Relational thread-modular static value analysis by abstract interpretation

Antoine Miné

Relational Thread-Modular Static Value Analysis by Abstract Interpretation*

Antoine Miné

CNRS & École Normale Supérieure
45, rue d'Ulm
75005 Paris, France
mine@di.ens.fr

Abstract. We study thread-modular static analysis by abstract interpretation to infer the values of variables in concurrent programs. We show how to go beyond the state of the art and increase an analysis precision by adding the ability to infer some relational and history-sensitive properties of thread interferences. The fundamental basis of this work is the formalization by abstract interpretation of a rely-guarantee concrete semantics which is thread-modular, constructive, and complete for safety properties. We then show that previous analyses based on non-relational interferences can be retrieved as coarse computable abstractions of this semantics; additionally, we present novel abstraction examples exploiting our ability to reason more precisely about interferences, including domains to infer relational lock invariants and the monotonicity of counters. Our method and domains have been implemented in the AstréeA static analyzer that checks for run-time errors in embedded concurrent C programs, where they enabled a significant reduction of the number of false alarms.

Keywords: static analysis, abstract interpretation, verification, safety, concurrency, embedded programs, rely-guarantee methods

1 Introduction

Programming is an error-prove activity and software errors are frequent; it is thus useful to design tools that help ensuring program correctness. In this article, we focus on static analyzers, which enjoy several benefits: they are fully automatic (always terminating and requiring minimal annotations, making them easy to deploy and cost-effective in industrial contexts), sound (no program behavior, and so, no bug is overlooked), and they offer a wide range of cost versus precision choices; however they can exhibit false positives (spurious alarms reported by the tool, that need to be checked manually), which we naturally wish to minimize.

* This work is supported by the INRIA project "Abstraction" common to CNRS and ENS in France and by the project ANR-11-INSE-014 from the French *Agence nationale de la recherche*.

Abstract interpretation [6] makes it possible to design sound static analyzers in a principled way, by abstraction of a concrete semantics expressing the properties of interest. Prior results on Astrée [3] showed that abstract interpretation could effectively drive the construction of an analyzer that is both efficient and extremely precise (no or few false alarms), by specializing the abstractions to a class of properties and a class of programs, in that case: the absence of run-time error in embedded synchronous control/command avionic C programs. We are now bent on achieving a similar result for *concurrent programs*: we are developing AstréeA [19], a static analyzer to prove the absence of run-time error in embedded concurrent C programs where several threads are scheduled by a real-time operating system, communicate through a shared memory, and synchronize through concurrency primitives (such as mutual exclusion locks).

Although concurrent programming is not new, its use has intensified recently with the rise of consumer multi-core systems. Concurrent programming is also increasingly used to improve cost-effectiveness in the critical embedded domain (e.g., Integrated Modular Avionics [23]), where the need for verification is important. Concurrent programs are more challenging to verify than sequential ones: as a concurrent execution is an interleaving of executions of individual threads often scheduled with a high level of non-determinism (e.g., driven by inputs from the environment), the number of possible executions is generally very high. The verification problem is further complicated by the advent of weakly consistent memories taking hardware and software optimization into account [2].

A solution to avoid considering interleavings explicitly and the associated combinatorial exposition of executions is to use thread-modular methods. Ideally, analyzing a concurrent program should be performed by analyzing individually each thread. Analyzing threads in isolation is not sound as it ignores their potential interactions, but previous work by Carré and Hymans [4] and ourself [19] showed that sequential analyses can be easily modified to take interactions and weakly memory models into account. Unfortunately, these methods are based on a simplistic, non-relational and flow-insensitive concrete semantics of thread interactions, which severely limits the precision of any analysis built by abstracting this semantics. In this article, we propose another, more precise semantics, from which former analyses can be recovered by abstraction, but that also allows more precise, relational abstractions of thread interferences. It is based on Jones' popular rely-guarantee proof method for concurrent programs [13], formulated as a constructive fixpoint semantics in abstract interpretation style.

The rest of this introduction presents our former non-relational interference analysis, exemplifies its shortcommings to motivate our work, and recalls the rely-guarantee reasoning proof technique.

Analysis based on non-relational interferences. We illustrate our former analysis [19] and its limits on the example of Fig. 1. This simple program is composed of two threads: Thread t_2 increments Y by a random value in $[1, 3]$ while it is smaller than 100, and Thread t_1 concurrently increments X while it is smaller than Y. Both variables are initialized to 0 before the program starts.

t_1	t_2
$^{(1a)}$ **while** random **do**	$^{(1b)}$ **while** random **do**
$\quad ^{(2a)}$ **if** $X < Y$ **then**	$\quad ^{(2b)}$ **if** $Y < 100$ **then**
$\quad\quad ^{(3a)}$ $X \leftarrow X + 1$	$\quad\quad ^{(3b)}$ $Y \leftarrow Y + [1, 3]$
$\quad ^{(4a)}$ **endif**	$\quad ^{(4b)}$ **endif**
$^{(5a)}$ **done**	$^{(5b)}$ **done**

Fig. 1. Concurrent program example.

$$
\begin{array}{ll}
\mathcal{X}_{1b} = \{[X \mapsto 0, Y \mapsto 0]\} & \textit{(initialization)} \\
\mathcal{X}_{2b} = \mathcal{X}_{1b} \cup \mathcal{X}_{5b} & \textit{(control-flow join at loop head)} \\
\mathcal{X}_{3b} = [\![\, Y < 100 \,]\!] \mathcal{X}_{2b} & \textit{(filtering by test condition on } Y\textit{)} \\
\mathcal{X}_{4b} = [\![\, Y \leftarrow Y + [1, 3] \,]\!] \mathcal{X}_{3b} & \textit{(assignment into } Y\textit{)} \\
\mathcal{X}_{5b} = \mathcal{X}_{4b} \cup [\![\, Y \geq 100 \,]\!] \mathcal{X}_{2b} & \textit{(control-flow join after test)}
\end{array}
$$

Fig. 2. Concrete equation system for Thread t_2 from Fig. 1.

Consider first the simpler problem of analyzing t_2 in isolation, viewed as a sequential program. We wish to infer the set of reachable memory states (i.e., the values of X and Y) at each program point. This can be expressed classically [7] as the least solution of the invariance equation system in Fig. 2, where each variable $\mathcal{X}_i$ is the memory invariant at program point i, with value in $\mathcal{D} \stackrel{\text{def}}{=} \mathcal{P}(\{X, Y\} \to \mathbb{Z})$, and $[\![\, \cdot \,]\!]$ is the effect of an atomic program operation (assignment or test) on a set of memory states, e.g.: $[\![\, Y < 100 \,]\!]\mathcal{X} \stackrel{\text{def}}{=} \{\, \rho \in \mathcal{X} \mid \rho(Y) < 100 \,\}$ models a test by filtering states while $[\![\, Y \leftarrow Y + [1, 3] \,]\!]\mathcal{X} \stackrel{\text{def}}{=} \{\, \rho[Y \mapsto \rho(Y) + v] \mid \rho \in \mathcal{X}, v \in [1, 3] \,\}$ models an incrementation by a non-deterministic value. We get, for instance, that the loop invariant at point $2b$ is: $X = 0 \wedge Y \in [0, 102]$. An effective analysis is obtained by replacing concrete variables $\mathcal{X}_i \in \mathcal{D}$ with abstract ones $\mathcal{X}_i^\sharp \in \mathcal{D}^\sharp$ living in an abstract domain $\mathcal{D}^\sharp$, concrete operations $\cup$ and $[\![\, \cdot \,]\!]$ with abstract ones $\cup^\sharp$ and $[\![\, \cdot \,]\!]^\sharp$, and employing convergence acceleration techniques $\triangledown$ to compute, by iteration, an abstract solution of the system where $\supseteq$ replaces $=$. We get an inductive (but not necessarily minimal) invariant. For Fig. 1, a simple interval analysis using widenings with thresholds can infer that $Y \in [0, 102]$. A similar analysis of t_1 would infer that X and Y stay at 0 as it would ignore, for now, the effect of t_2.

We now turn to the analysis of the full program under the simplest concurrent execution model, sequential consistency [14]: a program execution is an interleaving of tests and assignments from the threads, each operation being considered as atomic. A straightforward approach is to associate a variable $\mathcal{X}_{i,j}$ to each pair of control points, i for t_1 and j for t_2, and construct the product equation system from that of both threads. For instance, we would have:

$$\mathcal{X}_{3a,3b} = [\![\, X < Y \,]\!]\mathcal{X}_{2a,3b} \cup [\![\, Y < 100 \,]\!]\mathcal{X}_{3a,2b} \tag{1}$$

as the point $3a, 3b$ can be reached either with a step by t_1 from $2a, 3b$, or a step by t_2 from $3a, 2b$. However, this quickly results in large equation systems, and

we discard this method as impractical. The methods proposed in [4,19] consist instead in analyzing each thread independently as a sequential program, extracting from the analysis (an abstraction of) the set of values each thread stores into each variable, so-called interferences, and then reanalyzing each thread taking into account these interferences; more behaviors of the threads may be exposed in the second run, resulting in more interferences, hence, the thread analyses are iterated until the set of interferences becomes stable. On our example, in the concrete, after the first analysis of t_1 and t_2 reported above, we extract the fact that t_2 can store any value in $[1, 102]$ into Y. In the second analysis of t_1, this information is incorporated by replacing the equation $\mathcal{X}_{3a} = [\![X < Y]\!]\mathcal{X}_{2a}$ with:

$$\mathcal{X}_{3a} = [\![X < (Y \mid [1, 102])]\!]\mathcal{X}_{2a} \tag{2}$$

and similarly for $\mathcal{X}_{5a}$, where $Y \mid [a, b]$ denotes a non-deterministic choice between the current value of Y and an integer between a and b. The test thus reduces to $X < 102$, i.e., t_1 increments X to at most 102. Accordingly, it generates new interferences, on X. As X is not used in t_2, the third analysis round is identical to the second one, and the analysis finishes. By replacing concrete variables, interferences, and operations with abstract ones, and using extrapolation $\triangledown$ to stabilize abstract interferences, we obtain an effective analysis method.

This method is attractive because it is simple and efficient: it is constructed by slightly modifying existing sequential analyses and reuses their abstract domains, it does not require much more memory (only the cost of abstract interferences) nor time (few thread analyses are required in practice, even for large programs, as shown in Fig. 6 in Sec. 5). Unfortunately, modeling interferences as a set of variable values that effect threads in a non-deterministic way severely limits the analysis precision. Even when solving exactly the sequence of concrete equation systems, we can only deduce that $X \in [0, 102] \wedge Y \in [0, 102]$ at the end of Fig. 1 while, in fact, $X \leq Y$ also holds. Naturally, no derived abstract analysis can infer $X \leq Y$, even if it employs a relational domain able to express it (such as octagons [17]).

Relational rely-guarantee reasoning. Rely-guarantee is a proof method introduced by Jones [13] that extends Hoare's logic to concurrent programs. It is powerful enough to prove complex properties, such as $X \leq Y$ in our example. Rely-guarantee replaces Hoare's triples $\{P\}\ s\ \{Q\}$ with quintuples $R, G \vdash \{P\}\ s\ \{Q\}$, requiring us to annotate program points with invariants P and Q, but also relations R and G on whole thread executions; it states that, if the pre-condition P holds before s is executed and all the changes by other threads are included in R, then, after s, Q holds and all the thread's changes are included in G. The annotations required for the program in Fig. 1 are presented in Fig. 3, including the invariants holding at each program point $1a$ to $5b$, and rely assertions R_1, R_2. In particular, to prove that $X \leq Y$ holds in t_1, it is necessary to rely on the fact that t_2 can only increment Y, and so, does not invalidate invariants of the form $X \leq Y$. In Fig. 3, our assertions are very tight, so that each thread exactly guarantees what the other relies on ($R_1 = G_2$ and $R_2 = G_1$).

checking t_1 :

t_1	$R_1 = G_2$
$^{(1a)}$ **while** random **do**	X is unchanged
$\quad^{(2a)}$ **if** $X < Y$ **then**	Y is incremented
$\quad\quad^{(3a)}$ $X \leftarrow X + 1$	$0 \leq Y \leq 102$
$\quad^{(4a)}$ **endif**	
$^{(5a)}$ **done**	

checking t_2 :

$R_2 = G_1$	t_2
Y is unchanged	$^{(1b)}$ **while** random **do**
$0 \leq X \leq Y$	$\quad^{(2b)}$ **if** $Y < 100$ **then**
	$\quad\quad^{(3b)}$ $Y \leftarrow Y + [1, 3]$
	$\quad^{(4b)}$ **endif**
	$^{(5b)}$ **done**

$1a : X = 0 \wedge Y \in [0, 102]$
$2a : X \leq Y \wedge X, Y \in [0, 102]$
$3a : X < Y \wedge X \in [0, 101] \wedge Y \in [1, 102]$
$4a : X \leq Y \wedge X, Y \in [1, 102]$
$5a : X \leq Y \wedge X, Y \in [0, 102]$

$1b : X = 0 \wedge Y = 0$
$2b : X \leq Y \wedge X, Y \in [0, 102]$
$3b : X \leq Y \wedge X, Y \in [0, 99]$
$4b : X \leq Y \wedge X \in [0, 102] \wedge Y \in [1, 102]$
$5b : X \leq Y \wedge X \in [0, 102] \wedge Y \in [1, 102]$

Fig. 3. Rely-guarantee assertions proving that $X \leq Y$ holds in the program in Fig. 1.

Rely-guarantee is modular: each thread can be checked without looking at the other threads, but only at the rely assertions. This is in contrast to Owicki and Gries' earlier method [21], where checking a Hoare triple required delving into the full code of all other threads to check for non-interference. Intuitively, the rely assertions form an abstraction of the semantics of the threads. While attractive for its expressive power, classic rely-guarantee relies on user annotations. In the following, we use abstract interpretation to infer them automatically.

Overview. The article is organized as follows: Sec. 2 presents the formalization of rely-guarantee in constructive form; Sec. 3 shows how to retrieve our coarse analysis by abstraction while Sec. 4 presents novel abstractions that convey a degree of relationality and history-sensitivity; we also discuss there the analysis in the presence of locks and some uses of trace abstractions. Experimental results are presented in Sec. 5 and Sec. 6 concludes.

Related work. There is a large body of work on the analysis of concurrent programs; we discuss here only the ones most related to our work and refer the reader to Rinard's survey [22] for general information. We already mentioned previous work on thread-modular static analyses [4,19] which only support non-relational interferences and are limited in precision. Jeannet proposed a precise relational static analysis [12]; it is not thread-modular and may not scale up. Works such as [11] bring thread-modular reasoning to model checking. They inherit the limitations of the underlying model checking method; in the case of [11], the system must be finite-state. Moreover, Malkis et al. observed in [15] that it performs implicitly a non-relational (so-called Cartesian) abstraction; we make here the same observation concerning our previous work [19], but we go further by providing non-trivial relational abstractions. Recent works [10,1] seek to alleviate the burden of providing user annotations in rely-guarantee proof methods, but do not achieve complete automation. Our approach is fundamentally similar

to Cousot and Cousot's formulation of the Owicki, Gries, and Lamport proof methods in abstract interpretation form [8], but applied to Jones' method instead. The results in Sec. 2 and Sec. 3 have been partially described before in a research report [20] and course notes [18]; Sec. 4 and Sec. 5 are novel.

2 Rely-Guarantee in Abstract Interpretation Form

The first step in any abstract interpretation is the formalization of the concrete semantics in a constructive form, using fixpoints. We show how, in a very general setting, the concrete semantics of a concurrent program can be presented in a thread-modular way.

2.1 Programs and Transition Systems

Programs. Our programs are composed of a finite set $\mathcal{T}$ of *threads* (the unbounded case is discussed in Sec. 3.3). We denote by $\mathcal{L}$ the set of *program points*. A thread $t \in \mathcal{T}$ is specified as a control-flow graph by: an *entry point* $e_t \in \mathcal{L}$, and a set of *instructions* $inst_t \subseteq \mathcal{L} \times Inst \times \mathcal{L}$. For now, $Inst$ contains assignments $X \leftarrow e$ and comparisons $e \bowtie e'$ (it will be enriched with synchronization primitives in Sec. 4.4). We denote by $\mathcal{V}$ the (possibly unbounded) set of variables; they are global and shared by all the threads. We denote by $\mathbb{V}$ the domain of variable values. To stay general, we deliberately refrain from specifying the set $\mathbb{V}$, the syntax of expressions e, e' and of comparison operators $\bowtie$.

Transition systems. Following Cousot and Cousot [7], we model program semantics as *labelled transition systems*, a form of small-step semantics which is very general and allows reasoning independently from the chosen programming language. A transition system $(\Sigma, \mathcal{A}, I, \tau)$ is given by: a set Σ of *program states*; a set $\mathcal{A}$ of *actions*; a set $I \subseteq \Sigma$ of *initial states*; a *transition relation* $\tau \subseteq \Sigma \times \mathcal{A} \times \Sigma$; we will note $\langle \sigma, a, \sigma' \rangle \in \tau$ as $\sigma \xrightarrow{a}_\tau \sigma'$. We instantiate transition systems on our programs as follows:

- $\Sigma \stackrel{\text{def}}{=} \mathcal{C} \times \mathcal{M}$: states $\langle L, \rho \rangle \in \Sigma$ consist of a *control state* $L \in \mathcal{C} \stackrel{\text{def}}{=} \mathcal{T} \to \mathcal{L}$ associating a current location $L(t) \in \mathcal{L}$ to each thread $t \in \mathcal{T}$ and a *memory state* $\rho \in \mathcal{M} \stackrel{\text{def}}{=} \mathcal{V} \to \mathbb{V}$ associating a value $\rho(V) \in \mathbb{V}$ to each variable $V \in \mathcal{V}$;
- $I \stackrel{\text{def}}{=} \{ \langle \lambda t.\, e_t,\ \lambda V.\, 0 \rangle \}$: we start with all the threads at their entry point and variables at zero;
- $\mathcal{A} \stackrel{\text{def}}{=} \mathcal{T}$: actions record which thread generates each transition;
- transitions model atomic execution steps of the program:

$$\{ \langle L, \rho \rangle \xrightarrow{t}_\tau \langle L', \rho' \rangle \mid \langle L(t), \rho \rangle \to_t \langle L'(t), \rho' \rangle \wedge \forall t' \neq t : L(t') = L'(t') \}$$
$$\text{where } \langle \ell, \rho \rangle \to_t \langle \ell', \rho' \rangle \iff^{\text{def}} \exists i \in Inst : \langle \ell, i, \ell' \rangle \in inst_t \wedge \rho' \in [\![i]\!] \rho$$

i.e.: we choose a thread t to run and an instruction i from thread t; we let it update t's control state $L(t)$ and the global memory state ρ (the thread transition being denoted as $\langle L(t), \rho \rangle \to_t \langle L'(t), \rho' \rangle$), while the other threads $t' \neq t$ stay at their control location $L(t')$.

2.2 Monolithic Concrete Semantics

We first recall the standard, non-modular definition of the semantics of transition systems. An *execution trace* is a (finite or infinite) sequence of states interspersed with actions, which we denote as: $\sigma_0 \xrightarrow{a_1} \sigma_1 \xrightarrow{a_2} \cdots$. As we are interested solely in safety properties, our concrete semantics will ultimately compute the so-called *state semantics*, i.e., the set $\mathcal{R}$ of states reachable in any program trace. It is defined classically as the following least fixpoint:[1]

$$\mathcal{R} \stackrel{\text{def}}{=} \operatorname{lfp} R, \text{ where } R \stackrel{\text{def}}{=} \lambda S.\, I \cup \{\, \sigma \mid \exists \sigma' \in S, a \in \mathcal{A} : \sigma' \xrightarrow{a}_\tau \sigma \,\} \ . \tag{3}$$

We also recall [5] that $\mathcal{R}$ is actually an abstraction of a more precise semantics: the trace semantics $\mathcal{F}$, that gathers the finite partial traces (i.e., the finite prefixes of the execution traces). The semantics $\mathcal{F}$ can also be defined as a fixpoint:

$$\begin{array}{l} \mathcal{F} \stackrel{\text{def}}{=} \operatorname{lfp} F, \text{ where} \\ F \stackrel{\text{def}}{=} \lambda X.\, I \cup \{\, \sigma_0 \xrightarrow{a_1} \cdots \sigma_i \xrightarrow{a_{i+1}} \sigma_{i+1} \mid \sigma_0 \xrightarrow{a_1} \cdots \sigma_i \in X \wedge \sigma_i \xrightarrow{a_{i+1}}_\tau \sigma_{i+1} \,\} \ . \end{array}$$

Indeed, $\mathcal{R} = \alpha^{reach}(\mathcal{F})$, where $\alpha^{reach}(T) \stackrel{\text{def}}{=} \{\, \sigma \mid \exists \sigma_0 \xrightarrow{a_1} \cdots \sigma_n \in T : \exists i \leq n : \sigma = \sigma_i \,\}$ forgets the order of states in traces. The extra precision provided by the trace semantics will prove useful shortly in our thread-modular semantics, and later for history-sensitive abstractions (Sec. 4.3).

The connection with equation systems is well-known: $\mathcal{R}$ is the least solution of the equation $\mathcal{R} = R(\mathcal{R})$. By associating a variable $\mathcal{X}_c$ with value in $\mathcal{P}(\mathcal{M})$ to each $c \in \mathcal{C}$, we can rewrite the equation to the form $\forall c \in \mathcal{C} : \mathcal{X}_c = F_c(\mathcal{X}_1, \ldots, \mathcal{X}_n)$ for some functions F_c. The solution satisfies $\mathcal{R} = \{\, \langle c, \rho \rangle \mid c \in \mathcal{C}, \rho \in \mathcal{X}_c \,\}$, i.e., $\mathcal{X}_c$ partitions $\mathcal{R}$ by control location. We retrieve standard equation systems for sequential programs (as in Fig. 2) and derive effective abstract static analyses but, when applied to concurrent programs, $\mathcal{C}$ is large and we get unattractively large systems, as exemplified by (1) in the introduction.

2.3 Thread-Modular Concrete Semantics

We can now state our first contribution: a thread-modular expression of $\mathcal{R}$.

Local states. We define the reachable local states $\mathcal{Rl}(t)$ of a thread t as the state abstraction $\mathcal{R}$ where the control part is reduced to that of t only. The control part of other threads $t' \neq t$ is not lost, but instead stored in auxiliary variables $pc_{t'}$ (we assume here that $\mathcal{L} \subseteq \mathbb{V}$). Thread local states thus live in $\Sigma_t \stackrel{\text{def}}{=} \mathcal{L} \times \mathcal{M}_t$ where $\mathcal{M}_t \stackrel{\text{def}}{=} \mathcal{V}_t \to \mathbb{V}$ and $\mathcal{V}_t \stackrel{\text{def}}{=} \mathcal{V} \cup \{\, pc_{t'} \mid t' \neq t \,\}$. We get:

$$\begin{array}{l} \mathcal{Rl}(t) \stackrel{\text{def}}{=} \pi_t(\mathcal{R}) \text{ where} \\ \pi_t(\langle L, \rho \rangle) \stackrel{\text{def}}{=} \langle L(t), \rho\,[\forall t' \neq t : pc_{t'} \mapsto L(t')] \rangle \\ \text{extended element-wise as } \pi_t(X) \stackrel{\text{def}}{=} \{\, \pi_t(x) \mid x \in X \,\} \ . \end{array} \tag{4}$$

[1] Our functions are monotonic in complete powerset lattices. By Tarski's theorem, all the least fixpoints we use in this article are well defined.

π_t is one-to-one: thanks to the auxiliary variables, no information is lost, which is important for completeness (this will be exemplified in Ex. 3).

Interferences. For each thread $t \in \mathcal{T}$, the interferences it causes $\mathcal{I}(t) \in \mathcal{P}(\Sigma \times \Sigma)$ is the set of transitions produced by t in the partial trace semantics $\mathcal{F}$:

$$\begin{array}{l} \mathcal{I}(t) \stackrel{\text{def}}{=} \alpha^{itf}(\mathcal{F})(t), \text{ where} \\ \alpha^{itf}(X)(t) \stackrel{\text{def}}{=} \{\, \langle \sigma_i, \sigma_{i+1} \rangle \mid \exists \sigma_0 \stackrel{a_1}{\rightarrow} \sigma_1 \cdots \stackrel{a_n}{\rightarrow} \sigma_n \in X : a_{i+1} = t \,\} \ . \end{array} \tag{5}$$

Hence, it is a subset of the transition relation τ of the program, reduced to the transitions that appear in actual executions only.

Fixpoint characterization. $\mathcal{R}\!\mathit{l}$ and $\mathcal{I}$ can be directly expressed as fixpoints of operators on the transition system, without prior knowledge of $\mathcal{R}$ nor $\mathcal{F}$. We first express $\mathcal{R}\!\mathit{l}$ in fixpoint form as a function of $\mathcal{I}$:

$$\begin{array}{l} \mathcal{R}\!\mathit{l}(t) = \mathrm{lfp}\, R_t(\mathcal{I}), \text{ where} \\ R_t(Y)(X) \stackrel{\text{def}}{=} \pi_t(I) \cup \{\, \pi_t(\sigma') \mid \exists \pi_t(\sigma) \in X : \sigma \stackrel{t}{\rightarrow}_\tau \sigma' \,\} \cup \\ \qquad\qquad \{\, \pi_t(\sigma') \mid \exists \pi_t(\sigma) \in X : \exists t' \neq t : \langle \sigma, \sigma' \rangle \in Y(t') \,\} \\ R_t \text{ has type: } (\mathcal{T} \to \mathcal{P}(\Sigma \times \Sigma)) \to \mathcal{P}(\Sigma_t) \to \mathcal{P}(\Sigma_t) \ . \end{array} \tag{6}$$

The function $R_t(Y)$ is similar to R used to compute the classic reachability semantics $\mathcal{R}$ in (3) of a thread t, but it explores the reachable states by interleaving two kinds of steps: steps from the transition relation of the thread t, and interference steps from other threads (provided in the argument Y).

Secondly, we express $\mathcal{I}$ in fixpoint form as a function of $\mathcal{R}\!\mathit{l}$:

$$\begin{array}{l} \mathcal{I}(t) = B(\mathcal{R}\!\mathit{l})(t), \text{ where} \\ B(Z)(t) \stackrel{\text{def}}{=} \{\, \langle \sigma, \sigma' \rangle \mid \pi_t(\sigma) \in Z(t) \wedge \sigma \stackrel{t}{\rightarrow}_\tau \sigma' \,\} \\ B \text{ has type: } (\prod_{t \in \mathcal{T}} \{t\} \to \mathcal{P}(\Sigma_t)) \to \mathcal{T} \to \mathcal{P}(\Sigma \times \Sigma) \ . \end{array} \tag{7}$$

The function $B(Z)(t)$ collects all the transitions in the transition relation of the thread t starting from a local state in $Z(t)$.

There is a mutual dependency between equations (6) and (7), which we solve using a fixpoint. The following theorem, which characterizes reachable local states $\mathcal{R}\!\mathit{l}$ in a nested fixpoint form, is proved in [18]:

Theorem 1. *$\mathcal{R}\!\mathit{l} = \mathrm{lfp}\, H$, where $H \stackrel{\text{def}}{=} \lambda Z.\, \lambda t.\, \mathrm{lfp}\, R_t(B(Z))$.*

We have the following connection with rely-guarantee proofs $R, G \vdash \{P\}\ s\ \{Q\}$:

- the reachable local states $\mathcal{R}\!\mathit{l}(t)$ correspond to state assertions P and Q;
- the interferences $\mathcal{I}(t)$ correspond to rely and guarantee assertions R and G;
- proving that a given quintuple is valid amounts to checking that $\forall t \in \mathcal{T}$: $R_t(\mathcal{I})(\mathcal{R}\!\mathit{l}(t)) \subseteq \mathcal{R}\!\mathit{l}(t)$ and $B(\mathcal{R}\!\mathit{l})(t) \subseteq \mathcal{I}(t)$, i.e., a post-fixpoint check.

Our fixpoints are, however, constructive and can infer the optimal assertions instead of only checking user-provided ones. Computing $\mathrm{lfp}\, R_t(\mathcal{I})$ corresponds to inferring the state assertions P and Q of a thread t given the interferences $\mathcal{I}$, while computing $\mathrm{lfp}\, H$ infers both the interferences and the state assertions.

Thread-modularity is achieved as each function $R_t(Y)$ only explores the transitions $\langle \sigma,\, t,\, \sigma' \rangle$ generated by the thread t in isolation, while relying on its argument Y to know the transitions of the other threads without having to explore them. Note that $R_t(Y)$ has the same control space, $\mathcal{L}$, as the reachability operator R for t considered in isolation. Given an equation system characterizing $\mathrm{lfp}\, R$ after control partitioning: $\forall \ell \in \mathcal{L} :\ \mathcal{X}_\ell = F_\ell(\mathcal{X}_1, \ldots, \mathcal{X}_n)$, $\mathrm{lfp}\, R_t(Y)$ can be characterized very similarly, as $\forall \ell \in \mathcal{L} :\ \mathcal{X}_\ell = F'_\ell(\mathcal{X}_1, \ldots, \mathcal{X}_n) \cup \mathit{apply}_\ell(Y)(\mathcal{X}_\ell)$, where each F'_ℓ extends F_ℓ to pass auxiliary variables unchanged, but is still defined only from the instructions in inst_t, while apply_ℓ applies interferences from Y at ℓ. Hence, Y being fixed, $R_t(Y)$ is similar to an analysis in isolation of t. Finally, computing $\mathrm{lfp}\, H$ by iteration corresponds to reanalyzing threads with $R_t(Y)$ until Y stabilizes. Although the semantics is concrete and uncomputable, we already retrieve the structure of the thread-modular static analysis from previous work [19] recalled in Sec. 1.

Completeness. Given any $\mathcal{R}l(t)$, we can easily recover $\mathcal{R}$ as $\mathcal{R} = \pi_t^{-1}(\mathcal{R}l(t))$ because π_t is one-to-one. We deduce that Thm. 1 gives a complete method to infer all safety properties of programs.

Example 1. Consider our example from Fig. 1. We do not present $\mathcal{R}l$ and $\mathcal{I}$ in full as these are quite large; we focus on the interferences generated by t_2 at point $3b$. They have the form $\langle\langle (\ell, 3b),\, (x, y)\rangle,\, \langle (\ell, 4b),\, (x, y')\rangle\rangle$ where $y \in [0, 99]$, $y' \in [y+1, y+3]$, and $x = 0$ if $\ell = 1a$, $x \in [0, y]$ if $\ell = 2a$ or $\ell = 5a$, $x \in [0, y-1]$ if $\ell = 3a$, and $x \in [1, y]$ if $\ell = 4a$. Note that, in the full transition relation τ, $Y \leftarrow Y + [1, 3]$ generates a much larger set of transitions: $\langle\langle (\ell, 3b),\, (x, y)\rangle,\, t_2,\, \langle (\ell, 4b),\, (x, y')\rangle\rangle$ where $y' \in [y+1, y+3]$, with no constraint on x nor y.

3 Retrieving Existing Analyses

We now express our former analysis based on non-relational and flow-insensitive interferences as an abstraction of the thread-modular concrete semantics.

3.1 Flow-Insensitive Abstraction

A first abstraction consists in reducing the domains by forgetting as much control information as possible. In order to avoid losing too much precision, individual thread analyses should remain flow-sensitive with respect to their own control location. Thus, on local states, we remove the auxiliary variables using an abstraction $\alpha^{nf}_{\mathcal{R}}$ from $\mathcal{P}(\mathcal{L} \times \mathcal{M}_t)$ to $\mathcal{P}(\mathcal{L} \times \mathcal{M})$ and, on interferences, we remove the control part entirely using an abstraction $\alpha^{nf}_{\mathcal{I}}$ from $\mathcal{P}(\Sigma \times \Sigma)$ to $\mathcal{P}(\mathcal{M} \times \mathcal{M})$:

$$\alpha^{nf}_{\mathcal{R}}(X) \stackrel{\text{def}}{=} \{\, \langle \ell,\, \rho_{|\mathcal{V}} \rangle \mid \langle \ell,\, \rho \rangle \in X \,\}$$
$$\alpha^{nf}_{\mathcal{I}}(Y) \stackrel{\text{def}}{=} \{\, \langle \rho,\, \rho' \rangle \mid \exists L, L' \in \mathcal{C} :\ \langle\langle L,\, \rho \rangle,\, \langle L',\, \rho' \rangle\rangle \in Y \,\}\ .$$

Applying these abstractions to R_t and B gives rise to the following coarser version of (6)–(7), from which we derive an approximate fixpoint semantics $\mathcal{RI}^{nf}$:

$$\begin{array}{l}
\mathcal{RI}^{nf} \stackrel{\text{def}}{=} \mathrm{lfp}\, \lambda Z.\, \lambda t.\, \mathrm{lfp}\, R_t^{nf}(B^{nf}(Z)), \text{ where} \\
B^{nf}(Z)(t) \stackrel{\text{def}}{=} \{\, \langle \rho, \rho' \rangle \mid \exists \ell, \ell' \in \mathcal{L} : \langle \ell, \rho \rangle \in Z(t) \wedge \langle \ell, \rho \rangle \rightarrow_t \langle \ell', \rho' \rangle \,\} \\
R_t^{nf}(Y)(X) \stackrel{\text{def}}{=} R_t^{loc}(X) \cup A_t^{nf}(Y)(X) \\
R_t^{loc}(X) \stackrel{\text{def}}{=} \{\langle e_t, \lambda V.\, 0 \rangle\} \cup \{\, \langle \ell', \rho' \rangle \mid \exists \langle \ell, \rho \rangle \in X : \langle \ell, \rho \rangle \rightarrow_t \langle \ell', \rho' \rangle \,\} \\
A_t^{nf}(Y)(X) \stackrel{\text{def}}{=} \{\, \langle \ell, \rho' \rangle \mid \exists \rho, t' \neq t : \langle \ell, \rho \rangle \in X \wedge \langle \rho, \rho' \rangle \in Y(t') \,\} \ .
\end{array} \tag{8}$$

We retrieve in R_t^{nf} the interleaving of local transitions R_t^{loc} and interferences $A_t^{nf}(Y)$ from Y. Interferences are handled in a flow-insensitive way: if a thread t' can generate a transition between two memory states at some point, we assume that it can happen at any point in the execution of t. $\mathcal{RI}^{nf}$ could be turned into an effective static analysis by reusing stock abstractions for memory states $\mathcal{P}(\mathcal{M})$ and relations $\mathcal{P}(\mathcal{M} \times \mathcal{M})$; however, abstracting relations can be inefficient in the large and we will abstract interferences further in the next section.

Example 2. When computing $\mathcal{RI}^{nf}$ in Fig. 1, we obtain the assertions in Fig. 2. For instance, the interferences from t_2 are $\{\, \langle (x, y), (x', y') \rangle \mid x = x', y \leq y' \leq y+3, x, y \in [0, 99], x \leq y \,\}$. This shows that auxiliary variables and flow-sensitive interferences are not always necessary to infer interesting properties.

Example 3. Consider a program composed of two identical threads reduced to an incrementation: t_1 is $^{(1a)}X \leftarrow X + 1^{(2a)}$ and t_2 is $^{(1b)}X \leftarrow X + 1^{(2b)}$. At $2a$, the state with auxiliary variables is $(pc_2 = 1b \wedge X = 1) \vee (pc_2 = 2b \wedge X = 2)$. It implies $X \in [1, 2]$, but also the fact that t_2 can only increment X when $pc_2 = 1b$, i.e., when $X = 1$. If we forget the auxiliary variables, we also forget the relation between pc_2 and X, and no upper bound on X is stable by the effect of t_2; we get the coarser invariant: $X \geq 1$. We retrieve here a classic result: modular reasoning on concurrent programs is not complete without auxiliary variables.

3.2 Non-Relational Interference Abstraction

After removing control information, interferences live in $\mathcal{P}(\mathcal{M} \times \mathcal{M})$. Such relations provide two flavors of relationality: input-output relationality and relationships between variable values. To recover the analysis described in Sec. 1, we only remember which variables are modified by interferences and their new value, using the following abstraction $\alpha_{\mathcal{I}}^{nr}$ from $\mathcal{P}(\mathcal{M} \times \mathcal{M})$ to $\mathcal{V} \to \mathcal{P}(\mathbb{V})$:

$$\alpha_{\mathcal{I}}^{nr}(Y) \stackrel{\text{def}}{=} \lambda V.\, \{\, x \in \mathbb{V} \mid \exists \langle \rho, \rho' \rangle \in Y : \rho(V) \neq x \wedge \rho'(V) = x \,\} \tag{9}$$

which forgets variable relationships as it abstracts each variable separately, and all but the simplest input sensitivity. Applying this abstraction to the flow-insensitive interference semantics (8), we derive the following, further approxi-

mated fixpoint semantics:

$$\begin{array}{l}\mathcal{Rl}^{nr} \stackrel{\text{def}}{=} \operatorname{lfp} \lambda Z.\, \lambda t.\, \operatorname{lfp} R_t^{nr}(B^{nr}(Z)), \text{ where}\\ B^{nr}(Z)(t) \stackrel{\text{def}}{=} \alpha_{\mathcal{I}}^{nr}(B^{nf}(Z)(t))\\ R_t^{nr}(Y)(X) \stackrel{\text{def}}{=} R_t^{loc}(X) \cup A_t^{nr}(Y)(X)\\ A_t^{nr}(Y)(X) \stackrel{\text{def}}{=} \{\, \langle \ell,\, \rho[V \mapsto v]\rangle \mid \langle \ell,\, \rho\rangle \in X,\, V \in \mathcal{V},\, \exists t' \neq t :\, v \in Y(t')(V) \,\}\ . \end{array} \tag{10}$$

Example 4. When computing $\mathcal{Rl}^{nr}$ in Fig. 1, we obtain the abstract interferences $[X \mapsto [1, 102], Y \mapsto \emptyset]$ for t_1 and $[X \mapsto \emptyset, Y \mapsto [1, 102]]$ for t_2, which is sufficient to infer precise bounds for X and Y, but not to infer the relation $X \leq Y$: when t_1 is analyzed, we allow t_2 to store any value from $[1, 102]$ into Y, possibly decrementing Y and invalidating the relation $X \leq Y$.

Soundness. The soundness of (8) and (10) is stated respectively as $\forall t \in \mathcal{T} :\ \mathcal{Rl}^{nf}(t) \supseteq \alpha_{\mathcal{R}}^{nf}(\mathcal{Rl}(t))$ and $\mathcal{Rl}^{nr}(t) \supseteq \alpha_{\mathcal{R}}^{nr}(\alpha_{\mathcal{R}}^{nf}(\mathcal{Rl}(t)))$. It is a consequence of the general property: $\alpha(\operatorname{lfp} F) \subseteq \operatorname{lfp} F^\sharp$ when $\alpha \circ F \subseteq F^\sharp \circ \alpha$ [5, Thm. 1]. This soundness proof is far simpler than the ad-hoc proof from [19], and we find it more satisfying to construct systematically a sound analysis by abstraction of a concrete semantics rather than presenting an analysis first and proving its soundness *a posteriori*.

Static analysis. We can construct a static analysis based on (10): state sets X are abstracted by associating to each program point an element of a (possibly relational) domain abstracting $\mathcal{P}(\mathcal{M})$; interferences Y associate to each thread and variable in $\mathcal{V}$ an abstract value abstracting $\mathcal{P}(\mathbb{V})$ (for instance, an interval).

Actually, partitioning R_t^{nr} does not give the equations in Sec. 1 and [19], but an alternate form where interferences are applied on all variables at all equations. For instance, instead of $\mathcal{X}_{3a} = [\![\, X < (Y \mid [1, 102]) \,]\!] \mathcal{X}_{2a}$ (2), we would get:

$$\mathcal{X}'_{3a} = [\![\, X < Y \,]\!] \mathcal{X}'_{2a} \cup [\![\, Y \leftarrow [1, 102] \,]\!] \mathcal{X}'_{3a}\ .$$

The first form (2) is more efficient as it takes interferences into account lazily, when reading variables, and it avoids creating extra dependencies in equations. The correctness of this important optimization is justified by the fact that the variables $\mathcal{X}_\ell$ in (2) actually represent local states up to pending interferences. Given the non-relational interferences $Y \in \mathcal{T} \to \mathcal{V} \to \mathcal{P}(\mathbb{V})$, we have: $\mathcal{X}'_\ell = \{\, \rho \mid \exists \rho' \in \mathcal{X}_\ell :\, \forall V :\, \rho(V) = \rho'(V) \vee \exists t' \neq t :\, \rho(V) \in Y(t')(V) \,\}$. The operators $[\![\, \cdot \,]\!]$ are modified accordingly to operate on pairs $\langle \mathcal{X}_\ell,\, Y\rangle$ instead of $\mathcal{X}'_\ell$, as shown in (2) and, more systematically, in [19].

3.3 Unbounded Thread Instances

Up to now, we have assumed that the set $\mathcal{T}$ of threads is finite. Allowing an infinite $\mathcal{T}$ is useful, however, to analyze programs with an unbounded number of

threads. We consider, in this section only, the useful case where $\mathcal{T}$ is composed of a finite set $\mathcal{T}_s$ of syntactic threads, a subset of which $\mathcal{T}_\infty \subseteq \mathcal{T}_s$ can appear more than once (and possibly infinitely often) in $\mathcal{T}$.

The fixpoint formulations of Thm. 1, as well as (8), (10), (11) do not require a finite $\mathcal{T}$; an infinite $\mathcal{T}$ still results in a well defined, if uncomputable, concrete semantics. Finiteness is required to construct an effective static analysis, for three reasons: (*i*) iterating over the threads in Thm. 1 should terminate, (*ii*) control states must be finitely representable, and (*iii*) maps from threads to abstract interferences must be finitely representable. Applying the flow-insensitive abstraction from Sec. 3.1 removes infinite control states, solving (*ii*). As the local states and interferences of two instances of a syntactic thread are then isomorphic, we abstract threads from $\mathcal{T}$ to $\mathcal{T}_s$ by storing information for and iterating over only one instance of each thread in $\mathcal{T}_\infty$, solving (*i*) and (*iii*). This abstraction changes slightly the interpretation of the test $t' \neq t$ when applying interferences. For instance, A_t^{nr} from (10) is changed into:

$$A_t^{nr}(Y)(X) \stackrel{\text{def}}{=} \{\, \langle \ell, \rho[V \mapsto v] \rangle \mid \langle \ell, \rho \rangle \in X \wedge \exists t' : (t \neq t' \vee t \in \mathcal{T}_\infty) \wedge v \in Y(t')(V) \,\}$$

i.e., we consider self-interferences for threads with several instances. This abstraction makes the analysis of programs with an unbounded number of threads possible, but with some limit on the precision. The resulting analysis is uniform: it cannot distinguish between different instances of the same thread nor express properties that depend on the actual number of running threads.

4 Relational Interferences

We now construct novel interference abstractions that enjoy a level of relationality and flow-sensitivity. We apply them on some examples, including Fig. 1.

4.1 Invariant Interferences

The non-relational abstraction of Sec. 3.2 applies interferences independently to each variable, destroying any relationship. To improve the precision, we infer relationships maintained by interferences, i.e., holding both before and after the interference. We use the following abstraction $\alpha_{\mathcal{I}}^{inv}$ from $\mathcal{P}(\mathcal{M} \times \mathcal{M})$ to $\mathcal{P}(\mathcal{M})$, which joins the domain and the codomain of a relation on states:

$$\alpha_{\mathcal{I}}^{inv}(Y) \stackrel{\text{def}}{=} \{\, \rho \mid \exists \rho' : \langle \rho, \rho' \rangle \in Y \vee \langle \rho', \rho \rangle \in Y \,\} \ .$$

Note that this abstraction is able to express relations between variables modified by a thread and variables not modified, such as $X \leq Y$ for t_2 in Fig. 1. However, unlike our former abstraction $\alpha_{\mathcal{I}}^{nr}$ (10), $\alpha_{\mathcal{I}}^{inv}$ forgets which variables have been modified. To construct our analysis, we thus combine them in a *reduced product*:

$$\begin{array}{l} \mathcal{R}l^{rel} \stackrel{\text{def}}{=} \mathrm{lfp}\, \lambda Z.\, \lambda t.\, \mathrm{lfp}\, R_t^{rel}(B^{rel}(Z)), \text{ where} \\ B^{rel}(Z) \stackrel{\text{def}}{=} \langle \lambda t.\, \alpha_{\mathcal{I}}^{nr}(B^{nf}(Z)(t)),\, \lambda t.\, \alpha_{\mathcal{I}}^{inv}(B^{nf}(Z)(t)) \rangle \\ R_t^{rel}(\langle Y^{nr}, Y^{inv} \rangle)(X) \stackrel{\text{def}}{=} R_t^{loc}(X) \cup (A_t^{nr}(Y^{nr})(X) \cap A_t^{inv}(Y^{inv})) \\ A_t^{inv}(Y^{inv}) \stackrel{\text{def}}{=} \{\, \langle \ell, \rho \rangle \mid \ell \in \mathcal{L}, \rho \in Y^{inv}(t) \,\} \ . \end{array} \tag{11}$$

Designing a static analysis derived on this abstraction is straightforward. Interference invariants $Y^{inv}(t) \in \mathcal{P}(\mathcal{M})$ are abstracted in any classic domain (e.g., octagons [17] to express $X \leq Y$). Computing abstract invariants $\alpha^{inv}_{\mathcal{I}}(B^{nf}(Z)(t))$ reduces to computing the abstract join $\cup^\sharp$ of the abstract environments of all the program points of t. Applying abstract interferences in R^{rel}_t reduces to standard abstract set operators $\cup^\sharp$ and $\cap^\sharp$. A drawback is that the optimization used in (2) to apply interferences lazily, only at variable reads, can no longer be performed here, resulting in a much slower analysis. We will alleviate the problem in Sec. 4.4 by using relational invariant interferences only at a few key locations.

4.2 Monotonicity Interference

We now give an example abstraction providing input-output relationality on interferences. In order to complete the analysis of Fig. 1, we propose a simple domain that infers the monotonicity of variables. Interferences are abstracted from $\mathcal{P}(\mathcal{M} \times \mathcal{M})$ to maps $\mathcal{V} \to \mathbb{D}$, where $\mathbb{D} \stackrel{\text{def}}{=} \{\nearrow, \top\}$ indicates whether each variable is monotonic ($\nearrow$) or not ($\top$), hence the following abstraction:

$$\alpha^{mon}_{\mathcal{I}}(Y) \stackrel{\text{def}}{=} \lambda V.\, \text{if } \forall \langle \rho, \rho' \rangle \in Y : \rho(V) \leq \rho'(V) \text{ then } \nearrow \text{ else } \top \ .$$

We would, as before, apply $\alpha^{mon}_{\mathcal{I}}$ to (8) and combine it with (10) or (11) to get a new reduced product fixpoint semantics. We do not present these formulas, but rather focus on the key operations in a static analysis. Firstly, we infer approximate monotonicity information for interferences $\alpha^{mon}_{\mathcal{I}}(B^{nf}(Z)(t))$: during the analysis of t, we gather, for each variable V, the set of all the assignments into V, and set V to $\nearrow$ if they all have the form $V \leftarrow V + e$ where e evaluates to positive values, and set V to $\top$ otherwise. Secondly, we use monotonicity information when applying interferences after an affine comparison operator $e_1 \leq e_2$: if all the variables in e_1 and e_2 have monotonic interferences and appear in e_2 (resp. e_1) with positive (resp. negative) coefficient, then $[\![e_1 \leq e_2]\!]$ can be applied *after* applying the non-relational interferences. In Fig. 1, for instance, we would get: $\mathcal{X}_{3a} = [\![X < Y]\!]([\![X < (Y | [1, 102])]\!]\mathcal{X}_{2a})$ instead of (2). Using these abstractions, we can prove that, at the end of the program, $X \leq Y$ holds. The domain is inexpensive as it associates a single binary information to each variable.

4.3 Trace Abstractions

Although state semantics are complete for safety properties, it is often useful to start from a more expressive, trace concrete semantics to embed some information about the history of computations in subsequent abstractions. A classic example is trace partitioning [16] where, in order to avoid or delay abstract joins (which often cause imprecision), a disjunction of abstract elements is maintained, each one keyed to an abstraction of sequences of control locations leading to the current location (such as, which branch was followed in the previous test).

We can construct a trace version of the thread-modular concrete semantics from Sec. 2.3 and its abstractions from Sec. 3 and Sec. 4 by upgrading R_t to

t_1	t_2	t_3
while random **do**	**while** random **do**	**while** random **do**
if $H < 10,000$ **then**	$C \leftarrow H$	**if** random **then** $T \leftarrow 0$
$H \leftarrow H + 1$	**done**	**else** $T \leftarrow T + (C - L)$ **endif**
endif		$L \leftarrow C$
done		**done**

Fig. 4. Clock example motivating history-sensitive invariants.

append states to partial executions. Applying this idea, for instance, to the flow-insensitive semantics of Sec. 3.1 which is the basis of our abstractions, we get:

$$
\begin{array}{l}
R_t^{nf}(Y)(X) \stackrel{\text{def}}{=} \{\langle e_t, \lambda V.\, 0\rangle\} \cup \\
\{\, \langle\langle \ell_0, \rho_0\rangle, \ldots, \langle \ell, \rho\rangle, \langle \ell', \rho'\rangle\rangle \mid \langle\langle \ell_0, \rho_0\rangle, \ldots, \langle \ell, \rho\rangle\rangle \in X, \langle \ell, \rho\rangle \rightarrow_t \langle \ell', \rho'\rangle \,\} \cup \\
\{\, \langle\langle \ell_0, \rho_0\rangle, \ldots, \langle \ell, \rho\rangle, \langle \ell, \rho'\rangle\rangle \mid \langle\langle \ell_0, \rho_0\rangle, \ldots, \langle \ell, \rho\rangle\rangle \in X, \exists t' \neq t : \langle \rho, \rho'\rangle \in Y(t') \,\} \\
R_t^{nf} \text{ has type: } (\mathcal{T} \to \mathcal{P}(\mathcal{M} \times \mathcal{M})) \to \mathcal{P}((\mathcal{L} \times \mathcal{M})^*) \to \mathcal{P}((\mathcal{L} \times \mathcal{M})^*)
\end{array}
$$

From the point of view of thread t, an execution is composed of a sequence of local states (without auxiliary variables) in $\mathcal{L} \times \mathcal{M}$; it starts at its entry point e_t and is extended by either an execution step $\langle \ell, \rho\rangle \rightarrow_t \langle \ell', \rho'\rangle$ of thread t or an interference form Y that leaves its local control location unchanged. The semantics can be translated, as before, into an equation system resembling that of the sequential analysis of the thread t in isolation (Fig. 2) by associating to each control location $\ell \in \mathcal{L}$ a variable $\mathcal{X}_\ell$ that stores (an abstraction of) the partial traces that end in the control state ℓ. A natural consequence is the ability to use classic trace partitioning techniques intended for sequential programs [16] when analyzing each thread, independently from interferences.

We illustrate the use for concurrency-specific trace abstractions on the example in Fig. 4. This program, inspired from an actual software, contains three threads: t_1 increments a clock H, t_2 samples the clock in a latch C, and t_3 accumulates elapsed durations with respect to C into T. We wish to infer that $T \leq L \leq C \leq H$, i.e., the accumulated time does not exceed the total elapsed time. This information can be inferred from the monotonicity of L, C, and H; for instance, the assignment $L \leftarrow C$ where C is monotonic implies that $L \leq C$ holds despite interferences. However, the monotonicity domain of Sec. 4.2 can only infer the monotonicity of H, not that of C. In particular, in that domain, it would be unsound for the semantics $[\![\, C \leftarrow H \,]\!]^\sharp \mathcal{X}^\sharp$ to deduce the monotonicity of C from that of H. Otherwise, in the following example, if both H and H' were monotonic, we would deduce wrongly that C is also monotonic:

$$\textbf{if random then } C \leftarrow H \textbf{ else } C \leftarrow H' \textbf{ endif} \quad . \tag{12}$$

We need to infer a stronger property, namely that the sequence of values stored into C is a subsequence of the values stored into H. This is implied by the assignment $C \leftarrow H$ but not by (12), and it implies the monotonicity of C. The subsequence abstraction $\alpha^{sub}_{\mathcal{R}}$ is an abstraction from sequences of states local to

a thread, i.e., in $\mathcal{P}((\mathcal{L} \times \mathcal{M})^*)$, to $\mathcal{V} \to \mathcal{P}(\mathcal{V})$, which is defined as:

$$\alpha^{sub}_{\mathcal{R}}(X)(V) \stackrel{\text{def}}{=} \{\, W \mid \forall \langle \langle \ell_0, \rho_0 \rangle, \ldots, \langle \ell_n, \rho_n \rangle \rangle \in X : \exists i_0, \ldots, i_n : \\ \forall k : i_k \leq k \wedge i_k \leq i_{k+1} \wedge \forall j : \rho_j(V) = \rho_{i_j}(W) \,\} \ .$$

It associates to each variable V the set of variables W it can be considered a subsequence of. Some meaningful subsequence information can be inferred by only looking at simple assignments of the form $V \leftarrow W$. The domain is inexpensive; yet, in a reduced product with the monotonicity domain, it allows inferring all the properties required to precisely analyze Fig. 4.

4.4 Lock Invariants

We now enrich our programs with mutual exclusion locks, so-called *mutexes*, to provide thread synchronization. We assume a finite set $\mathbb{M}$ of mutexes and two instructions: **lock**(m) and **unlock**(m), to respectively acquire and release a mutex $m \in \mathbb{M}$. The semantics is that, at any time, each mutex can be held by at most one thread. To reflect this, our transition systems are enriched as follows:

- $\Sigma \stackrel{\text{def}}{=} \mathcal{C} \times \mathcal{M} \times \mathcal{S}$: states $\langle L, \rho, s \rangle \in \Sigma$ include a new *scheduler component* $s \in \mathcal{S} \stackrel{\text{def}}{=} \mathbb{M} \to (\mathcal{T} \cup \{\bot\})$ which remembers, for each mutex, which thread holds it, if any, or $\bot$ if the mutex is unlocked;
- $I \stackrel{\text{def}}{=} \{\langle \lambda t.\, e_t,\ \lambda V.\, 0,\ \lambda m.\, \bot \rangle\}$: all mutexes are initially unlocked;
- instructions $\langle \ell,\ \textbf{lock}(m),\ \ell' \rangle \in \mathit{inst}_t$ and $\langle \ell,\ \textbf{unlock}(m),\ \ell' \rangle \in \mathit{inst}_t$ generate respectively the following transition sets:

$$\{\, \langle L[t \mapsto \ell],\ \rho,\ s \rangle \xrightarrow{t}_{\tau} \langle L[t \mapsto \ell'],\ \rho,\ s[m \mapsto t] \rangle \mid \langle L,\ \rho,\ s \rangle \in \Sigma,\ s(m) = \bot \,\}$$
$$\{\, \langle L[t \mapsto \ell],\ \rho,\ s \rangle \xrightarrow{t}_{\tau} \langle L[t \mapsto \ell'],\ \rho,\ s[m \mapsto \bot] \rangle \mid \langle L,\ \rho,\ s \rangle \in \Sigma,\ s(m) = t \,\} \ .$$

Consider the example in Fig. 5.(a), where two identical threads increment a counter X up to 100. The use of a mutex m ensures that X is not modified between the test $X < 100$ and the subsequent incrementation. Ignoring the mutex in the concrete would give a range of $[0, 101]$ instead of $[0, 100]$ and, with flow-insensitive interferences, we would not find any upper bound on X (the case is similar to Ex. 3). We now show that partitioning with respect to the scheduler state (a technique introduced in [19]) can make the analysis more precise.

Returning to our most concrete, thread-modular semantics of Sec. 2.3, we enrich the local states $\mathcal{R}\!\mathit{l}$ of a thread t with information on the set of locks it holds: $\mathcal{R}\!\mathit{l}(t) \in \Sigma_t \stackrel{\text{def}}{=} \mathcal{L} \times \mathcal{M}_t \times \mathcal{P}(\mathbb{M})$. We define $\mathcal{R}\!\mathit{l}(t) \stackrel{\text{def}}{=} \pi_t(\mathcal{R})$ where the projection π_t to local states from (4) is extended to handle $s \in \mathcal{S}$ as follows:

$$\pi_t(\langle L,\ \rho,\ s \rangle) \stackrel{\text{def}}{=} \langle L(t),\ \rho\,[\forall t' \neq t : \mathit{pc}_{t'} \mapsto L(t')]\,,\ s^{-1}(t) \rangle \ . \tag{13}$$

Moreover, we distinguish two kinds of interferences in $(\mathcal{C} \times \mathcal{M}) \times (\mathcal{C} \times \mathcal{M})$:

- interferences from t that do not change the set M of mutexes held by t:

$$\mathcal{I}^u(t)(M) \stackrel{\text{def}}{=} \\ \{\, \langle \langle L_i,\ \rho_i \rangle,\ \langle L_{i+1},\ \rho_{i+1} \rangle \rangle \mid \\ \exists \langle L_0,\ \rho_0,\ s_0 \rangle \xrightarrow{a_1} \cdots \langle L_n,\ \rho_n,\ s_n \rangle \in \mathcal{F} : a_i = t,\ s_{i-1}^{-1}(t) = s_i^{-1}(t) = M \,\}$$

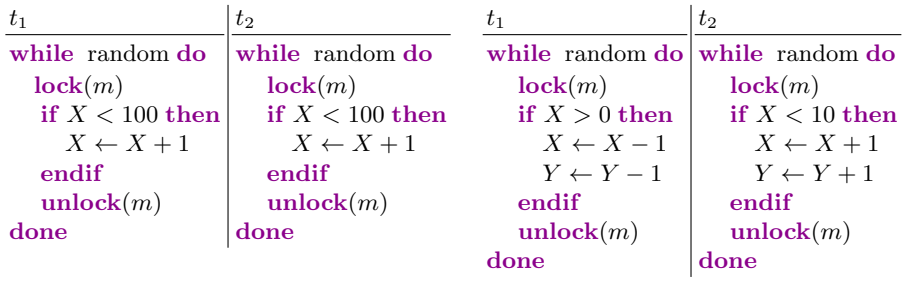

Fig. 5. (a) two identical threads concurrently incrementing a counter X protected by a lock m; and (b) an abstract producer/consumer with resources X and Y.

– critical sections that summarize a sequence of transitions beginning with **lock**(m) by t, ending with **unlock**(m) by t, and containing transitions from any threads in-between:

$$\begin{array}{l}\mathcal{I}^s(t)(m) \stackrel{\text{def}}{=} \\ \quad \{\, \langle\langle L_i,\, \rho_i\rangle,\, \langle L_j,\, \rho_j\rangle\rangle \mid \exists \langle L_0,\, \rho_0,\, s_0\rangle \xrightarrow{a_1} \cdots \langle L_n,\, \rho_n,\, s_n\rangle \in \mathcal{F} : \\ \quad\quad i < j,\, s_i(m) = s_j(m) = \bot,\, \forall k :\, i < k < j \implies s_k(m) = t \,\}\ .\end{array}$$

As in (6), the semantics of a thread t is computed by interleaving execution steps from the thread and from interferences. However, due to mutual exclusion, interferences in $\mathcal{I}^u(t')(M')$ cannot fire from a local state $\langle \ell,\, \rho,\, M\rangle$ of a thread $t \neq t'$ when $M \cap M' \neq \emptyset$. For instance, in Fig. 5.(a), no interference generated by $X \leftarrow X + 1$ in t_2 can run in t_1 between **lock**(m) and **unlock**(m). Moreover, mutual exclusion ensures that, if some interference in $\mathcal{I}^u(t')(M')$ such that $m \in M'$ fires from a local state $\langle \ell,\, \rho,\, M\rangle$ before t locks m, then t' must finish its critical section protected by m before t can lock m; hence, the interference is subsumed by a transition from $\mathcal{I}^s(t')(m)$. If the program is well-synchronized, i.e., every access to a variable V is protected by a mutex associated to the variable, then all the interferences are included in $\mathcal{I}^s(t')(m)$. Hence, it makes sense to abstract $\mathcal{I}^u(t')(M')$ in a coarse way, and use these interferences only in case (hopefully rare) of an unsynchronized access (i.e., a data race), while a more precise abstraction of $\mathcal{I}^s(t')(m)$ is used for well-synchronized accesses.

Following Sec. 3.2, we use a flow-insensitive and non-relational abstraction of $\mathcal{I}^u$, i.e.: $\mathcal{I}^{nr,u}(t')(M') \stackrel{\text{def}}{=} \alpha^{nr}_{\mathcal{I}}(\alpha^{nf}_{\mathcal{I}}(\mathcal{I}^u(t')(M')))$. By partitioning local thread states with respect to the program location $\ell \in \mathcal{L}$ and the mutexes held $M \in \mathcal{P}(\mathbb{M})$, and applying the optimization technique that an equation variable $\mathcal{X}_{\ell,M}$ represents a set of local states up to the pending interferences in $\mathcal{I}^{nr,u}(t')(M')$, assignments $V \leftarrow e$ give rise to equations of the form $\mathcal{X}_{\ell,M} = [\![V \leftarrow e]\!]\mathcal{X}_{\ell',M}$, where e is modified to incorporate all the interferences in $\mathcal{I}^{nr,u}(t')(M')$ such that $t' \neq t$ and $M \cap M' = \emptyset$ on variables appearing in e, and similarly for tests. An abstraction of the interferences in $\mathcal{I}^s(t')(m)$ is incorporated when t locks m.

When $\langle \ell, \textbf{lock}(m), \ell' \rangle \in inst_t$, we generate, for each $M \in \mathcal{P}(\mathbb{M})$, an equation:

$$\mathcal{X}_{\ell',M\cup\{m\}} = \mathcal{X}_{\ell,M\setminus\{m\}} \cup \bigcup \{\, apply(\mathcal{I}^s(t')(m))(\mathcal{X}_{\ell,M\setminus\{m\}}) \mid t' \neq t \,\}$$

where the exact definition of *apply* depends on the abstraction chosen to approximate $\mathcal{I}^s$ and is discussed below. An $\textbf{unlock}(m)$ instruction generates the simple equation: $\mathcal{X}_{\ell',M\setminus\{m\}} = \mathcal{X}_{\ell,M\cup\{m\}}$.

Example 5. As Fig. 5.(a) is well synchronized, the interference from $\mathcal{I}^{nr,u}$ on the assignment $X < 100$ and the test $X \leftarrow X + 1$ are empty. When choosing the flow-insensitive non-relational abstraction from Sec. 3.2 for $\mathcal{I}^s$ as well as $\mathcal{I}^u$, the interference caused by the critical section on both threads is $[X \mapsto [1, 100]]$, i.e., any value in $[1, 100]$ can be stored into X. The *apply* function is given by A^{nr}_t from (10). The resulting equation for $\textbf{lock}(m)$ is thus: $\mathcal{X}_{\ell',\{m\}} = [\![X \leftarrow [1, 100]]\!] \mathcal{X}_{\ell,\emptyset} \cup \mathcal{X}_{\ell,\emptyset}$. This is sufficient to infer that X is always in $[0, 100]$. Recall that an analysis with the same non-relational abstraction but without interference partitioning would not find any bound on X.

To gain more precision, the invariance abstraction from Sec. 4.1 can be used for $\mathcal{I}^s$. The resulting analysis will infer relational properties of variables that are guaranteed to hold outside critical sections, but are possibly broken inside. We call them *lock invariants* by analogy with class invariants: in a well synchronized program, threads cannot observe program states where the invariant is broken by the action of another thread. Unlike the method of Sec. 4.1, we do not need to apply complex relational operations at all program points: the interferences are inferred by joining the environments only at **lock** and **unlock** instructions, while the *apply* function that incorporates interferences, given by R^{rel}_t (11), is only applied at **lock** instructions, which ensures an efficient analysis.

Example 6. Consider the program in Fig. 5.(b) that models an abstract producer/consumer, where X and Y denote the amount of resources. The non-relational interference analysis is sufficient to prove that X is bounded thanks to the explicit tests on X, but it cannot find any bound on Y. Using the invariant interference abstraction parameterized with the octagon domain [17], it is possible to infer that $X = Y$ is a lock invariant. This information automatically infers a bound on Y from the bound on X.

4.5 Weakly Consistent Memories

In the previous sections, we have assumed a sequentially consist model of execution [14]. Actually, computers may execute programs under more relaxed models, where different threads may hold inconsistent views of the memory [2], hence creating behaviors outside the sequentially consistent ones.

We justify informally the soundness of our interference analysis with respect to weakly consistent memories as follows: firstly, as proved in [19], flow-insensitive non-relational interference abstractions are naturally sound in a wide variety of memory models, hence our abstraction $\mathcal{I}^{nr,u}$ is sound; secondly, **lock** and **unlock**

monotonicity domain	relational lock invariants	analysis time	memory	iterations	alarms
×	×	25h 26mn	22 GB	6	4616
✓	×	30h 30mn	24 GB	7	1100
✓	✓	110h 38mn	90 GB	7	1009

Fig. 6. Experimental results for AstréeA on our 1.7 Mlines 15 threads code target.

instructions provide memory synchronization points, so that any sound abstraction of $\mathcal{I}^s$ is also sound in relaxed models. Finally, the monotonicity abstractions proposed in Sec. 4.2 and Sec. 4.3 only rely on the ordering of sequences of assignments to each variable independently, and so, are sound in any memory model that guarantees it (such as the widespread Total Store Ordering).

5 Experimental Results

We have implemented our method in AstréeA, a static analyzer prototype [19] that extends Astrée. The Astrée analyzer [3] checks for run-time errors in embedded synchronous C programs. A specificity of Astrée is its specialization and design by refinement: starting from an efficient and coarse interval analyzer, we added new abstract domains until we reached the zero false alarm goal (i.e., a proof of absence of run-time error) on a pre-defined selection of target industrial codes, namely avionic control-command fly-by-wire software. The new abstractions are made tunable by end-users, to adapt the analysis to different codes in the same family. The result is an efficient and precise (few alarms) analyzer on general embedded C code, which is extremely precise (no alarm) on a restricted family of avionic embedded codes, and which is usable in industrial context [9].

The AstréeA prototype extends Astrée to analyze concurrent C programs. As Astrée, it does not support dynamic memory allocation nor recursivity (function calls are inlined for maximum precision) by design, as these are forbidden in most embedded platforms. It is also a specialized analyzer. Our main target is a large avionic code composed of 15 threads (without dynamic thread creation) totaling 1.7 Mlines of C and running under a real-time operating system based on the ARINC 653 specification; it performs a mix of numeric computations, reactive computations, network communications, and string formatting. More information on Astrée, AstréeA, ARINC 653, and our target application can be found in [3,20,19].

Using the design by refinement that made the success of Astrée, we started [19] with a simple analysis that reuses Astrée's sequential analysis and domains (including domains for machine integers, floats, pointers, relational domains, etc.), on top of which we added the simple non-relational abstraction of thread interferences from Sec. 3.2. The analysis time, peak memory consumption, as well as the number of iterations to stabilize interferences and the number of alarms are reported in the first line of Fig. 6. The framework presented in this article was developed when it became clear that non-relational interferences were too

coarse to infer the properties needed to remove some false alarms (an example of which was given in Fig. 4). The second line of Fig. 6 presents experimental results after adding the monotonicity and subsequence domains of Sec. 4.2 and Sec. 4.3, while the last line also includes the relational lock invariants from Sec. 4.4. The monotonicity domain provides a huge improvement in precision for a reasonable cost; this is natural as it is a specialized domain designed to handle very specific uses of clocks and counters in our target application. The relational lock invariant domain can remove a few extra alarms, but it is not as well tuned and efficient yet: for now, it inherits without modification Astrée's relational domains and packing strategies (i.e., choosing *a priori* which variables to track in a relational way). Nonetheless, relational lock invariants are versatile and general purpose by nature; we believe that, by parameterizing them in future work with adequate relational domains and packing strategies, they have the potential to further improve the precision at a more reasonable cost.

Implementation-wise, adding these new domains did not require a large effort; in particular, the overall architecture of AstréeA and existing domains required only marginal changes. We benefited from casting the former analysis as an abstraction of a more concrete semantics, from which alternate abstractions could be derived and combined under a unified thread-modular analysis framework.

6 Conclusion

We have proposed a framework to design thread-modular static analyses that are able to infer and use relational and history-sensitive properties of thread interferences. This was achieved by a reinterpretation of Jones' rely-guarantee proof method as a constructive fixpoint semantics, which is complete for safety properties and can be abstracted into static analyses in a systematic way, thus following the abstract interpretation methodology. We then proposed several example abstractions tailored to solve specific problems out of the reach of previous, non-relational interference abstractions, and motivated by actual analysis problems. We presented encouraging results on the analysis of an embedded industrial C code using the AstréeA prototype analyzer.

AstréeA is very much a work in progress, and further work is required in order to improve its precision (towards the zero false alarm goal) and widen its application scope (to analyze more classes of embedded concurrent C software and hopefully enable a deployment in industry). This will require the design of new abstractions, in particular to improve our relational lock invariant inference. Another interesting promising area is the development of trace-related abstractions, with potential generalization to inferring maximal trace properties, which includes liveness properties, in a thread-modular way.

References

1. H. Amjad and R. Bornat. Towards automatic stability analysis for rely-guarantee proofs. In *VMCAI'11*, volume 5403 of *LNCS*, pages 14–28. Springer, 2009.

2. M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In *POPL'10*, pages 7–18. ACM, Jan. 2010.
3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI'03*, pages 196–207. ACM, June 2003.
4. J.-L. Carré and C. Hymans. From single-thread to multithreaded: An efficient static analysis algorithm. Technical Report arXiv:0910.5833v1, EADS, Oct. 2009.
5. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1–2):47–103, 2002.
6. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *ISP'76*, pages 106–130. Dunod, Paris, France, 1976.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252. ACM, Jan. 1977.
8. P. Cousot and R. Cousot. Invariance proof methods and analysis techniques for parallel programs. In *Automatic Program Construction Techniques*, chapter 12, pages 243–271. Macmillan, New York, NY, USA, 1984.
9. D. Delmas and J. Souyris. Astrée: from research to industry. In *SAS'07*, volume 4634 of *LNCS*, pages 437–451. Springer, Aug. 2007.
10. C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular verification of multithreaded programs. *Theoretical Computer Science*, 338(1–3):153–183, 2005.
11. C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN'03*, volume 2648 of *LNCS*, pages 213–224. Springer, 2003.
12. B. Jeannet. Relational interprocedural verification of concurrent programs. *Software & Systems Modeling*, 12(2):285–306, 2013.
13. C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, Jun. 1981.
14. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. In *IEEE Trans. on Computers*, volume 28, pages 690–691. IEEE Comp. Soc., Sep. 1979.
15. A. Malkis, A. Podelski, and A. Rybalchenko. Thread-modular verification is cartesian abstract interpretation. In *ICTAC'06*, volume 4281 of *LNCS*, pages 183–197, 2006.
16. L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzer. In *ESOP'05*, volume 3444 of *LNCS*, pages 5–20. Springer, 2005.
17. A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
18. A. Miné. Static analysis by abstract interpretation of sequential and multi-thread programs. In *MOVEP'12*, pages 35–48, Dec. 2012.
19. A. Miné. Static analysis of run-time errors in embedded real-time parallel C programs. *Logical Methods in Computer Science*, 8(26):63, Mar. 2012.
20. A. Miné. Static analysis by abstract interpretation of concurrent programs. Habilitation report, École normale supérieure, May 2013.
21. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, Dec. 1976.
22. M. C. Rinard. Analysis of multithreaded programs. In *SAS'01*, volume 2126 of *LNCS*, pages 1–19. Springer, Jul 2001.
23. C. B. Watkins and R. Walter. Transitioning from federated avionics architectures to integrated modular avionics. In *DASC'07*, volume 2.A.1, pages 1–10. IEEE, Oct. 2007.