Verification of concurrent programs using trace abstraction refinement

Franck Cassez[1] and Frowin Ziegler[2]

[1] Macquarie University, NICTA/UNSW, Sydney, Australia
[2] Augsburg University, Germany

Abstract. Verifying concurrent programs is notoriously hard due to the state explosion problem: 1) the data state space can be very large as the variables can range over very large sets, and 2) the control state space is the Cartesian product of the control state space of the concurrent components and thus grows exponentially in the number of components. On the one hand, the most successful approaches to address the control state explosion problem are based on assume-guarantee reasoning or model-checking coupled with partial order reduction. On the other hand, the most successful techniques to address the data space explosion problem for sequential programs verification are based on the abstraction/refinement paradigm which consists in refining an abstract over-approximation of a program via predicate refinement. In this paper, we show that we can combine partial order reduction techniques with trace abstraction refinement. We apply our approach to standard benchmarks and show that it matches current state-of-the-art analysis techniques.

1 Introduction

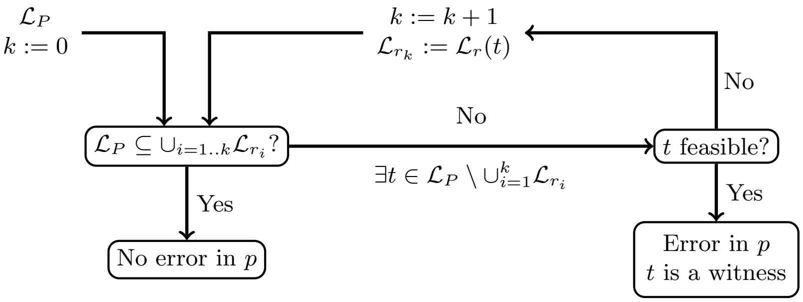
Multi-core architectures enable hardware consolidation i.e., less weight and less space which is highly desirable for embedded systems. Multi-threaded (or concurrent) programs are designed to take full advantage of the available computing power of the multi-cores. This is very appealing performance-wise but comes at a price: concurrent programs are a lot more difficult to reason about than sequential programs. They need to *synchronise* or *share* variables and this gives rise to a multitude of subtle bugs, among them deadlocks or data races that are sources of critical defects. At the same time, more and more control tasks are now implemented in software which results in large multi-threaded code bases. The major obstacle to the deployment of multi-threaded software in embedded safety critical systems is the difficulty of ensuring the absence of major critical defects. This calls for scalable automated verification techniques that can analyse multi-threaded software. Unfortunately, as witnessed by the latest *Software Verification Competition (SV-COMP 2015)* [1], there are only a few software verification tools that can analyse concurrent software. Most of them are bug finding tools and fall short of being able to establish the correctness of a program. Their applicability is often limited to rather small programs.

This is in stark contrast to the state-of-the-art for verification of sequential programs. One of the major breakthrough for verifying sequential programs is probably the *counter-example guided predicate-abstraction refinement* (CEGAR) technique [2]. CEGAR enables one to address the *data state explosion* problem by abstracting away the data into predicates. This has resulted in scalable and practicable techniques to analyse sequential programs, culminating in the design of industrial-strength tools like SLAM [3]. For concurrent programs, the *control state explosion* problem adds up to the data state explosion problem: the state space of a concurrent program is exponential in the size of the program.

Verification of Concurrent Programs. Two main techniques were designed to combat the state explosion problem in concurrent programs: *assume-guarantee* reasoning [4] and *partial order reduction* techniques [5,6,7]. An assume-guarantee property is a pair of predicates (A, G) (similar to pre and postconditions): a component guarantees to satisfy G if its environment satisfies A. Assume-guarantee reasoning consists in combining the assume-guarantee properties of each component to derive a property of the composition of the components in a modular way. Partial order reduction techniques on the other hand aim at reducing the state space to be explored in concurrent programs by removing *equivalent interleavings*. Both techniques have proved very useful in the context of finite state concurrent programs [8,9]. Combining assume-guarantee reasoning with predicate abstraction refinement for proving properties of multi-threaded C-like programs was first investigated in [10]. However the scope of the approach was limited to *modular* properties and was later extended to more general properties in [11]. The recent work in [12] introduced a combination of predicate abstraction refinement and partial order reduction. In this paper, we propose a method to combine *trace abstraction refinement* and partial order reduction techniques.

Trace Abstraction Refinement. The core principle of *trace abstraction* [13,14] for single-threaded program is to separate the data and the control flows. The abstraction of a program P is a set of *traces* e.g., sequences of *instructions* obtained by viewing the control flow graph of P as an automaton. The instructions are uninterpreted and should be viewed as mere letters in the abstraction. A trace t is *feasible* if there is at least one input of the program the trace of which is t. This is where the data flow comes into play. Some traces of a program abstraction are *error* traces e.g., leading to an error control location in the control flow graph. For instance if the program contains an `assert(cond)` statement, there is an edge in the control flow graph labelled `not(cond)` to the error location. The *language* of the program abstraction, $\mathcal{L}_P$, is composed of these error traces. Proving correctness of the program P amounts to proving that every trace in $\mathcal{L}_P$ is infeasible. This can be done by an iterative refinement algorithm as depicted in Fig. 1. A *refinement* of a program abstraction is also a language $\mathcal{L}_r$ composed exclusively of infeasible traces. One important result in [13] is that a refinement, $\mathcal{L}_r(t)$, can be computed for each infeasible trace t: this refinement contains t and other infeasible traces that are infeasible for the same reason as t is. Moreover, the refinement $\mathcal{L}_r(t)$ is a regular language. For sequential programs, the trace abstraction refinement algorithm refines the program abstraction by

computing larger and larger sets of infeasible traces, $\cup_{i=1..k}\mathcal{L}_{r_i}$. As a candidate feasible error trace must be in $\mathcal{L}_P \setminus (\cup_{i=1..k}\mathcal{L}_{r_i})$ a larger set $\cup_{i=1..k}\mathcal{L}_{r_i}$ of infeasible traces narrows down the search at each iteration of the algorithm and in this respect refines the trace abstraction. If at some point all the error traces of the program abstraction are infeasible, the program can be declared error-free. Of course the algorithm is not guaranteed to terminate for C-like programs (with more than two counters) but it is *sound* and relatively *complete* (see [13]).
Trace Abstraction Refinement for Concurrent Programs. The trace abstraction refinement algorithm can be used for concurrent programs, say $P_1 \parallel P_2$, by considering the language $\mathcal{L}_{P_1 \parallel P_2}$. However, the algorithm may need multiple iterations to rule out infeasible traces that are *equivalent* in the sense that some instructions along a trace may be swapped without altering the correctness of the concurrent program, e.g., deadlock freedom.



$\mathcal{L}_P$
$k := 0$
$k := k + 1$
$\mathcal{L}_{r_k} := \mathcal{L}_r(t)$
No
No
$\mathcal{L}_P \subseteq \cup_{i=1..k}\mathcal{L}_{r_i}$?
t feasible?
$\exists t \in \mathcal{L}_P \setminus \cup_{i=1}^{k}\mathcal{L}_{r_i}$
Yes
Yes
No error in p
Error in p
t is a witness

Fig. 1. Trace Abstraction Refinement Algorithm

The solution proposed in [15] is to extend the expressiveness of the trace abstraction refinements: instead of building an automaton that accepts a regular language $\mathcal{L}_r(t)$, the authors define a new refinement device: *Inductive Data Flow Graphs* (iDFGs). One of the nice features of iDFGs is that they can represent infeasibility reasons on sequences of actions while capturing some independence between the ordering of actions. The trace refinement algorithm of Fig. 1 can be adapted to this setting because: 1) iDFGs are closed under union, and 2) language inclusion between regular languages ($\mathcal{L}_P$) and iDFGs is decidable. The approach of [15] is elegant and versatile as it applies to a large spectrum of properties of concurrent programs (Owicki-Gries proof statements which are invariance properties closed under multi-threaded Cartesian abstraction). However the core operation of the algorithm is rather expensive: language inclusion between a parallel composition of n threads and an iDFG is in PSPACE. To the best of our knowledge this technique has not been implemented yet.

Our Contribution. In this work we propose a simple and powerful combination of trace abstraction refinement and partial order reduction. It turns out that combining trace abstraction refinement with partial order (or symmetry) reduction techniques is simplified when instead of proving general Owicki-Gries style statements one restricts to *reachability properties*[3]. This is supported by our main result, Theorem 3. We also argue that combining partial order reduction techniques with refinement techniques is easier and more natural with trace abstraction refinement (trace based) rather than predicate abstraction refinement (state based). The advantages of our combination are manifold. First, the refinement algorithm is simple and builds on two distinct and orthogonal techniques: partial order reduction algorithms and trace abstraction refinement. Second, the combination is even valid with any reasonable *reduction* e.g., symmetry reduction. More importantly, our technique goes beyond discovering bugs and is able to establish program correctness: this is in contrast to state-of-the-art tools (e.g., MU-CSEQ [16], LAZY-CSEQ [17]) for analysing concurrent programs that are based on *bounded model checking* techniques.

Outline of the paper. In Section 2 we define the model of concurrent programs. Section 3 shows how to reduce the existence of a feasible trace in a concurrent program to the existence of a trace in a reduced concurrent program (partial order reduction). Section 4 presents an algorithm that combines trace abstraction refinement with partial order reduction. Experimental results are presented in Section 5. Section 6 is devoted to related work.

2 Reachability Checking in Concurrent Programs

In this section we define concurrent programs. For the sake of clarity and following [10], we restrict to 2-threaded programs but all the definitions and proofs carry over to the general setting of n-threaded programs with $n \geq 2$ (see [18]).

2.1 Notations

Let V be a fixed finite set of integer variables. A *valuation* ν is a mapping $\nu : V \to \mathbb{Z}$ and we write $\mathbb{Z}^V$ for the set of valuations. We let Σ be a fixed set of *instructions* with variables in V. Instructions can be either *assignments* with side effects or *conditions* that are side-effect free. Σ^* is the set of finite sequences of instructions and ε is the *empty sequence*. We write $v.w$ for the concatenation of two words $v, w \in \Sigma^*$. We let $|w|$ be the length of $w \in \Sigma^*$ ($|\varepsilon| = 0$). Given $i \in \Sigma$, $\mathcal{R}(i) \subseteq V$ (resp. $\mathcal{W}(i) \subseteq V$) is the set of *read-from* (resp. *written-to*) variables. We let $\mathcal{V}(i) = \mathcal{R}(i) \cup \mathcal{W}(i)$.

The *semantics* of an instruction $i \in \Sigma$ is given by a relation $[\![i]\!] \subseteq \mathbb{Z}^V \times \mathbb{Z}^V$. The post operator is defined for each instruction as follows: given $U \subseteq \mathbb{Z}^V, \mathsf{post}(i, U) = \{u' \mid \exists u \in U, (u, u') \in [\![i]\!]\}$. The post operator extends to sequences of instructions: let $v \in \Sigma^*, i \in \Sigma, \mathsf{post}(v.i, U) = \mathsf{post}(i, \mathsf{post}(v, U))$ with $\mathsf{post}(\varepsilon, U) = U$. Given a sequence of instructions $w \in \Sigma^*$, w is *feasible* iff $\mathsf{post}(w, \mathbb{Z}^V) \neq \varnothing$. Otherwise w is infeasible. Sets of valuations can de defined by *predicates* e.g., as Boolean combinations of terms in a given logic (e.g., Linear

[3] Without loss of generality we focus on reachability of control locations as reachability of a specific data state can easily be encoded in this setting.

Integer Arithmetic). The predicate $True$ denotes the set of all valuations and $False$ the empty set of valuations. Feasibility of a trace $w \in \Sigma^*$ thus reduces to $\mathsf{post}(w, True) \not\subseteq False$.

A transition system $\mathcal{S}$ is a tuple (S, S_0, δ) with S a set of states, $S_0 \subseteq S$ the set of *initial* states and $\delta \subseteq S \times \Sigma \times S$ the *transition relation.*

An instruction $i \in \Sigma$ is *enabled* in s if $(s, i, s') \in \delta$ for some s'. A *path* in $\mathcal{S}$ from s_0 to s_n, $n \geq 0$, is a sequence $s_0\ i_0\ s_1\ i_1\ \cdots\ s_{n-1}\ i_{n-1}\ s_n$ with $\forall 0 \leq k \leq n-1, (s_k, i_k, s_{k+1}) \in \delta$. The trace of a path $s_0\ i_0\ s_1\ i_1\ \cdots\ s_{n-1}\ i_{n-1}\ s_n$ is $i_0.i_1.\cdots.i_{n-1}$. We write $s_0 \xrightarrow{i_0.i_1.\cdots.i_{n-1}} s_n$ when there is a path from s_0 to s_n with trace $i_0.i_1.\cdots.i_{n-1}$. A state s is reachable if $s_0 \xrightarrow{t} s$ for some $s_0 \in S_0$ and $t \in \Sigma^*$. We let $\textsc{Reach}(\mathcal{S})$ be the set of reachable states in $\mathcal{S}$.

```
1  shared int x, y, d, m;
2  // thread T1
3  thread T1
4  x = 0;
5  lock(m);
6  if (x == y) {
7    unlock(m);
8    d = 3;
9  } else {
10   unlock(m);
11 }
12 /* end */
13
14 // Thread T2
15 thread T2
16 y = 1;
17 lock(m);
18 if (x <= y) {
19   unlock(m);
20   d = 2;
21 } else {
22   unlock(m);
23 }
24 /* end */
```

Listing 1. Two Simple Threads

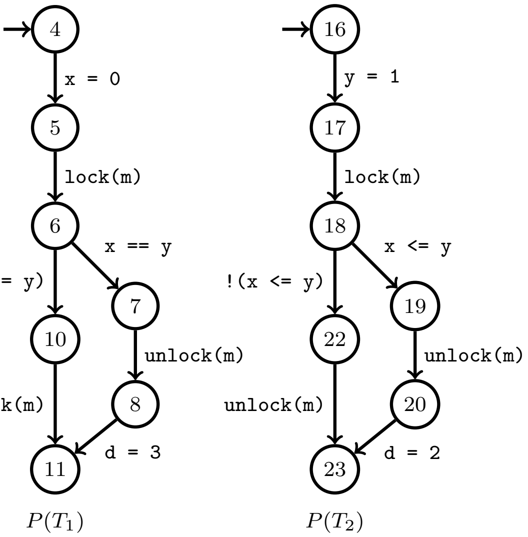


Fig. 2. Program Automata for T_1, T_2

2.2 2-Threaded Programs

A *program automaton* P is a tuple (L, ι, T) where: L is a finite set of program *locations*, $\iota \in L$ is the *initial location*, $T \subseteq L \times \Sigma \times L$ is *the control flow graph relation.* The set of *variables* of P is $\mathcal{V}(P) = \bigcup_{(\ell,i,\ell') \in T} \mathcal{V}(i)$.

A *state* of P is a pair $(\ell, \nu) \in L \times \mathbb{Z}^V$. Each program automaton induces a transition system $\mathcal{S}(P) = (L \times \mathbb{Z}^V, \{\iota\} \times \mathbb{Z}^V, \delta(P))$ with $\delta(P)$ defined by: $((\ell, \nu), i, (\ell', \nu')) \in \delta(P) \iff (\ell, i, \ell') \in T, (\nu, \nu') \in [\![i]\!]$.

A 2-*threaded program* is a pair (P_1, P_2) with $P_k = (L_k, \iota_k, T_k), k = 1, 2$ two program automata. The *shared variables* of (P_1, P_2) are $\mathcal{V}(P_1) \cap \mathcal{V}(P_2)$ and the other variables $\mathcal{V}(P_k) \backslash (\mathcal{V}(P_1) \cap \mathcal{V}(P_2))$ are the *local* variables for each $P_k, k = 1, 2$.

A 2-threaded program induces a transition system $\mathcal{S}(P_1, P_2) = (L_1 \times L_2 \times \mathbb{Z}^V, \{\iota_1\} \times \{\iota_2\} \times \mathbb{Z}^V, \delta(P_1, P_2))$ where $\delta(P_1, P_2)$ is the *interleaving* of $\delta(P_1)$ and

$\delta(P_2)$: $((\ell_1, \ell_2, \nu), i, (\ell'_1, \ell'_2, \nu')) \in \delta(P_1, P_2)$ iff either $((\ell_1, \nu), i, (\ell'_1, \nu')) \in \delta(P_1)$ and $\ell_2 = \ell'_2$ or $((\ell_2, \nu), i, (\ell'_2, \nu')) \in \delta(P_2)$ and $\ell_1 = \ell'_1$. A *state* of $\mathcal{S}(P_1, P_2)$ is a triple (ℓ_1, ℓ_2, ν) where $\ell_k \in L_k$ and ν is a valuation for V. Given a set of states $E \subseteq L_1 \times L_2 \times \mathbb{Z}^V$, the *reachability problem* asks whether $\textsc{Reach}(\mathcal{S}(P_1, P_2)) \cap E \neq \varnothing$ and $\textsc{Reach}(\mathcal{S}(P_1, P_2))$ may be infinite. For C-like 2-threaded programs this problem is undecidable as the reachability problem for single threaded programs with more than two variables (two-counter machines) is already undecidable. In the sequel, we thus consider semi-algorithms based on abstraction refinement to solve the reachability problem for multi-threaded programs.

Example 1. Listing 1, page 5, shows a 2-threaded C-like program. The program automata $P(T_1)$ and $P(T_2)$ for the threads `T1` and `T2` are given in Fig. 2. The `lock` and `unlock` C-like instructions are interpreted as *guarded* instructions the semantics of which is "When `(m == 0)` then `m = 1`" and the test and assignment happen in an atomic step. If a state of the form $(8, 20, \nu)$ is reachable in $\mathcal{S}(P(T_1), P(T_2))$ then the shared variable `d` can be written by the two threads. This is commonly referred to as a *data race.* $(P(T_1), P(T_2))$ is data race free (for variable `d`) iff $\textsc{Reach}(\mathcal{S}(P(T_1), P(T_2))) \cap \left(\{(8, 20)\} \times \mathbb{Z}^{\{x,y,d,m\}}\right) = \varnothing$. ■

Let $E \subseteq L_1 \times L_2$. We define the finite (product) automaton $P_1 \times P_2 = (L_1 \times L_2, \{(\iota_1, \iota_2)\}, T, E)$ with $T \subseteq (L_1 \times L_2) \times (L_1 \times L_2)$ defined by:

$$((\ell_1, \ell_2), i, (\ell'_1, \ell'_2)) \in T \iff (\ell_k, i, \ell'_k) \in T_k \textit{ for } k \in \{1, 2\} \textit{ and } \ell'_{3-k} = \ell_{3-k}.$$

E is the set of *accepting* states of $P_1 \times P_2$. $\mathcal{S}(P_1 \times P_2) = (L_1 \times L_2, (\iota_1, \iota_2), T)$ is a finite transition system. A *path* ϱ in $P_1 \times P_2$ is a path in $\mathcal{S}(P_1 \times P_2)$. The *language* $\mathcal{L}^E(P_1 \times P_2)$ is the set of traces $t \in \Sigma^*$ such that $(\iota_1, \iota_2) \xrightarrow{t} (\ell_1, \ell_2)$ for $(\ell_1, \ell_2) \in E$. When E is clear from the context we write $\mathcal{L}$ for $\mathcal{L}^E$.

From the definitions of $\mathcal{S}(P_1, P_2)$, the semantics of instructions $[\![\cdot]\!]$, the definition of the post operator for instructions, and the construction of the product of automata $P_1 \times P_2$, we straightforwardly get:

Fact 1 *$(\ell_0, \nu_0)\, i_0\, (\ell_1, \nu_1)\, i_1 \cdots i_{n-1}\, (\ell_n, \nu_n)$ with $(\nu_k, \nu_{k+1}) \in [\![i_k]\!], 0 \leq k < n$ is a path in $\mathcal{S}(P_1, P_2)$ if and only if $\ell_0\, i_0\, \ell_1 \cdots i_{n-1}\, \ell_n$ is a path in $P_1 \times P_2$ and $\mathsf{post}(i_0.i_1.\cdots.i_{n-1}, True) \not\subseteq False$.*

Given $E \subseteq L_1 \times L_2$, we use the shorthand $E \times \mathbb{Z}^V$ for $\{(\ell_1, \ell_2, \nu), (\ell_1, \ell_2) \in E, \nu \in \mathbb{Z}^V\}$. The following theorem is a direct consequence of Fact 1:

Theorem 1. *Let $E \subseteq L_1 \times L_2$. Then*

$$\textsc{Reach}(\mathcal{S}(P_1, P_2)) \cap (E \times \mathbb{Z}^V) \neq \varnothing \iff \exists t \in \mathcal{L}^E(P_1 \times P_2), \mathsf{post}(t, True) \not\subseteq False.$$

Remark 1. $\mathcal{S}(P_1, P_2)$ includes the data part of the program and can be infinite whereas $\mathcal{S}(P_1 \times P_2)$ includes only the locations of the CFG and is always finite.

3 Partial-Order Reduction

In this section we show that checking for the existence of a feasible trace in $\mathcal{L}(P_1 \times P_2)$ can be reduced to checking for the existence of a feasible trace in a *reduced* product automaton $(P_1 \times P_2)_R$. The reduced automaton $(P_1 \times P_2)_R$ is obtained by using standard partial order reduction algorithms that preserve properties of interest e.g., reachability of a location in a thread.

3.1 Independent Transitions

Partial order reduction techniques [5,6,7] were developed to address the state explosion problem in the analysis of concurrent systems.

These reductions rely on the notion of *dependency* and the complementary notion of *independency* between transitions. The intuition is that two *reads* on the same variable are independent whereas two *writes* or a *write* and a *read* to the same variable are dependent. For independent transitions the order of execution is irrelevant for certain properties and one *representative* order can be chosen to represent many interleavings.

Let $i, j \in \Sigma$ be two instructions. According to our definition, the same instruction can appear in two different threads or in the same thread. Instructions within the same sequential component are dependent and thus we have to differentiate these two cases. We assume that $\Sigma = \Sigma_1 \uplus \Sigma_2$, i.e., is partitioned into instructions for thread P_1 and P_2. i and j are *independent*, denoted $i \parallel j$, when $i \in \Sigma_k, j \in \Sigma_{3-k}$ for $k \in \{1, 2\}$ and $\mathcal{W}(i) \cap \mathcal{V}(j) = \varnothing$. By definition of the independency relation the following properties hold [5] for any $i \parallel j$ and any state s of $\mathcal{S}(P_1, P_2)$:

Enabledness if i is enabled in s and $s \xrightarrow{i} s'$, then j is enabled in s iff it is enabled in s' (independent transitions do not enable nor disable each other);

Commutativity if i and j are enabled in s, then $s \xrightarrow{i.j} s'$ and $s \xrightarrow{j.i} s'$, i.e., the order of i and j does not change the final target state (we assume here that the transition relations in P_1, P_2 are deterministic).

The independency relation induces a *trace equivalence* relation $\sim\ \subseteq \Sigma^* \times \Sigma^*$ which is the least congruence in the free monoid $(\Sigma^*, ., \varepsilon)$ that satisfies: $i \parallel j \implies i.j \sim j.i$. The equivalence classes of $\sim$ are called Mazurkiewicz traces.

Example 2. The instructions `x = 0` and `y = 1` in the automata of Fig. 2 are independent. Instructions `d = 3` and `d = 2` are not independent. ■

3.2 Selective Search Algorithm

We consider now a generic *selective search* algorithm SelectSearch. The purpose of such an algorithm is to explore a finite graph by avoiding to explore all the interleavings of $\sim$-equivalent sequences of transitions. We do not refer to a specific selective search algorithm but rather consider the minimum requirements needed to fit in our framework. Such a $\mathsf{SelectSearch}$ algorithm uses the independency relation $\|$ defined in the previous paragraph to prune out some edges (and states) during the exploration of the set of reachable states of $\mathcal{S}(P_1, P_2)$. If $\mathcal{S}(P_1, P_2)$ is finite, $\mathsf{SelectSearch}(\mathcal{S}(P_1, P_2))$ generates a finite transition system, called a *trace automaton* in [5]. We only require the $\mathsf{SelectSearch}$ algorithm to preserve the reachability of local states[4]. Assume $\mathcal{S}(P_1, P_2)$ is finite.

Theorem 2 (Theorem 6.14, [5]). *Let $\ell_1 \in L_1, \ell_2 \in L_2$ and $\nu \in \mathbb{Z}^V$. The state (ℓ_1, ℓ_2, ν) is reachable in $\mathcal{S}(P_1, P_2)$ iff there exists $\ell'_1 \in L_1$, $\nu' \in \mathbb{Z}^V$ such that state (ℓ'_1, ℓ_2, ν') is reachable in $\mathsf{SelectSearch}(\mathcal{S}(P_1, P_2))$.*

Actual implementations of the $\mathsf{SelectSearch}$ algorithms can be based on the selective search algorithm using *persistent sets, sleep sets and proviso* defined in [5, Figure 6.2, Chapter 6] or recent (optimal) algorithms as proposed in [19,20].

Let $\varrho = (\iota_1, \iota_2, \nu_0)\, i_0\, (\ell^1_1, \ell^2_1, \nu_1)\, i_1\, (\ell^1_2, \ell^2_2, \nu_2)\, i_2 \cdots i_{n-1}\, (\ell^1_n, \ell^2_n, \nu_n)$ be a path in $\mathsf{SelectSearch}(\mathcal{S}(P_1, P_2))$. $\mathsf{SelectSearch}$ is a *control location based* selective search if the set of transitions selected to be explored after ϱ only depends on the history $(\iota_1, \iota_2)(\ell^1_1, \ell^2_1) \cdots (\ell^1_n, \ell^2_n)$ of control locations of ϱ. Obtaining a location based selective search can be achieved by using a standard dependency relation as defined by $\|$ above. Notice that the selective search algorithm using *persistent sets, sleep sets and proviso* [5, Figure 6.2, Chapter 6] makes use of *conditional dependency relation* $\|_s$ that can vary according to the current state s and may depend on the value of ν in our 2-threaded programs. We disable this feature to obtain a control location based selective search algorithm and use an unconditional dependency relation like $\|$ above. This implies we may miss some pruning as our dependency relation is stronger than the conditional dependency one.

We now assume that the set of reachable states of $\mathcal{S}(P_1, P_2)$ may be infinite and show that Theorem 2 can be extended to infinite systems. Let $\mathsf{SelectSearch}$ be a control location based selective search algorithm. Let $P_1 \times P_2 = (L_1 \times L_2, (\iota_1, \iota_2), T, E)$ with $E = L_1 \times \{\ell_2\}$. Define $\mathsf{SelectSearch}(\mathcal{S}(P_1 \times P_2))$ to be the finite transition system obtained by applying $\mathsf{SelectSearch}$ on $\mathcal{S}(P_1 \times P_2)$. We can assume the selective search algorithm is Depth-First Search based and explores each state of $P_1 \times P_2$ at most once. We write $\mathcal{L}^E_R(P_1 \times P_2)$ for the set of traces $t \in \Sigma^*$ such that $(\iota_1, \iota_2) \xrightarrow{t} \ell$ in $\mathsf{SelectSearch}(\mathcal{S}(P_1 \times P_2))$ with $\ell \in E$. Using Theorem 2 we can prove the following:

Lemma 1. *$\exists t \in \mathcal{L}^E(P_1 \times P_2), \mathsf{post}(t, True) \not\subseteq False \iff \exists t' \in \mathcal{L}^E_R(P_1 \times P_2), \mathsf{post}(t', True) \not\subseteq False$.*

Proof. The *If* direction is easy as $\mathcal{L}^E_R(P_1 \times P_2) \subseteq \mathcal{L}^E(P_1 \times P_2)$.

To prove the *Only if* direction, let $t \in \mathcal{L}^E(P_1 \times P_2)$ and $\mathsf{post}(t, True) \not\subseteq False$. Let $n = |t|$. As $\mathsf{post}(t, True) \not\subseteq False$, there is some valuation $\nu_0 \in \mathbb{Z}^V$ such that $\mathsf{post}(t, \{\nu_0\}) \neq False$. Using Fact 1 (If direction), this implies that there exists a path of length n in $\mathcal{S}(P_1, P_2)$ that reaches a state (ℓ_1, ℓ_2, ν) from $q_0 = (\iota_1, \iota_2, \nu_0)$.

[4] We define it here for local states of P_2 but the property holds for each component of a multi-threaded program.

Let $\mathsf{Dag}_n(q_0)$ (we omit P_1, P_2 in the notation $\mathsf{Dag}_n()$ for clarity) be the Direct Acyclic Graph (DAG), that is obtained by building a depth-first search tree for $\mathcal{S}(P_1, P_2)$, from q_0, up to depth n. As this DAG is a finite transition system, and $E = L_1 \times \{\ell_2\}$, we can apply Theorem 2. This implies that a state (ℓ'_1, ℓ_2, ν') is reachable in $\mathsf{SelectSearch}(\mathsf{Dag}_n(q_0))$. Thus there exists a path $q_0\ i_0\ q_1\ \cdots\ i_{m-1}\ q_m$ in $\mathsf{SelectSearch}(\mathsf{Dag}_n(q_0))$ such that $q_m = (\ell'_1, \ell_2, \nu')$. This path is in $\mathcal{S}(P_1, P_2)$ and we can apply Fact 1 (Only If direction): there exists $t' = i_0.i_1.\cdots.i_m$ with $\mathsf{post}(t', True) \not\subseteq False$. It remains to prove that $t' \in \mathcal{L}^E_R(P_1 \times P_2)$. As $\mathsf{SelectSearch}$ is control location based, this follows directly from the fact that t' is a path in $\mathsf{SelectSearch}(\mathsf{Dag}_n(q_0))$. □

Lemma 1 together with Theorem 1 yield the following result:

Theorem 3. *Let* $E = L_1 \times \{\ell_2\}$. $\textsc{Reach}(\mathcal{S}(P_1, P_2)) \cap (E \times \mathbb{Z}^V) \neq \varnothing \iff \exists t \in \mathcal{L}^E_R(P_1 \times P_2), \mathsf{post}(t, True) \not\subseteq False.$

This reduces reachability of local states in the infinite system $\mathcal{S}(P_1, P_2)$ to the existence of a feasible trace in a finite reduced product $\mathsf{SelectSearch}(\mathcal{S}(P_1 \times P_2))$.

In the next section, we show how to use *trace refinement* to determine whether a feasible trace exists in $\mathcal{L}^E_R(P_1 \times P_2)$. In the sequel we assume E is fixed and omit it as a superscript.

4 Trace Abstraction Refinement for Concurrent Programs

In this section, we combine the trace refinement algorithm from [13] with partial order reduction. We first recall the trace abstraction refinement method and second present our algorithm that combines trace abstraction refinement and partial order reduction.

4.1 Interpolant Automata

The trace abstraction refinement algorithm from [13] relies on two keys conditions: a) given a sequence of instructions $t = i_0.i_1.\cdots.i_n \in \Sigma^*$, we can decide whether t is feasible (this can be done using SMT-solvers and decidable theories e.g., Linear Integer Arithmetic) and b) if t is infeasible there is an explanation in the form of an *inductive interpolant* i.e., a sequence of predicates $I_0, I_1, \cdots, I_{n+1}$ such that 1) $I_0 = True$ and $I_{n+1} = False$, 2) $\forall 0 \leq k \leq n, \mathsf{post}(i_k, I_k) \subseteq I_{k+1}$.

Let $P = (L, \iota, T)$ be a program automaton. Let $\varrho = \ell_0\ i_0\ \ell_1\ i_1\ \cdots i_{n-1}\ \ell_n$ be a path in P i.e., $\forall 0 \leq k \leq n-1, (\ell_k, i_k, \ell_{k+1}) \in T$. The trace of the path ϱ is $t = i_0.i_1.\cdots i_{n-1}$ and when the trace t is infeasible, the method introduced in [13] consists in building a finite automaton $\mathrm{IA}(t)$, called an *interpolant automaton* that accepts t *and many other traces that are infeasible for the same reason.*

A set of interpolant automata $\mathrm{IA}_1, \mathrm{IA}_2, \cdots, \mathrm{IA}_l$ each of which only accepts infeasible traces is a *refinement.* As regular languages are closed under union, it can actually be collapsed into one automaton that accepts $\cup_{k=1..l}\mathcal{L}(\mathrm{IA}_k)$.

We do not develop the theory of interpolant automata here and refer the reader to [13,14] for a more detailed explanation of the construction of $\mathrm{IA}(t)$.

4.2 Combining Trace Refinement and Partial Order Reduction

Using Theorem 3 we can design an iterative algorithm to check the control location reachability problem in 2-threaded programs. The input of the problem is (P_1, P_2) and a set of local states $E = L_1 \times \{\ell_2\}$. (P_1, P_2) is *safe* if $\textsc{Reach}(\mathcal{S}(P_1, P_2)) \cap (E \times \mathbb{Z}^V) = \varnothing$, otherwise it is unsafe. In the latter case we want the algorithm to return a witness trace t to reach a state in $E \times \mathbb{Z}^V$.

By Theorem 3, determining whether $\textsc{Reach}(\mathcal{S}(P_1, P_2)) \cap (E \times \mathbb{Z}^V) \neq \varnothing$ is equivalent to determining whether $\exists t \in \mathcal{L}_R^E(P_1 \times P_2)$ such that t is feasible. This can be done by adapting the generic iterative trace abstraction refinement algorithm of Fig. 1. The new algorithm that combines partial order reduction and trace refinement is given in Fig. 3. As pointed out in the previous paragraph, we can assume that a refinement is composed of one automaton A that accepts infeasible traces. Every time a new interpolant automaton $\mathrm{IA}(t)$ is obtained from an infeasible trace t we combine it with the previous refinement automaton A by computing $A \oplus \mathrm{IA}(t)$ where $A \oplus B$ denotes a finite automaton that accepts $\mathcal{L}(A) \cup \mathcal{L}(B)$. $\mathcal{L}_R(P_1 \times P_2)$ is a finite graph and can be viewed as an automaton with accepting locations in $L_1 \times \{\ell_2\}$. Checking emptiness of $\mathcal{L}_R(P_1 \times P_2) \cap \overline{\mathcal{L}(A)}$ reduces to a standard emptiness check in a synchronised product of automata. Notice that the reduce language $\mathcal{L}_R(P_1 \times P_2)$ should be constant during the iterative refinement. If we choose to generate different representatives of the same $\sim$-equivalent class at different iterations we may need to compute another interpolant automaton to reject the new representative. In our implementation we make sure that the same representative is generated at each refinement step. Ideally, we should compute the *closure* of an interpolant automaton under the equivalence relation $\sim$. This is one direction of future work to use *asynchronous automata* [31] to represent $\sim$-closures.

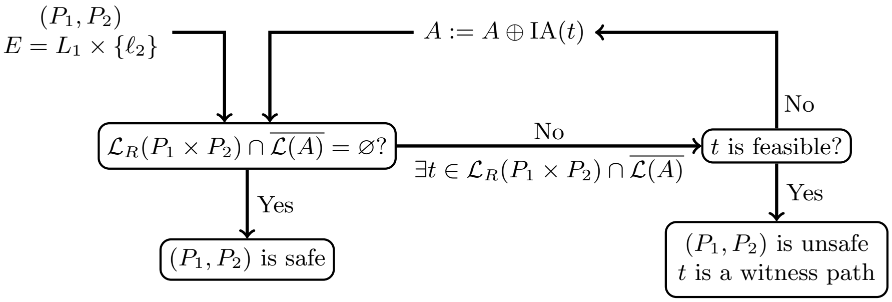

(P_1, P_2)
$E = L_1 \times \{\ell_2\}$
$A := A \oplus \mathrm{IA}(t)$
No
$\mathcal{L}_R(P_1 \times P_2) \cap \overline{\mathcal{L}(A)} = \varnothing$?
No
$\exists t \in \mathcal{L}_R(P_1 \times P_2) \cap \overline{\mathcal{L}(A)}$
t is feasible?
Yes
Yes
(P_1, P_2) is safe
(P_1, P_2) is unsafe
t is a witness path

Fig. 3. Trace Abstraction Refinement Algorithm

4.3 Beyond Reachability of Local States

Reachability of local states is general enough to encode reachability of *global* states which is needed to detect data races for instance. It suffices to add an extra component M, a *monitor*, and possibly extra shared variables. M has only one transition to a special location d and the condition to fire the transition is true iff there is a data race. Consider the 3-threaded program (P_1, P_2, M). There is a data race in $\mathcal{S}(P_1, P_2)$ iff a state (ℓ_1, ℓ_2, d, ν) is reachable in $\mathcal{S}(P_1, P_2, M)$ which is a local state reachability problem.

Deadlocks in $\mathcal{S}(P_1, P_2)$ can also be checked for if they are mapped to deadlock states in the product $P_1 \times P_2$ (independent of the data part of the system). This is usually the case, as deadlocks occur on lock/unlock operations or wait/signal that can be explicitly encoded in $\mathcal{S}(P_1 \times P_2)$. A deadlock in $\mathcal{S}(P_1, P_2)$ is then equivalent to the reachability of a global state.

Finally, a general reachability problem depending on data, e.g., specified by a statement of the form `assert(c)` in a multi-threaded program, can be checked using a monitor as well: the monitor has one transition to a location d, and the label of the edge to d is $\neg c$.

5 Implementation and Experiments

Implementation. We have implemented our combined partial order reduction and trace abstraction refinement algorithm in a prototype Raptor. The prototype is written is Scala and is comprised of: 1) a module to perform the partial order reduction implementing an algorithm based on [5, Figure 6.2, Chapter 6]; 2) a module to compute the automata accepting infeasible traces; this module uses a wrapper around SMTInterpol [21] to check feasibility of traces and get inductive interpolants when a trace is infeasible. The details of the algorithms and implementation are available in [18].

Our early prototype parses programs in a simple language of our own with integer variables and thread construct. Listing 1, page 5, is an example of such a program. As of now, it does not support C programs, arrays nor pointers yet. However, our language and product construction supports synchronisation and *wait/signal* primitives. We can then model *mutexes* directly as program automata. Another important implementation detail is that we do not compute $\overline{A}$ but rather determinise A *on-the-fly*.

Benchmarks. We used some examples from the Concurrency category [1] from the 4th Competition on Software Verification (SV-COMP 2015). They contain typical concurrent algorithms (Dekker, Lamport, Peterson) some of them coming in two flavours: *safe* and unsafe (column *Safe* in Table 1). Safety amounts to checking the reachability of a local state in one thread. We have translated (for now manually) the C programs of some of the benchmarks into our input language to analyse them. *LOC*, *#T* and *#V* contain respectively the number of lines of code in the source C program, the number of threads and the number of shared variables. Our simple language is very similar to C and the number of lines in the translated version is identical to the C version. *Red* gives the reduction in terms of explored states when partial order reduction is switched on compared to no reduction.

Program	Safe	Steps	States	Red.	LOC	#T	#V	Raptor	Mu-Cseq	Threader	Impara
stateful01	no	0	22	0%	34	3	6	1.1s/20	0.9s/1027	0.6s	N/A[5]
stateful01	yes	10	1628	17%	34	3	6	6.1s	TO	2.6s	N/A[5]
lazy01	no	1	11	0%	22	3	2	1.3s/9	0.6s/641	4.1s	0.16s
peterson	yes	29	1200	8%	31	2	4	5.7s	TO	4.6s	0.5s
dekker	yes	9	1276	7%	46	2	4	6.6s	TO	3.3s	0.7s
szymanski	yes	47	9811	13%	59	2	3	10s	TO	12s	1.43s
read_write_lock	no	11	2178	16%	65	4	5	6s/26	0.9s/992	55s	3.9s
read_write_lock	yes	38	10216	24%	63	4	5	9.5s	TO	57s	15s
time_var_mutex	yes	5	67	38%	33	2	5	0.69s	TO	4.9s	0.2s
fib_bench_false	no	284	10082	77%	25	3	2	29s/37	3.58s/949	TO	TO
ext-spin2003	yes	1	203	0%	44	4	2	3.4s	TO	176s	5.5s

Table 1. Raptor results on the SV-COMP benchmarks.

Comparison with Other Tools. We compared our results with one of the leadingtools in the category: Mu-Cseq [16] (silver medal at SV-COMP 2015), Threader 0.92 (winner of SV-COMP 2013 (winner of SV-COMP 2013), and Impara 0.2 [12]. We would have liked to compare with the winner tool lazy-CSeq [17] but the competition version is not available any more. We ran the analyses for Raptor and Mu-Cseq on an Virtual Linux Machine, running Ubuntu 13.10 64-bit on a MacBook Pro, Intel Core i5, 2.6GHz, 8GB of RAM.

Table 1 shows the results of Raptor on the selected benchmarks. Column *Steps* gives the number of refinement steps performed by Raptor. *States* is the (cumulative) total number of *explored states* in all the selective searches performed at each refinement step. The column Raptor contains the analysis time in seconds and, when the program is unsafe, the length of the counterexample (in terms of number of instructions in the program specified using our internal language). The column Mu-Cseq contains similar information for the tool Mu-Cseq [16]. For Threader and Impara we collected only the run times.[5] In table 1, "TO" stands for *TimeOut* with a time out bound set to 600s for Mu-Cseq. Notice that both Mu-Cseq and lazy-CSeq use a *bounded model checking* back-end (CBMC) and cannot formally establish the correctness of programs in most cases. In the SV-COMP 2015, Mu-Cseq and lazy-CSeq applied a strategy [16,17] to interpret the time out as "the program is safe". The result should rather be interpreted as "no bug was found" instead of a formal proof of correctness. In contrast, our method and tool can find bugs or establish program correctness (provided termination) even when programs have loops. This is a key feature of the trace abstraction refinement method that it can discover loop invariants and use them to reject infeasible traces.

On the negative instances (safe column is "no") we incur some overhead compared to Mu-Cseq to discover a counterexample. This can partially be explained by the fact that our Scala implementation is compiled into an executable Java jar file. Every analysis with Raptor thus needs to first spawn out the JVM. On the other hand our counterexamples are fairly short and easily mapped to the original programs. For the positive instances (safe column is "yes") we can *prove* all of them within seconds which is a clear advantage.

[5] In our experiments, Impara 0.2 failed to yield the correct analysis result.

The rough estimate of the distribution of the execution time for Raptor is 30% in the SMT-solver and the rest in the computation of the refinements and language inclusion check.

6 Related Work

Assume-guarantee reasoning for concurrent programs was first implemented in the Calvin model checker [22] to check concurrent Java programs. However, program refinement had to be done manually. To the best of our knowledge, the first paper to combine assume-guarantee reasoning with an automated abstraction refinement technique for multi-threaded programs is [10]. The proposed method is *modular* and can prove correctness for programs that admit a modular proof (the predicates on one thread never involve a local variable of another thread). As the authors point out (section 4 in [10]) "[. . .] the tool ignores thread interleavings [. . .] and may return false positives". Another limitation of [10] is intrinsic to the modular approach: not all multi-threaded programs admit modular proofs. This means that the algorithm may terminate because no better refinement can be derived and in this case might miss some bugs i.e., generate *false negatives*. This approach was later refined in [23]. Thread modular CEGAR was re-considered in [24] and compared against Spin. Threader [25,11] combines predicate abstraction and constraint solving but does not implement any symmetry or reduction techniques that deal with interleavings. Threader did not participate in the last two editions of SV-COMP 2014 and 2015. Recent work [26] by A. Miné introduced a big-step interference-based thread-modular static analysis as abstract interpretation based on assume-guarantee reasoning.

Partial order reduction techniques have long been recognised as effective for checking concurrent programs and tools implementing the techniques are numerous. The most-well known might be Spin [8] and VeriSoft [27]. Surprisingly enough, the combination of state-of-the-art predicate abstraction techniques like lazy abstraction (known as the Impact algorithm) with partial order reduction techniques has only been achieved recently in [28,12]. It turns out that obtaining a sound algorithm when combining lazy abstractions with partial order techniques is not trivial and it is not clear how other reduction techniques (e.g., symmetry reduction) can be accommodated for in these frameworks. Impara implements this technique and according to [12] outperforms all other tools. However, Impara did not participate in the SV-COMP 2015 and this is why we have not compared our results against it.

In [29,30], the authors address the verification problem for multi-threaded programs composed of threads executing the *same* procedure. They show how to derive constraints with *shared* and *local* variables, so-called *mixed predicates* but this comes at a rather expensive cost. Indeed, concurrency implies that the abstraction of a program is a concurrent boolean *broadcast* program for which the *image* computation of the *broadcast assignment* is expensive.

Other race checkers tools based on *lock-set* or *type-set* are limited in scope: they check for some conformance to generic patterns at the syntactic level when using mutexes (e.g., if a variable is used within a mutex m in one thread, it should be always used within the same mutex m in all other threads). On the programs we considered, they either report false positives or false negatives (do not detect the bug).

7 Conclusion and Ongoing Work

We have proposed a new method for verifying concurrent programs based on trace abstraction refinement and partial order reduction techniques. The results on some standard benchmarks from the *Software Verification Competition* (SV-COMP 2015) show that our approach compares favourably to existing techniques for finding bugs and is able to establish the correctness of the positive instances.

The combination we have proposed is very natural which is witnessed by the brevity of the correctness proofs (e.g., Lemma 1 and Theorem 3). This is in contrast with the combination of predicate abstraction refinement and partial order reduction [12] which is more involved. It is also clear that our approach extends to other reduction techniques, e.g., symmetry reduction.
Our current work is two-fold: *i*) on the theoretical side, we aim to compute refinements that are *asynchronous automata* [31] to capture infeasible traces and the all their equivalent traces. A second line of work is to design a modular algorithm (to avoid in-lining of function calls) in the spirit of our recent results [32]. *ii*) on the implementation side, we aim to add more capabilities to our tool, e.g., support for arrays, parsing for C programs and *iii*) implement recent *optimal* partial order reduction techniques [20,19].

References

1. Beyer, D.: International software verification competition http://sv-comp.sosy-lab.org/2015/.
2. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5) (2003) 752–794
3. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Commun. ACM **54**(7) (2011) 68–76
4. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4) (1983) 596–619
5. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem. Volume 1032 of LNCS. Springer (1996)
6. Peled, D.: All from one, one for all: on model checking using representatives. CAV. Volume 697 of LNCS., Springer (1993) 409–423
7. Valmari, A.: Stubborn sets for reduced state space generation. In: Applications and Theory of Petri Nets. Volume 483 of LNCS. (1989) 491–515
8. Holzmann, G.J.: Software model checking with spin. Advances in Computers **65** (2005) 78–109
9. Flanagan, C., Qadeer, S., Seshia, S.A.: A modular checker for multithreaded programs. CAV. Volume 2404 of LNCS., Springer (2002) 180–194
10. Henzinger, T.A., Jhala, R., Majumdar, R., Qadeer, S.: Thread-modular abstraction refinement. CAV. Volume 2725 of LNCS., Springer (2003) 262–274
11. Gupta, A., Popeea, C., Rybalchenko, A.: Predicate abstraction and refinement for verifying multi-threaded programs. POPL, ACM (2011), 331–344

12. Wachter, B., Kroening, D., Ouaknine, J.: Verifying multi-threaded software with impact. In: FMCAD, IEEE (2013) 210–217
13. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. SAS. Volume 5673 of LNCS., Springer (2009) 69–85
14. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. CAV. Volume 8044 of LNCS., Springer (2013) 36–52
15. Farzan, A., Kincaid, Z., Podelski, A.: Inductive data flow graphs. POPL, ACM (2013) 129–142
16. Tomasco, E. *et al.*: MU-CSeq 0.3: Sequentialization by Read-implicit and Coarse-grained Memory Unwindings. SV-COMP 2015 (TACAS), volume 9035 of LNCS., Springer (2015) 436–438.
17. Inverso, O. *et al.*: Lazy-CSeq 0.6c: An Improved Lazy Sequentialization Tool for C. SV-COMP 2015 (TACAS)..
18. Ziegler, F.: Verification of concurrent programs via partial-order reduction and trace refinement. MSc, Institut für Software & Systems Engineering, University of Ausgburg, Germany (2014)
19. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.F.: Optimal dynamic partial order reduction. POPL, ACM (2014) 373–384
20. Kahlon, V., Wang, C., Gupta, A.: Monotonic partial order reduction: An optimal symbolic partial order reduction technique. CAV. Volume 5643 of LNCS., Springer (2009) 398–413
21. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An Interpolating SMT Solver. SPIN 2012. Volume 7385 of LNCS., Springer (2012) 248–254
22. Flanagan, C., Qadeer, S.: Thread-modular model checking. SPIN 2003. Volume 2648 of LNCS., Springer (2003) 213–224
23. Henzinger, T.A., Jhala, R., Majumdar, R.: Race checking by context inference. PLDI 2004, ACM (2004) 1–13
24. Malkis, A., Podelski, A., Rybalchenko, A.: Thread-modular counterexample-guided abstraction refinement. SAS. Volume 6337 of LNCS., Springer (2010) 356–372
25. Gupta, A., Popeea, C., Rybalchenko, A.: Threader: A constraint-based verifier for multi-threaded programs. [34] 412–417
26. Miné, A.: Static analysis by abstract interpretation of concurrent programs. Habilitation à Diriger les Recherches, ENS, France (2013)
27. Godefroid, P.: Software model checking: The verisoft approach. Formal Methods in System Design **26**(2) (2005) 77–101
28. Cimatti, A., Narasamdya, I., Roveri, M.: Boosting lazy abstraction for systemc with partial order reduction. TACAS. Volume 6605 of LNCS., Springer (2011) 341–356
29. Donaldson, A.F., Kaiser, A., Kroening, D., Wahl, T.: Symmetry-aware predicate abstraction for shared-variable concurrent programs. [34] 356–371
30. Donaldson, A.F., Kaiser, A., Kroening, D., Tautschnig, M., Wahl, T.: Counterexample-guided abstraction refinement for symmetric concurrent programs. Formal Methods in System Design **41**(1) (2012) 25–44
31. Zielonka, W.: Notes on finite asynchronous automata. ITA **21**(2) (1987) 99–135
32. Cassez, F., Müller, C., Burnett, K.: Summary-based inter-procedural analysis via modular trace refinement. FSTTCS 2014, 545–556
33. Cassez, F., Pierzchalski, E., Smyth, N.: Perentie: Trace abstraction refinement and value tracking. SV-COMP 2015 (TACAS), volume 9035 of LNCS., Springer (2015) 439–442.
34. Gopalakrishnan, G., Qadeer, S., eds.: CAV. Volume 6806 of LNCS., Springer (2011)