Partial Orders for Efficient BMC of Concurrent Software

Jade Alglave[1], Daniel Kroening[2], and Michael Tautschnig[2]

[1] University College London [2] University of Oxford

Abstract. The vast number of interleavings that a concurrent program can have is typically identified as the root cause of the difficulty of automatic analysis of concurrent software. Weak memory is generally believed to make this problem even harder. We address both issues by modelling programs' executions with partial orders rather than the interleaving semantics (SC). We implemented a software analysis tool based on these ideas. It scales to programs of sufficient size to achieve first-time formal verification of non-trivial concurrent systems code over a wide range of models, including SC, Intel x86 and IBM Power.

1 Introduction

Automatic analysis of concurrent programs is a practical challenge. Hardly any of the very few existing tools for concurrency will verify a thousand lines of code [12]. Most papers name the number of *thread interleavings* that a concurrent program can have as a reason for the difficulty. This view presupposes an execution model, namely *Sequential Consistency* (SC) [23], where an execution is a *total order* (more precisely an interleaving) of the instructions from different threads. The choice of SC as the execution model poses at least two problems.

First, the large number of interleavings modelling the executions of a program makes their enumeration intractable. *Context bounded* methods [29, 26, 21] (which are unsound in general) and *partial order reduction* [28, 14] can reduce the number of interleavings to consider, but still suffer from limited scalability. Second, modern multiprocessors (e.g., Intel x86 or IBM Power) serve as a reminder that SC is an inappropriate model. Indeed, the *weak memory models* implemented by these chips allow more behaviours than SC.

We address these two issues using *partial orders* to model executions. This differs radically from partial order reduction, which uses partial orders to reduce the number of total orders to examine. We also aim at practical verification of concurrent programs. Rarely have these two communities met. Exceptions are [31, 32]. We show that the explicit use of partial orders generalises these works to weak memory, without affecting efficiency.

Our method is as follows: we map a program to a formula consisting of two parts. The first conjunct describes the data and control flow for each thread of the program; the second conjunct describes the concurrent executions of these threads as partial orders. We prove that for any satisfying assignment of this

formula there is a valid execution w.r.t. our models; and conversely, any valid execution gives rise to a satisfying assignment of the formula.

Thus, given an analysis for sequential programs (the per-thread conjunct), we obtain an analysis for concurrent programs. For programs with bounded loops, we obtain a sound and complete model checking method. Otherwise, if the program has unbounded loops, we obtain an exhaustive analysis up to a given bound on loop unrollings, i.e., a bounded model checking method.

To experiment with our approach, we implement a *symbolic decision procedure* answering reachability queries over concurrent C programs w.r.t. a given memory model. We support a wide range of models, including SC, Intel x86 and IBM Power. To exercise our tool w.r.t. weak memory, we verify 4500 tests used to validate formal models against IBM Power chips [30, 25]. Our tool is the first to handle the subtle *store atomicity relaxation* [2] specific to Power and ARM. We show that mutual exclusion is not violated in a queue mechanism of the Apache HTTP server software. We confirm a bug in the worker synchronisation mechanism in PostgreSQL, and that adding two fences fixes the problem. We verify that the Read-Copy-Update mechanism of the Linux kernel preserves data consistency of the object it is protecting. We analyse all examples for a wide range of memory models, from SC to IBM Power via Intel x86.

We provide our proofs, the sources of our tool, our experimental logs and our benchmarks at `http://www.cprover.org/wpo`.

Related Work Implementing an executable version of the memory models is an important step of our work, but we go further than [15, 17, 33, 30, 25] by studying the validity of systems code in C (as opposed to assembly or toy languages) w.r.t. both a given memory model and a property.

Memory models roughly fall into two classes: operational and axiomatic. The operational style models executions via interleavings, with transitions accessing buffers or queues, in addition to the memory (as on SC). Thus this approach inherits the limitations of interleaving-based verification. For example, [5] (restricted to Sun Total Store Order, TSO) bounds the number of context switches. The methods of [20, 19] have, in the words of [24], "severely limited scalability". The technique of [24] scales to 771 lines but does not aim to be sound: the tool picks an invalid execution, repairs it, then iterates. Abdulla et al. [1] reason over finite state transition systems instead of programs.

Axiomatic specifications constrain relations over memory accesses. Our work relates the most to [7, 13, 31, 32], which use axiomatic specifications of SC to compose the distinct threads. CheckFence [7] models SC with total orders and transitive closure constraints; [31, 32] use partial orders like us; they note redundancies in their constraints, but do not explain them; our semantic foundations (Sec. 2) allow us both to explain and avoid them.

The encodings of [7, 31, 32] are $\mathcal{O}(\mathrm{N}^3)$ for N shared memory accesses *to any address*; [13] is quadratic, but in the number of threads times the number of per-thread transitions, which may include arbitrary many local accesses. Our encoding is $\mathcal{O}(\mathrm{M}^3)$, with M the maximal number of events for a *single address*.

P_0	P_1	P_2	P_3
(a) `r1` $\leftarrow$ `x`	(c) `r3` $\leftarrow$ `y`	(e) `x` $\leftarrow$ `1`	(f) `y` $\leftarrow$ `1`
(b) `r2` $\leftarrow$ `y`	(d) `r4` $\leftarrow$ `x`		
Allowed? `r1=1; r2=0; r3=1; r4=0;`			

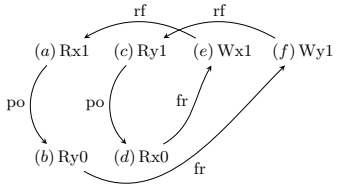

Fig. 1. Independent Reads of Independent Writes (**iriw**)

2 Context: Axiomatic Memory Model

We use the framework of [4], which provably embraces several architectures: SC [23], Sun TSO/x86 [27], PSO and RMO, Alpha, and a fragment of Power. We present this framework using a *litmus test*, shown in Fig. 1.

The keyword *allowed* asks if the architecture permits the outcome "`r1=1;r2=0; r3=1;r4=0`". This relates to the event graphs of this program, composed of relations over *read and write memory events*. A store instruction (e.g., `x` $\leftarrow$ `1` on P_2) corresponds to a write event ((e) Wx1), and a load (e.g., `r2` $\leftarrow$ `y` on P_0) to a read ((b) Ry0). The validity of an execution boils down to the absence of certain cycles in the event graph. Indeed, an architecture allows an execution when it represents a *consensus* amongst the processors. A cycle in an event graph is a potential violation of this consensus.

If a graph has a cycle, we check if the architecture *relaxes* some relations. The consensus ignores relaxed relations, hence becomes acyclic, i.e., the architecture allows the final state. In Fig. 1, on SC where nothing is relaxed, the cycle forbids the execution. Sun RMO relaxes the program order (po in Fig. 1) between reads, thus a forbidding cycle no longer exists for (a, b) and (c, d) are relaxed.

Executions Formally, an *event* is a read or a write memory access, composed of a unique identifier, a direction R for read or W for write, a memory address, and a value. We represent each instruction by the events it issues. In Fig. 1, we associate the store $x \leftarrow 1$ on processor P_2 to the event (e) Wx1.

We associate the program with an *event structure* $E \triangleq (\mathbb{E}, \mathsf{po})$, composed of its events $\mathbb{E}$ and the *program order* po, a per-processor total order. We write dp for the relation (included in po, the source being a read) modelling *dependencies* between instructions, e.g., an *address dependency* occurs when computing the address of a load or store from the value of a preceding load.

Then, we represent the *communication* between processors leading to the final state via an *execution witness* $X \triangleq (\mathsf{ws}, \mathsf{rf})$, which consists of two relations over the events. First, the *write serialisation* ws is a per-address total order on writes which models the *memory coherence* widely assumed by modern architectures. It links a write w to any write w' to the same address that hits the memory after w. Second, the *read-from* relation rf links a write w to a read r such that r reads the value written by w.

We include the writes in the consensus via the write serialisation. Unfortunately, the read-from map does not give us enough information to embed the

reads as well. To that aim, we derive the *from-read* relation fr from ws and rf. A read r is in fr with a write w when the write w' from which r reads hit the memory before w did. Formally, we have: $(r, w) \in \mathsf{fr} \triangleq \exists w', (w', r) \in \mathsf{rf} \wedge (w', w) \in \mathsf{ws}$.

In Fig. 1, the outcome corresponds to the execution on the right if each memory location and register initially holds 0. If `r1=1` in the end, the read (a) read its value from the write (e) on P_2, hence $(e, a) \in \mathsf{rf}$. If `r2=0`, the read (b) read its value from the initial state, thus before the write (f) on P_3, hence $(b, f) \in \mathsf{fr}$. Similarly, we have $(f, c) \in \mathsf{rf}$ from `r3=1`, and $(d, e) \in \mathsf{fr}$ from `r4=0`.

Relaxed or safe We model the weak behaviour permitted by modern architectures by some relations being *relaxed*, i.e., not included in the consensus. When a relation is not relaxed, we call it *safe*.

When a processor can read from its own store buffer [2] (the typical TSO/x86 scenario), we relax the internal read-from rfi. When two processors P_0 and P_1 can communicate privately via a cache (a case of *write atomicity* relaxation [2]), we relax the external read-from rfe, and call the corresponding write *non-atomic*. This is the main particularity of Power or ARM, and cannot happen on TSO/x86. We write grf_A for the read-from considered safe by A.

Some program-order pairs are relaxed (e.g., write-read pairs on x86), i.e., only a subset of po, written ppo_A, is guaranteed to occur in this order.

Architectures prevent weak behaviours using special *fence* (or *barrier*) instructions. We write ab_A for the relation induced by the fences of A. Following [4], the relation $\mathsf{fence} \subseteq \mathsf{po}$ induced by a fence is *non-cumulative* when it orders events surrounding the fence, i.e., fence is safe. The relation fence is *cumulative* when it makes writes atomic, e.g., by flushing caches. The relation fence is *A-cumulative* (resp. *B-cumulative*) if $\mathsf{rfe}; \mathsf{fence}$ (resp. $\mathsf{fence}; \mathsf{rfe}$) is safe. When stores are atomic (i.e., rfe is safe), e.g., on TSO, we do not need cumulativity.

Architectures An *architecture* A determines ppo_A, grf_A and ab_A, i.e., the relations embedded in the consensus. Following [4], we consider the write serialisation ws and the from-read relation fr to be always safe. SC relaxes nothing, i.e., rf and po are safe. TSO authorises the reordering of write-read pairs and store buffering (i.e., $\mathsf{po}_{\mathsf{WR}}$ and rfi are relaxed) but nothing else.

Finally, an execution (E, X) is *valid* on A when the three following conditions hold. 1. SC holds per address, i.e., the communication and the program order for accesses with same address $\mathsf{po\text{-}loc}$ are compatible: $\text{uniproc}(E, X) \triangleq \text{acyclic}(\mathsf{ws} \cup \mathsf{rf} \cup \mathsf{fr} \cup \mathsf{po\text{-}loc})$. 2. Values do not come out of thin air, i.e., there is no causal loop: $\text{thin}(E, X) \triangleq \text{acyclic}(\mathsf{rf} \cup \mathsf{dp})$. 3. There is a consensus on the safe relations: $\text{consensus}(E, X) \triangleq \text{acyclic}(\mathsf{ws} \cup \mathsf{grf}_A \cup \mathsf{fr} \cup \mathsf{ppo}_A \cup \mathsf{ab}_A)$. Formally: $\text{valid}_A(E, X) \triangleq \text{uniproc}(E, X) \wedge \text{thin}(E, X) \wedge \text{consensus}(E, X)$.

3 Symbolic Event Structures

For an architecture A and *one* execution witness X, the framework of Sec. 2 determines whether X is valid on A. To prove reachability of a program state, we need to reason about all its executions. To do so efficiently, we use symbolic

representations capturing all possible executions in a single constraint system. We then apply SAT or SMT solvers to decide if a valid execution exists for A, and, if so, get a satisfying assignment corresponding to an execution witness.

As said in Sec. 1, we build two conjuncts. The first one, ssa, represents the data and control flow per thread. The second, pord, captures the communications between threads (cf. Sec. 4). We include a reachability property in ssa; the program has a valid execution violating the property iff ssa $\wedge$ pord is satisfiable.

We mostly use *static single assignment form* (SSA) of the input program to build ssa (cf. [18] for details). This SSA variant augments each equation with a *guard*, which is the disjunction over all conjunctions of branching guards on paths to the assignment. To deal with concurrency, we use a fresh index for each occurrence of a given shared memory variable, resulting in a fresh symbol in the formula. CheckFence [7] and [31, 32] use a similarly modified encoding.

Together with ssa, we build a *symbolic event structure* (ses). As detailed below, it captures basic program information needed to build the second conjunct pord in Sec. 4. In Fig. 2, the formula ssa on top depicts the SSA form for Fig. 1. We print a column per thread, vertically following the control flow, but it forms a single conjunction. Each occurrence of a program variable carries its SSA index as a subscript. Each occurrence of the shared memory variables x and y has a unique SSA index. Here we omit the guards, as this program does not use branching or loops.

main	P_0	P_1	P_2	P_3
$x_0 = 0$				
$\wedge\ y_0 = 0$	$\wedge\ r1_0^1 = x_1$	$\wedge\ r3_0^2 = y_2$	$\wedge\ x_3 = 1$	$\wedge\ y_3 = 1$
	$\wedge\ r2_0^1 = y_1$	$\wedge\ r4_0^2 = x_2$		
$\wedge$ prop				
$(i_0)\,\mathrm{W}x x_0$				
$(i_1)\,\mathrm{W}y y_0$				
	$(a)\,\mathrm{R}x x_1$	$(c)\,\mathrm{R}y y_2$	$(e)\,\mathrm{W}x x_3$	$(f)\,\mathrm{W}y y_3$
	$(b)\,\mathrm{R}y y_1$	$(d)\,\mathrm{R}x x_2$		

Fig. 2. The formula ssa for **iriw** (Fig. 1) with prop $= (r1_0^1 = 1 \wedge r2_0^1 = 0 \wedge r3_0^2 = 1 \wedge r4_0^2 = 0)$, and its ses (guards omitted since all true)

From SSA to symbolic event structures A symbolic event structure (ses) $\gamma \triangleq (\mathbb{S}, \mathsf{po})$ is a set $\mathbb{S}$ of *symbolic events* and a *symbolic program order* po. A symbolic event holds a *symbolic value* instead of a concrete one as in Sec. 2. We define $\mathrm{g}(e)$ to be the Boolean guard of a symbolic event e, which corresponds to the guard of the SSA equation as introduced above. We use these guards to build the executions of Sec. 2: a guard evaluates to true if the branch is taken, false otherwise. The symbolic program order $\mathsf{po}(\gamma)$ gives a list of symbolic events per thread of the program. The order of two events in $\mathsf{po}(\gamma)$ gives the program order in a concrete execution if both guards are true.

Note that $\mathsf{po}(\gamma)$ is an implementation-dependent linearisation of the branching structure of a thread, induced by the path merging applied while constructing the SSA form. We write $\mathsf{po\text{-}br}(\gamma)$ for this branching structure (i.e., the unlinearised symbolic program order induced by the control flow graph).

We build the ses γ alongside the SSA form, as follows. Each occurrence of a shared program variable on the right-hand side of an assignment becomes a

symbolic read, with the SSA-indexed variable as symbolic value, and the guard is taken from the SSA equation. Similarly, each occurrence of a shared program variable on the left-hand side becomes a *symbolic write*. Fences do not affect memory states in a sequential setting, hence do not appear in SSA equations. We simply add a fence event to the ses when we see a fence. We take the order of assignments per thread as program order, and mark thread spawn points.

The bottom of Fig. 2 gives the ses of **iriw**. Each column represents the symbolic program order. We use the same notation as for the events of Sec. 2, but values are SSA symbols. Guards are omitted again, since all trivially true. We depict a thread spawn by starting the program order in the appropriate row.

From symbolic to concrete event structures To relate to the models of Sec. 2, we *concretise* symbolic events. A satisfying assignment to $\mathsf{ssa} \wedge \mathsf{pord}$, as computed by a SAT or SMT solver, induces, for each symbolic event, a concrete value (if it is a read or a write) and a valuation of its guard (for both accesses and fences). A valuation V of the symbols of ssa includes the values of each symbolic event. Since guards are formulas that are part of ssa, V allows us to evaluate the guards as well. For a valuation V, we write $\mathrm{conc}(e_s, \mathsf{V})$ for the concrete event corresponding to e_s, if there is one, i.e., if $\mathrm{g}(e_s)$ evaluates to true under V.

The concretisation of a set $\mathbb{S}$ of symbolic events is a set $\mathbb{E}$ of concrete events, as in Sec. 2, s.t. for each $e \in \mathbb{E}$ there is a symbolic version e_s in $\mathbb{S}$. We write $\mathrm{conc}(\mathbb{S}, \mathsf{V})$ for this concrete set $\mathbb{E}$. The concretisation $\mathrm{conc}(\mathsf{r}_s, \mathsf{V})$ of a symbolic relation r_s is the relation $\{(x, y) \mid \exists (x_s, y_s) \in \mathsf{r}_s . x = \mathrm{conc}(x_s, \mathsf{V}) \wedge y = \mathrm{conc}(y_s, \mathsf{V})\}$.

Given an ses γ, $\mathrm{conc}(\gamma, \mathsf{V})$ is the event structure (cf. Sec. 2), whose set of events is the concretisation of the events of γ w.r.t. V, and whose program order is the concretisation of $\mathsf{po}(\gamma)$ w.r.t. V. For example, the graph of Fig. 1 (erasing the rf and fr relations) is a concretisation of the ses of **iriw** (cf. Fig. 2).

4 Symbolic Encodings

For an architecture A and an ses γ, we need to represent the communications (i.e., rf, ws and fr) and the weak memory relations (i.e., $\mathsf{ppo}_A, \mathsf{grf}_A$ and ab_A) of Sec. 2. We encode them as a formula pord, s.t. $\mathsf{ssa} \wedge \mathsf{pord}$ is satisfiable iff there is an execution valid on A violating the property encoded in ssa. We avoid transitive closures to obtain a small number of constraints. We start with an overview of our approach, then describe how we encode partial orders, and finally detail the encoding for the relations rf and ppo of Sec. 2 (we omit $\mathsf{ws}, \mathsf{fr}, \mathsf{ab}$ for brevity).

Overview We present our approach on **iriw** (Fig. 1) and its ses γ (Fig. 2). In Fig. 1, we represent only the execution leading to the (non-SC) final state. In this section, we generate constraints representing all the executions of **iriw** on a given architecture. We give these constraints for the address x in Fig. 3 in the SC case (for brevity we skip y, analogous to x). Weakening the architecture removes some constraints: for Power, we omit the (rf-grf) and (ppo) constraints.

In Fig. 3, each symbol c_{ab} is a *clock constraint*, representing an ordering between the events a and b. A variable s_{wr} represents a read-from between the

(rf-val x)	$(s_{i_0a} \Rightarrow x_1 = x_0) \wedge (s_{i_0d} \Rightarrow x_2 = x_0) \wedge$ $(s_{ea} \Rightarrow x_1 = x_3) \wedge (s_{ed} \Rightarrow x_2 = x_3)$
(rf-grf x)	$(s_{i_0a} \Rightarrow c_{i_0a}) \wedge (s_{ea} \Rightarrow c_{ea}) \wedge$ $(s_{i_0d} \Rightarrow c_{i_0d}) \wedge (s_{ed} \Rightarrow c_{ed})$
(rf-some x)	$(s_{i_0a} \vee s_{ea}) \wedge (s_{i_0d} \vee s_{ed})$
(ws x)	$\neg c_{i_0e} \Rightarrow c_{ei_0}$
(fr x)	$((s_{i_0a} \wedge c_{i_0e}) \Rightarrow c_{ae}) \wedge ((s_{i_0d} \wedge c_{i_0e}) \Rightarrow c_{de}) \wedge$ $((s_{ea} \wedge c_{ei_0}) \Rightarrow c_{ai_0}) \wedge ((s_{ed} \wedge c_{ei_0}) \Rightarrow c_{di_0})$
(ppo main)	$c_{i_0i_1}$ (ppo P_0) c_{ab} (ppo P_1) c_{cd}

Fig. 3. Partial order constraints for address x in Fig. 1 on SC

write w and the read r. The constraints of Fig. 3 represent the preserved program order (cf. Sec. 4.2), e.g., on SC or TSO the read-read pairs (a, b) on P_0 (ppo P_0) and (c, d) on P_1 (ppo P_1), but nothing on Power. We generate constraints for the read-from (cf. Sec. 4.1), for example (rf-some x); the first conjunct $s_{i_0a} \vee s_{ea}$ expresses that the read a on P_0 can read either from the initial write i_0 or from the write e on P_2. The selected read-from pair also implies equalities of the values written and read (rf-val x): for instance, s_{i_0a} implies that x_1 equals the initialisation x_0. The constraints for write serialisation and from-read are specified as (ws x) and (fr x); (ws y) and (fr y) are analogous. As there are no fences in **iriw**, we do not generate any fence constraints (cf. Sec. 4.3).

We represent the execution of Fig. 1 as follows. For (e, a) and $(i_0, d) \in \mathsf{grf}$, we have the constraint $s_{ea} \Rightarrow c_{ea}$ and $s_{i_0d} \Rightarrow c_{i_0d}$ in (rf-grf x). This means that a reads from e (as witnessed by s_{ea}), and that we record that e is ordered before a in grf (as witnessed by c_{ea}); *idem* for d and i_0. To represent $(d, e) \in \mathsf{fr}$, we pick the appropriate constraint in (fr x), namely $(s_{i_0d} \wedge c_{i_0e}) \Rightarrow c_{de}$. This reads "if d reads from i_0 and i_0 is ordered before e (in ws, because i_0 and e are two writes to x), then d is ordered before e (in fr)."

Together with (ppo P_0) and (ppo P_1), these constraints represent the execution in Fig. 1. We cannot find a satisfying assignment of these constraints, as this leads to both a before b (by (ppo P_0)) and b before a (by (fr y), (rf-grf y), (ppo P_1), (fr x) and (grf x)). On Power, however, we neither have the ppo nor the grf constraints, hence we can find a satisfying assignment.

Symbolic partial orders We associate each symbolic event x of an ses γ with a *clock* variable clock_x (cf. [22, 31]) ranging over the naturals. For two events x and y, we define the Boolean *clock constraint* as $c_{xy} \triangleq (\mathrm{g}(x) \wedge \mathrm{g}(y)) \Rightarrow \mathsf{clock}_x < \mathsf{clock}_y$ ("$<$" being less-than over the integers). We encode a relation r over the symbolic events of γ as the formula $\phi(\mathsf{r}) \triangleq \bigwedge_{(x,y)\in\mathsf{r}} c_{xy}$.

Let C be a valuation of the clocks of the events of γ. Let V be a valuation of the symbols of the formula ssa associated to γ. As noted in Sec. 3, V gives us concrete values for the events of γ, and allows us to evaluate their guards. We show below that (C, V) satisfies $\phi(\mathsf{r})$ iff the concretisation of r w.r.t. V is acyclic, provided that this relation has *finite prefixes*.

input: γ, A **output:** $C_{\text{wf}}, C_{rf}, C_{grf}$

1 $\mathsf{reads} := \{(\alpha, \{r_1 \dots r_n\}) \mid r_i \text{ is read} \wedge \text{addr}(r_i) = \alpha\}$
2 $\mathsf{writes} := \{(\alpha, \{w_1 \dots w_n\}) \mid w_i \text{ is write} \wedge \text{addr}(w_i) = \alpha\}$
3 $C_{\mathsf{rf}} := \emptyset$; $C_{\mathsf{grf}} := \emptyset$
4 **foreach** α *s.t.* $\exists R, W.(\alpha, R) \in \mathit{reads} \wedge (\alpha, W) \in \mathit{writes}$ **do**
5 **foreach** $r \in R$ **do**
6 $\mathsf{rf\_some} := \emptyset$
7 **foreach** $w \in W$ **do**
8 **if** $(r, w) \notin \mathit{po}(\gamma)$ **then**
9 $\mathsf{rf\_some} := \mathsf{rf\_some} \cup \{s_{wr}\}$
10 $C_{\text{wf}} := C_{\text{wf}} \cup \{s_{wr} \Rightarrow (\text{g}(w) \wedge \mathsf{val}(r) = \mathsf{val}(w))\}$
11 $C_{\mathsf{rf}} := C_{\mathsf{rf}} \cup \{s_{wr} \Rightarrow c_{wr}\}$
12 **if** (w, r) *not relaxed on* A *and* $\text{tid}(w) \neq \text{tid}(r)$ **then**
13 $C_{\mathsf{grf}} := C_{\mathsf{grf}} \cup \{s_{wr} \Rightarrow c_{wr}\}$
14 $C_{\text{wf}} := C_{\text{wf}} \cup \{\text{g}(r) \Rightarrow \bigvee_{s \in \mathsf{rf\_some}} s\}$

Algorithm 1: Constraints for read-from

A prefix of x in a relation r is a (possibly infinite) list $S = [x_0, x_1, x_2, \dots]$ s.t. $x = x_0$ and for all i, $(x_{i+1}, x_i) \in \mathsf{r}$. The relation r has finite prefixes if for each x, there is a bound $l \in \mathbb{N}$ to the cardinality of the prefixes of x in r. We write $\text{card}(S)$ for the cardinality of a list $S = [x_0, x_1, x_2, \dots]$, i.e., $\text{card}(S) \triangleq \text{card}(\{x \mid \exists i. x = x_i\})$. We write $\text{pref}(\mathsf{r}, x)$ for the set of prefixes of x in r. Formally, r has finite prefixes when $\forall x. \exists l. \forall S \in \text{pref}(\mathsf{r}, x). \text{card}(S) < l$. In our proofs and in Alg. 2 we denote the concatenation of two lists S_1 and S_2 by $S_1 + + S_2$.

Our first lemma shows the acyclicity of a concrete relation equivalent to the satisfiability of the formula encoding this relation symbolically:

Lemma 1. (C, V) *satisfies* $\phi(r)$ *iff* $\text{conc}(r, V)$ *is acyclic and has finite prefixes.*

The formula $\phi(\mathsf{r}_1 \cup \mathsf{r}_2)$ is equivalent to $\phi(\mathsf{r}_1) \wedge \phi(\mathsf{r}_2)$. Thus we encode unions of relations, e.g., $\mathsf{ghb}_A \triangleq \mathsf{ws} \cup \mathsf{fr} \cup \mathsf{grf}_A \cup \mathsf{ppo}_A \cup \mathsf{ab}_A$, as the conjunction of their respective encodings. By Lem. 1, the acyclicity of ghb_A corresponds to the satisfiability of $\phi(\mathsf{ghb}_s)$, where ghb_s is a symbolic encoding of ghb_A.

We build $\phi(\mathsf{ghb}_s)$ as the conjunction of the formulas $\phi(\mathsf{r})$, for r being a symbolic encoding of ws, fr, grf_A, ppo_A and ab_A. We now present the encodings of $\mathsf{ppo}_A, \mathsf{rf}$ and ab, and omit the others for brevity. We define auxiliaries over symbolic events: $\text{tid}(e)$ is the thread identifier of e, $\text{addr}(e)$ the memory address read from or written to (e.g., x for $(e)\,\text{Rxy}$), and $\text{val}(e)$ its (symbolic) value. Each algorithm outputs constraints, whose conjunction we add to pord.

4.1 Read-from

For an architecture A and an ses γ, Alg. 1 encodes the read-from (resp. global read-from) as the set of constraints C_{rf} (resp. C_{grf}). Following Sec. 2, we add constraints to C_{grf} depending on: first, the relation being within one thread or

between distinct threads (derivable from $\mathrm{tid}(w)$ and $\mathrm{tid}(r)$); second, whether A exhibits store buffering, store atomicity relaxation, or both.

To form the potential read-from pairs, we introduce a free Boolean variable s_{wr} for each (w, r) of write and read to the same address (line 9), unless it contradicts program order (line 8) (as this violates uniproc). Following Sec. 2, each read must read from some write. We ensure this at line 14, by gathering in C_{wf}, for a given r, all the potential read-from s_{wr} collected in $\mathsf{rf\_some}$.

If s_{wr} evaluates to true (i.e., r reads from w), we record the *value constraint* $\mathrm{val}(r) = \mathrm{val}(w)$ in the set C_{wf} (line 10). The constraint added to C_{rf} is such that only if s_{wr} evaluates to true, the clock constraint c_{wr} is enforced (line 11). If (w, r) is not relaxed on A, we also add its clock constraint c_{wr} to C_{grf} (line 13).

We write $(w, r) \in \mathrm{WR}_\alpha$ when w writes to an address α and r reads from the same α, and $\mathsf{prf} \triangleq \{(w, r) \in \bigcup_\alpha \mathrm{WR}_\alpha \mid (r, w) \notin \mathsf{po}(\gamma)\}$. We write $\mathsf{rf}(\gamma)$ for the set $\{(w, r) \in \mathsf{prf} \mid s_{wr}\}$, and $\mathsf{grf}(\gamma)$ for $\mathsf{grf}_A(\mathsf{rf}(\gamma))$. Note that we build the external global read-from ($\mathsf{grfe}(\gamma)$) only, i.e., between two events from distinct threads. We compute the internal one as part of ppo_A, in Alg. 2.

Given an ses γ, Alg. 1 outputs $C_{\mathsf{rf}}, C_{\mathsf{grf}}$ and C_{wf}. Let WR be a valuation of the s_{wr} variables of γ. We write $\mathsf{inst}(r, \mathsf{WR})$ for r where WR instantiates the s_{wr} variables. Alg. 1 gives the clock constraints encoding grf:

Lemma 2. *$(\mathsf{C}, \mathsf{V}, \mathsf{WR})$ satisfies $\bigwedge_{c \in C_{\mathrm{wf}} \cup C_{\mathsf{grf}}} c$ iff (C, V) satisfies*
i) for all r s.t. $\mathrm{g}(r)$ is true, there is w s.t. $(w, r) \in \mathsf{inst}(\mathsf{rf}(\gamma), \mathsf{WR})$ and
ii) for all $(w, r) \in \mathsf{inst}(\mathsf{rf}(\gamma), \mathsf{WR})$, $\mathrm{g}(w)$ is true and $\mathrm{val}(w) = \mathrm{val}(r)$ and
iii) $\bigwedge_{(w,r) \in \mathsf{inst}(\mathsf{grfe}(\gamma), \mathsf{WR})} c_{wr}$.

4.2 Preserved program order

For an architecture A and an ses γ, Alg. 2 encodes the preserved program order as the set C_{ppo}. We reuse the notation ppo_A for the function collecting non-relaxed pairs in symbolic program order. Unlike in Sec. 2, the non-relaxed pairs in symbolic program order also include the internal global internal read-from, internal write serialisation, internal from-read, and the orderings due to Power's `isync` fence. We generate these constraints here, rather than in each of the communication encodings, to limit the redundancies. We write $\mathsf{ppo}_A(\gamma)$ for $\mathsf{ppo}_A(\mathsf{po\text{-}br}(\gamma))$, or only ppo_A if γ is clear from the context.

Alg. 2 avoids building redundant transitive closure constraints, taking into account the guards of events: for two events e_1, e_2, we build a constraint iff $(e_1, e_2) \in \mathsf{ppo}_A(\gamma)$. If, e.g., $\mathsf{ppo}_A(\mathsf{po\text{-}br}) = \mathsf{po\text{-}br}$ (on SC), Alg. 2 creates constraints only for neighbouring events in $\mathsf{po\text{-}br}(\gamma)$ in each control flow branch.

As SSA and loop unrolling yield $\mathsf{po}(\gamma)$ (i.e., lists of symbolic events per thread) rather than $\mathsf{po\text{-}br}(\gamma)$ (the corresponding DAG), we cannot construct C_{ppo} by analysing control flow branches of the program.

To build ppo_A, Alg. 2 uses the variable chains, a list of pairs (y, T). For a given y, its companion set T contains the events x occurring before y in ${\mathsf{ppo}_A}^+$ together with a formula r that characterises all paths of ${\mathsf{ppo}_A}^+$ between x and

input: γ, A **output:** C_{ppo}

```
1  C_ppo := ∅; foreach S ∈ po(γ) ∧ S ≠ ∅ do
2    S = [e]++S' ∩ {e | e is not fence}
3    chains := [(e, ∅)]; R := true
4    foreach e' ∈ S' do
5      T' := ∅
6      foreach (e'', T'') ∈ chains s.t. there is no r s.t.
                 (e'', r) ∈ T' and ((g(e') ∧ g(e'') ∧ R) ⇒ r) do
7        r_{e''e'} := not_relax A γ (e'', e')
8        if r_{e''e'} is satisfiable then
9          C_ppo := C_ppo ∪ {r_{e''e'} ⇒ c_{e''e'}}
10         T' := T' ∪ {(e'', r_{e''e'})}
11         foreach (e, r) ∈ T'' do
12           if ∃r'.(e, r') ∈ T' then
13             R := R ∧ (ρ ⇔ r' ∨ (r_{e''e'} ∧ r))
14             T' := {(e, ρ)} ∪ T' \ (e, r')
15           else T' := {(e, r_{e''e'} ∧ r)} ∪ T'
16     chains := [(e', T')]++[chains]
```

Algorithm 2: Constraints for preserved program order

y. We build r from formulas $r_{e''e'}$ asserting that $(e'', e') \in \mathsf{ppo}_A$, describing individual steps (e'', e') of a path between x and y.

We compute the formula $r_{e''e'}$ at line 7, using the function not_relax. Given an ses γ and a pair (e'', e'), not_relax $A\ \gamma\ (e'', e')$ returns a formula $r_{e''e'}$ expressing the condition under which (e'', e') is not relaxed. Depending on the specification of the architecture A, not_relax uses the direction of events only, determines data- and control dependencies using a definition-use data flow analysis, or considers Power's `isync` fence. In all cases, a conjunction of the guards of the instructions on the data flow path is returned.

For a given e', we initialise its companion set T' at line 5, then increment it in lines 10–15. In line 14, we use fresh variables ρ constrained in the formula R (line 13) to avoid repeating sub-formulas, as is standard in, e.g., CNF encodings [8]. In line 7 we compute the condition $r_{e''e'}$ for (e'', e') not being relaxed on A for each e'' in chains (unless skipped for transitivity, see below). We generate the constraint $r_{e''e'} \Rightarrow c_{e''e'}$ iff $r_{e''e'}$ is satisfiable (line 9), i.e., (e'', e') is not relaxed on A. We now characterise the output of Alg. 2:

Lemma 3. *Alg. 2 outputs* $\{r_{xy} \Rightarrow c_{xy} \mid (x, y) \in \mathit{ppo}_A\}$.

Since the r_{xy} are guard conditions, we just need to evaluate the guards to evaluate them. We show that Alg. 2 gives the clock constraints encoding ppo:

Lemma 4. (C, V) *satisfies* $\bigwedge_{c \in C_{ppo}} c$ *iff it satisfies* $\bigwedge_{(x,y) \in \mathit{ppo}_A} c_{xy}$.

4.3 Memory fences and cumulativity

Alg. 3 encodes the fence orderings as the set $C_{\mathsf{ab}'}$. A fence s potentially induces orderings over all (e, e') s.t. e is in po before s and e' after s, which is quadratic

input: γ, A **output:** $C_{ab'}$

1 $C_{\mathsf{ab}'} := \emptyset$; **foreach** $S \in \mathit{po}(\gamma) \wedge S \neq \emptyset$ **do**
2 $\quad \mathsf{fences} := \{s \mid s \in S \wedge \text{s is fence}\}$
3 $\quad$ **foreach** $e \in S \setminus \mathit{fences}$ **do**
4 $\quad\quad$ **foreach** $s \in \mathit{fences}$ **do**
5 $\quad\quad\quad$ **if** $(e, s) \in \mathit{po}(\gamma)$ **then**
6 $\quad\quad\quad\quad C_{\mathsf{ab}'} := C_{\mathsf{ab}'} \cup \{\mathrm{g}(s) \Rightarrow c_{es}\}$
7 $\quad\quad\quad\quad$ **if** A *is not store atomic* **then**
8 $\quad\quad\quad\quad\quad$ **foreach** (w, e) *being a w-r pair s.t.* $\mathrm{addr}(w) = \mathrm{addr}(e)$ *and* $\mathrm{tid}(w) \neq \mathrm{tid}(e)$ **do**
9 $\quad\quad\quad\quad\quad\quad C_{\mathsf{ab}'} := C_{\mathsf{ab}'} \cup \{(\mathrm{g}(s) \wedge s_{we}) \Rightarrow c_{ws}\}$
10 $\quad\quad\quad$ **else** $C_{\mathsf{ab}'} := C_{\mathsf{ab}'} \cup \{\mathrm{g}(s) \Rightarrow c_{se}\}$
11 $\quad\quad\quad\quad$ **if** A *is not store atomic* **then**
12 $\quad\quad\quad\quad\quad$ **foreach** (e, r) *being a w-r pair s.t.* $\mathrm{addr}(e) = \mathrm{addr}(r)$ *and* $\mathrm{tid}(e) \neq \mathrm{tid}(r)$ **do**
13 $\quad\quad\quad\quad\quad\quad C_{\mathsf{ab}'} := C_{\mathsf{ab}'} \cup \{(\mathrm{g}(s) \wedge s_{er}) \Rightarrow c_{sr}\}$

Algorithm 3: Constraints for memory fences

in the number of events in po for each fence. Cumulativity constraints depend on read-from, and again these are paired with all events before or after (in po) a fence. We alleviate this with the fence events (see below). The implementation supports x86's `mfence` and Power's `sync`, `lwsync` and `isync`. We present only x86's `mfence` and Power's `sync`, for brevity.

We test at line 5 for each pair (e, s) s.t. e is a non-fence event and s is fence if (e, s) is in program order. The result of this test determines the non-cumulative and cumulative constraints. For non-cumulativity, if e is before (resp. after) s in program order, Alg. 3 produces at line 6 the clock constraint c_{es} (resp. c_{se} at line 10). If A relaxes store atomicity, we build cumulativity constraints. For A-cumulativity, Alg. 3 adds at line 9 the constraint $s_{we} \Rightarrow c_{ws}$, for each (w, e) s.t. e is in po before the fence s, and e reads from the write w. The constraint reads "if $\mathrm{g}(s)$ is true (i.e., the fence is concretely executed) and if s_{we} is true (i.e., e reads from w), then c_{ws} is true (i.e., there is a global ordering, due to the fence s, from w to s)". All other constraints, i.e., the actual ordering of w before some event e' in po *after* s, follow by transitivity. We handle B-cumulativity in a similar way, given in lines 12 and 13.

As `lwsync` does not order write-read pairs [4], we avoid creating a constraint c_{wr} between a write w and a read r separated by an `lwsync`. To do so, we use two distinct clock variables clock_s^r and clock_s^w for an `lwsync` s.

Given an ses γ, Alg. 3 outputs C_{ab}. We let $\mathsf{ab}(\gamma)$ be the symbolic version of ab in Sec. 2. We only prove this encoding sound w.r.t. Sec. 2, as C_{ab} is more fine-grained than ab. Yet we prove our overall encoding complete in Thm. 1.

Lemma 5. *If* (C, V, WR) *satisfies* $\bigwedge_{c \in C_{ab}} c$ *then* (C, V) *satisfies* $\bigwedge_{(e_1,e_2) \in \mathit{inst}(\mathsf{ab}(\gamma), WR)} c_{e_1 e_2}$.

4.4 Soundness and completeness of the encoding

Given an architecture A and a program, the procedure of Sec. 3 and Sec. 4 outputs a formula $\mathsf{ssa} \wedge \mathsf{pord}$ and an ses γ. This formula provably encodes the executions of this program valid on A and violating the property encoded in ssa in a sound and complete way. Given an ses γ, we write ϕ for $\bigwedge_{c \in C_{\mathsf{ppo}} \cup C_{\mathsf{grf}} \cup C_{\mathsf{wf}} \cup C_{\mathsf{ws}} \cup C_{\mathsf{ab}'}} c$:

Theorem 1. *ϕ is satisfiable iff there is a valuation V of the symbols of ssa, and a well formed X s.t. $\mathsf{ghb}_A(\mathrm{conc}(\gamma, \mathsf{V}), X)$ is acyclic and has finite prefixes.*

5 Experimental Results

Our experiments suggest that our technique scales enough to verify non-trivial concurrent systems code, e.g. the worker-synchronisation logic of the relational database PostgreSQL, socket-handover in the Apache httpd, and the core API of the Read-Copy-Update (RCU) mutual exclusion code from Linux 3.2.21.

We implement our technique within the bounded model checker CBMC [10], using a SAT solver as an underlying decision procedure. We estimate the overhead of our method in two ways. First, we pass the benchmarks with a single, fixed interleaving to sequential CBMC. Our implementation performs comparably to sequential CBMC, as Fig. 4 shows (rows "sequential" and "concurrent"). Second, we compare to ESBMC [11], which also implements bounded model checking, but uses interleaving-based techniques.

	Fibo.	Litmus	PgSQL	RCU	Apache
LOC	41	50.9	5412	5834	28864
unroll	5	none	2	bounded	5
tot. addr	2	11.8	6	3	8
tot. shared	45	58.7	233	107	88
same addr	11	3.7	72	4	5
all constr	308	874	3762	90	160
most costly	rf (178)	ab (342)	rf (1868)	rf (33)	rf (49)
sequential	0.3 s	0.1 s	4.1 s	0.8 s	1.7 s
concurrent	3.3 s	0.2 s	90.0 s	1.0 s	2.8 s
ESBMC	13.8 s	609.8 s	t/o	parse err	parse err

Fig. 4. Facts about all examples

In Fig. 4, we gather facts about all examples: the Fibonacci example from [6] with N=5, 4500 litmus tests (see below), the worker synchronisation in PostgreSQL, RCU, and fdqueue in Apache httpd. For each we give the number of lines of code (LOC), the number of distinct memory addresses "tot. addr" (including unused shared variables), the total number of shared accesses "tot. shared", the maximal number of accesses to a single address "same addr", the total number of constraints "all constr" and the relation with the most costly encoding, in terms of the number of constraints generated. We give the loop unrolling bounds "unroll": we write "none" when there is no loop, and "bounded" when the loops in the program are natively bounded.

The total number of shared accesses is on average 13 times the maximal number of accesses to a single address. The most costly constraint is usually the read-from, or the barriers, which build on read-from. The time needed by our tool to analyse a program grows with the total number of constraints generated. ESBMC is 4 times slower than our tool on Fibonacci, 3050 times slower on the litmus tests, times out on PostgreSQL, and cannot parse RCU and Apache.

	CBMC	CBMC	CBMC	CheckFence	ESBMC	Poirot	SatAbs	Threader
	SC	TSO	Power	SC, TSO	SC	SC	SC	SC
F	CE $N = 300$	CE $N = 220$	CE $N = 240$	conv err	CE $N = 10$	fails $N \geq 1$	V $N = 3$	t/o $N = 1$
L	100%	100%	100%	18%	34%	47%	100%	8%
P	V	V	CE	conv err	t/o	parse err	t/o	n/a
Pf	V	V	V	conv err	t/o	parse err	t/o	n/a
R	V	V	V	conv err	parse err	parse err	ref err	n/a
A	V	V	V	conv err	parse err	parse err	aborts	n/a

Fig. 5. Comparison of all tools on all examples (time out 30 mins)

Other tools There are very few tools for verifying concurrent C programs, even on SC [12]. For weak memory, existing techniques are restricted to TSO, and its siblings PSO and RMO [7, 20, 19, 5, 1, 24]. Not all of them have been implemented, and only few handle systems code given as C programs. We have submitted a program transformation based approach that generalizes to Power to ESOP [3], and use the technique presented in the present paper in there.

We tried 5 ANSI-C model checkers: SatAbs, a verifier based on predicate abstraction [9]; ESBMC; Threader, a thread-modular verifier [16]; and Poirot, which implements a context-bounded translation to sequential programs [21]. We also tried CheckFence [7].

In Fig. 5, we compare all tools on all examples: F for Fibonacci, L for the litmus tests, P for PostgreSQL with its bug, Pf for our fix, R for RCU and A for Apache. For L, P, R and A, the bounds are as in Fig. 4; for Pf we take the one of P. For F we try the maximal N that the tool can handle within the time out of 30 mins. For each tool, we give the model below. When a tool verifies an example we write "V"; when it finds a counterexample we write "CE".

Fibonacci All tools, except ESBMC, SatAbs and ours, fail to analyse Fibonacci. Poirot claims the assertion violated for any N, which is not the case for $1 \leq \mathrm{N} \leq 5$. SatAbs does not reach beyond $\mathrm{N} = 4$. Our tool handles more than $\mathrm{N} = 300$, which is 30 times more loop unrolling than ESBMC, within the same amount of time.

Litmus tests We analyse 4500 tests (generated by the diy tool [4] exposing weak memory artefacts. For example, **iriw** (Fig. 1) can only be reached on RMO (by reordering the reads) or on Power (*idem*, or because the writes are non-atomic).

We convert these tests into C code, of 50 lines on average, involving 2 to 4 threads. Despite the small size of the tests, they prove challenging to verify, as Fig. 5 shows: most tools, except SatAbs and ours, give wrong results or fail in other ways on a vast majority of tests, even for SC. For each tool we give the average percentage of correct results over all models.

PostgreSQL Developers observed a regression failure on a PowerPC machine, and later identified the memory model as possible culprit: the processor could delay a write by a thread until after a token signalling the end of this thread's work had been set. Our tool confirmed the bug, and proved a patch we proposed. A detailed description of the problem is in [3].

Read-Copy-Update (RCU) is a synchronisation mechanism of the Linux kernel, introduced in version 2.5. Writers to a concurrent data structure prepare a fresh component (e.g., list element), then replace the existing component by adjusting the pointer variable linking to it. Clean-up of the old component is delayed until there is no process reading. Readers can rely on very lightweight (thus fast) lock-free synchronisation only. The protection of reads against concurrent writes is fence-free on x86, and uses only a light-weight fence (`lwsync`) on Power. We verify the original implementation of the 3.2.21 kernel for x86 (5824 lines) and Power (5834 lines) in less than 1 s, using a harness that asserts that the reader will not obtain an inconsistent version of the component. On Power, removing the `lwsync` makes the assertion fail.

Apache httpd is the most widely used HTTP server software. It supports a broad range of concurrency APIs distributing incoming requests to a pool of workers. The fdqueue module (28864 lines) is the central part of this mechanism, which implements the hand-over of a socket together with a memory pool to an idle worker. The implementation uses a central, shared queue for this purpose. Shared access is synchronised using an integer keeping track of the number of idle workers, which is updated via architecture-dependent compare-and-swap and atomic decrement operations. Hand-over of the socket and the pool and wake-up of the idle thread is then coordinated by means of a conventional, heavy-weight mutex and a signal. We verify that hand-over guarantees consistency of the payload data passed to the worker in 2.45 s on x86 and 2.8 s on Power.

6 Conclusion

Our experiments demonstrate the scalability of our method for programs with bounded loops. Our proofs suggest that this is not limited to bounded loops, but impracticable as it involves infinite structures. We hope that this work opens up new possibilities for over-approximation for programs with unbounded loops.

References

1. Abdulla, P.A., Atig, M.F., Chen, Y.F., Leonardsson, C., Rezine, A.: Counter-example guided fence insertion under TSO. In: TACAS (2012)
2. Adve, S.V., Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial. IEEE Computer (1995)
3. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation, submitted to ESOP 2013, available at `http://www.cprover.org/etaps/`
4. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Fences in Weak Memory Models (Extended Version). In: FMSD (2012)
5. Atig, M.F., Bouajjani, A., Parlato, G.: Getting Rid of Store-Buffers in the Analysis of Weak Memory Models. In: CAV (2011)
6. Beyer, D.: Competition on software verification - (SV-COMP). In: TACAS (2012)

7. Burckhardt, S., Alur, R., Martin, M.: CheckFence: Checking consistency of concurrent data types on relaxed memory models. In: PLDI (2007)
8. Chambers, B., Manolios, P., Vroon, D.: Faster sat solving with better cnf generation. In: DATE (2009)
9. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: TACAS (2005)
10. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS (2004)
11. Cordeiro, L., Fischer, B.: Verifying multi-threaded software using SMT-based context-bounded model checking. In: ICSE (2011)
12. D'Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. TCAD (2008)
13. Ganai, M., Gupta, A.: Efficient Modeling of Concurrent Systems in BMC. In: SPIN (2008)
14. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer (1996)
15. Gopalakrishnan, G., Yang, Y., Sivaraj, H.: QB or not QB: An Efficient Execution Verification Tool for Memory Orderings. In: CAV (2004)
16. Gupta, A., Popeea, C., Rybalchenko, A.: Threader: A Constraint-Based Verifier for Multi-Threaded Programs. In: CAV (2011)
17. Huynh, Q., Roychoudhury, A.: A memory sensitive checker for C#. In: FM (2006)
18. Kroening, D., Clarke, E., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: DAC (2003)
19. Kuperstein, M., Vechev, M., Yahav, E.: Partial-Coherence Abstractions for Relaxed Memory Models. In: PLDI (2011)
20. Kuperstein, M., Vechev, M., Yahav, E.: Automatic inference of memory fences. In: FMCAD (2010)
21. Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. In: FMSD (2009)
22. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. CACM (1978)
23. Lamport, L.: How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. IEEE Trans. Comput. (1979)
24. Liu, F., Nedev, N., Prisadnikov, N., Vechev, M., Yahav, E.: Dynamic synthesis for relaxed memory models. In: PLDI (2012)
25. Mador-Haim, S., Maranget, L., Sarkar, S., Memarian, K., Alglave, J., Owens, S., Alur, R., Martin, M., Sewell, P., Williams, D.: An Axiomatic Memory Model for Power Multiprocessors. In: CAV (2012)
26. Musuvathi, M., Qadeer, S.: Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In: PLDI (2005)
27. Owens, S., Sarkar, S., Sewell, P.: A better x86 model: x86-TSO. In: TPHOL (2009)
28. Peled, D.: All from one, one for all. In: CAV (1993)
29. Qadeer, S., Rehof, J.: Context-Bounded Model Checking of Concurrent Software. In: TACAS (2005)
30. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding Power Multiprocessors. In: PLDI (2011)
31. Sinha, N., Wang, C.: Staged Concurrent Program Analysis. In: FSE (2010)
32. Sinha, N., Wang, C.: On Interference Abstractions. In: POPL (2011)
33. Torlak, E., Vaziri, M., Dolby, J.: MemSAT: Checking Axiomatic Specifications of Memory Models. In: PLDI (2010)