KTH Computer Science
and Communication

Security by logic : characterizing Non-Interference in temporal logic

HENRI-CHARLES BLONDEEL

Master's Thesis at INRIA
Mandator at INRIA : Marieke Huisman
Supervisor at CSC : Dilian Gurov
Examiner at CSC : Johan Håstad

TRITA xxx yyyy-nn

Abstract

The information security community has still not reached consensus on how the notion of Non-Interference can be expressed for concurrent programs. Intuitively, Non-Interference expresses that the observation of public data during the execution of a program should not reveal information about private data. An attacker can infer information in many different ways, for instance by observing whether an execution terminates, or by studying the stochastic behaviour of a program.

As a step towards unifying the different proposals of Non-Interference, we have re-expressed them over a uniform program model. This program model captures the behaviour of a simple, parallel programming language with shared data. We also propose some variants and compare the main definitions. In particular, this comparison allows us to conclude that all proposed Non-Interference properties are incomparable.

A traditional way to enforce Non-Interference is by the use of a security type system. However, this method is not precise. An alternative is to use a temporal logic characterization that is amenable to model checking. We show how this can be achieved for the Non-Interference properties that do not involve probabilities.

Referat

Swedish title

Swedish translation of the abstract.

Foreword

This report describes the work done during my master's project, a final degree project, part of the Computer Science and Engineering's master program of KTH, the Royal Institute of Technology (Stockholm). It was undertaken at INRIA, the French national institute for research in computer science and control, in the EVEREST project-team (Environments for Verification and Security of Software). This master's project was under the conduction of Marieke Huisman, vice-leader of the team.

I would like to thank the EVEREST team for giving me the opportunity to conduct this work, and each of its members for the help I received and for making me feel very welcome. I especially thank Marieke Huisman for all the time she has devoted to me.

Contents

Chapter 1

Introduction

1.1 Non-Interference

With the emergence of web-based applications, preserving privacy is an increasingly important issue. Current security mechanisms, such as access control, only enforce some basic security policies and fail to guarantee confidentiality of data. Non-Interference is a confidentiality property of computer systems which regulates information flows. It was first introduced by Goguen and Meseguer in 1982 [10].

1.1.1 Intuitive definition

A confidentiality policy specifies how information is allowed to flow between the different components of a system so that confidential information do not leak from one component to another. Let us consider for instance a multi-level security (MLS) model, which assigns to each of its components a security level. This model respects the confidentiality policy if information does not flow to lower levels. On such a model, Non-Interference is a characterization of the confidentiality property. Thus a definition of Non-Interference depends on the formalism of the security model for which it is given.

Although Goguen and Meseguer's original definition of Non-Interference is given for any security policy, most of the investigations following their work have focused on MLS. More often, authors give themselves two security levels, L (low) and H (high), for public and private (or confidential) data, and mention that their work can be extended to any lattice of security levels. In the sequel we will always consider two security levels, L and H.

Borrowing from Volpano and Smith [18], Non-Interference in the sequential setting says that

> "The final values of low variables should be independent of the initial values of high variables."

For concurrent programs, the final values of variables do not depend uniquely on their initial content, but also on parallel executions and on the scheduling policy.

As a consequence, this definition needs to be generalised. Moreover, parallel composition allows parallel threads to observe the intermediate states of the memory, thus it is not sufficient to consider the final state only. However, the definition of Non-Interference for sequential programs gives a first intuitive definition in the setting of concurrency.

It can be convenient to read and compare different proposals of Non-Interference in terms of what an attacker can observe. Basically, an attacker (or a low security observer) can only observe public data. Many definitions of Non-Interference can be read as

> "an attacker cannot distinguish two executions that differ only in there private input"

However, for most proposals, the model of the attacker is not given explicitly (except in [2], where the authors define the knowledge of an attacker).

1.1.2 Comparison of existing definitions

The first goal of this work is to compare different proposals. We do this by re-expressing them over a uniform program model. Non-Interference properties from the literature differ by what an attacker can observe.

In the setting of sequential programs, the attacker is usually supposed to observe only low outputs of a program. Thus the security model could be described using a big-step semantics, which relates the initial configuration to the final one. For concurrent programs, this description of an attacker is no longer sufficient.

Many definitions of Non-Interference assume a small-step semantics, which describes the intermediate states of the system [18, 16, 11, 2]. Moreover, this kind of semantics allows parallel composition. In such a model, we often consider that what an attacker can observe is the low output of this program run in parallel with any other program which addresses only low variables.

Other models, like a CSP model, describing the system in terms of processes performing actions, are also suited for modeling parallel composition and have been used to define Non-Interference properties [14]. Actions are divided in low and high actions, and an attacker can only observe low actions. We are interested is knowing whether definitions given in terms of small-step semantics are comparable with definitions given in terms of actions.

An attacker can also learn from an execution through other mechanisms, called covert channels [15]. One example is the termination channel. Observing that a program enters an infinite loop, or that it terminates, gives additional information to the attacker. Termination-sensitive definitions of Non-Interference assume that an attacker can distinguish an execution that terminates from an execution that diverges. Some authors propose both a termination insensitive and a termination sensitive variant of their property [11].

Another covert channel is the time channel. An attacker could distinguish two executions because of different timings. In the small-step semantics, the time gran-

ularity is a transition. However, assuming that an attacker can distinguish two executions just because one does an extra transition (without visible effect) gives too much power to the attacker, thus many definitions of Non-Interference allow configurations to stutter [16, 11].

Finally, repeated runs can distinguish two configurations which have the same possible subsequent behaviours, but with different probabilities. We divide definitions of Non-Interference into possibilistic [11, 14, 2] and probabilistic definitions [18, 16].

We conclude this brief survey with compositionality. A compositional definition of Non-Interference ensures safe parallel composition of secure threads. Thus a definition which is compositional is more likely to have a practical application.

1.2 Enforcing Non-Interference

The second goal of this work is to show that we can verify Non-Interference by model checking, using a method called self-composition. Indeed, we are interested in having a static and precise method to enforce Non-Interference.

The most common approach to verify Non-Interference statically is to use a security type system [18, 16, 4]. However, this technique is not complete since it uses syntactic equality. Another approach relies on model checkers, *i.e.* tools that algorithmically verify satisfaction of a temporal logic formula for a given model. It requires a temporal logic characterization of the property we want to check. If the characterization is equivalent to the original property, then verification by model checking is sound and complete.

As we will see, Non-Interference is not a property of a single execution, *i.e.* it cannot be decided by considering all executions in isolation. However, temporal logics do not allow us to compare several executions. To overcome this difficulty, we consider an approach called self-composition [3]. It consists in executing the program in parallel with itself. Each part of the model has its own program store, so that they execute independently. Huisman *et al.* use self-composition to characterize observational determinism, a Non-Interference property [11]. We would like to do the same for different proposals, and then show the feasibility of this approach by concrete experiments with a model checker.

1.3 Contribution

As a step towards unifying the different proposals of Non-Interference, we propose a uniform model of program executions and re-express the main proposals over this model. This program model captures the behaviour of a simple, parallel programming language with shared data, defined by its operational semantics. We encode actions in configurations in order to consider definitions in CSP. The Non-Interference properties we re-express are observational determinism [19, 11], eager and lazy trace invariance [14], probabilistic Non-Interference [18], Non-Interference

based on partial probabilistic bisimulation [16], Non-Interference in terms of knowledge of an attacker [2] and abstract Non-Interference [9]. We also propose some variants. We compare the proposed definitions with a set of examples. In particular, we show that they are all incomparable.

In order to show the feasibility of verification by temporal logic, we give a modal μ-calculus characterization of observational determinism and eager invariance, based on self-composition. We prove the correctness of these characterization. We model our program semantics in CCS with the Concurrency Workbench, a model checker. We obtain convincing results for observational determinism. We do not have results for eager invariance because of the state space explosion problem.

1.4 Overview

The rest of this report is structured as follows. In Chapter 2, we formally define our uniform model of program executions. In Chapter 3, we re-express the properties cited above over this model, we present some variants and compare all the definitions. In Chapter 4, we recall the syntax and semantics of μ-calculus, next we give a μ-calculus characterization of observational determinism, prove that it is correct, and then we do the same for eager invariance. In Chapter 5, we describe the implementation of our program model with the Concurrency Workbench and comment on our results.

Part I

Comparing different definitions of Non-Interference in a common program model

Chapter 2

Program model

A program model gives an abstract description of a program. It defines abstract syntax and semantics of a programming language. Thus, the syntax describes which programs are part of the model, while the semantics specifies their behaviour.

This chapter defines a program model that will be used to (re)define the different proposals for Non-Interference over a single common model. The program language that we consider is a basic while language with parallel composition (it is an extension of the language used by Huisman *et al.* [11]). Its semantics is described in terms of a small-step operational semantics, giving rise to program traces of extended configurations.

In particular, we extend configurations with a notion of action history, that allows us to recast properties phrased over CSP processes for our program model.

The following section explains why we believe it is convenient to express properties like Non-Interference over such a model. Next, Section 2.2 gives its formal definition.

2.1 Choice of a program model

Joshi and Leino give several reasons why it is better to give a definition of Non-Interference in terms of program semantics directly, instead of relying on compiler data flow analysis techniques to ensure secure information flow [12]. First of all, they argue that this gives better precision. Moreover, the program semantics can be used to reason about indirect leaking of information by varying what an attacker can observe of the program behaviour. Finally, Joshi and Leino argue that such a definition can be extended to the case where the security levels are defined abstractly (*i.e.* not merely as a partition of the set of variables, but as functions of these variables). In the present text, we will not consider this extension, nevertheless it is important to know that it is possible to make this extension.

An alternative approach is to model programs with a process calculus such as CCS or CSP, which describe communications between independent processes in terms of actions. We would like to unify definitions given in terms of states and

definitions given in terms of actions, as the ones given by Roscoe in CSP [14]. Therefore, we extend configurations so that they keep track of the actions communicated so far with the store from the beginning of the execution.

2.2 Program model

This section defines formally our program model. All formal aspects defined below will be used in the next chapter in order to rephrase the different proposals of Non-Interference properties for multi-threaded programs that we have considered.

2.2.1 Memory

Var denotes the set of variables, $dom(v)$ the domain of a variable $v \in Var$. Each variable in *Var* is assigned a security-level value, which is either H (for high) or L (low). Var_H is the subset of high security-level variables, Var_L the subset of low security-level variables. $\{Var_H, Var_L\}$ is a partitioning of *Var*.

We assume that we have one single, global, shared memory. A store is a mapping from *Var* to values, such that each value belongs to the domain of the corresponding variable. *Store* denotes the set of stores :

$$Store = \left\{ \begin{array}{ll} \mu : & Var \rightarrow \bigcup\limits_{v \in Var} dom(v) \\ & v \mapsto x \ , \ x \in dom(v) \end{array} \right\}$$

The partial stores $Store_H$ and $Store_L$ are obtained by replacing *Var* by Var_H and Var_L in the definition of *Store*. If μ is in *Store*, then $\mu_{|H}$ denotes the restriction of μ to Var_H, and $\mu_{|L}$ its restriction to Var_L, respectively.

Definition 1 (L-equivalence on stores)**.** *Two stores μ and μ' are* L-equivalent*, denoted $\mu \approx_L \mu'$, if*

$$\mu_{|L} = {\mu'}_{|L}$$

Often, definitions of Non-Interference are defined in terms of L-equivalence.

2.2.2 Program syntax

We first define the expressions involved in the program syntax.

Expressions

A rough description of expressions is sufficient to express the most relevant security properties. Thus we do not give any grammar of expressions, but instead we assume that we have boolean and integer expressions, with their usual operators.

Exp denotes the set of expressions. If $E \in Exp$, then $E(\mu)$ is the evaluation of E where we replace each variable v by $\mu(v)$. *BExp* denotes the set of boolean expressions, $BExp \subset Exp$. *Const* denotes the set of constants, $Const \subset Exp$.

Syntax
Let *Stmt* denote the set of statements defined by the grammar below, where S is an element of *Stmt*, x is an element of *Var*, E is an element of *Exp*, and b is an element of *BExp*.

$$\begin{array}{rcl} S & ::= & x := E \mid S;S \mid \texttt{if (}b\texttt{) then } S \texttt{ else } S \mid \\ & & \texttt{while (}b\texttt{) \{ }S\texttt{ \}} \mid S \parallel_p S \mid \texttt{wait (}b\texttt{)} \mid \epsilon \end{array}$$

The label p attached to the parallel composition operator $\parallel_p$ denotes the probability that the next transition corresponds to a transition of the left statement.

2.2.3 Scheduler

A scheduling policy σ for a statement S is a mapping from the set of labels that appear on the parallel composition symbols in S to values in [0..1]. $Schedule_S$ denotes the set of all scheduling policies *w.r.t.* a statement S.

2.2.4 History of execution

Actions
We encode actions in our model in order to be able to re-express proposals of Non-Interference given in CSP (see Section 3.1.2). In our setting, we assume that an attacker can only infer information from another program execution via modifications of the store. In particular, there are no direct communications between two concurrent executions. Our actions must encompass all possible communications between a process executing a statement and the shared memory. It is only possible to read the current value of a variable or to assign a value to variable. Thus we define the set of actions, denoted by *Action*, as follows

$$Action = \{write_{v,x} \mid v \in Var\ ,\ x \in dom(v)\} \bigcup \{read_{v,x} \mid v \in Var\ ,\ x \in dom(v)\}$$

We have not defined a grammar of expressions, and in particular we do not want to specify an evaluation order for expressions in our model. Therefore, we cannot order the actions that occur during the evaluation of an expression. Instead we define a *multi-action* to be a set of actions and we require that the evaluation of an expression gives rise to exactly one multi-action. An assignment results in a multi-action containing exactly one *write* action.

Equality between multi-actions is defined simply as equality between sets of actions.

Histories
Now we can describe the history of communications as a list of multi-actions. Note that if we had given a grammar for expressions and a specific evaluation order, we could have described histories of communications by lists of actions directly. However, we do not need this precision to be able to characterize the relevant security properties.

Example 1.
The execution starting with $x := y + z$ *in store* μ *could result in two lists of actions which are*

$$read_{y,\mu(y)}, read_{z,\mu(z)}, write_{x,\mu(y)+\mu(z)}$$
$$read_{z,\mu(z)}, read_{y,\mu(y)}, write_{x,\mu(y)+\mu(z)}$$

but will only give one list of multi-actions

$$\{read_{y,\mu(y)}, read_{z,\mu(z)}\}, \{write_{x,\mu(y)+\mu(z)}\}$$

A *history* is defined as a list of non-empty multi-actions. *Hist* denotes the set of histories. The symbol $\hat{}$ denotes the concatenation of histories. We will often assimilate a multi-action into the history containing this single multi-action. Two histories are equal when they are equal element-wise.

Equivalence on histories
Let us introduce two convenient notations. If q is a list of multi-actions then $q^\bullet$ denotes the list obtained from q by removing all its empty sets, *i.e.*

$$q^\bullet = \mathit{filter}\ (\neq \{\})\ q$$

If $A \subset \mathit{Action}$, then $q_{|A}$ is the list defined by

$$q_{|A} = \mathit{map}\ (_{|A})\ q$$

where $_{|A}$ is the restriction to the elements of A.

Definition 2 (History Equivalence)**.** *Let* $\mathtt{hist}_1$ *and* $\mathtt{hist}_2$ *be two histories and* A *a set of actions. The histories* $\mathtt{hist}_1$ *and* $\mathtt{hist}_2$ *are* history equivalent *w.r.t.* A*, denoted by* $\mathtt{hist}_1 =_{|A} \mathtt{hist}_2$*, if and only if*

$$(\mathtt{hist}_{1|A})^\bullet = (\mathtt{hist}_{2|A})^\bullet$$

For any set of actions A, $=_{|A}$ is an equivalence relation.

2.2.5 Configurations

A configuration is the product of a statement, a store and a history. We denote the configuration composed of $S \in \mathit{Stmt}$, $\mu \in \mathit{Store}$ and $\mathtt{hist} \in \mathit{Hist}$ by :

$$\langle S, \mu, \mathtt{hist} \rangle$$

In a context where the history is not important, we will just write $\langle S, \mu \rangle$. The set of all configurations is :

$$\mathit{Config} = \mathit{Stmt} \times \mathit{Store} \times \mathit{Hist}$$

It is convenient to have accessor functions to extract the statement, store and history from a configuration, so we introduce *prog*, *store* and *hist* defined by

$$\mathit{prog}(\langle S, \mu, \mathtt{hist} \rangle) = S \quad \mathit{store}(\langle S, \mu, \mathtt{hist} \rangle) = \mu \quad \mathit{hist}(\langle S, \mu, \mathtt{hist} \rangle) = \mathtt{hist}$$

2.2.6 Program Semantics

To do model checking, the behaviour of a system is typically represented by a Kripke structure. We can use the small-step operational semantics to construct the transition relation of the Kripke structure directly. However, one requirement is that every state must have at least one successor. This leads us to add extra transition rules for terminated (`term`) and deadlocked (`deadl`) programs. The latter can arise due to the semantics of the *wait* statement. We define the property *deadlock* by structural induction over *Stmt*.

$$\begin{array}{lcl}
deadlock(\langle \texttt{wait (}b\texttt{)}, \mu\rangle) & = & \neg b(\mu) \\
deadlock(\langle x \texttt{ := } E, \mu\rangle) & = & false \\
deadlock(\langle \texttt{if (}b\texttt{) then } S_1 \texttt{ else } S_2, \mu\rangle) & = & false \\
deadlock(\langle \texttt{while (}b\texttt{) \{ } S \texttt{ \}}, \mu\rangle) & = & false \\
deadlock(\langle \epsilon, \mu\rangle) & = & false \\
deadlock(\langle S_1 \texttt{;} S_2, \mu\rangle) & = & deadlock(\langle S_1, \mu\rangle) \\
deadlock(\langle S_1 \parallel_p S_2, \mu\rangle) & = & deadlock(\langle S_1, \mu\rangle) \\
 & & \wedge\ deadlock(\langle S_2, \mu\rangle)
\end{array}$$

Figure 2.1 gives the semantics associated to our grammar. It uses a function *readV* which takes an expression E and a store μ as parameter and returns the multi-action that corresponds to the evaluation of E in μ.

$$readV : Exp \times Store \rightarrow \mathcal{P}(Action)$$

Given a concrete grammar for *Exp*, this function would typically look as follows :

$$\begin{array}{ll}
readV(c) = \emptyset & if c \in Const \\
readV(v, \mu) = \{read_{v,\mu(v)}\} & if v \in Var \\
readV(E_1 \ \oplus \ E_2, \mu) = readV(E_1, \mu) \cup readV(E_2, \mu) & for\ binary\ operator \oplus
\end{array}$$

As a side remark, given a particular evaluation order, one could also imagine a variation handling conditional evaluation, *i.e.* taking into account whenever one of the subexpressions at the left or at the right-hand side of an operator need not be evaluated in order to evaluate the whole expression.

Notice that the rules of Figure 2.1 also define how histories are constructed.

2.2.7 Traces of execution

We can now give the definition of program traces. Let $S \in Stmt$ and $\mu \in Store$. An infinite list $T = c_0, c_1, c_2, \ldots$ of elements in *Config* is a *program trace* of S, starting in the initial store μ, denoted $\langle S, \mu\rangle \Downarrow T$, if

- $c_0 = \langle S, \mu, \{\}\rangle$
- $\forall i \in \mathbb{N}.\ c_i \rightarrow c_{i+1}$

For any initial configuration c_0, we can always construct a trace T such that $c_0 \Downarrow T$, since the rules (`term`) and (`deadl`) of our operational semantics ensure that there is always at least one transition enabled.

$$\frac{}{\langle x := E, \mu, \texttt{hist}\rangle \rightarrow \langle \epsilon, \mu(x \rightarrow E(\mu)), \texttt{hist}\,\hat{}\,readV(E,\mu)\,\hat{}\,\{write_{x,E(\mu)}\}\rangle} \quad \texttt{(assign)}$$

$$\frac{\langle S_1, \mu, \texttt{hist}\rangle \rightarrow \langle \epsilon, \mu, \texttt{hist}'\rangle}{\langle S_1\,;S_2, \mu, \texttt{hist}\rangle \rightarrow \langle S_2, \mu, \texttt{hist}'\rangle} \quad \texttt{(seq.comp-A)}$$

$$\frac{\langle S_1, \mu, \texttt{hist}\rangle \rightarrow \langle S_1', \mu, \texttt{hist}'\rangle}{\langle S_1\,;S_2, \mu, \texttt{hist}\rangle \rightarrow \langle S_1'\,;S_2, \mu, \texttt{hist}'\rangle} \; \textit{if } S_1' \neq \epsilon \quad \texttt{(seq.comp-B)}$$

$$\frac{}{\langle \texttt{if (}b\texttt{) then } S_1 \texttt{ else } S_2, \mu, \texttt{hist}\rangle \rightarrow \langle S_1, \mu, \texttt{hist}\,\hat{}\,readV(b,\mu)\rangle} \; \textit{if } b(\mu) \quad \texttt{(if-t)}$$

$$\frac{}{\langle \texttt{if (}b\texttt{) then } S_1 \texttt{ else } S_2, \mu, \texttt{hist}\rangle \rightarrow \langle S_2, \mu, \texttt{hist}\,\hat{}\,readV(b,\mu)\rangle} \; \textit{if } \neg b(\mu) \quad \texttt{(if-f)}$$

$$\frac{}{\langle \texttt{while (}b\texttt{) \{ }S\texttt{ \}}, \mu, \texttt{hist}\rangle \rightarrow \langle S\texttt{;while (}b\texttt{) \{ }S\texttt{ \}}, \mu, \texttt{hist}\,\hat{}\,readV(b,\mu)\rangle} \; \textit{if } b(\mu) \quad \texttt{(while-t)}$$

$$\frac{}{\langle \texttt{while (}b\texttt{) \{ }S\texttt{ \}}, \mu, \texttt{hist}\rangle \rightarrow \langle \epsilon, \mu, \texttt{hist}\,\hat{}\,readV(b,\mu)\rangle} \; \textit{if } \neg b(\mu) \quad \texttt{(while-f)}$$

$$\frac{\langle S_1, \mu, \texttt{hist}\rangle \rightarrow \langle \epsilon, \mu, \texttt{hist}'\rangle}{\langle S_1 \parallel_p S_2, \mu, \texttt{hist}\rangle \rightarrow \langle S_2, \mu, \texttt{hist}'\rangle} \; \textit{with probability } \sigma(p) \quad \texttt{(par.comp-l-}\epsilon\texttt{)}$$

$$\frac{\langle S_2, \mu, \texttt{hist}\rangle \rightarrow \langle \epsilon, \mu, \texttt{hist}'\rangle}{\langle S_1 \parallel_p S_2, \mu, \texttt{hist}\rangle \rightarrow \langle S_1, \mu, \texttt{hist}'\rangle} \; \textit{with probability } 1-\sigma(p) \quad \texttt{(par.comp-r-}\epsilon\texttt{)}$$

$$\frac{\langle S_1, \mu, \texttt{hist}\rangle \rightarrow \langle S_1', \mu, \texttt{hist}'\rangle}{\langle S_1 \parallel_p S_2, \mu, \texttt{hist}\rangle \rightarrow \langle S_1' \parallel_p S_2, \mu, \texttt{hist}'\rangle} \; \textit{if } S_1' \neq \epsilon, \textit{with probability } \sigma(p) \quad \texttt{(par.comp-l)}$$

$$\frac{\langle S_2, \mu, \texttt{hist}\rangle \rightarrow \langle S_2', \mu, \texttt{hist}'\rangle}{\langle S_1 \parallel_p S_2, \mu, \texttt{hist}\rangle \rightarrow \langle S_1 \parallel_p S_2', \mu, \texttt{hist}'\rangle} \; \textit{if } S_2' \neq \epsilon, \textit{with probability } 1-\sigma(p) \quad \texttt{(par.comp-r)}$$

$$\frac{}{\langle \texttt{wait (}b\texttt{)}, \mu, \texttt{hist}\rangle \rightarrow \langle \epsilon, \mu, \texttt{hist}\,\hat{}\,readV(b,\mu)\rangle} \; \textit{if } b(\mu) \quad \texttt{(wait-t)}$$

$$\frac{}{\langle \epsilon, \mu, \texttt{hist}\rangle \rightarrow \langle \epsilon, \mu, \texttt{hist}\rangle} \quad \texttt{(term)}$$

$$\frac{}{\langle S, \mu, \texttt{hist}\rangle \rightarrow \langle S, \mu, \texttt{hist}\rangle} \; \textit{if } deadlock(\langle S, \mu, \texttt{hist}\rangle) \quad \texttt{(deadl)}$$

Figure 2.1. Operational Semantics

Trace denotes the set of all program traces. Further, $T(\mu)$ denotes the projection of trace T to its second element, *i.e.* elements in *Store*, $T(v)$ denotes the projection to a variable v of the store ($T(v)$ is called a location trace [11]), while $T(hist)$ denotes the projection to its third element, *i.e.* elements in *Hist*. Finally, T_i denotes the $(i+1)^{th}$ configuration of T, that is c_i if $T = c_0, c_1, \ldots$

Trace indistinguishability
We will also use the notion of equality up to stuttering for infinite sequences in order to define indistinguishability of traces. The definition below is from [11].

Definition 3 (Stuttering Equivalence)**.** *Two infinite sequences q and q' are* stuttering equivalent upto i and j*, denoted $[q,i] \sim_s [q',j]$, if and only if*

- $q_i = q'_j$*; and*
- *if* $k = \mathsf{min}(\{p \mid \forall n \in [p \ldots i].q_n = q_i\})$ *and*
  $l = \mathsf{min}(\{p \mid \forall n \in [p \ldots j].q'_n = q'_j\})$
  then $k = l = 0$ *or* $[q, k-1] \sim_s [q', l-1]$.

q and q' are stuttering equivalent*, denoted $q \sim_s q'$, if and only if*

- *for all i, there exists a j such that $[q,i] \sim_s [q',j]$; and*
- *for all j, there exists an i such that $[q,i] \sim_s [q',j]$.*

Stuttering equivalence ($\sim_s$) is an equivalence relation on *Trace*. It was introduced by Huisman *et al.* to define trace indistinguishability.

Definition 4 (Trace Indistinguishability)**.** *Traces T and T' are* indistinguishable *w.r.t. L (denoted $T \approx_L T'$) if*

$$\forall v \in Var_L.\, T(v) \sim_s T'(v)$$

Since $\sim_s$ is an equivalence relation, trace indistinguishability ($\approx_L$) is an equivalence relation on $Trace$.

We give a stronger variant of trace indistinguishability that does not consider each location trace separately but deals directly with the low store.

Definition 5 (Strong Trace Indistinguishability)**.** *Traces T and T' are* strongly indistinguishable *w.r.t. L, denoted $T \approx_L^{strong} T'$, if and only if*

$$T(\mu_{|L}) \sim_s T'(\mu_{|L})$$

Relation $\approx_L^{\mathtt{strong}}$ is also an equivalence relation on $Trace$.

Reachability
We also need the notion of reachable configurations *w.r.t.* a statement S, an initial store μ, and a history `hist` is :

$$reach(S, \mu, \mathtt{hist}) = \{T_i \in Config \mid \langle S, \mu\rangle \Downarrow T \wedge T(hist)_i = \mathtt{hist}\}$$

Then if Mem is a set of stores,

$$reach(S, Mem, \mathtt{hist}) = \bigcup_{\mu \in Mem} reach(S, \mu, \mathtt{hist})$$

Termination
To specify termination, we first define the predicate *term* :

$$term(T) \quad = \quad \exists i \in \mathbb{N}.\, prog(T_i) = \epsilon$$

Notice that if there is an i, such that $prog(T_i) = \epsilon$, then $prog(T_j) = \epsilon$ for all j greater than i, *i.e.* the transition rules do not allow a program to leave a terminal state.

If $term(T)$, we use T_f to denote the terminal configuration of T, *i.e.*

$$T_f = \varepsilon(\{c \in Config \mid \exists i \in \mathbb{N}.\, T_i = c \ \wedge \ prog(c) = \epsilon\})$$

where ε is the arbitrary choice operator. Since the store does not change anymore after the program has terminated, the terminal configuration of a trace in *Term* is uniquely defined.

Chapter 3

Comparison of proposals

3.1 Definitions of properties related to Non-Interference

We use our formally defined common program model to unify the main proposals of Non-Interference. Below we will detail the following properties :

- observational determinism [19, 11]
- eager and lazy trace invariance [14]
- probabilistic Non-Interference [18]
- Non-Interference based on partial probabilistic bisimulation [16]
- Non-Interference in terms of knowledge of an attacker [2]

For all properties, the following aspects will be discussed :

- completeness (in the sense that the property should not reject intuitively secure programs)
- compositionality
- whether it is a possibilistic or a probabilistic property
- assumptions about the scheduling policy (for possibilistic properties)

3.1.1 Observational determinism

First we consider observational determinism, as introduced by Zdancewic and Myers ([19], 2003). This definition has been discussed later by Huisman *et al.* ([11], 2006). Since our program model is a direct extension of the program model used by Huisman *et al.* , we can directly formulate the definition of observational determinism. We comment on interesting characteristics of this property. Next we propose two variants and compare them with the original definition.

Original definition

Observational determinism is a generalization of the classical Non-Interference property, by considering traces. Observational determinism is independent of any choice

of scheduler, thus each probability p associated to any symbol $\|_p$ that appears in any statement is implicitly quantified over all possible choices.

Observational determinism is defined as equivalence of location traces (upto stuttering) for all low-variables.

Traces are considered separately for each variable location (location traces), since according to Zdancewic and Myers, if it is the case that a program is race-free, then it is not possible to observe the relative ordering of updates [19]. A program is race-free if there is only one possible ordering of any two accesses to a same variable, one of which is a write (thus accesses that form a race do not even have to be simultaneous). Disallowing this form of races is a strong restriction of non-determinism and it seems too much a limiting factor to ask this from every application.

Definition 6 (Observational determinism)**.** *A program S is observationally deterministic if*

$$\forall \mu, \mu' \in \mathit{Store}.\, \forall T, T' \in \mathit{Trace}.$$
$$\mu \approx_L \mu' \wedge < S, \mu > \Downarrow T \wedge < S, \mu' > \Downarrow T' \;\Rightarrow\; T \approx_L T'$$

Thus, for any low-variable l, and traces T and T' of S, starting in L-equivalent initial stores, the location traces $T(l)$ and $T'(l)$ should be stuttering equivalent (see Definition 3).

Fairness

To avoid that certain harmless programs are rejected, one can require traces to be strongly fair. Indeed, let us consider Program 1.

Program 1.

$$\texttt{while}\ (\mathit{true})\ \{\epsilon\}\ \|_p\ l := 7$$

Program 1 has an execution for which the assignment never happens : the execution looping forever in the empty while loop. Let us denote the corresponding trace by T_{loop}. T_{loop} is not stuttering equivalent to any other trace of this program. This is due to the fact that, for this trace, $l := 7$ is infinitely often enabled but is never executed (thus T_{loop} is unfair). By considering only the executions that are fair, Program 1, which is intuitively secure, would not be rejected.

Non-determinism

An important aspect of observational determinism is that (citing [11] about [19]) *"different scheduling policies should not be observable in the low variables"*.

Program 2.

$$(l := \mathit{true}\ \|_p\ l := \mathit{false})\ \|_p\ l := h$$

This ensures that Program 2 is rejected, but also rejects Program 3, even though this is intuitively secure.

Program 3.

$$l := \mathit{true}\ \|_p\ l := \mathit{false}$$

This requirement seems too strong since it has no real security basis, it is just a limitation caused by the concurrency aspect. Other authors, who do not require this, have to assume that the scheduling policy is known [18, 16].

Compositionality
Observational determinism is not compositional. This means that the composition of two observationally deterministic programs does not necessarily give a new observationally deterministic program.

A common example related to compositionality is a program composed of two parallel programs, where one of them can modify the evaluation of a guard of the other. If this allows the second program to perform steps that it cannot perform when executed alone, then unexpected flows can occur. This is the case of the following example.

Example 2. *Let us consider the following program :*

Program 4.

$$l := 1; \texttt{if } (l = 0) \texttt{ then } l := h \texttt{ else } \epsilon \parallel_p l := 0$$

Each thread of Program 4 can generate only one location trace of l for a given initial value l_{init}, thus they are both observationally deterministic. The composed program can perform a direct flow (i.e. assigning the value of h to l); it is not observationally deterministic.

This example demonstrates that observational determinism is not compositional. Moreover, observational determinism is not even preserved by the intuitively secure composition of two trivially secure programs.

Example 3. *Let us consider Program 3. Its parallel threads $l := false$ and $l := true$ are trivially observationally deterministic. There are only two location traces possible for l, which are :*

$$[l_{init}, true, false, false, \ldots]$$

and

$$[l_{init}, false, true, true, \ldots]$$

We can obtain these two traces from any two L-equivalent stores, i.e. with the same value for l_{init}. They are clearly not stuttering equivalent. So Program 3, which intuitively should be secure since it does not handle any H variable, does not satisfy observational determinism.

This indicates that observational determinism is not a good generalization of Non-Interference, since it rejects too many intuitively secure programs.

Possibilistic property
Further, the following example shows that observational determinism does not ensure confidentiality when an attacker exploits probabilities.

Example 4. *Condider the following program.*

Program 5.

$$\texttt{if (h) then } (l_1 \texttt{ :=}true \parallel_p l_2 \texttt{ :=}true) \texttt{ else } (l_1 \texttt{ :=}true \parallel_{p'} l_2 \texttt{ :=}true)$$

Depending on h's value, this program becomes

$$l_1 \texttt{ :=}true \parallel_{p''} l_2 \texttt{ :=}true$$

where $p'' = p$ or $p'' = p'$. An attacker can estimate statistically the value of p'' by repeated computations (he only has to observe for each run whether l_1 is updated first, and then estimate the probability of this event). Assuming a scheduling policy σ such that $\sigma(p) \neq \sigma(p')$, he can tell whether p'' corresponds to p or p', thus he can infer which branch of the if *statement is executed, and thereby the value of h.*

However, this program is observationally deterministic (all location traces of l_1 are stuttering equivalent, the same for l_2), which means that it is accepted despite the fact that it leaks private information.

Thus observational determinism is a possibilistic property.

Variations

Using strong trace indistinguishability ($\approx_L^{\texttt{strong}}$) introduced in Definition 5, we can consider the following variation :

Definition 7 (Strong observational determinism)**.** *A program S is strongly observationally deterministic if*

$$\forall \mu, \mu' \in \mathit{Store}.\, \forall T, T' \in \mathit{Trace}.$$
$$\mu \approx_L \mu' \wedge < S, \mu > \Downarrow T \wedge < S, \mu' > \Downarrow T' \Rightarrow T \approx_L^{\mathit{strong}} T'$$

The original definition deals with all low variable separately, while this one considers them together by requiring that the projections on the low store of any two traces of S obtained from L-equivalent initial stores are stuttering equivalent. Thus it takes into account the relative ordering of changes of the store. Notice that :

$$\approx_L^{\texttt{strong}} \subset \approx_L$$

Thus strong observational determinism is stronger than observational determinism, *i.e.* it accepts less programs as being secure. Let us illustrate through a simple example that strong observational determinism is too strong for concurrent programs.

Example 5. *Consider Program 6. It is intuitively secure.*

Program 6.

$$l_1 := true \parallel_p l_2 := true$$

Program 6 is observationally deterministic but not strongly observationally deterministic.

Another variation can be obtained using equality on low stores :

Definition 8 (Non stuttering observational determinism)**.** *A program S is non-stuttering observationally deterministic if*

$$\forall \mu, \mu' \in Store. \forall T, T' \in Trace.$$
$$\mu \approx_L \mu' \wedge < S, \mu > \Downarrow T \wedge < S, \mu' > \Downarrow T' \Rightarrow T(\mu_L) = T'(\mu_L)$$

Here $T(\mu_L)$ and $T'(\mu_L)$ must be more than stuttering equivalent : they must be equal. This is the strongest definition we can give. Let us denote the relation between traces that appears in the conclusion by $=_{|Store_L}$. Next example compares $=_{|Store_L}$ and $\approx_L^{\texttt{strong}}$.

Example 6. *Let us consider the following program.*

Program 7.

$$(\texttt{if (h) then } l := false; l := false \texttt{ else } l := false) \texttt{ ; } l := true$$

Program 7 can produce four location traces for l (two for each value of l_{init}) :

$$l_{init}, \textit{ false, false, true } \ldots$$
$$l_{init}, \textit{ false, true, true } \ldots$$

Program 7 is not non-stuttering observationally equivalent, but is is strongly observationally equivalent.

In conclusion, we have :

$$=_{|Store_L} \subset \approx_L^{\texttt{strong}} \subset \approx_L$$

and the inclusions are strict (see Example 5 and 6). "S is non-stuttering observationally deterministic" implies that "S is strongly observationally deterministic", which implies that "S is observationally deterministic".

3.1.2 Definition given in the CSP process calculus

We now present eager trace invariance and lazy trace invariance, introduced by Roscoe ([14], 1995). Re-expressing these properties over our model is not straightforward, because of the large gap between our model, which is state-based, and the CSP trace model, which is action-based. Therefore we first present the original definitions in CSP notations. Next we re-express them in terms of traces.

In the author's model

Roscoe represents processes by a trace model using the CSP calculus [14], where traces are understood as the sequences of actions that a process can communicate. To avoid confusion with the term "trace" used for the common program model, here we will only talk about *sequences of actions*. A process is identified by the set of finite sequences it can communicate, denoted $Seq(P)$[1]. In this subsection we do not explicitly define what communications are; in the next subsection, actions will be instantiated as reads and writes.

Let $seq, seq' \in Seq(p)$. According to the CSP trace model, we use :

- $seq \hat{} seq'$ to denote the concatenation of seq and seq'.
- $seq \upharpoonright A$ to denote the sequence obtained from seq by eliminating the actions that are not in A.
- P/seq to denote the process P after having communicated the sequence seq. Formally this can be defined as follows :

$$Seq(P/seq) = \{seq' \mid seq \hat{} seq' \in Seq(P)\}$$

A process is non-interfering if after any pair of low-equivalent action sequences, the resulting processes still have low-equivalent behaviours. In other words, Non-Interference means determinism *w.r.t.* the low security values.

Roscoe points out that there is an ambiguity in what it means to prevent H observations : it could mean hiding H actions in process P (denoted $P \backslash H$), but it could also mean interleaving with arbitrary H actions in order to make H communications ambiguous, using the CSP interleaving operator $|||$. This operator is used to denote the composition of two processes without synchronization on any event. Roscoe proposes two properties which differ by how H actions are hidden :

Definition 9 (Trace Invariance in CSP notations)**.** *A process P is eagerly trace invariant if*

$$s, s' \in Seq(P) \wedge s \upharpoonright L = s' \upharpoonright L \Rightarrow Seq((P/s) \backslash H) = Seq((P/s') \backslash H)$$

and is lazily trace invariant if

$$s, s' \in Seq(P) \wedge s \upharpoonright L = s' \upharpoonright L \Rightarrow Seq((P/s) \,|||\, Run_H) = Seq((P/s') \,|||\, Run_H)$$

where Run_H is a process that always can do any H action. Note that the equality condition in the conclusion is an equality between the sets of possible behaviours of the two processes.

Notice that Roscoe's lazy trace invariance is stronger than eager trace invariance. This can be seen as follows (proof from Roscoe [14]).

[1] Whereas Roscoe writes $Traces(P)$.

Proof. Assume $seq, seq' \in Seq(P) \wedge seq =_L seq'$. Assume that after having produced a sequence of actions seq, the process P can produce a sequence seq_{subseq} when hiding H actions, *i.e.* $seq_{subseq} \in Seq((P/seq)\backslash H)$. Then restoring the hidden H actions in seq_{subseq} gives us a sequence of $Seq(P/seq)$, let us call it seq_{rest}. Then as $Seq(P/seq) \subset Seq(P/seq|||Run_H)$, we have $seq_{rest} \in Seq(P/seq|||Run_H)$ and lazy trace invariance implies that there is the same sequence in $Seq(P/seq'|||Run_H)$. Using $Seq((P/seq'|||Run_H)\backslash H) = Seq((P/seq')\backslash H)$, we get :

$$seq_{subseq} \in Seq((P/seq')\backslash H)$$

which concludes the proof. □

In our model

Once again the definitions implicitly quantify over all possible scheduling policies.

Roscoe's definitions of trace invariance [14] use the notion of behaviour equality, *i.e.* equality between sets of possible action sequences. Our purpose is to specify this notion for program traces T.

Instead of using arbitrary action sequences, we instantiate them as the histories we defined in Section 2.2. Histories are sequences of multi-actions; multi-actions are non-empty sets of actions, reads and writes, for which our model is not precise enough to give an ordering. The fact that some actions are not ordered affects only the granularity of the trace invariance properties. Notice that having only read and write actions limits the environment of programs to the store. This corresponds to the fact that in our model, communication happens via the shared memory (while in Roscoe's CSP view, processes communicate directly).

For a more complex language and memory, we could consider also other actions. For instance, programs maintain specific information for each procedure call, often called activation records. We could consider each entry and exit sequence as an action, since they provide additional information about the behaviour of the program.

The rules of the operational semantics (Figure 2.1) ensure that if $\langle S, \mu\rangle \Downarrow T$, the sequence of actions that program S has communicated after i steps corresponds to the history $T(hist)_i$.

Eager trace invariance

We use `L-Actions` to denote the set of actions involving L variables.

Definition 10 (Eager trace invariance)**.** *A statement S is eagerly trace invariant w.r.t. L if*

$$\begin{aligned}
&\forall\, \mathtt{hist}, \mathtt{hist}' \in Hist.\, \mathtt{hist} =_{|\mathtt{L\text{-}Actions}} \mathtt{hist}' \Rightarrow \\
&\quad \forall c \in reach(S, Store, \mathtt{hist}).\, \forall T \in Trace.\, c \Downarrow T \;\Rightarrow \\
&\qquad \exists c' \in reach(S, Store, \mathtt{hist}').\, \exists T' \in Trace. \\
&\qquad c' \Downarrow T' \;\wedge\; (\forall m \in \mathbb{N}.\, \exists n \in \mathbb{N}.\, T(hist)_m =_{|\mathtt{L\text{-}Actions}} T'(hist)_n)
\end{aligned}$$

Thus if c and c' are intermediate configurations of two executions starting with S, and if these two executions have communicated the same low actions when they reach c and c' respectively, then for all executions starting with c (T in the formula), there is an execution (T') starting in c' that can communicate the same low actions.

Note that the set $reach(S, Store, \texttt{hist})$ represents the subsequent behaviour of all executions of S which have communicated `hist` so far, and that it does not depend on a particular initial configuration. That is why we consider the set of configurations reachable from *Store*.

For a similar reason, it is not possible to find a one-step recursive definition equivalent to eager trace invariance. This would require that for any two configurations c_1 and c_2 that can communicate low-equivalent multi-actions by taking the steps $c_1 \rightarrow c_2$ and $c'_1 \rightarrow c'_2$, the resulting configurations c'_1 and c'_2 should also be able to communicate the same multi-actions, which has no reason to be.

Lazy trace invariance

Lazy trace invariance is more difficult to express. Indeed, which statement has the behaviour of $(P/seq)|||Run_H$ (in CSP notation)? To model this, we introduce a statement whose execution has the effect to assign an arbitrary value to all H variables. We denote this statement as HH ("havoc on H"), following Joshi and Leino [12]. How this statement is built from the constructions in our programming language does not really matter (but it can be done using parallel composition, the assignment operator and a random value generator for the right-hand expression of the assignments), we just assume that it satisfies the following specification :

$$\forall \mu, \mu' \in Store.\ \mu_{|L} = {\mu'}_{|L} \ \Rightarrow\ \langle HH, \mu\rangle \rightarrow^{+} \langle \epsilon, \mu'\rangle$$

Definition 11 (Lazy trace invariance)**.** *A statement S is lazily trace invariant w.r.t. L if*

$$\forall\, \texttt{hist}, \texttt{hist}' \in Hist.\ \texttt{hist} =_{|L\text{-}Actions}\ \texttt{hist}' \Rightarrow$$
$$\quad \forall c \in reach(S, Store, \texttt{hist}).\ \forall T \in Trace.\ \langle prog(c) \parallel_p HH, store(c)\rangle \Downarrow T \ \Rightarrow$$
$$\quad\quad \exists c' \in reach(S, Store, \texttt{hist}').\ \exists T' \in Trace.$$
$$\quad\quad \langle (prog(c')) \parallel_p HH, store(c')\rangle \Downarrow T' \ \wedge\ (\forall m \in \mathbb{N}.\ \exists n \in \mathbb{N}.\ T(hist)_m = T'(hist)_n)$$

Thus if c and c' are intermediate configurations of two executions starting with S, and if these two executions have communicated the same low actions when they reach c and c' respectively, then for all executions starting with c, with interleaving of arbitrary assignments on H variables (T in the formula), there is an execution (T') starting in c', with interleaving of H assignments too, that can communicate the same actions.

Contrary to eager trace invariance, the equality of histories in the conclusion is considered *w.r.t.* the whole set of actions *Action*. Since H actions have become ambiguous, there is no need to hide them anymore.

Comparison between eager and lazy trace invariance
Lazy trace invariance will explore paths that are theoretically unaccessible for a given statement. Consider for instance the following program :

Program 8.

$$h := 0; \texttt{if } (h = 1) \texttt{ then } l := h' \texttt{ else } \epsilon$$

When run in a non-concurrent context, Program 8 will never reach a configuration c such that

$$prog(c) = l := h'$$

Program 8 is eagerly trace invariant but not lazily trace invariant. By allowing changes to H variables, the statement $prog(c) \parallel_p HH$ might reach the unsecure assignment nested in the `if` construct, and the program will be rejected by the lazy trace invariance property. This can be considered as taking into account arbitrary high-modifications done by the parallel execution of other programs.

Compositionality
Further, both eager and lazy trace invariance are not compositional since nothing guarantees that these properties still hold after an arbitrary modification of the low store. We can illustrate this with Program 4. Both threads of this program are eagerly and lazily trace invariant, but the composition of them is neither eagerly nor lazily trace invariant. This can be shown by giving two traces for which the explicit flow is reached with different values of h' and proving that the configurations reachable from the corresponding histories are not equivalent.

Possibilistic property
Eager and lazy trace invariance do not reject Program 5 (on page 18), since this program can do the same actions, whatever the value of h is. We have already seen that this program is unsecure if the attacker can exploit probabilities. Thus eager trace invariance and lazy trace invariance are both possibilistic properties.

3.1.3 Probabilistic Non-Interference

Volpano and Smith's formulation of Non-Interference is described directly in terms of the operational semantics ([18], 1998), so, taking a few restrictions into account, their definition can be reformulated directly in terms of our model. They use a stochastic matrix, which is only defined for programs with a finite state-space. In particular, this implies that the set of initial stores must be finite. However, later we will show how we can manipulate the definition to lift this restriction.

Original definition

The definitions below are given for a statement S, a scheduling policy σ and a set of possible initial stores $Store_{init}$. Where possible, we will leave these implicit.

Once again, we need to consider the set of reachable configurations. However, here we are just interested in the set of configurations reachable from a statement S and a set of possible initial stores $Store_{init}$, denoted $reach(S, Store_{init})$. Formally, this is defined as the union of the states reachable for all possible histories. That is, according to our conventions

$$reach(S, Store_{init}) = \bigcup_{\mathtt{hist} \in Hist} reach(S, Store_{init}, \mathtt{hist})$$

If $reach(S, Store_{init})$ is infinite, we cannot define the stochastic matrix, thus Volpano and Smith's definition of Non-Interference cannot be used directly. Therefore, for now, let us assume that $reach(S, Store_{init})$ is finite (notice that this implies that $Store_{init}$ is finite too, since $Store_{init} \subset reach(S, Store_{init})$). Let n be the cardinal of the set $reach(S, Store_{init})$ and let $c_0, c_1, c_2, \ldots, c_{n-1}$ denote its elements.

Definition 12 (Stochastic Matrix)**.** *$Stoch_{S,\sigma}^{Store_{init}}$ is the $n \times n$ matrix defined by*

$$Stoch_{S,\sigma}^{Store_{init}}(i, j) = p_\sigma(c_i \to c_j)$$

where $p_\sigma(c_i \to c_j)$ means the conditional probability that "$c_i \to c_j$ occurs", given that "the current configuration is c_i", w.r.t. the scheduling policy σ.

We define the following rules to compute $p_\sigma(c_i \to c_j)$, recursively on the structure of $prog(c_i)$:

if $c_i = \langle S_1 \parallel_p S_2, \mu\rangle, c_j = \langle S_1' \parallel_p S_2, \mu'\rangle$ *and* $S_1 \neq S_1'$
$p_\sigma(c_i \to c_j) = \sigma(p) \times p_\sigma(\langle S_1, \mu\rangle \to \langle S_1', \mu'\rangle)$

if $c_i = \langle S_1 \parallel_p S_2, \mu\rangle, c_j = \langle S_1 \parallel_p S_2', \mu'\rangle$ *and* $S_2 \neq S_2'$
$p_\sigma(c_i \to c_j) = (1 - \sigma(p)) \times p_\sigma(\langle S_2, \mu\rangle \to \langle S_2', \mu'\rangle)$

if $c_i = \langle S_1 \parallel_p S_2, \mu\rangle, c_j = \langle S_2, \mu'\rangle$ *and* $S_1 \neq S_2$
$p_\sigma(c_i \to c_j) = \sigma(p) \times p_\sigma(\langle S_1, \mu\rangle \to \langle \epsilon, \mu'\rangle)$

if $c_i = \langle S_1 \parallel_p S_2, \mu\rangle, c_j = \langle S_1, \mu'\rangle$ *and* $S_1 \neq S_2$
$p_\sigma(c_i \to c_j) = (1 - \sigma(p)) \times p_\sigma(\langle S_2, \mu\rangle \to \langle \epsilon, \mu'\rangle)$

if $c_i = \langle S \parallel_p S, \mu\rangle,$ *and* $c_j = \langle S, \mu'\rangle$
$p_\sigma(c_i \to c_j) = p_\sigma(\langle S, \mu\rangle \to \langle \epsilon, \mu'\rangle)$

if $c_i = \langle S_1 \parallel_p S_2, \mu\rangle$
$p_\sigma(c_i \to c_i) = 1 - \sum_{i \neq j} p_\sigma(c_i \to c_j)$

otherwise there is an unique configuration c such that $c_i \to c$, thus
$p_\sigma(c_i \to c_j) = 1$ *if* $c_j = c$
$p_\sigma(c_i \to c_j) = 0$ *if* $c_j \neq c$

A distribution u is a row vector of size n such that $u[i]$ represents $p_\sigma(c_i)$, the probability of being in configuration c_i. We use *Dist* to denote the set of distributions.

$Dist_{init}$ is the set of distributions for which each configuration with a non-null probability has program S and a store in $Store_{init}$. This can be defined formally as follows :

$$Dist_{init}(S, Store_{init}) = \left\{ \begin{array}{l} u \in Dist \mid \forall i \in [0..n-1]. \\ \quad u[i] \neq 0 \Rightarrow prog(c_i) = S \wedge store(c_i) \in Store_{init} \end{array} \right\}$$

Volpano and Smith define equivalence of distributions as follows : "distributions u and u' are equivalent if and only if they are equal after high variables are projected out" [18]. This can be defined as a generalization of store indistinguishability in our model.

Definition 13 (L-Equivalence on distributions)**.** *Two distributions u and u' are L-equivalent, denoted $u \sim_L u'$ if :*

$$\forall s \in Store, \sum_{i \in [0..n].store(c_i) \approx_L s} u[i] = \sum_{i \in [0..n].store(c_i) \approx_L s} u'[i]$$

We can now give the definition of probabilistic Non-Interference:

Definition 14 (Probabilistic Non-Interference)**.** *A statement S and a set of initial stores $Store_{init}$, such that $reach(S, Store_{init})$ is finite, is probabilisticly non-interfering w.r.t. the security level L if*

$$\forall u, u' \in Dist.\ u \sim_L u' \Rightarrow u \times Stoch_{S,\sigma}^{Store_{init}} \sim_L u' \times Stoch_{S,\sigma}^{Store_{init}}$$

By construction of $Stoch_{S,\sigma}^{Store_{init}}$, distribution $u \times Stoch_{S,\sigma}^{Store_{init}}$ is the distribution obtained from distribution u after one step. Thus this definition says that transitions should preserve the L-equivalence relation on $Dist$.

Completeness

Notice that Definition 14 of Non-Interference ranges over all possible distributions. This makes one wonder if this definition is not too strong. Indeed, it rejects intuitively secure programs such as Program 3, as shown in Example 7 below.

Example 7.

$$l := true \parallel_{\frac{1}{2}} l := false$$

Let

$$Store_{init} = \{[l \mapsto false]\}$$

be the set (here a singleton) of initial stores. The accessible configurations are :

$$\begin{array}{lll} \langle l := true \parallel_{\frac{1}{2}} l := false\ , [l \mapsto false] \rangle & (c_0) \\ \langle l := true\ , [l \mapsto false] \rangle & (c_1) \\ \langle l := false\ , [l \mapsto true] \rangle & (c_2) \\ \langle \epsilon\ , [l \mapsto false] \rangle & (c_3) \\ \langle \epsilon\ , [l \mapsto true] \rangle & (c_4) \end{array}$$

The stochastic matrix associated to the initial configuration c_0 is :

$$Stoch_{c_0} = \begin{pmatrix} 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

There are only three possible distributions reachable from c_0, which are :

$$\begin{array}{lcccccll} u_0 & = & (1 & 0 & 0 & 0 & 0) & \textit{(initial distribution)} \\ u_1 & = & (0 & \frac{1}{2} & \frac{1}{2} & 0 & 0) & \textit{(in one step)} \\ u_2 & = & (0 & 0 & 0 & \frac{1}{2} & \frac{1}{2}) & \textit{(in two steps)} \end{array}$$

As c_0 is our only initial configuration, u_0 is the initial distribution. Definition 14 requires that $u \times Stoch_{S,\sigma}^{Store_{init}} \sim_L u' \times Stoch_{S,\sigma}^{Store_{init}}$ holds for every two L-equivalent distributions, including those that are unreachable. Now, let us consider the following unreachable distributions

$$\begin{array}{lcccccl} u & = & (0 & \frac{1}{2} & 0 & 0 & \frac{1}{2}) \\ u' & = & (0 & 0 & \frac{1}{2} & \frac{1}{2} & 0) \end{array}$$

According to the definition of L-equivalence for distributions, $u \sim_L u'$ (the probability that the store is $[l \mapsto true]$ is the same for u and u'). But

$$\begin{array}{lcccccl} u \times Stoch_{c_0} & = & (0 & 0 & 0 & 0 & 1) \\ u' \times Stoch_{c_0} & = & (0 & 0 & 0 & 1 & 0) \end{array}$$

and those two last distributions are not L-equivalent (the store is $[l \mapsto true]$ in $u \times Stoch_{c_0}$ and $[l \mapsto false]$ in $u' \times Stoch_{c_0}$), so the probabilistic Non-Interference property does not hold, even though this program does not handle any high variables.

Compositionality

Notice that Example 7 also makes it clear that probabilistic Non-Interference is not compositional.

Variant

To avoid rejecting completely innocent programs, we propose an alternative definition which does not take into account unreachable distributions. We know that the set of reachable distributions is :

$$\{u \times (Stoch_{S,\sigma}^{Store_{init}})^i \mid u \in Dist_{init}\ ,\ i \in \mathbb{N}\}$$

This enables us to give a new definition directly:

Definition 15 (Corrected probabilistic Non-Interference)**.** *A statement S and a set of initial stores $Store_{init}$, such that $reach(S, Store_{init})$ is finite, have the (corrected) probabilistic Non-Interference property if*

$$\begin{array}{l}\forall u, u' \in Dist_{init}.\ u \sim_L u' \Rightarrow \\ \forall i \in \mathbb{N}.\ u \times (Stoch_{S,\sigma}^{Store_{init}})^i \sim_L u' \times (Stoch_{S,\sigma}^{Store_{init}})^i\end{array}$$

Thus any pair of distributions reachable from a pair of L-equivalent distributions after i steps should be L-equivalent too.

The scenario described in the last example cannot be reproduced with this definition, since $Dist_{init} = \{u_0\}$ and the two distributions, $u \times Stoch_{c_0}$ and $u' \times Stoch_{c_0}$, are not equal to $u_0 \times (Stoch_{c_0})^i$ for any i.

Probabilistic property

Program 5 and Program 9 are correctly rejected by both the original and the corrected probabilistic Non-Interference property :

Program 9.

$$(\texttt{if (h) then } l := true \texttt{ else } \epsilon) \parallel_p (l := true \parallel_p l := false)$$

Notice that observational determinism and the trace invariance properties discussed above also reject Program 9. In contrast, only a probabilistic analysis rejects Program 5, the possibilistic definitions do not reject it.

Formulation with traces

For each $i \in \mathbb{N}$, we consider T_i to be a random variable. Its distribution is denoted by $dist(T_i)$. In order to link the stochastic matrix to the trace model, notice that

$$dist(T_0) = u \quad \Rightarrow \quad p_\sigma(T_i = c_j) = (u \times (Stoch_{S,\sigma}^{Store_{init}})^i)[j]$$

Thus, if a trace T has an initial configuration whose distribution is u, the i^{th} element of T is equal to c_j with probability $(u \times (Stoch_{S,\sigma}^{Store_{init}})^i)[j]$. We rewrite this as follows :

$$p_\sigma(T_i = c_j \mid dist(T_0) = u) = (u \times (Stoch_{S,\sigma}^{Store_{init}})^i)[j]$$

By definition of L-equivalence on distributions, the conclusion of Definition 15 is equivalent to :

$$\begin{array}{l}\forall i \in \mathbb{N}.\ \forall s \in Store. \\ \quad \sum_{j \in [0..n-1].\ store(c_j) \approx_L s} (u \times (Stoch_{S,\sigma}^{Store_{init}})^i)[j] \ = \\ \quad \sum_{j \in [0..n-1].\ store(c_j) \approx_L s} (u' \times (Stoch_{S,\sigma}^{Store_{init}})^i)[j]\end{array}$$

Thus it can be reformulated as follows :

$$\begin{array}{l}\forall i \in \mathbb{N}.\, \forall s \in \mathit{Store}. \\ \quad \sum\limits_{j \in [0..n-1].\ \mathit{store}(c_j) \approx_L s} p_\sigma(T_i = c_j | \mathit{dist}(T_0) = u)) \ = \\ \quad \sum\limits_{j \in [0..n-1].\ \mathit{store}(c_j) \approx_L s} p_\sigma(T_i = c_j | \mathit{dist}(T_0) = u'))\end{array}$$

As the definition we obtain does not involve the stochastic matrix, we can extend it to the more general case where the set of initial stores and the set of reachable configurations can be infinite :

Definition 16 (Extended probabilistic Non-Interference)**.** *A statement S has the (corrected, extended) probabilistic Non-Interference property if*

$$\begin{array}{l}\forall u, u' \in \mathit{Dist}.\ u \sim_L u' \Rightarrow \\ \quad \forall i \in \mathbb{N}.\, \forall s \in \mathit{Store}. \\ \quad\quad \sum\limits_{c \in \mathit{Config}.\ \mathit{store}(c) \approx_L s} p_\sigma(T_i = c | \mathit{dist}(T_0) = u)) \ = \\ \quad\quad \sum\limits_{c \in \mathit{Config}.\ \mathit{store}(c) \approx_L s} p_\sigma(T_i = c | \mathit{dist}(T_0) = u'))\end{array}$$

This definition says that, for any pair of L-equivalent distributions (u, u'), for any store s, the probability to reach, after a given number of steps i, a configuration whose store is L-equivalent to s, is the same when starting from distribution u or from u'.

We can show that Program 4 (see Subsection 3.1.1) does not satisfy Definition 16, whereas its two threads do, thus compositionality remains lost.

3.1.4 Non-Interference based on partial probabilistic bisimulation

Sabelfeld and Sands use the notion of partial probabilistic low-bisimulation to characterize Non-Interference ([16], 2000). They use symmetric and transitive relations, *i.e.* partial equivalence relations. They propose an alternative definition directly in terms of small-step operational semantics, that we reformulate in our notation. We will also propose a variant that accepts more programs than the original definition.

Original property

Traditionally, a bisimulation exists if there is an equivalence relation R on configurations for which, whenever two configurations are related by R, if one of them can reach a configuration c in one step, the other must be able to reach a configuration c' in one step such that cRc'.

Sabelfeld and Sands extend the notion of bisimulation to probabilistic bisimulation, by adding the requirement that, if cRd and A is a R-equivalence class, c and d must have the same probability to reach A in one step.

Then, the authors weaken this definition of probabilistic bisimulation to partial equivalence relations (per). They do this by exploiting the following observation : if R is a per, and $\mathit{dom}(R)$ denotes the set of configurations on which R is reflective,

then R is an equivalence relation on $dom(R)$. Further, they require that if a configuration participates in R, all configurations reachable from this configuration have to be in $dom(R)$. In this way, they obtain a partial probabilistic bisimulation. Thus a partial probabilistic bisimulation for which $dom(R) = Config$ is a probabilistic bisimulation.

Definition 17 (Partial probabilistic bisimulation on configurations)**.** *A per $R \subseteq Config \times Config$ is a partial probabilistic bisimulation on configurations if for all R-equivalence classes A, and for all c, $d \in Config$, whenever cRd then*

- $\forall A \in Config/R.\ c \rightarrow_p A \Rightarrow d \rightarrow_p A$
- $c \rightarrow_0 Config \backslash dom(R)$

where $c \rightarrow_p A$ is the equivalent of $\sum_{c' \in A} p_\sigma(c \rightarrow c') = p$ in our model (i.e. the probability to reach a configuration in A from c is p).

Sabelfeld and Sands use the notion of partial probabilistic low-bisimulation on statements to give a characterization of probabilistic Non-Interference.

Definition 18 (Partial probabilistic low bisimulation on statements)**.** *R is a partial probabilistic low-bisimulation on statements if $R\times \approx_L$ is a partial probabilistic bisimulation on configurations.*

It is clear that any two configurations related by $R\times \approx_L$ have L-equivalent stores.

Definition 19 (Non-Interference based on low bisimulation)**.** *A program S is secure w.r.t. L if there exists a partial probabilistic low-bisimulation on statements $\sim_L$ such that $S \sim_L S$.*

We can rephrase the authors' definition of a partial probabilistic low-bisimulation, expressed directly in terms of the operational semantics, using our notations, as follows :

Theorem 1. *A per R is a partial probabilistic low-bisimulation on Stmt iff*

$$
\begin{array}{l}
\forall S_1, S_2 \in Stmt. \forall \mu_1, \mu_2 \in Store. \\
(S_1 R S_2 \ \wedge\ \mu_1 \approx_L \mu_2 \ \wedge\ \langle S_1, \mu_1\rangle \rightarrow \langle S_1', \mu_1'\rangle) \ \Rightarrow \\
\quad \exists S_2' \in Stmt, \mu_2' \in Store. \\
\qquad \langle S_2, \mu_2\rangle \rightarrow \langle S_2', \mu_2'\rangle \wedge \\
\qquad S_1' R S_2' \wedge \\
\qquad \mu_1' \approx_L \mu_2' \wedge \\
\qquad \sum_{S \in [S1']_R,\ \mu \approx_L \mu_1'} p_\sigma(\langle S_1, \mu_1\rangle \rightarrow \langle S, \mu\rangle) = \\
\qquad \sum_{S \in [S2']_R, \mu \approx_L \mu_2'} p_\sigma(\langle S_2, \mu_2\rangle \rightarrow \langle S, \mu\rangle)
\end{array}
$$

If two configurations have L-equivalent stores and programs related by R, then if after one step the first can reach a configuration c, the second can reach in one step a configuration whose store is L-equivalent to $store(c)$ and whose program is related by R to $prog(c)$. Moreover, the probability to reach a configuration whose store is L-equivalent to $store(c)$ and whose program is related by R to $prog(c)$ is the same for the first and the second configuration.

Probabilistic property

As was already the case for Definition 14, Definition 19 prevents a low-attacker (able to observe only the value of low-variables) to derive information about the value of high-variables from repeated computations, *i.e.* by exploiting probabilities. For instance, it rejects Program 5. One advantage of this definition over the original definition of Volpano and Smith is that it does not require that the set of initial stores is finite.

Compositionality

Concerning compositionality, we can observe that whenever we want to prove that $S_1 R S_2$, where S_1 and S_2 result from the execution of S, we have to consider all pairs of configurations $\langle S_1, \mu_1 \rangle$ and $\langle S_2, \mu_2 \rangle$ where $\mu_1 \approx_L \mu_2$, and not only the configurations where the store results from the execution of S. This ensures that this definition is compositional. This also implies that this definition rejects programs that might be unsecure if a concurrent modification of $Store_L$ happens, even if they are harmless when run alone. This is for example the case for Program 10, as shown in Example 8.

Example 8. *Consider Program 10. It is intuitively secure when run in isolation.*

Program 10. l `:=`0`; if (`$l = 1$`) then` l `:=`h `else` ϵ

Proof Sketch : Definition 18 rejects Program 10. Proof by contradiction. Let

$$\begin{array}{lcl} S & = & l := 0; S_1 \\ S_1 & = & \texttt{if } (l = 1) \texttt{ then } l := h \texttt{ else } \epsilon \end{array}$$

Assume there exists an R such that SRS. Then we must have $S_1 R S_1$. As for any $\mu \in Store$ such that $\mu(l) = 1$,

$$\langle S_1, \mu \rangle \rightarrow \langle l := h, \mu \rangle$$

we should also have $(l := h) R (l := h)$, which is false (this is shown easily by choosing two stores with the same value for l, and different values for h). □

Variation

The last example leads us to give a variation where only reachable configurations have to belong to pairs contained in the relation. Thus we loose compositionality, but programs such as Program 10 are accepted.

This definition does not use a partial probabilistic low-bisimulation (on statements), but only partial probabilistic bisimulation on configurations as defined by Definition 17.

Definition 20 (Non-Interference based on bisimulation on configurations)**.** *A statement S is secure w.r.t. L, denoted $S \sim^c_L S$, if there exists a partial probabilistic bisimulation R_c on Config such that*

$$R_c \Rightarrow \approx_L \ \wedge$$
$$\forall \mu, \mu' \in Store.\ \mu \approx_L \mu' \ \Rightarrow \ \langle S, \mu \rangle R_c \langle S, \mu' \rangle$$

With this definition, Program 10, in isolation, is not rejected. However it is rejected when composed with program $l := 1$. Thus this definition is not compositional.

3.1.5 Knowledge of an attacker

Askarov and Sabelfeld introduce the notion of *knowledge* of an attacker ([2], 2007), and use it to characterize Non-Interference. Their definition does not consider infinite executions, thus it is defined only for terminating programs. We extend our model in order to re-express this definition, and then we consider possible definitions that take into account non-termination.

The notion of knowledge

The attacker is assumed to observe the projection of the store on $Store_L$ for the initial configuration and after each step. He can also observe the termination of an execution. The knowledge of an attacker is expressed as the set of stores that are possible initial stores *w.r.t.* what he has observed so far of an execution. With each new step, this set can only shrink. Notice what an attacker actually knows is that the stores that are not in this set cannot produce what he has observed, thus his knowledge about the initial store increases as the the set called "knowledge" shrinks.

The authors give a formal definition of Non-Interference using this notion. In the following, we rephrase what interests us with program traces and our conventions, adding the intermediate notion of trace-knowledge.

Definition 21. *The trace-knowledge (TK) associated to statement S, initial L store $\mu_L^{init} \in Store_L$ and sequence seq of L stores is*

$$TK(S, \mu_L^{init}, seq) = \left\{ \begin{array}{ll} T_0 T_1 \dots T_i \mid & \forall j.\, 0 \leq j \leq i-1 \ \Rightarrow \ T_j \rightarrow T_{j+1} \wedge \\ & store(T_0)_{|L} = \mu_L^{init} \wedge \\ & store(T_1)_{|L} \dots store(T_i)_{|L} = seq \end{array} \right\}$$

This set contains all (finite) traces that correspond to low observations. Until now, we have used T to denote an infinite sequence of successive configurations (an element of *Trace*). In the sequel of this section, we abuse the notation and allow T to denote finite traces.

The knowledge corresponding to a particular trace-knowledge is the set of (full) stores for all initial configurations.

Definition 22. *The knowledge associated to statement S, initial L store $\mu_L^{init} \in Store_L$ and sequence seq of L stores is*

$$K(S, \mu_L^{init}, seq) = \{\mu \mid \exists T \in TK(S, \mu_L^{init}, seq) \ \wedge \ \mu = store(T_0)\}$$

The sequence *seq* denotes what the attacker has observed during the execution. It is augmented with one element after each step. The initial knowledge is $K(S, \mu_L, \emptyset)$.

Let us see a simple example from Askarov and Sabelfeld.

Example 9.
All variables are natural numbers. Consider the following assignment :

Program 11.

$$l := h_1 + h_2$$

For all $\mu_L^{init} \in Store_L$, the knowledge before execution is

$$K(S, \mu_L^{init}, \emptyset) = \{\mu \in Store \mid \mu_{|L} = \mu_L^{init}\}$$

After the assignment, if for instance the L store is $[l \mapsto 3]$, then the knowledge is

$$K(S, \mu_L^{init}, [l \mapsto 3]) = \{\mu \in Store \mid \mu_{|L} = \mu_L^{init} \ \wedge \ \mu(h_1) + \mu(h_2) = 3\}$$

Trace-knowledge in our model

We extend configurations with trace-knowledge, and show how to update the rules of our operational semantics to update the trace-knowledge. In the sequel, $\langle S, \mu, TK\rangle$ denotes the configuration $\langle S, \mu\rangle$ with trace-knowledge TK. We define the accessor function *knowl*, that extracts the trace-knowledge from a configuration and returns the corresponding knowledge.

$$knowl(\langle S_1, \mu_1, TK(S, \mu_L^{init}, seq)\rangle) \quad = \quad K(S, \mu_L^{init}, seq)$$

The initial trace-knowledge associated to configuration $\langle S, \mu\rangle$ is

$$TK(S, \mu_{|L}, \emptyset) = \left\{\langle S, \mu'\rangle \mid \mu'_{|L} = \mu_{|L}\right\}$$

Thus, given a program S and a store μ, all executions start with the following extended configuration :

$$\langle S, \mu, \left\{\langle S, \mu'\rangle \mid \mu'_{|L} = \mu_{|L}\right\}\rangle$$

To simplify the notation, we will write this last configuration $\langle S, \mu, TK_0\rangle$.

The rule to update the trace-knowledge is given by

$$\frac{\langle S, \mu\rangle \rightarrow \langle S', \mu'\rangle}{\langle S, \mu, TK\rangle \rightarrow \langle S', \mu', TK'\rangle}$$

where

$$TK' = \left\{ T \,.\, \langle S, \mu'' \rangle \mid \mu'' \approx_L \mu' \wedge T_{|T|-1} \rightarrow \langle S, \mu \rangle \right\}$$

This construction has the following invariant : if $c \Downarrow T$ and $T_i = \langle S, \mu, TK \rangle$, then

$$\forall T' \in TK.\, |T'| = i + 1$$

Knowledge and Non-Interference

We would expect non-interfering statements to be the statements for which $K(S, \mu_L^{init}, \emptyset)$ is a subset of $K(S, \mu_L^{init}, seq)$ for any sequence seq that can perform S (*i.e.* the attacker can learn nothing new from low observations). Indeed, if there is a store μ' which belongs to $K(S, \mu_L^{init}, \emptyset)$ (thus $\mu'_{|L} = \mu_L^{init}$), but does not belong to $K(S, \mu_L^{init}, seq)$, this means that the observation of seq has permitted to infer that the initial high store was not $\mu'_{|H}$.

By adding the requirement that, if two executions start in L equivalent stores, they both terminate or they both diverge, we would obtain a termination-sensitive definition of Non-Interference. However, Askarov and Sabelfeld propose a termination-insensitive definition and consider only initial configurations that lead to termination. Therefore, they introduce the notion of initial knowledge about termination. In our notation :

Definition 23. *The initial knowledge about termination w.r.t. statement S and initial L store μ_L^{init} is*

$$K_{term}(S, \mu_L^{init}) = \left\{ \mu \in Store \mid \begin{array}{l} \mu_{|L} = \mu_L^{init} \wedge \\ \exists T \in Term.\, \langle S, \mu \rangle \Downarrow T \end{array} \right\}$$

Thus $K_{term}(S, \mu_L^{init})$ contains all possible initial stores from which S can terminate.

Now we can give the authors' definition of Non-Interference in the knowledge-based setting using our notation :

Definition 24. *A statement S is non-interfering with respect to L if*

$$\begin{array}{c} \forall \mu \in Store.\, \forall T \in Term.\, \langle S, \mu, TK_0 \rangle \Downarrow T \Rightarrow \\ \forall i \in \mathbb{N}.\, K_{term}(S, \mu_{|L}) \subset knowl(T_i) \end{array}$$

Thus suppose that T is a terminating trace, then in each configuration of T the knowledge about the initial store should never be more precise than the knowledge that the trace will terminate.

One advantage of this definition is that it can be extended to define gradual release, by allowing the knowledge to shrink at release events. Gradual release is

a policy introduced by Askarov and Sabelfeld in order to unify declassification, encryption and key release policies. Indeed, Non-Interference is too restrictive to express that, for instance, some private data might become public for a specific user, or after a release date.

The main drawback of this property is that it considers only initial configurations that can lead to termination. A property is said to be termination-insensitive when it assumes that the attacker does not learn anything from observing that an execution terminates or not. Here the attacker is assumed not to learn anything at all from an execution that cannot terminate, which is a stronger hypothesis.

Definition for reactive programs

The last definition is sufficient if we consider only "computing" programs, but cannot be used for reactive programs, that typically do not terminate.

An easy way to adapt it, is to accept that the attacker may learn information from program termination. Then we obtain another termination-insensitive version:

Definition 25. *A statement S is (TI) non-interfering with respect to L if*

$$\begin{array}{l}\forall \mu \in \mathit{Store}.\, \forall T \in \mathit{Trace}.\, \langle S, \mu, TK_0\rangle \Downarrow T \Rightarrow \\ \quad \forall i \in \mathbb{N}.\, K(S, \mu_{|L}, \emptyset) \subset \mathit{knowl}(T_i)\end{array}$$

Thus the knowledge in T should never be more precise than the initial knowledge.

We could also give a termination-sensitive version :

Definition 26. *A statement S is (TS) non-interfering w.r.t. L if*

$$\begin{array}{l}(1)\forall \mu \in \mathit{Store}.\, \forall T \in \mathit{Trace}.\, \langle S, \mu, TK_0\rangle \Downarrow T \Rightarrow \\ \quad \forall i \in \mathbb{N}.\, K(S, \mu_{|L}, \emptyset) \subset \mathit{knowl}(T_i)\\ (2)\forall \mu, \mu' \in \mathit{Store}.\, \forall T, T' \in \mathit{Trace}.\\ \quad \mu \approx_L \mu' \wedge \langle S, \mu\rangle \Downarrow T \wedge \langle S, \mu'\rangle \Downarrow T' \Rightarrow (T \in \mathit{Term} \Longleftrightarrow T' \in \mathit{Term})\end{array}$$

Compositionality

Once again we can use Program 4 to reason about compositionality. This program is rejected by the definitions of this section (the knowledge shrinks when a direct flow happens), whereas its two threads, in isolation, are accepted. Hence Non-Interference in terms of knowledge is not compositional.

Possibilistic property

All these definitions are possibilistic definitions. They do not reject an intuitively insecure program such as Program 5, where the attacker can infer h's value from repeated executions.

3.2 Comparison of definitions

We compare all proposals introduced in the last section. This could not be achieved before, since they were expressed over different models. We want to reveal all relations of the sort "definition A is stronger than definition B", in the sense that all programs rejected by B are also rejected by A.

We start with a general discussion about important properties that emerge from the definitions. Second, as an example, we compare observational determinism with eager and lazy trace invariance. Then we continue with the general comparison.

3.2.1 Important Characteristics and Properties

In this subsection, we compare the definitions we have seen so far with respect to some important properties of Non-Interference.

Definitions for concurrent programs

All proposals are given for concurrent programs. They do not just make sure that private input does not interfere with the final values of public variables, which would be sufficient for sequential programs, but also that intermediate observations of an attacker are described either in terms of intermediate states, either in terms of actions.

Compositionality

Compositionality is a very important notion in security, since it allows to compose safely secure programs. Any definition that is not compositional is very likely to have a marginal utility in concurrent programming.

Unfortunately, most of the definitions we have seen are not compositional. This is due to the fact that our intuitive idea of Non-Interference is not compositional either : a program can be intuitively safe when run alone, and leak private data when run in context. Thus we would expect a definition of Non-Interference to accept this program in the first case and reject it in the other. In other words, we do not expect a definition of Non-Interference close to our intuition to be compositional.

Definition 19 is compositional. We have seen that compositionality is achieved at the cost of completeness. But programs which are rejected are programs possibly unsecure; for instance Example 8 hides an explicit flow.

Possibilistic or probabilistic

We can divide the definitions in two groups : possibilistic and probabilistic definitions. Possibilistic definitions do not distinguish two executions where the same event can happen with two different (non-null) probabilities. In contrast, probabilistic definitions will consider those two executions as different, and thus tend to prevent a low-attacker to infer the value of H variables from repeated computations.

Therefore, we cannot compare possibilistic definitions with probabilistic definitions since they do not assume the same distinction abilities from the attacker.

For any probabilistic definition, it is important to know whether it assumes a random scheduling policy or whether it requires that one explicitly states the scheduling policy (σ in our model). A definition with a random scheduling policy is independent from the scheduler. In general, an application is not intended for a specific scheduling policy. Thus definitions which require the scheduling policy to be explicitly specified are of limited utility.

3.2.2 Observational determinism versus eager/lazy trace invariance

We show that observational determinism and eager/lazy trace invariance cannot be related. Program 3 is eagerly trace invariant but not observationally deterministic and Program 12 is observationally deterministic but not eagerly trace invariant. This is due to the gap between actions (here given by the histories) and stores.

Let us consider Program 3. We have already shown that it is not observationally deterministic. Any history of an execution of Program 3 is a prefix of one of these histories :

$$h_1 = \{write_{l,true}\}\{write_{l,false}\}$$
$$h_2 = \{write_{l,false}\}\{write_{l,true}\}$$

In this case, two histories are equal *w.r.t. L-Actions* if and only if they are equal. Thus the conclusion of eager trace invariance holds since $\equiv_{L-Actions}$ is an equivalence relation. Program 3 is eagerly trace invariant (and actually this is the case for any program which does not involve H variables).

Let us consider the following program.

Program 12.

$$l := true; \texttt{if } (h) \texttt{ then } (\texttt{if } (l) \texttt{ then } \epsilon \texttt{ else } \epsilon) \texttt{ else } \epsilon$$

Program 12 is observationally deterministic and can produce these two histories after two steps:

$$hist = \{write_{l,true}\}\{read_{h,true}\}$$
$$hist' = \{write_{l,true}\}\{read_{h,false}\}$$

They are equal *w.r.t. L-Actions*, but configurations reachable after $hist$ can produce the following history

$$hist'' = \{write_{l,true}\}\{read_{h,true}\}\{read_{l,true}\}$$

and there is no history produced by any configuration reachable after $hist'$ which is equal to $hist''$ *w.r.t. L-Actions*. Thus Program 12 is not eagerly trace invariant.

We have shown that observational determinism and eager trace invariance cannot be related. Observational determinism and lazy trace invariance are not related either (Program 12 is not eagerly trace invariant, so it is not lazyly trace invariant and Program 3 is also lazyly trace invariant).

3.2.3 Comparison for a set of examples

Table 3.1 indicates for a set of programs if they are accepted (a) or rejected (r) by the main definitions we have presented in Section 3. Most programs have already been used to illustrate one of these definitions.

Here are some remarks concerning the table. The first remarks are specific to some properties. Next we give more general remarks.

- If a program does not involve any high variable, then Id_{Stmt} is a partial probabilistic low-bisimulation on $Stmt$. Thus this program is accepted by Definition 19. This explains parts of the results for this definition.
- If a program does not involve any high variable, then the knowledge of an attacker is total from the beginning, thus it cannot grow and the Non-Interference property given in terms of knowledge (Definition 25) holds trivially.
- If a program does not involve any high variable, the relation $=_{L-Actions}$ on histories is the identity relation. As we have already seen in an example with Program 3, in this case the conclusions of the eager and lazy trace invariance properties (Definitions 10 and 11) hold trivially.

Note that it is not useless to check that programs involving only low variable are not rejected, see for instance the columns corresponding to observational determinism (Definition 6) and probabilistic Non-Interference (Definition 14).

About the division in possibilistic and probabilistic definitions, we can remark that Program 5 is accepted by all possibilistic definitions, whereas it is correctly rejected by all probabilistic definitions. Generally speaking, probabilistic definitions are stronger than possibilistic definitions, since they assume a stronger attacker.

The last column indicates our intuition. Notice that it is not always possible to decide if a program is intuitively secure or not, even given a model of the execution of programs. For instance the answer depends on whether one wants a compositional definition (see Program 8). It also depends on how one takes time aspects into account (see Program 12).

Conclusion for the comparison First of all, notice that it only makes sense to compare possibilistic definitions with other possibilistic definitions (and similarly for probabilistic definitions). We are mainly interested in relations between original proposals. The table shows that for any two definitions A and B, there exists a program accepted by A and rejected by B. Thus we conclude that all proposals are incomparable.

This comparison also suggests that observational determinism (6) and the original version of probabilistic Non-Interference (14) reject too many intuitively secure non-deterministic programs. The trace invariance properties (10, 11) and Non-Interference in terms of knowledge (25) are closer to our intuitive definition, but do not consider probabilistic attacks. Moreover, the latter requires to compute the set called "knowledge" in order to perform verification. Finally, Non-Interference

	Possibilistic					Probabilistic				intuition
	6	7	10	11	25	14	15	19	20	
Prog.1 (low)	R	r	a	a	a	a	a	a	a	secure
Prog.2	R	r	r	r	r	r	r	r	r	unsecure
Prog.3 (low)	R	r	A	A	a	R	A	a	a	secure
Prog.5	A	a	A	A	A	R	R	R	r	unsecure
Prog.6 (low)	A	R	a	a	a	a	a	a	a	secure
Prog.8	a	a	A	R	a	a	a	r	a	?
Prog.9	r	r	r	r	r	r	R	r	r	unsecure
Prog.12	A	a	R	R	a	a	a	a	a	?

Prog.1	`while` $(true)$ `{` ϵ `}` $\|_p\ l := 7$
Prog.2	$(l := true \|_p\ l := false) \|_p\ l := h$
Prog.3	$l := true \|_p\ l := false$
Prog.5	`if` (h) `then` $(l_1 := true \|_p\ l_2 := false)$ `else` $(l_1 := true \|_{p'}\ l_2 := false)$
Prog.6	$l_1 := true \|_p\ l_2 := true$
Prog.8	$h := 0$`;if` $(h = 1)$ `then` $l := h'$ `else` ϵ
Prog.9	(`if` (h) `then` $l := true$ `else` ϵ) $\|_p$ $(l := true \|_p\ l := false)$
Prog.12	$l := true$`;if` (h) `then` (`if` (l) `then` ϵ `else` ϵ) `else` ϵ

Definitions :

6 : observational determinism
7 : strong observational determinism
10 : eager trace invariance
11 : lazy trace invariance
25 : termination insensitive knowledge-based Non-Interference
14 : probabilistic Non-Interference
15 : corrected probabilistic Non-Interference
19 : Non-Interference based on low bisimulation on *Stmt*
20 : Non-Interference based on bisimulation on *Config*

Letter “a” means accepted and “r” means rejected.
Upper-case letters ("A" and "R") correspond to examples discussed in the text, and the tag "(low)" means that the program only deals with variables in Var_L.

Table 3.1. Accepted and rejected programs by the main definitions.

in terms of bisimulation (19) both matches our intuition of secure information flow and prevents the attacker from exploiting probabilities. In addition, this property is compositional.

3.3 Other related work

The security community has achieved an important body of work about Non-Interference, and it was not possible to cover all aspects in our comparison. This sections addresses related work about security type systems and abstract interpretation of Non-Interference.

3.3.1 Language based definitions

Probabilistic Non-Interference [18] and Non-Interference in terms of partial probabilistic bisimulation [16], presented in this text, were introduced by the authors along with a security type system to enforce them. Boudol and Castellani ([4], 2002) propose a less restrictive type system than the one presented by Volpano and Smith [18]. Sabelfeld and Sands's type system improves on earlier type systems and is scheduler independent [16]. Their work has led to elegant static verification methods, but also to a better understanding of the impact of concurrency on Non-Interference and of the different mechanisms of information flow.

3.3.2 Abstract interpretation

Mastroeni and Giacobazzi propose an abstract interpretation of Non-Interference ([9], 2004). It is a weaker notion of Non-Interference that takes into account what an attacker actually observes.

This property is defined for deterministic programs only. They make use of state transformers to capture the semantics of programs. Properties of the low store and high store are represented by upper-closure operators (u.c.o.) on $\mathcal{P}(Store_L)$ (respectively $\mathcal{P}(Store_H)$). Such operators are extensive, monotone and idempotent.

The definition of η-ϕ-ρ secrecy stipulates that, citing the authors [9],

> "when the attacker is able to observe the property η of public input and the property ρ of public output, then no information flow concerning the property ϕ of private input interferes in the observable property ρ of public output, under the assumption that the public input property η does not change"

At one extreme, taking $\eta = \rho = Id_{\mathcal{P}(Store_L)}, \phi = Id_{\mathcal{P}(Store_H)}$, we obtain the standard definition of Non-Interference for sequential programs.

The main advantage of this notion is that we can characterize the most concrete attacker for which a program is secure.

Since the definition depends on upper-closure operators, it is difficult to re-express it over our model. Moreover, it has not been applied to concurrent programs. That is why we did not include it in our comparison.

Part II

Verifying Non-Interference

Chapter 4

Characterization in modal μ-calculus

Up to now, we have seen different definitions that should ensure secure information flow of multi-threaded applications. Another important question is how one can verify that such a definition holds for a program. The traditional approach to the verification of secure information flow is the use of information flow type systems [17, 16, 4]. Assuming that every variable has a security level associated to it, typing rules guarantee that if a program is typable, it does not contain any unsecure flow. Type systems provide an efficient and automatic method to verify secure information flow. Moreover, security type systems are also applicable in a probabilistic context.

However, the use of type systems also has drawbacks. In particular, the method is not precise (some harmless programs are not typable), since it is based on syntactic equality, and does not consider values in context.

Therefore, recently as an alternative approach has been suggested to recast the problem into a standard program verification problem [3] : by composing a program with itself, two different program executions can be compared in a single program specification. We use this approach to recast verification into a model checking problem : as a model we have a product of two (or more) basic models representing a program, and we re-express the Non-Interference property as a temporal logic formula over this composed model. This chapter describes this process for observational determinism and eager trace invariance. However, we first give a more detailed description of the general approach.

Temporal logics have propositions qualified in terms of time. They are useful to state requirements about the behaviour of software systems. Our approach relies on model checkers, *i.e.* tools that algorithmically verify satisfaction of a temporal logic formula for a given model. An important contribution of our work is to verify Non-Interference for concurrent programs from a temporal logic characterization.

4.1 Non-Interference by temporal logic

This section describes our general approach and how we have applied it to observational determinism and eager trace invariance, using a temporal logic called modal

μ-calculus. First, we introduce temporal logics and give the syntax and semantics of modal μ-calculus.

4.1.1 Temporal logics

Two kinds of temporal logics are distinguished : linear-time logics, which give the ability to reason about one time line only, and branching-time logics, which allow to reason about several time lines. Linear Temporal Logic (LTL, [13]) belongs to the first category, while Computational Tree Logic (CTL, [8]) and modal μ-calculus [5] belong to the second. Both LTL and CTL can be encoded in modal μ-calculus, which is built on top of the Hennessy-Milner Logic (HML), a temporal logic with modalities, by adding fixed-point operators.

Earlier experiments have shown that full modal μ-calculus is needed in order to express Non-Interference properties.

4.1.2 Modal μ-calculus

Hennessy-Milner Logic allows to reason about possibilities and necessities, given a labelled transition system. Modal μ-calculus has two more formulae, called fixed-point formulae, introduced in order to add recursion to HML. We have added atomic propositions and operators from first order logic to the syntax.

Syntax Let *VarNames* be a set of variable names, ranged over by X. Let *Act* be the set of actions labels, ranged over by α, and let *Atoms* be the set of atomic propositions, ranged over by p. The syntax of modal μ-calculus formulae (*w.r.t.* to these sets) is given by

$$\Phi ::= \mathsf{tt} \mid \mathsf{ff} \mid p \mid X \mid \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \langle\alpha\rangle\,\Phi \mid [\alpha]\,\Phi \mid \mu X.\,\Phi \mid \nu X.\,\Phi$$

Approximants We will give the semantics of modal μ-calculus formulae for finite state systems. It makes use of fixed-point approximants, which are inductively defined as follows ($k \in \mathbb{N}$):

$$\begin{array}{llll} \mu X^0.\,\Phi & \stackrel{def}{=} & \mathsf{ff} & \nu X^0.\,\Phi \stackrel{def}{=} \mathsf{tt} \\ \mu X^{k+1}.\,\Phi & \stackrel{def}{=} & \Phi[\mu X^k.\Phi/X] & \nu X^{k+1}.\,\Phi \stackrel{def}{=} \Phi[\nu X^k.\Phi/X] \end{array}$$

Semantics Figure 4.1 (on page 45) gives the semantics of modal μ-calculus. It is given *w.r.t.* a labelled transition system $T = (\mathcal{S}, Act, \rightarrow, \lambda)$, where $\lambda : \mathcal{S} \rightarrow 2^{Atoms}$ is a labelling function. The symbol s ranges over $\mathcal{S}$.

4.1.3 Self-composition

The first characterization of a Non-Interference property in temporal logic was given by Barthe *et al.* ([3], 2004). Their approach is based on self-composition.

$$
\begin{array}{lcl}
s \models^T \mathsf{tt} & \stackrel{def}{\Leftrightarrow} & \mathit{true} \\
s \models^T \mathsf{ff} & \stackrel{def}{\Leftrightarrow} & \mathit{false} \\
s \models^T p & \stackrel{def}{\Leftrightarrow} & p \in \lambda(s) \\
s \models^T \neg\Phi & \stackrel{def}{\Leftrightarrow} & \neg\,(s \models^T \Phi) \\
s \models^T \Phi \wedge \Psi & \stackrel{def}{\Leftrightarrow} & s \models^T \Phi \;\wedge\; s \models^T \Psi \\
s \models^T \langle\alpha\rangle\,\Phi & \stackrel{def}{\Leftrightarrow} & \exists s' \in \mathcal{S}.\,(s \xrightarrow{\alpha} s' \;\wedge\; s' \models^T \Phi) \\
s \models^T [\alpha]\,\Phi & \stackrel{def}{\Leftrightarrow} & \forall s' \in \mathcal{S}.\,(s \xrightarrow{\alpha} s' \;\Rightarrow\; s' \models^T \Phi) \\
s \models^T \mu X.\,\Phi & \stackrel{def}{\Leftrightarrow} & \exists k \in \mathbb{N}.\, s \models^T \mu X^k.\Phi \\
s \models^T \nu X.\,\Phi & \stackrel{def}{\Leftrightarrow} & \forall k \in \mathbb{N}.\, s \models^T \nu X^k.\Phi
\end{array}
$$

Figure 4.1. Semantics of modal μ-calculus

Principle

In order to understand why self-composition is needed, we must consider the expressive power of temporal logics. They allow to reason about a single execution of a program model. However, the definitions of Non-Interference we have presented in Chapter 3 involve several executions. For instance, the conclusion of observational determinism requires indistinguishability of two traces. Thus it is not possible to find a characterization directly.

Self-composition consists in executing the program in parallel with itself, each part of the program having its own store. Barthe *et al.* show that Non-Interference of two programs with independent stores is equivalent to Non-Interference of the composition of those two programs. Thus it is sufficient to find a characterization of Non-Interference for the self-composed program model. The advantage is that, with a self-composed program model, it is possible to characterize properties such as trace indistinguishability.

Barthe *et al.* give a definition of Non-Interference which simply analyses the input/output behaviour of the self-composed program (thus this definition cannot prevent unsecure flow if we consider multi-threaded programs). They propose a CTL and a LTL characterization of this property, and perform model checking with the SPIN model checker.

Previous work in the setting of concurrency

Huisman *et al.* propose a characterization of observational determinism in CTL*, based on self-composition, and prove it correct [11]. CTL* is a logic that subsumes both LTL and CTL. They also propose a characterization directly in modal μ-calculus, but give no proof of its correctness.

None of the proposals of Non-Interference presented in this text has been verified before using model checking. Since there is no readily available model checker for CTL*, Huisman *et al.* rephrase there CTL* characterization of observational determinism in μ-calculus. But as their model checker (Evaluator, in the CADP

tool-set) only supports alternation-free modal μ-calculus, they are forced to modify the formula and model check a stronger property than observational determinism.

Nevertheless, their study indicates that a direct characterization of security properties in modal μ-calculus is probably the more promising approach to achieve model checking.

Following their work, we give a modal μ-calculus characterization of observational determinism and prove it correct. We also give a characterization of trace invariance. We have model checked tiny examples for these two properties with the Concurrency Workbench (CWB), a model checker that supports modal μ-calculus. Before giving the characterizations, we recall the syntax and semantics of the modal μ-calculus.

4.2 Characterization of observational determinism

This characterization is based on self-composition. We build a program model that describes two independent executions of the program we want to verify, each part with its own store. One state is composed of two configurations (one for each part). This program model is built from the model of Chapter 2.

4.2.1 Self-composed program model

First we define the set of action labels $Action_{OD}$ for a simple program. Notice that it is different from $Action$, defined in Chapter 2 to introduce histories, since histories are not important here.

$$Action_{OD} = \{\mathsf{c}_{x,v} \mid x \in Var \ \wedge \ v \in dom(x)\} \cup \{\mathsf{nc}\}$$

For all x in Var, we also define

$$c_x = \{\mathsf{c}_{x,v} \mid v \in dom(x)\}$$

In order to have labelled transitions, we update the operational semantics of Figure 2.1 with actions in $Action_{OD}$. We use $\mathsf{c}_{x,v}$ to label a transition that assigns v to variable x, where v is different from x's former value. All other transitions leave the store unchanged and will be labelled with nc ("$\mathsf{no\ change}$"). The `(assign)` rule is updated as follow :

$$\frac{\mu(x) \neq E(\mu) \qquad v = E(\mu)}{\langle x := E, \mu\rangle \xrightarrow{\mathsf{c}_{x,v}} \langle \epsilon, \mu(x \rightarrow v)\rangle}$$

$$\frac{\mu(x) = E(\mu)}{\langle x := E, \mu\rangle \xrightarrow{\mathsf{nc}} \langle \epsilon, \mu(x \rightarrow E(\mu))\rangle}$$

All other rules that have no premise are labelled with nc. Finally, the rules for parallel and sequential composition inherit the transition label from the action that is executed.

We can now define the labelled transition system $T = (\mathcal{S}, Act, \rightarrow, \lambda)$ on which our characterization is defined.

- The set of states $\mathcal{S}$ is $Config \times Config$.
- Act is the set defined by

$$Act = \{(a)_j \mid a \in Action_{OD} \ \wedge \ j \in \{1,2\}\}$$

  The index j denotes which part of the self-composed program performs the action.

- The transition relation $\rightarrow$ on $\mathcal{S}$ is defined from the updated operational semantics :

$$\frac{\langle S_1, \mu_1\rangle \xrightarrow{a} \langle S_1', \mu_1'\rangle}{(\langle S_1, \mu_1\rangle, \langle S_2, \mu_2\rangle) \xrightarrow{(a)_1} (\langle S_1', \mu_1'\rangle, \langle S_2, \mu_2\rangle)}$$

$$\frac{\langle S_2, \mu_2\rangle \xrightarrow{a} \langle S_2', \mu_2'\rangle}{(\langle S_1, \mu_1\rangle, \langle S_2, \mu_2\rangle) \xrightarrow{(a)_2} (\langle S_1, \mu_1\rangle, \langle S_2', \mu_2'\rangle)}$$

- Finally λ is defined by

$$\forall x \in Var. \quad \mathsf{eq}_x \in \lambda((c_1, c_2)) \overset{def}{\Leftrightarrow} c_1(x) = c_2(x)$$

4.2.2 Characterization

First we recall the definition of observational determinism.

Definition (Observational Determinism)**.** *A program S is observationally deterministic if*

$$\forall \mu, \mu' \in Store.\, \forall T, T' \in Trace.$$
$$\mu \approx_L \mu' \wedge \ < S, \mu > \Downarrow T \wedge \ < S, \mu' > \Downarrow T' \ \Rightarrow \ T \approx_L T'$$

We want to prove that observational determinism is characterized by Formula Φ_{OD}.

$$\begin{array}{lcll} \Phi_{OD} & = & (\bigwedge_{l \in Var_L} \mathsf{eq}_l) \Rightarrow \bigwedge_{l \in Var_L} \Psi_l & \\ \Psi_l & = & \nu R.\ \ \mathsf{always}^{(\overline{c_l})_1}([(\mathsf{c}_l)_1]\,(& \mathsf{finally}^{(-)_2^L}(\mathsf{eq}_l) \wedge \\ & & & \mathsf{always}^{(\overline{c_l})_2}([(\mathsf{c}_l)_2]\,(\mathsf{eq}_l \wedge R)))) \end{array} \tag{4.1}$$

where

$$\begin{array}{lcll} (-)_i^L & = & \{(a)_i \mid \exists l \in Var_L.\, a = c_l \vee a = \mathsf{nc}\} & , i \in \{1,2\} \\ (\overline{c_l})_i & = & \{(a)_i \mid a \neq c_l\} & , i \in \{1,2\} \end{array}$$

Φ_{OD} uses the following abbreviations :

$$\begin{array}{lcl} \mathsf{always}^A(\phi) & = & \nu X.\ \ \phi \wedge \ (\bigwedge\limits_{a \in A} [a]\, X) \\ \mathsf{finally}^A(\phi) & = & \mu X.\ \ \phi \vee \ (\bigwedge\limits_{a \in A} [a]\, X) \end{array}$$

A state s satisfies formula $\mathsf{always}^A(\phi)$ *iff* ϕ holds along all paths starting from s with transitions labelled in A. Next, s satisfies $\mathsf{finally}^A(\phi)$ *iff* all paths starting from s with transitions labelled in A have a state for which ϕ holds.

Formula Φ_{OD} says that if the stores of the two parts of the self-composed program model are L-equivalent ($\bigwedge_{l \in Var_L} \mathsf{eq}_l$), then the trace corresponding to the transitions of the first part and the trace corresponding to the transitions of the second part are indistinguishable ($\bigwedge_{l \in Var_L} \Psi_l$).

Formula Ψ_l characterizes stuttering equivalence for the location traces of l. It says that whenever the first part changes l's value ($[(\mathsf{c}_l)_1] \ldots$), then

- the second part will finally assign the same value to l ($\mathsf{finally}^{(-)_2^L}(\mathsf{eq}_l)$)
- if the second part is the only one to take transitions ($\mathsf{always}^{(\overline{c_l})_2}(\ldots)$), after it changes l's value for the first time ($[(\mathsf{c}_l)_2] \ldots$), the two stores will be equal and the whole formula will hold again ($\mathsf{eq}_l \wedge R$).

4.2.3 Proof of the Characterization

We now give the theorem.

Theorem 2 (μ-Calculus characterization of observational determinism)**.**
A program S is observationally deterministic if and only if, for all stores μ and μ',

$$(\langle S, \mu \rangle, \langle S, \mu' \rangle) \models^T \Phi_{OD}$$

In what follows, "S is observationally deterministic" will be denoted by $\mathsf{Obs.Det.}(S)$

First Case : $(\forall \mu, \mu' \in \mathit{Store}.\, (\langle S, \mu \rangle, \langle S, \mu' \rangle) \models^T \Phi_{OD}) \Rightarrow \mathsf{Obs.Det.}(S)$

Proof of the first case. We assume that

$$\forall \mu, \mu' \in \mathit{Store}.\, (\langle S, \mu \rangle, \langle S, \mu' \rangle) \models^T \Phi_{OD}$$

Let μ, $\mu' \in \mathit{Store}$, T, $T' \in \mathit{Trace}$ such that $\mu \approx_L \mu'$, $< S, \mu > \Downarrow T$, and $< S, \mu' > \Downarrow T'$. We want to prove that $T \approx_L T'$.
Notice that since $\mu \approx_L \mu'$, we have $(\langle S, \mu \rangle, \langle S, \mu' \rangle) \models^T \bigwedge_{l \in Var_L} \mathsf{eq}_l$. Thus from $(\langle S, \mu \rangle, \langle S, \mu' \rangle) \models^T \Phi_{OD}$, we can conclude that $(\langle S, \mu \rangle, \langle S, \mu' \rangle) \models^T \bigwedge_{l \in Var_L} \Psi_l$. Hence in order to prove the first case, it is sufficient to prove that

$$\begin{array}{l} \forall l \in \mathit{Var}_L.\, \forall \mu, \mu' \in \mathit{Store}.\, \forall T, T' \in \mathit{Trace}. \\ \mu(l) = \mu'(l) \wedge < S, \mu > \Downarrow T \wedge < S, \mu' > \Downarrow T' \wedge (\langle S, \mu \rangle, \langle S, \mu' \rangle) \models^T \Psi_l \Rightarrow T(l) \sim_s T'(l) \end{array}$$

Let $l \in \mathit{Var}_L$. We can rewrite the property above for l, leaving out μ and μ' (which correspond to T_0 and T_0' respectively), as follows :

$$\begin{array}{l} \forall T, T' \in \mathit{Trace}. \\ T_0(l) = T_0'(l) \wedge < S, T_0 > \Downarrow T \wedge < S, T_0' > \Downarrow T' \wedge (T_0, T_0') \models^T \Psi_l \Rightarrow T(l) \sim_s T'(l) \end{array}$$

In order to show that $T(l)$ and $T'(l)$ are stuttering equivalent (denoted $T(l) \sim_s T'(l)$, see Definition 3), we show that for all $i \in \mathbb{N}$, there exists a j such that $[T(l), i] \sim_s [T'(l), j]$ (the problem is symmetric in T and T'). We fix i and define the property Π_n.

$$\begin{aligned} \Pi_n \quad = \quad & \forall T, T' \in \mathit{Trace}.\, T_0(l) = T'_0(l) \wedge (T_0, T'_0) \models^T \Psi_l \Rightarrow \\ & \forall m \leq i.\, \mathit{change}_l^n(T, m) \Rightarrow \exists j.\, [T(l), m] \sim_s [T'(l), j] \end{aligned} \tag{4.2}$$

Π_n uses the abbreviation $\mathit{change}_l^n(T, m)$, which denotes that l's value changes exactly n times between T_0 and T_m.

$$\mathit{change}_l^n(T, m) = (\, |\{i \in \mathbb{N} \mid 0 \leq i < m \wedge T_i(l) \neq T_{i+1}(l)\}| = n \,)$$

The conclusion of Π_n says that if the initial fragment of T contains n changes of l, we can match these changes (up to stuttering) with an initial fragment of T'. We show that Π_n holds for all naturals. We use mathematical induction.

Basis (Π_0) : Let $m \in \mathbb{N}$, $m \leq i$. If $\mathit{change}_l^0(T, m)$, then $[T(l), m] \sim_s [T'(l), 0]$. Thus we can choose $j = 0$.

Inductive step ($\Pi_n \Rightarrow \Pi_{n+1}$) : Now assume Π_n, and let us show Π_{n+1}. Let $m \leq i$ and assume $\mathit{change}_l^{n+1}(T, m)$. As $n + 1 \geq 1$, we can define

$$k = min\{j \mid T_j(l) \neq T_0(l)\}$$

There exists a (possibly empty) sequence of actions in $(\overline{c_l})_1$ (*i.e.* transitions by the first transition system, that does not change l), such that

$$(T_0, T'_0) \xrightarrow{(\overline{c_l})_1}{}^{*} (T_{k-1}, T'_0)$$

Since $(T_0, T'_0) \models^T \Psi_l$, let us consider this formula. As the first $k - 1$ non-trivial approximants of the always subformula must hold, this implies that (T_{k-1}, T'_0) must satisfy Formula (4.3).

$$\begin{aligned} [(\mathsf{c}_l)_1] (\; & \mathsf{finally}^{(-)_2^L}(\mathsf{eq}_l) \wedge \\ & \mathsf{always}^{(\overline{c_l})_2}([(\mathsf{c}_l)_2]\,(\mathsf{eq}_l \wedge R))) \end{aligned} \tag{4.3}$$

Since $(T_{k-1}, T'_0) \xrightarrow{(c_l)_1} (T_k, T'_0)$, state (T_k, T'_0) must satisfy Formula (4.4).

$$\begin{aligned} & \mathsf{finally}^{(-)_2^L}(\mathsf{eq}_l) \wedge \\ & \mathsf{always}^{(\overline{c_l})_2}([(\mathsf{c}_l)_2]\,(\mathsf{eq}_l \wedge R)) \end{aligned} \tag{4.4}$$

Consider $\mathsf{finally}^{(-)_2^L}(\mathsf{eq}_l)$. This is a least fixed point, thus we know that one of its approximant holds in (T_k, T'_0), *i.e.* that there exists a m such that

$$(T_k, T'_0) \models^T \mu X^m.\, \mathsf{eq}_l \vee \left[(-)_2^L\right] X$$

This means that for all sequences of transitions by the second transition system, eq_l eventually holds.
Now let U be the composed trace starting in (T_k, T'_0) defined by

$$\forall j \in \mathbb{N}.\, U_j = (T_k, T'_j)$$

All transitions in U are transitions of the second transition system. Thus subformula $\mathsf{finally}^{(-)^L_2}(\mathsf{eq}_l)$ ensures that it will do a transition after which eq_l holds. By construction of U, this implies that T' has to do a c_l action (since all transitions of U are transitions of T'), and $[(\mathsf{c}_l)_2]\,(\mathsf{eq}_l \wedge R)$ ensures that after this transition the whole formula holds again (by unfolding R). Thus we can define

$$k' = min\{i \in \mathbb{N} \mid T'_i(l) = T_k(l)\}$$

We define the traces T_{sub} and T'_{sub} by removing the first k, k' elements of T and T', respectively.

$$\forall j \in \mathbb{N}.\, (T_{sub})_j = T_{k+j} \;\wedge\; (T'_{sub})_j = T_{k'+j}$$

We have shown that $((T_{sub})_0, (T'_{sub})_0) \models^T \Psi_l$ (since $(T_k, T'_{k'})$ satisfies R). Since $\mathsf{change}^n_l(T_{sub}, m-k)$ and $(T_{sub})_0(l) = (T'_{sub})_0(l)$, the induction hypothesis Π_n instantiated with $((T_{sub})_0, (T'_{sub})_0)$ ensures that there exists j_1 such that $[T_{sub}(l), m-k] \sim_s [T'_{sub}(l), j_1]$.
Moreover, since

$$\forall q < k.\, T_q(l) = T_0(l)$$
$$\forall q < k'.\, T'_q(l) = T'_0(l)$$

we have $[T(l), k-1] \sim_s [T'(l), k'-1]$ and we can conclude that $[T(l), m] \sim_s [T'(l), k' + j_1]$. Thus we can choose $j = k' + j_1$ in Π_{n+1}.

Conclusion : By induction, Π_n holds for all n. Now let T, $T' \in \mathit{Trace}$, such that $T_0(l) = T'_0(l)$ and $(T_0, T'_0) \models^T \Psi_l$. There always exists $n \in \mathbb{N}$ such that $\mathit{change}^n_l(T, i)$. Let $m \in \mathbb{N}$, $m \leq i$. Π_n allows us to conclude that $\exists j.\, [T(l), m] \sim_s [T'(l), j]$. This concludes the first case. □

Second Case : $\mathsf{Obs.Det.}(S) \Rightarrow (\forall \mu, \mu' \in \mathit{Store}.\, (\langle S, \mu\rangle, \langle S, \mu'\rangle) \models^T \Phi_{OD})$

First we introduce two auxiliary lemmas that will be used in the proof.

Lemma 1. *Let $A \subset Act$. If $(c, c') \models^T \neg(\bigwedge_{a \in A} [a]\,\phi)$, and if c' cannot do a transition labelled in A, then there exists $a \in A$ such that $(c, c') \xrightarrow{a} (c'', c')$ and $(c'', c') \models^T \neg\phi$.*

Lemma 2. *Let $A \subset Act$. If $(c, c') \models^T \neg\mathsf{always}^A(\phi)$, and if c' cannot do a transition labelled in A, then there exists i and $T \in Trace$ such that*

- $T_0 = c$
- $(T_i, c') \models^T \neg\phi$
- $\forall j \in [0, i-1].\ \exists a \in A.\ (T_j, c') \xrightarrow{a} (T_{j+1}, c')$

Let us prove Lemma 2. The proof of Lemma 1 is similar, but easier.

Proof of Lemma 2. Assume that $(c, c') \models^T \neg\mathsf{always}^A(\phi)$. By definition,

$$\neg\mathsf{always}^A(\phi) = \neg\, \nu X.\ (\phi \wedge\ (\bigwedge_{a\in A} [a]\, X))$$

We use the following identity :

$$\neg\nu Z.\neg\psi(\neg Z) = \mu Z.\psi(Z)$$

The identification $\neg\psi(\neg Z) = \phi \wedge (\bigwedge_{a\in A} [a]\, Z)$ yields $\psi(Z) = \neg\phi \vee \bigvee_{a\in A} \langle a\rangle\, Z$. Combined with the identity, we obtain :

$$\neg\nu Z.(\phi \wedge\ (\bigwedge_{a\in A} [a]\, Z)) = \mu Z.\ (\neg\phi\ \vee \bigvee_{a\in A} \langle a\rangle\, Z)$$
$$\neg\mathsf{always}^A(\phi) = \mu Z.\ (\neg\phi\ \vee \bigvee_{a\in A} \langle a\rangle\, Z)$$

Thus $(c, c') \models^T \mu X.\, (\neg\phi \vee \bigvee_{a\in A} \langle a\rangle\, X)$. An approximant of this μ formula must hold, and we can define

$$i = min\{k \mid (c, c') \models^T \mu X^k.\ (\neg\phi\ \vee \bigvee_{a\in A} \langle a\rangle\, X)\}$$

Formula $\mu X^i.(\neg\phi \vee \bigvee_{a\in A} \langle a\rangle\, X)$ ensures that there is a path of at most $i-1$ transitions in A leading to a state verifying $\neg\phi$. Assume there exists such a path with n transitions, $n < i-1$. Then we would have $(c, c') \models^T \mu X^{n+1}.\ (\neg\phi\ \vee \bigvee_{a\in A} \langle a\rangle\, X)$, which contradicts the minimality of i. Thus we know that this path must be of length $i-1$.

$$(c, c') \models^T \underbrace{\langle A\rangle\, \langle A\rangle \ldots \langle A\rangle}_{i-1\ times}\, \neg\phi$$

As c' cannot make any transition in A, the $i-1$ transitions are done by the first model. Thus there exists $c_0 = c$, $c_1, c_2, \ldots c_{i-1}$ such that

$$\forall j \in [0, i-2].\ \exists a \in A.\ (c_j, c') \xrightarrow{a} (c_{j+1}, c')$$

and $(c_{i-1}, c') \models^T \neg\phi$. Let T be a trace such that $T_j = c_j$ for $0 \leq j \leq i-1$. Trace T satisfies Lemma 2. □

Proof of the second case. Now we are ready to prove the second case. We prove by contradiction.
Assume that there exist two configurations c and c', with L-equivalent stores, such that $(c, c') \models^T \neg\Phi_{OD}$. Let us show that we can construct two traces from c and c' which are not stuttering equivalent.
There exists l in Var_L for which Ψ_l does not hold. This property is a greatest fixed point, thus one of its approximant does not hold. Let Ψ_l^k denote the k^{th} approximant of Ψ_l. We define Π_k as follows :

$$\begin{aligned}\Pi_k = \quad & \forall c, c' \in \mathit{Config}.\ (c, c') \models^T \neg\Psi_l^k \Rightarrow \\ & \exists T, T'.\ c \Downarrow T \wedge c' \Downarrow T' \wedge \neg(T(l) \sim_s T'(l))\end{aligned}$$

Let us show by induction that Π_k holds for all k in $\mathbb{N}$.

Basis (Π_0) : The first approximant Ψ_l^0 always holds, so Π_0 holds trivially.

Inductive step ($\Pi_n \Rightarrow \Pi_{n+1}$) : Let us assume Π_k. Let c and c' be two configurations such that $(c, c') \models^T \neg\Psi_l^{k+1}$. By unfolding once, we get :

$$\begin{aligned}\neg\mathsf{always}^{(\overline{c_l})_1}\,([(\mathsf{c}_l)_1]\,(\ & \mathsf{finally}^{(-)_2^L}(\mathsf{eq}_l)\ \wedge \\ & \mathsf{always}^{(\overline{c_l})_2}([(\mathsf{c}_l)_2]\,(\mathsf{eq}_l \wedge \Psi_l^k))))\end{aligned}$$

Since all transitions in $(\overline{c_l})_1$ are transitions of model 1, Lemma 2 yields :

$$\begin{aligned}&\exists T \in \mathit{Trace}. \exists i \in \mathbb{N}. \\ &\quad T_0 = c \\ &\quad (T_i, c') \models^T \neg\psi \\ &\quad \forall j \in [0, i-1].\ \exists a \in (\overline{c_l})_1.\ (T_j, c') \xrightarrow{a} (T_{j+1}, c')\end{aligned}$$

where ψ is :

$$\begin{aligned}[(\mathsf{c}_l)_1]\,(\ & \mathsf{finally}^{(-)_2^L}(\mathsf{eq}_l)\ \wedge \\ & \mathsf{always}^{(\overline{c_l})_2}([(\mathsf{c}_l)_2]\,(\mathsf{eq}_l \wedge \Psi_l^k)))\end{aligned}$$

Lemma 1 enables us to choose T such that $(T_i, c') \xrightarrow{(c_l)_1} (T_{i+1}, c')$ and $(T_{i+1}, c') \models^T \neg\psi_2$ where ψ_2 is :

$$\begin{aligned}&\mathsf{finally}^{(-)_2^L}(\mathsf{eq}_l)\ \wedge \\ &\mathsf{always}^{(\overline{c_l})_2}([(\mathsf{c}_l)_2]\,(\mathsf{eq}_l \wedge \Psi_l^k))\end{aligned}$$

Until now, we have built the $i+2$ first elements of T such that $(T_{i+1}, c') \models^T \neg\psi_2$.

- Either $(T_{i+1}, c') \models^T \neg\mathsf{finally}^{(-)_2^L}(\mathsf{eq}_l)$. By definition,

$$\neg\mathsf{finally}^{(-)_2^L}(\mathsf{eq}_l) = \neg\mu X.\ \mathsf{eq}_l \vee \left[(-)_2^L\right] X$$

We use the following identity :

$$\neg\mu Z.\psi(Z) = \nu Z.\neg\psi(\neg Z)$$

which yields $(T_{i+1}, c') \models^T \nu X.\,(\neg\mathsf{eq}_l \wedge \left\langle (-)_2^L \right\rangle X)$. This last formula guarantees that, starting from (T_{i+1}, c'), there is an infinite sequence of transitions labelled in $(-)_2^L$ such that eq_l does not hold throughout this sequence. All these transitions are transitions of model 2, thus there is an infinite sequence of configurations $c'_0 = c', c'_1, c'_2, \ldots$ such that

$$\forall j \geq 0.\, \exists a \in (-)_2^L.\, (T_{i+1}, c'_j) \xrightarrow{a} (T_{i+1}, c'_{j+1}) \wedge (T_{i+1}, c'_j) \models^T \neg\mathsf{eq}_l$$

Let T' be the trace defined by

$$\forall j \geq 0.\, T'_j = c'_j$$

We have seen that

$$c \Downarrow T \wedge c' \Downarrow T' \wedge \forall j \geq 0.\, T'_j(l) \neq T_{i+1}(l)$$

Thus there exists no j such that $[T(l), i+1] \sim_s [T'(l), j]$. T and T' start respectively in c and c', but $T(l)$ and $T'(l)$ are not stuttering equivalent, which contradicts $\mathsf{Obs.Det.}(S)$.

- In other cases, $(c, c') \models^T \neg\mathsf{always}^{(\overline{c_l})_2}([(\mathsf{c}_l)_2]\,(\mathsf{eq}_l \wedge \Psi_l^k))$
  If we swap model 1 and model 2 in Lemma 1 and 2, we can show that

$$\begin{array}{l}\exists j \in \mathbb{N}, T'.\\ \quad \forall n \in [0, j-1]. \exists a \in (\overline{c_l})_2.\, (T_{i+1}, T'_n) \xrightarrow{a} (T_{i+1}, T'_{n+1}) \wedge\\ \quad T'_j \xrightarrow{(c_l)_2} T'_{j+1} \wedge\\ \quad (T_{i+1}, T'_{j+1}) \models^T \neg(\mathsf{eq}_l \wedge \Psi_l^k)\end{array}$$

  - Either $(T_{i+1}, T'_{j+1}) \models^T \neg\mathsf{eq}_l$. Trace $T(l)$ has done exactly one change upto $T_{i+1}(l)$ and trace $T'(l)$ one change upto $T'_{j+1}(l)$. As two traces which are stuttering equivalent must have the same value after the same number of changes, T and T' are not stuttering equivalent.
  - In other cases, $(T_{i+1}, T'_{j+1}) \models^T \mathsf{eq}_l \wedge \neg\Psi_l^k$. Thus, applying the induction hypothesis, there exist two traces T'' and T''' which are not stuttering equivalent starting in T_{i+1} and T'_{j+1}. We choose T and T' such that they correspond to these traces from T_{i+1} and T'_{j+1} :

$$\forall k \geq 0.\, T_{i+1+k} = T''_k \wedge T'_{j+1+k} = T'''_k$$

T and T' are not stuttering equivalent (since this would imply that T'' and T''' are stuttering equivalent), which contradicts $\mathsf{Obs.Det.}(S)$ and concludes the proof.

□

4.3 Characterization of eager trace invariance

This characterization is also based on the idea of self-composition, however it uses a different labelled transition system than the one defined in Section 4.2.1.

First, we do not have to define actions for a simple program here, since we have already defined them in the program model of Chapter 2, in order to be able to express Roscoe's definitions.

Eager trace invariance requires that for every two executions that have the same low actions, there exists a third execution that performs all actions of the second and then copies all future low actions of the first. Therefore we need a model of three executions.

Moreover, our model must allow us to keep the store of the third part uninitialized, while the two other parts are executing. The reason is that initialization reduces the set of possible histories that can communicate this model.

4.3.1 Triple program model

Consider the operational semantics of Figure 2.1 (on page 12). For simplicity, we assume that multi-actions are composed of one action only. Thus histories are sequences of actions. We add labels to transitions according to the following rules :

$$\frac{\langle S, \mu, \mathtt{hist}\rangle \rightarrow \langle S', \mu', \mathtt{hist}\rangle}{\langle S, \mu, \mathtt{hist}\rangle \xrightarrow{\tau} \langle S', \mu', \mathtt{hist}\rangle}$$

$$\frac{\langle S, \mu, \mathtt{hist}\rangle \rightarrow \langle S', \mu', \mathtt{hist}\,\hat{}\,\{a\}\rangle}{\langle S, \mu, \mathtt{hist}\rangle \xrightarrow{a} \langle S', \mu', \mathtt{hist}\,\hat{}\,\{a\}\rangle}$$

where τ is the silent action of process algebra.

For a given statement S, we define the labelled transition system $T_S = (\mathcal{S}, Act, \rightarrow)$.

- The set of states $\mathcal{S}$ is $State^3$, where

$$State = \{\perp\} \cup \{Config\}$$

The symbol $\perp$ denotes a configuration with program S and an uninitialized store. In the sequel, the letter s will always denote an element of $State$, whereas c will denote an element of $Config$.

- *Act* is the set defined by

$$Act = \{\tau\} \cup \{(a)_j \mid a \in Action \cup \{init\} \ \wedge \ j \in \{1,2,3\}\}$$

  The index j denotes which part of the composed program performs the action.

- The transition relation $\rightarrow$ on $\mathcal{S}$ is defined from the updated operational semantics :

$$\frac{\cdot}{(\bot, s_2, s_3) \xrightarrow{(init)_1} (\langle S, \mu\rangle, s_2, s_3)} \quad \textit{for all } \mu \in \textit{Store}$$

$$\frac{c_1 \xrightarrow{\tau} c_1'}{(c_1, s_2, s_3) \xrightarrow{\tau} (c_1', s_2, s_3)}$$

$$\frac{c_1 \xrightarrow{a} c_1'}{(c_1, s_2, s_3) \xrightarrow{(a)_1} (c_1', s_2, s_3)}$$

  (with symmetric cases for model 2 and model 3)

4.3.2 Abstracting away from internal actions

In modal μ-calculus, weak modalities $\langle\!\langle\alpha\rangle\!\rangle$ and $[\![\alpha]\!]$ abstract away from internal communication actions τ. Their semantics uses the notion of weak transitions, that are defined as follows :

Let $(\mathcal{S}, Act, \rightarrow)$ be a labelled transition system. Let $\tau \in Act$ denote a silent transition. We define $\stackrel{\epsilon}{\Rightarrow}$ as $(\stackrel{\tau}{\rightarrow})^*$ (zero or more silent transitions). For $\alpha \in Act$, $\alpha \neq \tau$, we define the $\stackrel{\alpha}{\Rightarrow}$ weak transition relation as follows:

$$\forall s, s' \in \mathcal{S}. \quad s \stackrel{\alpha}{\Rightarrow} s' \quad \stackrel{def}{\Leftrightarrow} \quad s \stackrel{\epsilon}{\Rightarrow}\stackrel{\alpha}{\rightarrow}\stackrel{\epsilon}{\Rightarrow} s'$$

We are now ready to add weak modalities to the syntax of modal μ-calculus :

$$\Phi ::= \mathsf{tt} \mid \mathsf{ff} \mid p \mid X \mid \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \langle\alpha\rangle\,\Phi \mid [\alpha]\,\Phi \mid \langle\!\langle\alpha\rangle\!\rangle\,\Phi \mid [\![\alpha]\!]\,\Phi \mid \mu X.\,\Phi \mid \nu X.\,\Phi$$

The semantics of weak modalities is given by :

$$\begin{array}{lll} s \models^T \langle\!\langle\alpha\rangle\!\rangle\,\Phi & \stackrel{def}{\Leftrightarrow} & \exists s' \in \mathcal{S}.\,(s \stackrel{\alpha}{\Rightarrow} s' \ \wedge \ s' \models^T \Phi) \\ s \models^T [\![\alpha]\!]\,\Phi & \stackrel{def}{\Leftrightarrow} & \forall s' \in \mathcal{S}.\,(s \stackrel{\alpha}{\Rightarrow} s' \ \Rightarrow \ s' \models^T \Phi) \end{array}$$

Moreover, we will use weak transitions to define special transitions that abstract away from high actions.

4.3.3 Abstracting away from high actions locally

In some parts of the formula, we would like to abstract away also from actions in `H-Actions`. Therefore, for the labelled transition system of this section, we define the transition relation $\Rightarrow_H$ by

$$\Rightarrow_H \quad \stackrel{def}{=} \quad (\stackrel{\tau_H}{\Rightarrow})^*$$

where $\stackrel{\tau_H}{\Rightarrow}$ is defined by

$$\forall s, s' \in \mathcal{S}. \quad s \stackrel{\tau_H}{\Rightarrow} s' \quad \stackrel{def}{\Leftrightarrow} \quad \exists a_h \in \texttt{H-Actions}.\, \exists j \in \{1,2,3\}.\, s \stackrel{(a_h)_j}{\Rightarrow} s'$$

($\stackrel{(a_h)_j}{\Rightarrow}$ is a standard weak transition relation, see Section 4.3.2)
For all $a_l \in \texttt{L-Actions}$, we define the $\stackrel{a_l}{\Rightarrow}_H$ transition relation :

$$s \stackrel{a_l}{\Rightarrow}_H s' \quad \stackrel{def}{\Leftrightarrow} \quad s \Rightarrow_H \stackrel{a_l}{\Rightarrow} \Rightarrow_H s'$$

Next, we define the modalities $\langle\!\langle a \rangle\!\rangle_H$ and $[\![a]\!]_H$ by replacing $\stackrel{a}{\Rightarrow}$ by $\stackrel{a}{\Rightarrow}_H$ in the definition of $\langle\!\langle a \rangle\!\rangle$ and $[\![a]\!]$, respectively. Thus, for instance,

$$s \models^T \langle\!\langle a \rangle\!\rangle_H \Phi \quad \stackrel{def}{\Leftrightarrow} \quad \exists s' \in \mathcal{S}.\, (s \stackrel{a}{\Rightarrow}_H s' \wedge s' \models^T \Phi)$$

4.3.4 Characterization

We recall the definition of eager trace invariance :

Definition (Eager trace invariance)**.** *A statement S is eagerly-trace-invariant w.r.t. L if*

$$\begin{array}{l} \forall\, \texttt{hist}, \texttt{hist}' \in \mathit{Hist}.\, \texttt{hist} =_{|\mathit{L\text{-}Actions}} \texttt{hist}' \Rightarrow \\ \quad \forall c \in \mathit{reach}(S, \mathit{Store}, \texttt{hist}).\, \forall T \in \mathit{Trace}.\, c \Downarrow T \;\Rightarrow \\ \quad\quad \exists c' \in \mathit{reach}(S, \mathit{Store}, \texttt{hist}').\, \exists T' \in \mathit{Trace}. \\ \quad\quad c' \Downarrow T' \wedge (\forall m \in \mathbb{N}.\, \exists n \in \mathbb{N}.\, T(\mathit{hist})_m =_{\mathit{L\text{-}Actions}} T'(\mathit{hist})_n) \end{array}$$

The following μ-calculus formula characterizes eager trace invariance :

$$[(init)_1]\,[(init)_2]\,\Phi \tag{4.5}$$

where Φ is

$$\begin{array}{ll} \nu X. & \langle (init)_3 \rangle\, mimic_{3,1} \wedge \\ & \bigwedge_{a_h \in \texttt{H-Actions}} [\![(a_h)_1]\!]\, X \\ & \bigwedge_{a_h \in \texttt{H-Actions}} [\![(a_h)_2]\!]\, \langle (init)_3 \rangle\, \langle\!\langle (a_h)_3 \rangle\!\rangle\, \Psi \\ & \bigwedge_{a_l \in \texttt{L-Actions}} [\![(a_l)_1]\!]\, [\![(a_l)_2]\!]\, \langle (init)_3 \rangle\, \langle\!\langle (a_l)_3 \rangle\!\rangle\, \Psi \end{array}$$

where Ψ is

$$\begin{aligned}\nu Y.\quad & mimic_{3,1} \wedge \\ & \bigwedge_{a_h \in \texttt{H-Actions}} [\![(a_h)_1]\!]\, Y \\ & \bigwedge_{a_h \in \texttt{H-Actions}} [\![(a_h)_2]\!]\, \langle\!\langle (a_h)_3 \rangle\!\rangle\, Y \\ & \bigwedge_{a_l \in \texttt{L-Actions}} [\![(a_l)_1]\!]\, [\![(a_l)_2]\!]\, \langle\!\langle (a_l)_3 \rangle\!\rangle\, Y\end{aligned}$$

and $mimic_{3,1}$ is

$$\nu Z. \bigwedge_{a_l \in \texttt{L-Actions}} [\![(a_l)_1]\!]_H\, \langle\!\langle (a_l)_3 \rangle\!\rangle_H\, Z$$

In the sequel, we will use "model 1", "model 2" and "model 3" to refer to the first, second and third part of our triple program model, respectively.

Formula $mimic_{3,1}$ says that for all histories that generates model 1, model 3 can generate a history which is low equivalent.

Formula Φ involves a third formula, Ψ, which assumes that the store of the third model is already initialized. Intuitively, we loop in Φ until $(init)_3$ has happened. Then we loop in Ψ. Except for the initialization transition $(init)_3$, Φ and Ψ have the same meaning.

Formula Φ and Ψ define all states where $mimic_{3,1}$ should hold. These are all states where

- model 1 and 2 have communicated low equivalent histories, and
- model 3 has communicated exactly the same history (including high actions) as model 2.

(Notice : model 3 might have taken a different execution path in the program, it is only required to communicate the same actions)

In other words, formula Φ and Ψ say that as long as model 1 and model 2 have low equivalent histories (*i.e.* one of them does a high action, or they do the same low action), model 3 can reproduce the actions that model 2 has done so far (including high actions), and then mimic model 1 in its future low actions.

4.3.5 Proof of the characterization (incomplete)

We have not found enough time to prove the characterization completely. However, we prove a lemma that corresponds to part of the characterization.

Theorem 3 (μ-Calculus Characterization of Eager Invariance)**.**
A program S is eager invariant if and only if

$$(\bot, \bot, \bot) \models^{T_S} [(init)_1]\, [(init)_2]\, \Phi$$

In the sequel, "S is eager invariant" will be denoted by $\mathsf{Eag.Inv.}(S)$.

We introduce an auxiliary lemma for which we give a proof.

Lemma 3.

$$\begin{array}{l}(c, c', c'') \models^{T_S} mimic_{3,1} \Rightarrow \\ \quad \forall T^1 \in \mathit{Trace}.\, c \Downarrow T^1 \Rightarrow \exists T^3 \in \mathit{Trace}. \\ \quad\quad (c'' \Downarrow T^3 \wedge \\ \quad\quad\ \forall m_1 \in \mathbb{N}.\, \exists m_3 \in \mathbb{N}.\, T^1(hist)_{m_1} =_{\texttt{L-Actions}} T^3(hist)_{m_3})\end{array}$$

The conclusion of this lemma is close to the conclusion of eager trace invariance. We renamed T into T^1 and T' into T^3 to make it explicit that these traces correspond to transitions in model 1 and 3 of T_S, respectively. This lemma says that if $mimic_{3,1}$ holds, then for any trace T^1 of model 1, model 3 has a trace that matches the low actions of T^1.

Proof of Lemma 3. Let c, c' and c'' be in *Config*. Assume $(c, c', c'') \models^{T_S} mimic_{3,1}$. Let Π_n be the property defined as follows.

$$\begin{array}{rl}\Pi_n = & \forall T^1 \in \mathit{Trace}.\, c \Downarrow T^1 \Rightarrow \exists T^3 \in \mathit{Trace}. \\ & (c'' \Downarrow T^3 \wedge \forall m_1 \leq n.\, \exists m_3 \in \mathbb{N}. \quad T^1(hist)_{m_1} =_{\texttt{L-Actions}} T^3(hist)_{m_3} \wedge \\ & \qquad\qquad\qquad\qquad\qquad\quad (T^1_{m_1}, c', T^3_{m_3}) \models^{T_S} mimic_{3,1})\end{array}$$

In Π_n, model 3 only has to match the low actions corresponding to the first n transitions of model 1. We will show that Π_n holds for all $n \in \mathbb{N}$.

Basis (Π_0) : Let T^1, $T^3 \in \mathit{Trace}$, such that $c \Downarrow T^1$ and $c'' \Downarrow T^3$.

$$T^1(hist)_0 = \emptyset = T^3(hist)_0$$

and

$$(T^1_0, c', T^3_0) = (c, c', c'')$$

Thus, choosing T_3 and $m_3 = 0$, Π_0 holds trivially.

Inductive step ($\Pi_n \Rightarrow \Pi_{n+1}$) : Now assume Π_n holds and let us show Π_{n+1}. Let $T^1 \in \mathit{Trace}$ such that $c \Downarrow T^1$. The induction hypothesis Π_n ensures that there exists a trace T^{ih} such that

$$\begin{array}{rl} c'' \Downarrow T^{ih} \wedge \forall m_1 \leq n.\, \exists m_3 \in \mathbb{N}. & T^1(hist)_{m_1} =_{\texttt{L-Actions}} T^{ih}(hist)_{m_3} \wedge \\ & (T^1_{m_1}, c', T^{ih}_{m_3}) \models^{T_S} mimic_{3,1}\end{array}$$

For any $m_1 \leq n$, let κ_{m_1} denote the witness for m_1 in the formula above. We have three possibilities.

Case(1): $(T^1)_n \xrightarrow{\tau} (T^1)_{n+1}$. Thus $T^1(hist)_n =_{|\texttt{L-Actions}} T^1(hist)_{n+1}$. Since relation $=_{|\texttt{L-Actions}}$ is transitive, we obtain

$$T^1(hist)_{n+1} =_{|\texttt{L-Actions}} T^{ih}(hist)_{\kappa_n}$$

Moreover, since $(T^1_n, c', T^{ih}_{\kappa_n}) \xrightarrow{\tau} (T^1_{n+1}, c', T^{ih}_{\kappa_n})$, and $mimic_{3,1}$ abstracts away from internal actions, we can conclude that $(T^1_{n+1}, c', T^{ih}_{\kappa_n}) \models^{T_S} mimic_{3,1}$. Thus we can choose $T^3 = T^{ih}$, $m_3 = \kappa_{m_1}$ for $m_1 \leq n$ and $m_3 = \kappa_n$ for $m_1 = n+1$. Hence Π_{n+1} holds in this case.

Case(2): Either there exists a_h in `H-Actions` such that ${T^1}_n \xrightarrow{a_h} {T^1}_{n+1}$. Once again $T^1(hist)_n =_{|\texttt{L-Actions}} T^1(hist)_{n+1}$ and we can have the same reasoning as in the case ${T^1}_n \xrightarrow{\tau} {T^1}_{n+1}$. This time the fact that $mimic_{3,1}$ is established is guaranteed by the fact that $mimic_{3,1}$ abstracts away from high actions.

Case(3): There exists a_l in `L-Actions` such that ${T^1}_n \xrightarrow{a_l} {T^1}_{n+1}$. The induction hypothesis ensures that $(T^1_n, c', T^{ih}_{\kappa_n}) \models^{T_S} mimic_{3,1}$. By unfolding $mimic_{3,1}$ once, we derive that

$$(T^1_n, c', T^{ih}_{\kappa_n}) \models^{T_S} [\![(a_l)_1]\!]_H \, \langle\!\langle (a_l)_3 \rangle\!\rangle_H \, mimic_{3,1}$$

Thus

$$(T^1_{n+1}, c', T^{ih}_{\kappa_n}) \models^{T_S} \langle\!\langle (a_l)_3 \rangle\!\rangle_H \, mimic_{3,1}$$

Hence there exist $(c_1, c_2, c_3) \in \mathcal{S}$ such that $(T^1_{n+1}, c', T^{ih}_{\kappa_n}) \overset{(a_l)_3}{\Rightarrow}_H (c_1, c_2, c_3)$ and $(c_1, c_2, c_3) \models^{T_S} mimic_{3,1}$. We also have, by preventing model 1 and 2 to perform transitions, $(T^1_{n+1}, c', T^{ih}_{\kappa_n}) \overset{(a_l)_3}{\Rightarrow}_H (T^1_{n+1}, c', c_3)$. Since $(T^1_{n+1}, c', c_3) \Rightarrow_H (c_1, c_2, c_3)$ and $mimic_{3,1}$ abstracts away from high actions, we also have $(T^1_{n+1}, c', c_3) \models^{T_S} mimic_{3,1}$.
Moreover, c_3 is reachable from $T^{ih}_{\kappa_n}$. Thus there exists T^3 such that $c'' \Downarrow T^3$, and $i_3 \geq \kappa_n$ such that $T^3_{i_3} = c_3$ and $T^3(hist)_{i_3} =_{|\texttt{L-Actions}} T^{ih}(hist)_{\kappa_n} \hat{} \{a_l\}$.
We choose T_3 to show that Π_{n+1} holds. We can choose $m_3 = \kappa_{m_1}$ for $m_1 \leq n$ (since T^3 and T^{ih} coincide until $T^3_{\kappa_n}$) and $m_3 = i_3$ for $m_1 = n+1$.

Conclusion for the lemma : By induction, Π_n holds for all $n \in \mathbb{N}$, which concludes the proof of the lemma. □

Rest of the proof
We believe that Lemma 3 can be used in the proof of Theorem 3.

An important point to show is that for all states (c_1, c_2, c_3) where $mimic_{3,1}$ should hold in Φ and Ψ, if `hist`, `hist'` are the histories communicated so far by model 1 and model 3 respectively, then c_1 is in $reach(S, Store, \texttt{hist})$ and c_3 in $reach(S, Store, \texttt{hist}')$. Then, we can use Lemma 3 to conclude that for any trace T^1 starting in c_1, there exists a trace T^3 starting in c_3 that has low equivalent history.

4.4 Other proposals

In this section we discuss how to characterize the other proposals presented in Chapter 3.

4.4.1 Possibilistic properties

We believe that it would be easy to characterize lazy trace invariance in modal μ-calculus, since eager and lazy trace invariance are very similar properties. The only thing to do is to modify formula $mimic_{3,1}$ so that model 1 and model 3 can take transitions corresponding to the statement HH ("havoc on H", see Definition 11 on page 22).

The last possibilistic definition left is Non-Interference in terms of knowledge of an attacker. Since the knowledge of an attacker involves all possible executions starting with the same program, we are skeptical about finding a temporal logic characterization of Non-Interference in the knowledge setting.

4.4.2 Probabilistic properties

Both probabilistic Non-Interference and Non-Interference in terms of partial probabilistic low bisimulation have probabilistic requirements regarding their execution choices. In order to characterize them, we need a logic that can express that a transition happens with a given probability. Thus the semantics of this logic must be given with respect to a probabilistic transition system. As a consequence, we cannot find a characterization in standard modal μ-calculus.

Cleaveland *et al.* present a μ-calculus based modal logic for describing properties of probabilistic transitions systems [6]. This logics includes formulae as $\mathsf{Pr}_{>p}\Psi$, which is satisfied by a state s if the measure of the observation of Ψ rooted at s exceeds p. Unfortunatelly this is not expressive enough to characterize Non-Interference. Indeed, we would need an operator that permits us to check that, for instance, the measure of the observation of Φ is equal to the measure of the observation of Ψ.

More generally, temporal logics with probabilistic transition systems have been introduced to state properties about the performance of distributed systems and communication protocols [6]. They can be useful for optimization purposes, but they are not intended to express properties like the properties in this text.

Chapter 5

Model checking with the Concurrency Workbench

There are not so many model checkers that support full modal μ-calculus. The Concurrency Workbench (CWB, [7]) supports full modal μ-calculus while, to the best of our knowledge, most other tools only support alternation-free modal μ-calculus. As we expected to obtain modal μ-calculus characterizations of Non-Interference that are not encodable in alternation-free modal μ-calculus, we chose CWB as our model checker.

This chapter describes our experiments with CWB. We successfully achieved model checking of observational determinism with the set of examples of Section 3.2.3. However, we failed to obtain results for eager trace invariance because of state-space explosion.

The following section gives important remarks about modeling our program semantics in CCS, common to both properties. The next section describes the encoding of our formulation of observational determinism. The last one describes our encoding of eager trace invariance.

5.1 Implementation in CCS

CWB has been mainly applied to the verification of communication protocols. It allows us to define processes in CCS, the Calculus of Communicating Systems. CWB's language for CCS only handles basic CCS. In particular, it does not provide any data language. This implies that there are no parameterized actions, nor conditional statements. This is not convenient at all, since we want to encode variables, to be able to change and lookup their values. Instead, we encode this using the agents of basic CCS.

In CCS, when a process performs action a, some parallel process or the environment must simultaneously perform its co-action $'a$ (a corresponds to receiving on channel a and $'a$ corresponds to sending on channel a). If $'a$ is performed by a parallel process, then a and $'a$ form together a silent action τ. This action cor-

responds to an internal choice, it is ignored by the weak modalities $\langle\langle\alpha\rangle\rangle$ and $[\![\alpha]\!]$ of modal μ-calculus. Some actions we introduce only to implement the internal mechanism. Actions implementing the internal mechanism will be hidden thanks to the hiding operator of CCS. Thus they cannot be matched by the environment (the environment cannot do the corresponding co-action), and always appear as silent actions.

All other actions are for communication with the environment (external choices). We have only one input action per model, for instance input-m1 for model 1. After this action, all variables in model 1 are initialized. The other actions contain "output" in their name, denoting a message output to the environment.

Observational determinism and trace invariance assume different actions, thus we have to give two different CCS models.

5.2 Model checking of observational determinism

The following subsection explains why we cannot use directely the $(\mathsf{c}_{x,v})_i$ actions of our characterization (see Section 4.2.2, page 47).

5.2.1 Encoding $(\mathsf{c}_{x,v})_i$ actions

Ideally, we would have one action output to the environment for each transition, corresponding to the label of this transition. But sometimes it takes several actions in the CCS model to carry out one transition of our program model. Therefore, we cannot guarantee that x's value is not updated before action $(\mathsf{c}_{x,v})_i$ and is updated just after, as assumed by our labelled transition system.

To overcome this problem, we introduce one "begin" and one "end" action for each label of the form $(\mathsf{c}_{x,v})_i$, marking respectively the beginning and the end of the transition. This ensures that the store is still unchanged before the "begin" action and that x has its new value after the "end" action. The "begin" action corresponding to $(\mathsf{c}_{x,v})_i$ is

'output-begin-change-x-to-v-mi

We need only one "end" action for all possible $(\mathsf{c}_{x,v})_i$:

'output-end-change-mi

We do not have this problem with transitions labelled with $(\mathsf{nc})_i$, since these transitions leave the store unchanged. We write this action

'output-nochange-mi

We can now define the sets ProgressActions-mi and BeginChangeLowActions-mi that will be used later in order to define properties.

set ProgressActions-mi =
{ 'output-begin-change-x-to-v-mi | $x \in Store \wedge v \in dom(x)$} ∪
{ 'output-end-change-mi, 'output-nochange-mi};
set BeginChangeLowActions-mi =
{ 'output-begin-change-x-to-v-mi | $x \in Store_L \wedge v \in dom(x)$} ∪

5.2.2 Implementation of the store

The agents composing the store must always be able to output the value of a low variable to the environment (*i.e.* the values of low variables can always be inspected), so that we can evaluate the predicates eq_x for $x \in Var_L$. Moreover, they must also be able to signal the values of all variables to the parallel processes composing the program (for instance in order to evaluate guards in conditionals).

In each model, we define one agent per variable and per value. Let $x \in Store$ be a variable of the store and $v \in dom(x)$ one of its possible value. The action signaling to the environment that x's value in model i is v is

'output-value-x-v-mi

The action signaling that x's value in model i is v to the other processes composing the program is

'value-x-v-mi

This last action is matched when some parallel process does the co-action value-x-v-mi. This results in a silent action τ. As this action will be hidden, the environment will not be able to observe it.

Finally if $v_j \in Store$, $v_j \neq v$, and if the agent corresponding to v receives on a channel

change-x-v_j-mi

then it must change into the agent corresponding to v_j.

If $dom(x) = \{v, v_1, \ldots, v_n\}$, we define the agents X$v$-m$i$ by

agent Xv-mi = 'output-value-x-v-mi.Xv-mi +
'value-x-v-mi.Xv-mi +
{change-x-v_j-mi.Xv_j-mi | $j \in [1, n]$}

Xv-mi means "store for x is v for model i".

Example 10. *If x is a boolean variable, the corresponding agents for model* 1 *are :*

*agent Xtrue-m*1 = *'output-value-x-true-m*1*.Xtrue-m*1 +
*'value-x-true-m*1*.Xtrue-m*1 +
*change-x-false-m*1*.Xfalse-m*1*;*
*agent Xfalse-m*1 = *'output-value-x-false-m*1*.Xfalse-m*1 +
*'value-x-false-m*1*.Xfalse-m*1 +
*change-x-true-m*1*.Xtrue-m*1*;*

Notice that in the last example, the property

prop Eq-x = (< 'output-value-x-*true*-m1>T ⇒ < 'output-value-x-*true*-m2 > T) &
(< 'output-value-x-*true*-m2>T ⇒ < 'output-value-x-*true*-m1 > T);

corresponds to the predicate eq_x. We define it for each $x \in Var_L$.

Finally we define an agent to initialize the store. This correspond to an external choice. If we have n variables, x_1, x_2, ... x_n, then Store-mi is the process that can perform the action init-mi, changing into any process of the form

$\mathtt{X1}v_1\mathtt{-m}i \,||\, \mathtt{X2}v_2\mathtt{-m}i \,||\, \ldots \,||\, \mathtt{Xn}v_n\mathtt{-m}i$

where $v_j \in dom(x_j)$ for all j such that $1 \leq j \leq n$.

5.2.3 Implementation of the commands

We need a way to simulate atomicity of transitions. Indeed, the operational semantics implies that all transitions are carried out atomically, thus we must prevent all other transitions to start execution while another transition is not finished (at the granularity of the operational semantics). To model this, commands will take a lock and release it when they are done. The lock can be implemented with one simple agent per model :

agent Lock-mi = 'takeLock-mi.releaseLock-mi.Lock-mi;

We have introduced all the actions used to implement the internal mechanism. Thus for each model we can define the set

set InternalActions-mi =
{ value-x-v-mi, change-x-v-mi | $x \in$ *Store* $\wedge$ $v \in dom(x)$} ∪
{ takeLock-mi, releaseLock-mi};

Commands have to take and release the lock, respect the control flow and update the store according to the semantics. They also have to output the actions corresponding to the labels of transitions.

In the following, we give the CCS parameterized agents implementing the if and assign constructs, in the special case where variables are booleans and guards are variable names.

agent If-mi(value-b-*true*-mi, value-b-*false*-mi, S1-mi, S2-mi) = takeLock-mi.
(value-b-*true*-mi.'output-nochange-mi.'releaseLock-mi.S1-mi +
value-b-*false*-mi.'output-nochange-mi.'releaseLock-mi.S2-mi);

In each model, we have one agent for the assignment of a constant (AssignValue-mi) and one agent for the assignment of the value stored in another variable

(AssignVariable-mi).

agent AssignValue-mi(output-begin-change-x-to-v_1-mi, change-x-v_1-mi, value-x-v_1-mi, value-x-v_2-mi, Follow-mi) =
takeLock-mi. (value-x-v_2-mi.'output-begin-change-x-to-v_1-mi.'change-x-v_1-mi.
'output-end-change-mi.'releaseLock-mi.Follow-mi +
value-x-v_1-mi.'output-nochange-mi.'releaseLock-mi.Follow-mi);

Example 11. *For instance, the agent that carries out command* x `:=` $false$ *in model* 1 *is :*

*AssignValue-m*1*(output-begin-change-x-to-false-m*1*, change-x-false-m*1*, value-x-false-m*1*, value-x-true-m*1*, 0)*

*After the agent has taken the lock, either x's value is already false (that corresponds to value-x-false-m*1*), thus nothing needs to be changed; the agent does 'output-nochange-m*1 *and then 'releaseLock-mi. In other cases, x is true (value-x-true-m*1*), the agent signals the change to the environment with output-begin-change-x-to-false-m*1*, triggers the change internally with change-x-false-m*1*, signals the end of the change to the environment with 'output-end-change-m*1 *and then releases the lock.*

When the boolean is assigned the value of another (boolean) variable, we have the following agent :

agent AssignVariable-mi(output-begin-change-x-to-v_1-mi, change-x-v_1-mi,
output-begin-change-x-to-v_2-mi, change-x-v_2-mi, value-x-v_1-mi, value-x-v_2-mi,
value-y-v_1-mi, value-y-v_2-mi, Follow-mi) =
takeLock-mi. (
value-y-v_1-mi.(value-x-v_2-mi.'output-begin-change-x-to-v_1-mi.'change-x-v_1-mi.
'output-end-change-mi.'releaseLock-mi.Follow-mi +
value-x-v_1-mi.'output-nochange-mi.'releaseLock-mi.Follow-mi) +
value-y-v_2-mi.(value-x-v_1-mi.'output-begin-change-x-to-v_2-mi.'change-x-v_2-mi.
'output-end-change-mi.'releaseLock-mi.Follow-mi +
value-x-v_2-mi.'output-nochange-mi.'releaseLock-mi.Follow-mi)
);

It is not possible to define a parameterized agent for the while construct directly because of limitations of CWB's language for CCS. Nevertheless, it is still possible to create an agent executing `while (`b`) { `C `}` using the parameterized agent for if and the agent executing C.

The parallel composition is implemented directly using CCS's parallel composition operator ||. Sequential composition, as in S_1`;`S_2, is achieved by giving the agent for S_2 as a parameter to the agent for S_1.

5.2.4 Complete CWB program model

We have defined all the agents needed for the execution of a program. If Program-mi is the agent for the program in model mi, and Store-mi the agent initializing

the store of mi, then the whole agent for model mi is

agent Example-mi = (Lock-mi | Store-mi | Program-mi)\ InternalActions-mi ;

Thus the complete model of the self-composed program is given by

Example-m1 | Example-m2

5.2.5 Properties

Although CWB's language for propositions is the modal μ-calculus, the notations are slightly different than the one we have introduced. In the sequel, we stick to CWB's notations.

We have already given the formulae for predicate eq_x, for all $x \in Var_L$.

Two different notions of deadlock A deadlock is written ⟦-⟧F, where "-" is the wildcard. It means that we cannot do any (non-silent) action. Notice that for each variable x, the store can always do action 'value-x-v-mi where v is x's current value. Thus our program model is never deadlocked in the sense of CCS.

Here we want to express for instance that model mi cannot do any action corresponding to the labelled transitions. This corresponds to a deadlock or a termination in the sense of the operational semantics, since it occurs when model mi cannot do any other step, but it is not a deadlock in the sense of CCS. We define :

prop Deadlock-mi = [[ProgressActions-mi]]F;

Taking into account ending executions Remember that the operational semantics of our program model, described in Figure 2.1, ensures that a program can always do at least one transition, even when it is finished or deadlocked. Unfortunately, it was not possible to have the equivalent of the `(deadl)` and `(term)` transitions in this CCS model. Except for these two transitions, the semantics is respected.

As we have no `(deadl)` and `(term)` transitions in the CCS model, Deadlock-mi might evaluate to T (true). That changes the meaning of the modal μ-calculus formula $\mathsf{finally}^A(\phi)$, compared to the characterization of Section 4.2.2.

$$\mathsf{finally}^A(\phi) = \mu X.\phi \vee (\bigwedge_{a \in A} [a]\, X)$$

The reason is that if Deadlock-mi evaluates to T, then min(X. Phi | ⟦A⟧ X) might evaluate to T even if Phi does not hold. To obtain the original meaning of $\mathsf{finally}^A(\phi)$, which was that ϕ must finally hold along all paths in A, we exclude the unwanted scenario. Thus the property corresponding to $\mathsf{finally}^{(-)^L_i}(\phi)$ is

Finally-mi(Phi) & ∼CanHoldUntilDead-mi(∼Phi)

where

prop Finally-mi(Phi) = min(X. Phi | ⟦`ProgressActions-m`i⟧X);

prop CanHoldUntilDead-mi(Phi) =
min(X. (Phi & Deadlock-mi) | (Phi & ⟪`ProgressActions-m`i⟫X));

Formula ∼CanHoldUntilDead-mi(∼Phi) ensures that there is no path on which ¬Phi (written ∼Phi with CWB's notation) holds until it deadlocks.

Always predicate Another predicate, $\mathsf{always}^{(\overline{c_x})_i}(\phi)$, leads us to introduce the set Compl-change-x-mi, equal to ProgressAction-mi bereft of the actions in {'change-x-v-out-mi | $v \in dom(x)$}.

prop Always-x-mi(Phi) = max(X. Phi & [[Compl-change-x-mi]]X);

Observational Determinism Observational Determinism requires that trace indistinguishability holds whenever the initial stores are L-equivalent.

prop ObervationalDeterminism = [[init-m1]][[init-m2]] $\bigcup_{x \in Var_L}$ (Eq-x ⇒ TraceInd-x);

where TraceInd-x is defined using all properties seen until now :

```
prop TraceInd-x = max(R. Always-x-m1(
   [[BeginChangeLowActions-m1]][['output-end-change-m1]]
   (Finally-m2(Eq-x) & ∼CanHoldUntilDead-m2(∼Eq-x) &
    Always-x-m2([[BeginChangeLowActions-m2]][['output-end-change-m2]](Eq-x & R)) )
));
```

5.2.6 Results

We have model checked the programs of Section 3.2.3 for observational determinism. They are correctly accepted or rejected (*i.e.* we have the same results as in Table 3.1, on page 38). Moreover results are given within a few seconds.

We leave as future work to model check observational determinism for a larger store and more complex programs. The encoding of commands needs to be changed if we consider other variables than booleans. More complex stores or programs might give rise to state space explosion, that hampers the use of model checkers to verify properties (we have this problem with eager trace invariance, thus it will be discussed in the next section).

Our results show the feasibility of our approach : to recast verification of Non-Interference for multi-threaded programs as a model checking problem. In particular, they confirm that this can be achieved using self-composition.

5.3 Model checking of eager trace invariance

The CWB model of our program semantics for eager trace invariance is almost the same as the model for observational determinism, therefore we will only specify in what the two models differ. In particular, we will not describe how we have implemented the commands.

5.3.1 Implementation of the store

The modal μ-calculus characterization of eager trace invariance does not use any eq_x predicate. Thus we do not need the

$$\text{'output-value-}x\text{-}v\text{-m}i$$

actions anymore, which had been introduced for observational determinism.

The set of internal actions is

set InternalActions-mi =
{ read-x-v-mi, write-x-v-mi | $x \in \mathit{Store} \wedge v \in \mathit{dom}(x)$} ∪
{ takeLock-mi, releaseLock-mi};

and the agents of the store are defined as follows :

agent Xv-mi = 'read-x-v-mi.Xv-mi +
{write-x-v_j-mi.Xv_j-mi | $v_j \in \mathit{dom}(x)$ }

5.3.2 Different action labels

The actions corresponding to the labelled transitions are different in the two characterizations. We still need to duplicate actions in a "begin" and a "end" action. We also need to know if the "begin" action is a low or high action. Therefore we define the sets ProgressLowActions-mi and ProgressHighActions-mi as

set ProgressLowActions-mi =
{ 'output-begin-write-x-value-v-mi | $x \in \mathit{Store}_L \wedge v \in \mathit{dom}(x)$} ∪
{output-end-write-mi} ∪
{ 'output-begin-read-low-mi} ∪
{ 'output-end-read-x-value-v-mi | $x \in \mathit{Store}_L \wedge v \in \mathit{dom}(x)$}
set ProgressHighActions-mi =
{ 'output-begin-write-x-value-v-mi | $x \in \mathit{Store}_H \wedge v \in \mathit{dom}(x)$} ∪
{output-end-write-mi} ∪
{ 'output-begin-read-high-mi} ∪
{ 'output-end-read-x-value-v-mi | $x \in \mathit{Store}_L \wedge v \in \mathit{dom}(x)$}

5.3.3 Abstracting high actions in modalities

Formula $mimic_{3,1}$ introduced in Section 4.3.4 uses modalities $\langle\!\langle a_l \rangle\!\rangle_H$ and $[\![a_l]\!]_H$ that abstract away from high actions. We have to rewrite these formulae in the more

standard modal μ-calculus language of CWB, *i.e.* with basic and weak modalities only. We use $\texttt{begin}(a_l)\texttt{-m}i$ and $\texttt{end}(a_l)\texttt{-m}i$ to denote the two actions belonging to ProgressLowActions-mi that correspond to a_l (for instance, 'output-begin-write-x-value-v-mi and output-end-write-mi). Property $\langle\!\langle a_l \rangle\!\rangle_H \, \phi$ can be expressed for model mi as

Possible-High-mi(
$\langle\!\langle \texttt{begin}(a_l)\texttt{-m}i \rangle\!\rangle \, \langle\!\langle \texttt{end}(a_l)\texttt{-m}i \rangle\!\rangle$
Possible-High-mi(ϕ))

and $[\![a_l]\!]_H \, \phi$ as

Always-High-mi(
$[\![\texttt{begin}(a_l)\texttt{-m}i]\!] \, [\![\texttt{end}(a_l)\texttt{-m}i]\!]$
Always-High-mi(ϕ))

where Possible-High-mi and Always-High-mi are defined as follows :

prop Possible-High-mi(Phi) =
  min(Y. Phi | $\langle\!\langle \texttt{ProgressHighActions-m}i \rangle\!\rangle \, \langle\!\langle \texttt{ProgressHighActions-m}i \rangle\!\rangle$Y);

prop Always-High-mi(Phi) =
  max(Y. Phi & $[\![\texttt{ProgressHighActions-m}i]\!] \, [\![\texttt{ProgressHighActions-m}i]\!]$Y);

These two formulae take into account the fact that actions come by pairs (a "begin" and a "end" action). That is why we always have two successive weak modalities.

We are now ready to express $mimic_{3,1}$ for our CWB model, using the expressions given above for $\langle\!\langle a_l \rangle\!\rangle_H \, \phi$ and $[\![a_l]\!]_H \, \phi$.

prop Mimic-m3-m1 =
  max(R. $\bigwedge_{a_l \in \texttt{L-Actions}}$
  Always-High-m1(
  $[\![\texttt{begin}(a_l)\texttt{-m1}]\!] \, [\![\texttt{end}(a_l)\texttt{-m1}]\!]$
  Always-High-m1(
    Possible-High-m3(
    $\langle\!\langle \texttt{begin}(a_l)\texttt{-m3} \rangle\!\rangle \, \langle\!\langle \texttt{end}(a_l)\texttt{-m3} \rangle\!\rangle$
    Possible-High-m3(R))))

5.3.4 Eager trace invariance

We conclude with the eager trace invariance property.

prop EagerInv = [init-m1][init-m2] Phi;

prop Phi = max(S.
(⟨init-m3⟩ Mimic-m3-m1) &

$$\Big(\bigwedge_{a_h \in \texttt{L-Actions}} [\![\texttt{begin}(a_h)\texttt{-m1}]\!]\,[\![\texttt{end}(a_h)\texttt{-m1}]\!]\text{S }\Big)\ \&$$

$$\Big(\bigwedge_{a_h \in \texttt{L-Actions}} [\![\texttt{begin}(a_h)\texttt{-m2}]\!]\,[\![\texttt{end}(a_h)\texttt{-m2}]\!]\ \langle\texttt{init-m3}\rangle\ \langle\!\langle\texttt{begin}(a_h)\texttt{-m3}\rangle\!\rangle\,\langle\!\langle\texttt{end}(a_h)\texttt{-m3}\rangle\!\rangle\ \text{Psi }\Big)\ \&$$

$$\Big(\bigwedge_{a_l \in \texttt{L-Actions}} [\![\texttt{begin}(a_l)\texttt{-m1}]\!]\,[\![\texttt{end}(a_l)\texttt{-m1}]\!]\ [\![\texttt{begin}(a_l)\texttt{-m2}]\!]\,[\![\texttt{end}(a_l)\texttt{-m2}]\!]\ \langle\texttt{init-m3}\rangle\ \langle\!\langle\texttt{begin}(a_l)\texttt{-m3}\rangle\!\rangle\,\langle\!\langle\texttt{end}(a_l)\texttt{-m3}\rangle\!\rangle\ \text{Psi }\Big)$$

prop Psi = max(S.
Mimic-m3-m1 &

$$\Big(\bigwedge_{a_h \in \texttt{L-Actions}} [\![\texttt{begin}(a_h)\texttt{-m1}]\!]\,[\![\texttt{end}(a_h)\texttt{-m1}]\!]\text{S }\Big)\ \&$$

$$\Big(\bigwedge_{a_h \in \texttt{L-Actions}} [\![\texttt{begin}(a_h)\texttt{-m2}]\!]\,[\![\texttt{end}(a_h)\texttt{-m2}]\!]\ \langle\!\langle\texttt{begin}(a_h)\texttt{-m3}\rangle\!\rangle\,\langle\!\langle\texttt{end}(a_h)\texttt{-m3}\rangle\!\rangle\text{S }\Big)\ \&$$

$$\Big(\bigwedge_{a_l \in \texttt{L-Actions}} [\![\texttt{begin}(a_l)\texttt{-m1}]\!]\,[\![\texttt{end}(a_l)\texttt{-m1}]\!]\ [\![\texttt{begin}(a_l)\texttt{-m2}]\!]\,[\![\texttt{end}(a_l)\texttt{-m2}]\!]\ \langle\!\langle\texttt{begin}(a_l)\texttt{-m3}\rangle\!\rangle\,\langle\!\langle\texttt{end}(a_l)\texttt{-m3}\rangle\!\rangle\text{S }\Big)$$

5.3.5 Results

We failed to have results for eager trace invariance (checking property EagerInv takes too much time). This is due to the so-called state space explosion problem. Model checking algorithms are resource consuming, both in terms of CPU time and memory. This can be caused by the size of the state space and the nesting depth of the formula that is checked. Here, we have both a large state space and a high nesting depth.

First, we use a triple program model to model check eager trace invariance, whereas we only need a double program model to model check observational determinism. This increases the total number of states. Note that we consider only very tiny examples, with a store limited to a few variables. The size of the state space would be larger for "real-life" examples.

Next, we use fixed point operators to abstract away from high actions in the Possible-High-mi() and Always-High-mi() formulae. This increases the nesting depth of the whole formula.

Finally, formula Phi and formula Psi are composed of many conjunctions. We believe that this branching, combined with the looping due to the maximal fixed point, causes the explosion.

Chapter 6

Conclusion

We have re-expressed several proposals for Non-Interference for multi-threaded programs found in the literature over a uniform program model. This model supports both state-based and action-based definitions.

We have divided the definitions in two groups : possibilistic and probabilistic definitions. Possibilistic definitions are weaker, since they do not consider probabilistic attacks. However, some probabilistic definitions require the scheduling policy to be specified, while most applications are not intended for a particular scheduling policy.

Most proposals are not compositional, and compositionality is always achieved at the cost of completeness. Compositionality and Non-Interference seem to be incompatible in the setting of concurrency. In particular, our intuitive understanding of Non-Interference is not compositional. However, we have seen examples that do not even allow the composition of two threads which handle only low variables. Therefore, it might be useful to consider an alternative notion of compositionality, where two secure threads can always be safely combined if this does not create interference of high variables.

We have compared the different proposals with respect to a set of examples. State-based and action-based definitions have different granularities. The definition given in terms of probabilistic bisimulation both considers probabilistic attacks and seems to match our intuition of secure information flow best. An important result is that all proposals are incomparable.

We have proposed a characterization for observational determinism and eager trace invariance in modal μ-calculus, and proved the first one correct. Our characterizations use self-composition. We have encoded our program model in CCS and used a model checker, the Concurrency Workbench, to verify Non-Interference for a set of examples. Our results confirm the feasibility of this method.

In this text, we check Non-Interference on self-composition. An alternative approach, also based on model checking, was presented in recent work by Alur *et al.* ([1], 2007). They enrich the traditional tree models for model checking with jump-

edges that capture the notion of *observational indistinguishability*. They extend traditional logics, CTL and modal μ-calculus, so that they can be interpreted over these enriched structures (called equivalence graphs) and present model checking algorithms. These logics allow them to reason about the existence of another execution which has a different observational behaviour. In particular, they give a formulation of Non-Interference which has a flavour of non-stuttering observational determinism, a variation of observational determinism we have presented in this text, which is too strong for concurrent programs. Since their method is based on observational indistinguishability, it might not be possible to use it for other proposals. Moreover, it requires to implement new model checking algorithms. Nevertheless, it provides an alternative approach to self-composition.

Future work Each proposal for Non-Interference has its limitations. We have not hit one that matches our intuition of secure information flow perfectly. Our program model can be used to propose a definition of Non-Interference as close as possible to our intuition, possibly borrowing from the proposals that we have reformulated.

It would be interesting to characterize and verify other security properties by means of temporal logic. Integrity, the dual of confidentiality, specifies how information is allowed to flow to upper levels (*i.e.* it prevents from malicious and accidental altering of data). A characterization for integrity is likely to require a logic as expressive as for confidentiality. Thus our approach could be extended to this property as well.

Finally, we have not been able to characterize any of the probabilistic Non-Interference properties with the traditional temporal logics. An active area of research focusing on bisimulations has led to the development of a set of tools to reason about bisimulations. Such tools could be used to enforce Non-Interference in terms of probabilistic bisimulation. We leave as future work to find precise and automated techniques for verification of probabilistic Non-Interference properties.

Bibliography

[1] R. Alur, P. Cerny, and S. Chaudhuri. Model checking on trees with path equivalences. In *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 664–678. Springer-Verlag, 2007.

[2] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Symposium on Security and Privacy*, May 2007.

[3] G. Barthe, P. D'Argenio, and T. Rezk. Secure Information Flow by Self-Composition. In R. Foccardi, editor, *Computer Security Foundations Workshop*, pages 100–114. IEEE Press, 2004.

[4] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theor. Comput. Sci.*, 281(1-2):109–130, 2002.

[5] J. Bradfield and C. Stirling. Modal logics and mu-calculi: an introduction, 2001.

[6] R. Cleaveland, P. Iyer, and M. Narasimha. Probabilistic temporal logics via the modal mu-calculus. In *Foundations of Software Science and Computation Structure*, pages 288–305, 1999.

[7] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[8] E. Emerson and J. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 169–180, New York, NY, USA, 1982. ACM.

[9] R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. *SIGPLAN Not.*, 39(1):186–197, 2004.

[10] J. Goguen and J. Meseguer. Security policies and security models. In *Symposium on Operating Systems Principles*, pages 11–22. IEEE Press, 1982.

[11] M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *Computer Security Foundations Workshop*, 2006.

[12] R. Joshi and K.R.M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1-3):113–138, 2000.

[13] M. Mukund. Linear-time temporal logic and Büchi automata. 1997.

[14] A. Roscoe. CSP and determinism in security modelling. In *Symposium on Security and Privacy*, pages 114–127. IEEE Computer Society Press, 1995.

[15] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication*, 21:5–19, January 2003.

[16] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *IEEE Computer Security Foundations Workshop*, pages 200–215, Cambridge, UK, July 2000. IEEE Computer Society.

[17] G. Smith and D. Volpano. Secure Information Flow in a Multi-threaded Imperative Language. In *Principles of Programming Languages*, pages 355–364, San Diego, California, January 1998.

[18] D. Volpano and G. Smith. Confinement properties for programming languages. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 29, 1998.

[19] S. Zdancewic and A.C. Myers. Observational determinism for concurrent program security. In *IEEE Computer Security Foundations Workshop*, pages 29–43, Pacific Grove, California, USA, June 2003. IEEE Press.