Verifying Multi-threaded Software with Impact

Björn Wachter
Department of Computer Science
University of Oxford
Email: bjoern.wachter@cs.ox.ac.uk

Daniel Kroening
Department of Computer Science
University of Oxford
Email: daniel.kroening@cs.ox.ac.uk

Joël Ouaknine
Department of Computer Science
University of Oxford
Email: joel.ouaknine@cs.ox.ac.uk

***Abstract*—Lazy abstraction with interpolants, also known as the *Impact algorithm*, is en vogue as a state-of-the-art software model-checking technique for sequential programs. However, a direct extension of the Impact algorithm to concurrent programs is bound to be inefficient as it has to explore all thread interleavings, which leads to control-state explosion. To this end, we present a new algorithm that combines a new, symbolic form of partial-order reduction with Impact. Our algorithm carries out the dependence analysis on-the-fly while constructing the abstraction and is thus able to deal precisely with dynamic dependencies arising from accesses to tables or pointers — a setting where classical static partial-order reduction techniques struggle. We have implemented the algorithm in a prototype tool that analyses concurrent C program with POSIX threads and evaluated it on a number of benchmark programs. To our knowledge, this is the first application of an Impact-like algorithm to concurrent programs.**

I. Introduction

Concurrent software is gaining importance owing to the advent of power-efficient multi-core architectures. Model checking for concurrent software is thus one of the most pressing problems facing the verification community. Concurrent software in C/C++ is usually written using mainstream APIs such as POSIX, or via a combination of language and library support as in Java. Typically, multiple threads are spawned—either up-front or dynamically—which communicate via shared variables. While software verification generally has to cope with *data state explosion*, threads introduce the problem of state explosion due to the need of keeping track of a plethora of thread interleavings.

Lazy abstraction with interpolants [1], also known as the *Impact algorithm*, has emerged as one of the most efficient algorithms for addressing the data state explosion problem for sequential programs. *Impact* unwinds the control-flow graph of the program in the form of an abstract reachability tree. Whenever the exploration arrives at an error state, the nodes on the error path are annotated with invariants that prove infeasibility of the error path. The crux of the algorithm is a covering check that allows the algorithm to soundly stop the unwinding and terminate with a correctness proof of the program. The underlying observation is that tree nodes represent sets of program states which are related by subset relations. Roughly, a node w labeled with $x > 0$ "contains" a node v labeled with $x > 1$. If we have established that the superset

Supported by ERC project 280053, EPSRC project EP/H017585/1 and the Semiconductor Research Corporation (SRC) under task 2269.002.

`main()`	thread T_1	thread T_2
`assume(i!=j);`		
`v[i]=0; v[j]=0;`	A: `v[i]=1;`	a: `v[j]=-2;`
`pthread_create(`T_1`);`	B: `v[i]=v[i]+1;`	b: `v[j]=v[j]+1;`
`pthread_create(`T_2`);`	C: `v[i]=v[j];`	c: `v[i]=v[i]+1;`
`pthread_join(`T_1`);`		
`pthread_join(`T_2`);`		
`assert(v[j]` $\geq$ `0);`		

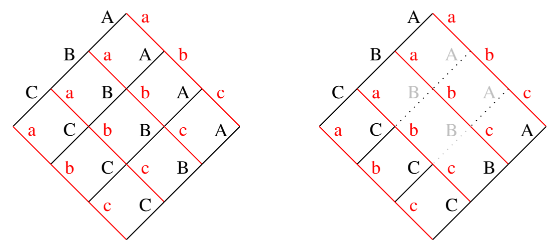


Fig. 1: An example program (top) and its complete interleaving (left) and reduced interleaving semantics (right).

node w cannot be on an error path, we do not need to search for an error path from subset node v. This combination of low-cost program unwindings combined with path-based refinement and covering checks gives rise to an efficient software model checking algorithm.

However, the original *Impact* algorithm has been devised for sequential code only. A direct extension of *Impact* to multi-threaded programs amounts to an enumeration of thread interleavings. Let us illustrate this with the example program with two threads given in Figure 1. On the left-hand side of the figure, the state graph with the complete set of interleavings is shown. Note that there is a diamond-shaped structure where program paths merge, e.g., executing instruction A and then a leads to the same state as executing a first and then A, making certain sequences of instructions redundant. This situation is very common in multi-threaded programs.

Impact produces the full program unwinding, as the exploration of the abstract tree has to reach an error location to discover the right invariants. The algorithm may find identical invariants for redundant paths, but this does not prune the abstract exploration, as, at that point, the program paths have already been completely unwound.

Force cover, an optimization of *Impact*, improves this situation by giving *Impact* the power to discover that certain program executions merge without fully exploring the paths to the error location. This reduces the number of paths to be explored. On our example, the application of force covers results in a tree of a similar size as the graph on the left-hand side of Figure 1. In particular, even with force covers, *Impact* still explores *all* thread interleavings in our example, which can be prohibitively expensive.

A principal method to reduce the number of interleavings in the exploration of concurrent programs is *partial-order reduction* (POR) [2]–[5]. The right-hand side of Figure 1 shows an exploration reduced by means of partial-order reduction. A key contribution of this paper is a novel combination of *Impact* with POR, which produces the abstract tree shown in Figure 2. *Impact* with force cover alone explores a tree with five further nodes, as it does not know in advance that the executions merge, while partial-order reduction is able to discover this earlier. Discovering redundant paths early on during the exploration is crucial to avoid path explosion.

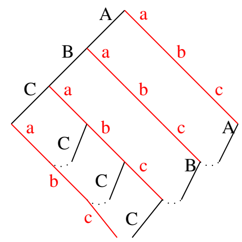


Fig. 2: *Impact* with POR and force cover

Contributions:

- We present an extension of the Impact algorithm to concurrent software.
- We show how to combine partial-order reduction with Impact. Due to a subtle interplay between node coverings and POR, obtaining a sound verification procedure is non-trivial. To this end, we give a general framework to prove such combinations correct, and an algorithm based on this framework which combines Impact with monotonic partial-order reduction [6].
- We compare the effect of partial-order reduction and force covers; our conjecture is that the two techniques yield orthogonal benefits and are best combined.

We present background and basic definitions in Section II. We develop a variant of the *Impact* algorithm for concurrent software in Section III. We present a combination of partial-order reduction and *Impact* in Section IV. Experimental results are discussed in Section V.

II. Basic Definitions

Program semantics: We consider a concurrent program $\mathscr{P}$ composed of a finite set of threads $\mathscr{T}$, which communicate by performing operations on shared variables.

A state of a concurrent system consists of the local states S_{local} of each thread, i.e., the value of the thread's program counter given by a program location $l \in L$ and values of the local variables of the thread, and of the shared states S_{shared}, i.e., values for communication objects such as locks, tables and the like. Thus, we have a global state space $S = S_{shared} \times S_{local}$.

A global control location is a function $\mathfrak{l} : \mathscr{T} \to L$ from threads to control locations. Let L_G be the set of global control locations. The global location in state s is denoted by $\mathfrak{l}(s)$. For a given global location $\mathfrak{l}$ and thread T, we write $\mathfrak{l}_T$ as a shorthand for $\mathfrak{l}(T)$. By $\mathfrak{l}[T \mapsto l]$, we denote the global location where the location of thread T maps to l, while the locations for all other threads T' remain unchanged.

We characterize program data in terms of formulas in standard first-order logic. We denote the set of well-formed formulas over symbols Σ by $\mathscr{F}(\Sigma)$. For a given formula F we denote the set of formulas over the same symbols by $\mathscr{F}(F)$.

Let V be the vocabulary that represents the program variables. A state formula is a formula in $\mathscr{F}(V)$ and represents a set of global states. A transition formula, from now on, typically denoted by the letter R, is a formula in $\mathscr{F}(V \cup V')$.

Formally, we model a program as a pair $(init, \mathscr{T})$ where $init \in \mathscr{F}(V)$ is the initial-state predicate, and $\mathscr{T}$ a finite set of threads. We assume that the set of threads is endowed with some total order $<$. A thread $T \in \mathscr{T}$ is a tuple $T = (L, l^i, l^{\lightning}, A)$ consisting of a finite set of control locations L, an initial location $l^i \in L$, an error location $l^{\lightning}$, and a set of actions A. An action is a pair $a = (l, N) \in L \times 2^{\mathscr{F}(V \cup V') \times L}$, consisting of a current location l and a set of successor control locations l', each associated with a transition constraint. An assignment l_1`: x=y+1;` l_2`:` ... is represented as $(l_1, \{(x' = y + 1 \wedge y' = y; l_2)\})$. An assertion l_1`: assert(x<y);` l_2`:` ... becomes an action $(l_1, \{(x \geq y \wedge x' = x \wedge y' = y, l^{\lightning}), (x < y \wedge x' = x \wedge y' = y, l_2)\})$, which enters the error location $l^{\lightning}$ if the condition is violated. Sets of successors are used to represent branching control flow, e.g., the encoding of the `if`-statement l_1`: if(x==1) goto` l_3`;` l_2`:` ... is $(l_1, \{(x = 1 \wedge x' = x, l_3), (x \neq 1 \wedge x' = x, l_2)\})$.

We write $L(T)$ and $A(T)$ to denote the locations and actions of a thread. For an action $a = (l, N) \in A(T)$ of thread T, action a is enabled at location l and at global location $\mathfrak{l}$ if a is enabled at $\mathfrak{l}_T$. We assume that exactly one action $a_{T,l}$ of any given T is enabled at any location $l \in L$.

The control-flow graph $CFG_T = (l^i, E)$ of thread $T = (L, l^i, l^{\lightning}, A)$ is defined by entry node l^i and edges $E = \bigcup_{a \in A(T)} E_a$ where $E_a = \{(l, l') \in L \times L \mid a = (l, N), (R, l') \in N\}$. The control-flow nodes are topologically ordered. We say that an action a induces a back edge if E_a contains a back edge.

We say that an action $a = (l, N) \in T$ is enabled at a state if a is enabled at global location $\mathfrak{l}(s)$. We denote the enabled actions at a state s by *enabled*(s). We assume that an action $a = (l, N)$ defines a total function $\{s \in S \mid a \in enabled(s)\} \to S$ on all program states for which it is enabled.

For ease of notation, we identify a with this function and write $a(s)$ to denote the successor of a state s under action a.

Invariants and Correctness Proofs: A program path π is a sequence $(\mathfrak{l}_0, T_0, a_0, \mathfrak{l}_1) \ldots (\mathfrak{l}_{N-1}, T_{N-1}, a_{N-1}, \mathfrak{l}_N)$. For a thread T, and $l, l' \in L(T)$ with $l \neq l'$, we write $l \sqsubset l'$ if there exists a program path from l to l'.

A path is an *error path* if $\mathfrak{l}_0$ is the vector of initial locations for all threads, and $\mathfrak{l}_{N-1}$ contains an error location of a thread. We denote by $\mathscr{F}(\pi)$ the sequence of formulas $init^{(0)} \wedge R_0^{(0)}, \ldots R_{N-1}^{(N-1)}$ obtained by shifting each R_i i time frames into the future. We say that π is feasible if $\bigwedge R_i^{(i)}$ is logically satisfiable. A solution to $\bigwedge R_i^{(i)}$ corresponds to a program execution assigning values to the program variables at each execution step. The program is said to be safe if all error paths are infeasible.

An *inductive invariant* is a mapping $I : L_G \to \mathscr{F}(V)$ such that $init \Rightarrow I(\mathfrak{l}^i)$ and for all locations $\mathfrak{l} \in L_G$, all threads $T \in \mathscr{T}$, and actions $a = (l, R, l') \in T$ in thread T enabled in $\mathfrak{l}$, we have $I(\mathfrak{l}) \wedge R \Rightarrow I(\mathfrak{l}[T \mapsto l'])$. A *safety invariant* is an inductive invariant with $I(\mathfrak{l}) \equiv False$ for all error locations $\mathfrak{l}$. If there is a safety invariant the program is safe.

Interpolants: In case a path is infeasible, an explanation can be extracted in the form of an interpolant. To this end, we define *sequent interpolants* [7]. A sequent interpolant for formulas $A_1, \ldots, A_N$, is a sequence $\widehat{A}_1, \ldots, \widehat{A}_N$ where the first formula is equivalent to true $\widehat{A}_1 \equiv True$, the last formula is equivalent to false $\widehat{A}_N \equiv False$, consecutive formulas imply each other, i.e., for all $i \in \{1, \ldots, N\}$, $\widehat{A}_{i-1} \wedge A_i \Rightarrow \widehat{A}_i$, and, the i-th sequent is a formula over the common symbols of its prefix and postfix, i.e., for all $i \in \{1, \ldots N\}$, $\widehat{A}_i \in \mathscr{F}(A_1, \ldots, A_i) \cap \mathscr{F}(A_{i+1}, \ldots, A_N)$. For certain theories, quantifier-free interpolants can be generated for inconsistent, quantifier-free sequences $A_1, \ldots, A_N$ [7].

III. Impact Algorithm for Concurrent Programs

We now present an extension of the original Impact algorithm to concurrent programs. The algorithm returns either a safety invariant for a given program, finds a counterexample or diverges (the verification problem is undecidable). To this end, the algorithm constructs an abstraction of the program in the form of an abstract reachability tree, which corresponds to a program unwinding annotated with invariants.

Definition 3.1 (ART): An *abstract reachability tree* (ART) $\mathscr{A}$ for program $\mathscr{P}$ is a tuple $(V, \epsilon, \rightarrow, \rhd)$ consisting of a tree with nodes V, root node $\epsilon \in V$, edges $\rightarrow \subseteq V^2$, and a covering relation $\rhd \subseteq V^2$ between tree nodes such that:

- every nodes $v \in V$ is labeled with a tuple $(\mathfrak{l}, \phi)$ consisting of a current global control location $\mathfrak{l}$, and a state formula ϕ. We write $\mathfrak{l}(v)$ and $\phi(v)$ to denote the control location and annotation, respectively, of node v.
- edges correspond to program actions, and tree branching represents both branching in the control flow within a thread and thread interleaving. Formally, an edge is a tuple (v, T, R, w) where $v, w \in V$, $T \in \mathscr{T}$, and R the transition constraint of the corresponding action.

We write $v \xrightarrow{T} w$ if there exists an edge $(v, T, R, w) \in \rightarrow$. We denote by $\rightsquigarrow$ the transitive closure of $\rightarrow$.

To put abstract reachability trees to work for proving program correctness for unbounded executions, we need a criterion to prune the tree without missing any error paths. This role is assumed by the covering relation $\rhd$.

Intuitively, the purpose of node labels is to represent inductive invariants, i.e., over-approximations of sets of states, and the covering relation is the equivalent of a subset relation between nodes. Suppose that two nodes v, w share the same control location, and $\phi(v)$ implies $\phi(w)$. If there was a feasible error path from v, there would be a feasible error path from w. Therefore, if we can find a safety invariant for w, we do not need to explore successors of v, as $\phi(v)$ is at least as strong as the already sufficient invariant $\phi(w)$.

Note that, therefore, if w is safe, all nodes in the subtree rooted in v are safe as well. Therefore, a node is covered if and only if the node itself or any of its ancestors has a label implied by another node's label at the same control location.

To obtain a proof from an ART, the ART needs to fulfill certain conditions, summarized in the following definition:

Definition 3.2 (Safe ART): Let $\mathscr{A} = (V, \epsilon, \rightarrow, \rhd)$ be an ART.

- $\mathscr{A}$ is *well-labeled* if the labeling is inductive, i.e., $\forall (v, T, R, w) \in \rightarrow$: $\mathfrak{l}(v) = \mathfrak{l}(w) \wedge \phi(v) \wedge R \Rightarrow \phi(w)'$ and compatible with covering, i.e., $(v, w) \in \rhd : \phi(v) \Rightarrow \phi(w)$ and w not covered.
- $\mathscr{A}$ is *complete* if all of its nodes are covered, or have an out-going edge for every action that is enabled at $\mathfrak{l}$.
- $\mathscr{A}$ is *safe* if all error nodes are labeled with *False*.

Theorem 3.3: If there is a safe, complete, well-labeled ART of program $\mathscr{P}$, the program is safe.

Proof As in [1], the labeling immediately gives a safety invariant M, $M(\mathfrak{l}') = \bigvee \{\phi(v) \mid \mathfrak{l}(v) = \mathfrak{l}'\}$. ∎

A. Concurrent Impact with Full Interleaving

The concurrent version of the Impact algorithm we describe next (Algorithm 1) constructs an ART by alternating three different operation on nodes: Expand, Refine, and Close. At all times, the algorithm maintains the invariant that the tree is well-labeled and safe, i.e., to produce a correctness proof the algorithm needs to make the tree complete.

To keep track of nodes where the tree is incomplete, uncovered leaf nodes are kept in a work list Q.

Expand takes an uncovered leaf node and computes its successors. To this end, it iterates over all threads. For every enabled action, it creates a fresh tree node w, and sets its location to the control successor l' given by the action. To ensure that the labeling is inductive, the formula $\phi(w)$ is set to *True*. Then the new node is added to the work list Q. Finally, a tree edge is added (Line 23), which records the step from v to w and the transition formula R. Note that if w is an error location, the labeling is not safe; in which case, we need to refine the labeling, invoking operation Refine.

Refine takes an error node v and, detects if the error path is feasible and, if not, restores a safe tree labeling. First, it determines if the unique path π from the initial node to v is feasible by checking satisfiability of $\mathscr{F}(\pi)$. If $\mathscr{F}(\pi)$ is satisfiable, the solution gives a counterexample in the form of a concrete error trace, showing that the program is unsafe. Otherwise, an interpolant is obtained, which is used to refine the labeling. Note that strengthening the labeling may destroy the well-labeledness of the ART. To recover it, pairs $w \rhd v_i$ for strengthened nodes v_i are deleted from the relation, and the node w is put into the work list again.

Close takes a node v and checks if v can be added to the covering relation. As potential candidates for pairs $v \rhd w$, it only considers nodes created before v, denoted by the set $V^{<v} \subsetneq V$. This is to ensure stable behavior, as covering in arbitrary order may uncover other nodes, which may not terminate. Thus only for uncovered nodes $w \in V^{<v}$, it is checked if $\mathfrak{l}(w) = \mathfrak{l}(v)$ and $\phi(v)$ implies $\phi(w)$. If so, (v, w) is added to the covering

Algorithm 1 Impact with support for concurrent programs

procedure MAIN()
  $Q := \{\epsilon\}$, $\rhd := \emptyset$
  while $Q \neq \emptyset$ **do**
    select and remove v from Q
    CLOSE(v)
    if v not covered **then**
      if $error(v)$ **then**
        REFINE(v)
      EXPAND(v)
  return $\mathscr{P}$ is safe

procedure EXPAND(v)
  for $T \in \mathscr{T}$ **do**
    EXPAND-THREAD(T, v)

procedure EXPAND-THREAD(T, v)
  $(\mathfrak{l}, \phi) := v$
  for $(l, N) \in A(T)$ with $\mathfrak{l}_T = l$ **do**
    for $(R, l') \in N$ **do**
      $w :=$ fresh node
      $\mathfrak{l}(w) := \mathfrak{l}[T \mapsto l']$
      $\phi(w) := True$
      $Q := Q \cup \{w\}$, $V := V \cup \{w\}$
      $\rightarrow := \rightarrow \cup \{(v, T, R, w)\}$

procedure CLOSE(v)
  for $w \in V^{\prec v}$: w uncovered **do**
    if $\mathfrak{l}(v) = \mathfrak{l}(w) \wedge \phi(v) \Rightarrow \phi(w)$ **then**
      $\rhd := \rhd \cup \{(v, w)\}$
      $\rhd := \rhd \setminus \{(x, y) \in \rhd \mid v \rightsquigarrow y\}$

procedure REFINE(v)
  if v not error node or $\phi(v) \equiv False$ **then**
    return
  $\pi := v_0, \dots v_N$ path from ϵ to v
  if $\mathscr{F}(\pi)$ has interpolant $A_0 \dots A_N$ **then**
    for $i = 0 \dots N$ **do**
      $\phi := A_i^{-i}$
      if $\phi(v_i) \not\models \phi$ **then**
        $Q := Q \cup \{w \mid w \rhd v_i\}$
        $\rhd := \rhd \setminus \{(w, v_i) \mid w \rhd v_i\}$
        $\phi(v_i) := \phi(v_i) \wedge \phi$
    for $w \in V$ s.t. $w \rightsquigarrow v$ **do**
      CLOSE(w)
  else
    abort (program unsafe)

relation $\rhd$. To restore well-labeling, all pairs (x, y) where y is a descendant of v, denoted by $v \rightsquigarrow y$, are removed from $\rhd$, as v and all its descendants are covered.

MAIN first initializes the queue with the initial node ϵ, and the relation $\rhd$ with the empty set. It then runs the main loop of the algorithm until Q is empty, i.e., until the ART is complete, unless an error is found which exits the loop. In the main loop, a node is selected from Q. First, CLOSE is called to try and cover it. If the node is not covered and it is an error node, REFINE is called. Finally, the node is expanded, unless it was covered, and evicted from the work list.

An important optimization of the algorithm is another subroutine, called *force cover*. Initially, all new nodes are labeled with invariant *True*. Therefore, they will not be covered by an existing node with a non-trivial invariant, although this may be a permissible labeling. To check coverage, force cover finds the nearest common ancestor of two nodes and then checks the characteristic formula to the new node to see if the invariant of the other node also holds at the new node. Beyer [8] showed that this optimization is essential for the performance of *Impact*.

Wrapping up the extension of the original Impact algorithm to concurrent programs: the single control location becomes a vector, and the EXPAND routine enumerates all possible interleavings. This algorithm is very inefficient in its basic form: due to the full interleaving semantics, the number of global control locations grows very quickly. We shall amend this in the next section.

IV. Partial Order Reduction

Performing a thread interleaving at every step would be prohibitively expensive. *Impact* needs some way of reducing interleaving. Therefore, we present an algorithm that combines partial-order reduction with the *Impact* algorithm. A very simple kind of partial order reduction is to only allow interleaving when shared-variable accesses occur, however a much stronger reduction is possible in many cases. In this section, we consider a more advanced partial exploration strategy that generates monotonic program paths Π_{mono}, wherein consecutive independent actions only occur in the order of increasing thread ids [6].

Recall that the soundness proof of the original IMPACT algorithm rests on three pillars, namely: completeness, safety and well-labeledness of ARTs. However, partial order reduction clashes with the original completeness criterion of IMPACT that requires the very thing we aim to avoid: full expansion of all thread interleavings. Thus we need a new soundness proof and, in particular, a weaker completeness criterion, to combine abstraction with partial-order reduction.

To this end, we introduce the new concept of Π-*completeness*, which is parameterized with an exploration strategy via a set of program paths Π, and gives a systematic framework to combine abstraction with partial-order reduction. Based on this concept, we also present the dPOR-IMPACT algorithm, which explores monotonic paths and produces Π_{mono}-complete ARTs.

Before we come to Π-completeness and dPOR-IMPACT, we first need to review some basic POR concepts and notation.

A. Independence and Mazurkiewicz Equivalence

Partial-order reduction is based on the notion of independence of actions. Intuitively, two actions are independent if they commute and we can execute them in any order:

Definition 4.1 (Independence): Two actions a_1 and a_2 are *independent*, denoted by $a_1 \mid\mid a_2$, if for all states $s \in S$ where a_1 and a_2 are co-enabled, i.e., $a_1, a_2 \in enabled(\mathfrak{l}(s))$, we have $a_1(a_2(s)) = a_2(a_1(s))$). Otherwise, we say that they are dependent and write $a_1 \not\parallel a_2$.

Partial-order reduction techniques are based on finding a representative subset of the interleavings avoiding the exploration of all equivalent interleavings, i.e., interleavings that lead to equivalent orderings of actions. This leads to the notion of Mazurkiewicz equivalence [9]:

Definition 4.2 (Mazurkiewicz equivalence): Two program paths are *Mazurkiewicz equivalent* if they result from exchanging the order of two independent actions.
We call a set of program paths Π *representative* if it contains a representative path for every Mazurkiewicz equivalence class.

An example for a representative set of program paths are the monotonic program paths, which are defined as follows:

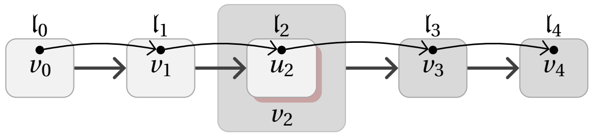


Fig. 3: Path correspondence. Rounded rectangles represent ART nodes $v_0,\ldots,v_4$ and u_2. We have $u_2 \rhd v_2$. The gray arrows depict ART edges. The path $l_0 \ldots l_5$ is a control flow path.

Definition 4.3 (Monotonic paths): A program path $\pi = (\ell_0, T_0, a_0, \ell_1)\ldots(\ell_{N-1}, T_{N-1}, a_{N-1}, \ell_N)$ is *monotonic* if for all $i, j \in \{0,\ldots,N-1\}$ with $i < j$, $a_i \parallel a_j$ and $T_i > T_j$, we have $j \neq i+1$. Let Π_{mono} be the set of monotonic program paths.

B. Π-*completeness*

We will say that an ART $\mathscr{A}$ is Π-*complete* with respect to a set of program paths Π if each path $\pi \in \Pi$ is covered by $\mathscr{A}$. Intuitively, a program path is covered if there exists a corresponding sequence of nodes in the tree, where corresponding means that it visits the same control locations and takes the same actions. In absence of covers, the matching between control paths and sequences of nodes is straightforward.

However, a path of the ART may end in a covered node. For example, consider the path $\ell_0 \ldots \ell_5$ in Figure 3. While prefix $\ell_0\ell_1\ell_2$ can be matched by node sequence $v_0 v_1 u_2$, node u_2 is covered by node v_2, formally $u_2 \rhd v_2$. But how we can match the remainder of the path? We are stuck at node u_2, a leaf with no out-going edges. Our solution is to allow the corresponding sequence to "climb up" the covering order $\rhd$ to a more abstract node, here we climb from u_2 to v_2. Node v_2 in turn must have a corresponding out-going edge, as it cannot be covered and its control location is also ℓ_2. Finally, the corresponding node sequence for $\ell_0 \ldots \ell_4$ is $v_0 \ldots v_4$.

Figure 4 illustrates the formalization of our notion of path correspondence. On top of the figure, we depict a fragment of a program path with locations l_i, l_{i+1} and l_{i+2}, and, at the bottom, the corresponding path which climbs from node u_{i+1} to node v_{i+1} where u_{i+1} and v_{i+1} are both at location $\ell(u_{i+1}) = \ell(v_{i+1}) = \ell_{i+1}$ and $u_{i+1} \rhd v_{i+1}$. A corresponding path is allowed to climb up not only at one position i but at any position i (or none) and at arbitrarily many positions.

This notion is formalized in the following definition:

Definition 4.4 (Corresponding paths & path cover): Consider a program $\mathscr{P}$. Let $\mathscr{A}$ be an ART for $\mathscr{P}$ and let $\pi = (\ell_0, T_0, a_0, \ell_1)\ldots(\ell_{N-1}, T_{N-1}, a_{N-1}, \ell_N)$ be a program path. A *corresponding path* for π in $\mathscr{A}$ is a sequence $v_0,\ldots,v_n$ in $\mathscr{A}$ such that, for all $i \in \{0,\ldots,N-1\}$, $\ell(v_i) = \ell_i$, and

$$\exists u_{i+1} \in V:\ v_i \xrightarrow{T_i, a_i} u_{i+1} \wedge (u_{i+1} = v_{i+1} \vee u_{i+1} \rhd v_{i+1})$$

A program path π is covered by $\mathscr{A}$ if there exists a corresponding path $v_0,\ldots,v_n$ in $\mathscr{A}$.

We are now ready to define our new completeness criterion:

Definition 4.5 (Π-completeness): Let $\mathscr{P}$ be a program and Π a set of program paths. ART $\mathscr{A}$ for $\mathscr{P}$ is Π-complete if every path $\pi \in \Pi$ is covered by $\mathscr{A}$.

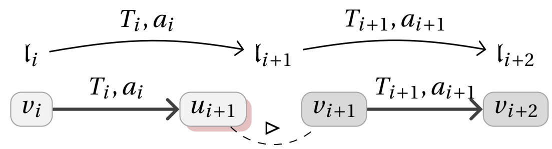


Fig. 4: Illustration of Definition 4.4. The diagram shows a fragment of an ART $\mathscr{A}$ with notation for the nodes of the definition. The dashed line represents a covering edge.

A Π-complete, safe, well-labeled ART constitutes a proof of program correctness, as stated in the following proposition:

Proposition 4.6: Let $\mathscr{P}$ be a program. Let Π be a representative set of program paths. Assume that $\mathscr{A}$ is safe, well-labeled and Π-complete. Then program $\mathscr{P}$ is safe.

C. *Abstraction Algorithm*

We now combine POR with IMPACT. The obvious starting point is to modify the EXPAND function in Algorithm 1. We first introduce the modified expansion function $\text{EXPAND}_\Diamond$. However, changing only the expansion function turns out to be insufficient. Due to a subtle interplay between coverings and POR, the resulting algorithm does not guarantee Π_{mono}-completeness, and is unsound, which we illustrate with a small example. We then describe a method to fix this problem and present Algorithm 2, a sound variant of Impact with POR.

First, we change EXPAND such that only monotonic paths are unwound. To this end, instead of expanding all threads at a node, $\text{EXPAND}_\Diamond$ first checks if expanding with T yields a non-monotonic program path. This check is carried out in function $\text{SKIP}_\Diamond$ for given node v and thread T. Function $\text{SKIP}_\Diamond$ analyses the thread T' and action a' executed by the parent u of v, and returns true if the thread T is smaller than $T < T'$ and action a' is independent of a.

Algorithm 2 dPOR-IMPACT

---

procedure $\text{EXPAND}_\Diamond(v)$
  for $T \in \mathscr{T}$ with $\neg\text{SKIP}_\Diamond(v, T)$ **do**
    $\text{EXPAND-THREAD}(T, v)$

procedure $\text{SKIP}_\Diamond(v, T)$
  choose unique T', a' s.t. $u \xrightarrow{T', a'} v$
  return $\big(T < T' \wedge (\text{ACTION}(v, T) \parallel a')\big) \wedge \neg\text{LOOP}(u, T')$

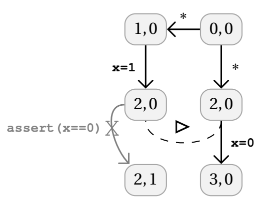
procedure $\text{CLOSE}_\Diamond(v)$
  for $w \in V^{<v}$: w uncovered **do**
    if $\ell(v) = \ell(w) \wedge \phi(v) \Rightarrow \phi(w)$ **then**
      $\rhd := (\rhd \cup \{(v, w)\}) \setminus \{(x, y) \in \rhd \mid v \rightsquigarrow y\}$
      for T with $v \xrightarrow{T, \ldots} v'$ and not $w \xrightarrow{T, \ldots} w'$ **do**
        $\text{EXPAND-THREAD}(T, w)$

Intuitively, two writes to the same variable are dependent, a read and a write to the same variable are dependent, but two reads to the same variable are independent. Two actions a and a' are independent, denoted by $a \parallel a'$, if $R_a \cap W_{a'} = \emptyset \ \wedge\ W_a \cap (R_{a'} \cup W_{a'}) = \emptyset$ where R_a and $R_{a'}$ are the variables being read, and, W_a and $W_{a'}$ the variables being written.

Additionally, we introduce function LOOP to detect control-flow loops. Function $\text{LOOP}(u, T)$ returns true if action $a = (l, N)$

of T at node u induces a back edge in the thread's control flow. This completes our discussion of EXPAND$_\Diamond$.

As mentioned before, just modifying EXPAND yields an unsound algorithm that does not guarantee Π_{mono}-completeness. Consider the example program below. Note that to violate the assertion, the context switch between the two threads has to happen right after T_1 has executed `x=1`. However, the covering between the left and the right $(2,0)$-node prevents this expansion, leading to an ART that is not Π_{mono}-complete. In particular, the counterexample path is not covered by the resulting ART, i.e., there is no corresponding path, as `assert(x==0)` is not expanded at the covering $(2,0)$-node.



T_1	T_2
`0: if(*)`	`0: assert(x==0);`
`1:    x=1;`	`1:`
`2: x=0;`	
`3:`	

the counterexample is
T_1: `*`, T_1: `x=1;`, T_2: `assert(x==0)`

To guarantee Π_{mono}-completeness, we modify CLOSE to carry out expansions at the covering node, so-called cover expansions – yielding function CLOSE$_\Diamond$. We consider actions that would have been expanded at the covered node, had there been no cover. These actions are now expanded in the covering node. In our example, this results in an expansion of `assert(x==0)` on the right $(2,0)$-node, which triggers a refinement that uncovers the left $(2,0)$-node and reveals the counterexample in the next step.

This combination of EXPAND$_\Diamond$ and CLOSE$_\Diamond$ guarantees Π_{mono}-completeness, as proved in the following lemma, which also establishes the correctness of dPOR-Impact:

Lemma 4.7: If Algorithm 2 reports that the program is safe, the computed ART $\mathscr{A}$ is Π_{mono}-complete.

Proof We need to show that every path $\pi \in \Pi_{mono}$ is covered by $\mathscr{A}$. We carry out a proof by induction on the length $N = |\pi|$ of π. The base case for $N = 1$ is trivial. Assume that $N \geq 2$ and that every path of length at most $N-1$ is covered. Let $\pi = (\mathfrak{l}_0, T_0, a_0, \mathfrak{l}_1)\dots(\mathfrak{l}_{N-1}, T_{N-1}, a_{N-1}, \mathfrak{l}_N) \in \Pi_{pm}$ be a path of length N. We need to prove that there exists a corresponding path $v_0, \dots, v_N$ that meets the criteria of Definition 4.4.

By induction hypothesis, there exists a corresponding path $v'_0, \dots, v'_{N-1}$ for the length $N-1$ prefix of π. As $\pi \in \Pi_{mono}$, we have that $\text{SKIP}_\Diamond(v'_{N-1}, T_{N-1}) = False$. Hence, if v'_{N-1} is not covered, it will be expanded yielding a suitable successor v'_N, and choosing $v_i = v'_i$ for all $i \in \{1, \dots, N\}$ we are done. So let us assume that v_{N-1} is covered. Then there exists v_{N-1} distinct from v_{N-1} such that $v_{N-1} \rhd v_{N-1}$ and v_{N-1} is not covered. It could be that $\text{SKIP}_\Diamond(v_{N-1}, T_{N-1}) = True$, however the covering $v'_{N-1} \rhd v_{N-1}$ must result from an invocation of CLOSE$_\Diamond$, forcing expansion of T_{N-1} at v_{N-1} and thus yields a suitable successor v_N. Thus we choose $v_i = v'_i$ for $i \in \{1, \dots, N-2\}$, v_{N-1} and v_N as above, and $u_{N-1} = v'_{N-1}$. ∎

D. Conditional Dependence

We now describe how to deal with aliasing in presence of pointers and shared tables. This leads to dynamic dependencies determined by the execution state, e.g., when dereferencing pointers, the dependence relation is determined by pointer aliasing. Two pointer variables may point to the same location leading to a dependency, or to disjoint locations. When accessing tables via indices, dependencies may arise when two threads access the same position in a table, which depends on the value of the indexing variable.

Dynamic dependencies can be accommodated in our framework by considering so-called *conditional dependence* between actions [2]. Effectively, the dependence relation, which was a binary relation between actions until now, becomes a ternary relation, such that dependencies are triples consisting of a state and two actions. When carrying out partial-order reduction, the ART is built in the same way as before, except that the dependency check takes into account the aliasing information.

Computation of the aliasing information can be carried out by simply inspecting the history of the state. However, note that covering produces nodes that represent states with potentially different histories. Hence, if aliasing information is used to prune expansions, this alias information must also be annotated in the node labels, to ensure soundness. This can be achieved as follows: we carry out a simple aliasing analysis along the history of a node, if we find that there is no aliasing (and hence no dependence), we refine the nodes along the path with inductive invariants that enforce absence of the alias. For a pair of accesses, we define an alias expression $alias$, such that the expression becomes true if and only if the two accesses go to the same address. The construction of alias expressions for typical array accesses is described, e.g., in [6].

For illustration, consider the example in Figure 1. For the path aA, we need to check independence of the access `v[i]=2` and `v[j]=-2`, which gives the alias expression $i = j$. Let π be the path to the node at which we check the alias relation, in our example aA. The accesses are independent if the conjunction of path formula and alias expression $\mathscr{F}(\pi) \wedge alias^{(|\pi|-1)}$ is unsatisfiable. In our example, this formula is unsatisfiable, due to the assume statement in line 1 of `main`, and the nodes along the path are refined with the interpolant $i \neq j$, and we can make the reduction depicted in the figure.

V. EXPERIMENTS

We have implemented the techniques described in this paper in a prototype tool, called IMPARA, a software model checker for concurrent C programs with POSIX or WIN32 threads. Experiments were run on an Intel Xeon machine with 8 cores at 3.07 GHz with 50 GB RAM. The timeout is 900 s and the memory limit is 15 GB. We make the implementation and detailed results available online at http://www.cprover.org/concurrent-impact/ for evaluation.

Comparison with Other Tools: We compare the performance of IMPARA 0.2 with the tools CBMC 4.5 [10] (bounded model checking with partial-order encoding), ESBMC 1.20 [11] (bounded model checking, POR and state

program	safe	CBMC	ESBMC	SatAbs	Threader	Impara
`dekker`	y	0.6*	2.2*	0.2	TO	**0.1**
`lamport`	y	12.4*	18.1*	**0.3**	38.1	**0.3**
`peterson`	y	0.2*	2.0*	0.3	4.8	**0.1**
`szymanski`	y	0.5*	4.7*	**0.2**	13.5	**0.2**
`read_write_u`	n	**0.2**	TO	0.8	58.4	0.6
`read_write_s`	y	**0.4**	TO	0.8	58.1	0.9
`time_var_mutex`	y	0.2	110.3	95.4	4.3	**0.1**
`stack_u`	n	1.0	TO	TO	80.6	**0.5**
`stack_s`	y	**33.5**	TO	TO	250.1	38.8

TABLE I: IMPARA vs. other tools on competition benchmarks

hashing), THREADER 0.92 [12] (predicate abstraction and thread-modular reasoning), and SATABS 3.1 [13] (SAT-based predicate abstraction).

To this end, we use the concurrency benchmarks from the *Second Competition on Software Verification* [14], which includes typical mutual exclusion protocols, such as Dekker, Peterson, Szymanski and Lamport, as well as programs that manipulate concurrent data structures.

Some benchmarks contain unbounded loops, which can be handled by IMPARA, SATABS and THREADER, while CBMC and ESBMC require an unwinding limit, which we set to 6, the maximum among the *bounded* loops. Partial loop exploration is marked with a star superscript at the respective running time.

We observe that IMPARA shows promising performance compared to the other tools, despite its prototype status. The running time for selected benchmarks are given in Table I. Each program contains assertions to be verified. Column "safe" indicates if the respective program is safe.

IMPARA 0.2 uses CBMC 4.5 as a front end. The back end, including the symbolic-execution engine, was written from scratch. To focus the implementation effort on the concurrency aspect, we use syntactic weakest preconditions as an interpolation procedure. For many typical concurrency benchmarks, weakest preconditions give sufficient invariants. However, we anticipate that leveraging a more advanced interpolation procedure could further improve performance.

We have implemented optimizations to speed up the frequently occurring cover checks. In a cascaded approach, we first use syntactic checks to cover trivial implications that can be resolved syntactically, e.g., $x > 0 \wedge y > 0$ trivially implies $y > 0$. Then we look up the implication in a table. Finally, if that fails, we invoke a SMT solver to check implication.

Benchmarks Using Weak Memory Consistency: The presented algorithm assumes interleaving semantics. Modern multi-core architectures, however, implement weaker consistency models, and therefore permit additional behaviors. Our technique can be extended to support popular consistency models including TSO (x86), PSO, RMO and PowerPC by combining it with the instrumentation proposed in [15].

The `sql` benchmark is a bug in PostgreSQL worker synchronization that occurs on the PowerPC architecture. A developer fix has also been found to be buggy. IMPARA is able to verify the safe programs and find counterexamples except for the PowerPC variant of the PostgreSQL benchmark where the tool times out. We anticipate that this can be fixed by a more aggressive expression simplification.

Effect of dPOR and Force Covers: To evaluate the benefit of dynamic partial-order reduction, and to compare different

program	safe	CBMC		ESBMC		SatAbs		Threader		Impara	
Sober benchmark											
`SC`	y	**0.3**	✓	1.2	✓	**0.3**	✓	120	FN	0.7	✓
`TSO`	n	**0.5**	✓	TO		2.6	✓	ERR		3.7	✓
`RMO`	n	**0.5**	✓	TO		2.5	✓	ERR		3.6	✓
`PSO`	n	**0.3**	✓	TO		1.4	✓	ERR		1.7	✓
`Power`	n	**0.3**	✓	TO		1.4	✓	ERR		1.7	✓
`fix_SC`	y	**0.3**	✓	1.3	✓	0.4	✓	120	FN	0.7	✓
`fix_TSO`	y	**0.3**	✓	TO		5.5	✓	ERR		1.3	✓
`fix_PSO`	y	**0.3**	✓	TO		5.6	✓	ERR		1.4	✓
`fix_power`	y	**0.3**	✓	TO		5.6	✓	ERR		1.4	✓
SQL benchmark											
`SC`	y	1.8*	(✓)	475.6*	✓	**0.3**	✓	1.7	FN	0.4	✓
`TSO`	y	6.9*	(✓)	TO		**0.3**	✓	3.25	FN	0.5	✓
`Power`	n	824.9*	✓	TO		TO		ERR		TO	
`dev_fix_Power`	n	TO		TO		17.7	FP	ERR		TO	

TABLE III: IMPARA on weak memory benchmarks

combinations of force cover and partial-order reduction, we experiment with four different configurations of IMPARA:

- **sPOR**: expands interleavings only when an action is executed that operates on shared variables; the original implementation of CLOSE is used.
- **sPOR+FC**: **sPOR** with force cover (FC).
- **dPOR**: dPOR-IMACT without force cover; this requires the $\text{CLOSE}_\Diamond$ function described in Sec. IV-C.
- **dPOR+FC**: **dPOR** with force cover.

Table II compares the four different configurations in terms of their running time ("s" for seconds), number of nodes ("$|V|$") and number of cover checks that require an implication check by an SMT solver ("impl"). Runs that have timed out are recorded with "TO" in the time field, and all other fields are filled with "–". To quantify the penalty incurred by cover expansions from $\text{CLOSE}_\Diamond$, we give the percentage ("C") of nodes resulting from cover expansions, e.g., 15% for `read_write_s` and around 27% for safe Sober weak-memory examples (to save space, we omit detailed results for weak memory benchmarks). Note that cover expansions were crucial to find assertion violations in the weak-memory benchmarks. For safe programs, we find that dPOR always produces less nodes than sPOR despite cover expansions.

Clearly, all configurations beat sPOR. On the other hand, we observe that POR and FC are complementary techniques. POR removes redundancies arising from thread interleaving, while FC covers thread-internal branching, e.g., from conditionals and loops, *as well as* redundant thread interleavings. For the latter, FC needs more unwindings than POR. For the smaller examples, these additional unwindings are few, as paths remain short, but the cost increases in larger programs. Comparing FC and POR, we observe that POR tends to reduce the number of necessary implications checks. This is because FC catches redundant interleavings that are removed by POR, and because it is a refinement technique, which triggers implication checks. Again, the cost of implication checks increases with program size, which can make POR scale better to larger problems.

VI. RELATED WORK

Partial-order reduction (POR) [2]–[4] has been proposed as a technique to combat state explosion by exploring only a representative subset of all possible interleavings, and has been implemented in the explicit-state model checkers SPIN [16] and Verisoft [5]. Dynamic POR techniques [17], [18] are based on the same concepts as classical static POR but capture dynamic

	LOC	safe	sPOR			sPOR+FC			dPOR				dPOR+FC			
			s	$\lvert V\rvert$	impl	s	$\lvert V\rvert$	impl	s	$\lvert V\rvert$	impl	C	s	$\lvert V\rvert$	impl	C
`dekker`	57	y	1.3	7407	9	0.2	504	8	0.1	433	3	0%	**0.1**	331	2	0%
`lamport`	79	y	9.6	36624	223	1.0	4740	226	1.0	7418	149	54%	**0.3**	1700	205	13%
`peterson`	45	y	0.7	3155	55	**0.1**	419	89	0.3	1081	43	0%	**0.1**	199	16	0%
`szymanski`	57	y	2.3	13332	14	0.3	1059	5	**0.2**	1264	5	0%	**0.2**	673	2	0%
`read_write_u`	59	n	5.4	31786	23	1.0	7628	119	1.6	18261	59	0%	**0.6**	4899	53	0%
`read_write_s`	68	y	75.2	109096	148	7.0	12932	1457	5.0	36777	129	14%	**0.9**	7065	223	15%
`time_var_mutex`	92	y	0.2	867	4	0.2	867	6	0.2	435	3	0%	**0.1**	252	1	0%
`stack_u`	144	n	TO	–	–	TO	–	–	3.1	2589	717	0%	**0.5**	424	81	0%
`stack_s`	144	y	TO	–	–	TO	–	–	TO	–	–	–	**38.8**	2037	4420	0%

TABLE II: Comparison of different configurations of IMPARA

dependencies induced by pointers on-the-fly during the state-space exploration.

Our monotone exploration strategy dPOR corresponds to the one used in [6], where POR is applied to SMT-based bounded model checking. The idea of cover expansions in function $\text{CLOSE}_{\Diamond}$ of our algorithm is inspired by a similar precaution in stateful dynamic POR [19].

Cimatti et al. combine static POR with lazy abstraction [20] to verify SystemC programs. There are several differences to our approach: our POR technique aims at dynamic dependencies induced by pointers, we are using Impact rather than predicate abstraction, and our approach is geared towards multi-threaded programs rather than SystemC programs.

Gupta et al. combine predicate abstractions with thread-modular proof rules [21], [22] in a tool called THREADER [23].

In the setting of single-threaded programs, the IMPACT algorithm has been re-implemented in a tool called WOLVERINE and compared with SATABS [24]. Beyer et al. have developed an approach where different invariant-generation techniques can be combined in a configurable tool CPA-CHECKER [25], together with techniques such as large block encoding [26]. Using CPA-CHECKER, they compare predicate abstraction with Impact [8] and evaluate the effectiveness of force covers.

VII. CONCLUSION

We have presented a new software model checking technique for concurrent programs based on lazy abstraction with interpolants and partial-order reduction, which performs very favorably compared to existing tools. In the future, we would like to incorporate more advanced invariant-generation techniques and investigate more aggressive POR techniques.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers, and also Subodh Sharma, Luis María Ferrer Fioriti and Matt Lewis for their valuable feedback.

REFERENCES

[1] K. L. McMillan, "Lazy abstraction with interpolants," in *CAV*, ser. LNCS, vol. 4144. Springer, 2006, pp. 123–136.

[2] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems*, ser. LNCS. Springer, 1996, vol. 1032.

[3] D. Peled, "All from one, one for all: on model checking using representatives," in *CAV*, ser. LNCS, vol. 697. Springer, 1993, pp. 409–423.

[4] A. Valmari, "Stubborn sets for reduced state space generation," in *Applications and Theory of Petri Nets*, ser. LNCS, vol. 483. Springer, 1989, pp. 491–515.

[5] P. Godefroid, "Software model checking: The VeriSoft approach," *Formal Methods in System Design*, vol. 26, no. 2, pp. 77–101, 2005.

[6] C. Wang, Z. Yang, V. Kahlon, and A. Gupta, "Peephole partial order reduction," in *TACAS*, ser. LNCS, vol. 4963. Springer, 2008, pp. 382–396.

[7] K. L. McMillan, "An interpolating theorem prover," *Theor. Comput. Sci.*, vol. 345, no. 1, pp. 101–121, 2005.

[8] D. Beyer and P. Wendler, "Algorithms for software model checking: Predicate abstraction vs. Impact," in *FMCAD*. IEEE, 2012, pp. 106–113.

[9] A. W. Mazurkiewicz, "Trace theory," in *Advances in Petri Nets*, ser. LNCS, vol. 255. Springer, 1986, pp. 279–324.

[10] J. Alglave, D. Kroening, and M. Tautschnig, "Partial orders for efficient bounded model checking of concurrent software," in *CAV*, 2013, pp. 141–157.

[11] L. Cordeiro and B. Fischer, "Verifying multi-threaded software using SMT-based context-bounded model checking," in *ICSE*. ACM, 2011, pp. 331–340.

[12] A. Gupta, C. Popeea, and A. Rybalchenko, "Threader: A constraint-based verifier for multi-threaded programs," in *CAV*, 2011.

[13] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "Predicate abstraction of ANSI–C programs using SAT," *Formal Methods in System Design (FMSD)*, vol. 25, pp. 105–127, September–November 2004.

[14] D. Beyer, "Second competition on software verification – (summary of SV-COMP 2013)," in *TACAS*, ser. LNCS, vol. 7795. Springer, 2013, pp. 594–609.

[15] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig, "Software verification for weak memory via program transformation," in *ESOP*, ser. LNCS, vol. 7792. Springer, 2013, pp. 512–532.

[16] G. J. Holzmann, "Software model checking with SPIN," *Advances in Computers*, vol. 65, pp. 78–109, 2005.

[17] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *POPL*. ACM, 2005, pp. 110–121.

[18] G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv, "Cartesian partial-order reduction," in *SPIN*, ser. LNCS, vol. 4595. Springer, 2007, pp. 95–112.

[19] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby, "Efficient stateful dynamic partial order reduction," in *SPIN*, ser. LNCS, vol. 5156. Springer, 2008, pp. 288–305.

[20] A. Cimatti, I. Narasamdya, and M. Roveri, "Boosting lazy abstraction for SystemC with partial order reduction," in *TACAS*, ser. LNCS, vol. 6605. Springer, 2011, pp. 341–356.

[21] S. S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs I," *Acta Inf.*, vol. 6, pp. 319–340, 1976.

[22] C. B. Jones, "Tentative steps toward a development method for interfering programs," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 4, pp. 596–619, 1983.

[23] A. Gupta, C. Popeea, and A. Rybalchenko, "Predicate abstraction and refinement for verifying multi-threaded programs," in *POPL*. ACM, 2011, pp. 331–344.

[24] D. Kroening and G. Weissenbacher, "Interpolation-based software verification with WOLVERINE," in *CAV*, ser. LNCS, vol. 6806. Springer, 2011, pp. 573–578.

[25] D. Beyer and M. E. Keremoglu, "CPAchecker: A tool for configurable software verification," in *CAV*, ser. LNCS, vol. 6806. Springer, 2011, pp. 184–190.

[26] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani, "Software model checking via large-block encoding," in *FMCAD*. IEEE, 2009, pp. 25–32.