RAIRO-Theor. Inf. Appl. 43 (2009) 249–268
DOI: 10.1051/ita:2008026

Available online at:
www.rairo-ita.org

THREAD ALGEBRA FOR NONINTERFERENCE

Thuy Duong Vu[1]

Abstract. Thread algebra is a semantics for recent object-oriented programming languages [J.A. Bergstra and M.E. Loots, *J. Logic Algebr. Program.* **51** (2002) 125–156; J.A. Bergstra and C.A. Middelburg, *Formal Aspects Comput.* (2007)] such as C# and Java. This paper shows that thread algebra provides a process-algebraic framework for reasoning about and classifying various standard notions of noninterference, an important property in secure information flow. We will take the noninterference property given by Volpano *et al.* [D. Volpano, G. Smith and C. Irvine, *J. Comput. Secur.* **4** (1996) 167–187] on type systems as an example of our approach. We define a comparable notion of noninterference in the setting of thread algebra. Our approach gives a similar result to the approach of [G. Smith and D. Volpano, in *POPL'98* **29** (1998) 355–364] and can be applied to unstructured and multithreaded programming languages.

Mathematics Subject Classification. 68Q60.

1. Introduction

Thread algebra (TA) is an algebraic framework for the description and analysis of multithreaded programming languages proposed recently by Bergstra and Middelburg [6,7]. It is designed on top of *basic thread algebra* (BTA) [7], a theory about sequential program behaviors, also introduced as *basic polarized process algebra* (BPPA) in [5]. TA is a collection of *strategic interleaving operators* that turns a sequence of threads into a single thread in BTA capturing essential aspects of multithreading. It has been outlined in [5,6] how and why thread algebra is a natural candidate for recent object-oriented and multithreaded program semantics such as C# and Java.

Keywords and phrases. Noninterference, thread algebra, formal methods, security verification.

[1] Sectie Software Engineering, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands; tdvu@science.uva.nl

Article published by EDP Sciences

© EDP Sciences 2008

One may argue that thread algebra with strategic interleaving is technically less elegant in dealing with parallelism than process algebras such as CCS [18] and ACP [4] with arbitrary interleaving. However, thread algebra is designed specifically for multithreaded programs whose executions are on virtual machines that make use of scheduling. Additionally, process algebras introduce nondeterminism which might be a disadvantage for a programmer's intuition. On the other hand, in thread algebra, the programmer can always expect what might happen by considering a significant collection of different interleaving strategies. TA is a promising approach for the study of computer viruses and virtual machines [8,9].

This paper shows that thread algebra provides a process-algebraic framework for reasoning about and classifying various standard notions of noninterference [15] (see [22] for an overview), an important property in language-based security [2,11]. It characterizes systems whose execution does not reveal secret information. Formalizing and analyzing this property becomes increasingly important because of the privacy question raised in real-life applications such as mail and banking transactions. Various definitions of noninterference have been introduced. However, as stated in [22], "existing theoretical frameworks for expressing these security properties are inadequate, and practical techniques for enforcing these properties are unsatisfactory".

Most approaches on noninterference in language-based security are based on type systems [17,23,24]. The advantage of these approaches is that the type checker only needs to work on program texts, so these approaches are decidable and easy to implement. However, the programs must satisfy some structure. We take the standard notion of *termination-insensitive noninterference* (TINI) defined by Volpano *et al.* [24] as an example of our approach. We will present an alternative definition of TINI in the setting of thread algebra. We prove soundness for this definition, meaning that if a thread satisfies one of these properties then it satisfies the noninterference property proposed by Goguen and Meseguer [15]. We show that our approach accepts all secure programs that are typable by the type system in [24], and can be applied to unstructured and multithreaded programming languages. Furthermore, we can also use existing tools such as in [10,12,13] for checking process-equivalence to develop our security checkers. In the paper, we will also propose a particular interleaving strategy for thread algebra called the *cyclic strategic interleaving with persistence* operator. This strategy invokes the rotation of the thread vector only in the case that the current action is not a high action. Hence, it maintains the order of the high actions, and therefore, the analysis can be made compositional.

The structure of the paper is as follows. Section 2 recalls the basic concepts of basic thread algebra and thread algebra. We also introduce the notion of noninterference given by Goguen and Meseguer [15] and the type system of [24]. Section 3 characterizes actions for threads. Section 4 provides bisimulation up to a set of high actions, in order to define the noninterference property given in Section 5. Section 6 proposes the cyclic interleaving with persistence operator, and shows

that TINI satisfies compositionality with respect to the cyclic interleaving with persistence operator. The paper is ended with some concluding remarks in Section 7.

2. Preliminaries

This section recalls from [5–7] the basic concepts of *basic thread algebra* (BTA) and *thread algebra* (TA). We note that basic thread algebra was introduced as *basic polarized process algebra* (BPPA) in [5]. Furthermore, we introduce a simple programming language `Lang` which is used to illustrate our approach. We also describe the earliest notion of noninterference given by Goguen and Meseguer [15] and the type systems of [23,24] for checking this property.

2.1. Basic thread algebra (BTA)

2.1.1. *Primitives of Basic thread algebra.*

Let Σ be a set of *actions*. Each action is supposed to return a boolean value after its execution. BTA is constructed over Σ by the following operators:

Successful termination: $S \in \mathrm{BTA}$ yields successful terminating behavior.

Unsuccessful termination or **deadlock**: $D \in \mathrm{BTA}$ represents inaction behavior.

Postconditional composition: $(-) \trianglelefteq a \trianglerighteq (-)$ with $a \in \Sigma$. The thread $P \trianglelefteq a \trianglerighteq Q$ first performs a and then proceeds with P if true was returned and with Q otherwise. In case $P = Q$ we abbreviate this thread by the **action prefix** $a \circ (-)$: $a \circ P = P \trianglelefteq a \trianglerighteq P$.

Let BTA_Σ denote the set of **finite** threads which are made from S and D by means of a finite number of applications of postconditional composition.

2.1.2. *Infinite threads*

Threads can be infinite. To define an **infinite** thread in BTA, we use a sequence of its finite approximations.

Definition 2.1. For every $n \in \mathbb{N}$, the **approximation operator** $\pi_n : \mathrm{BTA}_\Sigma \to \mathrm{BTA}_\Sigma$ is defined inductively by

$$\begin{aligned} \pi_0(P) &= D, \\ \pi_{n+1}(S) &= S, \\ \pi_{n+1}(D) &= D, \\ \pi_{n+1}(P \trianglelefteq a \trianglerighteq Q) &= \pi_n(P) \trianglelefteq a \trianglerighteq \pi_n(Q). \end{aligned}$$

A **projective sequence** is a sequence $(P_n)_{n\in\mathbb{N}}$ such that for each $n \in \mathbb{N}$, $\pi_n(P_{n+1}) = P_n$.

We note that for all (finite or infinite) threads P and Q, $P = Q \Leftrightarrow \forall n \in \mathbb{N} : \pi_n(P) = \pi_n(Q)$. If $P = a \circ a \circ \cdots$ (P can do subsequently infinitely many

actions a), then we write $P = a^\infty$. Let $\mathrm{BTA}_\Sigma^\infty$ be the set of all threads represented by projective sequences in BTA_Σ. Then $\mathrm{BTA}_\Sigma \subseteq \mathrm{BTA}_\Sigma^\infty$.

2.1.3. *Regular threads*

Regular threads in BTA are defined as follows.

Definition 2.2. A thread P is **regular** over Σ if $P = E_1$, where E_1 is defined by a finite system of the form ($n \geq 1$):

$$\{E_i = t_i | 1 \leq i \leq n, t_i = S \text{ or } t_i = D \text{ or } t_i = E_{il} \trianglelefteq a_i \trianglerighteq E_{ir}\}$$

with $E_{il}, E_{ir} \in \{E_1, \ldots, E_n\}$ and $a_i \in \Sigma$.

These regular threads are well-defined in $\mathrm{BTA}_\Sigma^\infty$ (see [25]) and are used to represent program behaviors.

2.2. Thread algebra

Thread algebra (TA) is designed on top of BTA and is meant to specify a collection of *strategic interleaving operators*, capturing some essential aspects of multithreading. We assume that a collection of threads to be interleaved takes the form of a sequence, called a *thread vector*. Strategic interleaving operators turn a thread vector of arbitrary length into a single thread in BTA. This single thread obtained via a strategic interleaving operator is called a *multithread*. We recall from [6] the basic interleaving strategy of thread algebra called *cyclic rotation*. This interleaving strategy works in a round-robin fashion which invokes the rotation of a thread vector at every step. Furthermore, if one thread in the thread vector deadlocks, the whole does not deadlock till all others have terminated or deadlocked.

Let $\langle\rangle$ denote the empty sequence, $\langle x\rangle$ stand for a sequence of length one, and $\alpha \curvearrowright \beta$ the concatenation of two sequences. We assume that the following identity holds: $\alpha \curvearrowright \langle\rangle = \langle\rangle \curvearrowright \alpha = \alpha$.

Definition 2.3. The axioms for the **cyclic strategic interleaving** $\|_{\mathrm{csi}}$ operator are given as follows:

$$\begin{array}{lcl}
\|_{\mathrm{csi}}\, (\langle\rangle) & = & S \\
\|_{\mathrm{csi}}\, (\langle S\rangle \curvearrowright \alpha) & = & \|_{\mathrm{csi}}\, (\alpha) \\
\|_{\mathrm{csi}}\, (\langle D\rangle \curvearrowright \alpha) & = & S_D(\|_{\mathrm{csi}}\, (\alpha)) \\
\|_{\mathrm{csi}}\, (\langle x \trianglelefteq a \trianglerighteq y\rangle \curvearrowright \alpha) & = & \|_{\mathrm{csi}}\, (\alpha \curvearrowright \langle x\rangle) \trianglelefteq a \trianglerighteq \|_{\mathrm{csi}}\, (\alpha \curvearrowright \langle y\rangle)
\end{array}$$

where the auxiliary deadlock at termination operator S_D turns termination into deadlock and is defined by

$$\begin{array}{lcl}
S_D(S) & = & D \\
S_D(D) & = & D \\
S_D(x \trianglelefteq a \trianglerighteq y) & = & S_D(x) \trianglelefteq a \trianglerighteq S_D(y).
\end{array}$$

For a thread vector α of arbitrary (finite or infinite) threads $\alpha = \alpha_1 \curvearrowright \cdots \curvearrowright \alpha_n$, $\|_{\mathrm{csi}}\,(\alpha)$ is determined by its projective sequence (*e.g.* see [25,26]):

$$\pi_n(\|_{\mathrm{csi}}\,(\alpha)) = \pi_n(\|_{\mathrm{csi}}\,(\pi_n(\alpha_1) \curvearrowright \cdots \curvearrowright \pi_n(\alpha_n))).$$

Example 2.4. Let $P = a^\infty$ and $Q = b^\infty$ be two single threads. Then $\|_{\mathrm{csi}}\,(\langle P \rangle \curvearrowright \langle Q \rangle) = a \circ b \circ a \circ b \cdots$

2.3. The programming language Lang

It has been outlined in [3,5,6] that program behaviors of sequential and multithreaded programming languages can be represented as threads in BTA. To illustrate our approach, we consider threads as program behaviors of programs written in a simple imperative programming language `Lang` which, similar to that of [23,24], is defined as follows.

$$\begin{array}{lll} X, Y, \ldots & ::= & x := e \mid X; Y \mid \\ & & \texttt{IF}\ e\ \texttt{THEN}\ X\ \texttt{ELSE}\ Y\ \texttt{END IF} \mid \\ & & \texttt{WHILE}\ e\ \texttt{DO}\ X\ \texttt{END WHILE} \end{array}$$

where e stands for a boolean or an arithmetic expression, whose syntax we do not describe here.

We now consider assignments $x{:=}e$ and tests e of `Lang` as the actions in Σ, written as $[x{:=}e]$ and $\langle e \rangle$. Then program behaviors of `Lang` are regular threads, as illustrated in the following example.

Example 2.5. Let X and Y be given as follows.

```
X::= IF h==1 THEN l:=1 ELSE l:=0 END IF.
Y::= l:=0;
     WHILE h==0 DO
           h:=0;
     END WHILE;
     l:=1.
```

The behaviors of X and Y, denoted by $|X|$ and $|Y|$, are determined by: $|X| = ([l{:=}1] \circ S) \trianglelefteq \langle h{=}{=}1 \rangle \trianglerighteq ([l{:=}0] \circ S)$ and $|Y| = [l{:=}0] \circ P$, where $P = ([h{:=}0] \circ P) \trianglelefteq \langle h{=}{=}0 \rangle \trianglerighteq ([l{:=}1] \circ S)$.

2.4. Input-output transformations

In this section, we present the notion of *input-output transformations* of program behaviors. This notion is based on the *effect* and *yield* of an action on a state space. We assume the existence of a set `Var` of program variables, and a state space $\mathbb{S}$ whose elements play the role of inputs as well as of outputs of programs (or threads) in the programming language `Lang`. All results of the paper are related to these two sets.

2.4.1. *The effect and yield of an action on the state space* $\mathbb{S}$

Suppose that upon the execution of a program, the values of a program variable x are ranging over the set $\mathtt{values}(x)$. A **state of a variable** is a pair of the form $\langle x, v\rangle$ with $v \in \mathtt{values}(x)$. A **state of the state space** $\mathbb{S}$ is a set s consisting of states of all variables in $\mathtt{Var}$, in which there is a one-to-one correspondence between variables and their states. If s is a state of the state space and $\langle x, v\rangle \in s$, we write $s.x.\mathtt{value} = v$.

For an action $a \in \Sigma$, there is an operation $\mathtt{effect}_a : \mathbb{S} \to \mathbb{S}$ that changes the *state* due to the execution of a called the **effect** operation, and an operation $y_a : \mathbb{S} \to \{\mathtt{true}, \mathtt{false}\}$ that determines a boolean value when a is performed in a state of $\mathbb{S}$, called the **yield** operation.

We assume that if an action has some effect on the state space then its yield always determines true. Formally, for an action $a \in \Sigma$, if there exists a state $s \in \mathbb{S}$ such that $\mathtt{effect}_a(s) \neq s$ then $y_a(s') = \mathtt{true}$ for all states $s' \in \mathbb{S}$.

In the following, we will specifically define the effect and yield of an assignment and a test on the state space $\mathbb{S}$. An assignment $[x{:=}e]$ may have effect on the state space and always returns $\mathtt{true}$ after its execution. Formally, for all states $s \in \mathbb{S}$,

$$\begin{array}{lcl} \mathtt{effect}_{[x:=e]}(s) & = & s \setminus \{\langle x, s.x.\mathtt{value}\rangle\} \cup \{\langle x, \sigma(e)\rangle\}, \\ y_{[x:=e]}(s) & = & \mathtt{true} \end{array}$$

where $\sigma(e)$ is obtained by first replacing occurrences of variables y in the expression e by $s.y.\mathtt{value}$, and then calculating its value in the set $\mathtt{values}(x)$.

A test $\langle e\rangle$ has no effect on the state space, and can produce a negative reply. Formally, for all states $s \in \mathbb{S}$, $\mathtt{effect}_{\langle e\rangle}(s) = s$ and $y_{\langle e\rangle} = \sigma(e)$, where $\sigma(e)$ is obtained by first replacing occurrences of variables y in e by $s.y.\mathtt{value}$, and then computing the result in the set $\{\mathtt{true}, \mathtt{false}\}$.

2.4.2. *Program behavior and input-output relations*

Input-output transformations are derived from program behaviors rather than from programs themselves. An action of a program behavior must be viewed as a transformation of the states in a state space, producing a boolean whenever applied. This action is taken as an **input value** of a behavior. The state reached after the final action has been performed represents the **output value** of a computation. Let D represent a failure value that can't be computed. The notion of input-output transformations can be captured formally as follows.

Definition 2.6. Given a finite thread P, a function $P \bullet (-) : \mathbb{S} \to \mathbb{S} \cup \{D\}$ which represents what P computes on an input s in $\mathbb{S}$ is defined inductively as follows.

(1) $D \bullet s = D$,
(2) $S \bullet s = s$,
(3) $(a \circ P) \bullet s = P \bullet \mathtt{effect}_a(s)$,
(4) $P \trianglelefteq a \trianglerighteq Q \bullet s = (a \circ P) \bullet s$ if $y_a(s) = \mathtt{true}$, otherwise $(a \circ Q) \bullet s$.

In case P is infinite, *i.e.* $P = (P_n)_n$ for $(P_n)_n$ a monotone sequence, $P \bullet s = \bigsqcup_n P_n \bullet s$. Note that the partial ordering $\leq$ on $\mathbb{S}$ is defined by $D \leq s$ for all $s \in \mathbb{S}$.

If $P \bullet s = D$ for a state $s \in \mathbb{S}$ then we say that this computation produces no result. In other words, $P \bullet s = D$ precisely if for all $n \in \mathbb{N}$, $\pi_n(P) \bullet s = D$.

The previous definition suggests an equivalence called *input-output equivalence* for threads.

Definition 2.7. Two threads P and Q are **input-output equivalent** ($P \approx_{\mathbb{S}} Q$) over the state space $\mathbb{S}$ if $P \bullet s = Q \bullet s$ for all $s \in \mathbb{S}$.

We adopt the following convention on states of the state space $\mathbb{S}$: if in a program behavior only the variables $x_1, \dots, x_k$ occur, we represent states simply as $[\langle x_1, v_1\rangle, \dots, \langle x_k, v_k\rangle]$ with $v_i \in \texttt{values}(x_i)$ for $1 \leq i \leq k$.

Example 2.8. Consider the program

```
X ::= WHILE x>0 DO
              x:=x+1;
      END WHILE.
```

Then the behavior $|X|$ of X is defined as $|X| = ([x := x+1] \circ |X|) \trianglelefteq \langle x > 0\rangle \trianglerighteq S$. The effect and yield of the actions $[x{:=}x + 1]$ and $\langle x > 0\rangle$ are given as follows. For all $v \in \texttt{values}(x)$, $\texttt{effect}_{[x:=x+1]}([\langle x, v\rangle]) = [\langle x, v{+}1\rangle]$ and $y_{[x:=x+1]}([\langle x, v\rangle]) = \texttt{true}$. Moreover, $\texttt{effect}_{\langle x>0\rangle}([\langle x, v\rangle]) = [\langle x, v\rangle]$, and $y_{\langle x>0\rangle}([\langle x, v\rangle]) = \texttt{true}$ if $v > 0$ and $y_{\langle x>0\rangle}([\langle x, v\rangle]) = \texttt{false}$ otherwise. It is easy to see that if initially $v > 0$, then the computation $|X| \bullet [\langle x, v\rangle]$ goes on forever, *i.e.*, X produces no result for every input v of x that is greater than 0.

2.5. Noninterference based on input-output transformations

The earliest definition of *noninterference* of security information flow was given by Goguen and Meseguer [15]. Following the idea of [15], we provide a definition of noninterference based on input-output transformations for threads in BTA.

We suppose that program variables in $\texttt{Var}$ are classified into two security classes $\texttt{Var}_L$ (low) and $\texttt{Var}_H$ (high), $\texttt{Var}_L \cap \texttt{Var}_H = \emptyset$ and $\texttt{Var}_L \cup \texttt{Var}_H = \texttt{Var}$. We provide some notions of equivalence for states and threads based on security classes as follows.

Definition 2.9. Two states s and t of the state space $\mathbb{S}$ are **low equivalent**, denoted by $s =^L_{\mathbb{S}} t$, if for all $l \in \texttt{Var}_L$, $s.l.value = t.l.value$, otherwise $s \neq^L_{\mathbb{S}} t$. Similarly, two states s and t of the state space $\mathbb{S}$ are **high equivalent**, denoted by $s =^H_{\mathbb{S}} t$, if for all $h \in \texttt{Var}_H$, $s.h.value = t.h.value$, otherwise $s \neq^H_{\mathbb{S}} t$.

Definition 2.10. Two threads P and Q are **low quasi-equivalent** ($P \approx^L_{\mathbb{S}} Q$) over the state space $\mathbb{S}$ if for all low equivalent states s and t ($s =^L_{\mathbb{S}} t$), $P \bullet s = Q \bullet t = D$ or $P \bullet s$ and $Q \bullet t$ are low equivalent ($P \bullet s =^L_{\mathbb{S}} Q \bullet t$).

Informally speaking, a program is secure if its low output does not depend on its high input. This notion is translated to threads in BTA as follows.

Definition 2.11. A thread P is **noninterfering** ($P \in \texttt{NI}$) if it is low quasi-equivalent to itself. A program is **secure** if its behavior is noninterfering.

Example 2.12. Let $h \in \mathtt{Var}_H$ and $l \in \mathtt{Var}_L$. Consider the program behavior below.

$$\begin{array}{rcl} P & = & Q \unlhd \langle h == 1\rangle \unrhd R, \\ Q & = & [l := l+1] \circ Q, \\ R & = & [l := l-1] \circ R. \end{array}$$

It can be derived that $P \approx_{\mathbb{S}} D$. Hence $P \approx^L_{\mathbb{S}} P$. Thus, $P \in \mathtt{NI}$.

The previous example shows that non-termination implies NI. The following example illustrates insecure programs.

Example 2.13. Let $h \in \mathtt{Var}_H$ and $l \in \mathtt{Var}_L$. Consider the programs X and Y given in Example 2.5.

$$\begin{array}{rcl} |X| & = & ([l{:=}1] \circ S) \unlhd \langle h{==}1\rangle \unrhd ([l{:=}0] \circ S) \\ |Y| & = & [l{:=}0] \circ P \end{array}$$

where $P = ([h{:=}0] \circ P) \unlhd \langle h{==}0\rangle \unrhd ([l{:=}\, 1] \circ S)$. One can check that Y produces no result in the case that the input of h is 0. Furthermore, X and Y are not secure, since

$$\begin{array}{lcl} |X| \bullet [\langle h,0\rangle, \langle l,0\rangle] = [\langle h,0\rangle, \langle l,0\rangle] & \textit{while} & |X| \bullet [\langle h,1\rangle, \langle l,0\rangle] = [\langle h,1\rangle, \langle l,1\rangle]; \\ |Y| \bullet [\langle h,0\rangle, \langle l,0\rangle] = D & \textit{while} & |Y| \bullet [\langle h,1\rangle, \langle l,0\rangle] = [\langle h,1\rangle, \langle l,1\rangle]. \end{array}$$

2.6. Noninterference based on type systems

The definition of noninterference given by Goguen and Meseguer [15] is precise, but might require a complex computation. To simplify the checking, the approaches based on *type systems* (see [22] for an overview) have been developed. In these approaches, if a program is *well-typed* according to the *typing rules* of a type system then it has the noninterference property. This section introduces the type system of [24] in order to compare their results with ours later.

We suppose that there are two security classes $\mathcal{L}$ (low) and $\mathcal{H}$ (high), and a partial order $\subseteq$ between security classes with $\mathcal{L} \subseteq \mathcal{H}$ ($\mathcal{L} \neq \mathcal{H}$) ($\mathcal{L}$ is a subtyping of $\mathcal{H}$). The types used in the type systems of [24] are:

$$\begin{array}{llll} (\textit{data types}) & \tau & ::= & \mathcal{L} \mid \mathcal{H} \\ (\textit{phrase types}) & \rho & ::= & \tau \mid \tau\,\mathtt{var} \mid \tau\,\mathtt{cmd} \end{array}$$

Here type $\tau\,\mathtt{var}$ is the type of a program variable. Type judgments are of the form $\gamma \vdash X : \tau\,\mathtt{cmd}$ where X is an expression or a program and γ is a mapping from variables to types of variables, *i.e.* $\gamma(x) = \mathcal{L}\,\mathtt{var}$ if $x \in \mathtt{Var}_L$ and $\gamma(x) = \mathcal{H}\,\mathtt{var}$ otherwise. The typing rules of [24] are given in Table 1. We assume that all constants have type $\mathcal{L}$ (rule (INT)). Furthermore, we omit typing rules for some expressions since they are similar to rule (SUM).

According to these typing rules, every test and every expression is well-typed. In particular, a test (or an expression) has type $\mathcal{H}$ if it contains a high variable, otherwise it has type $\mathcal{L}$.

TABLE 1. Typing and subtyping rules.

(IDENT)	$\dfrac{\gamma(x) = \rho}{\vdash x : \rho}$
(INT)	$\vdash n : \mathcal{L}$
(R-VAL)	$\dfrac{\vdash e : \tau\ \mathtt{var}}{\vdash e : \tau}$
(SUM)	$\dfrac{\vdash e_1 : \tau \quad \vdash e_2 : \tau}{\vdash e_1 + e_2 : \tau}$
(ASSIGN)	$\dfrac{\vdash x : \tau\ \mathtt{var} \quad \vdash e : \tau}{\vdash x{:=}e : \tau\ \mathtt{cmd}}$
(COMPOSE)	$\dfrac{\vdash c_1 : \tau\ \mathtt{cmd} \quad \vdash c_2 : \tau\ \mathtt{cmd}}{\vdash c_1; c_2 : \tau\ \mathtt{cmd}}$
(IF)	$\dfrac{\vdash e : \tau \quad \vdash c_1 : \tau\ \mathtt{cmd} \quad \vdash c_2 : \tau\ \mathtt{cmd}}{\vdash \mathtt{IF}\ e\ \mathtt{THEN}\ c_1\ \mathtt{ELSE}\ c_2\ \mathtt{END\ IF} : \tau\ \mathtt{cmd}}$
(WHILE)	$\dfrac{\vdash e : \tau \quad \vdash c : \tau\ \mathtt{cmd}}{\vdash \mathtt{WHILE}\ e\ \mathtt{DO}\ c\ \mathtt{END\ WHILE} : \tau\ \mathtt{cmd}}$
(BASE)	$\mathcal{L} \subseteq \mathcal{H}$
(REFLEX)	$\rho \subseteq \rho$
(CMD$^-$)	$\dfrac{\tau_1 \subseteq \tau_2}{\tau_1\ \mathtt{cmd} \supseteq \tau_2\ \mathtt{cmd}}$
(SUBTYPE)	$\dfrac{\vdash X : \rho_1 \quad \rho_1 \subseteq \rho_2}{\vdash X : \rho_2}$

Assignments of the form $x{:=}e$, where $x \in \mathtt{Var}_L$ and e contains a high variable, are untypable in this type system because of the rule (ASSIGN). By this rule, $x{:=}e$ is accepted and has type $\mathcal{L}\ \mathtt{cmd}$ if both x and e have type $\mathcal{L}$, or it is accepted and has type $\mathcal{H}\ \mathtt{cmd}$ if x has type $\mathcal{H}$.

The meaning of $\gamma \vdash X : \tau\ \mathtt{cmd}$ is that type τ is a lower bound for the security level of the assigned variables of X. Hence, if the condition of a well-typed conditional statement (or a well-typed while-loop) has type $\mathcal{H}$ then every assigned variable contained in its branches (or its body) has type $\mathcal{H}$ as well.

Example 2.14. We consider the insecure program X taken from Example 2.5.

```
X ::= IF h==1 THEN l:=1 ELSE l:=0 END IF
```

where $l \in \texttt{Var}_L$ and $h \in \texttt{Var}_H$. This program is untypable in the type system of [24]. Here the condition $h == 1$ has type $\mathcal{H}$. According to rule (IF), X is accepted only if both l:=0 and l:=1 have type $\mathcal{H}\,\texttt{cmd}$ or a lower one. However, these assignments have type $\mathcal{L}\,\texttt{cmd}$ which contains type $\mathcal{H}\,\texttt{cmd}$ (rule (CMD$^-$)).

Example 2.15. Similar to the previous example, the following while-loop statement

```
WHILE h==1 DO
      l:=0;
END WHILE
```

is not well-typed, either.

We note that the typing rules in Table 1 respect termination-insensitive noninterference. That is to say, a well-typed program is secure by these typing rules if the program terminates successfully.

3. Characterizing actions with respect to security

This section characterizes actions of a thread as *insecure, secure, invisible, low* and *high* actions with respect to security.

We consider the following examples. The untypable action [l:=h] with $l \in \texttt{Var}_L$ and $h \in \texttt{Var}_H$ is *insecure* in our approach because it reveals information of h. The assignment [l:=1] is a *low* action because it has effect on the low subspace. The test $\langle h$==1$\rangle$ and the assignment [h:=1] are regarded as *high* actions since they have something to do with the high subspace. Finally, the secure actions that have no effect on the low subspace such as tests and high actions are *invisible.* Formally:

Definition 3.1.

1. An action a is **secure** if it does not reveal any high-security information, *i.e.* $\texttt{effect}_a(s) =^L_{\mathbb{S}} \texttt{effect}_a(t)$ for all low equivalent states $s, t \in \mathbb{S}$ ($s =^L_{\mathbb{S}} t$).
2. A secure action a is **low** if it has effect on the low subspace, *i.e.* $\texttt{effect}_a(s) \neq^L_{\mathbb{S}} s$ for some $s \in \mathbb{S}$.
3. A secure action a is **invisible** if it has no effect on the low subspace, *i.e.* $\texttt{effect}_a(s) =^L_{\mathbb{S}} s$ for all states $s \in \mathbb{S}$.
4. An invisible action a is **high** if it has effect or yield on the high subspace, *i.e.* $\texttt{effect}_a(s) \neq^H_{\mathbb{S}} s$ for some state $s \in \mathbb{S}$, or $y_a(s) \neq y_a(t)$ for some states $s, t \in \mathbb{S}$ such that $s \neq^H_{\mathbb{S}} t$.

We note that the terminology that actions are secure (or not) in Definition 3.1 is obtained from the standard terminology of secure information flow of [24].

Let Σ_S be the set of secure actions, Σ_I the set of invisible actions, Σ_L the set of low actions and Σ_H the set of high actions. Then $\Sigma_S = \Sigma_I \cup \Sigma_L \subseteq \Sigma$, $\Sigma_I \cap \Sigma_L = \emptyset$ and $\Sigma_H \subseteq \Sigma_I$.

Lemma 3.2. *Let a be an action in Σ, $a \notin \Sigma_H$. Then for all low equivalent states $s, t \in \mathbb{S}$, $y_a(s) = y_a(t)$.*

Proof. We distinguish three cases:

1. $s =^H_{\mathbb{S}} t$. Since $s =^L_{\mathbb{S}} t$, $s = t$. Thus, $y_a(s) = y_a(t)$.
2. $s \neq^H_{\mathbb{S}} t$ and a has some effect on the state space. Then $y_a(s) = y_a(t) =$ `true`.
3. $s \neq^H_{\mathbb{S}} t$ and a has no effect on the state space. If $y_a(s) \neq y_a(t)$ then a has yield on the high subspace. By Definition 3.1, a is a high action. This contradicts the assumption that $a \notin \Sigma_H$. Therefore, $y_a(s) = y_a(t)$. □

To choose secure, invisible, low and high actions, let $\Sigma_{\texttt{welltyped}}$ be the set of well-typed actions by the typing rules in Table 1, $\Sigma_{\mathcal{L}+\mathcal{H}}$ the set of tests, $\Sigma_{\mathcal{L}\texttt{cmd}}$ the set of assignments with type $\mathcal{L}$ `cmd`, and $\Sigma_{\mathcal{H}+\mathcal{H}\texttt{cmd}}$ the set of actions with type $\mathcal{H}$ and $\mathcal{H}$ `cmd` (see Sect. 2.6). Then:

Lemma 3.3.

1. $\Sigma_{\texttt{welltyped}} \subseteq \Sigma_S$.
2. $\Sigma_{\mathcal{H}+\mathcal{H}\texttt{cmd}} \cup \Sigma_{\mathcal{L}+\mathcal{H}} \subseteq \Sigma_I$.
3. $\Sigma_L \subseteq \Sigma_{\mathcal{L}\texttt{cmd}} \subseteq \Sigma_L \cup \Sigma_I = \Sigma_S$.
4. $\Sigma_H \subseteq \Sigma_{\mathcal{H}+\mathcal{H}\texttt{cmd}} \subseteq \Sigma_H \cup \Sigma_I = \Sigma_I$.

Hence one can choose the sets Σ_S of secure actions, Σ_L of low actions and Σ_H of high actions as the sets $\Sigma_{\texttt{welltyped}}$, $\Sigma_{\mathcal{L}\texttt{cmd}}$ and $\Sigma_{\mathcal{H}+\mathcal{H}\texttt{cmd}}$, respectively. The set Σ_I of invisible actions can be chosen as $\Sigma_I = \Sigma_H \cup \Sigma_{\mathcal{L}+\mathcal{H}}$. The previous lemma shows that these decisions are merely approximations of Σ_S, Σ_L, Σ_H and Σ_I. However, we can extend or restrict these sets with certain associated actions, in order to optimize them. The closer we get to the optimal solutions, the more secure programs can be accepted. For instance, the set Σ_S can be extended with actions of the form $[l{:=}h - h]$. The set Σ_H can be restricted by removing the actions $\langle l + h > h \rangle$.

4. Labeled transition systems over BTA

In this section, we define a *labeled transition system* over BTA, and a *bisimulation* equivalence that is used for our definition of noninterference.

4.1. Labeled transition systems

A **labeled transition system** (LTS) with **termination** S and **deadlock** D is a pair $(\mathbb{P}, \rightarrow)$ with $\mathbb{P}$ a class of threads, and $\rightarrow \subseteq \mathbb{P} \times (\Sigma \times \{T, F\}) \times \mathbb{P}$ a set of **transitions**. We write $P \xrightarrow{a,\kappa} Q$ with $a \in \Sigma$ and $\kappa \in \{T, F\}$ for $(P, (a, \kappa), Q) \in \rightarrow$. An LTS is a **finite-state** (regular) LTS if both $\mathbb{P}$ and Σ are finite.

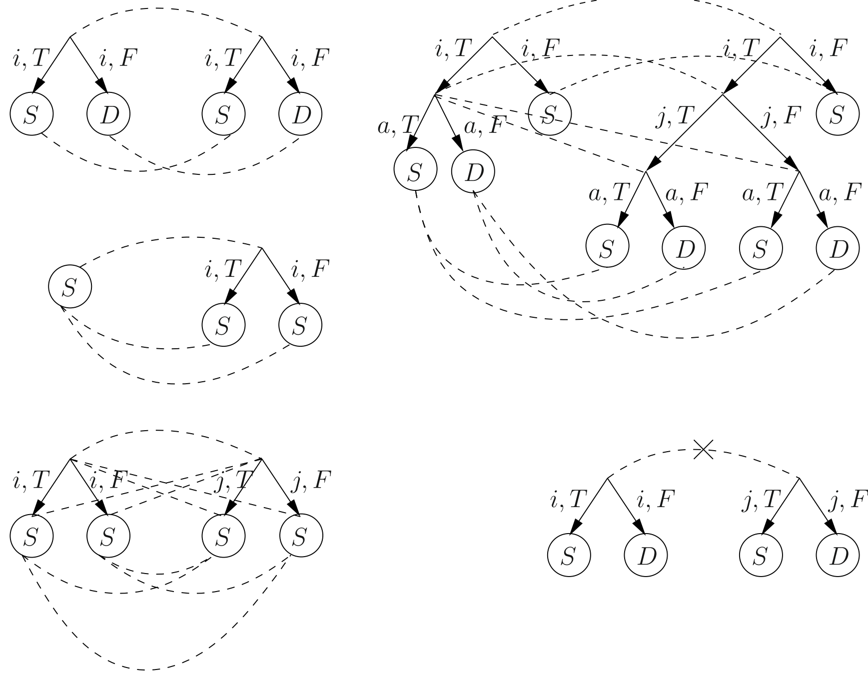

FIGURE 1. Examples of bisimulation with $I = \{i, j\}$ and $a \in \Sigma$. The dashed lines represent bisimulation up to I between threads.

For a state P, if $P = S$ then it is a termination state. If $P = D$ then it is a deadlock. If $P = P_1 \trianglelefteq a \trianglerighteq P_2$ then $P \xrightarrow{a,T} P_1$ and $P \xrightarrow{a,F} P_2$.

We note that since program behaviors in the programming language `Lang` can be represented as regular threads, they can always be associated with a finite-state LTS.

4.2. Bisimulation up to I

Let $I \subseteq \Sigma$ be a set of actions. The relation *bisimulation up to I* identifies threads behaving the same from the view of the actions which are not in I. In other words, this bisimilarity is obtained by ignoring the presence of actions of I. We will use the following notion to define bisimulation up to I.

Definition 4.1. Let P be a thread. A thread Q is a **residual** thread of P, written as $P \Rightarrow Q$, if there is a path of transitions $P = P_0 \xrightarrow{a_0,\kappa_0} P_1 \xrightarrow{a_1,\kappa_1} \cdots \xrightarrow{a_{n-1},\kappa_{n-1}} P_n = Q$ with $n \geq 0$. We write $P \overset{I}{\Rightarrow} Q$ if $a_i \in I$ for all $1 \leq i \leq n$.

Definition 4.2. A **bisimulation up to** I is a symmetric binary relation $\mathcal{B}$ on threads satisfying:

1. If $(S, Q) \in \mathcal{B}$ then there exists a path $Q \overset{I}{\Rightarrow} S$.

2. If $(P, Q) \in \mathcal{B}$ and $P \xrightarrow{a,\kappa} P'$ then either:
   (a) $a \in I$ and $(P', Q) \in \mathcal{B}$, or:
   (b) there exists a path $Q \overset{I}{\Rightarrow} Q_1 \xrightarrow{a,\kappa} Q'$ such that $(P, Q_1) \in \mathcal{B}$ and $(P', Q') \in \mathcal{B}$.

Two threads P and Q are **bisimilar up to** I, denoted by $(P \Leftrightarrow_I Q)$, if there is a bisimulation up to I relation $\mathcal{B}$ such that $(P, Q) \in \mathcal{B}$.

We note that in Clause 2(b) of the above definition, it is allowed that $a \in I$. Furthermore, in case $I = \emptyset$, bisimulation up to I coincides with *bisimulation equivalence* proposed by [20].

Proposition 4.3. *Bisimulation up to I is an equivalence.*

Figure 1 illustrates the notion of bisimulation up to I between threads.

Proposition 4.4. *Let $I \subseteq I' \subseteq \Sigma$. Then $\Leftrightarrow_I \subseteq \Leftrightarrow_{I'}$.*

We omit the proof of the previous proposition, since it is similar to the proof for branching bisimulation [14] given in [1].

5. Termination-insensitive noninterference in TA

We assume the existence of a set I of invisible actions that contains the set Σ_H of high actions ($\Sigma_H \subseteq I \subseteq \Sigma_I$). In this section, we present *termination-insensitive noninterference up to I* ($\mathtt{TINI}_I$) for threads. We prove soundness for our definition and show that we accept all secure programs that are typable in the type system of [24].

Let $\sigma(P)$ denote the set of actions occurring in a process P. We consider the following programs.

```
X ::= l:=h
Y ::= IF h==1 THEN l:=0 ELSE l:=1 END IF
Z ::= IF h==1 THEN l:=0 ELSE l:=0 END IF
```

where $l \in \mathtt{Var}_L$ and $h \in \mathtt{Var}_H$ are low and high variables. Program X is insecure and is not accepted by the type system of [24] since the assignment $l{:=}h$ reveals the value of h. The insecure program Y is untypable in this type system, too. It is because the type of the assigned variable l in the branches of the conditional statement is lower than the type of the condition $h{==}1$ (see Sect. 2.6). By the reason of the same fact, program Z is rejected by these type systems although it is secure. In our approach, we also reject X and Y, but accept Z. Because this program behaves the same after the execution of $\langle h{==}1\rangle$. Hence, an attacker cannot learn the value of h of program Z through branching on the condition $\langle h == 1\rangle$.

We then propose a notion of termination-insensitive noninterference for threads as follows. A thread is termination-insensitive noninterfering if its actions are secure. Furthermore, its behavior from the view of low actions, is always the same regardless of the returned boolean value after the execution of a high action is. Formally:

Definition 5.1. Let $\Sigma_H \subseteq I \subseteq \Sigma_I$. A thread P is **termination-insensitive noninterfering up to** I ($P \in \mathtt{TINI}_I$) if

1. all actions of P are secure, *i.e.* $\sigma(P) \subseteq \Sigma_S$,
2. for all residual threads Q of P such that $Q \xrightarrow{a,\kappa} Q'$ with $a \in \Sigma_H$, $Q \leftrightharpoons_I Q'$.

In the definition above, in case $I = \Sigma_I$, $\mathtt{TINI}_I$ would accept the most secure programs, while in case $I = \Sigma_H$, $\mathtt{TINI}_I$ would accept the least secure programs. For instance, consider the following example.

Example 5.2. Let U, V and W be programs given as follows.

```
U ::= IF h==1 THEN h:=h+1 ELSE h:=h-1 END IF.
V ::= l:=0;
      IF h==0 THEN
         IF l==1 THEN h:=1 ELSE h:=2 END IF;
      ELSE
         h:=3;
      END IF;
W ::= WHILE h<10 DO
            h:=h+1;
      END WHILE.
```

Program U and W are accepted in both cases $I = \Sigma_I$ and $I = \Sigma_H$. However, program V is only accepted in case $I = \Sigma_I$. This program is rejected by $\mathtt{TINI}_{\Sigma_H}$ since it will proceed with the test $\langle l == 1\rangle \notin \Sigma_H$ if $h = 0$, while in the other case it will not.

Proposition 5.3. *Let* $\Sigma_H \subseteq I \subseteq I' \subseteq \Sigma_I$. *Then* $\mathtt{TINI}_I \subseteq \mathtt{TINI}_{I'}$.

Proof. This follows from Proposition 4.4. □

We now show that in case $I = \Sigma_I$, we accept all secure programs that are accepted by the type system of [24].

Theorem 5.4. *Let X be a program in the programming language* Lang. *If X is well-typed by the typing rules in Table 1 then* $|X| \in \mathtt{TINI}_{\Sigma_I}$.

Proof. Since all actions in X are well-typed, they are secure. Let Q be a residual thread of X such that $Q \xrightarrow{a,\kappa} Q'$ with $a \in \Sigma_H$. By the typing rules of Table 1, if a is the condition of a conditional statement (or a while-loop) then all the assignments within the branches (or the body) of that statement are high. Hence there exists a residual thread P of Q satisfying the following property: for every path $Q = Q_0 \xrightarrow{a_0,\kappa_0} Q_1 \xrightarrow{a_1,\kappa_1} \cdots$ from Q, there exists $n \in \mathbb{N}$ such that $Q_n = P$ and $a_i \in \Sigma_I$ for all $i < n$. We define that $(Q_i, P) \in \mathcal{B}$ for all $i \leq n$. Then $\mathcal{B}$ is a bisimulation up to Σ_I. Hence $Q \leftrightharpoons_{\Sigma_I} Q'$. By Definition 5.1, $|X| \in \mathtt{TINI}_{\Sigma_I}$. □

It should be noticed that $\mathtt{TINI}_I$ will accept *insecure* programs in certain cases as can be seen in Example 5.5.

Example 5.5. We consider the program below:

```
T ::= WHILE h>0 DO
          h:=h+1;
      END WHILE
```

where $h \in \mathtt{Var}_H$. It can be derived that $T \in \mathtt{TINI}_I$ for all sets $\Sigma_H \subseteq I \subseteq \Sigma_I$. However, $T \notin \mathtt{NI}$ since $|X| \bullet [\langle h, 0\rangle] = [\langle h, 0\rangle]$ while $|X| \bullet [\langle h, 1\rangle] = D$.

The program in Example 5.5 exemplifies a *termination-leak* insecure program. To preserve the noninterference property for $\mathtt{TINI}_I$ we impose a condition that *the program terminates successfully* as in [24], given as follows.

Definition 5.6. A thread P **terminates successfully** if for all $s \in \mathbb{S}$, $P \bullet s \neq D$. Let BTA^T_Σ be the set of all threads that terminates successfully.

Definition 5.6 implies that for a thread $P \in \mathrm{BTA}^T_\Sigma$ and a state $s \in \mathbb{S}$, there is a finite deterministic path $(P_0 = P, s_0 = s) \xrightarrow{a_0,\kappa_0} (P_1, s_1) \xrightarrow{a_1,\kappa_1} \cdots \xrightarrow{a_{n-1},\kappa_{n-1}} (P_n = S, s_n)$ satisfying that $P_i \xrightarrow{a_i,\kappa_i} P_{i+1}$, $\kappa_i = y_{a_i}(s_i)$ and $s_{i+1} = \mathtt{effect}_{a_i}(s_i)$ for all $0 \leq i < n$.

Theorem 5.7 (soundness of $\mathtt{TINI}_I$)**.** *Let $\Sigma_H \subseteq I \subseteq \Sigma_I$. Then $\mathtt{TINI}_I \cap \mathrm{BTA}^T_\Sigma \subseteq \mathtt{NI}$.*

Proof. Let $P \in \mathtt{TINI}_I \cap \mathrm{BTA}^T_\Sigma$. We show that for all low equivalent states $s, r \in \mathbb{S}$ $(s =^L_{\mathbb{S}} r)$, $P \bullet s =^L_{\mathbb{S}} P \bullet r$. Since P terminates successfully, there are two finite deterministic paths obtained by the computations of P with s and r, given as in the following.

$$(P_0 = P, s_0 = s) \overset{I}{\Rightarrow} (P'_0, s'_0) \xrightarrow{a_0,\kappa_0} (P_1, s_1) \overset{I}{\Rightarrow} \cdots \xrightarrow{a_{n-1},\kappa_{n-1}} (P_n = S, s_n) \text{ and}$$
$$(Q_0 = P, r_0 = r) \overset{I}{\Rightarrow} (Q'_0, r'_0) \xrightarrow{b_0,\gamma_0} (Q_1, r_1) \overset{I}{\Rightarrow} \cdots \xrightarrow{b_{m-1},\gamma_{m-1}} (Q_m = S, r_m)$$

where $P_i \overset{I}{\Rightarrow} P'_i$, $P_i \Leftrightarrow_I P'_i$ and $P'_i \not\Leftrightarrow_I P_{i+1}$ for $0 \leq i < n$, and where $Q_j \overset{I}{\Rightarrow} Q'_j$, $Q_j \Leftrightarrow_I Q'_j$ and $Q'_j \not\Leftrightarrow_I Q_{j+1}$ for $0 \leq j < m$. We prove by induction on i that $P_i \Leftrightarrow_I Q_i$ and $s_i =^L_{\mathbb{S}} r_i$ for all $0 \leq i \leq n$. We consider the following possibilities:

1. $i = 0$. Then $P_0 = Q_0 = P$. Thus, $P_0 \Leftrightarrow_I Q_0$ and $s_0 = s =^L_{\mathbb{S}} r = r_0$.
2. Assume that $P_i \Leftrightarrow_I Q_i$ and $s_i =^L_{\mathbb{S}} r_i$. We prove that $P_{i+1} \Leftrightarrow_I Q_{i+1}$ and $s_{i+1} =^L_{\mathbb{S}} r_{i+1}$. One can derive that $P'_i \Leftrightarrow_I Q'_i$. Moreover $a_i = b_i$ because of $P'_i \xrightarrow{a_i,\kappa_i} P_{i+1}$, $Q'_i \xrightarrow{b_i,\gamma_i} Q_{i+1}$, $P'_i \not\Leftrightarrow_I P_{i+1}$ and $Q'_i \not\Leftrightarrow_I Q_{i+1}$. Furthermore, by Definition 5.1, $a_i \notin H$. Since invisible actions do not have effect on the low space, $s'_i =^L_{\mathbb{S}} s_i =^L_{\mathbb{S}} r_i =^L_{\mathbb{S}} r'_i$. It follows from Lemma 3.2 and $s'_i =^L_{\mathbb{S}} r'_i$ that $\kappa_i = y_{a_i}(s'_i) = y_{b_i}(r'_i) = \gamma_i$. This implies that $P_{i+1} \Leftrightarrow_I Q_{i+1}$ and $s_{i+1} = \mathtt{effect}_{a_i}(s'_i) =^L_{\mathbb{S}} = \mathtt{effect}_{a_i}(r'_i) = r_{i+1}$.

Hence $P_i \Leftrightarrow_I Q_i$ and $s_i =^L_{\mathbb{S}} r_i$ for all $0 \leq i \leq n$. Since $P_n = Q_m = S$, $n = m$. Therefore, $P \bullet s =^L_{\mathbb{S}} s_n =^L_{\mathbb{S}} r_n =^L_{\mathbb{S}} P \bullet r$. By Definition 2.11, $P \in \mathtt{NI}$. $\square$

Our definition of TINI can be applied to unstructured programming languages because they are based on program behaviors. Furthermore, it is also suitable for considering noninterference properties in multithreaded languages since a multi-thread is also a single thread in thread algebra.

6. An interleaving strategy with respect to noninterference

It would be natural if termination-insensitive noninterference is compositional with respect to strategic interleaving operators. Unfortunately, it is shown in the following example that the compositionality property does not hold for $\texttt{TINI}_I$ with respect to the simplest interleaving scheduling like cyclic rotation.

Example 6.1. Let $h \in \texttt{Var}_H, l \in \texttt{Var}_L$. Let α and β be two single threads defined as

$$\begin{array}{lll} \alpha & = & (([h{:=}h+1] \circ [l{:=}0] \circ S) \trianglelefteq \langle h{=}{=}1 \rangle \trianglerighteq ([l{:=}0] \circ S); \\ \beta & = & [l{:=}1] \circ [l{:=}2] \circ S. \end{array}$$

Let $I = \Sigma_H = \{[h{:=}h+1], \langle h{=}{=}1 \rangle\}$. It can be checked that α and β are secure. However $\|_{\text{csi}} (\alpha \curvearrowright \beta)$ is not secure, since

$$\begin{array}{ll} \|_{\text{csi}} (\alpha \curvearrowright \beta) = & ([l{:=}1] \circ [h{:=}h+1] \circ [l{:=}2] \circ [l{:=}0] \circ S) \\ & \trianglelefteq \langle h{=}{=}1 \rangle \trianglerighteq \\ & ([l{:=}1] \circ [l{:=}0] \circ [l{:=}2] \circ S) \end{array}$$

which produces $l = 0$ if $h = 1$, and $l = 2$ otherwise.

To preserve compositionality for $\texttt{TINI}_I$, we propose in this section a variant of the cyclic interleaving operator called the *cyclic strategic interleaving with persistence* operator ($\|^I_{\text{csi}}$) for thread algebra. This strategy is similar to the *current thread persistence* operator of [6] and will not invoke the rotation of a thread vector if the current action is considered *persistent* in I. We will show that bisimulation up to I is a congruence under this interleaving strategy. And therefore, $\texttt{TINI}_I$ satisfies compositionality with respect to the cyclic strategic interleaving with persistence operator.

6.1. The cyclic strategic interleaving with persistence operator

Cyclic strategic interleaving with persistence $\|^I_{\text{csi}}$ is defined formally as follows.

Definition 6.2. The axioms for the **cyclic strategic interleaving with persistence** $\|^I_{\text{csi}}$ operator are given for finite threads by

$$\begin{array}{lll} \|^I_{\text{csi}} (\langle\rangle) & = & S \\ \|^I_{\text{csi}} (\langle S \rangle \curvearrowright \alpha) & = & \|^I_{\text{csi}} (\alpha) \\ \|^I_{\text{csi}} (\langle D \rangle \curvearrowright \alpha) & = & S_D(\|^I_{\text{csi}} (\alpha)) \\ \|^I_{\text{csi}} (\langle x \trianglelefteq a \trianglerighteq y \rangle \curvearrowright \alpha) & = & \|^I_{\text{csi}} (\langle x \rangle \curvearrowright \alpha) \trianglelefteq a \trianglerighteq \|^I_{\text{csi}} (\langle y \rangle \curvearrowright \alpha) \; \textit{if } a \in I \\ \|^I_{\text{csi}} (\langle x \trianglelefteq a \trianglerighteq y \rangle \curvearrowright \alpha) & = & \|^I_{\text{csi}} (\alpha \curvearrowright \langle x \rangle) \trianglelefteq a \trianglerighteq \|^I_{\text{csi}} (\alpha \curvearrowright \langle y \rangle) \; \textit{otherwise.} \end{array}$$

We note that $\|^{\emptyset}_{\text{csi}} = \|_{\text{csi}}$. For a thread vector α of arbitrary (finite or infinite) threads $\alpha = \alpha_1 \curvearrowright \cdots \curvearrowright \alpha_n$, $\|^I_{\text{csi}}(\alpha)$ is determined by its projective sequence:

$$\pi_n(\|^I_{\text{csi}}(\alpha)) = \pi_n(\|^I_{\text{csi}}(\pi_n(\alpha_1) \curvearrowright \cdots \curvearrowright \pi_n(\alpha_n))).$$

Example 6.3. We now return to Example 6.1. The multithread $\|^I_{\text{csi}}(\alpha \curvearrowright \beta)$ is secure, since

$$\begin{aligned} \|^I_{\text{csi}}(\alpha \curvearrowright \beta) = \quad & ([h{:=}h+1] \circ [l{:=}0] \circ [l{:=}1] \circ [l{:=}2] \circ S) \\ & \trianglelefteq \langle h{==}1 \rangle \trianglerighteq \\ & ([l{:=}0] \circ [l{:=}1] \circ [l{:=}2] \circ S) \end{aligned}$$

which always produces l=2.

6.2. Congruence with respect to $\texttt{TINI}_I$

This section shows that bisimulation equivalence is a congruence with respect to $\|^I_{\text{csi}}$ if all threads are deadlock-free.

Definition 6.4. A thread is **deadlock-free** if it does not contain a residual deadlock D.

Lemma 6.5 (congruence with respect to $\texttt{TINI}_I$)**.** *Let $\Sigma_H \subseteq I \subseteq \Sigma_I$, and let P_i and Q_i $(1 \leq i \leq n)$ be deadlock-free threads such that $P_i \leftrightarrow_I Q_i$. Then*

$$\|^I_{\text{csi}}(\langle P_1 \rangle \curvearrowright \langle P_2 \rangle \curvearrowright \cdots \curvearrowright \langle P_n \rangle) \leftrightarrow_I \|^I_{\text{csi}}(\langle Q_1 \rangle \curvearrowright \langle Q_2 \rangle \curvearrowright \cdots \curvearrowright \langle Q_n \rangle).$$

Proof. Let $\mathcal{B}$ be a binary relation defined as follows. For threads P and Q, $(P, Q) \in \mathcal{B}$ if there are sequences α and β of the same length n for some $n \in \mathbb{N}$ such that $P = \|^I_{\text{csi}}(\alpha)$, $Q = \|^I_{\text{csi}}(\beta)$, and for all process components α_i and β_i of α and β, $\alpha_i \leftrightarrow_I \beta_i$ respectively. We show that $\mathcal{B}$ is a bisimulation up to I.

1. $P = S$. Then for all i, $1 \leq i \leq n$, $\alpha_i = S$. Since, $\beta_i \leftrightarrow_I \alpha_i$. It is not hard to see that there exists a finite path $Q \overset{I}{\Rightarrow} Q'$ with $Q' = S$.
2. $P \xrightarrow{a,\kappa} P'$. Since $\alpha_j \neq D$ for all $1 \leq j \leq n$, there exists i such that for all $j < i$, $\alpha_j = S$ and $\alpha_i \xrightarrow{a,\kappa} x$. Furthermore, $P = \|^I_{\text{csi}}(\alpha_i \curvearrowright \cdots \curvearrowright \alpha_n)$ and $P' = \|^I_{\text{csi}}(x \curvearrowright \alpha_{i+1} \curvearrowright \cdots \curvearrowright \alpha_n)$. Since for all $j < i$, $\alpha_j \leftrightarrow_I \beta_j$, there exist finite paths $\beta_j \overset{I}{\Rightarrow} S$. Since $\alpha_i \leftrightarrow_I \beta_i$, there exists a finite path $\beta_i \overset{I}{\Rightarrow} \beta'_i \xrightarrow{a,\kappa} y$ with $\alpha_i \leftrightarrow_I \beta'_i$ and $x \leftrightarrow_I y$. Let $Q_1 = \|^I_{\text{csi}}(\beta'_i \curvearrowright \cdots \curvearrowright \beta_n)$ and $Q' = \|^I_{\text{csi}}(y \curvearrowright \beta_{i+1} \curvearrowright \cdots \curvearrowright \beta_n)$. Then $Q \overset{I}{\Rightarrow} Q_1 \xrightarrow{a,\kappa} Q'$ with $P \leftrightarrow_I Q_1$ and $P' \leftrightarrow_I Q'$.

Thus, $\mathcal{B}$ is a bisimulation up to I. □

One may expect that Lemma 6.5 also works on the case that threads can have deadlocks. However, the following example shows that it is not the case.

Example 6.6. Let $P = \|_{\mathrm{csi}}^{\{i\}} (\langle i^\infty \rangle \curvearrowright \langle a \circ S \rangle)$ and $Q = \|_{\mathrm{csi}}^{\{i\}} (\langle D \rangle \curvearrowright \langle a \circ S \rangle)$. Then $P = i^\infty$ and $Q = a \circ D$. It is obvious that $P \not\leftrightarrows_{\{i\}} Q$ although $i^\infty \leftrightarrows_{\{i\}} D$ and $a \circ S \leftrightarrows_{\{i\}} a \circ S$.

6.3. Compositionality of $\mathtt{TINI}_I$

We now show that $\mathtt{TINI}_I$ satisfies compositionality with respect to $\|_{\mathrm{csi}}^I$, provided that the threads contain no deadlocks.

Theorem 6.7 (compositionality of $\mathtt{TINI}_I$)**.** *Let $\Sigma_H \subseteq I \subseteq \Sigma_I$ and let $P_i \in \mathtt{TINI}_I$ be deadlock-free threads for all $1 \leq i \leq n$. Then*

$$\|_{\mathrm{csi}}^I (\langle P_1 \rangle \curvearrowright \langle P_2 \rangle \curvearrowright \cdots \curvearrowright \langle P_n \rangle) \in \mathtt{TINI}_I.$$

Proof. Let $P = \|_{\mathrm{csi}}^I (\langle P_1 \rangle \curvearrowright \langle P_2 \rangle \curvearrowright \cdots \curvearrowright \langle P_n \rangle)$. We show that for all residual threads Q of P, if $Q = Q' \trianglelefteq a \trianglerighteq Q''$ with $a \in \Sigma_H$ then $Q' \leftrightarrows_I Q''$. It can be derived that $Q = S$ or $Q = \|_{\mathrm{csi}}^I (\langle Q_{i_1} \rangle \curvearrowright \langle Q_{i_2} \rangle \curvearrowright \cdots \curvearrowright \langle Q_{i_m} \rangle)$, where Q_k is a residual thread of P_k for $1 \leq k \leq n$. Let $Q_{i_1} = Q'_{i_1} \trianglelefteq a \trianglerighteq Q''_{i_1}$. Then

$$\begin{array}{rcl} Q' & = & \|_{\mathrm{csi}}^I (\langle Q'_{i_1} \rangle \curvearrowright \langle Q_{i_2} \rangle \curvearrowright \cdots \curvearrowright \langle Q_{i_m} \rangle) \\ Q'' & = & \|_{\mathrm{csi}}^I (\langle Q''_{i_1} \rangle \curvearrowright \langle Q_{i_2} \rangle \curvearrowright \cdots \curvearrowright \langle Q_{i_m} \rangle). \end{array}$$

Since $P_{i_1} \in \mathtt{TINI}_I$, $Q'_{i_1} \leftrightarrows_I Q''_{i_1}$. It follows from Lemma 6.5 that $Q' \leftrightarrows_I Q''$. Therefore, $P \in \mathtt{TINI}_I$. □

7. Concluding remarks

In this paper, we have interpreted the notion of termination-insensitive noninterference, comparable to the one given on type systems in [24], in thread algebra. We have proven soundness for our definition, and shown that we accept all secure programs that are accepted by the type system of [24]. Hence, thread algebra is suitable as a process-algebraic framework for formalizing and analyzing security properties in multithreaded languages. Furthermore, it is also an applicable framework for considering security properties for unstructured programs.

In order to preserve compositionality for termination-insensitive noninterference, we have proposed the cyclic interleaving with persistence operator, and shown the analysis can be made compositional if all high actions are persistent and the threads are deadlock-free.

For checking the noninterference properties presented in this paper, one can apply existing techniques [16,19] and existing tools [10,12,13] for checking process-equivalence to develop our security checkers.

We note that other standard notions of noninterference such as termination-sensitive noninterference [23] and timing-sensitive noninterference [21] are also modeled in thread algebra (see *e.g.* [26]). We hereby show that previous work on security for sequential and multithreaded systems can be reconsidered in thread algebra.

References

[1] T. Basten, Branching bisimulation is an equivalence indeed. *Inform. Process. Lett.* **58** (1996) 333–337.

[2] D.E. Bell and L.J. La Padula, *Secure computer systems: mathematical foundations and model.* Tech. Rep. M74-244, MITRE Corporation, Bedford, Massachussets (1973).

[3] J.A. Bergstra and I. Bethke, Molecular dynamics. *J. Logic Algebr. Program.* **51** (2002) 125–156.

[4] J.A. Bergstra and J.W. Klop, *Fixed point semantics in process algebra*. Technical Report IW 208, Mathematical Center, Amsterdam (1982).

[5] J.A. Bergstra and M.E. Loots, Program algebra for sequential code. *J. Logic Algebr. Program.* **51** (2002) 125–156.

[6] J.A. Bergstra and C.A. Middelburg, Thread algebra for strategic interleaving. *Formal Aspects Comput.* **19** (2007) 445–474. Preliminary version: Computer Science Report PRG0404, Sectie Software Engineering, University of Amsterdam.

[7] J.A. Bergstra and C.A. Middelburg, A thread algebra with multi-level strategic interleaving. *Theor. Comput. Syst.* **41** (2007). Preliminary versions: in *CiE*, edited by S.B. Cooper, B. Loewe and L. Torenvliet. *Lect. Notes Comput. Sci.* **3526** (2005) 35–48; Computer Science Report 06-28, Department of Mathematics and Computing Science, Eindhoven University of Technology.

[8] J.A. Bergstra and C.A. Middelburg, Maurer computers for pipelined instruction processing. *J. Math. Struct. Comput. Sci.* **18** (2008) 373–409.

[9] J.A. Bergstra and A. Ponse, A bypass of Cohen's impossibility result. in *Advances in Grid Computing-EGC 2005*, edited by P.M.A. Sloot, A.G. Hoekstra, T. Priol, A. Reinefeld and M. Bubak. *Lect. Notes Comput. Sci.* **3407** (2005) 1097–1106.

[10] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I.A. van Langevelde, B. Lisser and J.C. van de Pol, μCRL: a toolset for analysing algebraic specifications. in *Proc. 13th Conference on Computer Aided Verification-CAV'01*, edited by G. Berry, H. Common and A. Finkel. *Lect. Notes Comput. Sci.* **2102** (2001) 250–254.

[11] D.E. Denning, A lattice model of secure information flow. *Commun. ACM* **19** (1976) 236–243.

[12] R. Focardi and R. Gorrieri, Automatic compositional verification of some security properties for process algebras, in *Proc. of TACA'96*, edited by T. Margaria and B. Steffen. *Lect. Notes Comput. Sci.* **1055** (1996) 111–130.

[13] R. Focardi and R. Gorrieri, The compositional security checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering* **23** (1997) 550–571.

[14] R.J. van Glabbeek and W.P. Weijland, Branching time and abstraction in bisimulation semantics. *J. ACM* **43** (1996) 555–600.

[15] J. Goguen and J. Meseguer, Secure policies and security models, in *Proc. IEEE Symp. Security and Privacy* (1982) 11–20.

[16] J.F. Groote and F.W. Vaandrager, An efficient algorithm for branching bisimulation and stuttering equivalence, in *Proc. ICALP 90*, edited by M.S. Paterson. *Lect. Notes Comput. Sci.* **443** (1990) 626–638.

[17] A.C. Meyers, Jflow: Practical mostly-static information flow control, in *Proc. ACM Symp. on Principles of Programming Languages* (1999) 228–241.

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

[19] R. Paige and R. Tarjan, Three partition refinement algorithms. *SIAM J. Comput.* **16** (1987) 973–989.

[20] D.M.R. Park, Concurrency and automata on infinite sequences, in *Proc. 5th GI Conference*, edited by P. Deussen, *Lect. Notes Comput. Sci.* **104** (1982) 167–183.

[21] A. Sabelfeld and H. Mantel, Static confidentiality enforcement for distributed programs. in *Proc. Symp. on Static Analysis. Lect. Notes Comput. Sci.* **2477** (2002) 376–394.

[22] A. Sabelfeld and A. Myers, Language-based information flow security. *IEEE J. Sel. Areas Commun.* **21** (2003) 5–19.
[23] G. Smith and D. Volpano, Secure information flow in multi-threaded imperative languages, in *Proc. POPL'98* **29** (1998) 355–364.
[24] D. Volpano, G. Smith and C. Irvine, A sound type system for secure flow analysis. *J. Comput. Secur.* **4** 167–187 (1996).
[25] T.D. Vu, Denotational semantics for thread algebra. *J. Logic Algebr. Program.* **74** (2007) 94–111.
[26] T.D. Vu, *Semantics and applications of process and program algebra*. Ph.D. thesis, University of Amsterdam (2007).

Communicated by Giuseppe Longo.
Received May 8, 2007. Accepted September 11, 2008.