \notin L_1^t : l \in \lambda_1^t \Leftrightarrow l \in \lambda_2^t$$

Then, by applying the rule (E_M-BRANCH_{exit}) to $E(\otimes(P_2^e, P_2^u)[\emptyset])$, we get that there exists ζ'_2 such that

$$\langle \zeta_2 \vdash S_2 \rangle \xrightarrow{\epsilon}_M^* \langle \zeta'_2 \vdash S'_1 \rangle$$

and

$$V'_1 = V'_2, \quad W'_1 = W'_2, \quad L'_1 = L'_2 \quad \text{and} \quad w'_1 = w'_2$$

$$\forall x \notin V'_1 : \sigma'_1(x) = \sigma'_2(x)$$

$$\forall l \notin L'_1 : l \in \lambda'_1 \Leftrightarrow l \in \lambda'_2$$

Therefore $S'_1 = S'_2$ and $\zeta'_1 \cong \zeta'_2$.

Following a proof similar to the one used in the sub-case 1.a, we can conclude that:

$$(\vartheta'_1 \vdash \Theta'_1) \sim (\vartheta'_2 \vdash \Theta'_2)$$

□

A.2 Partial Transparency

A.2.1 Definitions

$\frac{}{\gamma \vdash_E \mathbf{skip} : \tau \text{ cmd}}$	(T _E -SKIP)
$\frac{\gamma(e) \leq \gamma(x) \quad \tau \leq \gamma(x)}{\gamma \vdash_E x := e : \tau \text{ cmd}}$	(T _E -ASSIGN)
$\frac{\gamma(e) \leq L}{\gamma \vdash_E \mathbf{output} e : L \text{ cmd}}$	(T _E -OUTPUT)
$\frac{\gamma \vdash_E S^h : \tau \text{ cmd} \quad \gamma \vdash_E S^t : \tau \text{ cmd}}{\gamma \vdash_E S^h ; S^t : \tau \text{ cmd}}$	(T _E -SEQ)
$\frac{\gamma(e) \leq \tau' \quad \gamma \vdash_E S^{\text{true}} : \tau' \text{ cmd} \quad \gamma \vdash_E S^{\text{false}} : \tau' \text{ cmd} \quad \tau \leq \tau'}{\gamma \vdash_E \mathbf{if} e \mathbf{then} S^{\text{true}} \mathbf{else} S^{\text{false}} \mathbf{end} : \tau \text{ cmd}}$	(T _E -IF)
$\frac{\gamma(e) \leq L \quad \gamma \vdash_E S^1 : L \text{ cmd}}{\gamma \vdash_E \mathbf{while} e \mathbf{do} S^1 \mathbf{done} : L \text{ cmd}}$	(T _E -WHILE)
$\frac{\gamma(e) \leq L \quad \gamma \vdash_E S^s : L \text{ cmd}}{\gamma \vdash_E \mathbf{with} \bar{x} \mathbf{when} e \mathbf{do} S^s \mathbf{done} : L \text{ cmd}}$	(T _E -SYNC)
<hr/>	
$\frac{}{\gamma \vdash_E \emptyset : \tau \text{ cmd}}$	(T _E -NOC)
$\frac{\gamma \vdash_E S : L \text{ cmd}}{\gamma \vdash_E \odot \bar{x}[S] : L \text{ cmd}}$	(T _E -LOCKED)
$\frac{\gamma \vdash_E P^e : \tau' \text{ cmd} \quad \gamma \vdash_E P^u : \tau' \text{ cmd} \quad \gamma \vdash_E S^b : \tau' \text{ cmd} \quad \tau \leq \tau'}{\gamma \vdash_E \otimes (P^e, P^u)[S^b] : \tau \text{ cmd}}$	(T _E -BRANCHED)

Figure 11: The extended type system used for monitored statements.

An extended type system for monitored statements is given in Figure 11.

strip() takes into parameter a monitored statements and returns an execution statement.

$$\begin{aligned}
\text{strip}(\emptyset) &= \emptyset \\
\text{strip}(x := e) &= x := e \\
\text{strip}(\mathbf{output } e) &= \mathbf{output } e \\
\text{strip}(\mathbf{skip}) &= \mathbf{skip} \\
\text{strip}(\mathbf{if } e \mathbf{ then } S_1 \mathbf{ else } S_2 \mathbf{ end}) &= \mathbf{if } e \mathbf{ then } \text{strip}(S_1) \mathbf{ else } \text{strip}(S_2) \mathbf{ end} \\
\text{strip}(\mathbf{while } e \mathbf{ do } S \mathbf{ done}) &= \mathbf{while } e \mathbf{ do } \text{strip}(S) \mathbf{ done} \\
\text{strip}(\mathbf{with } \bar{x} \mathbf{ when } e \mathbf{ do } S \mathbf{ done}) &= \mathbf{with } \bar{x} \mathbf{ when } e \mathbf{ do } \text{strip}(S) \mathbf{ done} \\
\text{strip}(S_1 ; S_2) &= \text{strip}(S_1) ; \text{strip}(S_2) \\
\text{strip}(\odot \bar{x}[S]) &= \odot \bar{x}[\text{strip}(S)] \\
\text{strip}(\otimes(P_1, P_2)[S]) &= \text{strip}(S)
\end{aligned}$$

A.2.2 Lemmas

Lemma A.9. For all monitored statements S and typing environments γ , if “ $\gamma \vdash_E S : H \text{ cmd}$ ” then $\forall x \in \text{modify}(S). \gamma(x) = H, \text{needs}(S) = \emptyset$ and $\neg \text{stops}(S)$.

Proof. The proof goes by induction on the derivation tree of the typing judgment and follows directly from the definitions of the different functions. \square

Lemma A.10. For all monitored statements S , execution statements S'_u , execution states (σ, λ) and (σ', λ') , automaton states (V, W, L, w) , typing environments γ , and types τ if:

- ★₁ $((\sigma, \lambda), (V, W, L, w)) \vdash S$,
- ★₂ $\forall x \in V, \gamma(x) = H$,
- ★₃ $L = \emptyset$,
- ★₄ $w \notin \{\perp\}^* \Rightarrow \tau = H$,
- ★₅ $\gamma \vdash_E S : \tau \text{ cmd}$,
- ★₆ $\langle (\sigma, \lambda) \vdash \text{strip}(S) \rangle \xrightarrow{o}_s \langle (\sigma', \lambda') \vdash S'_u \rangle$,

then there exists an automaton state (V', W', L', w') , a monitored statement S'_m , and a type τ' such that:

- $\langle ((\sigma, \lambda), (V, W, L, w)) \Vdash S \rangle \xrightarrow{o}_m^* \langle ((\sigma', \lambda'), (V', W', L', w')) \Vdash S'_m \rangle$,
- $\forall x \in V', \gamma(x) = H$,
- $L' = \emptyset$,
- $w' \notin \{\perp\}^* \Rightarrow \tau' = H$,

- $\gamma \vdash_E S'_m : \tau' \text{ cmd}$,
- $S'_u = \text{strip}(S'_m)$.

Proof. The proof goes by induction on the derivation tree of the global hypothesis \star_5 . If the last typing rule used is:

(**T_E-NOC**) then we can conclude that the lemma is vacuously true as the global hypothesis \star_6 does not hold.

(**T_E-SKIP**) then we can conclude that the lemma is true as the monitored or unmonitored execution of **skip** only replace it by \emptyset .

(**T_E-ASSIGN**) then we can conclude that :

(1) S is “ $x := e$ ”.

It follows directly from the typing rule.

(2) there exists an automaton state (V', W', L', w') such that $W' = W, L' = L, w' = w$, and:

$$\langle\langle (\sigma, \lambda), (V, W, L, w) \rangle\rangle \Vdash S \Downarrow \xrightarrow{\epsilon}_M^* \langle\langle (\sigma', \lambda'), (V', W', L', w') \rangle\rangle \Vdash \emptyset \Downarrow$$

It follows directly from the rule (**E_M-OK**) as the security automaton always validate the execution of assignments.

(3) $\forall y \in V', \gamma(y) = H$.

Case 1: $x \notin W$ and $FV(e) \cap V = \emptyset$.

(a) $V' \subseteq V$.

It follows directly from the case hypothesis and the rule (**T-ASSIGN-pub**).

(o) $\forall y \in V', \gamma(y) = H$.

It follows from the local conclusion (a) and the global hypothesis \star_2 .

Case 2: $x \in W$.

(a) $x \in V$.

It follows directly from the case hypothesis and the global hypothesis \star_1 .

(o) $\forall y \in V', \gamma(y) = H$.

It follows from the local conclusion (a) and the global hypothesis \star_2 .

Case 3: $FV(e) \cap V \neq \emptyset$.

(a) $V' = V \cup \{x\}$.

It follows directly from the case hypothesis and the rule (**T-ASSIGN-sec**).

(b) $\gamma(x) = H$.

Because of the case hypothesis, there exists a variable z in $FV(e)$ such that $z \in V$.

The global hypothesis \star_2 implies that $\gamma(z) = H$. Then, the global hypothesis \star_5 and the typing rule (**T_E-ASSIGN**) imply that $\gamma(x) = H$.

(o) $\forall y \in V', \gamma(y) = H$.

It follows from the global hypothesis \star_2 and the local conclusions (a) and (b).

(•) the lemma holds.

It follows directly from the local conclusions (2) and (3), the global hypotheses \star_3 and \star_4 , and the typing rule (**T_E-NOC**).

(T_E-OUTPUT) then we can conclude that :

- (1) S is “**output** e ”, $\gamma(e) = L$, and $\tau = L$ It follows directly from the rule (T_E-OUTPUT).
- (2) $w \in \{\perp\}^*$ and $FV(e) \cap V = \emptyset$.
The global hypothesis \star_4 and the local conclusion (1) imply $w \in \{\perp\}^*$. The global hypothesis $\star_{??}$ and the local conclusion (1) imply $FV(e) \cap V = \emptyset$.
- (3) the following is true:

$$\langle (\sigma, \lambda), (V, W, L, w) \rangle \Vdash S \rangle \xrightarrow{\circ}_M^* \langle (\sigma', \lambda'), (V, W, L, w) \rangle \Vdash \emptyset \rangle$$

It follows directly from the rule (E_M-OK) and the local conclusion (2).

- the lemma holds.
It follows directly from the local conclusions (3), the global hypotheses $\star_{??}$, \star_3 and \star_4 , and the typing rule (T_E-NOC).

(T_E-SEQ) then we can conclude that :

- (1) S is “ $S^h ; S^t$ ” and:
 - $\gamma \vdash S^h : \tau \text{ cmd}$,
 - $\gamma \vdash S^t : \tau \text{ cmd}$.
 It follows directly from the rule (T_E-SEQ).
- the lemma holds.
From the inductive hypothesis, the lemma holds for S^h . Therefore, from the rules for monitored and unmonitored executions, both executions evaluate the same way.

(T_E-IF) then we can conclude that :

- (1) S is “**if** e **then** S^{true} **else** S^{false} **end**” and there exists a type τ' such that $\gamma(e) \leq \tau'$ and:
 - $\gamma \vdash S^{\text{true}} : \tau' \text{ cmd}$,
 - $\gamma \vdash S^{\text{false}} : \tau' \text{ cmd}$.
 It follows directly from the rule (T_E-IF).
- (2) $FV(e) \cap V = \emptyset$ or $\text{needs}(S^{\text{true}}) \cup \text{needs}(S^{\text{false}}) = \emptyset$.
If $FV(e) \cap V \neq \emptyset$ then there exists a variable z in $FV(e)$ such that $z \in V$. The global hypothesis \star_2 implies that $\gamma(z) = H$. Then $\gamma(e) = H$. Therefore, lemma A.9 and the local conclusion (1) imply $\text{needs}(S^{\text{true}}) = \text{needs}(S^{\text{false}}) = \emptyset$
- (3) $FV(e) \cap V \neq \emptyset$ implies that both S^{true} and S^{false} can be typed as $H \text{ cmd}$ with the typing environment γ .
If $FV(e) \cap V \neq \emptyset$ then there exists a variable z in $FV(e)$ such that $z \in V$. The global hypothesis \star_2 implies that $\gamma(z) = H$. Then $\gamma(e) = H$ and the desired result follows from the local conclusion (1).
- the lemma holds.
From the local conclusion (2), one of the two transitions (T-BRANCH-low) or (T-BRANCH-high) applies. Therefore, the monitored execution evaluates similarly to the unmonitored execution. Additionally, lemma A.9, the global hypotheses and the local conclusion (3) imply the remaining global conclusions.

(T_E-WHILE) then we can conclude that :

- (1) S is “**while** e **do** S^1 **done**”, $\gamma(e) = L$, $\tau = L$ and $\gamma \vdash S^1 : L$ cmd. It follows directly from the rule (T_E-WHILE).
- (2) $FV(e) \cap V = \emptyset$.
If $FV(e) \cap V \neq \emptyset$ then there exists a variable z in $FV(e)$ such that $z \in V$. The global hypothesis \star_2 implies that $\gamma(z) = H$. Then $\gamma(e) = H$, which is in contradiction with the local conclusion (1).
- (•) the lemma holds.
From the local conclusion (2), the transition (T-BRANCH-low) applies. Therefore, the monitored execution evaluates similarly to the unmonitored execution. Additionally, the global hypotheses and local conclusion (1) imply the remaining global conclusions.

(T_E-SYNC) then we can conclude that :

- (1) S is “**with** \bar{x} **when** e **do** S^s **done**”, $\gamma(e) = L$, $\tau = L$ and $\gamma \vdash S^s : L$ cmd. It follows directly from the rule (T_E-SYNC).
- (2) $FV(e) \cap V = \emptyset$.
If $FV(e) \cap V \neq \emptyset$ then there exists a variable z in $FV(e)$ such that $z \in V$. The global hypothesis \star_2 implies that $\gamma(z) = H$. Then $\gamma(e) = H$, which is in contradiction with the local conclusion (1).
- (•) the lemma holds.
From the local conclusion (2), the transition (T-SYNC) applies. Therefore, the monitored execution evaluates similarly to the unmonitored execution. Additionally, the global hypotheses and local conclusion (1) imply the remaining global conclusions.

(T_E-LOCKED) then we can conclude that :

- (1) S is “ $\odot \bar{x}[S^s]$ ”, $\tau = L$ and $\gamma \vdash S^s : L$ cmd. It follows directly from the rule (T_E-SYNC).
- (•) the lemma holds.
From the rules of both semantics for locked statements, the monitored execution evaluates similarly to the unmonitored execution. Additionally, the global hypotheses and local conclusion (1) imply the remaining global conclusions.

(T_E-BRANCHED) then we can conclude that :

- (1) S is “ $\otimes(P^e, P^u)[S^b]$ ” and there exist a type τ' such that:
 - $\tau \leq \tau'$
 - $\gamma \vdash S^b : \tau'$ cmd.
 It follows directly from the rule (T_E-SYNC).
- (2) $\text{strip}(S) = \text{strip}(S^b)$.
It follows directly from the definition of $\text{strip}()$.
- (3) $S^b \neq \emptyset$.
Otherwise, the global hypothesis \star_6 does not hold.

(•) the lemma holds.

From the local conclusions (1) and (2), the lemma holds for S^b , and from the local conclusion (3), rule (E_M-BRANCH_{step}) applies to S . Therefore, the lemma holds. □

Lemma A.11. *For all monitored statements S and S' , execution states (σ, λ) and (σ', λ') , automaton states (V, W, L, w) and (V', W', L', w') , typing environments γ , and types τ if:*

★₁ $((\sigma, \lambda), (V, W, L, w)) \vdash S$,

★₂ $\forall x \in V, \gamma(x) = H$,

★₃ $L = \emptyset$,

★₄ $w \notin \{\perp\}^* \Rightarrow \tau = H$,

★₅ $\gamma \vdash_E S : \tau \text{ cmd}$,

★₆ $\langle (\sigma, \lambda), (V, W, L, w) \rangle \Vdash S \Downarrow \xrightarrow{o}_M \langle (\sigma', \lambda'), (V', W', L', w') \rangle \Vdash S' \Downarrow$,

then there exists a type τ' such that:

- $\langle (\sigma, \lambda) \vdash \text{strip}(S) \Downarrow \xrightarrow{o}_s^* \langle (\sigma', \lambda') \vdash \text{strip}(S') \Downarrow$,

- $\forall x \in V', \gamma(x) = H$,

- $L' = \emptyset$,

- $w' \notin \{\perp\}^* \Rightarrow \tau' = H$,

- $\gamma \vdash_E S' : \tau' \text{ cmd}$.

Proof. The proof is similar to lemma A.10; however more simple. □

References

- [BH02] Per Brinch Hansen, editor. *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*. Springer-Verlag, 2002.
- [Bro04] Stephen D. Brookes. A semantics for concurrent separation logic. In Gardner and Yoshida [GY04], pages 16–34.
- [GM82] Joseph A. Goguen and José Meseguer. Security policies and security models. In *Proc. IEEE Symp. Security and Privacy*, pages 11–20. IEEE Computer Society Press, April 1982.
- [GY04] Philippa Gardner and Nobuko Yoshida, editors. *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*, volume 3170 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2004.
- [Hoa72] Charles Antony Richard Hoare. Towards a theory of parallel programming. In Charles Antony Richard Hoare and Ronald H. Perrott, editors, *Operating Systems Techniques*, pages 61–71. Academic Press, 1972. Reprinted in [BH02].
- [Kah87] Gilles Kahn. Natural Semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39, Passau, Germany, February 1987. Springer-Verlag.
- [LGBJS06] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David Schmidt. Automaton-based confidentiality monitoring. In *Proceedings of the 11th Annual Asian Computing Science Conference*, December 2006. To appear.
- [NRNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
- [O’H04] Peter W. O’Hearn. Resources, Concurrency and Local Reasoning. In Gardner and Yoshida [GY04], pages 49–67.
- [RHNS06] Alejandro Russo, John Hughes, David Naumann, and Andrei Sabelfeld. Closing internal timing channels by transformation. In *Proceedings of the 11th Annual Asian Computing Science Conference*, LNCS, December 6-8 2006.
- [Sab01] A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2244 of *Lecture Notes in Computer Science*, pages 227–241. Springer-Verlag, July 2001.
- [SV98] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 355–364, January 1998.