A temporal logic characterisation of observational determinism*

Marieke Huisman
INRIA Sophia Antipolis, France

Pratik Worah
IIT Kharagpur, India

Kim Sunesen
Esterel Technologies, France

Abstract

This paper studies observational determinism, a generalisation of non-interference for multi-threaded programs. Standard notions of non-interference only consider input and output of programs, but to ensure the security of multi-threaded programs, one has to consider execution traces.

In earlier work, Zdancewic and Myers propose to consider a multi-threaded program secure when it behaves deterministic w.r.t. its public (or low) variables, i.e. traces of public variables should not depend on private (or high) variables. This property is called observational determinism. *The original definition of observational determinism still allows to reveal private data; this paper corrects this.*

The main contribution of this paper is a rephrasing of the definition of observational determinism in terms of a temporal logic. This allows to use standard model checking techniques to verify observational determinism, which has the advantage that the verification is automatic and precise. Moreover in case the verification fails, model checking can produce a counterexample. We characterise observational determinism in CTL and in the polyadic modal μ-calculus. For both logics, model checking algorithms exist.*

1 Introduction

With the emergence of Internet and other models of global computing, privacy guarantees are becoming more and more important. Typical applications such as electronic banking and health care information systems only are acceptable to users if their privacy is sufficiently guaranteed. *Non-interference* [13] is a standard notion of security that allows to preserve privacy. Basically, an application is said to be non-interfering if changes to the private (or secret) data are not reflected in the public data, in other words public data only depend on public data.

*This work was partially funded by the IST FET programme of the EC under the IST-2005-015905 Mobius project. This paper reflects only the author's views and the Community is not liable for any use that may be made of the information contained therein.

Several analyses have been developed to check for non-interference, of which type checking [25] is the most used approach. The type checking analysis has been shown to be sound, and in addition it is fully automatic. However, it is necessarily incomplete, because it cannot take context-sensitive information into account. Therefore, reformulations of the definition of non-interference have been proposed that allow to use classical logic-based verification techniques; expressing non-interference for example as a Hoare triple, see [4, 8]. Notice that this allows to combine type checking and logic-based verification.

However, the definition of non-interference only considers the input and output of an application. For concurrent and reactive systems it is often considered more appropriate to also look at the intermediate states of the application, and to require that private data are never revealed. Zdancewic and Myers (inspired by earlier work by Roscoe [21]) therefore propose a generalised notion of non-interference: *observational determinism* [26], which avoids attacks exploiting information about the thread scheduler. They propose a type system to check observational determinism.

This paper studies this notion of observational determinism further, and fine-tunes its definition. In fact, the original definition allows programs to reveal information about private data. We illustrate this on some examples, and propose an improvement.

The main contribution of this paper is the rephrasing of the definition of observational determinism in temporal logic. We present two different characterisations, one in CTL* [11], using a special non-standard synchronous composition operator, and one in the polyadic modal μ-calculus [3] (a variation of the modal μ-calculus [16]). For both logics efficient model checking algorithms exist, thus observational determinism is decidable (for finite state programs). Since these temporal logic characterisations are equivalent to the original definition, this gives a sound and complete verification method for observational determinism. Moreover, if the verification fails, the model checking procedure is able to produce a counterexample.

Related work For a general overview of techniques to verify information-flow, and in particular non-interference,

we refer to the survey paper of Sabelfeld and Myers [22].

There exist several characterisations of non-interference as a program logic problem. Darvas, Hähnle and Sands [8] use dynamic logic to express non-interference as a program property. Both Barthe, D'Argenio and Rezk [4] and Darvas *et al.* propose a Hoare logic formulation, using self-composition, *i.e.* a construction where the copy is composed with itself, and where each program copy keeps an independent memory. Terauchi and Aiken [24] combine these approaches with a type-directed program transformation, so that one is able to use automatic safety analysis tools, such as BLAST. Barthe *et al.* also present separation logic and CTL formulations of non-interference. However, in all these approaches only relations between program inputs and program outputs are considered, in contrast to our approach, which considers program traces.

Pottier and Simonet [20, 19] use a special program extension which allows them to establish properties over two program configurations. They use this to establish non-interference. Their verification is based on type checking. The apply this to reason about ML and the π-calculus.

Joshi and Leino, Amtoft, Bandhakavi and Banerjee and Dam present other logics for verifying information flow properties [15, 2, 1, 7]. They define sound and complete proof methods. Joshi and Leino define a weakest precondition calculus for their logic, while Amtoft *et al.* use a strongest postcondition, which is used to check the information flow properties algorithmically. Dam also shows that proof search for his logic is decidable. The difference with our approach is that they define a special logic, that is designed to verify the particular information flow property, while we recast the information flow property in a general logic, for which we can use existing verification techniques.

Several definitions are proposed to generalise the notion of non-interference to multi-threaded programs. We mention a few. Roscoe was the first to state the importance of determinism [21] to ensure secure information flow of multi-threaded programs. Volpano and Smith [23] propose to use either a purely nondeterministic thread scheduling, or to put (severe) restrictions on the language. Boudol and Castellani lift these restrictions by proposing a definition based on bisimulation [5]. Zdancewic and Myers [26] propose to eliminate timing channels by disallowing race conditions. Storage leakage can then be detected by a type system, which only has to consider each low variable trace independent of the other low variable traces, which puts less restrictions on the kind of programs that are still considered secure.

Overview of the paper Section 2 describes the notion of observational determinism in more detail. Next, Section 3 introduces the formal program model and defines observational determinism formally. Section 4 describes the characterisation in CTL*, while Section 5 describes the characterisation in the (polyadic) modal μ-calculus. Finally, Section 6 compares the two different approaches, and discusses the encoding of our formalisation within a model checker. It also gives suggestions for future work.

2 Security for Multi-threaded Programs

In general, a program is considered to be secure if it protects its confidential data. The basic idea behind information flow policies is that the flow of confidential data should be restricted, so that they are only visible to whoever has the appropriate permissions.

For simplicity, we consider a simple two-point security lattice: the data are divided into public and private data. However, it is straightforward to generalise the results of this paper to a general lattice with more security levels. Throughout this paper, we use the convention that variable names starting with `l` denote low variables, while variable names starting with `h` denote high variables.

Information flow policies are often based on non-interference, which is defined as a relation between the input and the output of a program: if the low input values are indistinguishable, running the program should result in indistinguishable low output values. This implicitly ensures that publicly visible results of a computation do not depend on private data. However, for multi-threaded (or concurrent) programs, this notion is not sufficient to ensure security: the outcome of the program depends on the scheduling strategy, thus different scheduling policies can actually reveal information. Moreover, concurrent programs (and in particular reactive programs) are often used in a context where intermediate states can be observed.

2.1 Observational determinism

To overcome these problems, Zdancewic and Myers propose a generalisation of non-interference, called *observational determinism* [26], that can be applied to multi-threaded programs. They observe the following:

- since intermediate states might be observed, whole execution traces of a program need to be considered, instead of only input and output states; and
- different scheduling policies should not be observable in the low variables, *i.e.* all traces of a program should be equivalent *w.r.t.* the low variables.

Zdancewic and Myers argue further that a program such as

```
l := true ‖ l := false ‖ l := h
```

should not be considered secure, according to their observations. A scheduling policy that would always execute

the last assignment (`l := h`) last would reveal information about `h`. And even a random scheduler would reveal information, because the program is more likely to terminate with `l` containing `h`, than with `l` containing $\neg$`h`.

Therefore they propose the following definition. A program is said to be *observationally deterministic* if: given two initial configurations m and m' that are indistinguishable *w.r.t.* the low variables, any two traces that start with m and m' are also indistinguishable *w.r.t.* the low variables. Below, we will define precisely when two traces are said to be indistinguishable *w.r.t.* the low variables.

2.2 Indistinguishability of traces

In security analysis one often distinguishes internal and external leakage of information. Internal leakage of information is said to happen when the (public) behaviour of a thread is influenced by private data manipulated by another thread. External leakage of information happens when an outside observer of the system can deduce information about private data by observing the behaviour of a single thread. For example, suppose you have the following thread:

```
if (h > 0) then l1 := 4; l2 := 7
           else l2 := 7; l1 := 4
```
(T_1)

This program has an information leak, *i.e.* by observing the order in which the variables `l1` and `l2` change, one can deduce information about the value of `h`. For example, if we run T_1 in parallel with T_2 we can deduce information about `h` from observing the value of the variable `l3`.

```
wait (l1 == 4);
if (l2 == 7) then l3 := l3 + 1
```
(T_2)

If `l3` has not been increased, the assignment to `l2` must have happened after the assignment to `l1`, thus `h` must have been positive (provided `l1` and `l2` were initially not equal to 4 and 7, respectively).

To avoid such information leaks, Zdancewic and Myers present a type system and they prove that if a program does not contain races on low variables, and is accepted by their type system, then it is observationally deterministic. They claim that by requiring a program to be race-free (where a race is said to happen if two accesses to the same variable can happen at the same time, and one of these two accesses is a write), this weakens the ability of a thread to observe the behaviour of another thread, and therefore it is sufficient to check that two traces are indistinguishable for each location independently. They argue that the relative ordering of two updates can only be observed by code that contains a race. Thus, traces T and T' are indistinguishable *w.r.t.* a set of low variables $\mathcal{L}$ if for any variable location $v \in \mathcal{L}$, the projections (called *location traces*) $T(v)$ and $T'(v)$ are indistinguishable. This definition of observational determinism accepts the single-threaded program T_1. But when T_1 is put in parallel with T_2, there are races on `l1` and `l2`, because the reads to `l1` and `l2` in T_2 could happen at the same moment as the assignments to them in T_1, thus they would reject this program (notice that T_1 alone is also rejected by their type system, as it contains a branching on a high variable).

The next question is when two location traces are indistinguishable. Zdancewic and Myers propose that this is the case if they are *equivalent upto stuttering and prefixing*, *i.e.* traces $[v_0, v_1, v_2]$ and $[v_0, v_0, v_1, v_1, v_2, v_3]$ would be considered indistinguishable. Zdancewic and Myers argue that prefixing is a sufficiently strong equivalence relation, as this only causes external termination leaks, which they say reveals only one bit of information.

However, we claim that allowing prefixing of location traces actually can reveal more information. To illustrate our claim, consider the following program.

```
l1 := 0; l2 := 0;
while (h > 0) {
  l1 := l1 + 1; h := h - 1};
l2: = 1
```
(T_3)

Notice that T_3 is sequential, and thus does not contain any races. For any initial configuration m (where `h` ≥ 0), the trace for location `l1` will always be equivalent (upto stuttering) to $[\texttt{l1}_m, 0, 1, 2, \ldots, \texttt{h}_m]$, while the trace for `l2` will be $[\texttt{l2}_m, 0, 0, \ldots, 0, 1, 1, \ldots]$, where the transition of `l2` to 1 signals that the loop has terminated. Thus, for any two initial configurations m and m', both the location traces $T(\texttt{l1})$ and $T'(\texttt{l1})$, and $T(\texttt{l2})$ and $T'(\texttt{l2})$ will always be equivalent upto prefixing and stuttering. According to the definitions above, the program should be considered observationally deterministic (and thus secure). However, it is clear that this program reveals more than one bit of information: the initial value of `h` is revealed as the last value in the trace of `l1` (where the transition in the trace of `l2` indicates that this value indeed corresponds exactly to `h`). Thus in particular, if we would run it in parallel with a thread:

```
wait (l2 == 1);
print(l1)
```
(T_4)

the initial value of `h` would always be printed.

Notice that also the first example, running T_1 in parallel with T_2 is an acceptable program for the definition where trace equality uses prefixing: the variable `l3` either always keeps its initial value, or it is updated once. Thus, we always have trace equality *w.r.t.* prefixing and stuttering. However, as we have argued above, the program leaks information internally, and thus should be rejected as an insecure program.

We strengthen the definition of location trace indistinguishability by requiring that location traces are equivalent

upto stuttering. This allows to define a notion of *termination insensitive* (TI) observational determinism. Moreover, by adding the condition that traces started in indistinguishable states should either always or never reach a final configuration, we can define a notion of *termination sensitive* (TS) observational determinism. These notions are formally defined in the next section. Notice that T_1 in parallel with T_2, and T_3 (possibly in parallel with T_4) will be rejected by this definition of observational determinism. (Notice that all examples are rejected by Zdancewic and Myers's type system)

The stronger definition of observational determinism will also reject some harmless programs, such as

```
l := 7 || while (true) {}
```

because we cannot ensure that the assignment to `l` will actually happen. The reason that Zdancewic and Myers proposed to use equivalence upto stuttering and prefixing could be the desire to accept programs like the one above. However, another possibility to ensure that such programs would be accepted, would be to impose a fairness condition: all enabled program steps will eventually be executed.

This example shows that the type system proposed by Zdancewic & Myers is not sound for our notion of observational determinism. However, we believe that if we would add a fairness condition to the execution model, their type system would be sound, but we did not verify this formally.

Finally, in the examples above, in a more advanced language the race conditions could easily be eliminated by adding synchronisation or declaring the variables to be *volatile* (*cf.* the Java semantics [14]), which has the implicit effect of declaring a lock that is to be obtained before any read or write access to such a variable. Thus even in the absence of race conditions, threads might be able to observe the relative ordering of two updates. Therefore, one could strengthen the definition of observational determinism further by requiring that traces are deterministic for subsets of the set of low variables (where the union of all these subsets should be the complete set).

In the remainder of this paper, we do not further discuss possible variations of the definition of observational determinism. We restrict attention to the case where location traces are considered independently, and are supposed to be equivalent upto stuttering. However, for any of these similar logical characterisations (both in CTL* and in the polyadic μ-calculus) could be given.

3 Observational Determinism Formally

3.1 Programs and Traces

The programming language that we consider in this paper is a simple while-language, extended with parallel composition $\|$ and a blocking statement `wait`[1]. Statements are defined by the following grammar, where x denotes a variable location in memory, E an expression (defined in the obvious way), and b an expression of Boolean type.

$$S \;=\; x := E \mid S; S \mid \texttt{if } (b) \texttt{ then } S \texttt{ else } S \mid \texttt{while } (b) \{ S \} \mid S \parallel S \mid \texttt{wait } (b)$$

For simplicity, we consider the program to run with a single program store μ, which is a mapping from variable locations $v, v_0, v_1, \ldots$ to values. A program configuration c is defined as the product of a statement S and a program store μ, *i.e.* $c = \langle S, \mu \rangle$, where S denotes the program that remains to be executed, and μ denotes the current program store. We use ε to denote a finished program. Figure 1 shows several of the a small step operational semantics rules for this language, where parallel composition is asynchronous (for space reasons we have left out several symmetric cases and obvious cases). Given a configuration, we use accessor functions prog and store, such that:

$$\mathsf{prog}(\langle S, \mu \rangle) = S \qquad \mathsf{store}(\langle S, \mu \rangle) = \mu$$

Next, we are ready to define program traces. To ensure that all traces are infinite, Figure 2 defines transition rules for terminated and deadlocked programs, leaving the configuration unchanged. Thus, a terminated or deadlocked program will remain in its final state forever. A program is said to be deadlocked when all its parallel programs are waiting for a condition to become true. This is defined using primitive recursion.

$$\begin{aligned}
\mathsf{deadlock}(\langle \texttt{wait } (b), \mu \rangle) &= \neg b(\mu) \\
\mathsf{deadlock}(\langle S_1; S_2, \mu \rangle) &= \mathsf{deadlock}(\langle S_1, \mu \rangle) \\
\mathsf{deadlock}(\langle S_1 \parallel S_2, \mu \rangle) &= \mathsf{deadlock}(\langle S_1, \mu \rangle) \wedge \mathsf{deadlock}(\langle S_2, \mu \rangle) \\
\mathsf{deadlock}(\langle S, \mu \rangle) &= \mathsf{ff}
\end{aligned}$$

$$\langle x := E, \mu \rangle \rightarrow \langle \varepsilon, \mu(x \mapsto E(\mu)) \rangle$$

$$\frac{\langle S_1, \mu \rangle \rightarrow \langle S_1', \mu' \rangle}{\langle S_1; S_2, \mu \rangle \rightarrow \langle S_1'; S_2, \mu' \rangle} \qquad \frac{\langle S_1, \mu \rangle \rightarrow \langle \varepsilon, \mu' \rangle}{\langle S_1; S_2, \mu \rangle \rightarrow \langle S_2, \mu' \rangle}$$

$$\langle \texttt{if } (b) \texttt{ then } S_1 \texttt{ else } S_2, \mu \rangle \rightarrow \langle S_1, \mu \rangle \quad \text{if } b(\mu)$$

$$\langle \texttt{while } (b) \{ S \}, \mu \rangle \rightarrow \langle S; \texttt{while } (b) \{ S \}, \mu \rangle \quad \text{if } b(\mu)$$

$$\frac{\langle S_1, \mu \rangle \rightarrow \langle S_1', \mu' \rangle}{\langle S_1 \parallel S_2, \mu \rangle \rightarrow \langle S_1' \parallel S_2, \mu' \rangle} \qquad \frac{\langle S_1, \mu \rangle \rightarrow \langle \varepsilon, \mu' \rangle}{\langle S_1 \parallel S_2, \mu \rangle \rightarrow \langle S_2, \mu' \rangle}$$

$$\langle \texttt{wait } (b), \mu \rangle \rightarrow \langle \varepsilon, \mu \rangle \quad \text{if } b(\mu)$$

Figure 1. Operational semantics

[1] Zdancewic and Myers present their type system for observational determinism for a language $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ [26]. The language that we use here can be encoded in this more fundamental language.

$\langle \varepsilon, \mu \rangle \rightarrow \langle \varepsilon, \mu \rangle \qquad \langle S, \mu \rangle \rightarrow \langle S, \mu \rangle \quad \text{if deadlock}\langle S, \mu \rangle$

Figure 2. Transition rules for terminated and deadlocked programs

A program trace is an infinite sequence of configurations. The first configuration in the trace corresponds to the initial configuration. The i^{th} configuration in the trace is reached by i steps as described by the operational semantics, together with the non-blocking transition rules.

Definition 1 (Program trace) *Given configuration $\langle S, \mu \rangle$, an infinite list of configurations $T = c_0, c_1, c_2, \ldots$ is a* program trace *of $\langle S, \mu \rangle$, denoted $\langle S, \mu \rangle \Downarrow T$, if and only if*

- $c_0 = \langle S, \mu \rangle$*; and*
- *for all i we have $c_i \rightarrow c_{i+1}$.*

We use $T(\mu)$ to denote the projection of a program trace to the store, and $T(v)$ to denote the location trace that is constructed by projecting $T(\mu)$ to the variable location v. We use T_i to denote the i^{th} element in the trace.

3.2 Indistinguishability

Given a set of low variables $\mathcal{L}$ we define indistinguishability of stores and configurations.

Definition 2 (Indistinguishability) *Stores μ and μ' are* indistinguishable *w.r.t. $\mathcal{L}$, denoted $\mu \approx_{\mathcal{L}} \mu'$, if and only if*

$$\forall v \in \mathcal{L}. \mu(v) = \mu'(v)$$

Configurations c and c' are indistinguishable *w.r.t. $\mathcal{L}$, denoted $c \approx_{\mathcal{L}} c'$, if and only if* $\mathsf{store}(c) \approx_{\mathcal{L}} \mathsf{store}(c')$.

To define indistinguishability of traces, we define stuttering equivalence formally. This definition uses an auxiliary notion of stuttering equivalence upto indices i and j: traces T and T' are said to be stuttering equivalent upto i and j if (1) T_i and T'_j are the same, and (2) if we take the smallest indices k and l such that the sequences $T_k, \ldots, T_i$ and $T'_l, \ldots, T'_j$ do not change, then we check that T and T' are stuttering equivalent upto $k-1$ and $l-1$. We say that T and T' are stuttering equivalent if for all i we can find a j such that T and T' are stuttering equivalent upto i and j, and we can find a j' such that T' and T are stuttering equivalent upto i and j'. Other equivalent definitions of stuttering equivalence are possible, see *e.g.* [9, 18].

Definition 3 (Stuttering equivalent) *Traces T and T' are* stuttering equivalent upto i and j, *denoted $[T, i] \sim_s [T', j]$, if and only if*

- $T_i = T'_j$*; and*
- *if* $k = \mathsf{min}(\{p \mid \forall n \in [p \ldots i]. T_n = T_i\})$ *and*
  $l = \mathsf{min}(\{p \mid \forall n \in [p \ldots j]. T'_n = T'_j\})$
  then $k = l = 0$ *or* $[T, k-1] \sim_s [T', l-1]$.

T *and* T' *are* stuttering equivalent, *denoted $T \sim_s T'$, if and only if*

- *for all i, there exists a j such that $[T, i] \sim_s [T', j]$; and*
- *for all j, there exists an i such that $[T, i] \sim_s [T', j]$.*

Notice that stuttering equivalence defines an equivalence relation, *i.e.* it is reflexive, symmetric and transitive.

Indistinguishability of traces is defined as stuttering equivalence of the location traces for all low variables.

Definition 4 (Trace indistinguishability) *Traces T and T' are* indistinguishable *w.r.t. $\mathcal{L}$, denoted $T \approx_{\mathcal{L}} T'$, if and only if*

$$\forall v \in \mathcal{L}. T(v) \sim_s T'(v)$$

When $\mathcal{L}$ is a singleton set $\{v\}$ we write $T \approx_v T'$ instead of $T \approx_{\{v\}} T'$.

Finally, we are ready to formally define observational determinism. TI observational determinism only requires trace indistinguishability, TS observational determinism also requires that if one execution terminates, the other execution also terminates.

Definition 5 (Observational determinism) *Program S is* Termination Insensitive (TI) observationally deterministic *w.r.t. $\mathcal{L}$ if and only if for all stores μ, μ' such that $\mu \approx_{\mathcal{L}} \mu'$, and for all traces T and T', we have*

$$\langle S, \mu \rangle \Downarrow T \wedge \langle S, \mu' \rangle \Downarrow T' \Rightarrow T \approx_{\mathcal{L}} T' \tag{1}$$

We say S is Termination Sensitive (TS) observationally deterministic *w.r.t. $\mathcal{L}$ if and only if for all stores μ, μ' such that $\mu \approx_{\mathcal{L}} \mu'$, and for all traces T and T', we have*

$$\begin{array}{l} \langle S, \mu \rangle \Downarrow T \wedge \langle S, \mu' \rangle \Downarrow T' \Rightarrow T \approx_{\mathcal{L}} T' \wedge \\ (\exists i. \mathsf{prog}(T_i) = \varepsilon \Leftrightarrow \exists j. \mathsf{prog}(T_j) = \varepsilon) \end{array} \tag{2}$$

Notice that termination of a trace is expressed as a property of the global program trace, not of the location traces: in the formalisation the location traces only contain the values of the variable, they do not keep track of the program that remains to be executed.

In particular, if the set of low variables is empty, TI observational determinism vacuously holds, while TS observational determinism still puts restrictions on the termination behaviour of the program: either it always terminates, or it never terminates.

4 A CTL* Characterisation of Observational Determinism

As discussed above, a common approach to check non-interference properties such as observational determinism is to use a type system. This provides an efficient, but imprecise verification technique, that might reject many correct programs. Therefore, it is desirable to have more precise verification techniques. We achieve this by characterising observational determinism as a temporal logic property for which model checking algorithms exist. Verifying whether a program is secure, *i.e.* respects the observational determinism property, thus reduces to a standard model checking problem.

We consider two temporal logics to characterise observational determinism. This section presents a characterisation in CTL* [11], the next section presents a characterisation in the (polyadic) modal μ-calculus [3]. The CTL* characterisation stays close to the phrasing of the observational determinism definition. As we will see, this results in a relatively complicated model, while the temporal formula is quite intuitive to understand. In the μ-calculus formulation our model is much simpler (and smaller), but the characterisation is more difficult to understand.

Our approach is based on self-composition of the program that is being verified (*cf.* [4]). Basically, this means that we execute the program in parallel with itself, in such a way that we can decompose the program store. The correctness condition is phrased as a property over single program states, verifying whether the two parts of the program store satisfy a particular relation. Before presenting the self-composition of programs in more detail, we first give a brief recapitulation of CTL* (following [11]).

4.1 CTL*

CTL* is a branching-time temporal logic. It contains path quantifiers, that can be combined with arbitrary linear-time formulae. CTL* is defined to subsume CTL (branching-time temporal logic) and LTL (linear-time temporal logic) [12].

The syntax of a CTL* formula is defined as follows (assuming that A is a set of atomic predicates).

Definition 6 (CTL* syntax) *A* formula *in CTL* is either a* state formula sf *or a* path formula pf*, defined by the following grammar, where* $p \in A$*.*

$$\begin{aligned} \mathsf{sf} &= p \mid \mathsf{sf} \wedge \mathsf{sf} \mid \neg\mathsf{sf} \mid \mathsf{E}\,\mathsf{pf} \mid \mathsf{A}\,\mathsf{pf} \\ \mathsf{pf} &= \mathsf{sf} \mid \mathsf{pf} \wedge \mathsf{pf} \mid \neg\mathsf{pf} \mid \mathsf{X}\,\mathsf{pf} \mid \mathsf{pf}\,\mathsf{U}\,\mathsf{pf} \end{aligned}$$

The path quantifiers E and A are inherited from CTL. Intuitively, a formula $\mathsf{E}\ \mathsf{pf}$ expresses that there exists an execution path in the program for which the path formula pf holds, while $\mathsf{A}\ \mathsf{pf}$ expresses that pf holds for all program executions. Path formulae can contain arbitrary linear-time expressions, formed using the next (X) and the until (U) operators. We use $\mathsf{F}\,p$ (eventually p) to abbreviate $(\mathsf{true}\ \mathsf{U}\ p)$ and $\mathsf{G}\,p$ (globally p) to abbreviate $\neg\mathsf{F}\,\neg p$.

$$\begin{aligned} M, s \models p &\Leftrightarrow p \in \lambda(s) \\ M, s \models \mathsf{sf}_1 \wedge \mathsf{sf}_2 &\Leftrightarrow M, s \models \mathsf{sf}_1 \text{ and } M, s \models \mathsf{sf}_2 \\ M, s \models \neg\mathsf{sf} &\Leftrightarrow \text{not } M, s \models \mathsf{sf} \\ M, s \models \mathsf{E}\,\mathsf{pf} &\Leftrightarrow \exists T. s \Downarrow T \wedge M, T \models \mathsf{pf} \\ M, s \models \mathsf{A}\,\mathsf{pf} &\Leftrightarrow \forall T. s \Downarrow T \Rightarrow M, T \models \mathsf{pf} \\ \\ M, T \models \mathsf{sf} &\Leftrightarrow M, T_0 \models \mathsf{sf} \\ M, T \models \mathsf{pf}_1 \wedge \mathsf{pf}_2 &\Leftrightarrow M, T \models \mathsf{pf}_1 \text{ and } M, T \models \mathsf{pf}_2 \\ M, T \models \neg\mathsf{pf} &\Leftrightarrow \text{not } M, T \models \mathsf{pf} \\ M, T \models \mathsf{X}\,\mathsf{pf} &\Leftrightarrow M, T^1 \models \mathsf{pf} \\ M, T \models \mathsf{pf}_1\,\mathsf{U}\,\mathsf{pf}_2 &\Leftrightarrow \exists i.\ M, T^i \models \mathsf{pf}_2 \wedge \\ & \qquad (\forall j < i. M, T^j \models \mathsf{pf}_1) \end{aligned}$$

Figure 3. Semantics of CTL* formulae

The semantics of a CTL* formula is defined *w.r.t.* a model $M = (S, \rightarrow, \lambda, I)$, where S is a set of states, $\rightarrow \subseteq S \times S$ is a total transition relation between states, $\lambda\colon S \rightarrow \mathcal{P}(A)$ defines a valuation on the states, and $I \subseteq S$ denotes the set of initial states. Given a trace $T = s_0, s_1, s_2, \ldots$, we use T^i to denote the trace starting with s_i, *i.e.* $s_i, s_{i+1}, s_{i+2}, \ldots$. As above, we use $s \Downarrow T$ to denote that T is an execution of M, starting in s.

Definition 7 (CTL* semantics) *Given a state* $s \in S$ *and a trace* T *of* M*, Figure 3 defines* $M, s \models \mathsf{sf}$ *and* $M, T \models \mathsf{pf}$*. A model* M *satisfies a state formula* sf*, denoted* $M \models \mathsf{sf}$*, if for all* $i \in I$ *we have* $M, i \models \mathsf{sf}$*.*

A model M *satisfies a path formula* pf*, denoted* $M \models \mathsf{pf}$*, if for all traces* T *starting in an initial state* $i \in I$*, we have* $M, T \models \mathsf{pf}$*.*

4.2 Synchronised Composition with Skips

To enable logical verification, we compose the program that we wish to verify with itself. This results in a single program, for which we can check validity of CTL* formulae. Below, we will give CTL* formulations equivalent to TI and TS observational determinism.

Remember that we are interested in execution traces of the (self-composed) program, and not just in initial and final states. If we would use a standard asynchronous composition to compose the program with itself, we would have to express that if one program copy changes a variable, the other program copy would have to change this variable at some "nearby" moment. This would be a too complicated formula to express in CTL*, because there is an unbounded window between the two points to be compared and we can

only use atomic state predicates; therefore we choose to define a special parallel composition operator $S \bigsqcup S'$ instead. Recalling the definition of observational determinism, we want to show that executions are deterministic in the low variables; thus if a variable changes in one copy of the program, it should be able to change in the other copy of the program as well. Therefore, the definition of $S \bigsqcup S'$ should allow S and S' to take synchronous steps. However, we cannot use a standard synchronous composition operator, because when manipulating private data, S and S' should be allowed to execute in different ways (in particular one should be allowed to wait, while the other manipulates private data). Thus, the composition operator allows the two programs to skip execution steps as they wish, but otherwise to execute synchronously.

The store of $S \bigsqcup S'$ is the product of the two individual stores: (μ, μ'). This ensures that the variables from the two programs are disjoint, and that updates of program copies are done locally, *i.e.* not affecting variables of the other program copy (*cf.* [4]). We will use conf_1, conf_2, prog_1, prog_2, store_1 and store_2 as accessor functions on a configuration of the composed program, such that:

$$\begin{array}{rcl} \mathsf{conf}_1(c) & = & \langle \mathsf{prog}_1(c), \mathsf{store}_1(c) \rangle \\ \mathsf{conf}_2(c) & = & \langle \mathsf{prog}_2(c), \mathsf{store}_2(c) \rangle \end{array}$$

We extend the notion of configuration to self-composed programs in the obvious way. We extend the operational semantics of the program language with the semantics of the synchronised composition with skips as follows.

Definition 8 ($S \bigsqcup S'$) *The transition relation between two composed configurations is described by the following transition rules.*

$$\frac{\langle S_1, \mu_1 \rangle \to \langle S_1', \mu_1' \rangle \qquad \langle S_2, \mu_2 \rangle \to \langle S_2', \mu_2' \rangle}{\langle S_1 \bigsqcup S_2, (\mu_1, \mu_2) \rangle \to \langle S_1' \bigsqcup S_2', (\mu_1', \mu_2') \rangle}$$

$$\frac{\langle S_1, \mu_1 \rangle \to \langle S_1', \mu_1' \rangle}{\langle S_1 \bigsqcup S_2, (\mu_1, \mu_2) \rangle \to \langle S_1' \bigsqcup S_2, (\mu_1', \mu_2) \rangle}$$

$$\frac{\langle S_2, \mu_2 \rangle \to \langle S_2', \mu_2' \rangle}{\langle S_1 \bigsqcup S_2, (\mu_1, \mu_2) \rangle \to \langle S_1 \bigsqcup S_2', (\mu_1, \mu_2') \rangle}$$

The CTL* characterisation will distinguish particular executions of the composed programs, namely those in which if one copy of the program changes a low variable, the other copy changes the same low variable, unless it has terminated or is deadlocked. We call these particular executions *synchronous*. The CTL* formula then specifies that on these synchronous executions, the value of the low variables should always be the same for both program copies.

To be able to specify when a program copy changes a variable, we define the difference between stores and configurations *w.r.t.* $\mathcal{L}$.

Definition 9 (Difference) *The* difference *between stores μ and μ' w.r.t. $\mathcal{L}$, denoted $\mu -_{\mathcal{L}} \mu'$, is defined as*

$$\{v \mid v \in \mathcal{L} \wedge \mu(v) \neq \mu'(v)\}$$

The difference *between configurations c and c' w.r.t. $\mathcal{L}$, denoted $c -_{\mathcal{L}} c'$, is equal to* $\mathsf{store}(c) -_{\mathcal{L}} \mathsf{store}(c')$.

As for indistinguishability, we write $-_v$ to denote $-_{\{v\}}$. Notice that we have the following.

$$\mu \approx_{\mathcal{L}} \mu' \Leftrightarrow \mu -_{\mathcal{L}} \mu' = \emptyset$$

To define a synchronous trace, we take the difference from the i^{th} and the $i+1^{th}$ configuration for both program copies, and we require that these differences are the same, *i.e.* both programs change the same variable, unless one of the programs has already terminated or is deadlocked. This leads us to the following definition of a synchronous trace.

Definition 10 (Synchronous trace) *Let T be a trace of a composed program, and let $\mathcal{L}$ be a set of low variables. Trace T is* synchronous *w.r.t. $\mathcal{L}$, denoted* $\mathsf{synchro}_{\mathcal{L}}(T)$*, if and only if for all i either*

$$\begin{array}{c} \mathsf{store}_1(T_i) -_{\mathcal{L}} \mathsf{store}_1(T_{i+1}) = \\ \mathsf{store}_2(T_i) -_{\mathcal{L}} \mathsf{store}_2(T_{i+1}) \end{array}$$

or $\mathsf{prog}_1(T_i) = \varepsilon$, $\mathsf{prog}_2(T_i) = \varepsilon$, $\mathsf{deadlock}(\mathsf{conf}_1(T_i))$ *or* $\mathsf{deadlock}(\mathsf{conf}_2(T_i))$ *holds.*

We write $\mathsf{synchro}_v$ instead of $\mathsf{synchro}_{\{v\}}$. It is straightforward to prove that when $\mathcal{L}$ is a singleton set, we can always construct a synchronous execution.

4.3 A Model for Self-Composed Programs

As a last step before formulating the CTL* formula, we define precisely the model over which we want the formula to hold. First, we identify the necessary atomic predicates:

- end_1, end_2, denoting whether one of the program copies has reached termination;
- $\mathsf{deadlock}_1$, $\mathsf{deadlock}_2$, denoting whether one of the program copies is deadlocked;
- $\mathsf{eq_stores}_v$, for each low variable v in $\mathcal{L}$, denoting whether the stores of the two copies are the same *w.r.t.* v;
- $\mathsf{eq_store}_{1v}$, $\mathsf{eq_store}_{2v}$, for each low variable v in $\mathcal{L}$, denoting whether the last transition changed the store related to one of the program copies *w.r.t.* v; and
- $\mathsf{eq_changes}_v$, for each low variable v in $\mathcal{L}$, denoting whether the last transition changed v for the two different program copies.

$$
\begin{array}{rcl}
\mathsf{end}_1 \in \lambda(c) & \Leftrightarrow & \mathsf{prog}_1(\mathsf{old}(c)) = \varepsilon \\
\mathsf{end}_2 \in \lambda(c) & \Leftrightarrow & \mathsf{prog}_2(\mathsf{old}(c)) = \varepsilon \\
\mathsf{deadlock}_1 \in \lambda(c) & \Leftrightarrow & \mathsf{deadlock}(\mathsf{old}(c)) \\
\mathsf{deadlock}_2 \in \lambda(c) & \Leftrightarrow & \mathsf{deadlock}(\mathsf{old}(c)) \\
\mathsf{eq_stores}_v \in \lambda(c) & \Leftrightarrow & \mathsf{store}_1(\mathsf{new}(c)) \approx_v \mathsf{store}_2(\mathsf{new}(c)) \\
\mathsf{eq_store}_{1_v} \in \lambda(c) & \Leftrightarrow & \mathsf{store}_1(\mathsf{new}(c)) \approx_v \mathsf{store}_1(\mathsf{old}(c)) \\
\mathsf{eq_store}_{2_v} \in \lambda(c) & \Leftrightarrow & \mathsf{store}_2(\mathsf{new}(c)) \approx_v \mathsf{store}_2(\mathsf{old}(c)) \\
\mathsf{eq_changes}_v \in \lambda(c) & \Leftrightarrow & \mathsf{store}_1(\mathsf{new}(c)) -_v \mathsf{store}_1(\mathsf{old}(c)) = \\
 & & \mathsf{store}_2(\mathsf{new}(c)) -_v \mathsf{store}_2(\mathsf{old}(c))
\end{array}
$$

Figure 4. Equations defining λ

To be able to define the valuation λ, the state space also needs to contain the previous state of the execution. Therefore, we define a special *memorising transition relation*, remembering the previous state of each transition.

Definition 11 (Memorising transition relation) *Let* $\rightarrow \subseteq S \times S$ *be a transition relation. The* memorising transition relation $\rightarrow_m \subseteq (S \times S) \times (S \times S)$ *is defined as*

$$(s, s') \rightarrow_m (t, t') \Leftrightarrow s \rightarrow t \wedge t' = s$$

Thus, (s, s') makes a transition to (t, t') if s makes a transition to t in the original system, and t' remembers the old state s. We use accessor functions new and old to access the components of the memorised state, such that:

$$\mathsf{new}(s, t) = s \qquad \mathsf{old}(s, t) = t$$

We define the model over which we will formulate the CTL* formulae.

Definition 12 (Program model) *Given program S and initial stores μ, μ' we define the* program model $M_{S,\mu,\mu'} = (\Sigma, \rightarrow, \lambda, I)$*, where*

- Σ *denotes the set of products of all configurations of self-composed programs,* i.e. *elements in Σ are of the form $(\langle S_1 \parallel S_2, (\mu_1, \mu_2)\rangle, \langle S_3 \parallel S_4, (\mu_3, \mu_4)\rangle)$;*
- $\rightarrow$ *is the memorising transition relation of the program semantics of self-composed programs (as described in Figure 1 and Definition 8, plus the non-blocking rules defined in Figure 2);*
- λ *is as defined in Figure 4; and*
- $I = \{\langle S \parallel S, (\mu, \mu')\rangle\}$.

For brevity, we write $M_{S,c}$, where c is a configuration, to denote $M_{S,\mathsf{store}_1(c),\mathsf{store}_2(c)}$.

4.4 CTL* characterisations

As explained above, the CTL* formulation of observational determinism requires that for each low variable v we identify synchronous traces *w.r.t.* v, and that these traces have equal stores *w.r.t.* v (and in the termination-sensitive case that if one copy terminates, the other copy terminates as well). Every synchronous program trace *w.r.t.* v satisfies the CTL* formula

$$\mathsf{G}\,\mathsf{synchro}_v \tag{3}$$

(where $\mathsf{synchro}_v$ abbreviates $\mathsf{eq_changes}_v \vee \mathsf{end}_1 \vee \mathsf{end}_2 \vee \mathsf{deadlock}_1 \vee \mathsf{deadlock}_2$).

Lemma 1 *Let S be a program and T a trace of $S \parallel S'$ (*i.e. *$\langle S \parallel S', T_0\rangle \Downarrow T$). Then*

$$\mathsf{synchro}_v(T) \quad \Leftrightarrow \quad M_{S,T_0}, T \models \mathsf{G}\,\mathsf{synchro}_v$$

□

Unfortunately, an insecure program might have synchronous traces for which the stores are always equal. This can happen if one of the two program copies goes into an infinite loop manipulating only private data (or doing nothing at all), while the other program copy skips steps forever, even though it is able to progress on a low variable. Consider for example the following program:

```
if (h) then l := 7
       else while (true) {}
```
(S)

Suppose we execute program $S \parallel S$ in store (μ, μ'), where $\mu(\mathtt{h}) = \mathtt{true}$ and $\mu(\mathtt{h'}) = \mathtt{false}$. Then $S \parallel S$ can have a synchronous trace where only the second copy takes steps, looping forever in the empty while loop. Therefore the CTL* formula will only accept programs for which the synchronous traces satisfy a mutual fairness property. This can be expressed by the following CTL* formula:

$$\begin{array}{l}
\neg\mathsf{F}\,\mathsf{G}\,(\mathsf{eq_store}_{1_v} \wedge \mathsf{EX}\,\neg\mathsf{eq_store}_{2_v}) \wedge \\
\neg\mathsf{F}\,\mathsf{G}\,(\mathsf{eq_store}_{2_v} \wedge \mathsf{EX}\,\neg\mathsf{eq_store}_{1_v})
\end{array} \tag{4}$$

Intuitively, this formula states the following: it cannot be the case that from some point on the program copy 1 (2) never changes the value of v anymore, while program copy 2 (1) always has a possible next state in which it could change the value of v. In other words, if program copy 2 (1) can change v, then it eventually should do so.

Thus, a program is TI observation deterministic if: for all low variables v and for all program traces that are synchronous *w.r.t.* v, the stores should be always equal *w.r.t.* v, and the mutual fairness requirement should hold, *i.e.*

all possible changes to v eventually happen. This is expressed in the following CTL* formula, characterising observational determinism.

$$\bigwedge_{v \in \mathcal{L}} \quad \mathsf{A}\,(\mathsf{G}\,\mathsf{synchro}_v \Rightarrow \mathsf{G}\,\mathsf{eq_stores}_v \wedge \neg \mathsf{F}\,\mathsf{G}\,(\mathsf{eq_store}_{1v} \wedge \mathsf{E}\,\mathsf{X}\,\neg \mathsf{eq_store}_{2v}) \wedge \neg \mathsf{F}\,\mathsf{G}\,(\mathsf{eq_store}_{2v} \wedge \mathsf{E}\,\mathsf{X}\,\neg \mathsf{eq_store}_{1v})) \tag{5}$$

To characterise TS observational determinism we need to add an extra condition on the synchronous traces: if one of the two program copies terminates, the other program copy has to terminate as well. Notice again that this is a global property, independent of the set of low variables v.

$$\bigwedge_{v \in \mathcal{L}} \quad \mathsf{A}\,(\mathsf{G}\,\mathsf{synchro}_v \Rightarrow \mathsf{G}\,\mathsf{eq_stores}_v \wedge \neg \mathsf{F}\,\mathsf{G}\,(\mathsf{eq_store}_{1v} \wedge \mathsf{E}\,\mathsf{X}\,\neg \mathsf{eq_store}_{2v}) \wedge \neg \mathsf{F}\,\mathsf{G}\,(\mathsf{eq_store}_{2v} \wedge \mathsf{E}\,\mathsf{X}\,\neg \mathsf{eq_store}_{1v})) \wedge \mathsf{G}\,((\mathsf{end}_1 \Rightarrow \mathsf{F}\,\mathsf{end}_2) \wedge (\mathsf{end}_2 \Rightarrow \mathsf{F}\,\mathsf{end}_1)) \tag{6}$$

4.5 A proof of equivalence

We prove that the characterisation is correct. First, we define the auxiliary notion of initial trace of a program, *i.e.* a program execution that at some arbitrary point does not make any progress anymore, but stutters forever.

Definition 13 (Initial program trace) *Given a configuration $\langle S, \mu \rangle$, an infinite list of configurations $T = c_0, c_1, c_2, \ldots$ is an* initial program trace *of $\langle S, \mu \rangle$ if and only if*

- $c_0 = \langle S, \mu \rangle$*; and*
- *for all i either $c_i \rightarrow c_{i+1}$ or for all j $c_i = c_{i+j}$.*

Lemma 2 *Every initial program trace can be extended to a program trace.* □

To distinguish initial and complete program traces, below we will use the term *full program trace* to denote the latter. We prove equivalence of the CTL* characterisation and the definition of observational determinism.

Theorem 1 *Termination Insensitive Observational determinism (equation 1) is characterised exactly by the CTL* formula in equation 5, that is for any program S and initial states μ and μ' such that $\mu \approx_{\mathcal{L}} \mu'$, we have*

$$(\forall T, T'. \langle S, \mu \rangle \Downarrow T \wedge \langle S, \mu' \rangle \Downarrow T' \Rightarrow T \approx_{\mathcal{L}} T') \Leftrightarrow M_{S,\mu,\mu'} \models (5)$$

Proof sketch

Case: (1) $\Rightarrow$ (5)
For any variable $v \in \mathcal{L}$, suppose that T is a synchronous trace *w.r.t.* v. Let T_1 denote the projection of T to conf_1, while T_2 denotes the projection of T to conf_2. It is immediate to see that T_1 and T_2 are stutter equivalent to initial program traces, and at least one of them is stutter equivalent to a full program trace.

Suppose *w.l.o.g.* that T_1 is stutter equivalent to a full program trace T_1'. Let T_2' be an extension the initial trace that is stutter equivalent to T_2. By (1) we know that $T_1'(v) \sim_s T_2'(v)$ and thus $T_1(v) \sim_s T_2'(v)$. We know that T_1 changes v exactly as many times as T_2, because T is synchronous. Since we have that $T_1(v) \sim_s T_2'(v)$, T_2 changes v exactly as many times as T_1 (stutter equivalence implies that the number of times a value is changed is the same at any matching point). Thus $T_2(v) \sim_s T_2'(v)$, from which we conclude $T_1(v) \sim_s T_2(v)$. By induction we conclude $\mathsf{G}\ \mathsf{eq_stores}_v$.

The mutual fairness property follows by contradiction. If T_2 would have a v-changing step enabled in its i^{th} state (for some i), which it would never take, then there would be a full program trace extension T_3 that would take this transition. But since $T_1(v) \sim_s T_3(v)$ this would imply that T_1 also changes v once more. Thus, since the trace T is synchronous, T_2 also should make this additional change on v, which leads to a contradiction.

Case: (5) $\Rightarrow$ (1)
For any variable $v \in \mathcal{L}$, suppose T_1 and T_2 are program traces. We combine them into one synchronous trace T:

1. **Case:** both programs can make a v-changing step
   Action: both programs make a step
2. **Case:** copy 1 is about to change v, while copy 2 preserves v
   Action: copy 1 sleeps, copy 2 makes step
   (and vice versa)
3. **Case:** copy 1 finished or deadlocked
   Action: copy 2 makes a step
   (and vice versa)

The formal definition of this procedure uses an auxiliary recursive definition, that keeps track of how many steps the traces T_1 and T_2 have taken to reach the combined configuration T_i.

We use T_1' and T_2' to denote the projections of T to conf_1 and conf_2, respectively. T_1' is stutter equivalent to an initial trace of T_1, T_2' is stutter equivalent to an initial trace of T_2, and at least one of the two is stutter equivalent to a full program trace (*i.e.* to T_1 or T_2). *W.l.o.g.* we assume that T_1' is a full program trace. Thus $T_1 \sim_s T_1'$. We know that the stores in T_1' and T_2' are equal *w.r.t.* v, thus we conclude

$T_1(v) \sim_s T_2'(v)$. Further, we also know that the mutual fairness requirement holds, thus the unfinished trace T_2' is not enabled to make any v-changing steps anymore. Since T_2 is a possible extension of T_2' this gives us $T_2(v) \sim_s T_2'(v)$, and we conclude $T_1(v) \sim_s T_2(v)$. □

Theorem 2 *Termination Sensitive Observational determinism (equation 2) is characterised exactly by the CTL* formula in equation 6.* □

5 A Modal μ-calculus Characterisation of Observational Determinism

We also studied how to express observational determinism in a more expressive logic, namely the modal μ-calculus [16]. The CTL* formula can be directly embedded in the modal μ-calculus [6]; however, using a variation of the modal μ-calculus, namely the polyadic modal μ-calculus [3], another characterisation is possible, using a simpler model. The polyadic μ-calculus, pμK, allows to express directly properties over tuples of (program) models that do not interact at all. Formulae in pμK can be translated back into classical μ-calculus formulae. However, the use of pμK supports a way of thinking that is particularly suited to *e.g.* express notions as bisimulation equivalence and simulation preorders, and that also matches precisely with the idea of self-composition.

Below, we will propose a formula characterising observational determinism that is defined over a two-tuple containing models representing the same program, but with possibly different initial states. The formula expresses when the behaviour of the two models will be deterministic for all low variables v. The formula holds for two models with low-equivalent initial states if and only if the program represented by the models is observationally deterministic. We end the section by proposing an alternative formulation, using standard modal μ-calculus.

5.1 Polyadic modal μ-calculus

Before showing the characterisation of observational determinism, we give a short introduction to pμK. Validity of pμK formulae is undecidable, however model checking is still efficiently decidable (in fact, model checking of pμK can be reduced to model checking of the standard μ-calculus). As above, a model is a tuple $M = (S, \rightarrow, \lambda, I)$, where S are the states, $\rightarrow$ the transition relation, λ a valuation and $I \subseteq S$ denotes the set of initial states.

First we introduce syntax and semantics of pμK (where L is a set of labels and A a set of atomic predicates).

Definition 14 (Syntax of pμK) *Let $\mathcal{V}$ be a countably infinite set of variables over sets of states, ranged over by X, Y, Z etc. The formulae in pμK are inductively defined by:*

$$\phi = p_i \mid \neg\phi \mid \phi_0 \vee \phi_1 \mid \langle a \rangle_i \, \phi \mid X \mid \mu X.\phi$$

where $p \in A$ and $a \in L$.

We use standard abbreviations as

$$\begin{array}{rclcrcl} \phi_0 \wedge \phi_1 & = & \neg(\neg\phi_0 \vee \neg\phi_1) & \quad & [a]_i \, \phi & = & \neg \langle a \rangle_i \neg\phi \\ \phi_0 \Rightarrow \phi_1 & = & \neg\phi_0 \vee \phi_1 & & \mathsf{ff} & = & \mu X.X \\ \nu X.\phi & = & \neg\mu X.\neg\phi[\neg X/X] & & \mathsf{tt} & = & \neg\mathsf{ff} \end{array}$$

The difference with the standard modal μ-calculus is that the atomic predicates and the modalities are labelled with an index i. These indices refer to the i^{th} model, *i.e.* the atomic predicate has to hold for the current state of the i^{th} model, and the modalities refer to a transition labelled a of the i^{th} model[2].

Next we define the semantics of a formula ϕ over a tuple of models $\vec{M}$[3], relative to a mapping ρ that assigns subsets of the states to the free variables in ϕ.

Definition 15 (Semantics of pμK) *Let ϕ be a formula, defined over $\vec{M}$ with arity n and states $S_1, \ldots, S_n$, and ρ be an environment. The* semantics *of ϕ w.r.t. $\vec{M}$ and ρ, denoted $\|\phi\|_\rho^{\vec{M}}$, is defined as a subset of $S = S_1 \times \ldots \times S_n$, by induction over ϕ:*

$$\begin{array}{rcl} \|p_i\|_\rho^{\vec{M}} & = & \{(s_1, \ldots, s_i, \ldots s_n) \mid p \in \lambda(s_i)\} \\ \|\neg\phi\|_\rho^{\vec{M}} & = & S \setminus \|\phi\|_\rho^{\vec{M}} \\ \|\phi_0 \vee \phi_1\|_\rho^{\vec{M}} & = & \|\phi_0\|_\rho^{\vec{M}} \cup \|\phi_1\|_\rho^{\vec{M}} \\ \|\langle a \rangle_i \, \phi\|_\rho^{\vec{M}} & = & \{(s_1, \ldots, s_n) \mid \\ & & \quad \exists s_i'.s_i \xrightarrow{a}_i s_i' \wedge \\ & & \quad (s_1, \ldots, s_i', \ldots s_n) \in \|\phi\|_\rho^{\vec{M}}\} \\ \|\mu X.\phi\|_\rho^{\vec{M}} & = & \bigcap\{U \subseteq S \mid \|\phi\|_{\rho(X \mapsto U)}^{\vec{M}} \subseteq U\} \end{array}$$

Definition 16 (Satisfiability of ϕ) *A tuple of models $\vec{M}$ with sets of initial states $I_1, \ldots I_n$ satisfies ϕ, denoted $\vec{M} \models \phi$, if and only if for all ρ, $I_1 \times \ldots \times I_n \in \|\phi\|_\rho^{\vec{M}}$.*

5.2 Observational determinism in pμK

We express observational determinism in pμK. Let $\mathcal{V}$ be the set of program variables. We introduce action labels $\mathsf{c}_{x,v}$ for any variable $x \in \mathcal{V}$, and any value v in the domain

[2] Atomic predicates are not handled in [3]. However, since an atomic predicate p can be encoded by a self-loop labelled p (a so-called probe), this does not add any new complexity, while it increases the readability of the formulae.

[3] We assume the tuple of models contains as least as many models as the indexes used in the modalities in ϕ. Formally, to ensure this we would have to define an extension of $\vec{M}$ with one-point models, to compensate for any non-existing elements *cf.* [3].

of x. These will be used to label transitions that change the valuation of x. Notice that in order to have a finite set of action labels, we need to have finite domains for all program variables. Further we use τ to label all other (invisible) transitions. We use $\mathcal{A}$ to denote the set of action labels:

$$\mathcal{A} = \{\mathsf{c}_{x,v} \mid x \in \mathcal{V} \wedge v \in \mathsf{dom}(x)\} \cup \{\tau\}$$

The operational semantics (as defined in Figure 1) for assignments is updated as follows.

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

$$\frac{\mu(x) = E(\mu)}{\langle x := E, \mu\rangle \xrightarrow{\tau} \langle \varepsilon, \mu(x \mapsto E(\mu))\rangle}$$

The rules for the conditional, the loop and the wait statement are labelled with τ. The rules for parallel and sequential composition inherit the transition label from the action that is executed. Thus for any label $a \in \mathcal{A}$ we have for example the following rule:

$$\frac{\langle S_1, \mu\rangle \xrightarrow{a} \langle S_1', \mu'\rangle}{\langle S_1 ; S_2, \mu\rangle \xrightarrow{a} \langle S_1' ; S_2, \mu'\rangle}$$

Thus, a transition will be labelled with $\mathsf{c}_{x,v}$ if it executed an assignment to the variable x *and* it actually changed its value, otherwise it will be labelled τ. The condition that the assignment actually changed the value is necessary to ensure that the logical characterisation correctly applies to programs as:

```
l := 7;
while (h >= 0) {l := 7; h := h - 1};
l := 3
```

which can contain a sequence of invisible updates, repeatedly assigning 7 to `l`.

As atomic predicates we use end and $\mathsf{deadlock}$, denoting whether the configuration contains a terminated or deadlocked program, respectively. We define the program model over which we will formulate the pμK formula.

Definition 17 (Program model) *Given program S and initial store μ we define the program model $M_{S,\mu} = (\Sigma, \rightarrow, \lambda, I)$ where*

- *Σ denotes the set of all program configurations;*
- *$\rightarrow$ is the transition relation as defined by the updated operational semantics above,* i.e. *with transitions labelled in $\mathcal{A}$;*
- *$\mathsf{end} \in \lambda(c)$ if and only if $\mathsf{prog}(c) = \varepsilon$, $\mathsf{deadlock} \in \lambda(c)$ if an only if $\mathsf{deadlock}(c)$; and*
- *$I = \{(S, \mu)\}$*

Before presenting the formula, we introduce some convenient abbreviations. Let $A, B \subseteq \mathcal{A}$ be a set of actions.

$$\begin{array}{rcl}
\langle A\rangle_i \phi & = & \bigvee_{a\in A} \langle a\rangle_i \phi \\
\langle \bar{A}\rangle_i \phi & = & \langle \mathcal{A}\backslash A\rangle_i \phi \\
\mathsf{c}_x & = & \{\mathsf{c}_{x,v} \mid v \in \mathsf{dom}(x)\} \\
\langle A\rangle_i^* \phi & = & \mu X.\phi \vee \langle A\rangle_i X \\
[A]_i^* \phi & = & \mu X.\phi \vee [A]_i X \\
\mathsf{always}_i(\phi) & = & \nu X.\phi \wedge [\mathcal{A}]_i X \\
\langle A\backslash B\rangle_i^* \phi & = & \mu X.\phi \vee ([B]_i \,\mathsf{ff} \wedge \langle A\rangle_i X) \\
\mathsf{change}_i(x,v)\phi & = & \langle \mathsf{c}_{x,v}\rangle_i \,\mathsf{tt} \wedge [\mathsf{c}_{x,v}]_i \phi \wedge \\
 & & \bigwedge_{v'\neq v} [\mathsf{c}_{x,v'}]_i \,\mathsf{ff}
\end{array}$$

Thus $\langle A\rangle_i^* \phi$ expresses that there exists a finite path of transitions in A after which ϕ holds, while $[A]_i^* \phi$ expresses that ϕ should hold after all finite paths with transitions in A. We use $\mathsf{always}_i\phi$ to express that ϕ holds everywhere, while $\langle A\backslash B\rangle_i^* \phi$ expresses that there is a finite path containing only transitions in A, on which never any transition in B is enabled, and at the end of which ϕ holds. Finally, $\mathsf{c}_{x,v}$ is true if x can be changed to v, after which ϕ holds, and it is not possible to change x to any value different from v. Notice that we can derive immediately:

$$[\mathsf{c}_x]_i \phi = \bigwedge_{v\in\mathsf{dom}(x)} [\mathsf{c}_{x,v}]_i \phi$$

We express TI observational determinism as follows.

$$\begin{array}{l}
\bigwedge_{x\in\mathcal{L}} \nu R.\, [\overline{\mathsf{c}_x}]_1^* \,(\bigwedge_{v\in\mathsf{dom}(x)} \langle \mathsf{c}_{x,v}\rangle_1 \langle \overline{\mathsf{c}_x}\backslash \mathsf{c}_x\rangle_2^* \,\mathsf{change}_2(x,v) R \\
\qquad\qquad \wedge (\mathsf{end}_1 \vee \mathsf{deadlock}_1 \Rightarrow \mathsf{always}_2\,[\mathsf{c}_x]_2\,\mathsf{ff})) \\
\quad \wedge \\
\quad \nu R.\, [\overline{\mathsf{c}_x}]_2^* \,(\bigwedge_{v\in\mathsf{dom}(x)} \langle \mathsf{c}_{x,v}\rangle_2 \langle \overline{\mathsf{c}_x}\backslash \mathsf{c}_x\rangle_1^* \,\mathsf{change}_1(x,v) R \\
\qquad\qquad \wedge (\mathsf{end}_2 \vee \mathsf{deadlock}_2 \Rightarrow \mathsf{always}_1\,[\mathsf{c}_x]_1\,\mathsf{ff}))
\end{array} \tag{7}$$

This formula says that if model 1 takes a finite number of steps, not changing the low variable x – thus all transitions are in $\overline{\mathsf{c}_x}$ – and then it changes x to v, model 2 should be forced to do the same. Thus, there is a finite path with transitions in $\overline{\mathsf{c}_x}$ on which it is not possible to change x, and at the end of this path x can only be changed to v, after which the formula recursively holds again. A special case is when model 1 terminates or deadlocks: in this case model 2 is allowed to do anything, except changing x. And of course we also have the same formula for model 2.

5.3 Observational determinism in the standard modal μ-calculus

As mentioned above, every model and formula in pμK can be reduced into a model and formula of the standard

modal μ-calculus, by renaming action a with index i into action a_i, and predicate p with index i into predicate p_i [3]. If we do this, we can define a state predicate eq_x which holds if the store in model 1 contains the same value for x as the store in model 2. With such a predicate, TI observational determinism is characterised by the following modal μ-calculus formula.

$$\begin{aligned}
&\bigwedge_{x\in\mathcal{L}} \nu R.\, [\overline{\mathsf{c}_{x,1}}]^*\,(\langle \mathsf{c}_{x,1}\rangle\, \langle \overline{\mathsf{c}_{x,2}}\backslash \mathsf{c}_{x,1}\rangle^*\ \ (\langle \mathsf{c}_{x,2}\rangle\, \mathsf{tt} \wedge \\
&\qquad\qquad [\mathsf{c}_{x,2}]\, \mathsf{eq}_x \wedge R) \\
&\qquad \wedge (\mathsf{end}_1 \vee \mathsf{deadlock}_1 \Rightarrow \mathsf{always}\, [\mathsf{c}_{x,2}]\, \mathsf{ff}) \\
&\wedge \\
&\nu R.\, [\overline{\mathsf{c}_{x,2}}]^*\,(\langle \mathsf{c}_{x,2}\rangle\, \langle \overline{\mathsf{c}_{x,1}}\backslash \mathsf{c}_{x,1}\rangle^*\ \ (\langle \mathsf{c}_{x,1}\rangle\, \mathsf{tt} \wedge \\
&\qquad\qquad [\mathsf{c}_{x,1}]\, \mathsf{eq}_x \wedge R) \\
&\qquad \wedge (\mathsf{end}_2 \vee \mathsf{deadlock}_2 \Rightarrow \mathsf{always}\, [\mathsf{c}_{x,1}]\, \mathsf{ff})
\end{aligned} \tag{8}$$

where $\mathsf{c}_{x,i}$ denotes an action label for the i^{th} model changing the variable x. Notice that here we do not need action labels $\mathsf{c}_{x,v}$; it is sufficient to label assignments with the variable that is changed.

6 Conclusions and Future Work

We have presented a characterisation of observational determinism using two different logics, namely CTL* and the polyadic modal μ-calculus. For both logics the model checking problem is decidable (for finite state programs), thus this shows that observational determinism is a decidable security property. The CTL* characterisation is a fairly direct translation of the definition of observational determinism, but it is defined over a relatively complicated model. In contrast, the modal μ-calculus characterisations are more complicated, but expressed over a simpler model. We believe that this difference is due to the fact that we can use transition labels in the modal μ-calculus, which allow us to describe the permitted executions. In CTL* we cannot do this (in a straightforward way), so we need a more complicated parallel composition which ensures that we can easily characterise the relevant executions. We will use our model checking experiments, described below, to see which of the two approaches in the end is the most pertinent to model check security properties.

The characterisations are based on the idea of self-composition [4], *i.e.* composing a program with a copy of itself, so that properties can be expressed as properties of the composed program. However, we widen this idea, because the composition operators that we use are an extension of the programming language.

Experiments To study the feasibility of our approach, we modelled our program semantics and composition operators in LOTOS [17]. To ensure that we can check the properties over arbitrary input states, we have added a special input step. This step initialises each variable to some arbitrary value. Every formula is expressed to hold after this special initialisation step, if the stores are equal *w.r.t.* the low variables. In the future we would like to find means to abstract away from this input step, and to be able to express and validate properties over all low-equivalent initial states.

Our model can be used as input for the CADP tool set[4]. We choose this tool set, because of the flexibility of its model specification language LOTOS, and because of its well-established reputation. One of the tools in CADP is Evaluator, which allows to model check (alternation-free) modal μ-calculus formulae.

Therefore we have rephrased the CTL* characterisation (formula (1)) (based on the translation given by Dam [6]) into the modal μ-calculus. The model checker correctly rejects all examples where the synchronous traces are not having equal stores (and generates appropriate counter examples). Unfortunately, the mutual fairness condition cannot be expressed in alternation-free modal μ-calculus. Instead we use a stronger property $\mathsf{G\,A}\,(\mathsf{G\,eq_store}_1 \wedge \mathsf{synchro}_v \Rightarrow \mathsf{G\,A\,X\,eq_store}_2)$ (at any point in time: if on any execution path the store of the first program copy for variable v does not change any more, then the second program copy is also not able to change this variable). We are currently studying whether this property is not so strong that it rejects all example programs.

We are also expressing the modal μ-calculus formulae (7) and (8) in the input language for Evaluator. To express the polyadic modal μ-calculus formula, we have to make the implicit conjunctions over all input labels explicit (*e.g.* replacing $\langle \overline{c_x} \rangle$ by $\langle c_x,1\rangle \vee \langle c_x,2\rangle \vee \ldots$ for all possible values of x, which results in an enormous formula. This already shows the advantage of falling back on the standard modal μ-calculus, and not holding on to the strict separation of the two models. The models that are generated for this approach are significantly smaller than the models that are generated for the CTL* formulation. Since the formulae that result from the translation of the CTL* formulae into the modal μ-calculus is not significantly simpler than the formula we phrased for the modal μ-calculus characterisation directly, this indicates that the latter is probably the more promising approach to achieve model checking of security properties.

Future work Since formulae in the modal μ-calculus can be essentially specified over a collection of independent (program) models, this formalism seems especially suited to characterise also other definitions of non-interference and other security notions (in particular for multi-threaded programs), including generalisations of observational determinism, *e.g.* by requiring that a program behaves observa-

[4] See `http://www.inrialpes.fr/vasy/cadp/`.

tionally deterministic in any context, which gives a compositional security notion.

Having formulations of these different definitions in the same formalism allows for an easier comparison, classifying for example the restrictiveness of the different definitions of non-interference. Since model checking of the modal μ-calculus is decidable, any security property that can be rephrased as such a formula can be checked.

Finally, model checking has its limitations, due to scalability and its demand for a finite state space. However, there is a large body of research on techniques to overcome these limitations, *e.g.* state space reduction and special algorithms to verify specific program constructs such as dynamic allocation [10]. Eventually, we hope to exploit these techniques to provide a tool that can efficiently model check security properties on realistic multi-threaded programs.

Acknowledgements

We thank Pedro D'Argenio, Tamara Rezk and the reviewers for their useful comments. Further, we thank Judi Romijn, Radu Mateescu and Dilian Gurov for their help with the model checking experiments.

References

[1] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *POPL'06*, pages 91–102. ACM Press, 2006.

[2] T. Amtoft and A. Banerjee. A logic for information flow analysis with an application to forward slicing of simple imperative programs. *Science of Computer Programming*, 200x. To appear.

[3] H.R. Andersen. A polyadic modal μ-calculus. Technical Report 1994-145, Tech. Univ. of Denmark (DTU), 1994.

[4] G. Barthe, P. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *CSFW'17*. IEEE Press, 2004.

[5] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *TCS*, 281(1):109–130, 2002.

[6] M. Dam. CTL* and ECTL* as fragments of the modal mu-calculus. *TCS*, 126(1):77–96, 1994.

[7] M. Dam. Decidability and proof systems for language-based noninterference relations. In *POPL'06*, 2006.

[8] Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *Security in Pervasive Computing*, number 3450 in LNCS, pages 193 – 209. Springer, 2005.

[9] J.K. Deka. *Model Checking Techniques for Reasoning about Events and Extremal Properties in Timed Systems*. PhD thesis, IIT Kharagpur, 2000.

[10] D. Distefano, A. Rensink, and J.-P. Katoen. Model checking birth and death. In *Foundations of Information Technology in the Era of Network and Mobile Computing*, volume 223 of *IFIP Conference Proceedings*, pages 435–447. Kluwer, 2002.

[11] E.A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*. Elsevier, 1990.

[12] E.A. Emerson and J.Y. Halpern. "Sometimes" and "Not Never" revisited: on branching versus linear time temporal logic. *J. of the ACM*, 33:151–178, 1986.

[13] J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[14] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specifi cation, third edition*. The Java Series. Addison-Wesley, 2005.

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

[16] D. Kozen. Results on the propositional μ-calculus. *TCS*, 27:333–354, 1983.

[17] Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour, 1989. ISO International Standard 8807.

[18] D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Inf. Processing Letters*, 63:243–246, 1997.

[19] F. Pottier. A simple view of type-secure information flow in the pi-calculus. In *CSFW 15*, pages 320–330, 2002.

[20] F. Pottier and V. Simonet. Information flow inference for ML. *ACM TOPLAS*, 25:117–158, 2003.

[21] A.W. Roscoe. CSP and determinism in security modelling. In *IEEE Symposium on Security and Privacy*, page 114, Washington, DC, USA, 1995. IEEE Computer Society.

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

[23] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *POPL'98*, pages 355–364, 1998.

[24] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Symposium on Static Analysis (SAS'05)*, number 3672 in LNCS, pages 352–367. Springer, 2005.

[25] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[26] S. Zdancewic and A.C. Myers. Observational determinism for concurrent program security. In *CSFW'16*, 2003.