Formally Verifying Information Flow Type Systems for Concurrent and Thread Systems*

Gilles Barthe[1] and Leonor Prensa[2] **

[1] INRIA Sophia-Antipolis, France Gilles.Barthe@inria.fr
[2] LORIA, France Leonor.Prensa@loria.fr

Abstract. Information flow type systems provide an elegant means to enforce confidentiality of programs. Using the proof assistant Isabelle/HOL, we have machine-checked a recent work of Boudol and Castellani [4], which defines an information flow type system for a concurrent language with scheduling, and shows that typable programs are non-interferent. As a benefit of using a proof assistant, we are able to deal with a more general language than the one studied by Boudol and Castellani, and to solve a (minor) flaw in the original proof. The development constitutes to our best knowledge the first machine-checked account of non-interference for a concurrent language.

1 Introduction

Background Security models for mobile and embedded code, such as the Java Virtual Machine and the Common Language Runtime, partially guarantee the innocuity of downloaded applications, but are too weak to enforce strong security. For example, these security models ensure that applications will not perform illegal memory accesses, but fail to guarantee that confidential data shall not be accessed by an unauthorized party. In fact, conventional mechanisms such as static bytecode verification and access control mechanisms are not appropriate to prevent attacks by untrusted code, and platforms for mobile and embedded code should be endorsed with appropriate mechanisms that guarantee end-to-end security. One such mechanism for confidentiality is provided by information flow type systems, which track the flow of information in a program execution, and provide a means to enforce statically that no information leakage will result from executing the program, see [16] for a recent survey. In fact, such type systems guarantee non-interference [5], a high-level security property that characterizes programs whose execution does not reveal information about secret data. More formally, non-interference for a program P assumes a separation between secret inputs and public inputs on the one hand, and secret outputs and public outputs on the other hand, and requires that the value of public outputs does not depend on the value of public inputs.

* Work partially supported by the IST Project Profundis.
** Most of this work was performed while at INRIA Sophia-Antipolis.

While information flow type systems provide an effective means to enforce end-to-end confidentiality for sequential programs, concurrency raises a number of subtle issues, starting from the definition of non-interference in a concurrent setting. For example, several alternative proposals have been suggested to handle non-determinism, including possibilistic non-interference [4, 26] which considers the set of possible outputs of a program execution (instead of the output in a sequential setting), and probabilistic non-interference [18, 20, 24], which considers the probability distribution for the possible outputs of a program execution. Similarly for scheduling, possibilistic and probabilistic definitions co-exist, but further options arise: one can opt for scheduler independent security, as e.g. in [18], or extend the programming language with primitives for scheduling and synchronization, as e.g. in [4, 15]. A number of other approaches have been pursued in the context of the π-calculus [19], e.g. by Honda, Yoshida and co-workers [8, 9], by Hennessy and Riely [6], and by Pottier [12], and more recently by Zdancewic and Myers [27] in the context of a concurrent λ-calculus.

Formal proofs of non-interference Non-interference for sequential and concurrent languages is very intricate, both in the definition of the type system used for the analysis, and in the soundness proof which establishes that typable programs are non-interferent. It is therefore natural to resort to proof assistants for managing the complexity of the definitions and proofs involved in establishing non-interference for languages. Yet most works on machine-checked type systems focus on "traditional" type systems. In contrast, few formalizations focus on language-based non-interference and in particular, we do not know of any machine-checked proof of non-interference for a concurrent programming language. Existing works include a formalization of unwinding theorems for intransitive non-interference by Rushby [14], and recent formalizations by Naumann (as yet unpublished) and by Strecker [22] of an information flow type system for a fragment of Java [2, 1] in PVS and Isabelle respectively.

Our work The purpose of this paper is to report on an experiment with the mechanical verification of a type system for a concurrent language with scheduling, using the Isabelle proof assistant [11]. The programming language is an extension of a simple WHILE language with:

- a parallel composition operator endorsed with an interleaving semantics;
- primitive operators for scheduling and synchronization.

Our language is inspired from [4], but features unrestricted sequential composition, whereas [4] considers requires that the first process of a sequential composition is sequential.

The type system is also taken from [4], but we show its validity for our more general language, using their tools and definitions, including various notions of bisimulations. Our work not only indicates that proof assistants are mature for verifying state-of-the-art type systems for information flow, but also shows that proof assistants help reduce the complexity of proofs and thereby allow to discard convenient, but unnecessary assumptions in proofs. Perhaps less importantly, our work reveals a few minor imprecisions in the definitions and proofs of [4].

Contents of the paper The remainder of the paper is organized as follows: Section 2 briefly reviews the main issues in defining non-interference in a concurrent setting. Section 3 gives an introduction to Isabelle/HOL. Section 4.1 presents the concurrent programming language, and its associated type system. Section 4.2 establishes non-interference of the concurrent language (as a result of allowing for unrestricted sequential composition, we do not need to prove sequential non-interference prior to concurrent non-interference). In Section 5.1, we extend the language with scheduling primitives, and endow it with appropriate typing rules. Section 5.2 is devoted to a proof of non-interference for the extended language. We conclude in Section 6 with directions for further work. Due to lack of space we cannot show all the definitions and proofs. A technical report containing a detailed description is in preparation.

2 A primer on non-interference

Non-interference assumes a separation between the secret inputs and the public inputs of a program, and requires that the program execution does not result in any observable information that may reveal, even partially, the secret inputs. The definition of non-interference may be made precise in a number of ways, depending on the notion of observable information one adopts. In its simplest instance, non-interference takes the program public outputs as the sole observable information about the program; this definition of non-interference is known as termination-insensitive non-interference (termination of the program is not considered as observable), and is the one used for instance in a sequential setting by Volpano, Smith and Irvine in their seminal paper [25], and later in a concurrent setting by Volpano and Smith [21]. There are a number of variations of the definition, for example termination-sensitive non-interference [23], where the termination of the program is considered as observable, or bisimulation-based non-interference, where intermediate results of the computation are considered as observable. Boudol and Castellani [4] among others argue that such a notion of non-interference is more appropriate in a concurrent setting, where a thread can observe the behavior of other threads and adapts its behavior accordingly. Consequently, they define a program P to be non-interferent if $P \approx P$, where $\approx$ is the largest bisimulation, for some appropriate notion of bisimulation on programs. In order to enforce statically such a notion of non-interference for concurrent programs, they propose a refined type system that keeps track of the level of loop guards, as well as the level of assignments like in the work of Volpano and Smith. Their proof of non-interference relies on exhibiting an appropriate bisimulation relation $\mathcal{R}$ such that $P \mathrel{\mathcal{R}} P$ for every typable program P.

Scheduling introduces further considerations as regards non-interference. One solution is to assume a non-deterministic scheduler, as in [21], but such an approach is subject to refinement attacks. Other solutions involve probabilistic non-interference, e.g. later work by Volpano and Smith, and by Sabelfeld and Sands [18]. However, Boudol and Castellani [4] argue that scheduling policies should be encoded in the program, a scenario that has been suggested independently in the context of embedded systems [7]. In such an approach, scheduling policies are represented by a concurrent program that is type-checked using the same rules as other concurrent programs.

3 Isabelle/HOL

Isabelle [11] is a generic interactive theorem prover which can be instantiated with different object logics and Isabelle/HOL is the instance for Higher-Order Logic. From now on, Isabelle means Isabelle/HOL. Extensive documentation can be found on the Web at http://isabelle.in.tum.de. We do not assume that the reader is already familiar with Isabelle and summarize the relevant notation below.

We use two base types (*bool* and *nat*) and construct others by type constructors like *list*, *set* or the product type ($\times$) and by function application ($\Rightarrow$). Isabelle also supports inductive definitions of data types (keyword **datatype**). The product type $\alpha \times \beta$ comes with the projection functions *fst* and *snd*. Tuples are pairs nested to the right, e.g. $(a,\ b,\ c) = (a,\ (b,\ c))$. They may also be used as patterns like in $\lambda(x, y).\ f\ x\ y$. List notation is similar to ML. The ith component of a list xs is written $xs\,!\,i$. The functional $map :: (\alpha \Rightarrow \beta) \Rightarrow \alpha\ list \Rightarrow \beta\ list$ applies a function to all elements of a list. The syntax $xs[i := x]$ denotes the list xs with the ith component replaced by x.

Isabelle distinguishes between object level ($\longrightarrow$) and meta-level ($\Longrightarrow$) implication, and similarly for universal quantification, but this distinction is unimportant for our purposes. The notation $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ represents an implication with assumptions $A_1, \ldots, A_n$ and conclusion A.

All functions must be explicitly declared (keyword **consts**). Non-recursive definitions are defined via the meta-equality ($\equiv$), recursive ones are introduced by **primrec**. The rules for inductively defined sets are introduced by **inductive**.

The statements we prove are preceded by **lemma** or **theorem**, with no formal difference between them. Isabelle provides powerful proof tactics based on rewriting and the classical reasoner, which implements a tableau based prover for predicate logic and sets. These automatic tactics must be supplied with information about which definitions to expand and which lemmas to use.

4 Non-interference for Concurrent Systems

4.1 The Language and Type System

We start by defining the memory as a mapping from locations to values. The type of locations and values is left unspecified. We can introduce new types in Isabelle by a *type declaration*, which merely introduces its name.

typedecl *locs*
typedecl *val*
types *memory* = *locs* $\Rightarrow$ *val*

Processes are built up from arithmetic and boolean expressions. Following [4, Assumption 3.2], we assume that expressions always evaluate to a result.

types *aexp* = *memory* $\Rightarrow$ *val*
types *bexp* = *memory* $\Rightarrow$ *bool*

The program syntax is defined via a **datatype** definition. Our language is more general than the one presented in [4], where programs of the form $(P \parallel Q);; R$ are not allowed.

datatype *par* =
Skip (**skip**)
| *Assign* *locs aexp* (- :== -)
| *Seq* *par par* (-;; -)
| *Cond* *bexp par par* (**if** - **then** - **else** -)
| *While* *bexp par* (**while** - **do** -)
| *Par* *par par* (- $\parallel$ -)

Enclosed in parenthesis we give concrete syntax for each construct. We use :== for assignments and ;; for sequential composition to avoid clashes with the predefined := and ; of Isabelle.

Semantics of commands is defined inductively via transition rules between configurations. A configuration is a pair (P, μ) where P is a program and μ is the memory. We use a readable infix syntax for transitions rules. This is easily done in Isabelle by declaring the syntax and the corresponding translation. [1]

consts *exec*1 :: $((par \times memory) \times (par \times memory))$ *set*
inductive *exec*1
Assign: $(x :== a, \mu) \longrightarrow_1 (\textsf{skip}, \mu[x \mapsto (a\ \mu)])$
*Seq*1: $(P, \mu) \longrightarrow_1 (P', \mu') \Longrightarrow (\textsf{skip};;\ P, \mu) \longrightarrow_1 (P', \mu')$
*Seq*2: $(P, \mu) \longrightarrow_1 (P', \mu') \Longrightarrow (P;;\ Q, \mu) \longrightarrow_1 (P';;\ Q, \mu')$
CondT: $b\ \mu \Longrightarrow (\textsf{if}\ b\ \textsf{then}\ P\ \textsf{else}\ Q, \mu) \longrightarrow_1 (P, \mu)$
CondF: $\neg b\ \mu \Longrightarrow (\textsf{if}\ b\ \textsf{then}\ P\ \textsf{else}\ Q, \mu) \longrightarrow_1 (Q, \mu)$
WhileT: $b\ \mu \Longrightarrow (\textsf{while}\ b\ \textsf{do}\ P, \mu) \longrightarrow_1 (P;;\ \textsf{while}\ b\ \textsf{do}\ P, \mu)$
WhileF: $\neg b\ \mu \Longrightarrow (\textsf{while}\ b\ \textsf{do}\ P, \mu) \longrightarrow_1 (\textsf{skip}, \mu)$
ParL: $(P, \mu) \longrightarrow_1 (P', \mu') \Longrightarrow (P \parallel Q, \mu) \longrightarrow_1 (P' \parallel Q, \mu')$
ParR: $(Q, \mu) \longrightarrow_1 (Q', \mu') \Longrightarrow (P \parallel Q, \mu) \longrightarrow_1 (P \parallel Q', \mu')$

The function $\mu[l \mapsto v]$ for memory update is defined as $\lambda x.$ *if* $x{=}l$ *then* v *else* μ x. These transition rules describe the execution of programs step by step. This so-called *small step semantics* is the natural choice when dealing with parallelism since the interactions among the components have to be observable.

[1] Most of the definitions presented in this paper are endowed with a readable infix syntax whose formal declaration is not explicitly shown.

The types of data and expressions are *security levels*. In [4] they are modeled as elements of a lattice. For our purposes, it suffices to declare a new type *levels* as an instance of the axiomatic class *partial-order*, which is a predefined type class with three axioms: reflexivity, transitivity and antisymmetry (w.r.t. a binary relation "$\sqsubseteq$").

Type judgments are of the form $\vdash P \triangleright t\ s$, where t is a *lower bound* on the level of the *assigned variables* of P and s is the *guard type*, i.e. an *upper bound* on the level of the loop and conditional guards occurring in P. In [4] the context Γ is a mapping from variables to security levels. Our formalization leaves contexts unspecified and assumes instead the existence of a function that extracts the security level of locations.

consts $getlevel :: locs \Rightarrow levels$

The type system is relative to functions *seca* and *secb* that provide the security level of arithmetic and boolean expressions respectively, and is defined by an inductive set of inference rules.

consts $typing$:: $(par \times levels \times levels)\ set$
inductive *typing*

Skip: $\vdash \textsf{skip} \triangleright [t,\ s]$

Assign: $[\![\ seca\ a \sqsubseteq getlevel\ x;\ t \sqsubseteq getlevel\ x\ ]\!] \Longrightarrow \vdash x{:}{=}{=}a \triangleright [t,\ s]$

Seq: $[\![\vdash P \triangleright [t,\ s]; \vdash Q \triangleright [t',\ s']; s \sqsubseteq t'; t'' \sqsubseteq t; t'' \sqsubseteq t'; s \sqsubseteq s''; s' \sqsubseteq s''\ ]\!]$
$\Longrightarrow \vdash P;;\ Q \triangleright [t'',\ s'']$

Cond: $[\![\vdash P \triangleright [t,\ s]; \vdash Q \triangleright [t,\ s]; secb\ b \sqsubseteq t; secb\ b \sqsubseteq s'; s \sqsubseteq s'; t' \sqsubseteq t\ ]\!]$
$\Longrightarrow \vdash \textsf{if}\ b\ \textsf{then}\ P\ \textsf{else}\ Q \triangleright [t',\ s']$

While: $[\![\vdash P \triangleright [t,\ s]; secb\ b \sqsubseteq t; s \sqsubseteq t; secb\ b \sqsubseteq s'; s \sqsubseteq s'; t' \sqsubseteq t\ ]\!]$
$\Longrightarrow \vdash \textsf{while}\ b\ \textsf{do}\ P \triangleright [t',\ s']$

Par: $[\![\vdash P \triangleright [t,\ s]; \vdash Q \triangleright [t,\ s]\ ]\!] \Longrightarrow \vdash P \parallel Q \triangleright [t,\ s]$

To simplify proofs, we formalize a syntax-directed set of rules which includes the necessary subtyping relations. This avoids the use of the meet and join operators as well as dealing with the subtyping rule proper. However, the subtyping rule is derivable.

lemma *Subtyping*: $[\![\vdash P \triangleright [t,\ s]; t' \sqsubseteq t; s \sqsubseteq s'\ ]\!] \Longrightarrow \vdash P \triangleright [t',\ s']$

The definition of non-interference presupposes a set of low security levels which is *downward-closed*:

consts $L :: levels \Rightarrow bool$
axioms *Ldown*: $[\![\ L\ x';\ x \sqsubseteq x'\ ]\!] \Longrightarrow L\ x$

Equality on memories is defined relative to L.

$\mu \simeq \mu' \equiv \forall x.\ L\ (getlevel\ x) \longrightarrow \mu\ x{=}\mu'\ x$

In the sequel we assume that the security level of expressions is correct in the sense that evaluating a low expression with low equal memories should yield the same result. Such an assumption corresponds to [4, Assumption 3.3.].

axioms
$\quad$ *beh-aexp*: $[\![\ \mu \simeq \mu'; L\ (seca\ a)\ ]\!] \Longrightarrow (a\ \mu){=}(a\ \mu')$
$\quad$ *beh-bexp*: $[\![\ \mu \simeq \mu'; L\ (secb\ b)\ ]\!] \Longrightarrow (b\ \mu){=}(b\ \mu')$

4.2 Properties of Typed Programs

A program is *typable* if there exist types such that a typing judgment can be derived in the system.

$$typable\ P \equiv \exists\, t\ s.\ \vdash P \rhd [t,\ s]$$

In this section we formally define and prove some properties of typable programs. The final goal is to establish that they are secure in the sense of non-interference. Some preliminary lemmas are needed. The first one states that types are preserved along execution.

lemma *subject-reduction*: $[\![\ \vdash P \rhd [t,\ s]\ ;\ (P,\ \mu) \longrightarrow_1 (P',\ \mu')\ ]\!] \Longrightarrow\ \vdash P' \rhd [t,\ s]$

Following [4], we use various notions of bisimulations to state and reason about non-interference. First we define the relation $\twoheadrightarrow$ as the reflexive closure of $\longrightarrow_1$. It is defined as a set of pairs of configurations inductively generated by two rules:

$\quad$ *execr-refl*: $cf \twoheadrightarrow cf$
$\quad$ *execr-inj*: $cf \longrightarrow_1 cf' \Longrightarrow cf \twoheadrightarrow cf'$

We say that P' is a *derivative* of P, written $P \leadsto P'$, if for some μ and μ' we have $(P,\ \mu) \longrightarrow^* (P',\ \mu')$, where $\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow_1$. This relation is defined inductively:

$\quad$ *der-refl*: $P \leadsto P$
$\quad$ *der-step*: $[\![\ (P,\ \mu) \longrightarrow_1 (P',\ \mu');\ P' \leadsto Q\ ]\!] \Longrightarrow P \leadsto Q$

(Semantically) high programs are programs that never modify the low part of the memory. We formalize this definition and prove some useful properties:

$$ship\ P \equiv \forall\, P'\ Q'\ \mu\ \mu'.\ P \leadsto P' \longrightarrow (P',\ \mu) \longrightarrow_1 (Q',\ \mu') \longrightarrow \mu \simeq \mu'$$

lemma *ship-sr*: $[\![\ (P,\ \mu) \longrightarrow_1 (P',\ \mu');\ ship\ P\ ]\!] \Longrightarrow ship\ P'$
lemma *ship-skip*: *ship* **skip**
lemma *ship-seq*: $[\![\ ship\ P;\ ship\ Q\ ]\!] \Longrightarrow ship\ (P;;\ Q)$

The predicate *bis* defines when a relation R on configurations is a *bisimulation*.

$$\begin{aligned} bis\ R \equiv\ & sym\ R \wedge (\forall\, P\ Q.\ (P,\ Q) \in R \longrightarrow snd\ P \simeq snd\ Q \\ & \wedge (\forall\, P'.\ P \longrightarrow_1 P' \longrightarrow (\exists\, Q'.\ Q \twoheadrightarrow Q' \wedge (P',\ Q') \in R))) \end{aligned}$$

where *sym* R means that the relation R is symmetric. The domain of bisimulations characterizes *secure* programs.

$secure\ P \equiv\ \exists S.\ (P,\ P) \in S \wedge bis\ \{((P,\ \mu),\ (Q,\ \nu)).\ (P,\ Q) \in S \wedge \mu \simeq \nu\}$

The first result establishes bisimilarity of high programs using the relation S_0.

constdefs $S0 :: (par \times par)\ set$
$S0 \equiv \{(P,\ Q).\ ship\ P \wedge ship\ Q\}$

We define the corresponding relation R_0 between configurations and prove that it is a bisimulation.

constdefs $R0 :: ((par \times memory) \times (par \times memory))\ set$
$R0 \equiv \{((P,s),\ (Q,t)).\ P\ S_0\ Q \wedge s \simeq t\}$

lemma *R0-is-bis*: $bis\ R0$

Two stronger notions of bisimulation, namely *quasi-strong* and *strong* bisimulations, are defined in [4]. The first one is needed to prove non-interference for the sequential sublanguage. Thanks to our generalization of the language we do not need this definition to prove non-interference of parallel programs. However, both of them will be necessary in section 5.2 when we prove non-interference for scheduled thread systems.

We define *bounded* and *guarded* programs and show some useful properties.

$bounded\ P \equiv \forall t\ s.\ \vdash P \triangleright [t,\ s] \longrightarrow L\ t$
$guarded\ P \equiv \exists t\ s.\ \vdash P \triangleright [t,\ s] \wedge (L\ s)$

A program which is not bounded cannot write on variables of low level, and therefore such a program is high.

lemma *notbounded-ship*: $\neg\ bounded\ P \Longrightarrow ship\ P$

However, a high program is not necessarily not bounded (see counterexample in [4]). The next lemma characterizes the behaviour of guarded programs.

lemma *behaviour-of-guarded-programs*:
$[\![guarded\ P;\ \mu \simeq \nu;\ (P,\ \mu) \longrightarrow_1 (P',\ \mu')]\!] \Longrightarrow \exists \nu'.\ (P,\ \nu) \longrightarrow_1 (P',\ \nu') \wedge \mu' \simeq \nu'$

A relation S_2 on parallel programs is defined inductively by the following rules:

consts $S2 :: (par \times par)\ set$
inductive $S2$
clause1: $[\![\ ship\ P;\ ship\ Q;\ typable\ P;\ typable\ Q\ ]\!] \Longrightarrow P\ S_2\ Q$
clause2: $[\![\ bounded\ P;\ typable\ P\ ]\!] \Longrightarrow P\ S_2\ P$
clause3: $[\![\ P\ S_2\ Q;\ \neg\ bounded\ R;\ typable\ (P;;\ R);\ typable\ (Q;;\ R)\ ]\!]$
$\Longrightarrow (P;;\ R)\ S_2\ (Q;;\ R)$
clause4: $[\![\ P1\ S_2\ P2;\ Q1\ S_2\ Q2;\ typable\ (P1 \parallel Q1);\ typable\ (P2 \parallel Q2)\ ]\!]$
$\Longrightarrow (P1 \parallel Q1)\ S_2\ (P2 \parallel Q2)$
clause5: $P\ S_2\ Q \Longrightarrow (\mathsf{skip};;\ P)\ S_2\ Q$
clause6: $P\ S_2\ Q \Longrightarrow P\ S_2\ (\mathsf{skip};;\ Q)$

A minor imprecision in the proof of [4] is solved by adding rules *clause5* and *clause6* to the original definition of S_2. The following three properties are used in the proof of non-interference:

lemma *S2-sym*: $P\ S_2\ Q \Longrightarrow Q\ S_2\ P$
lemma *S2-refl*: $\mathit{typable}\ P \Longrightarrow P\ S_2\ P$
lemma *S2-skip-ship*: $\mathsf{skip}\ S_2\ P \Longrightarrow \mathit{ship}\ P$

The proofs of the first two lemmas follow [4]. Due to the added rules the proof of the third lemma needs rule induction whereas in [4] case distinction was sufficient. We now prove the non-interference result for programs of type *par*.

theorem *Concurrent-Noninterference*: $\mathit{typable}\ P \Longrightarrow \mathit{secure}\ P$

By instantiating with S_2 we obtain the following subgoals:

1. $\mathit{typable}\ P \Longrightarrow P\ S_2\ P$
2. $\mathit{typable}\ P \Longrightarrow \mathit{bis}\ \{((P,\ \mu),\ Q,\ \nu).\ P\ S_2\ Q \wedge \mu \simeq \nu\}$

The first subgoal is solved using lemma *S2-refl*. It remains to prove that the relation R_2 (the extended relation of S_2 for configurations) is a bisimulation.

lemma *R2-is-bis*: $\mathit{bis}\ R2$

This follows from the auxiliary lemma on which we apply induction on the derivation of S_2:

lemma *R2-is-bis-aux*: $P\ S_2\ P' \Longrightarrow$
$\forall \mu\ \mu'\ Q.\ \mu \simeq \mu' \longrightarrow (P,\ \mu) \longrightarrow_1 Q \longrightarrow (\exists\, Q'.\ (P',\ \mu') \twoheadrightarrow Q' \wedge Q\ R_2\ Q')$

We show the step revealing the need for rules *clause5* and *clause6*. This case corresponds to the rule *clause2*:

$$\forall \mu\ \mu'\ Q.\ \mathit{bounded}\ P \longrightarrow \mathit{typable}\ P \longrightarrow \mu \simeq \mu'$$
$$\longrightarrow (P,\ \mu) \longrightarrow_1 Q \longrightarrow (\exists\, Q'.\ (P,\ \mu') \twoheadrightarrow Q' \wedge Q\ R_2\ Q')$$

We proceed by induction on the structure of P. Consider the case $P{=}P1;;\ P2$. By rule inversion on the transition $(P1;;\ P2,\ \mu) \longrightarrow_1 Q$ we obtain two cases. The problem appeared for $P1{=}\mathsf{skip}$:

$$\llbracket\ \mathit{bounded}\ P2 \longrightarrow \mathit{typable}\ P2 \longrightarrow$$
$$(\forall \mu\ \mu'.\ \mu \simeq \mu' \longrightarrow (\forall\, a\ b.\ (P2,\ \mu) \longrightarrow_1 (a,\ b) \longrightarrow$$
$$(\exists\, a'\ b'.\ (P2,\ \mu') \twoheadrightarrow (a',\ b') \wedge (a,\ b)\ R_2\ (a',\ b'))));$$
$$\mathit{bounded}\ (\mathsf{skip};;\ P2);\ \mathit{typable}\ (\mathsf{skip};;\ P2);\ \mu \simeq \mu';\ (P2,\ \mu) \longrightarrow_1 (P',\ \mu'')\ \rrbracket$$
$$\Longrightarrow \exists\, a\ b.\ (\mathsf{skip};;\ P2,\ \mu') \twoheadrightarrow (a,\ b) \wedge (P',\ \mu'')\ R_2\ (a,\ b)$$

Obviously we need to use the induction hypothesis. To prove $\mathit{bounded}\ P2$ we proceed like in [4]. That is, we assume that $\neg\ \mathit{ship}\ (\mathsf{skip};;\ P2)$ (otherwise the proof reduces to the case for *clause1*), hence we have $\neg\ \mathit{ship}\ P2$ by *ship-seq* and *ship-skip*. Using *notbounded-ship*, we have $\mathit{bounded}\ P2$. It is easy to prove that $\mathit{typable}\ (P;;\ Q) \Longrightarrow \mathit{typable}\ P \wedge \mathit{typable}\ Q$. So, we obtain $(P2,\ \mu') \twoheadrightarrow (a,\ b) \wedge (P',\ \mu'')\ R_2\ (a,\ b)$ for some pair $(a,\ b)$. By case analysis on $(P2,\ \mu') \twoheadrightarrow (a,\ b)$ we encounter, for the reflexive case, the following situation:

$$\llbracket \mathit{bounded}\ (\mathsf{skip};;\ P2);\ \mathit{typable}\ (\mathsf{skip};;\ P2);\ \mu \simeq \mu';\ \neg\ \mathit{ship}\ (\mathsf{skip};;\ P2);$$
$$(P2,\ \mu) \longrightarrow_1 (P',\ \mu'');\ (P',\ \mu'')\ R_2\ (P2,\ \mu')\rrbracket$$
$$\Longrightarrow \exists\, a\ b.\ (\mathsf{skip};;\ P2,\ \mu') \twoheadrightarrow (a,\ b) \wedge (P',\ \mu'')\ R_2\ (a,\ b)$$

The only possibility to prove the first part of the conclusion is using the rule *execr-refl*. It remains to prove that $P'\ S_2$ (**skip**;; $P2$) which required adding the clause *clause*6 (and *clause*5 to preserve symmetry). The case for *execr-inj* is easy.

The rest of the proof follows the guidelines given in [4]. We proved a total of 44 lemmas and needed 350 lines (interactions with the theorem prover). These numbers are relatively small considering that the original proof uses 4 fully written pages.

5 Non-interference for Scheduling Sequential Programs

5.1 The Language and Type System

We consider parallel execution of sequential threads controlled by a scheduler: $Sched\ [\![T_1,\ldots,\ T_n]\!]$. The scheduler is a parallel program of type *par* and each thread is a program of the following type:

datatype $thread = When\ \ bexp\ \ seq$ (**when** - **do** -)

The type *seq* of sequential programs is defined like parallel programs in the previous section without the parallel construct. A thread T is thus a sequential program S with a guard b and concrete syntax **when** b **do** S. The execution of S is allowed to proceed, for one step, when the condition b holds, i.e. execution can be triggered and suspended by the scheduler. A controlled thread system has the following type:

datatype $control = Control\ \ par\ \ (thread\ list)$ (-[\![-]\!])

A controlled thread system $P[\![T]\!]$ is legal iff the variables written in P are disjoint from the variables written in T.

In [4], parallelism of threads is defined via a binary operator ($\|$), which allows us to express parallelism of a concrete number of threads. Using lists we can also reason about parameterized parallel composition of threads. This is not relevant for the formalization of [4] but it is useful for reasoning about concrete programs. Given a thread $T(i)$ depending on a parameter i that varies between 0 and n we can formally express the list $[T(0),\ \ldots,\ T(n)]$ using the HOL function *map* and the construct $[0..n]$, which represents the list of natural numbers from 0 to n, i.e. $map\ (\lambda i.\ T(i))\ [0..n]$. Consequently, our formalization proves non-interference also for parameterized systems of threads.

The semantics of *When*-instructions is described by two rules:

consts $texec :: ((thread \times memory) \times (thread \times memory))\ set$
inductive $texec$

$$When\text{-}op1\text{: } [\![\ b\ \mu;\ (S,\ \mu) \longrightarrow_s (S',\ \mu')\ ]\!] \Longrightarrow (\textbf{when}\ b\ \textbf{do}\ S,\ \mu) \longrightarrow_t (\textbf{when}\ b\ \textbf{do}\ S',\ \mu')$$

$$When\text{-}op2\text{: } [\![\ b\ \mu;\ \neg\ (\exists\, s.\ (S,\ \mu) \longrightarrow_s s)\ ]\!] \Longrightarrow (\textbf{when}\ b\ \textbf{do}\ S,\ \mu) \longrightarrow_t (\textbf{when}\ b\ \textbf{do}\ S,\ \mu)$$

where $\longrightarrow_s$ is the transition relation for sequential programs. The first rule allows the sequential program S to proceed from μ, for one step, when the condition b holds in μ. The rule *When-op*2 is technically convenient but harmless. It simply allows to ignore a terminated thread.

To define the operational semantics rules for *Control* we need functions that calculate the set of variables that are written by a thread or by the scheduler. The function *wpar* calculates this set for parallel programs:

consts $\mathit{wpar} :: \mathit{par} \Rightarrow \mathit{locs\ set}$
primrec
$\mathit{wpar}\ (\textsf{skip}) = \{\}$
$\mathit{wpar}\ (x{:}{==}\ a) = \{x\}$
$\mathit{wpar}\ (P;;\ Q) = \mathit{wpar}\ P \cup \mathit{wpar}\ Q$
$\mathit{wpar}\ (\textsf{if}\ b\ \textsf{then}\ P\ \textsf{else}\ Q) = \mathit{wpar}\ P \cup \mathit{wpar}\ Q$
$\mathit{wpar}\ (\textsf{while}\ b\ \textsf{do}\ P) = \mathit{wpar}\ P$
$\mathit{wpar}\ (P \parallel Q) = \mathit{wpar}\ P \cup \mathit{wpar}\ Q$

The function *wseq* for sequential programs is analogous to *wpar* without the last rule. Finally, writable variables of a thread are those of the sequential body:

$\mathit{wthread}\ (\textsf{when}\ b\ \textsf{do}\ S) = \mathit{wseq}\ S$

The semantics of *Control*-instructions is defined via two rules:

consts $\mathit{cexec} :: ((\mathit{control} \times \mathit{memory}) \times (\mathit{control} \times \mathit{memory}))\ \mathit{set}$
inductive *cexec*
$\mathit{Conrol\text{-}op}1$: $[\![\ (P,\ \mu) \longrightarrow_1 (P',\ \mu');\ i{<}\mathit{length}\ T;\ (T!i,\ \mu) \longrightarrow_t (t',\ \mu'')\ ]\!]$
$\quad\Longrightarrow (P[\![T]\!],\ \mu) \longrightarrow_c (P'[\![T[i{:=}t']]\!],\ \mu''\ (\mu' \mid (\mathit{wpar}\ P)))$

$\mathit{Control\text{-}op}2$: $[\![\ (P,\ \mu) \longrightarrow_1 (P',\ \mu');\ \forall\, i{<}\mathit{length}\ T.\ \neg(\exists\, s.\ (T!i,\ \mu) \longrightarrow_t s)\ ]\!]$
$\quad\Longrightarrow (P[\![T]\!],\ \mu) \longrightarrow_c (P'[\![T]\!],\ \mu')$

The execution of *Control-op*1 returns the memory μ with the conjunction of the updates operated by P and T. This is expressed using the Isabelle function *overwrite* defined as $f(g|A) \equiv \lambda a.\ \mathit{if}\ a \in A\ \mathit{then}\ g\ a\ \mathit{else}\ f\ a$. We could also write it in the opposite order, i.e. $\mu'\ (\mu'' \mid (\mathit{wthread}\ T!i))$. Both are equivalent under the restriction that the writable variables of P and T be disjoint.

The typing rules for the new operators also include the subtyping relations in the premises. The rule for *When* has the premise $\vdash_s S \triangleright [t,\ s]$ that corresponds to the typing judgement for the sequential program.

consts $\mathit{ttyping} :: (\mathit{thread} \times \mathit{levels} \times \mathit{levels})\ \mathit{set}$
inductive *ttyping*
When: $[\![\ \vdash_s S \triangleright [t,\ s];\ \mathit{secb}\ b \sqsubseteq s';\ \mathit{secb}\ b \sqsubseteq t;\ t' \sqsubseteq t\ ]\!]$
$\quad\Longrightarrow\ \vdash_t (\textsf{when}\ b\ \textsf{do}\ S) \triangleright [t',\ s']$

consts $\mathit{ctyping} :: (\mathit{control} \times \mathit{levels} \times \mathit{levels})\ \mathit{set}$
inductive *ctyping*
$\mathit{Control}$: $[\![\ \vdash P \triangleright [t,\ s];\ \forall\, i{<}\mathit{length}\ T.\ \vdash_t T!i \triangleright [t,\ s];\ s \sqsubseteq t;\ s \sqsubseteq s';\ t' \sqsubseteq t]\!]$
$\quad\Longrightarrow\ \vdash_c P[\![T]\!] \triangleright [t',\ s']$

5.2 Properties of Typed Programs

All definitions introduced in section 4.2 are also defined for the new types *seq*, *thread* and *control* with their names preceded by s, t and c respectively.

The proof of non-interference for controlled thread systems requires two stronger notions of bisimulation: *strong* and *quasi-strong bisimulation*.

A relation R between configurations of sequential programs is a *quasi−strong bisimulation* if the following predicate holds:

constdefs

$$
\begin{aligned}
&qsbis :: (((seq \times memory) \times (seq \times memory))\ set) \Rightarrow bool\\
&qsbis\ R \equiv sym\ R \wedge (\forall P\ Q.\ (P,\ Q) \in R \longrightarrow snd\ P \simeq snd\ Q\\
&\qquad \wedge ((\forall P'.\ P \longrightarrow_s P' \longrightarrow (\exists Q'.\ Q \longrightarrow_s Q' \wedge (P',\ Q') \in R))\\
&\qquad \vee (sship(fst\ P) \wedge sship(fst\ Q))))
\end{aligned}
$$

A relation S_1 between sequential programs is inductively defined by three rules analogous to *clause*1, *clause*2 and *clause*3 of the relation S_2 in the previous section. It is then proven that the corresponding R_1 is a quasi-strong bisimulation.

lemma *R*1*-is-qsbis*: *qsbis R*1

The proof is about 300 lines and is very similar to the proof of *R*2*-is-bis* for the common three clauses.

A relation R is a *strong bisimulation* if it satisfies the conditions of a bisimulation where $\twoheadrightarrow$ is replaced by $\longrightarrow$. The formal definition for the case of programs of type *control* is:

constdefs $csbis :: (((control \times memory) \times (control \times memory))\ set) \Rightarrow bool$

$$
\begin{aligned}
&csbis\ R \equiv sym\ R \wedge (\forall P\ Q.\ (P,\ Q) \in R \longrightarrow snd\ P \simeq snd\ Q\\
&\qquad \wedge (\forall P'.\ P \longrightarrow_c P' \longrightarrow (\exists Q'.\ Q \longrightarrow_c Q' \wedge (P',\ Q') \in R)))
\end{aligned}
$$

The predicate *cqsbis* defines quasi-strong bisimulations for controlled thread systems. It is easy to prove that a strong bisimulation is also a quasi-strong bisimulation.

lemma *csbis-is-cqsbis*: $csbis\ R \Longrightarrow cqsbis\ R$

We define a relation S_3 (and the corresponding R_3) on controlled thread systems and prove that it is a strong bisimulation. For clarity we use two functions, *guard-of-when* and *seq-of-when*, which given a thread return the boolean expression and sequential program, respectively.

constdefs $S3 :: (control \times control)\ set$

$$
\begin{aligned}
&S3 \equiv \{(U,\ V).\ \exists P\ T\ T'.\ U{=}P[\![T]\!] \wedge V{=}P[\![T']\!] \wedge length\ T{=}length\ T'\\
&\qquad \wedge ctypable\ U \wedge ctypable\ V \wedge guarded\ P\\
&\qquad \wedge (\forall i{<}length\ T.\ guard\text{-}of\text{-}when\ (T!i) = guard\text{-}of\text{-}when\ (T'!i)\\
&\qquad \wedge L\ (secb(\ guard\text{-}of\text{-}when\ (T!i)))\\
&\qquad \wedge (\exists S.\ (seq\text{-}of\text{-}when\ (T!i),\ seq\text{-}of\text{-}when\ (T'!i)) \in S\\
&\qquad \wedge qsbis\ \{((P,\ \mu),\ (Q,\ \nu)).\ (P,\ Q) \in S \wedge \mu \simeq \nu\})\ )\}
\end{aligned}
$$

Then, two typable controlled thread systems U and V satisfy the relation $S3$ if they have the same number of threads, the same scheduler P, which must be a guarded program, the same (low) guards for each thread and all the sequential threads satisfy one-to-one a relation, say S, that is a quasi-bisimulation.

lemma $R3$-*is-csbis*: *csbis* $R3$

There is still one last relation:

constdefs $S4 :: (\mathit{control} \times \mathit{control})\ \mathit{set}$
  $S4 \equiv \{(U,\ V).\ (\mathit{cship}\ U \wedge \mathit{cship}\ V) \vee U\ S_3\ V\}$

The non-interference result relies on the following two lemmas. The first one establishes that the relation R_4 between configurations of controlled thread systems is a quasi-strong bisimulation.

lemma $R4$-*is-cqsbis*: *cqsbis* $R4$

The proof is easy using the previous two results. The second lemma connects typable programs and the relation S_4.

lemma $S4$-*refl*: $\mathit{ctypable}\ U \Longrightarrow U\ S_4\ U$

We are now ready to prove the main result.

theorem *ScheduledThreadSystems-Noninterference*: $\mathit{ctypable}\ U \Longrightarrow \mathit{csecure}\ U$

By instantiating with $S4$ the only tricky step that remains is the implication $\mathit{cqsbis}\ R \Longrightarrow \mathit{cbis}\ R$. This is not true in general. It holds for any relation R satisfying the following condition:

$$\forall\, P\ Q\ \mu\ \nu.\ \mathit{ship}\ P \longrightarrow \mathit{ship}\ Q \longrightarrow \mu \simeq \nu \longrightarrow ((P,\ \mu),\ (Q,\ \nu)) \in R$$

which is easily proven for R_4.

In total, the proof of this theorem takes over 600 lines, where about 500 lines are devoted to the proof of $R3$-*is-csbis*.

6 Conclusion

We have presented a machine-checked proof of non-interference for a concurrent language with scheduling. The proof follows and generalizes [4], by considering unrestricted sequential composition of processes.

Our work demonstrates that it is natural, and even benefitial, to use proof assistants in the design of advanced type systems for programming languages, even more so if the type system is to be used in a very sensitive context such as platforms for mobile code, or smartcards—early 2000, we were contacted by a leading smartcard industrial that was considering using an information flow type system in a new generation virtual machine for smartcards.

We intend to pursue our work in three directions: first, we would like to generalize our proofs by allowing `when` and control expressions arbitrarily in programs (or equivalently to collapse processes, threads, and controlled thread systems into a single inductive definition), and to adapt our proofs to other settings, including:

- reactive programming, as studied by Boudol, Castellani and Matos (as yet unpublished). The reactive language they study is close to the language presented here, and it should be relatively direct to adapt our formal proofs;
- low-level languages such as the JVM language. There is no information flow type system currently at hand for concurrent fragments of the JVM. However we have begun formalizing the information flow type system [3] for a sequential fragment of the JVM, and it would seem interesting to combine both formalizations;
- calculi of mobile processes, such as the π-calculus, for which Isabelle formalization exist, see e.g. [13]. One could use these existing formalizations as a basis to formally machine-check the correctness of information flow type systems for the π-calculus, such as those discussed in the introduction.

Second, we would like to machine-check recent results about downgrading [10] and controlled declassification [17], as standard definitions of non-interference, such as the one adopted in this paper, are often too restrictive in practice.

Third, we would like to develop dedicated tools for proving type soundness and for establishing that a relation is indeed a bisimulation. We believe that increasing proof automation is crucial to encourage machine-checked soundness proofs of information flow type systems.

References

1. A. Banerjee and D. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 2003. Submitted.
2. A. Banerjee and D. A. Naumann. Secure Information Flow and Pointer Confinement in a Java-like Language. In *Proceedings of CSFW'02*. IEEE Computer Society Press, 2002.
3. G. Barthe and T. Rezk. Secure information flow for the JVM. Manuscript, 2003.
4. G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, 2002. Preliminary version available as INRIA Research report 4254.
5. J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of SOSP'82*, pages 11–22. IEEE Computer Society, 1982.
6. M. Hennessy and J. Riely. Information flow vs. resource access in the information asynchronous pi-calculus. In U. Montanari, J. D. P. Rolim, and E. Welzl, editors, *Proceedings of ICALP'00*, volume 1853 of *Lecture Notes in Computer Science*, pages 415–427. Springer, 2000.
7. T.A. Henzinger, C.M. Kirsch, and S. Matic. Schedule-carrying code. In R. Alur and I. Lee, editors, *Proceedings of EMSOFT'03*, volume 2855 of *Lecture Notes in Computer Science*, pages 241 – 256, 2003.
8. K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In G. Smolka, editor, *Proceedings of ESOP'00*, volume 1782 of *Lecture Notes in Computer Science*, pages 180–199. Springer-Verlag, 2000.

9. K. Honda and N. Yoshida. A Uniform Type Structure for Secure Information Flow. In *Proceedings of POPL'02*, pages 81–92. ACM Press, 2002.
10. H. Mantel and D. Sands. Controlled downgrading based on intransitive (non)interference. Presented at Dagsthul Language-Based Security Seminar, 2003.
11. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
12. F. Pottier. A simple view of type-secure information flow in the pi-calculus. In *Proceedings of CSFW'02*, pages 320–330. IEEE Press, 2002.
13. C. Röckl and D. Hirschkoff. A fully adequate shallow embedding of the π-calculus in Isabelle/HOL with mechanized syntax analysis. *Journal of Functional Programming*, 2003.
14. J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report SRI-CSL-92-02, Computer Science Laboratory, SRI International, dec 1992.
15. A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proceedings of PSI'01*, volume 2244 of *Lecture Notes in Computer Science*, pages 227–241. Springer-Verlag, 2001.
16. A. Sabelfeld and A. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Comunications*, 21:5–19, January 2003.
17. A. Sabelfeld and A. Myers. A model for delimited information release. Manuscript, 2003.
18. A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001.
19. D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
20. G. Smith. A New Type System for Secure Information Flow. In *Proceedings of CSFW'01*, pages 115–125, 2001.
21. G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of POPL'98*, pages 355–364. ACM Press, 1998.
22. M. Strecker. Formal analysis of an information flow type system for MicroJava (extended version). Technical report, Technische Universität München, July 2003.
23. D. Volpano and G. Smith. A Type-Based Approach to Program Security. In M. Bidoit and M. Dauchet, editors, *Proceedings ofTAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621. Springer-Verlag, 1997.
24. D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7:231–253, November 1999.
25. D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, pages 167–187, December 1996.
26. D. M. Volpano and G. Smith. Verifying secrets and relative secrecy. In *Proceedings of POPL'00*, pages 268–276. ACM Press, 2000.
27. S. Zdancewic and A.C. Myers. Observational determinism for concurrent program security. In *Proceedings of CSFW'03*. IEEE Press, 2003.