Abstract Interpretation with Unfoldings*

Marcelo Sousa[1], César Rodríguez[2,3], Vijay D'Silva[4] and Daniel Kroening[1,3]

[1] University of Oxford, United Kingdom
[2] Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS, France
[3] Diffblue Ltd., United Kingdom
[4] Google Inc., San Francisco

Abstract. We present and evaluate a technique for computing path-sensitive interference conditions during abstract interpretation of concurrent programs. In lieu of fixed point computation, we use prime event structures to compactly represent causal dependence and interference between sequences of transformers. Our main contribution is an unfolding algorithm that uses a new notion of independence to avoid redundant transformer application, thread-local fixed points to reduce the size of the unfolding, and a novel cutoff criterion based on subsumption to guarantee termination of the analysis. Our experiments show that the abstract unfolding produces an order of magnitude fewer false alarms than a mature abstract interpreter, while being several orders of magnitude faster than solver-based tools that have the same precision.

1 Introduction

This paper is concerned with the problem of extending an abstract interpreter for sequential programs to analyze concurrent programs. A naïve solution to this problem is a global fixed point analysis involving all threads in the program. An alternative that seeks to exploit the scalability of local analyses is to analyze each thread in isolation and exchange invariants on global variables between threads [19,18,3]. Much research on abstract interpretation of concurrent programs, including this paper, aims to discover analyses that combine the scalability of the local fixed point computation with the precision of a global fixed point.

The *abstract unfolding* data structure and algorithm presented in this paper combines an abstract domain with the type algorithm used to analyze Petri nets. An unfolding is a tree-like structure that uses partial orders to compactly represent concurrent executions and uses conflict relations to represent interference between executions. There are several obstacles to combining unfoldings with abstract domains. First, unfolding construction requires interference information that is absent from abstract domains. Second, an unfolding compactly represents the traces of a system, while abstract domains approximate states and transitions. Finally, unfolding algorithms perform explicit-state analysis of deterministic systems while abstract domains are inherently symbolic and non-deterministic owing to abstraction.

* Supported by ERC project 280053 (CPROVER) and a Google Fellowship.

```
Thread 1:                 Thread 2:
while (i++ < 99)          while (j++ < 149)
  if (random)               if (random)
    break;                    break;
g += i;                   g += j;
                          assert(g <= 250);
```

(a)

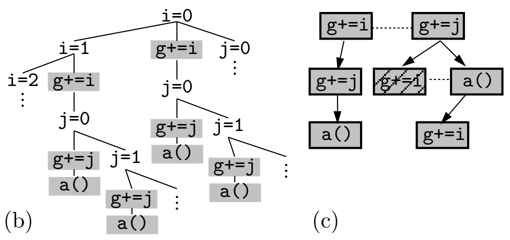


Fig. 1. (a) Example program (b) Its POR exploration tree (c) Our unfolding

The main idea of this paper is to construct an unfolding *of an analyzer*, rather than a program. An event is the application of a transformer in an analysis context, and concurrent executions are replaced by a partial order on transformer applications. We introduce independence for transformers and use this notion to construct an unfolding of a domain given a program and independence relation. The unfolding of a domain is typically large and we use thread-local fixed point computation to reduce its size without losing interference information.

Most pairs of transformers are not related by standard notions of independence. A counterintuitive observation in this paper is that by increasing the path-sensitivity of the analysis, we decrease interference, which reduces the number of interleavings to explore and improves scalability. From a static analysis perspective, our analyser uses the unfolding to represent a history abstraction (or trace partition) which is constructed using independence. From a dynamic analysis perspective, our approach is a form of partial-order reduction (POR) [23] that uses an abstract domain to collapse branches of the computation tree originating from thread-local control decisions.

Contribution We make the following contributions towards reusing an abstract interpreter for sequential code for the analysis of a concurrent program.

1. A new notion of transformer independence for unfolding with domains (Sec. 4).
2. The unfolding of a domain, which provides a sound way to combine transformer application and partial-order reduction (Sec. 5.1).
3. A method to construct the unfolding using thread-local analysis and pruning techniques (Sec. 6.1, Sec. 6).
4. An implementation and empirical evaluation demonstrating the trade-offs compared to an abstract interpreter and solver-based tools (Sec. 7).

The proofs of the formal results presented in this paper can be found in the extended version [24].

2 Motivating Example and Overview

Consider the program given in Fig. 1 (a), which we wish to prove safe using an interval analysis. Thread 1 (resp. 2) increments `i` (resp. `j`) in a loop that can non-deterministically stop at any iteration. All variables are initialized to 0 and the program is safe, as the `assert` in thread 2 cannot be violated.

When we use a POR approach to prove safety of this program, the exploration algorithm exploits the fact that only the interference between statements that modify the variable `g` can lead to distinct final states. This interference is typically known as *independence* [22,10]. The practical relevance of independence is that one can use it to define a safe fragment, given in Fig. 1 (b), of the computation tree of the program which can be efficiently explored [23,1]. At every iteration of each loop, the conditionals open one more branch in the tree. Thus, each branch contains a different write to the global variable, which is dependent with the writes of the other thread as the order of their application reaches different states. As a result, the exploration tree becomes intractable very fast. It is of course possible to bound the depth of the exploration at the expense of completeness of the analysis.

The thread-modular static analysis that is implemented in AstreeA [19] or Frama-c [26] incorrectly triggers an alarm for this program. These tools statically analyze each thread in isolation assuming that `g` equals 0. Both discover that thread 1 (resp. 2) can write $[0, 100]$ (resp. $[0, 150]$) to `g` when it reads 0 from it. Since each thread can modify the variable read by the other, they repeat the analysis starting from the join of the new interval with the initial interval. In this iteration, they discover that thread 2 can write $[0, 250]$ to `g` when it reads $[0, 150]$ from it. The analysis now incorrectly determines that it needs to re-analyze thread 2, because thread 1 also wrote $[0, 250]$ in the previous iteration and that is a larger interval than that read by thread 2. This is the reasoning behind the false alarm. The core problem here is that these methods are path-insensitive across thread context switches and that is insufficient to prove this assertion. The analysis is accounting for a thread context switch that can never happen (the one that flows $[0, 250]$ to thread 2 before thread 2 increments `g`). More recent approaches [14,20] can achieve a higher degree of flow-sensitivity but they either require manual annotations to guide the trace partitioning or are restricted to program locations outside of a loop body.

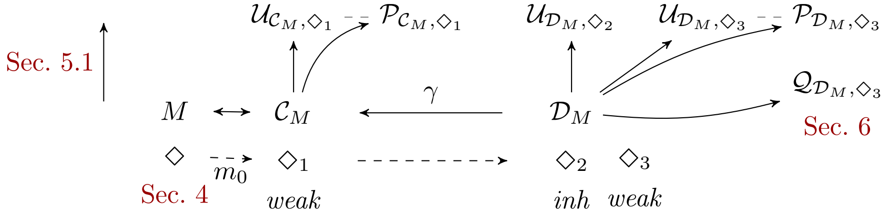
Our key contribution is an unfolding that is flow- and path-sensitive across interfering statements of the threads, and path-insensitive inside the non-interfering blocks of statements. Figure 1 (c) shows the unfolding structure that our method explores for this program. The boxes in this structure are called *events* and they represent the action of firing a transformer after a history of firings. The arrows depict *causality* constraints between events, i.e., the *happens-before* relation. Dotted lines depict the immediate *conflict relation*, stating that two events cannot be simultaneously present in the same concurrent execution, known as *configuration*. This structure contains three maximal configurations (executions), which correspond to the three meaningful ways in which the statements reading or writing to variable `g` can interleave.

Conceptually, we can construct this unfolding using the following idea: start by picking an arbitrary interleaving. Initially we pick the empty one which reaches the initial state of the program. Now we run a sequential abstract interpreter on one thread, say thread 1, from that state and stop on every location that reads or writes a global variable. In this case, the analyzer would stop at the statement `g += i` with the invariant that $\langle g \mapsto [0,0], i \mapsto [0,100]\rangle$. This invariant corresponds to the first event of the unfolding (top-left corner). The unfolding contains now a new execution, so we iterate again the same procedure by picking the execution consisting of the event we just discovered. We run the analyser on thread 2 from the invariant reached by that execution and stop on any global action. That gives rise to the event `g+=j`, and in the next step using the execution composed of the two events we have seen, we discover its causal successor `a()` (representing the assert statement). Note however that before visiting that event, we could have added event `g+=j` corresponding to the invariant of running an analyser starting from the initial state on thread 2. Furthermore, since both invariants are related to the same shared variable, these two events must not be present in the same execution. We enforce that with the conflict relation.

Our method mitigates the aforementioned branching explosion of the POR tree because it never unfolds the conflicting branches of one thread (loop iterations in our example). In comparison to thread-modular analysis, it remains precise about the context switches because it uses a history-preserving data structure.

Another novelty of our approach is the observation that certain events are *equivalent* in the sense that the state associated with one is *subsumed* by the second. In our example, one of these events, known as a *cutoff event*, is labelled by `g+=i` and denoted with a striped pattern. Specifically, the configuration $\{$`g+=i`, `g+=j`$\}$ reaches the same state as $\{$`g+=j`, `g+=i`$\}$. Thus, no causal successor of a cutoff event needs to be explored as any action that we can discover from the cutoff event can be found somewhere else in the structure.

Outline. The following diagram displays the various concepts and transformations presented in the paper:



$$\begin{tikzcd}
 & & \mathcal{U}_{\mathcal{C}_M,\diamond_1} \arrow[r, dashed] & \mathcal{P}_{\mathcal{C}_M,\diamond_1} & \mathcal{U}_{\mathcal{D}_M,\diamond_2} & \mathcal{U}_{\mathcal{D}_M,\diamond_3} \arrow[r, dashed] & \mathcal{P}_{\mathcal{D}_M,\diamond_3} \\
\text{Sec. 5.1} \arrow[u] & & & & & & \mathcal{Q}_{\mathcal{D}_M,\diamond_3} \\
 & M \arrow[r, leftrightarrow] & \mathcal{C}_M \arrow[uu] \arrow[uur, bend left] & & \mathcal{D}_M \arrow[ll, "\gamma"'] \arrow[uu] \arrow[uur] \arrow[uurr, bend left] \arrow[urr, bend right] & & \text{Sec. 6} \\
 & \diamond \arrow[r, dashed, "m_0"'] & \diamond_1 \arrow[rr, dashed] & & \diamond_2 & \diamond_3 & \\
 & \text{Sec. 4} & \textit{weak} & & \textit{inh} & \textit{weak} &
\end{tikzcd}$$

Fig. 2. Overview diagram

Let M be the program under analysis, whose concrete semantics $\mathcal{C}_M$ is abstracted by a domain $\mathcal{D}_M$. The relations $\diamond$ and $\diamond_i$ are independence relations with different levels of granularity over the transformers of M, $\mathcal{C}_M$, or $\mathcal{D}_M$. We denote by $\mathcal{U}_{\mathcal{D}',\diamond'}$ the *unfolding* of $\mathcal{D}'$ (either $\mathcal{C}_M$ or $\mathcal{D}_M$) under independence relation $\diamond'$ (either $\diamond_1$ or $\diamond_2$). This transformation is defined in Sec. 5.1. Whenever we unfold a domain using a weak independence relation ($\diamond_2$ on $\mathcal{C}_M$ and $\diamond_3$ on $\mathcal{D}_M$), we can use cutoffs to prune the unfolding, represented by the dashed line between unfoldings. The resulting unfolding, defined in Sec. 6.1, is denoted by the letter $\mathcal{P}$. The main contribution of our work is the *compact unfolding*, $\mathcal{Q}_{\mathcal{D}_M,\diamond_3}$, an example of which was given in Fig. 1 (c).

3 Preliminaries

There is no new material in this section, but we recommend the reader to review the definition of an analysis instance, which is not standard.

Concurrent programs. We model the semantics of a concurrent, non-deterministic program by a labelled transition system $M := \langle \Sigma, \to, A, s_0 \rangle$, where Σ is the set of *states*, A is the set of *program statements*, $\to \subseteq \Sigma \times A \times \Sigma$ is the transition relation, and s_0 is the *initial state*. The identifier of the thread containing a statement a is given by a function $p: A \to \mathbb{N}$. If $s \xrightarrow{a} s'$ is a transition, the statement a is *enabled* at s, and a can *fire* at s to produce s'. We let *enabl*(s) denote the set of statements enabled at s. As statements may be non-deterministic, firing a may produce more than one such s'. A sequence $\sigma := a_1 \dots a_n \in A^*$ is a *run* when there are states $s_1, \dots, s_n$ satisfying $s_0 \xrightarrow{a_1} s_1 \dots \xrightarrow{a_n} s_n$. For such σ we define $\mathit{state}(\sigma) := s_n$. We let $\mathit{runs}(M)$ denote the set of all runs of M, and $\mathit{reach}(M) := \{\mathit{state}(\sigma) \in \Sigma : \sigma \in \mathit{runs}(M)\}$ the set of all *reachable states* of M.

Analysis Instances. A lattice $\langle D, \sqsubseteq_D, \sqcup_D, \sqcap_D \rangle$ is a poset with a binary, least upper bound operator $\sqcup_D$ called *join* and a binary, greatest lower bound operator $\sqcap_D$ called *meet*. A *transformer* $f: D \to D$ is a monotone function on D. A *domain* $\langle D, \sqsubseteq, F \rangle$ consists of a lattice and a set of transformers. We adopt standard assumptions in the literature that D has a least element $\bot$, called *bottom*, and that transformers are *bottom-strict*, i.e. $f(\bot) = \bot$. To simplify presentation, we equip domains with sufficient structure to lift notions from transition systems to domains, and assume that domains represent control and data states.

Definition 1. *An analysis instance $\mathcal{D} := \langle D, \sqsubseteq, F, d_0 \rangle$, consists of a domain $\langle D, \sqsubseteq, F \rangle$ and an initial element $d_0 \in D$.*

A transformer f is *enabled* at an element d when $f(d) \neq \bot$, and the result of *firing* f at d is $f(d)$. The element *generated by* or *reached by* a sequence of transformers $\sigma := f_1, \dots, f_m$ is the application $\mathit{state}(\sigma) := (f_m \circ \dots \circ f_1)(d_0)$ of transformers in σ to d_0. Let $\mathit{reach}(\mathcal{D})$ be the set of reachable elements of $\mathcal{D}$. The sequence σ is a *run* if $\mathit{state}(\sigma) \neq \bot$ and $\mathit{runs}(\mathcal{D})$ is the set of all runs of $\mathcal{D}$.

The *collecting semantics* of a transition system M is the analysis instance $\mathcal{C}_M := \langle \mathcal{P}(\Sigma), \subseteq, F, \{s_0\} \rangle$, where F contains a transformer $f_a(S) := \{s' \in \Sigma : s \in S \land s \xrightarrow{a} s'\}$ for every statement a of the program. The *pointwise-lifting* of a relation $R \subseteq A \times A$ on statements to transformers in $\mathcal{D}$ is $R_{\mathcal{D}} = \{\langle f_a, f_{a'} \rangle \mid \langle a, a' \rangle \in R\}$. Let $m_0: A \to F$ be map from statements to transformers: $m_0(a) := f_a$. An analysis instance

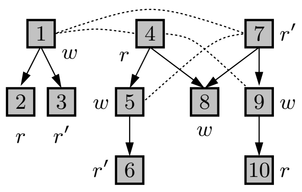


Fig. 3. A PES with 10 events, labelled by elements in $\{w, r, r'\}$.

$\bar{\mathcal{D}} = \langle \bar{D}, \bar{\sqsubseteq}, \bar{F}, \bar{d}_0 \rangle$ is an *abstraction* of $\langle D, \sqsubseteq, F, d_0 \rangle$ if there exists a *concretization function* $\gamma : \bar{D} \to D$, which is monotone and satisfies that $d_0 \sqsubseteq \gamma(\bar{d}_0)$, and that $f \circ \gamma \sqsubseteq \gamma \circ \bar{f}$, where the order between functions is pointwise.

Labelled Prime Event Structures. Event structures are tree-like representations of system behaviour that use partial orders to represent concurrent interaction. Fig. 3 depicts an event structure. The nodes are events and solid arrows, represent causal dependencies: events 4 and 7 must fire before 8 can fire. The dotted line represents conflicts: 4 and 7 are not in conflict and may occur in any order, but 4 and 9 are in conflict and cannot occur in the same execution.

A *labelled prime event structure* [21] (PES) is a tuple $\mathcal{E} := \langle E, <, \#, h \rangle$ with a set of events E, a causality relation $< \subseteq E \times E$, which is a strict partial order, a conflict relation $\# \subseteq E \times E$ that is symmetric and irreflexive, and a labelling function $h : E \to X$. The components of $\mathcal{E}$ satisfy (1) the *axiom of finite causes*, that for all $e \in E$, $\{e' \in E : e' < e\}$ is finite, and (2) the *axiom of hereditary conflict*, that for all $e, e', e'' \in E$, if $e \# e'$ and $e' < e''$, then $e \# e''$.

The *history* of an event $\lceil e \rceil := \{e' \in E : e' < e\}$ is the least set of events that must fire before e can fire. A *configuration* of $\mathcal{E}$ is a finite set $C \subseteq E$ that is (i) (causally closed) $\lceil e \rceil \subseteq C$ for all $e \in C$, and (ii) (conflict free) $\neg(e \# e')$ for all $e, e' \in C$. We let $\mathit{conf}(\mathcal{E})$ denote the set of all configurations of $\mathcal{E}$. For any $e \in E$, the *local configuration* of e is defined as $[e] := \lceil e \rceil \cup \{e\}$. In Fig. 3, the set $\{1, 2\}$ is a configuration, and in fact it is a local configuration, i.e., $[2] = \{1, 2\}$. The set $\{1, 2, 3\}$ is a $\subseteq$-maximal configuration. The local configuration of event 8 is $\{4, 7, 8\}$.

Given a configuration C, we define the *interleavings* of C as the set

$$\mathit{inter}(C) := \{h(e_1), \ldots, h(e_n) : \forall e_i, e_j \in C, e_i < e_j \implies i < j\}.$$

An interleaving corresponds to the sequence labelling any topological sorting (sequentialization) of the events in the configuration. We say that $\mathcal{E}$ is finite iff E is finite. In Fig. 3, the interleavings of configuration $\{1, 2, 3\}$ are wrr' and $wr'r$.

Event structures are naturally (partially) ordered by a *prefix* relation $\trianglelefteq$. Given two PESs $\mathcal{E} := \langle E, <, \#, h \rangle$ and $\mathcal{E}' := \langle E', <', \#', h' \rangle$, we say that $\mathcal{E}$ is a *prefix* of $\mathcal{E}'$, written $\mathcal{E} \trianglelefteq \mathcal{E}'$, when $E \subseteq E'$, $<$ and $\#$ are the projections of $<'$ and $\#'$ to E, and $E \supseteq \{e' \in E' : e' < e \wedge e \in E\}$. Moreover, the set of prefixes of a given PES $\mathcal{E}$ equipped with $\trianglelefteq$ is a complete lattice.

4 Independence for Transformers

Partial-order reduction tools use a notion called independence to avoid exploring concurrent interleavings that lead to the same state. Our analyzer uses independence between transformers to compactly represent transformer applications that lead to the same result. The contribution of this section is a notion of independence for transformers (represented by the lowest horizontal arrows in Fig. 2) and a demonstration that abstraction may both create and violate independence relationships.

We recall a standard notion of independence for statements [22,10]. Two statements a, a' of a program M *commute at* a state s iff

- if $a \in enabl(s)$ and $s \xrightarrow{a} s'$, then $a' \in enabl(s)$ iff $a' \in enabl(s')$; and
- if $a, a' \in enabl(s)$, then there is a state s' such that $s \xrightarrow{a.a'} s'$ and $s \xrightarrow{a'.a} s'$.

Independence between statements is an underapproximation of commutativity. A relation $\diamond \subseteq A \times A$ is an *independence* for M if it is symmetric, irreflexive, and satisfies that every $(a, a') \in \diamond$ commute at every reachable state of M. In general, M has multiple independence relations; $\varnothing$ is always one of them.

Suppose that independence for transformers is defined by replacing statements and transitions with transformers and transformer application, respectively. Ex. 1 illustrates that an independence relation on statements cannot be lifted to obtain transformers that are independent under such a notion.

Example 1. Consider the collecting semantics $\mathcal{C}_M$ of a program M with two variables, `x` and `y`, two statements $a :=$ `assume(x==0)` and $a' :=$ `assume(y==0)`, and initial element $d_0 := \{\langle x \mapsto 0, y \mapsto 1\rangle, \langle x \mapsto 1, y \mapsto 0\rangle\}$. Since a and a' read different variables, $R := \{\langle a, a'\rangle, \langle a', a\rangle\}$ is an independence relation on M. Now observe that $\{\langle f_a, f_{a'}\rangle, \langle f_{a'}, f_a\rangle\}$ is not an independence relation on $\mathcal{C}_M$, as f_a and $f_{a'}$ disable each other. Note, however, that $f_a(f_{a'}(d_0))$ and $f_{a'}(f_a(d_0))$ are both $\bot$.

Weak independence, defined below, allows transformers to be considered independent even if they disable each other.

Definition 2. *Let $\mathcal{D} := \langle D, \sqsubseteq, F, d_0\rangle$ be an analysis instance. A relation $\diamond \subseteq F \times F$ is a weak independence on transformers if it is symmetric, irreflexive, and satisfies that $f \diamond f'$ implies $f(f'(d)) = f'(f(d))$ for every $d \in reach(\mathcal{D})$. Moreover, $\diamond$ is an* independence *if it is a weak independence and satisfies that if $f(d) \neq \bot$, then $(f \circ f')(d) \neq \bot$ iff $f'(d) \neq \bot$, for all $d \in reach(\mathcal{D})$.*

Recall that $R_{\mathcal{D}}$ is the lifting of a relation on statements to transformers. Observe that the relation R in Ex. 1, when lifted to transformers is a weak independence on $\mathcal{C}_M$. The proposition below shows that independence relations on statements generate weak independence on transformers over $\mathcal{C}_M$.

Proposition 1 (Lifted independence). *If $\diamond$ is an independence relation on M, the lifted relation $\diamond_{\mathcal{C}_M}$ is a weak independence on the collecting semantics $\mathcal{C}_M$.*

Figure 2 graphically depicts the process of lifting an independence relation. Relation $\diamond$ (on the left of the figure) is an independence relation on M. Relation $\diamond_1 \coloneqq \diamond_{\mathcal{C}_M}$ is the lifting of $\diamond$ to $\mathcal{C}_M$, and by Prop. 1 it is a weak independence relation on $\mathcal{C}_M$.

We now show that independence and abstraction are distinct notions in that transformers that are independent in a concrete domain may not be independent in the abstract, and those that are not independent in the concrete may become independent in the abstract.

Consider an analysis instance $\bar{\mathcal{D}} \coloneqq \langle \bar{D}, \bar{\sqsubseteq}, \bar{F}, \bar{d}_0 \rangle$ that is an abstraction of $\mathcal{D} \coloneqq \langle D, \sqsubseteq, F, d_0 \rangle$ and a weak independence $\diamond \subseteq F \times F$. The *inherited relation* $\bar{\diamond} \subseteq \bar{F} \times \bar{F}$ contains $\langle \bar{f}, \bar{f}' \rangle$ iff $\langle f, f' \rangle$ is in $\diamond$.

Example 2 (Abstraction breaks independence). Consider a system M with the initial state $\langle x \mapsto 0, y \mapsto 0 \rangle$, and two threads $t_1 :$ `x = 2`, $t_2 :$ `y = 7`. Let $\mathcal{I}$ be the domain for interval analysis with elements $\langle i_x, i_y \rangle$ being intervals for values of x and y. The initial state is $\bar{d}_0 = \langle x \mapsto [0,0], y \mapsto [0,0] \rangle$. Abstract transformers for t_1 and t_2 are shown below. These transformers are deliberately imprecise to highlight that sound transformers are not the most precise ones.

$$f_1(\langle i_x, i_y \rangle) = \langle [2,4], i_y \rangle \quad f_2(i_x, i_y) = \langle i_x,\ (\text{if } 3 \in i_x \text{ then } [7,9] \text{ else } [6,8]) \rangle$$

The relation $\diamond \coloneqq \{(t_1, t_2), (t_2, t_1)\}$ is an independence on M, and when lifted, $\diamond_{\mathcal{C}_M}$ is a weak independence on $\mathcal{C}_M$ (in fact, $\diamond_{\mathcal{C}_M}$ is an independence). However, the relation $\diamond_{\mathcal{I}}$ is not a weak independence because f_1 and f_2 do not commute at d_0, due to the imprecision introduced by abstraction. Consider the statements `assume(x != 9)` and `assume(x < 10)` applied to $\langle x \mapsto [0,10] \rangle$ to see that even best transformers may not commute.

On the other hand, even when certain transitions are not independent, their transformers may become independent in an abstract domain.

Example 3 (Abstraction creates independence). Consider two threads $t_1 :$ `x = 2` and $t_2 :$ `x = 3`, with abstract transformers $f_1(i_x) = [2,3]$ and $f_2(i_x) = [2,3]$. The transitions t_1 and t_2 do not commute, but owing to imprecision, $R = \{(f_1, f_2), (f_2, f_1)\}$ is a weak independence on $\mathcal{I}$.

5 Unfolding of an Abstract Domain with Independence

This section shows that unfoldings, which have primarily been used to analyze Petri nets, can be applied to abstract interpretation. This section defines the vertical arrows of Fig. 2.

An abstract unfolding is an event structure in which an event is recursively defined as the application of a transformer after a minimal set of interfering events; and a configuration represent equivalent sequences of transformer applications (events). Analogous to an invariant map in abstract interpreters and an abstract reachability tree in software model checkers, our abstract unfolding allows for constructing an over-approximation of the set of fireable transitions in a program.

5.1 The Unfolding of a Domain

Our construction generates a PES $\mathcal{E} := \langle E, <, \#, h \rangle$. Recall that a configuration is a set of events that is closed with respect to $<$ and that is conflict-free. Events in $\mathcal{E}$ have the form $e = \langle f, C \rangle$, representing that the transformer f is applied after the transformers in configuration C are applied. The order in which transformers must be applied is given by $<$, while $\#$ encodes transformer applications that cannot belong to the same configuration.

The unfolding $\mathcal{U}_{\mathcal{D},\bowtie}$ of an analysis instance $\mathcal{D} := \langle D, \sqsubseteq, F, d_0 \rangle$ with respect to a relation $\bowtie \subseteq F \times F$ is defined inductively below. Recall that a configuration C generates a set of interleavings $\mathit{inter}(C)$, which define the *state* of the configuration,

$$\mathit{state}(C) := \bigsqcap_{\sigma \in \mathit{inter}(C)} \mathit{state}(\sigma)$$

If $\bowtie$ is a weak independence relation, all interleavings lead to the same state.

Definition 3 (Unfolding). *The unfolding $\mathcal{U}_{\mathcal{D},\bowtie}$ of $\mathcal{D}$ under the relation $\bowtie$ is the structure returned by the following procedure:*

1. *Start with a PES $\mathcal{E} := \langle E, <, \#, h \rangle$ equal to $\langle \varnothing, \varnothing, \varnothing, \varnothing \rangle$.*
2. *Add a new event $e := \langle f, C \rangle$ to E, where the configuration $C \in \mathit{conf}(\mathcal{E})$ and transformer f satisfy that f is enabled at $\mathit{state}(C)$, and $\neg(f \bowtie h(e))$ holds for every $<$-maximal event e in C.*
3. *Update $<$, $\#$, and h as follows:*
   - *for every $e' \in C$, set $e' < e$;*
   - *for every $e' \in E \setminus C$, if $e \neq e'$ and $\neg(f \bowtie h(e'))$, then set $e' \# e$;*
   - *set $h(e) := f$.*
4. *Repeat steps 2 and 3 until no new event can be added to E; return $\mathcal{E}$.*

Def. 3 defines the events, the causality, and conflict relations of $\mathcal{U}_{\mathcal{D},\bowtie}$ by means of a saturation procedure. Step 1 creates an empty PES. Step 2 defines a new event from a transformer f that can be applied after configuration C. Step 3 defines e to be a causal successor of every dependent event in C, and defines e to be in conflict with dependent events not in C. Since conflicts are inherited in a PES, causal successors of e will also be in conflict with all e' satisfying $e \# e'$. Events from $E \setminus C$, which are unrelated to f in $\bowtie$, will remain concurrent to e.

Proposition 2. *The structure $\mathcal{U}_{\mathcal{D},\diamond}$ generated by Def. 3 is a uniquely defined PES.*

If $\bowtie$ is a weak independence, every configuration of $\mathcal{U}_{\mathcal{D},\bowtie}$ represents sequences of transformer applications *that produce the same element*. If C is a configuration that is local, meaning it has a unique maximal event, or if C is generated by an independence, then $\mathit{state}(C)$ will not be $\bot$. Treating transformers as independent if they generate $\bot$ enables greater reduction during analysis.

Theorem 1 (Well-formedness of $\mathcal{U}_{\mathcal{D},\diamond}$). *Let $\diamond$ be a* weak *independence on $\mathcal{D}$, let C be a configuration of $\mathcal{U}_{\mathcal{D},\diamond}$ and σ, σ' be interleavings of C. Then:*

1. $state(\sigma) = state(\sigma')$;
2. $state(\sigma) \neq \bot$ when $\diamond$ is additionally an independence relation;
3. If C is a local configuration, then also $state(\sigma) \neq \bot$.

Thm. 2 shows that the unfolding is adequate for analysis in the sense that every sequence of transformer applications leading to non-$\bot$ elements that could be generated during standard analysis with a domain will be contained in the unfolding. We emphasize that these sequences are only symbolically represented.

Theorem 2 (Adequacy of $\mathcal{U}_{\mathcal{D},\diamond}$). *For every weak independence relation $\diamond$ on $\mathcal{D}$, and sequence of transformers $\sigma \in runs(\mathcal{D})$, there is a unique configuration C of $\mathcal{U}_{\mathcal{D},\diamond}$ such that $\sigma \in inter(C)$.*

We discuss the above theorems in the context of Fig. 2. We know that $runs(M)$ is in bijective correspondance with $runs(\mathcal{C}_M)$. We said that $\diamond_1$ is a weak independence in $\mathcal{C}_M$ (see Sec. 4). By Thm. 2, every run of M is represented by a unique configuration in $\mathcal{U}_{\mathcal{C}_M,\diamond_1}$, and by Thm. 1 every configuration C of $\mathcal{U}_{\mathcal{C}_M,\diamond_1}$ such that $state(C) \neq \bot$ is such that $inter(C) \subseteq runs(M)$.

5.2 Abstract Unfoldings

The soundness theorems of abstract interpretation show when a fixed point computed in an abstract domain soundly approximates fixed points in a concrete domain. Our analysis constructs unfoldings instead of fixed points. The soundness of our analysis *does not* follow from fixed point soundness because the abstract unfolding we construct depends on the independence relation used. Though independence may not be preserved under lifting, as shown in Ex. 2, lifted relations can still be used to obtain sound results.

Example 4. In Ex. 2, the transformer composition $f_1 \circ f_2$ produces $\langle x \mapsto [2,4], y \mapsto [6,8] \rangle$, while $f_2 \circ f_2$ produces $\langle x \mapsto [2,4], y \mapsto [7,9] \rangle$. If f_1 and f_2 are considered independent, the state of the configuration $\{f_1, f_2\}$ is $state(f_1, f_2) \sqcap state(f_2, f_1)$, which is the abstract element $\langle x \mapsto [2,4], y \mapsto [7,7] \rangle$ and contains the final state $\langle x \mapsto 2, y \mapsto 7 \rangle$ reached in the concrete.

Thus, with sound abstractions, abstract transformers can be treated as (weakly) independent if their concrete counterparts were (weakly) independent, without compromising soundness of the analysis. The soundness theorem below asserts a correspondence between sequences of concrete transformer applications and the abstract unfolding. The concrete and abstract objects in Thm. 3 have different type: we are not relating a concrete unfolding with an abstract unfolding, but concrete transformer sequences with abstract configurations. Since $state(C)$ is defined as a meet of transformer sequences, the proof of Thm. 3 relies on the independence relation and has a different structure from standard proofs of fixed point soundness from transformer soundness.

Theorem 3 (Soundness of the abstraction). *Let $\bar{\mathcal{D}}$ be a sound abstraction of the analysis instance $\mathcal{D}$, let $\diamondsuit$ be a weak independence on $\mathcal{D}$, and $\bar{\diamondsuit}$ be the lifted relation on $\bar{\mathcal{D}}$. For every sequence $\sigma \in runs(\mathcal{D})$ satisfying $state(\sigma) \neq \bot$, there is a unique configuration C of $\mathcal{U}_{\bar{\mathcal{D}},\bar{\diamondsuit}}$ such that $m(\sigma) \in inter(C)$.*

Thm. 3 and Thm. 2 are fundamentally different. Thm. 2 shows that, given a domain and a weak independence relation, the associated unfolding represents all sequences of transformer applications that may be generated during the analysis of that domain. Thm. 3 relates a concrete domain with the unfolding of an abstract one. Given a concrete domain, a concrete weak independence, and an abstract domain, Thm. 3 shows that every sequence of concrete transformers has a corresponding configuration in the unfolding of the abstract domain.

Surprisingly enough, the abstract unfolding in Thm. 3 may not represent all sequences of applications of the abstract domain in isolation (because the lifted relation $\bar{\diamondsuit}$ is not necessarily a weak independence in $\bar{\mathcal{D}}$), but will represent (this is what the theorem asserts) all transformer applications of the concrete domain.

In Fig. 2, let $\diamondsuit_2$ be the lifted independence of $\diamondsuit_1$. Thm. 3 asserts that for any $\sigma \in runs(\mathcal{C}_M)$ there is a configuration C of $\mathcal{U}_{\mathcal{D}_M,\diamondsuit_2}$ such that $m(\sigma) \in inter(C)$.

6 Plugging Thread-Local Analysis

Unfoldings compactly represent concurrent executions using partial orders. However, they are a branching structure and one extension of the unfolding can multiply the number of branches, leading to a blow-up in the number of branches. Static analyses of sequential programs often avoid this explosion (at the expense of precision) by over-approximating (using join or widening) the abstract state at the CFG locations where two or more program paths converge. Adequately lifting this simple idea of merging at CFG locations from sequential to concurrent programs is a highly non-trivial problem [8].

In this section, we present a method that addresses this challenge and can mitigate the blow-up in the size of the unfolding caused by conflicts between events of the same thread. The key idea of our method is to merge abstract states generated by statements that work on local data of one thread, i.e., those whose impact over the memory/environment is invisible to other threads. Intuitively, the key insight is that we can merge certain configurations of the unfolding and still preserve its structural properties with respect to interference. The state of the resulting configuration will be a sound over-approximation of the states of the merged configurations at no loss of precision with respect to conflicts between events of different threads.

Our approach is to analyse M by constructing the unfolding of an abstract domain $\mathcal{D} := \langle D, \subseteq, F, d_0 \rangle$ and a weak independence relation $\diamondsuit$ using a thread-local procedure that over-approximates the effect of transformers altering local variables.

Assume that M has n threads. Let $F_1, \ldots, F_n$ be the partitioning of the set of transformers F by the thread to which they belong. For a transformer

Algorithm 1: Unfolding using thread-local fixpoint analysis

```
Procedure unfold($\mathcal{D}, \diamond, n$)
    Set $\mathcal{E} := \langle E, <, \#, h\rangle$ to $\langle \varnothing, \varnothing, \varnothing, \varnothing\rangle$
    forall $i, C$ in $\mathbb{N}_n \times conf(\mathcal{E})$
        for $f$ enabled on tla($i$, state($C$))
            $e :=$ mkevent($f, C, \diamond$)
            if iscutoff($e, \mathcal{E}$) continue
            Add $e$ to $E$
            Extend $<$, $\#$, and $h$ with $e$.
Procedure mkevent($f, C, \diamond$)
    do
        Remove from $C$ any $<$-maximal
          event $e$ such that $f \diamond h(e)$
    while $C$ changed
    return $\langle$mklabel($f$)$, C\rangle$
Procedure mklabel($f$)
    return $d \mapsto f($tla$(p(f), d))$
```

$f \in F_i$, we let $p(f) := i$ denote the thread to which f belongs. We define, per thread, the (local) transformers which can be used to run the merging analysis. A transformer $f \in F_i$ is *local* when, for all other threads $j \neq i$ and all transformers $f' \in F_j$ we have $f \diamond f'$. A transformer is *global* if it is not local. We denote by F_i^{loc} and F_i^{glo}, respectively, the set of local and global transformers in F_i. In Fig. 1 (a), the global transformers would be those representing the actions to the variable `g`. The remaining statements correspond to local transformers.

We formalize the thread-local analysis using the function $\texttt{tla}\colon \mathbb{N} \times D \to D$, which plays the role of an off-the-shelf static analyzer for sequential thread code. A call to $\texttt{tla}(i, d)$ will run a static analyzer on thread i, restricted to F_i^{loc}, starting from d, and return its result which we assume to be a sound fixed point. Formally, we assume that if $\texttt{tla}(i, d)$ returns $d' \in D$, then for every sequence $f_1 \ldots f_n \in (F_i^{\text{loc}})^*$ we have $(f_n \circ \ldots \circ f_1)(d) \sqsubseteq d'$. This condition requires any implementation of $\texttt{tla}(i, d)$ to return a sound approximation of the state that thread i could possibly reach after running only local transformers starting from d.

Alg. 1 presents the overall approach proposed in this paper. Procedure `unfold` builds an *abstract unfolding* for $\mathcal{D}$ under independence relation $\diamond$. It non-deterministically selects a thread i and a configuration C and runs a sequential static analyzer on thread i starting on the state reached by C. If a global transformer $f \in F_i^{\text{glo}}$ is eventually enabled, the algorithm will try to insert it into the unfolding. For that it first calls the function `mkevent`, which constructs a history for f from C according to Def. 3, i.e., by removing from C events independent with f until all maximal events are dependent. Function `mkevent` then calls `mklabel` to construct a suitable label for the new event. Labels are functions in $D \to D$, we discuss them below. If the resulting event $e := \langle \bar{f}, H\rangle$ is a *cutoff*, i.e., an *equivalent* event is already in the unfolding prefix, then it will be ignored. Otherwise, we add it to E. Finally, we update relations $<$, $\#$, and h using exactly the same procedure as in Step 3 of Def. 3. In particular, we set $h(e) := \bar{f}$.

Unlike the unfolding $\mathcal{U}_{\mathcal{D},\diamond}$ of Def. 3, the events of the PES constructed by Alg. 1 are not labelled by transformers in F, but by ad-hoc functions constructed by `mklabel`. An event in this PES represents the aggregated application of multiple local transformers, summarized by `tla` into a fixed point, followed by the application of a global transformer. Function `mklabel` constructs a *collapsing transformer* that represents such transformation of the program state. Given a global transformer f it returns a function $\overline{f} \in D \to D$ that maps a state d to another state obtained by first running `tla` on f's thread starting from d and then running f. While an efficient implementation of Alg. 1 does not need to actually construct this transformer, we formalize it here because it is necessary to define how to compute the state of a configuration, $\mathit{state}(C)$.

We denote by $\mathcal{Q}_{\mathcal{D},\diamond}$ the PES constructed by a call to $\texttt{unfold}(\mathcal{D},\diamond,n)$. When the `tla` performs a path-insensitive analysis, the structure $\mathcal{Q}_{\mathcal{D},\diamond}$ is (i) path-insensitive for runs that execute only local code, (ii) partially path-sensitive for runs that execute one or more global transformer, and (iii) flow-sensitive with respect to interference between threads. We refer to this analysis as a *causally-sensitive* analysis as it is precise with respect to the *dynamic* interference between threads.

Alg. 1 embeds multiple constructions explained in this paper. For instance, when `tla` is implemented by the function $g(d,i) \coloneqq d$ and the check of cutoffs is disabled (`iscutoff` systematically returns *false*), the algorithm is equivalent to Def. 3. We now show that $\mathcal{Q}_{\mathcal{D},\diamond}$ is a safe abstraction of $\mathcal{D}$ when `tla` performs a non-trivial operation.

Theorem 4 (Soundness of the abstraction). *Let $\diamond$ be a weak independence on $\mathcal{D}$ and $\mathcal{P}_{\mathcal{D},\diamond}$ the PES computed by a call to* $\texttt{unfold}(\mathcal{D},\diamond,n)$ *with cutoff checking disabled. Then, for any execution $\sigma \in \mathit{runs}(\mathcal{D})$ there is a unique configuration C in $\mathcal{P}_{\mathcal{D},\diamond}$ such that $\hat{\sigma} \in \mathit{inter}(C)$.*

6.1 Cutoff Events: Pruning the Unfolding

If we remove the conditional statement in line 6 of Alg. 1, the algorithm would only terminate if every run of $\mathcal{D}$ contains finitely many global transformers. This conditional check has two purposes: (1) preventing infinite executions from inserting infinitely many events into $\mathcal{E}$; (2) pruning branches of the unfolding that start with *equivalent* events. The procedure `iscutoff` decides when an event is marked as a *cutoff* [17]. In such cases, no causal successor of the event will be explored. The implementation of `iscutoff` cannot prune "too often", as we want the computed PES to be a *complete* representation of behaviours of $\mathcal{D}$ (e.g., if a transformer is fireable, then some event in the PES will be labelled by it).

Formally, given $\mathcal{D}$, a PES $\mathcal{E}$ is $\mathcal{D}$*-complete* iff for every reachable element $d \in \mathit{reach}(\mathcal{D})$ there is a configuration C of $\mathcal{E}$ such that $\mathit{state}(C) \sqsupseteq d$. The key idea behind cutoff events is that, if event e is marked as a cutoff, then for any configuration C that includes e it must be possible to find a configuration C' without cutoff events such that $\mathit{state}(C) \sqsubseteq \mathit{state}(C')$. This can be achieved by defining $\texttt{iscutoff}(e,\mathcal{E})$ to be the predicate: $\exists e' \in \mathcal{E}$ such that $\mathit{state}([e]) \sqsubseteq \mathit{state}([e'])$ and $|[e']| < |[e]|$. When such e' exists, including the event e in $\mathcal{E}$ is unnecessary because any configuration C such that $e \in C$ can be replayed in $\mathcal{E}$ by first executing $[e']$ and then (copies of) the events in $C \setminus [e]$.

We now would like to prove that Alg. 1 produces a $\mathcal{D}$-complete prefix when instantiated with the above definition of `iscutoff`. However, a subtle an unexpected interaction between the operators `tla` and `iscutoff` makes it possible to prove Thm. 5 only when `tla` *respects independence*. Formally, we require `tla` to satisfy the following property: for any $d \in \mathit{reach}(\mathcal{D})$ and any two global transformers $f \in F_i^{\text{glo}}$ and $f' \in F_j^{\text{glo}}$, if $f \diamond f'$ then

$$(f' \circ \mathtt{tla}(j) \circ f \circ \mathtt{tla}(i))(d) = (f \circ \mathtt{tla}(i) \circ f' \circ \mathtt{tla}(j))(d)$$

When `tla` does not respect independence, it may over-approximate the global state (e.g. via joins and widening) in a way that breaks the independence of otherwise independent global transformers. This triggers the cutoff predicate to incorrectly prune necessary events.

Theorem 5. *Let $\diamond$ be a weak independence in $\mathcal{D}$. Assume that* `tla` *respects independence and that* `iscutoff` *uses the procedure defined above. Then the PES $\mathcal{Q}_{\mathcal{D},\diamond}$ computed by Alg. 1 is $\mathcal{D}$-complete.*

Note that Alg. 1 terminates if the lattice order $\sqsubseteq$ is a well partial order (every infinite sequence contains an increasing pair). This includes, for instance, all finite domains. Furthermore, it is also possible to accelerate the termination of Alg. 1 using widenings in `tla` to *force cutoffs*. Finally, notice that while we defined `iscutoff` using McMillan's size order [17], Thm. 5 also holds if `iscutoff` is defined using adequate orders [5], known to yield smaller prefixes. See [24] for additional details.

7 Experimental Evaluation

In this section we evaluate our approach based on abstract unfoldings. The goal of our experimental evaluation is to explore the following questions:

- Are abstract unfoldings practical? (i.e., is our approach able to yield efficient algorithms that can be used to prove properties of concurrent programs that require precise interference reasoning?)
- How does abstract unfoldings compare with competing approaches such as thread-modular analysis and symbolic partial order reduction?

Implementation. To address these questions, we have implemented a new program analyser based on abstract unfoldings baptized APoet, which implements an efficient variant of the exploration algorithm described in Alg. 1. The exploration strategy is based on Poet [23], an explicit-state model checker that implements a super-optimal partial order reduction method using unfoldings.

As described in Alg. 1, APoet is an analyser parameterized by a domain and a set of procedures: `tla`, `iscutoff` and `mkevent`. As a proof of concept, we have implemented an interval analysis and a basic parametric segmentation functor for arrays [4], which we instantiate with intervals and concrete integers values (to represent offsets). In this way, we are able to precisely handle arrays of threads and mutexes. APoet supports dynamic thread creation and uses CIL to inline functions calls. The analyser receives as input a concurrent C program that uses the POSIX thread library and parameters to control the widening level and the use of cutoffs. We implemented cutoffs according to the definition in Sec. 6.1 using an hash table that maps control locations to abstract values and the size of the local configuration of events.

APoet is parameterized by a domain functor of actions that is used to define independence and control the `tla` procedure. We have implemented an instance of the domain of actions for memory accesses and thread synchronisations. Transformers *record* the segments of the memory, intervals of addresses or sets of addresses, that have been read or written and synchronisation actions related to thread creation, join and mutex lock and unlock operations. This approach is used to compute a conditional independence relation as transformers can perform different actions depending on the state. The conditional independence relation is dynamically computed and is used in the procedure `mkevent`.

Finally, the `tla` procedure was implemented with a worklist fixpoint algorithm which uses the widening level given as input. In the interval analysis, we guarantee that `tla` respects independence using a predicate over the actions that identifies whether a transformer is local or global. We currently support two modes in APoet: one that assumes programs are data-race free and considers any thread synchronisation (i.e., thread creation/join and mutex lock/unlock) a global transformer, and a second that considers any heap access or thread synchronisation a global transformer and can be used to detect data races.

Benchmark Selection. We used six benchmarks adapted from the SV-COMP'17 (corresponding to nine rows in Table 1) and four parametric programs (the remaining fifteen rows in Table 1) written by the authors: map-reduce DNA sequence analysis, producer-consumer, parallel sorting, and a thread pool. The majority of the SV-COMP benchmarks are not applicable for this evaluation since they are data deterministic (whereas our approach is primarily for data non-deterministic programs) or create unboundedly many threads, or use non-integer data types (e.g., structs, which are not currently supported by our prototype). Thus, we devised parametric benchmarks that expose data non-determinism and complex synchronization patterns, where the correctness of assertions depend on the synchronization history. We believe that all new benchmarks are as complex as the most complex ones of the SV-COMP (excluding device drivers).

Each program was annotated with assertions enforcing, among others, properties related to thread synchronisation (e.g., after spawning the worker threads, the master analyses results only after all workers finished), or invariants about data (e.g., each thread accesses a non-overlapping segment of the input array).

Tool Selection. We compare our approach against the two approaches most closely related to ours: abstract interpretation based on thread-modular methods

Table 1. Experimental results. All experiments with APoet, Impara and cbmc were performed on an Intel Xeon CPU with 2.4 GHz and 4 GB memory with a timeout of 30 minutes; AstreeA was ran on HP ZBook with 2.7 GHz i7 processor and 32 GB memory. Columns are: P: nr. of threads; A: nr. of assertions; $t(s)$: running time (TO - timeout); E: nr. of events in the unfolding; E_{cut}: nr. of cutoff events; W: nr. of warnings; V: verification result (S - safe; U - unsafe); N: nr. of node states; A $*$ marks programs containing bugs. <2 reads as "*less than 2*".

Benchmark			APoet				AstreeA		Impara			cbmc 5.6	
Name	P	A	$t(s)$	E	E_{cut}	W	$t(s)$	W	V	$t(s)$	N	V	$t(s)$
atgc(2)	3	7	0.37	47	0	1	1.07	2	-	TO	-	S	2.37
atgc(3)	4	7	5.78	432	0	1	1.69	2	-	TO	-	S	6.6
atgc(4)	5	7	132.08	7195	0	1	2.68	2	-	TO	-	S	20.22
cond	5	2	0.55	982	0	2	0.71	2	-	TO	-	S	34.39
fmax(2,3)	2	8	0.70	100	15	0	0.31	0	-	TO	-	-	TO
fmax(3,3)	2	8	0.58	85	11	0	<2	2	-	TO	-	-	TO
fmax(5,3)	2	8	0.56	85	11	0	1.50	2	-	TO	-	-	TO
fmax(2,4)	2	8	3.38	277	43	0	<2	2	-	TO	-	-	TO
fmax(2,6)	2	8	45.82	1663	321	0	<2	2	-	TO	-	-	TO
fmax(4,6)	2	8	61.32	2230	207	0	<2	2	-	TO	-	-	TO
fmax(2,7)	2	8	146.19	3709	769	0	1.87	2	-	TO	-	-	TO
fmax(4,7)	2	8	285.23	6966	671	0	<2	2	-	TO	-	-	TO
lazy	4	2	0.01	72	0	0	0.50	2	-	TO	-	S	3.59
lazy*	4	2	0.01	72	0	1	0.49	2	-	TO	-	U	3.50
monab1	5	1	0.27	982	0	0	0.61	0	-	TO	-	S	38.51
monab2	5	1	0.25	982	0	0	0.58	1	-	TO	-	S	37.34
rand	5	1	0.40	657	0	0	3.32	0	-	TO	-	-	TO
sigma	5	5	2.62	7126	0	0	0.43	0	-	TO	-	S	189.09
sigma*	5	5	2.64	7126	0	1	0.43	1	-	TO	-	U	141.35
stf	3	2	0.01	69	0	0	0.66	2	S	5.93	250	S	2.12
tpoll(2)*	3	11	1.23	141	7	1	1.97	2	U	0.64	80	-	TO
tpoll(3)*	4	11	109.22	1712	90	2	3.77	3	U	0.72	113	-	TO
tpoll(4)*	5	11	1111.46	33018	1762	2	8.06	3	U	0.78	152	-	TO
thpool	2	24	33.47	353	103	0	1.44	5	S	TO	-	-	TO

(represented by the tool AstreeA) and partial-order reductions (PORs) handling data-nondeterminism (represented by two tools, Impara and cbmc 5.6). AstreeA implements thread-modular abstract interpretation for concurrent programs [19], Impara combines POR with interpolation-based reasoning to cope with data non-determinism [25], and cbmc uses a symbolic encoding based on partial orders [2]. We sought to compare against symbolic execution approaches for multithreaded programs, but we were either unable to obtain the tools from the authors or the tools were unable to parse the benchmarks.

Experimental Results. Table 1 presents the experimental results. When the program contained non-terminating executions (e.g., spinlocks), we used 5 loop unwindings for cbmc as well as cutoffs in APoet and a widening level of 15. For the family of fmax benchmarks, we were not able to run AstreeA on all instances, so we report approximated execution times and warnings based on the results provided by Antoine Miné on some of the instances. With respect to the size of the abstract unfolding, our experiments show that APoet is able to explore unfoldings up to 33K events and it was able to terminate on all benchmarks with an average execution time of 81 seconds. In comparison with AstreeA, APoet is far more precise: we obtain only 12 warnings (of which 5 are false positives) with APoet compared to 43 (32 false positives) with AstreeA. We observe a similar trend when comparing APoet with the mthread plugin for Frama-c [26] and confirm that the main reason for the source of imprecision in AstreeA is imprecise reasoning of thread interference. In the case of APoet, we obtain warnings in benchmarks that are buggy (lazy*, sigma* and tpoll* family), as expected. Furthermore, APoet reports warnings in the atgc benchmarks caused by imprecise reasoning of arrays combined with widening and also in the cond benchmark as it contains non-relational assertions.

APoet is able to outperform Impara and cbmc on all benchmarks. We believe that these experiments demonstrate that effective symbolic reasoning with partial orders is challenging as cbmc only terminates on 46% of the benchmarks and Impara only on 17%.

8 Related Work

In this section, we compare our approach with closely related program analysis techniques for (i) concurrent programs with (ii) a bounded number of threads and that (iii) handle data non-determinism.

The thread-modular approach in the style of rely-guarantee reasoning has been extensively studied in the past [19,18,3,16,9,14,12,20]. In [19], Miné proposes a flow-insensitive thread-modular analysis based on the interleaving semantics which forces the abstraction to cope with interleaving explosion. We address the interleaving explosion using the unfolding as an algorithmic approach to compute a flow and path-sensitive thread interference analysis. A recent approach [20] uses relational domains and trace partitioning to recover precision in thread modular analysis but requires manual annotations to guide the partitioning and does not scale with the number of global variables. The analysis in [7] is not as precise as our approach (confirmed by experiments with Duet on a simpler version of our benchmarks) as it employs an abstraction for unbounded parallelism. The work in [14] presents a thread modular analysis that uses a lightweight interference analysis to achieve an higher level of flow sensitivity similar to [7]. The interference analysis of [14] uses a constraint system to discard unfeasible pairs of read-write actions which is static and less precise than our approach based on independence. The approach is also flow-insensitive in the presence of loops with global read operations. Finally, the method in [12] focuses on manual thread-modular proofs, while our method is automatic.

The interprocedural analysis for recursive concurrent programs of [13] does not address the interleaving explosion. A related approach that uses unfoldings is the causality-based bitvector dataflow analysis proposed in [8]. There, unfoldings are used as a method to obtain dataflow information while in our approach they are the fundamental datastructure to drive the analysis. Thus we can apply thread-local fixpoint analysis while their unfolding suffers from path explosion due to local branching. Furthermore, we can build unfoldings for general domains even with invalid independence relations while their approach is restricted to the independence encoded in the syntax of a Petri net and bitvector domains.

Compared to dynamic analysis of concurrent programs [1,6,15,11], our approach builds on top of a (super-)optimal partial-order reduction [23] and is able to overcome a high degree of path explosion unrelated to thread interference.

9 Conclusion

We introduced a new algorithm for static analysis of concurrent programs based on the combination of abstract interpretation and unfoldings. Our algorithm explores an abstract unfolding using a new notion of independence to avoid redundant transformer application in an optimal POR strategy, thread-local fixed points to reduce the size of the unfolding, and a novel cutoff criterion based on subsumption to guarantee termination of the analysis.

Our experiments show that APoet generates about 10x fewer false positives than a mature thread modular abstract interpreter and is able to terminate on a large set of benchmarks as opposed to solver-based tools that have the same precision. We observed that the major reasons for the success of APoet are: (1) the use of cutoffs to cope with and prune cyclic explorations caused by spinlocks and (2) `tla` mitigates path explosion in the threads. Our analyser is able to scale with the number of threads as long as the interference between threads does not increase. As future work, we plan to experimentally evaluate the application of local widenings to force cutoffs to increase the scalability of our approach.

Acknowledgments. The authors would like to thank Antoine Miné for the invaluable help with AstreeA and the anonymous reviewers for their helpful feedback.

References

1. Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In *Principles of Programming Languages (POPL)*, pages 373–384. ACM, 2014.
2. Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *Computer Aided Verification (CAV)*, volume 8044 of *LNCS*, pages 141–157. Springer, 2013.
3. Jean-Loup Carre and Charles Hymans. From Single-thread to Multithreaded: An Efficient Static Analysis Algorithm. *arXiv:0910.5833 [cs]*, October 2009.
4. Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Principles of Programming Languages (POPL)*, pages 105–118. ACM, 2011.

5. Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of McMillan's unfolding algorithm. *Formal Methods in System Design*, 20:285–310, 2002.
6. Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. Con2Colic testing. In *Foundations of Software Engineering (FSE)*, pages 37–47. ACM, 2013.
7. Azadeh Farzan and Zachary Kincaid. Verification of parameterized concurrent programs by modular reasoning about data and control. In *Principles of Programming Languages (POPL)*, pages 297–308. ACM, 2012.
8. Azadeh Farzan and P. Madhusudan. Causal dataflow analysis for concurrent programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *LNCS*, pages 102–116. Springer, 2007.
9. Cormac Flanagan and Shaz Qadeer. Thread-modular model checking. In *Model Checking Software*, volume 2648 of *LNCS*, pages 213–224. Springer, May 2003.
10. Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer, 1996.
11. Henning Günther, Alfons Laarman, Ana Sokolova, and Georg Weissenbacher. Dynamic reductions for model checking concurrent software. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 10145 of *LNCS*, pages 246–265. Springer, 2017.
12. Jochen Hoenicke, Rupak Majumdar, and Andreas Podelski. Thread modularity at many levels: A pearl in compositional verification. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 473–485, New York, NY, USA, 2017. ACM.
13. Bertrand Jeannet. Relational interprocedural verification of concurrent programs. *Software & Systems Modeling*, 12(2):285–306, March 2012.
14. Markus Kusano and Chao Wang. Flow-sensitive composition of thread-modular abstract interpretation. In *Foundations of Software Engineering (FSE)*, pages 799–809. ACM, 2016.
15. Kari Kähkönen, Olli Saarikivi, and Keijo Heljanko. Unfolding based automated testing of multithreaded programs. *Automated Software Engineering*, 22:1–41, May 2014.
16. Alexander Malkis, Andreas Podelski, and Andrey Rybalchenko. Precise thread-modular verification. In *Static Analysis (SAS)*, volume 4634 of *LNCS*, pages 218–232. Springer, August 2007.
17. Kenneth L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Computer Aided Verification (CAV)*, volume 663 of *LNCS*, pages 164–177. Springer, 1993.
18. Antoine Miné. Static analysis of run-time errors in embedded real-time parallel C programs. *Logical Methods in Computer Science*, 8(1), March 2012.
19. Antoine Miné. Relational thread-modular static value analysis by abstract interpretation. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 8318 of *LNCS*, pages 39–58. Springer, 2014.
20. Raphaël Monat and Antoine Miné. Precise thread-modular abstract interpretation of concurrent programs using relational interference abstractions. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 10145 of *LNCS*, pages 386–404. Springer, 2017.
21. Mogens Nielsen, Gordon Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13(1):85–108, 1981.
22. Doron Peled. All from one, one for all: on model checking using representatives. In *Computer Aided Verification (CAV)*, volume 697 of *LNCS*, pages 409–423. Springer, 1993.

23. César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based partial order reduction. In *Concurrency Theory (CONCUR)*, volume 42 of *LIPIcs*, pages 456–469. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2015.
24. Marcelo Sousa, César Rodríguez, Vijay D'Silva, and Daniel Kroening. Abstract interpretation with unfoldings. *CoRR*, abs/1705.00595, 2017.
25. Björn Wachter, Daniel Kroening, and Joël Ouaknine. Verifying multi-threaded software with Impact. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 210–217, 2013.
26. Boris Yakobowski and Richard Bonichon. Frama-C's Mthread plug-in. Report, Software Reliability Laboratory, 2012.