November 2015

Verification of Concurrent Software

Daniel KROENING
Department of Computer Science, University of Oxford

Abstract. We provide a tutorial on the verification of concurrent software. We first discuss semantics of modern shared-variable concurrent software. We then provide an overview of algorithmic methods for analysing such software. We begin with Bounded Model Checking, a technique that performs an analysis of program paths up to a user-specified length. We then discuss methods that are, in principle, able to provide an unbounded analysis. We discusshow to apply predicate abstraction to concurrent software and then present an extension of lazy abstraction with interpolants to such software.

Keywords. program analysis, software verification, concurrent software, model checking

1. Introduction

Software engineers use of numerous flavours of concurrent programming to achieve better scalability, savings in power, increased reliability, and to boost performance. The need for programs that make diligent use of concurrent computational resources has been exacerbated by power-efficient multi-core CPUs, which are now widely deployed, but still underutilised due to the lack of appropriate software. Concurrent software is particularly difficult to test, as bugs depend on particular interlavings between the sequential computations. Defects are therefore difficult to reproduce and diagnose, and often elude even very experienced programmers.

Random simulation and testing, while automated, has severe limitations, particularly in the case of concurrent software, in which the plethora of possible thread interleavings often conspires to conceal design flaws. Formal verification, on the other hand, can also be automated, and tools that implement it check a concurrent program for *all* its possible behaviours. It therefore promises higher gains in productivity, if done the right way.

Numerous tools to hunt down functional flaws in hardware designs have been available commercially for a number of years. The use of such tools is widespread, and there is a broad range of vendors. By contrast, the market for formal tools that address the need for quality software—and even more so for concurrent software—is still in its infancy.

The promise of automated bug-finding in complex software is not new. The *Verifying Compiler* was identified as a research goal in the 70s, and was recently re-stated as a Grand Challenge for computing research by Sir Tony Hoare [29]. Whilst the problem in general is not yet solved, tools that specialise in very specific aspects of the problem demonstrated significant results. An instance is the SLAM project at Microsoft Research. By focussing on lightweight, control-flow dominated properties, the SLAM toolkit is able to verify Windows device drivers with hundreds of thousands of lines of code [5]. SLAM

checks for violations of rules such as "a thread must not acquire a lock it has already acquired, nor release a lock it does not hold". SLAM was shipped as Static Driver Verifier (SDV) to developers.

However, automatic analysis of concurrent programs is still a challenge in practice. Hardly any of the very few existing tools for software of this kind will prove safety properties for a thousand lines of code [20]. Most papers name the number of *thread interleavings* of a concurrent program as a reason for the difficulty.

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. In this tutorial, we discuss a technique that offers a clear view on how to address each of these aspects when analysing concurrent programs [49]. Lazy Abstraction with Interpolants [43], also known as the Impact algorithm, is an 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. We discuss an extension of this method to shared-variable concurrent programs where the explosion in the number of thread interleavings is dealt with by means of *partial-order reduction*.

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 we focus on this form of concurrency in this tutorial, we nevertheless discuss alternatives, e.g., message passing and usage of memories that are not shared.

Outline We discuss semantics of shared-variable concurrent programs and related models in Sec. 2. In Sec. 3, we give an extension to the IMPACT algorithm to concurrent programs. Finally, we conclude in Sec. 4 and give references for further reading.

2. Semantics of Concurrent Software

There is a broad variety of concurrent computation, and we briefly survey the most prominent representatives. We focus on the aspect of concurrency; the semantics of programming languages for purely sequential computation is a challenge in itself, and so is the extraction of formal models from program source code. We refer the reader to [37] for a discussion of the practical problems that arise when analysing modern software using formal tools.

2.1. Synchronous Systems

Synchronous reactive processes are widely used for modeling and model-based design of embedded software systems. Processes of this kind synchronize at designated points in their control flow. Harels Statecharts [28] is a popular formalism for specifying such processes. Statecharts extend conventional state transition diagrams with the notion of hierarchy, concurrency and communication.

The processes that form a synchronous system may write to the same variable, which is called a *race*. As the processes are controlled by means of guards, it is difficult to establish statically whether a given design has a race or not. Approaches to formal verification of synchronous reactive systems predominantly rely on building a global

transition relation for the system. This permits the application of a wide range of standard methods for state space exploration such as BDD-based Model Checking, Bounded Model Checking, k-induction or interpolation. In addition, there are approaches that implement domain-specific optimisations for the analysis of synchronous systems. For instance, a variant of the IMPACT algorithm for synchronous reactive systems is given in [40]. SystemC is frequently used to model hardware, and thus is an instance of a reactive synchronous system. In [10, 11], the use of Bounded Model Checking and Predicate Abstraction is discussed to compute a sufficient condition for the existence of a race between any pair of proceesses in the model.

2.2. Interleaving Semantics

Popular programming languages such as C, C++ and Java readily embrace concurrent programming, via their pthread and thread class APIs, respectively. Communication among threads is naturally enabled via shared (global-scope) variables and mutexes, as well as via nonblocking API constructs such as signals and broadcasts. Much existing system-level code such as device drivers make use of these mechanisms to protect shared data and to synchronize thread execution; operating systems such as many Unix derivatives and Mac OS X directly support them.

If mutexes and other synchronisation is used correctly, it is possible to reason about such programs using Lamport's *Sequential Consistency* (SC) [39]. In this case, *"the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program"*. This semantics is also called the *interleaving semantics*. Most Model Checking and program analysis methods assume that interleaving semantics is applicable, and we discuss one such method in Sec. 3. Interleaving semantics does not apply to certain lock-free programs, which we discuss next.

2.3. Shared Memories with Weak Consistency

Multiprocessors, or multi-core CPUs, implement *weak memory models*, which feature optimisations such as *instruction reordering* and *store buffering* (both appearing on x86), or *store atomicity relaxation* (a particularity of Power and ARM). Hence, multiprocessors allow more behaviours than Lamport's sequential consistency. This has a dramatic effect on programmers, most of whom learned to program with sequential consistency.

Fence Insertion Fortunately, architectures provide special *fence* (or *barrier*) instructions to prevent certain relaxed behaviours. Yet both the questions of *where* and *how* to insert fences are contentious, as fences are architecture-specific and expensive in terms of runtime.

Attempts at automatically placing fences include Visual Studio 2013, which offers an option to guarantee acquire/release semantics. The C++11 standard provides an elaborate API for inter-thread communication, giving the programmer some control over which fences are used, and where. But the use of such APIs might be a hard task, even for expert programmers. For example, Norris and Demsky reported a bug found in a published C11 implementation of a work-stealing queue [45].

There are two approaches to generating fences automatically, which differ in their goal. One goal is to enforce robustness/stability, which implies that sequential consistency is restored. A representative of this approach is [2], in which a set of fences for low-level memory models such as TSO and POWER/ARM is computed.

Restoring sequential consistency may require a large number of fences. Other works, therefore, focus on adding the minimal number of fences required to guarantee a user-specified correctness property. For instance, the work in [33] uses Bounded Model Checking to identify the fences necessary to fix a given assertion violation.

Checking Lock-Free Code High-performance concurrent data structures deliberately permit certain relaxed behaviours in order to achieve better performance. This motivates the need to analyse such code. Criteria for the soundness of data flow analyses for such programs are given in [1]. It is possible to re-use an analyser for programs that have interleaving semantics by instrumenting the program with additional buffers to mimick the behaviour of the hardware [3]. A verification method that natively supports weak-memory semantics is [4].

2.4. Scratch-Pad Memories

Heterogeneous multicore processors such as GPUs circumvent the shared memory bottleneck by equipping cores with small "scratch-pad" memories. These fast, private memories are not coherent with main memory, and allow independent calculations to be processed in parallel by separate cores without contention. While this can boost performance, it places heterogeneous multicore programming at the far end of the concurrent programming spectrum in terms of its complexity. The programmer can no longer rely on the hardware and operating system to seamlessly transfer data between the levels of the memory hierarchy, and must instead manually orchestrate data movement between memory spaces using direct memory access (DMA).

Low-level data movement code is error-prone: misuse of DMA operations can lead to DMA races, where concurrent DMA operations refer to the same portion of memory, and at least one modifies the memory. DMA races can lead to nondeterministic bugs that are difficult to reproduce and fix. A BMC-based tool to detect DMA races for IBM's Cell BE architecture is presented in [16–19].

2.5. Interrupts

Interrupts are a key design primitive for embedded software that interacts closely with hardware. The interrupt mechanism enables timely response to outside stimuli in a power-efficient way. Interrupts are commonplace in all styles of computing platforms, including safety-critical embedded software, lowpower mobile platforms and high-end information systems.

But interrupt-driven code is difficult to engineer. Device drivers, typical examples of software that uses interrupts heavily, are known as the "fault-hotspots" of the Linux Kernel. The root cause of the problem is the nondeterminism inherent in systems that use interrupts; the hardware can divert control to the interrupt service routine (ISR) at any point in time, resulting in possibly surprising interactions between the code that is interrupted and the ISR. The problem is exacerbated by interrupt nesting, where the ISR itself can be preempted by interrupts with higher priority.

Most existing approaches to validating interrupt-driven software rely on testing. But testing is particularly ineffective in the case of nested interrupts, as the number of possible interleavings—i.e. interspersions of interrupts within a run of the code—grows exponentially in the number of interrupts that occur. Bugs are therefore easily missed, and any errors that are observed are difficult to reproduce. The concurrent nature of interrupt generation and handling and the sheer size of the search space suggest a formal approach to validation. Model Checking, in particular, shines when applied to problems with many subtle corner cases. Several model-checking based approaches to the problem have therefore been explored. A standard technique is to instrument the source code of the program with calls to the interrupt procedure, in effect mimicking the semantics of the hardware interrupt. The instrumented program is then passed to a conventional analyser for sequential programs. The approach is straightforward to implement, and is in principle viable in the case of nested interrupts as well. Sophisticated instances of this type of approach use over-approximating analyses, akin to partial-order reduction, to determine program locations where the call to the ISR can be omitted safelyfor example when variables not shared with the ISR are read or written.

The interleaving semantics of programs with nested interrupts strongly resembles that of programs with multiple threads. In particular, it is clear that the behaviours under sequential consistency are a superset of the interrupt semantics. This suggests an analysis using tools that use SC. False alarms caused by the overapproximation can be addressed by means of suitable instrumentation. A third, alternative approach, is given in [36]. It gives a *strengthening* of the encoding discussed in [4] to eliminate those behaviours that are not SC.

2.6. Message Passing

Distributed systems are often developed using the *message passing* paradigm, where the only way to share data between processes is by passing messages over a network. The Message Passing Interface (MPI) is the *lingua franca* of high-performance computing (HPC) and remains one of the most widely used APIs for building distributed message-passing applications.

However, message passing systems are hard to design as they require implementing and debugging complex protocols. These protocols and their interleaved executions are often non-trivial to analyse as the safety and liveness properties of such systems are usually violated only during some intricate, low-probability interleavings. Given the wide adoption of the MPI in large-scale studies in science and engineering, it is important to have means to establish some formal guarantees, like deadlock-freedom, on the behaviour of MPI programs.

Reasoning about MPI programs is hard. This is primarily due to the presence of non-determinism that is induced by various MPI primitives and the buffering/arbitration effects in the MPI nodes and the network. For instance, a popular choice in MPI programs to achieve better performance is the use of receive calls with `MPI_ANY_SOURCE` argument; such calls are called “wildcard receives”. A wildcard receive in a process can be matched with any sender targeting the process, thus the matching between senders and receivers is susceptible to network delivery nondeterminism. MPI calls such as probe and wait are sources of nondeterminism as well. This prevalence—and indeed, preference—for nondeterminism renders MPI programs susceptible to the schedule-space explosion prob-

lem. This suggests that analysis techniques for concurrent programs might be applicable. A variant of the BMC-based method from [4] is presented in [22].

3. Checking Concurrent Software with IMPACT

3.1. Overview

We give an exemplar of a method for verifying shared-variable concurrent programs using interleaving semantics. The presentation that follows is based on [49].

Lazy abstraction with interpolants [43], 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 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) [23, 24, 47, 48]. 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

main()	thread T_1	thread T_2

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

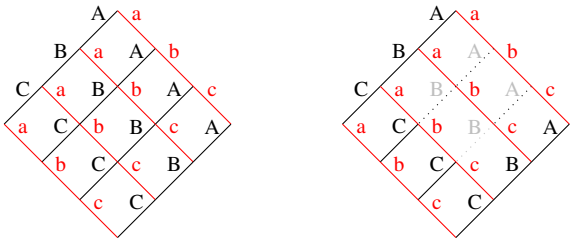


Figure 1. An example program (top) and its complete interleaving (left) and reduced interleaving semantics (right).

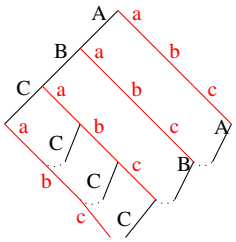


Figure 2. *Impact* with POR and force cover

reduction is able to discover this earlier. Discovering redundant paths early on during the exploration is crucial to avoid path explosion.

3.2. 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 $(\mathit{init}, \mathscr{T})$ where $\mathit{init} \in \mathscr{F}(V)$ 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: $\ldots$ is represented as $(l_1, \{(x' = y+1 \wedge y' = y; l_2)\})$. An assertion l_1: `assert(x<y);` l_2: $\ldots$ 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: $\ldots$ 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 $\mathit{enabled}(s)$. We assume that an action $a = (l, N)$ defines a total function $\{s \in S \mid a \in \mathit{enabled}(s)\} \to S$ 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 $\mathit{init}^{(0)} \wedge R_0^{(0)}, \ldots R_{N-1}^{(N-1)}$ 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 $\mathit{init} \Rightarrow I(\mathfrak{l}^i)$ 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 $I(\mathfrak{l}) \equiv \mathit{False}$ 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* [42]. 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 $\widehat{A}_1 \equiv \mathit{True}$, the last formula is equivalent to false $\widehat{A}_N \equiv \mathit{False}$, 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 $A_1, \ldots, A_N$ [42].

3.3. 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 $(V, \varepsilon, \rightarrow, \triangleright)$ consisting of a tree with nodes V, root node $\varepsilon \in V$, edges $\rightarrow \subseteq V^2$, and a covering relation $\triangleright \subseteq V^2$ 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 $\triangleright$.

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 $\mathscr{A} = (V, \varepsilon, \rightarrow, \triangleright)$ 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., $(v,w) \in \triangleright$: $\phi(v) \Rightarrow \phi(w)$ 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 [43], the labeling immediately gives a safety invariant M, $M(\mathfrak{l}') = \bigvee\{\phi(v) \mid \mathfrak{l}(v) = \mathfrak{l}'\}$. ∎

Algorithm 1 Impact with support for concurrent programs

```
 1: procedure MAIN()
 2:     Q := {ε}, ▷ := ∅
 3:     while Q ≠ ∅ do
 4:         select and remove v from Q
 5:         CLOSE(v)
 6:         if v not covered then
 7:             if error(v) then
 8:                 REFINE(v)
 9:             EXPAND(v)
10:     return 𝒫 is safe
11:
12: procedure EXPAND(v)
13:     for T ∈ 𝒯 do
14:         EXPAND-THREAD(T, v)
15:
16: procedure EXPAND-THREAD(T, v)
17:     (l, φ) := v
18:     for (l, N) ∈ A(T) with l_T = l do
19:         for (R, l') ∈ N do
20:             w := fresh node
21:             𝔩(w) := 𝔩[T ↦ l']
22:             φ(w) := True
23:             Q := Q ∪ {w}, V := V ∪ {w}
24:             →:=→ ∪{(v, T, R, w)}
30: procedure CLOSE(v)
31:     for w ∈ V^{≺v} : w uncovered do
32:         if 𝔩(v) = 𝔩(w) ∧ φ(v) ⇒ φ(w) then
33:             ▷ := ▷ ∪ {(v, w)}
34:             ▷ := ▷ \ {(x, y) ∈ ▷ | v ⇝ y}
35:
36: procedure REFINE(v)
37:     if v not error node or φ(v) ≡ False then
38:         return
39:     π := v_0, ... v_N path from ε to v
40:     if 𝒻(π) has interpolant A_0 ... A_N then
41:         for i = 0 ... N do
42:             φ := A_i^{-i}
43:             if φ(v_i) ⊭ φ then
44:                 Q := Q ∪ {w | w ▷ v_i}
45:                 ▷ := ▷ \ {(w, v_i) | w ▷ v_i}
46:                 φ(v_i) := φ(v_i) ∧ φ
47:         for w ∈ V s.t. w ⇝ v do
48:             CLOSE(w)
49:     else
50:         abort (program unsafe)
```

3.3.1. 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 24), 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 $w \triangleright v_i$ 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 $v \triangleright w$, it only considers nodes created before v, denoted by the set $V^{\prec v} \subsetneq V$. This is to ensure stable behavior, as covering in arbitrary order may uncover other nodes, which may not terminate. Thus only for uncovered nodes $w \in V^{\prec v}$, 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 relation $\triangleright$. To restore well-labeling, all pairs (x,y) where y is a descendant of v, denoted by $v \rightsquigarrow y$, are removed from $\triangleright$, as v and all its descendants are covered.

MAIN first initializes the queue with the initial node ε, and the relation $\triangleright$ 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 [9] 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 subsection.

3.4. 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 com-

bines 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 subsection, 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 [50].

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.

3.4.1. 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 3.4 (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 $a_1 \not\mid\mid a_2$.*

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 [41]:

Definition 3.5 (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:

Definition 3.6 (Monotonic paths) *A program path*

$$\pi = (\mathfrak{l}_0, T_0, a_0, \mathfrak{l}_1) \dots (\mathfrak{l}_{N-1}, T_{N-1}, a_{N-1}, \mathfrak{l}_N)$$

is monotonic *if for all $i, j \in \{0, \dots, N-1\}$ with $i < j$, $a_i \mid\mid a_j$ and $T_i > T_j$, we have $j \neq i+1$. Let Π_{mono} be the set of monotonic program paths.*

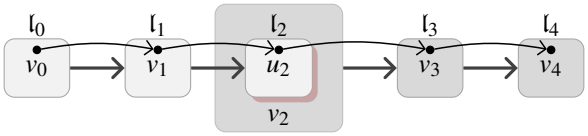


Figure 3. Path correspondence. Rounded rectangles represent ART nodes $v_0, \ldots, v_4$ and u_2. We have $u_2 \triangleright v_2$. The gray arrows depict ART edges. The path $l_0 \ldots l_5$ is a control flow path.

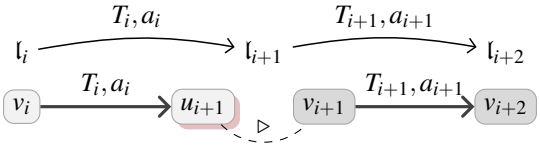


Figure 4. Illustration of Definition 3.7. The diagram shows a fragment of an ART $\mathscr{A}$ with notation for the nodes of the definition. The dashed line represents a covering edge.

3.4.2. 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 $l_0 \ldots l_5$ in Figure 3. While prefix $l_0 l_1 l_2$ can be matched by node sequence $v_0 v_1 u_2$, node u_2 is covered by node v_2, formally $u_2 \triangleright v_2$. 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 $\triangleright$ 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 $l_0 \ldots l_4$ 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 $l(u_{i+1}) = l(v_{i+1}) = l_{i+1}$ and $u_{i+1} \triangleright v_{i+1}$. 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 3.7 (Corresponding paths & path cover) *Consider a program $\mathscr{P}$. Let $\mathscr{A}$ be an ART for $\mathscr{P}$ and let $\pi = (l_0, T_0, a_0, l_1) \ldots (l_{N-1}, T_{N-1}, a_{N-1}, l_N)$ 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 $i \in \{0, \ldots, N-1\}$, $l(v_i) = l_i$, and*

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

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 3.8 (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 $\mathscr{A}$.*

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

Proposition 3.9 *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.*

3.4.3. 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.

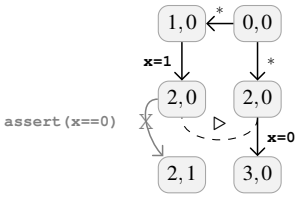
First, we change EXPAND such that only monotonic paths are unwound. To this end, instead of expanding all threads at a node, $\text{EXPAND}_\Diamond$ first checks if expanding with T yields a non-monotonic program path. This check is carried out in function $\text{SKIP}_\Diamond$ for given node v and 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

```
1:  procedure EXPAND◇(v)
2:      for T ∈ 𝒯 with ¬SKIP◇(v,T) do
3:          EXPAND-THREAD(T,v)
4:
5:  procedure SKIP◇(v,T)
6:      choose unique T′,a′ s.t. u -T′,a′-> v
7:      return (T < T′ ∧ (ACTION(v,T) || a′)) ∧ ¬LOOP(u,T′)
8:
9:  procedure CLOSE◇(v)
10:     for w ∈ V^≺v : w uncovered do
11:         if ι(v) = ι(w) ∧ φ(v) ⇒ φ(w) then
12:             ▷ := (▷ ∪ {(v,w)}) \ {(x,y) ∈ ▷ | v ⇝ y}
13:             for T with v -T,...-> v′ and not w -T,...-> w′ do
14:                 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 $a \,||\, a'$, 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 $\text{EXPAND}_\Diamond$.



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

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

Figure 5. Example to illustrate that just modifying EXPAND yields an unsound algorithm that does not guarantee Π_{mono}-completeness

As mentioned before, just modifying EXPAND yields an unsound algorithm that does not guarantee Π_{mono}-completeness. Consider the example program in Fig. 5. 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.

To guarantee Π_{mono}-completeness, we modify CLOSE to carry out expansions at the covering node, so-called cover expansions – yielding function $\text{CLOSE}_\Diamond$. 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 $\text{EXPAND}_\Diamond$ and $\text{CLOSE}_\Diamond$ guarantees Π_{mono}-completeness, as proved in the following lemma, which also establishes the correctness of dPOR-Impact:

Lemma 3.10 *If Algorithm 2 reports that the program is safe, the computed ART $\mathscr{A}$ is Π_{mono}-complete.*

3.4.4. 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 [23]. 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 [50].

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.

3.5. Related Work

Partial-order reduction (POR) [23, 47, 48] 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 [30] and Verisoft [24]. Dynamic POR techniques [21, 26] are based on the same concepts as classical static POR but capture dynamic dependencies induced by pointers on-the-fly during the state-space exploration.

Our monotone exploration strategy dPOR corresponds to the one used in [50], where POR is applied to SMT-based bounded model checking. The idea of cover expansions in function CLOSE$_\Diamond$ of our algorithm is inspired by a similar precaution in stateful dynamic POR [52].

Cimatti et al. combine static POR with lazy abstraction [12] 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 [32, 46] in a tool called THREADER [27].

In the setting of single-threaded programs, the IMPACT algorithm has been re-implemented in a tool called WOLVERINE and compared with SATABS [38]. Beyer et al. have developed an approach where different invariant-generation techniques can be combined in a configurable tool CPA-CHECKER [8], together with techniques such as large block encoding [7]. Using CPA-CHECKER, they compare predicate abstraction with Impact [9] and evaluate the effectiveness of force covers.

4. Conclusion and Further Reading

We have given an account of IMPARA [49], an algorithmic method for checking shared-variable concurrent programs that under interleaving semantics. The technique combines Lazy Abstraction with Interpolants to deal with data and partial order reduction to deal with the exploding number of thread interleavings. However, there are numerous further techniques for verifying concurrent programs.

In *Bounded Model Checking* (BMC), the loops in the program are unwound together with a property to form a logical formula, which is passed to a suitable decision procedure [13]. We discuss an extension of BMC with partial-order constraints to obtain a variant of BMC for shared-variable concurrent software [4]. An asymptotically-improved variant of the encoding is presented in [31].

Predicate Abstraction [14, 25] is a method for systematic abstraction of programs. It consists in abstracting data by only keeping track of certain predicates on program variables. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. Abstracting a concurrent program yields a concurrent Boolean program, which can be given to a suitable model checker. The method can be applied to concurrent software, which gives rise to a concurrent Boolean program [6, 15, 51].

Concurrent programs frequently have an unknown number of threads, created either on program start or dynamically during execution. This scenario is most relevant in practice: it covers workload-aware programs, where the number of worker threads is increased at runtime in response to client requests. The difficulty for search-based analysis methods is that such program models give rise to infinite state spaces. Nevertheless, the detection of specific classes of bugs in such programs has long been known to be decidable, such as by reduction to the coverability problem for wellquasi-ordered systems. An instance of this approach is explored in [34, 35]. Data abstraction is performed using predicate abstraction.

References

[1] J. Alglave, D. Kroening, J. Lugton, V. Nimal, and M. Tautschnig. Soundness of data flow analyses for weak memory models. In H. Yang, editor, *Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5-7, 2011. Proceedings*, volume 7078 of *Lecture Notes in Computer Science*, pages 272–288. Springer, 2011.

[2] J. Alglave, D. Kroening, V. Nimal, and D. Poetzl. Don't sit on the fence – A static analysis approach to automatic fence insertion. In A. Biere and R. Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 508–524. Springer, 2014.

[3] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In M. Felleisen and P. Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 512–532. Springer, 2013.

[4] J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In N. Sharygina and H. Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2013.

[5] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011.

[6] G. Basler, M. Mazzucchi, T. Wahl, and D. Kroening. Context-aware counter abstraction. *Formal Methods in System Design*, 36(3):223–245, 2010.

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

[8] D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In G. Gopalakrishnan and S. Qadeer, editors, *CAV*, volume 6806 of *LNCS*, pages 184–190. Springer, 2011.

[9] D. Beyer and P. Wendler. Algorithms for software model checking: Predicate abstraction vs. Impact. In G. Cabodi and S. Singh, editors, *FMCAD*, pages 106–113. IEEE, 2012.

[10] N. Blanc and D. Kroening. Race analysis for SystemC using model checking. In S. R. Nassif and J. S. Roychowdhury, editors, *2008 International Conference on Computer-Aided Design, ICCAD 2008, San Jose, CA, USA, November 10-13, 2008*, pages 356–363. IEEE Computer Society, 2008.

[11] N. Blanc and D. Kroening. Race analysis for SystemC using model checking. *ACM Trans. Design Autom. Electr. Syst.*, 15(3), 2010.

[12] A. Cimatti, I. Narasamdya, and M. Roveri. Boosting lazy abstraction for SystemC with partial order reduction. In P. A. Abdulla and K. R. M. Leino, editors, *TACAS*, volume 6605 of *LNCS*, pages 341–356. Springer, 2011.

[13] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[14] M. Colón and T. E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, volume 1427 of *Lecture Notes in Computer Science*, pages 293–304. Springer, 1998.

[15] B. Cook, D. Kroening, and N. Sharygina. Over-approximating boolean programs with unbounded thread creation. In *Formal Methods in Computer-Aided Design, 6th International Conference, FMCAD 2006, San Jose, California, USA, November 12-16, 2006, Proceedings*, pages 53–59. IEEE Computer Society, 2006.

[16] A. F. Donaldson, L. Haller, and D. Kroening. Strengthening induction-based race checking with lightweight static analysis. In R. Jhala and D. A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2011.

[17] A. F. Donaldson, D. Kroening, and P. Rümmer. Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 280–295. Springer, 2010.

[18] A. F. Donaldson, D. Kroening, and P. Rümmer. Automatic analysis of DMA races using model checking and k-induction. *Formal Methods in System Design*, 39(1):83–113, 2011.

[19] A. F. Donaldson, D. Kroening, and P. Rümmer. SCRATCH: a tool for automatic analysis of DMA races. In C. Cascaval and P. Yew, editors, *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pages 311–312. ACM, 2011.

[20] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[21] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In J. Palsberg and M. Abadi, editors, *POPL*, pages 110–121. ACM, 2005.

[22] V. Forejt, D. Kroening, G. Narayanaswamy, and S. Sharma. Precise predictive analysis for discovering communication deadlocks in MPI programs. In C. B. Jones, P. Pihlajasaari, and J. Sun, editors, *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, volume 8442 of *Lecture Notes in Computer Science*, pages 263–278. Springer, 2014.

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

[24] P. Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[25] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[26] G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv. Cartesian partial-order reduction. In D. Bosnacki and S. Edelkamp, editors, *SPIN*, volume 4595 of *LNCS*, pages 95–112. Springer, 2007.

[27] A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In T. Ball and M. Sagiv, editors, *POPL*, pages 331–344. ACM, 2011.

[28] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[29] C. A. R. Hoare. The verifying compiler, a grand challenge for computing research. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, volume 3385 of *Lecture Notes in Computer Science*, pages 78–78. Springer, 2005.

[30] G. J. Holzmann. Software model checking with SPIN. *Advances in Computers*, 65:78–109, 2005.

[31] A. Horn and D. Kroening. On partial order semantics for SAT/SMT-based symbolic encodings of weak memory concurrency. In S. Graf and M. Viswanathan, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 35th IFIP WG 6.1 International Conference, FORTE 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings*, volume 9039 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2015.

[32] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[33] S. Joshi and D. Kroening. Property-driven fence insertion using reorder bounded model checking. In N. Bjørner and F. D. de Boer, editors, *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, volume 9109 of *Lecture Notes in Computer Science*, pages 291–307. Springer, 2015.

[34] A. Kaiser, D. Kroening, and T. Wahl. Lost in abstraction: Monotonicity in multi-threaded programs. In P. Baldan and D. Gorla, editors, *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*, volume 8704 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2014.

[35] A. Kaiser, D. Kroening, and T. Wahl. A widening approach to multithreaded program verification. *ACM Trans. Program. Lang. Syst.*, 36(4):14:1–14:29, 2014.

[36] D. Kroening, L. Liang, T. Melham, P. Schrammel, and M. Tautschnig. Effective verification of low-level software with nested interrupts. In Nebel and Atienza [44], pages 229–234.

[37] D. Kroening and M. Tautschnig. Automating software analysis at large scale. In P. Hlinený, Z. Dvorak, J. Jaros, J. Kofron, J. Korenek, P. Matula, and K. Pala, editors, *Mathematical and Engineering Methods in Computer Science - 9th International Doctoral Workshop, MEMICS 2014, Telč, Czech Republic, October 17-19, 2014, Revised Selected Papers*, volume 8934 of *Lecture Notes in Computer Science*, pages 30–39. Springer, 2014.

[38] D. Kroening and G. Weissenbacher. Interpolation-based software verification with WOLVERINE. In *CAV*, volume 6806 of *LNCS*, pages 573–578. Springer, 2011.

[39] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[40] K. Madhukar, M. K. Srivas, B. Wachter, D. Kroening, and R. Metta. Verifying synchronous reactive systems using lazy abstraction. In Nebel and Atienza [44], pages 1571–1574.

[41] A. W. Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Advances in Petri Nets*, volume 255 of *LNCS*, pages 279–324. Springer, 1986.

[42] K. L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.

[43] K. L. McMillan. Lazy abstraction with interpolants. In T. Ball and R. B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[44] W. Nebel and D. Atienza, editors. *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*. ACM, 2015.

[45] B. Norris and B. Demsky. CDSchecker: checking concurrent data structures written with C/C++ atomics. In A. L. Hosking, P. T. Eugster, and C. V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 131–150. ACM, 2013.

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

[47] D. Peled. All from one, one for all: on model checking using representatives. In C. Courcoubetis, editor, *CAV*, volume 697 of *LNCS*, pages 409–423. Springer, 1993.

[48] A. Valmari. Stubborn sets for reduced state space generation. In G. Rozenberg, editor, *Applications and Theory of Petri Nets*, volume 483 of *LNCS*, pages 491–515. Springer, 1989.

[49] B. Wachter, D. Kroening, and J. Ouaknine. Verifying multi-threaded software with IMPACT. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 210–217. IEEE, 2013.

[50] C. Wang, Z. Yang, V. Kahlon, and A. Gupta. Peephole partial order reduction. In *TACAS*, volume 4963 of *LNCS*, pages 382–396. Springer, 2008.

[51] T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher. Model checking concurrent Linux device drivers. In R. E. K. Stirewalt, A. Egyed, and B. Fischer, editors, *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, pages 501–504. ACM, 2007.

[52] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Efficient stateful dynamic partial order reduction. In K. Havelund, R. Majumdar, and J. Palsberg, editors, *SPIN*, volume 5156 of *LNCS*, pages 288–305. Springer, 2008.