

Precise Thread-Modular Abstract Interpretation of Concurrent Programs Using Relational Interference Abstractions

Raphaël Monat, Antoine Miné

▶ **To cite this version:**

Raphaël Monat, Antoine Miné. Precise Thread-Modular Abstract Interpretation of Concurrent Programs Using Relational Interference Abstractions. Verification, Model Checking, and Abstract Interpretation (VMCAI) 2017, Jan 2017, Paris, France. pp.386-404, 10.1007/978-3-319-52234-0_21 . hal-01490178

HAL Id: hal-01490178
https://inria.hal.science/hal-01490178

Submitted on 14 Mar 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Precise Thread-Modular Abstract Interpretation of Concurrent Programs using Relational Interference Abstractions

Raphaël Monat[1] and Antoine Miné[2]

[1] École Normale Supérieure de Lyon and École Normale Supérieure, France
raphael.monat@ens-lyon.org
[2] Sorbonnes Universités, UPMC Univ Paris 6,
Laboratoire d'informatique de Paris 6 (LIP6)
4, pl. Jussieu, 75005 Paris, France
antoine.mine@lip6.fr

Abstract. We present a static analysis by abstract interpretation of numeric properties in multi-threaded programs. The analysis is sound (assuming a sequentially consistent memory), parameterized by a choice of abstract domains and, in order to scale up, it is modular, in that it iterates over each thread individually (possibly several times) instead of iterating over their product. We build on previous work that formalized rely-guarantee verification methods as a concrete, fixpoint-based semantics, and then apply classic numeric abstractions to abstract independently thread states and thread interference. This results in a flexible algorithm allowing a wide range of precision versus cost trade-offs, and able to infer even flow-sensitive and relational thread interference. We implemented our method in an analyzer prototype for a simple language and experimented it on several classic mutual exclusion algorithms for two or more threads. Our prototype is instantiated with the polyhedra domain and employs simple control partitioning to distinguish critical sections from surrounding code. It relates the variables of all threads using polyhedra, which limits its scalability in the number of variables. Nevertheless, preliminary experiments and comparison with ConcurInterproc show that modularity enables scaling to a large number of thread instances, provided that the total number of variables stays small.

Keywords: Program verification, Concurrent programs, Abstract interpretation, Thread-modular analyses, Rely-guarantee methods, Numeric invariant generation

1 Introduction

In order to exploit the full potential of multi-core processors, it is necessary to turn to parallel programming, a trend also followed in critical application domains, such as avionics and automotive. Unfortunately, concurrent programs are difficult to design correctly, and difficult to verify. In particular, the large

space of possible executions makes it impractical for tests to achieve a good coverage. Likewise, formal methods that do not cover the whole range of possible executions (e.g. context bounded methods [30]) can miss errors. In this article, we study sound static analysis methods based on abstract interpretation [7] that consider a superset of all possible executions, and are thus suitable for the certification of concurrent critical software. The concurrency model we consider is that of multi-thread software, with arbitrary preemption and a global, shared memory. This model is challenging to analyze as thread instructions that can actually execute concurrently are not apparent at the syntactic level, and every access to a global variable can actually be a communication between threads.

In the last few years, sound static analysis methods based on abstract interpretation [7] have established themselves as successful techniques to verify non-functional correctness properties (e.g. the absence of run-time error) on *sequential* programs with a large *data space* and complex numeric computations. Our aim is to extend these sound methods to multi-threaded concurrent programs by tackling the large *control space* inherent to these programs. We focus in particular on scaling in the number of threads. This is achieved by combining precise, relational abstractions with thread-modular analysis methods. More precisely, we base our work on the thread-modular concrete semantics introduced in [27], but our abstractions differ as we employ polyhedral abstractions [9] and control partitioning [8] to represent fully-relational flow-sensitive thread-local invariants and thread interference relations, aiming at a higher precision.

Thread-Modular Analysis. A classic view of abstract interpretation consists in propagating an abstract representation of a set of memory states (e.g., a box or polyhedron) along the control-flow graph, using loop acceleration such as widening until a stable abstract element is found [4]. While this technique can be extended easily to concurrent programs by constructing a product control-flow graph of threads, it does not scale up for a large number of threads due to the combinatorial explosion of the control space. Another view consists in a more literal interpretation, defined as usual by induction on the program syntax, but using abstract representations of collected program states. This flavor of abstract interpretation is much more efficient in memory (it does not store an abstract element for each control location) and has been exploited for the efficient analysis of large sequential programs [2]. The extension to concurrent programs is more difficult than for control-flow graphs as the set of interleavings of thread executions cannot be defined conveniently by induction on the syntax. In this article, we consider thread-modular analysis methods, which decompose the analysis of a multi-threaded program into the analysis of its individual threads. They combine two benefits: the complexity of analyzing a program is closer to that of analyzing the sum of its threads than their products, and existing efficient analysis methods for sequential programs (such as abstract interpretation by induction on the syntax) can be reused without much effort on concurrent ones.

The main difficulty in designing a thread-modular analysis is to soundly and precisely account, during the analysis of one thread, for the effect of the other threads. One solution would be to specify a range of possible values for each

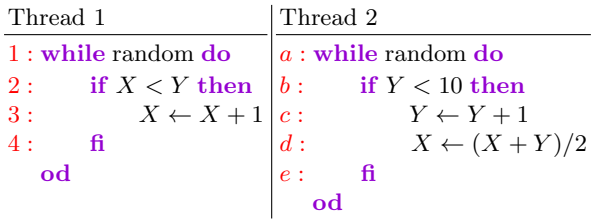


Fig. 1: An example of concurrent program.

shared variable, but this puts a large burden on the programmer. Moreover, as shown in the following paragraph, range information is not always sufficient. We solve both problems by providing a method capable of *automatically* inferring *relations* over the shared variables, whose shape is completely specified by the choice of an abstract domain.

Simple Interference. Consider, as motivating example, the program in Fig. 1. Thread 1 repeatedly increments X while it is less than Y. Thread 2 repeatedly increments Y until 10 and computes into X the average of X and Y. Initially, $X \leftarrow 0$ and $Y \leftarrow 1$. Although this example is artificial for the sake of presentation, experimental evidence (Sec. 4) shows that the precision brought by our analysis is also required when analyzing real-world algorithms, such as Lamport's Bakery algorithm and Peterson's mutual exclusion algorithm. In the following, we assume a sequentially consistent execution model [22].

Existing thread-modular analyses [11,21,12,15,5,27] first analyze each thread in isolation, then gather interference from this first, unsound analysis, and perform a second analysis of each thread in the context of this interference; these analyses uncover more behaviors of threads, hence more interference, so that other rounds of analyses with increasing sets of interference will be performed, until the interference set reaches a fixpoint (possibly with the help of a widening), at which point the analysis accounts for all the possible behaviors of the concurrent program. In [5,25], the interference corresponds to the set of values that each thread can store into each variable, possibly further abstracted (e.g., as intervals). A thread reading a variable may either read back the last value it stored into it, or one of the values from the interference set of other threads. These analyses are similar to rely-guarantee reasoning [20], a proof method which is precise (it is relatively complete), but relies on the user to provide interference as well as program invariants. However, we consider here automated analyses, which are necessarily incomplete, but parameterized by a choice of abstractions.

In the example, using interval abstractions, the first analysis round uncovers the interference $X \leftarrow [0,1], Y \leftarrow [1,1]$ from Thread 1, and $X \leftarrow [0,5], Y \leftarrow [1,10]$ from Thread 2. A second round uncovers $X \leftarrow [1,10]$, at which point the set of interference is stable. When running a final time every thread analysis using these interference, we get that X and Y are bounded by $[0,10]$. However, the

relation $X \leq Y$ that additionally holds cannot be found with the method of [5,25]. The reason is that interference is handled in a non-relational way: the analysis cannot express (and so, infer) that, when $X \leq Y$ is established by a thread, the relation is left invariant by the interference of the other thread.

Relational Interference. In the present article, we enrich the notion of interference to allow such relational information to be inferred and exploited during the analysis. More precisely, following [27], we see interference generated by an instruction of a thread as a relation linking the variable values before the execution of the instruction (denoted x, y) and the variable values after its execution (denoted as x', y'). Our contribution is then to use relational domains to represent both relations between variables (such as $x \leq y \wedge x' \leq y'$ when $X \leq Y$ remains invariant) and input-output relations (such as $x' = x + 1$ for $X \leftarrow X + 1$). Furthermore, we distinguish interference generated at different control states, in our case, a pair (l, l') of control locations of Thread 1 and Thread 2, to enable flow-sensitive interference. In the example of Fig. 1, we have a piece of interference from Thread 1 being $(l, l') = (3, 4)$; $x < y$; $x' = x+1$; $y = y'$. For Thread 2, the interference from program point d to e is: $y \leq 10$; $x \leq y$; $y = y'$; $2x' = x + y$. We note that the global invariant $X \leq Y$ is preserved. To achieve an effective analysis, the states and relations manipulated by the semantics are actually abstracted, using classic numeric abstract domains (such as polyhedra [9]), as well as partitioning of control locations. The analysis is thus parametric, and allows a large choice of trade-offs between cost, precision, and expressiveness. Our prototype implementation uses the Apron [19] and BddApron [17] libraries. The correctness of the analysis is proved in the thread-modular abstract interpretation framework introduced in [27]. Yet, we stress on the fact that the abstract analysis subsequently derived in [27] is not able to precisely analyze the example from Fig. 1, as it does not exploit the capabilities of relational numeric domains to represent interference as relations, while we do.

Contribution. To sum up, we build on previous theoretical work [27] to develop a thread-modular static analysis by abstract interpretation of a simple numeric language, which goes beyond the state of the art [11,21,12,15,5,27] by being fully relational and flow-sensitive. In [27], a small degree of flow-sensitivity and relationality was added to an existing scalable analysis in the form of specialized abstract domains designed to remove specific false alarms, following the design by refinement methodology of Astrée [3]. Here, starting from the same thread-modular semantics, we build a different abstract analysis targeting small but intricate algorithms that require a higher level of precision than [27]. Merging our analysis with [27] in order to achieve a higher precision only for programs parts that require it while retaining overall scalability remains a future work.

We present examples and experimental results to demonstrate that the analysis is sufficiently precise to prove the correctness of small but non-trivial mutual exclusion algorithms, and that it scales well, allowing the analysis of a few hundreds (albeit small) threads in a matter of minutes. An experimental comparison

with ConcurInterproc [18] shows that our approach is several orders of magnitude more scalable, with a comparable level of precision.

Limitations. We assume a sequentially consistent execution model [22], i.e., the execution of a concurrent program is an interleaving of executions of its threads. We ignore the additional difficulty caused by weakly consistent memories [1]; indeed, we believe that an extension to weakly consistent memories is possible, but orthogonal to our work. Our analysis is nevertheless useful with respect to a model obeying the "data-race freedom guarantee" as it can be easily extended to detect data races (a race is simply a read or write instruction at a control-point where some interference can occur). An additional limitation is that we consider a fixed number of threads. Our method can be extended to consider several instances of threads, possibly an unbounded number, while keeping a fixed, finite number of variables in the abstract. We would employ a uniform abstraction: firstly, thread variables are replaced with summary abstract variables that account for all the possible instances of that variable; secondly, we add interference from a thread to itself, to account for the effect of one instance of a thread on another instance of the same thread. This would achieve a sound, but uniform analysis (unlike [12]). More importantly, our implementation is only a limited prototype. We analyze small programs written in a basic language with no functions and only numeric variables. Yet, these programs are inspired from actual algorithms and challenging parts of real programs. We focus on scalability in the number of threads, and did not include in our prototype state-of-the-art abstractions necessary to handle full languages and achieve scalability for large data space (as done in [27]). Indeed, our prototype employs a polyhedron domain to relate all the variables of all the threads, without any form of packing. The extension to realistic languages and experimentation on larger programs is thus left as future work. Finally, the control partitioning we currently use relies on user annotations (Sec. 3.5), although it could easily be automated (e.g., using heuristics to detect which program parts are likely to be critical sections).

Outline. The rest of the article is organized as follows: Sec. 2 introduces a simple numeric multi-threaded language and its thread-modular concrete semantics; Sec. 3 presents our abstract, computable semantics, parameterized by a classic numeric abstraction; Sec. 4 presents our prototype implementation and our experimental results; Sec. 5 discusses related work and Sec. 6 concludes.

2 Concrete semantics of the analysis

2.1 Programs

We focus on a simple language presented in Fig. 2. A program is a set of threads, defined over a set of global variables. There are no local variables in our language: they are transformed into global variables. The construction $[k_1, k_2]$ is the syntax for a random number chosen between k_1 and k_2. It expresses non-determinism, useful to model, for instance, program inputs.

$$
\begin{aligned}
\langle \text{arithmetic expressions} \rangle &::= \mathrm{k} \in \mathbb{Z} \mid [k_1,\, k_2] \mid \mathrm{X} \in \mathcal{V} \mid a_1 \dagger a_2,\ \dagger \in \{+, -, \times, /, \%\} \\
\langle \text{boolean expressions} \rangle &::= b_1 \bullet b_2 \mid \text{not } b_1 \mid a_1 \boxdot a_2 \\
&\quad \bullet \in \{\vee, \wedge\},\ \boxdot \in \{<, >, \leq, \geq, =, \neq\} \\
\langle \text{threads} \rangle &::= c_1 \ ; \ c_2 \mid {}^{l_1}\text{ if } b \text{ then } {}^{l_2}c_1 \text{ else } {}^{l_3}c_2 \text{ fi } {}^{l_4} \\
&\quad \mid \text{while } {}^{l_1}b \text{ do } {}^{l_2} \text{ c od } {}^{l_3} \mid {}^{l_1}X \leftarrow e^{l_2} \\
\langle \text{program} \rangle &::= \text{thread 1} \parallel \text{thread 2} \parallel \ldots \parallel \text{thread n}
\end{aligned}
$$

Fig. 2: Simple language to analyze.

We assume we have a fixed number of threads $\mathcal{T}$: in particular, there is no dynamic creation of threads. $\mathcal{L}$ is the set of program points, and $\mathcal{V}$ is the set of variables. A control state associates a current control point to each thread. It is defined as $\mathcal{C} = \mathcal{T} \to \mathcal{L}$. A memory state maps each variable to a value; the domain of memory states is $\mathcal{M} = \mathcal{V} \to \mathbb{Z}$ and the domain of program states is $\mathcal{S} = \mathcal{C} \times \mathcal{M}$. An interference is created by a thread when it assigns a value to a variable. It can be seen as a transition between two program states. The interference domain is denoted as $\mathcal{I} = \mathcal{T} \times (\mathcal{S} \times \mathcal{S})$. An interference $(t, (c_1, \rho_1), (c_2, \rho_2))$ means that, when the memory state is $\rho_1 \in \mathcal{M}$ and the control points are defined on every thread using $c_1 \in \mathcal{C}$, the execution of thread t changes the memory state into ρ_2, and the control points are now defined using c_2. Moreover, an interference generated by a threat t changes only the control point of t, so: $\forall t' \in \mathcal{T} \setminus \{t\}, c_1(t') = c_2(t')$.

2.2 Thread-modular concrete semantics

We present in Fig. 3 a concrete semantics of programs with interference, using an accumulating semantics. By accumulating semantics, we mean that $(R, I) \subseteq \mathbb{S}[\![stat]\!]_t(R, I)$: the analysis adds new reachable states to R and interference to I, keeping already accumulated states and interference intact. It is important to gather the set of all interference accumulated during all possible executions of each thread, to inject into the analysis of other threads, and we also accumulate states to keep a consistent semantic flavor. Such a semantics is also the natural result of specializing the thread-modular reachability semantics of [27] to the transition systems generated by our language (left implicit here for space reasons). $\mathbb{E}[\![expr]\!]\rho$ is the usual evaluation of an arithmetic or a boolean expression $expr$ given a memory state ρ. Its signature is $\mathbb{E}[\![expr]\!] : \mathcal{M} \to \mathcal{P}(\mathbb{Z})$, due to the non-determinism of $[k_1, k_2]$. Intuitively, the analysis is the iteration of two steps:

1. For each thread t, analyze t and take into account any number of "valid" interference created by other threads.
2. For each thread t, collect the interference created by this thread t.

In Eq. (1), we express the fact that $\mathbb{S}[\![stat]\!]_t$ needs a statement $stat$ of a thread t, a global program state, and a set of interference, in order to compute a resulting program state and a new set of interference. The interference given in input is used to compute the effect of the other threads on thread t. The set

$$\mathbb{S}[\![stat]\!]_t, \mathbb{B}[\![bexpr]\!]_t : \mathcal{P}(\mathcal{S}) \times \mathcal{P}(\mathcal{I}) \to \mathcal{P}(\mathcal{S}) \times \mathcal{P}(\mathcal{I}) \tag{1}$$

$$\mathbb{S}[\![^{l_1} X \leftarrow e^{l_2}]\!]_t(R, I) = \tag{2}$$
$$\quad \text{let } I_1 = \{(t, (c, \rho), (c[t \mapsto l_2], \rho[X \mapsto v])) \mid (c, \rho) \in R, v \in \mathbb{E}[\![e]\!]\rho, c(t) = l_1\} \text{ in}$$
$$\quad \text{let } R_1 = \{(c', \rho') \mid \exists (c, \rho), (t, (c, \rho), (c', \rho')) \in I_1\} \text{ in}$$
$$\quad \text{let } R_2 = lfp\, \lambda S.\, itf(S, t, I, R_1) \text{ in } R \cup R_2, I \cup I_1$$

$$\mathbb{B}[\![^{l_1} b^{l_2}]\!]_t(R, I) = \tag{3}$$
$$\quad \text{let } I_1 = \{(t, (c, \rho), (c[t \mapsto l_2], \rho) \mid (c, \rho) \in R, true \in \mathbb{E}[\![e]\!]\rho, c(t) = l_1\} \text{ in}$$
$$\quad \text{let } R_1 = \{(c', \rho') \mid \exists (c, \rho), (t, (c, \rho), (c', \rho')) \in I_1\} \text{ in}$$
$$\quad \text{let } R_2 = lfp\, \lambda S.\, itf(S, t, I, R_1) \text{ in } R \cup R_2, I \cup I_1$$

$$itf : (S, t, I, R) \mapsto R \cup \{(c', \rho') \mid \exists t' \in \mathcal{T} \setminus \{t\}, (c, \rho) \in S, (t', (c, \rho), (c', \rho')) \in I\}$$

$$\mathbb{S}[\![\text{stat}_1 \text{ ; } \text{stat}_2]\!]_t = \mathbb{S}[\![\text{stat}_2]\!]_t \circ \mathbb{S}[\![\text{stat}_1]\!]_t$$

$$\mathbb{S}[\![^{l_1}\text{if } b \text{ then } ^{l_2} tt \text{ else } ^{l_3} ff \text{ fi } ^{l_4}]\!]_t X = \tag{4}$$
$$\quad \text{let } T = \mathbb{S}[\![^{l_2} tt^{l_4}]\!]_t(\mathbb{B}[\![^{l_1} b^{l_2}]\!]_t X) \text{ in}$$
$$\quad \text{let } F = \mathbb{S}[\![^{l_3} ff^{l_4}]\!]_t(\mathbb{B}[\![^{l_1} \neg b^{l_3}]\!]_t X) \text{ in}$$
$$\quad X \dot\cup T \dot\cup F \qquad \text{where } \dot\cup \text{ is the element-wise union on pairs}$$

$$\mathbb{S}[\![\text{while } ^{l_1} b \text{ do } ^{l_2} c \text{ od}^{l_3}]\!]_t X = \tag{5}$$
$$\quad X \dot\cup \mathbb{B}[\![^{l_1} \neg b^{l_3}]\!]_t(lfp\, \lambda Y.(X \dot\cup \mathbb{S}[\![^{l_2} c^{l_1}]\!]_t(\mathbb{B}[\![^{l_1} b^{l_2}]\!]_t Y)))$$

(a) Thread-modular concrete semantics.

$$f : \begin{cases} \mathcal{P}(\mathcal{S}) \times \mathcal{P}(\mathcal{I}) & \longrightarrow \mathcal{P}(\mathcal{S}) \times \mathcal{P}(\mathcal{I}) \\ (R, I) & \longmapsto \dot\cup_{t \in \mathcal{T}} \mathbb{S}[\![\text{stats}_t]\!]_t(R \cup S_0, I) \end{cases}$$

(b) Definition of the thread-modular analysis operator f.

Fig. 3: Definition of the concrete analysis.

of all interference, including those collected during this analysis, is given in the output. Similarly, we can filter a domain with a boolean expression using $\mathbb{B}[\![b]\!]_t$. In that case, the new interference only reflects the change in control point.

We now detail the concrete semantics, presented in Fig. 3a. In Eq. (2), I_1 is the interference created by the new assignment $X \leftarrow e$: it is a transition from the state before the assignment to the state after the assignment. R_1 is the set of program states before any interference is applied (only the assignment is applied). In R_2, the function *itf* applies one interference to the set of states S, i.e., a transition that can be performed by another thread $t' \neq t$ according to the interference set I. As an arbitrary number of transitions from other threads can be executed between two transitions of the current thread, we actually apply the reflexive transitive closure of this interference relation, which can be expressed

as the least fixpoint (*lfp*) of *itf*. This fixpoint is computed on $(\mathcal{P}(\mathcal{S}), \subseteq)$; it exists as $\lambda S.\, itf(S, t, I, R)$ is monotonic. We note that the computation of I_1 is very similar to the transfer function for the assignment in the usual sequential case ($\mathbb{S}[\![^{l_1} X \leftarrow e^{l_2}]\!](R) = \{(c[t \mapsto l_2], \rho[X \mapsto v]) \mid (c, \rho) \in R, v \in \mathbb{E}[\![e]\!]\rho\}$); our thread-modular semantics applies such an operation, followed by interference-related computations. We will use this remark in the next section to design an abstract thread-modular semantics on top of well-known abstract operators for sequential programs. This will allow us to reuse existing abstract domains and know-how in our concurrent analysis. Equation (3) is quite similar: we create a new instance of interference, that can change the control points from l_1 to l_2 if b is satisfied. This way, R_1 is the set of program states obtained just after the boolean filtering has been applied. Then, we collect the reachable states found when applying interference caused by the other threads. The computation of R_2 is the same in Eqs. (2) and (3), and relies on the fixpoint computation of *itf*.

The rules for the other statements (sequences, conditionals, loops) are similar to the usual semantics for sequential programs. These rules can be reduced to the rules of assignment and boolean filtering by induction over the syntax.

Let $stats_t$ be the statement body of thread t, and S_0 be the set of initial program states. The thread-modular concrete semantics is defined as computing *lfp* f, where f is defined in Fig. 3b. This fixpoint is computed over the lifting of powersets; it exists as f is increasing. When f is called, it analyzes each thread once and returns two sets: the accumulated program states and the accumulated interference. During the first few iterations, some pieces of interference are not discovered yet, so the analysis may not be sound yet. When the set of interference is stable, however, the analysis is sound: all execution cases are taken into account. This is why we compute a *fixpoint* of f, and not only the first iterations.

The soundness and completeness (for state properties) of this semantics was proved in [26,27] on an arbitrary transition system. This semantics is naturally too concrete to be computable. We will abstract it in the next section.

3 Abstract semantics

Compared to the semantics of a sequential program, the semantics of the previous section embeds a rich and precise control information due to the possible thread preemption, which makes it difficult to analyze efficiently. In order to be able to reuse classic state abstraction techniques, we first reduce the complexity of the control information by abstraction in Sec. 3.1. Then, we simplify our analysis by abstracting the memory states and the interference into numeric domains.

3.1 Abstractions of states and interference

We suppose we are given a partition of control locations $\mathcal{L}$ into a set $\mathcal{L}^\#$ of abstract control locations, through some abstraction $\alpha_{\mathcal{L}} : \mathcal{L} \to \mathcal{L}^\#$, which is extended pointwise to control states as $\dot{\alpha_{\mathcal{L}}} : (\mathcal{T} \to \mathcal{L}) \to (\mathcal{T} \to \mathcal{L}^\#)$. In our

$$\alpha_{\mathcal{M}}(t) : \begin{cases} \mathcal{P}(\mathcal{S}) \longrightarrow \mathcal{L} \to \mathcal{P}(\mathcal{M}) \\ X \longmapsto \lambda L.\{e \mid (c,e) \in X \ \wedge c(t) = L\} \end{cases} \tag{6}$$

$$\alpha_{\mathcal{C}} : \begin{cases} \mathcal{P}(\mathcal{I}) \longrightarrow \mathcal{T} \to \mathcal{P}(((\mathcal{T} \to \mathcal{L}^{\#}) \times \mathcal{M})^2) \\ X \longmapsto \lambda t.\{((\dot{\alpha_{\mathcal{L}}}(c_b), b), (\dot{\alpha_{\mathcal{L}}}(c_e), e)) \mid (c_b, c_e) \in \mathcal{C}^2, (t, (c_b, b), (c_e, e)) \in X\} \end{cases} \tag{7}$$

$$\gamma_{\mathcal{D}^{\#}} : (\mathcal{L} \to \mathcal{D}^{\#}) \longrightarrow (\mathcal{L} \to \mathcal{P}(\mathcal{M})) \tag{8}$$

$$\gamma_{\mathcal{I}^{\#}} : (\mathcal{T} \to \mathcal{I}^{\#}) \longrightarrow (\mathcal{T} \to \mathcal{P}(((\mathcal{T} \to \mathcal{L}^{\#}) \times \mathcal{M})^2) \tag{9}$$

Fig. 4: Abstractions and concretizations of states and interference.

implementation, control locations are manually specified using a program annotation, and the automatic determination of a good partitioning is left as future work. This function is used to abstract interference information, as shown by the function $\alpha_{\mathcal{C}}$ in Eq. (7). In contrast, for program states, our control abstraction $\alpha_{\mathcal{M}}(t)$ in Eq. (6) keeps the control information $c(t) \in \mathcal{L}$ intact for the current thread and abstracts away the control information $c(t')$ of other threads $t' \neq t$, hence the shift from control in $\mathcal{C} = \mathcal{T} \to \mathcal{L}$ to control in $\mathcal{L}$, which makes the analysis of a thread rather similar to a flow-sensitive sequential analysis. The abstract semantics presented in the next section will have its precision and computational cost strongly dependent upon the choice of $\mathcal{L}^{\#}$.

3.2 Thread-modular abstract semantics

We assume we are given an arbitrary numeric domain $\mathcal{D}^{\#}$ to abstract the contents of the program states, and denote by $\gamma_{\mathcal{D}^{\#}}$ the associated concretization function. Likewise, we assume that an arbitrary numeric domain $\mathcal{I}^{\#}$ is provided to abstract interference, and denote by $\gamma_{\mathcal{I}^{\#}}$ its concretization. The signatures of these concretizations are given in Eqs. (8) and (9). Our concrete states feature control point information as well as numeric variables. This is translated in the abstract domains $\mathcal{D}^{\#}, \mathcal{I}^{\#}$ by two types of variables: original numeric variables, and auxiliary control point variables, called `aux_t` for $t \in \mathcal{T}$. We thus consider here that numbers are used as control points $\mathcal{L}$ in order to stay within the realm of numeric domains. Moreover, interference abstraction is partitioned by thread.

As the sets of interference are abstracted into a relational numeric domain, we can represent a transition using an initial, non-primed variable, and a final, primed variable. For example, to express that x can change from 1 to 2, we write $x = 1 \wedge x' = 2$. Similarly, an interference increasing a variable x, can be written as $x' \geq x+1$. We show how we can use existing numeric abstract domains used in the analysis of sequential programs to abstract our concrete thread-modular analysis. We assume given abstract operations, such as simultaneously assigning a set of arithmetic expressions $\mathcal{A}$ to a set of variables, adding uninitialized variables, renaming and deleting variables. The signature of these functions is presented in Fig. 5. We also suppose we have a join and a meet (abstracting respectively

$$assign : \mathcal{X}^\# \times \mathcal{P}(\mathcal{V} \times \mathcal{A}) \to \mathcal{X}^\# \qquad add : \mathcal{X}^\# \times \mathcal{P}(\mathcal{V}) \to \mathcal{X}^\#$$
$$rename : \mathcal{X}^\# \times \mathcal{P}(\mathcal{V}^2) \to \mathcal{X}^\# \qquad delete : \mathcal{X}^\# \times \mathcal{P}(\mathcal{V}) \to \mathcal{X}^\#$$

Fig. 5: Signatures of usual abstract operators with $\mathcal{X}^\# \in \{\mathcal{D}^\#, \mathcal{I}^\#\}$.

$$extend : \begin{cases} \mathcal{D}^\# \longrightarrow \mathcal{I}^\# \\ R^\# \longmapsto add(R^\#, \{x' \mid x \in \mathrm{Var}(R^\#)\}) \end{cases}$$
$$img : \begin{cases} \mathcal{I}^\# \longrightarrow \mathcal{D}^\# \\ I^\# \longmapsto \text{let } X = \{x \in \mathrm{Var}(I^\#) \mid x' \in \mathrm{Var}(I^\#)\} \text{ in} \\ \qquad \text{let } R_1^\# = delete(I^\#, X) \text{ in} \\ \qquad rename(R_2^\#, \{(x', x) \mid x \in X\}) \end{cases}$$
$$apply : \begin{cases} \mathcal{D}^\# \times \mathcal{I}^\# \longrightarrow \mathcal{D}^\# \\ R^\#, I^\# \longmapsto \text{let } R_1^\# = extend(R^\#) \text{ in} \\ \qquad \text{let } R_2^\# = R_1^\# \cap^\# I^\# \text{ in} \\ \qquad img(R_2^\#) \end{cases}$$

Fig. 6: Definition of *extend*, *img* and *apply*.

the union and the intersection), and a widening operator, respectively denoted $\cup^\#$, $\cap^\#$, and $\triangledown$. These are standard operations, implemented in abstract domain libraries, such as Apron [19] and BddApron [17].

We now define a function *apply*, applying an instance of interference to a numeric domain. In a sense, *apply* gives the image of an abstract domain under an abstract interference relation. We first implement two auxiliary functions, called *extend* and *img*, defined in Fig. 6. We introduce a new function Var associating to each abstract domain its variables. The function *extend* creates a copy of every variable in the abstract domain, thus creating an interference relation. On the other hand, *img* returns the image set of an interference relation. With these two functions, we can now give a procedure computing the result of applying a piece of interference, given an initial abstract memory domain: we first add primed copies of the variables of the abstract memory domain. We can then intersect the resulting abstract memory domain $R_1^\#$ with the interference. Then, we have to get the image of the relation, which is the part where the variables are primed. The obtained abstract domain is restricted to the states reachable after an instance of interference is applied to the abstract initial domain.

The abstract semantics is presented in Fig. 7a. We abstract the concrete semantics of Eq. (2) and Eq. (3) in Eq. (11) and Eq. (12). The transition from the concrete to the abstract semantics is straightforward, by composition of *apply*, *assign*, *extend*, and *img*. We only briefly comment on the definition of Eq. (11), as Eq. (12) is similar. $I_{l''}^\#$ represents any interference starting from the abstract state $R^\#(l_1)$ (by definition of *extend*). Then, we constrain this interference so

$$\mathbb{S}^\#[\![stat]\!]_t : (\mathcal{L} \to \mathcal{D}^\#) \times (\mathcal{T} \to \mathcal{I}^\#) \longrightarrow (\mathcal{L} \to \mathcal{D}^\#) \times (\mathcal{T} \to \mathcal{I}^\#) \tag{10}$$

$$\mathbb{S}^\#[\![^{l_1} X \leftarrow e^{l_2}]\!]_t(R^\#, I^\#) = \tag{11}$$
$$\quad \text{let } I^\#_{l''} = extend(R^\#(l_1)) \text{ in}$$
$$\quad \text{let } I^\#_{l'} = assign(I^\#_{l''}, \{(X', e)\} \cup \{(Y', Y) \mid Y \in Var(R^\#) \setminus \{X, \texttt{aux_t}\}\}) \text{ in}$$
$$\quad \text{let } I^\#_{l} = assign(I^\#_{l'}, \{(\texttt{aux_t}, \alpha_{\mathcal{L}}(l_1)), (\texttt{aux_t'}, \alpha_{\mathcal{L}}(l_2))\}) \text{ in}$$
$$\quad \text{let } R^\#_1 = img(I^\#_l) \text{ in}$$
$$\quad \text{let } R^\#_2 = \lim \lambda Y^\#.\ Y^\# \triangledown itf^\#(Y^\#, t, I^\#, R^\#_1) \text{ in}$$
$$\quad R^\#[l_2 \mapsto R^\#(l_2) \cup^\# R^\#_2], I^\#[t \mapsto I^\#(t) \cup^\# I^\#_l]$$

$$\mathbb{B}^\#[\![^{l_1} b^{l_2}]\!]_t(R^\#, I^\#) = \tag{12}$$
$$\quad \text{let } I^\#_{l''} = extend(\mathbb{F}^\#[\![b]\!](R^\#(l_1))) \text{ in}$$
$$\quad \text{let } I^\#_{l'} = assign(I^\#_{l''}, \{(Y', Y) \mid Y \in Var(R^\#) \setminus \{\texttt{aux_t}\}) \text{ in}$$
$$\quad \text{let } I^\#_{l} = assign(I^\#_{l'}, \{(\texttt{aux_t}, \alpha_{\mathcal{L}}(l_1)), (\texttt{aux_t'}, \alpha_{\mathcal{L}}(l_2))\}) \text{ in}$$
$$\quad \text{let } R^\#_1 = img(I^\#_l) \text{ in}$$
$$\quad \text{let } R^\#_2 = \lim \lambda Y^\#.\ Y^\# \triangledown itf^\#(Y^\#, t, I^\#, R^\#_1) \text{ in}$$
$$\quad R^\#[l_2 \mapsto R^\#(l_2) \cup^\# R^\#_2], I^\#[t \mapsto I^\#(t) \cup^\# I^\#_l]$$

$$itf^\# : (S^\#, t, I^\#, R^\#) \mapsto R^\# \cup^\# \bigcup^\#_{t' \in \mathcal{T} \setminus \{t\}} apply(S^\#, I^\#(t'))$$

$$\mathbb{S}^\#[\![^{l_1} \text{if } b \text{ then } ^{l_2} tt \text{ else } ^{l_3} ff \text{ fi } ^{l_4}]\!]_t X^\# =$$
$$\quad \text{let } T = \mathbb{S}^\#[\![^{l_2} tt^{l_4}]\!]_t \circ \mathbb{B}^\#[\![^{l_1} b^{l_2}]\!]_t X^\# \text{ in}$$
$$\quad \text{let } F = \mathbb{S}^\#[\![^{l_3} ff^{l_4}]\!]_t \circ \mathbb{B}^\#[\![^{l_1} \neg b^{l_3}]\!]_t X^\# \text{ in}$$
$$\quad X^\# \dot\cup^\# T \dot\cup^\# F$$

$$\mathbb{S}^\#[\![\text{while } ^{l_1} b \text{ do } ^{l_2} c \text{ od}^{l_3}]\!]_t X =$$
$$\quad X^\# \dot\cup^\# \mathbb{B}^\#[\![^{l_1} \neg b^{l_3}]\!]_t(\lim \lambda Y. Y \ \dot\triangledown\ (X^\# \ \dot\cup^\# \ \mathbb{S}^\#[\![^{l_2} c^{l_1}]\!]_t \circ \mathbb{B}^\#[\![^{l_1} b^{l_2}]\!]_t Y))$$

(a) Definition of the abstract semantics.

$$f^\# : \begin{cases} (\mathcal{T} \to (\mathcal{L} \to \mathcal{D}^\#)) \times (\mathcal{T} \to \mathcal{I}^\#) \longrightarrow (\mathcal{T} \to (\mathcal{L} \to \mathcal{D}^\#)) \times (\mathcal{T} \to \mathcal{I}^\#) \\ (R^\#, I^\#) \qquad\qquad \longmapsto \lambda t. R'^\#_t, \dot\cup^\#_{t \in \mathcal{T}} I'^\#_t \\ \qquad\qquad\qquad \text{with } R'^\#_t, I'^\#_t = \mathbb{S}^\#[\![\text{stats}_t]\!]_t(R^\#(t) \cup^\# S^\#_0,\ I^\#) \end{cases}$$

(b) Definition of the analysis operator $f^\#$.

$$\gamma : \begin{cases} (\mathcal{T} \to (\mathcal{L} \to \mathcal{D}^\#)) \times (\mathcal{T} \to \mathcal{I}^\#) \longrightarrow \mathcal{P}(\mathcal{S}) \times \mathcal{P}(\mathcal{I}) \\ (r, i) \qquad\qquad\qquad\quad \longmapsto \bigcup_{t \in \mathcal{T}}\ \gamma_{\mathcal{M}}(t) \circ \gamma_{\mathcal{D}^\#}(r(t)), \gamma_{\mathcal{C}} \circ \gamma_{\mathcal{I}^\#}(i) \end{cases}$$

(c) Definition of the main concretization.

Fig. 7: Definitions of our abstract analysis.

that $I_l^\#$ represents interference from $R^\#(l_1)$ to the state of the program just after the assignment is done. $I_l^\#$ is by construction the abstract version of I_1 from Eq. (2). Recall that `aux_t` are auxiliary variables representing control locations, and that $\alpha_{\mathcal{L}}$ returns a numeric abstract control point. By construction of *img*, $R_1^\#$ is the abstract version of R_1, representing the image of the interference $I_l^\#$. $R_2^\#$ represents all reachable states after any possible interference has been applied to $R_1^\#$. This concludes the case of assignments. The if and while statements are abstracted in a classic way, by induction on the syntax. As usual [7], we changed the *lfp* operators into limits of an iteration with widening, so that convergence is ensured. $\mathbb{F}^\#[\![b]\!]$ is the usual abstract boolean filtering operator.

The abstract version $f^\#$ of the analysis operator f from Fig. 3b is defined in Fig. 7b. $S_0^\#$ represents the abstracted initial program states, abstracting S_0.

3.3 Soundness of the analysis

We first define a concretization operator going from our abstract states to our concrete ones, before stating the soundness result. Fig. 7c presents the global concretization, where $\gamma_{\mathcal{M}}(t)$ and $\gamma_{\mathcal{C}}$ are the adjoints associated to the abstractions $\alpha_{\mathcal{M}}(t)$ and $\alpha_{\mathcal{C}}$ presented in Fig. 4. The analysis presented is sound, i.e.:

$$lfp\ f \subseteq \gamma(\lim \lambda Y.Y\ \dot{\nabla}\ f^\#(Y))$$

3.4 Retrieving flow-insensitive interference analysis

The analysis presented in Sec 3.2 can by very costly, depending on the choice of $\mathcal{L}^\#$. For many analyses, choosing $\mathcal{L}^\#$ to be a singleton is sufficient to be precise enough and having an easily computable analysis. In that case, the set of interference becomes flow-insensitive, and the analysis is an extension of existing analyses. For example, the thread-modular analysis presented in [28] is non-relational and can be retrieved by choosing a non-relational domain for $\mathcal{I}^\#$ and $\mathcal{L}^\#$ to be a singleton. On the contrary, when choosing $\mathcal{L}^\#$ to be $\mathcal{L}$, the analysis would be roughly equivalent to analyzing the product of the control flow graphs (the main difference being that the control information would be stored in auxiliary variables, and possibly subject to numeric abstraction).

3.5 On the way to proving mutual exclusion: growing $\mathcal{L}^\#$

When $\mathcal{L}^\#$ is a singleton, the interference set is too coarse, and some properties are impossible to prove. For example, to verify mutual exclusion properties, we need to use a separate abstract control point at the beginning of the critical section to partition the abstract state. This is what we call control partitioning. In order to partition the abstract state, we need a richer $\mathcal{L}^\#$.

Let us consider Peterson's mutual exclusion algorithm as described in ConcurInterproc [18], and presented in Fig. 8. We suppose that `b0`, `b1`, and `turn` are boolean variables, and that at the beginning, $\neg$`b0` $\wedge$ $\neg$`b1` holds. If there is

	Thread 1	Thread 2
1	b1 ← true	b2 ← true
2	turn ← false	turn ← true
3	**while**(b2 ∧ ¬turn) **do skip od**	**while**(b1 ∧ turn) **do skip od**
4	**skip**	**skip**
5	b1 ← false	b2 ← false

Fig. 8: Peterson's mutual exclusion algorithm.

no control point separation between lines 1-3 and 4-5, then, the following execution order is possible: we first execute lines 1-2 of each thread. We suppose that then, `turn` is true (the other case is symmetric). At that point, the condition in the while loop of Thread 1 is satisfied, and, in Thread 2, the interference $\texttt{b1}' = \neg\texttt{b1}$ (created at line 5 in Thread 1) can be applied, enabling the two threads to access simultaneously the mutual exclusion section, here embodied by the skip statements. If a label separation is created, instead, this spurious execution is not possible anymore, and mutual exclusion is actually inferred. Indeed, let us set $\mathcal{L}^{\#} = \{[1,3];[4,5]\}$ for both threads. This time, when the variable `turn` holds, the control state is $(1,2) \mapsto (3,4)$, and we cannot apply the interference $\texttt{b1}' = \neg\texttt{b1}$ of Thread 1: we are at label 3, and $3 \notin [4,5]$. In all the cases we observed in practice (discussed in the following section), splitting the control locations at the beginning of the critical section provides a sufficient gain in precision to infer mutual exclusion.

4 Implementation and experimental results

4.1 Implementation

We implemented an analyzer prototype, called Batman, in order to assess the precision and scalability of our analysis. It consists of roughly 1700 lines of OCaml code, and can use either the Apron [19] or BddApron [17] libraries to manipulate abstract domains. We implemented a simple widening with thresholds, as well as increasing and decreasing iterations. The analyzer uses functors, so that switching from one relational domain to another is easy. In order to show the benefit of thread-modular analyses, we compare our results with those obtained by ConcurInterproc [18], another academic analyzer for numeric properties of multi-threaded programs, which is relational but not thread-modular. We use a similar type of language: it supports a fixed number of integer and boolean variables, if and while statements, assignments, and a fixed number of threads. Our analyzer does not support procedures, unlike ConcurInterproc.

4.2 Precision of the analysis

Batman is able to automatically infer the relational invariants described previously. We present the results we obtained on some examples.

Relational analysis. Using our fully relational analysis, we are able to prove more properties on the example provided in Fig. 1 than what was presented in [27], because the assignment $x \leftarrow (x + y)/2$ is keeping the invariant $x \leq y$. This cannot be expressed using the analysis provided in [27]. Moreover, invariants are simpler to express using the polyhedron-based interference: $x + 1 \leq x'$ means that when this interference is applied, it increases x. We experimented on several simple examples proposed in recent work [27]. The results are presented in Table 9b. The flow-sensitivity column describes whether the abstraction of interference was flow-sensitive (i.e., $\mathcal{L}^\#$ is not a singleton) or flow-insensitive.

Mutual exclusion algorithms. We are also able to analyze classic mutual exclusion algorithms such as Peterson's algorithm [29], presented in Fig. 8, and Lamport's Bakery algorithm [23]. To infer the mutual exclusion property automatically, we need to give the analyzer a partition of the control points, and we use a simple annotation system for this. Splitting the control locations at the beginning of the critical section was sufficient for all our tests. We also need the interference abstraction to be relational to infer mutual exclusion. Partitioning heuristics could be developed to improve the automation of the analysis. These heuristics were not in the prototype used for our experiments due to lack of time, and their development is left for future work.

4.3 Scalability of the analysis

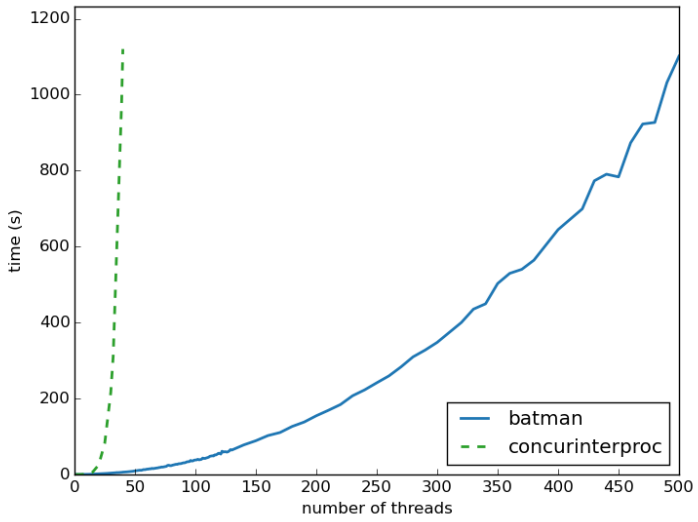
We also studied the scalability of the analysis as a function of the number of threads. We thus considered algorithms able to synchronize an arbitrary number of threads. Note that the polyhedral domain does not scale up with the number of variables; hence, there is no hope to scale up in the number of threads for algorithms that require additional variables for each thread. We performed some experiments with both the polyhedron and the more scalable octagon domain.

Ideal case. In order to study the scalability limit of the thread-modular analysis independently from the scalability of the numeric domain, we can consider the ideal case of programs where the number of global variables is fixed, whatever the number of threads. As example, we present, in Fig. 9a, a simple mutual exclusion algorithm based on token passing, with only two variables, and show that our method scales up to hundreds of threads, while ConcurInterproc does not. In this example, we used the polyhedral domain. As the numeric domain is a parameter of our analysis, we can consider the scalability in the number of variables to be an orthogonal problem to that of the scalability in the number of threads, and this article addresses the later rather than the former.

Lamport's Bakery algorithm. We tested the scalability of Lamport's Bakery algorithm, ensuring mutual exclusion for an arbitrary number n of threads. However, the number of variables for the global program is linear in n, and the size of the whole program is quadratic in n (this is a consequence of our encoding of arrays into scalar variables). This setting does not really promise to be scalable,

Thread 1	Thread 2	Thread 3

while true **do**
  while f != 1 **do**
  skip
  od
  $X \leftarrow 1$
  $f \leftarrow [1, 3]$
od
	while true **do**
	  while f != 2 **do**
	  skip
	  od
	  $X \leftarrow 2$
	  $f \leftarrow [1, 3]$
	od
		while true **do**
		  while f != 3 **do**
		  skip
		  od
		  $X \leftarrow 3$
		  $f \leftarrow [1, 3]$
		od



(a) Analysis of a token-passing mutual exclusion algorithm.

Reference (in [27])	Flow-sensitivity	Results	Time, polyhedron
Fig. 1	✗	$0 \leq X \leq Y$	0.30s
Fig. 4	✗	$0 \leq T \leq L \leq C \leq H \leq 10^4$	0.26s
Fig. 5 (a)	✗	$0 \leq X$	0.44s
Fig. 5 (a)	✓	$0 \leq X \leq 100$	0.35s
Fig. 5 (b)	✗	thread 1: $X \leq Y \leq 100$ thread 2: $0 \leq Y \leq X$	0.78s
Fig. 5 (b)	✓	$0 \leq X = Y \leq 100$	0.44s

Algorithm name	Number of threads	Flow-sensitivity	Mutual exclusion	Time, polyhedron	Time, octagons
Peterson	2	✓	✓	0.67s	0.72s
Lamport	3	✓	✓	6.5s	27s
Lamport	4	✓	✓	49s	6m 33s
Lamport	5	✓	✓	5m 10s	49m 45s
Lamport	6	✓	✓	–	151m 8s
Lamport	7	✓	✓	–	12h

(b) Analysis result.

Fig. 9: Experimental evaluation of our approach.

but we are still able to analyze up to 7 threads, and the mutual exclusion is inferred each time. As mentioned above, to infer the mutual exclusion property within the critical section, both the flow-sensitive and relational properties of the interference are required. For each thread, we have two different elements of $\mathcal{L}^{\#}$, one before the critical section and one after. This is sufficient to infer the mutual exclusion property. The results are presented in Table 9b. ConcurInterproc seems to be unable to infer the mutual exclusion, and is less scalable here: it takes 90 seconds to analyze 3 threads, and 22 hours to analyze 4 threads.

5 Related Work

Many articles have been devoted to the analysis of multi-threaded programs. We cite only the most relevant. Our article can be seen as an extension of the first thread-modular abstract interpreters [5,25] that were non-relational and flow-insensitive. We achieve higher levels of precision, when parameterized with relational domains. We build on a theoretical framework for complete concrete thread-modular semantics [27]. However, while [27] then explores the end of the spectrum concerned with scalable but not very precise analyses (using relationality only in a few selected points), we explore the other end in order to prove properties of small but intricate programs not precisely analyzed by [27]. We also study the scalability of fully-relational analyses for large numbers of threads. ConcurInterproc [18] is a static analyzer for concurrent programs. It is not thread-modular and, as shown in our benchmarks, does not scale as well as our approach, even though both use the same polyhedral numeric abstraction. Kusano and Wang [21] extend the flow-insensitive abstract interpretation of [5,25] by maintaining flow-sensitive information about interference using constraints, while not maintaining numeric information on them; hence, their method is complementary to ours. Farzan and Kincaid [12] also model interference in a thread-modular abstract interpreter using constraints, but focus instead on parameterized programs with an unbounded number of thread instances.

Modular verification techniques for concurrent programs also include flow analyses. Dwyer [11] proposed a flow method to check properties expressed with finite automata. Grunwald and Srinivasan [15] consider the reaching definition problem for explicitly parallel programs. These works focus on monotone dataflow equations, which is not the general case for abstract interpretation.

Model-checking of concurrent programs is a very well developed field. To prevent the state explosion problem in the specific case of concurrent programs, partial order reduction methods have been introduced by Godefroid [13]. Our method differs in that we do not have to consider explicit interleavings, and that we employ abstractions, allowing a loss of precision to achieve a faster analysis. A more recent method to reduce the cost of model-checking consists in only analyzing a program up to a fixed number of interleavings [30]. Our approach differs as we retain the soundness of the analysis. Counter-example guided abstract refinement methods have also been adapted to concurrent programs; one such example is [10]. As in the case of analyzing sequential programs, these

methods are based on a sequence of analyses with increasingly more expressive finite abstract domains of predicates, which may not terminate. By contrast, our method is based on abstract interpretation, and so iterates in (possibly infinite) abstract domains employing widenings to ensure termination. Thread-modular model-checking has also been advocated, as in [6], which helps with the scalability issue. However, the method uses BDDs, and is thus limited to finite data-spaces. By contrast, we employ abstract interpretation techniques in order to use infinite-state abstract domains (such as polyhedra). [14] proposes a general framework to express and synthesize a large variety of static analyses from sets of rules, based on Horn clauses, which includes rely-guarantee analyses for multi-threaded programs; while [16] proposes a related approach based on constraints. Following predicate abstraction and CEGAR methods, the memory abstraction in these works is often limited by the inherent Cartesian abstraction [24], which cannot thus infer relations between variables from different threads.

6 Conclusion

We have proposed a general analysis by abstract interpretation for numeric properties of concurrent programs that is thread-modular, takes soundly into account all the thread interleavings, and is parameterized by a choice of numeric and control abstractions. The novelty of our approach is its ability to precisely control trade-offs between cost and precision when analyzing thread interference, from a coarse, flow-insensitive and non-relational abstraction (corresponding to the state-of-the-art) to fully flow-sensitive and relational abstractions. We showed on a few simple example analyses with our prototype that relational interference allows proving properties that could only be handled by non-modular analyses before, while we also benefit from the scalability of thread-modular methods. We believe that this opens the door to the design of scalable and precise analyses of realistic concurrent programs, provided that adequate abstractions are designed.

Future work will include designing such abstractions, and in particular designing heterogeneous abstractions able to restrict the flow-sensitivity and relationality to program parts requiring more precision. We would also like to remove the current limitation that every variable is considered to be global and appears in all thread-local and interference abstractions, which limits the scalability of our analysis. It would also be interesting to have a more comprehensive experimental evaluation, as well comparisons with other approaches. Fine-grained control of which variables are taken into account in each abstraction should be possible using packing techniques or weakly relational domains [2,3]. Likewise, the control abstraction used in the interference is currently set manually, but we believe that automation is possible using heuristics (such as guessing plausible locations of critical sections). We consider integrating this method into [27] in order to gain more precision when necessary while retaining the overall scalability. Other future work includes handling weakly consistent memories, and the non-uniform analysis of unbounded numbers of threads, which requires integrating other forms of abstractions into our analysis.

References

1. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: POPL'10. pp. 7–18. ACM (Jan 2010)
2. Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Static analysis and verification of aerospace software by abstract interpretation. In: AIAA Infotech@Aerospace. pp. 1–38. No. 2010-3385, AIAA (Apr 2010)
3. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI'03. pp. 196–207. ACM (June 2003)
4. Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: FMPA'93. LNCS, vol. 735, pp. 128–141. Springer (June 1993)
5. Carré, J.L., Hymans, C.: From single-thread to multithreaded: An efficient static analysis algorithm. Tech. Rep. arXiv:0910.5833v1, EADS (Oct 2009)
6. Cohen, A., Namjoshi, K.S.: Local proofs for global safety properties. Formal Methods in System Design 34(2), 104–125 (2008)
7. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL'77. pp. 238–252. ACM (Jan 1977)
8. Cousot, P., Cousot, R.: Invariance proof methods and analysis techniques for parallel programs. In: Automatic Program Construction Techniques, chap. 12, pp. 243–271. Macmillan, New York, NY, USA (1984)
9. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL'78. pp. 84–97. ACM (1978)
10. Donaldson, A., Kaiser, A., Kroening, D., Tautschnig, M., Wahl, T.: Counterexample-guided abstraction refinement for symmetric concurrent programs. Formal Methods in System Design 41(1), 25–44 (2012)
11. Dwyer, M.B.: Modular flow analysis for concurrent software. In: ASE'97. pp. 264–273. IEEE Computer Society (1997)
12. Farzan, A., Kincaid., Z.: Duet: Static analysis for unbounded parallelism. In: CAV'13. LNCS, vol. 8044, pp. 191–196 (2013)
13. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem. Ph.D. thesis, University of Liege, Computer Science Department (1994)
14. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI'12. pp. 405–416. ACM (2012)
15. Grunwald, D., Srinivasan, H.: Data flow equations for explicitly parallel programs. In: PPOPP '93. pp. 159–168. ACM (1993)
16. Gupta, A., Popeea, C., Rybalchenko, A.: Threader: A constraint-based verifier for multi-threaded programs. In: CAV'11. LNCS, vol. 6806, pp. 412–417. Springer (2011)
17. Jeannet, B.: BddApron, `http://pop-art.inrialpes.fr/~bjeannet/bjeannet-forge/bddapron/bddapron.pdf`
18. Jeannet, B.: Relational interprocedural verification of concurrent programs. Software & Systems Modeling 12(2), 285–306 (2013)
19. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: CAV'09. LNCS, vol. 5643, pp. 661–667. Springer (2009)
20. Jones, C.B.: Development Methods for Computer Programs including a Notion of Interference. Ph.D. thesis, Oxford University (Jun 1981)

21. Kusano, M., Wang, C.: Flow-sensitive composition of thread-modular abstract interpretation. In: FSE 2016. pp. 799–809. ACM (2016)
22. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. In: IEEE Trans. on Computers. vol. 28, pp. 690–691. IEEE Comp. Soc. (Sep 1979)
23. Lamport, L.: A new solution of dijkstra's concurrent programming problem. Communications of the ACM 17(8), 453–455 (1974)
24. Malkis, A., Podelski, A., Rybalchenko, A.: ICTAC 2006, LNCS, vol. 4281, chap. Thread-Modular Verification Is Cartesian Abstract Interpretation, pp. 183–197. Springer (2006)
25. Miné, A.: Static analysis of run-time errors in embedded critical parallel C programs. In: ESOP'11. LNCS, vol. 6602, pp. 398–418. Springer (Mar 2011)
26. Miné, A.: Static analysis by abstract interpretation of sequential and multi-thread programs. In: Proc. of the 10th School of Modelling and Verifying Parallel Processes (MOVEP 2012). pp. 35–48 (3–7 Dec 2012)
27. Miné, A.: Relational thread-modular static value analysis by abstract interpretation. In: VMCAI'14. LNCS, vol. 8318, pp. 39–58. Springer (Jan 2014)
28. Miné, A.: Static analysis of run-time errors in embedded real-time parallel C programs. Logical Methods in Computer Science (LMCS) 8(26), 63 (Mar 2012)
29. Peterson, G.L.: Myths about the mutual exclusion problem. Information Processing Letters 12(3), 115–116 (1981)
30. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: TACAS'05. LNCS, vol. 3440, pp. 93–107. Springer (2005)