Partial Orders for Efficient Bounded Model Checking of Concurrent Software*

Jade Alglave[1], Daniel Kroening[2], and Michael Tautschnig[3]

[1] University College London
[2] University of Oxford
[3] Queen Mary, University of London

Abstract. The number of interleavings of a concurrent program makes automatic analysis of such software very hard. Modern multiprocessors' execution models make this problem even harder. Modelling program executions with partial orders rather than interleavings addresses both issues: we obtain an efficient encoding into integer difference logic for bounded model checking that enables first-time formal verification of deployed concurrent systems code. We implemented the encoding in the CBMC tool and present experiments over a wide range of memory models, including SC, Intel x86 and IBM Power. Our experiments include core parts of PostgreSQL, the Linux kernel and the Apache HTTP server.

1 Introduction

Automatic analysis of concurrent programs is 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 [14]. Most papers name the number of *thread interleavings* of a concurrent program as a reason for the difficulty. This view presupposes an execution model, i.e., *Sequential Consistency* (SC) [25], where an execution is a *total order* (more precisely an interleaving) of the instructions from different threads. This 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 [31, 23] (unsound in general) and *partial order reduction* [29, 17] 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. For example, a processor can commit a write first to a store buffer, then to a cache, and finally to memory. While the write is in transit through buffers and caches, a read can occur before the value is actually available to all processors from the memory.

We address both issues by using *partial orders* to model executions, an established theoretical tradition [30, 36, 6]. We aim at bug finding and practical

* Supported by SRC/2269.002, EPSRC/H017585/1, EU FP7 STREP PINCETTE, ARTEMIS/VeTeSS, and ERC/280053.

verification of concurrent programs [11, 23, 13] – where partial orders have hardly ever been considered. Notable exceptions are [33, 34] (but we do not have access to an implementation), forming with [10] the closest related work. We show that the explicit use of partial orders generalises these works to concurrency at large, from SC to weak memory. On the technical side, partial orders permit a drastic reduction of the formula size over the use of total orders. Our experiments confirm that this reduction is desirable, as it *increases scalability* by lowering the memory footprint. Furthermore our experiments show, contrasting folklore belief, that the verification time is *hardly affected by the choice of memory model*.

We emphasise that we study *hardware* memory models as opposed to software ones. We believe that verification of concurrent software is still bound to hardware models. Indeed, concurrent systems software is racy on purpose (see our experiments in Sec. 5). Yet, software memory models either banish or give an undefined semantics to racy programs [27, 8]. Thus, to give a semantics to concurrent programs, we lift the hardware models to the software level.

In addition to the immediate support of weak memory models, we find that partial orders permit a *very natural and non-intrusive* extension of bounded model checking (BMC) of software to concurrent programs: SAT- and SMT-based BMC builds a formula that describes the data and control flow of a program. For concurrent programs, we do so for each thread of the program. We *add* a conjunct that 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 memory models; and conversely, any valid execution gives rise to a satisfying assignment of the formula. We impose no additional bound on context switches, and SC is merely a particular case of our method.

To experiment with our approach, we implement a *symbolic decision procedure for partial orders* in CBMC [12], enabling BMC of concurrent C programs w.r.t. a given memory model for systems code. For SC, we show the efficiency and competitiveness of our approach on the benchmarks of the TACAS 2013 software verification competition [7]. We furthermore support a wide range of weak memory models, including Intel x86 and IBM Power. To exercise our tool on these models, we prove and disprove safety properties in more than 5800 loop-free *litmus tests* previously used to validate formal models against IBM Power chips [32, 26]. Our tool is the first to handle the subtle *store atomicity relaxation* [1] specific to Power and ARM. We furthermore perform first-time verification of *core components of systems software*. 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 show that the Read-Copy-Update (RCU) mechanism of the Linux kernel preserves data consistency of the object it is protecting. For all examples we perform the analysis for SC, Intel x86, as well as IBM Power. For each of the systems examples we either succeed in bug finding (for PostgreSQL) or can soundly limit the number of loop iterations required to show the desired property using BMC (RCU is loop-free and the loop in Apache is stateless).

We provide the sources of our tool, our benchmarks and log files, and a long version [3] of the paper, with proof sketches, at http://www.cprover.org/wmm.

P_0	P_1
(a) `x` ← `1`	(c) `y` ← `1`
(b) `r1` ← `y`	(d) `r2` ← `x`
Allowed? `r1=0`; `r2=0`	

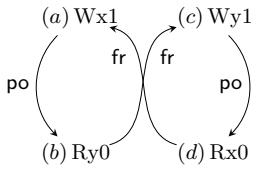

Fig. 1. Store Buffering (**sb**)

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

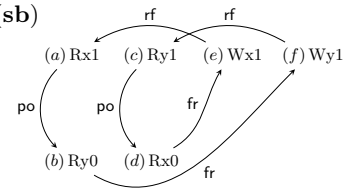

Fig. 2. Independent Reads of Independent Writes (**iriw**)

2 Executions as Partial Orders

Our symbolic decision procedure builds on the framework of [4], which was originally conceived to model weak memory semantics. *Relations* over *read and write memory events* are at the core of this framework.

We introduce this framework on *litmus tests*, as shown in Fig. 1. On the left-hand side we show a multi-threaded program. The shared variables `x` and `y` are initialised to zero. A store (e.g., `x` ← `1` on P_0) gives rise to a write event ((a) Wx1), and a load (e.g., `r1` ← `y` on P_0) to a read event ((b) Ry0). The property of interest is whether there exists an execution of the program such that the final state is `r1=0` and `r2=0`. To determine this, we study the *event graph*, given on the right-hand side of the figure. An architecture allows an execution when it represents an order of all events consistent across all processors. We call this order *global happens before*. A cycle in an event graph is a violation thereof.

For memory models weaker than SC, the architecture possibly *relaxes* some of the relations contributing to this cycle. Such a relaxation makes the graph acyclic, which implies that the architecture allows the final state. In SC, nothing is relaxed, thus the cycle in Fig. 1 forbids the execution. Intel x86 relaxes the program order (po in Fig. 1) between writes and reads, thus the forbidding cycle no longer exists, and the given final state is observed.

Formalisation 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. 2, we associate the store `x` ← `1` on processor P_2 with the event (e) Wx1.

The set of events $\mathbb{E}$ and program order, po, form an *event structure*[4] $E \triangleq (\mathbb{E}, \mathsf{po})$; po is a per-processor total order over $\mathbb{E}$. We write $\mathsf{dp} \subseteq \mathsf{po}$ for the relation modelling *dependencies* between instructions, e.g., an *address dependency* occurs when computing an address to access from the value of a preceding load.

[4] We use this term to remain consistent with [4], but note that it differs from G. Winskel's event structures [36].

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 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 s.t. r reads the value written by w. We distinguish the internal read-from rfi (between events on the same processor) from the external rfe (between events on distinct processors).

Given a pair of writes $(w_0, w_1) \in \mathsf{ws}$ s.t. $(w_0, r) \in \mathsf{rf}$, we have w_0 globally happening before w_1 by ws and r reading from w_0 by rf. To ensure that r does not read from w_1, we impose that r globally happens before w_1 in the *from-read* relation fr from ws and rf. A read r is in fr with a write w_1 when the write w_0 from which r reads hit the memory before w_1 did. Formally, we have: $(r, w_1) \in \mathsf{fr} \triangleq \exists w_0, (w_0, r) \in \mathsf{rf} \wedge (w_0, w_1) \in \mathsf{ws}$.

In Fig. 2, the final state corresponds to the execution on the right if each memory location initially holds 0. If `r1=1` in the end, the read (a) obtained its value from the write (e) on P_2, hence $(e, a) \in \mathsf{rf}$. If `r2=0` in the end, the read (b) obtained 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 weak memory effects by *relaxing* subrelations of program order or read-from. Thereby [4] provably embraces several models: SC [25], Sun TSO (i.e., x86 [28]), PSO and RMO, Alpha, and a fragment of Power.

We model reads occurring in advance, as described in the introduction, by subrelations of the read-from rf being *relaxed*, i.e., not included in global happens before. When a processor can read from its own store buffer [1] (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 [1]), we relax the external read-from rfe, and call the corresponding write *non-atomic*. This is a particularity of Power or ARM, and cannot happen on TSO/x86. Some program-order pairs may be relaxed by an architecture (defined below) A (e.g., write-read pairs on x86, and all but dp ones on Power), i.e., only a subset of po is guaranteed to occur in this order. This subset is the *preserved program order*, ppo_A. When a relation may not be relaxed, we call it *safe*.

An architecture A may provide *fence* (or *barrier*) instructions to prevent non-SC behaviours. Following [4], the relation $\mathsf{fence}_A \subseteq \mathsf{po}$ induced by a fence is *non-cumulative* when it only orders certain pairs of events surrounding the fence, i.e., fence_A is safe. The relation fence_A is *cumulative* when it makes writes atomic, e.g., by flushing caches. This amounts to making sequences of external read-from and fences ($\mathsf{rfe}; \mathsf{fence}_A$ or $\mathsf{fence}_A; \mathsf{rfe}$) safe, even though rfe alone would not be safe for A. We denote the union of fence_A and additional cumulativity by ab_A.

Architectures An *architecture* A determines which relations are safe, i.e., embedded in global happens before. Following [4], we always consider the write serialisation ws and the from-read relation fr safe. We denote the safe subset of read-from, i.e., the read-from relation globally agreed on by all processors, by

grf_A. SC relaxes nothing, i.e., rf and po are safe. TSO authorises the reordering of write-read pairs and store buffering but nothing else. Fences are safe by design.

Finally, an execution (E, X) is *valid* on A when three conditions hold: 1. SC holds per address, i.e., communication and 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 exists a linearisation of events in global happens before, i.e., $\mathsf{ghb}_A(E, X) \triangleq \mathsf{ws} \cup \mathsf{grf}_A \cup \mathsf{fr} \cup \mathsf{ppo}_A \cup \mathsf{ab}_A$ does not form a cycle. Formally:

$$\text{valid}_A(E, X) \triangleq \text{uniproc}(E, X) \wedge \text{thin}(E, X) \wedge \text{acyclic}(\mathsf{ghb}_A(E, X))$$

From the validity of executions we deduce a comparison of architectures: We say that an architecture A_2 is *stronger* than another one A_1 when the executions valid on A_2 are valid on A_1. Equivalently we would say that A_1 is *weaker* than A_2. Thus, SC is stronger than any other architecture discussed above.

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 safety properties of programs, however, we need to reason about all possible executions of the program. 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 whether a valid execution exists for A, and, if so, get a satisfying assignment corresponding to an execution witness. If no such satisfying assignment exists, the program is proved safe for the given loop unwinding depth.

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 $\mathsf{ssa} \wedge \mathsf{pord}$ is satisfiable.

We mostly use *static single assignment form* (SSA) of the input program to build ssa (cf. [21] for details), as common in symbolic execution and bounded model checking. In our SSA variant, each equation is augmented with a *guard*: the guard is the disjunction over all conjunctions of branching guards on paths to the assignment. To counter exponential-sized guards, control-flow join points result in simplified guards. 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.

main	P_0	P_1	P_2	P_3
$x_0 = 0$				
$\wedge\ y_0 = 0$	$\wedge\ r1_0^1 = x_1$	$\wedge\ r3_0^2 = y_2$	$\wedge\ x_3 = 1$	$\wedge\ y_3 = 1$
	$\wedge\ r2_0^1 = y_1$	$\wedge\ r4_0^2 = x_2$		
$\wedge\ \mathsf{prop}$				

$(i_0)\,\mathrm{W}x x_0$
$(i_1)\,\mathrm{W}y y_0$

$(a)\,\mathrm{R}x x_1$ $(b)\,\mathrm{R}y y_1$ $(c)\,\mathrm{R}y y_2$ $(d)\,\mathrm{R}x x_2$ $(e)\,\mathrm{W}x x_3$ $(f)\,\mathrm{W}y y_3$

Fig. 3. The formula ssa for **iriw** (Fig. 2) with $\mathsf{prop} = (r1_0^1 = 1 \wedge r2_0^1 = 0 \wedge r3_0^2 = 1 \wedge r4_0^2 = 0)$, and its ses (guards omitted since all true)

We add additional equality constraints (cf. Sec. 4.2) to pord to subsequently constrain these. CheckFence [10] and [33, 34] use a similar encoding.

Together with ssa, we build a *symbolic event structure* (ses), which captures program information needed to build the second conjunct pord in Sec. 4. Fig. 3 illustrates this section: the formula ssa on top corresponds to the ses beneath.

The top of Fig. 3 gives ssa for Fig. 2. We print a column per thread, vertically following the control flow, but it forms a single conjunction. Each 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 neither uses branching nor loops.

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.

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.

At the bottom of Fig. 3, we give the ses of **iriw**. Each column represents the symbolic program order, per thread. We use the same notation as for the events of Sec. 2, but values are SSA symbols. Guards are omitted again. We depict the thread spawn events 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 model V of $\mathsf{ssa} \wedge \mathsf{pord}$, as computed by a satisfiability 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).

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 concretise a symbolic relation similarly. Given an ses γ, $\mathrm{conc}(\gamma, \mathsf{V})$ is the event structure 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. 2 (erasing the rf and fr relations) concretises the ses of **iriw** (cf. Fig. 3).

4 Symbolic Decision Procedure for Partial Orders

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 first describe how we encode partial orders in general, and then discuss the construction and optimisations for each of the above partial orders: the key challenge is to avoid transitive closures in order to obtain a small number of constraints.

4.1 Symbolic Representation of Partial Orders

We associate each symbolic event x of an $\mathsf{ses}\ \gamma$ with a unique *clock* variable clock_x (cf. [24, 33]) 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 the usual order on natural numbers). We encode a relation r over the symbolic events of γ as the formula $\phi(\mathsf{r})$ defined as the conjunction of the clock constraints c_{xy} for all $(x, y) \in \mathsf{r}$, i.e., $\phi(\mathsf{r}) \triangleq \bigwedge_{(x,y)\in\mathsf{r}} c_{xy}$. 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., ghb_A) as the conjunction of their respective encodings.

Let C be a valuation of the clocks of γ. Let V be a valuation of the formula ssa associated to γ. One can show 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.

Overview We first present our approach on **iriw** (Fig. 2) and its ses γ (Fig. 3), and give the construction of constraints for this example. The algorithms implementing the general case for each of the relations are presented in the extended version of this paper [3], which also includes proofs of correctness for each of the algorithms.

(ppo main)	$c_{i_0 i_1}$ (ppo P_0) c_{ab} (ppo P_1) c_{cd}
(rf-val x)	$(s_{i_0 a} \Rightarrow x_1 = x_0) \wedge (s_{i_0 d} \Rightarrow x_2 = x_0) \wedge$ $(s_{ea} \Rightarrow x_1 = x_3) \wedge (s_{ed} \Rightarrow x_2 = x_3)$
(rf-grf x)	$(s_{i_0 a} \Rightarrow c_{i_0 a}) \wedge (s_{ea} \Rightarrow c_{ea}) \wedge$ $(s_{i_0 d} \Rightarrow c_{i_0 d}) \wedge (s_{ed} \Rightarrow c_{ed})$
(rf-some x)	$(s_{i_0 a} \vee s_{ea}) \wedge (s_{i_0 d} \vee s_{ed})$
(ws x)	$\neg c_{i_0 e} \Rightarrow c_{e i_0}$
(fr x)	$\big((s_{i_0 a} \wedge c_{i_0 e}) \Rightarrow c_{ae}\big) \wedge \big((s_{i_0 d} \wedge c_{i_0 e}) \Rightarrow c_{de}\big) \wedge$ $\big((s_{ea} \wedge c_{e i_0}) \Rightarrow c_{a i_0}\big) \wedge \big((s_{ed} \wedge c_{e i_0}) \Rightarrow c_{d i_0}\big)$

Fig. 4. Partial order constraints for x in Fig. 2 on SC

In Fig. 2, we represent only one possible execution, namely the one corresponding to the (non-SC) final state of the test. 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. 4 in the SC case (for brevity we skip y, analogous to x). As we explain below in detail, weakening the architecture removes some constraints: for example, for Power, we do not include the (rf-grf) and (ppo) constraints. For TSO, all constraints are the same as for SC.

Each symbol c_{ab} of Fig. 4 is a clock constraint, as introduced in Sec. 4.1 above, and thus represents an ordering between the events a and b. A variable s_{wr} represents a read-from between the write w and the read r.

The constraints of Fig. 4 first represent the preserved program order, 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, for example (rf-some x); the first conjunct $s_{i_0 a} \vee s_{ea}$ concerns the read a on P_0. This means

that a 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, $s_{i_0 a}$ implies that x_1 equals the initialisation x_0. The architecture-independent 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 memory fence constraints.

Valid Executions We represent the execution of Fig. 2 as follows. For (e, a) and $(i_0, d) \in \mathsf{grf}_A$, we have the constraint $s_{ea} \Rightarrow c_{ea}$ and $s_{i_0 d} \Rightarrow c_{i_0 d}$ 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_A (as witnessed by c_{ea}); *idem* for d and i_0. The constraint $(s_{i_0 d} \wedge c_{i_0 e}) \Rightarrow c_{de}$ in (fr x) represents $(d, e) \in \mathsf{fr}$. It 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. 2. 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.

4.2 Encoding the Axiomatic Memory Model

We now present a systematic account of the encoding of partial orders required for global happens before, as defined in Sec. 2. The constraints for uniproc and thin are only added when not redundant with ghb for a given architecture. When required, their construction follows the same rules as defined for ghb below, but the constraints use distinct sets of clock variables.

Preserved Program Order As described in Sec. 3, symbolic execution, including loop unrolling, yields lists of symbolic events per thread gathered in $\mathsf{po}(\gamma)$. We encode the required ppo_A via clock constraints derived from $\mathsf{po}(\gamma)$ in the set C_{ppo}. Let $S \in \mathsf{po}(\gamma)$ be a list of events of some thread. We require for any events $e_1, e_2 \in S$ in this order in S, a clock constraint $c_{e_1 e_2}$ to appear in C_{ppo} when:

1. (e_1, e_2) is safe for the architecture A. This test is architecture-specific. For SC, all pairs are safe. IBM Power only guarantees instruction dependencies to be respected, i.e., $c_{e_1 e_2} \in C_{\mathsf{ppo}}$ iff $(e_1, e_2) \in \mathsf{dp}$.
2. There is a control-flow path from e_1 to e_2. This avoids adding clock constraints that are trivially satisfied. Such a case arises when their precondition $\mathrm{g}(e_1) \wedge \mathrm{g}(e_2)$ is false, i.e., the guards cannot both be true.
3. The constraint $c_{e_1 e_2}$ is not in the transitive closure of C_{ppo}. Consider an event e_3 s.t. $c_{e_1 e_2}, c_{e_2 e_3} \in C_{\mathsf{ppo}}$. If all three events share the same guards, i.e., stem from the same control-flow branch, the constraint $c_{e_1 e_3}$ is redundant, as it is in the transitive closure of existing constraints. This is an optimisation.

For the ses γ of **iriw**, we have $\mathsf{po}(\gamma) = \{[i_0, i_1], [a, b], [c, d], [e], [f]\}$. Given an architecture, we thus build the set C_{ppo} as follows: IBM Power only maintains dependencies, which do not exist for the instructions of **iriw**. *Thus C_{ppo} is empty.* RMO relaxes read-read pairs, resulting in $C_{\mathsf{ppo}} = \{c_{i_0 i_1}\}$. For PSO and stronger architectures, read-read pairs are maintained, thus the constraints (ppo P_0) and (ppo P_1) are added as well.

Read-From We encode read-from (resp. safe 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; second, whether A exhibits store buffering, store atomicity relaxation, or both.

The framework of [4] summarised in Sec. 2 follows a post-mortem reasoning with known fixed values, whereas we need to consider all possible executions. Thus, in contrast to Sec. 2, we need to take two additional facts into account:

1. For any read r there are several candidate writes w to the same address. For each such potential pair (w, r) we introduce a free Boolean variable s_{wr}. The set of eligible writes is determined by collecting all writes to the same address as r, with the exception of writes in program order after r (such writes violate the uniproc check of Sec. 2). Each candidate pair contributes a constraint $s_{wr} \Rightarrow c_{wr}$ to C_{grf}, if the pair is safe for the selected architecture A.
   By Sec. 2, rf maps each read to exactly one write. This would induce an exactly-one (pigeon hole) constraint over all s_{wr} for each read. Such constraints can be challenging for CDCL-style SAT solvers; moreover these are redundant in our case: the exclusivity follows from write serialisation and from-read (cf. [3]). We thus instead add a disjunction over all s_{wr} to C_{rf}.
2. For any pair $(w, r) \in \mathsf{rf}$ encoded as s_{wr} we need to ensure that the guard of the write w is true as well as equality over values, because our modified SSA form encoded in ssa has free symbols for each shared memory access. We add such constraints $s_{wr} \Rightarrow \mathrm{g}(w) \wedge x_w = x_r$ to C_{rf}.

For **iriw**, Fig. 4 contains the above encoding in (rf-grf x), (rf-some x) and (rf-val x). For instance, a may read either from the initial write i_0 or from the write e on P_2. These possible pairs are encoded in (rf-grf x), and will be added for all architectures other than Power, which relaxes store atomicity (therefore C_{grf} remains empty on Power). We then enforce that at least one of these read-from pairs exists via the disjunction in (rf-some x). The selected read-from pair also implies equalities of the values written and read (rf-val x): for instance, $s_{i_0 a}$ implies that x_1 equals the initialisation x_0.

Write Serialisation We encode ws as the set of constraints C_{ws}. By definition, ws is a total order over writes to a given address. We implement the totality by ensuring that for two writes $w \neq w'$ to the same address either $c_{ww'}$ or $c_{w'w}$ holds, i.e., $\mathsf{clock}_w \neq \mathsf{clock}_{w'}$. Note that if $(w, w') \in \mathsf{po}$, then necessarily $c_{ww'}$ must hold by uniproc. For **iriw** we have $\mathsf{writes} = \{(\mathrm{x}, \{i_0, e\}), (\mathrm{y}, \{i_1, f\})\}$, and the constraint (ws x) for x.

From-Read We encode fr as the set of constraints C_{fr}. Recall that $(r, w) \in \mathsf{fr}$ means $\exists w'.(w', r) \in \mathsf{rf} \wedge (w', w) \in \mathsf{ws}$. We implement the existential quantifier by a disjunction: $\bigvee_{w' \text{is write}} (w', r) \in \mathsf{rf} \wedge (w', w) \in \mathsf{ws} \Rightarrow c_{rw} \in C_{\mathsf{fr}}$. Our implementation expands the disjunction into multiple constraints added to C_{fr}: for each write w' we add a constraint $s_{w'r} \wedge c_{w'w} \Rightarrow c_{rw}$ to C_{fr}. Observe that $s_{w'r}$ encodes $(w', r) \in \mathsf{rf}$, and $c_{w'w}$ encodes $(w', w) \in \mathsf{ws}$.

For **iriw** and x, we obtain the constraint (fr x), where $(s_{i_0 a} \wedge c_{i_0 e}) \Rightarrow c_{ae}$, reads "if $s_{i_0 a}$ is true (i.e., if a reads from i_0), and if $c_{i_0 e}$ is true (i.e., $(i_0, e) \in \mathsf{ws}$) then c_{ae} is true (i.e., a is in fr before e)."

4.3 Memory Fences and Cumulativity

As noted in Sec. 2, to counter the effects of weak memory models, architectures provide fence instructions. We collect their encoding the set C_{ab} which will always be empty on SC. Our implementation supports x86's `mfence` and Power's `sync`, `lwsync` and `isync`. We handle `isync` as part of ppo_A. We first present x86's `mfence` and Power's `sync`, then `lwsync`.

For the non-cumulative part, a fence orders all events in program order before the fence instruction with events in program order after the fence (for `lwsync` this excludes write-read pairs). A naive encoding thereof results in a quadratic number of clock constraints for each fence. For cumulativity, a similar concern applies. To alleviate this, we introduce *fence events*.

Assume fences between the read-read pairs of P_0 and P_1 of **iriw**, and thus fence events s_0 and s_1. We then have $\mathsf{po}(\gamma) = \{[i_0, i_1], [a, s_0, b], [c, s_1, d], [e], [f]\}$. We will instantiate these fences as `sync` and `lwsync` in the following paragraphs and describe the resulting symbolic encodings.

Fences `mfence` and `sync` For all $e, s \in S$ for some $S \in \mathsf{po}(\gamma)$, with s being a fence event, we first build constraints for *non-cumulativity*: if e is before (resp. after) s in program order, we add c_{es} to C_{ab} (resp. c_{se}).

In **iriw** with the additional fences mentioned above instantiated with `sync`, we generate c_{as_0} (resp. c_{cs_1}) for the event a (resp. c) in po before the fence s_0 (resp. s_1) on P_0 (resp. P_1). We generate c_{s_0b} (resp. c_{s_1d}) for b (resp. d), in po after the fence s_0 (resp. s_1) on P_0 (resp. P_1).

If stores are not atomic, we build *cumulativity constraints*. For A-cumulativity, we add 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 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.

As Power relaxes store atomicity, the `sync` fences between the read-read pairs of **iriw** create A-cumulativity constraints, namely for s_0 (and analogous ones for s_1): $(s_{i_0a} \Rightarrow c_{i_0s_0}) \wedge (s_{ea} \Rightarrow c_{es_0})$.

Fence `lwsync` As `lwsync` does not order write-read pairs (cf. Sec. 2), we need to 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. This avoids the wrong transitive constraint c_{wr} implied by c_{ws} and c_{sr}. Fig. 5 illustrates this setup: the write-read pair (w_1, r_2) will not be ordered by any of the constraints, but all other pairs are ordered.

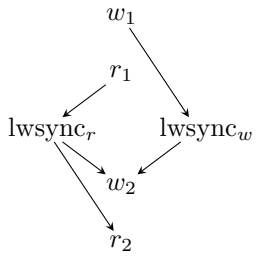

Fig. 5. `lwsync`'s constraints

To create a clock constraint, we then pick one or both clock variables, as follows. If e is a read, the clock constraint is $\mathsf{clock}_e < \mathsf{clock}_s^r$ when e is before s (or $\mathsf{clock}_s^r < \mathsf{clock}_e$ if e is after). If e is a write preceding s, the clock constraint is $\mathsf{clock}_e < \mathsf{clock}_s^w$. Finally, if e is a write after s, the clock constraint is the conjunction $(\mathsf{clock}_s^w < \mathsf{clock}_e) \wedge (\mathsf{clock}_s^r < \mathsf{clock}_e)$.

In **iriw**, if we use `lwsync` instead of `sync` as discussed above, we obtain the following constraints: $(\mathsf{clock}_a < \mathsf{clock}^r_{s_0}) \wedge (\mathsf{clock}^r_{s_0} < \mathsf{clock}_b) \wedge (s_{i_0 a} \Rightarrow \mathsf{clock}_{i_0} < \mathsf{clock}^w_{s_0}) \wedge (s_{ea} \Rightarrow \mathsf{clock}_e < \mathsf{clock}^w_{s_0})$. These constraints will *not* order the writes i_0 or e with the read b, because i_0 and e are ordered w.r.t. to $\mathsf{clock}^w_{s_0}$, but b is only ordered w.r.t. the distinct $\mathsf{clock}^r_{s_0}$. This corresponds to the fact that placing `lwsync` fences in **iriw** does not forbid the non-SC execution.

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. Proving this requires showing that any assignment to the system corresponds to a valid execution of the program, and vice versa. This result requires three steps, one for uniproc, one for thin and one for the acyclicity of ghb. By lack of space, we show only the last one. Given an ses γ, we write ϕ for $\bigwedge_{c \in C_{\mathsf{ppo}} \cup C_{\mathsf{grf}} \cup C_{\mathsf{ws}} \cup C_{\mathsf{fr}} \cup C_{\mathsf{ab}}} c$:

Thm. 1. *The formula $\mathsf{ssa} \wedge \phi$ is satisfiable iff there are V, a valuation of ssa, and a well formed X s.t. $\mathsf{ghb}_A(\mathrm{conc}(\gamma, \mathsf{V}), X)$ is acyclic and has finite prefixes.*

To decide the satisfiability of ϕ, we can use any solver supporting propositional combinations of integer difference logic constraints. The procedure reveals the concrete executions, as expressed by Thm. 1.

5 Experimental Results

We implemented the encoding described above in the bounded model checker CBMC [12], built with the SAT solver MiniSat 2.2.0 as back-end decision procedure. We study the efficiency on standard benchmarks, show the support and correct implementation of a broad range of memory models on litmus tests, and demonstrate the real-world fitness on widely deployed systems code. The full raw data of our results are available on our web page http://www.cprover.org/wmm.

As is elaborated below, the limited availability of proposed related techniques as well as, where available, unfitness to process real-world C programs, restricts what we can conclude about pre-existing techniques. Yet we find that our technique is scalable enough to verify non-trivial, real-world concurrent systems code, including the worker-synchronisation logic of the relational database PostgreSQL, code for socket-handover in the Apache httpd, and the core API of the Read-Copy-Update (RCU) mutual exclusion code from Linux 3.2.21.

In Tab. 1 we present key facts of our benchmarks: we give the average over all 34 examples of the Software Verification Competition 2013 [7]; similarly we use averages for our 5800 litmus tests; the last three columns provide data for the systems code we study: the worker synchronisation in PostgreSQL, RCU, and fdqueue in Apache httpd. For each we give the number of lines of code (LOC), and the loop unwinding bound used in the experiments (loop unwind) – "none" when there is no loop, and "bounded" when the loops in the program are natively bounded. The number of equations in the resulting ssa is listed as SSA size. We

further list key characteristics concerning the symbolic encoding of partial orders: the number of distinct shared memory addresses (#addresses), the total number of shared memory accesses plus fence events (#events), the maximal number of accesses to a single address (max/addr), the total number of constraints required for the partial order encoding (#constraints), and the relation accounting for the largest fraction of constraints (most costly).

	SV-COMP	Litmus	PgSQL	RCU	Apache
LOC	798.3	51.2	5412	5834	28864
loop unwind	6	none	1	bounded	5
SSA size	716	80.2	245	161	1027
#addresses	5.8	6.6	5	7	9
#events	160.2	40.9	74	37	140
max/addr	53.6	3.8	17	4	93
#constraints	3576.4	362.0	1089	393	1137
most costly	rf (1587)	rf (81.3)	rf (306)	rf (67)	rf (247)

Table 1. Statistics about all examples

Observe that the total number of shared accesses is on average more than 5.7 times the maximal number of accesses to a single address, making a strong case for the use of partial orders: *the number of constraints generated for total orders would thus be larger by a factor of* 5^3*, i.e., two orders of magnitude more costly.* The most costly constraint is usually the encoding of read-from.

Other tools Few tools verify concurrent C programs, even on SC [14]. In particular, the implementation of [33, 34] is not available. For weak memory, solutions were restricted to TSO, and its siblings PSO and RMO [10, 5, 22], of which only CheckFence [10] is available and able to handle C programs. With [2] we were the first to present a program transformation-based approach for weaker models.

In addition to CheckFence, we tried five further tools, covering a range of techniques for verifying C programs on SC: SatAbs [11], based on predicate abstraction; ESBMC [13], a bounded model checker exploring interleavings with partial order reduction; Threader [19], a thread-modular verifier; CSeq [15] and Poirot, both implementing a context-bounded translation to sequential programs [23]. Poirot and CheckFence, however, could only parse litmus tests.

5.1 Efficiency: SV-COMP'13

We use the 34 concurrency benchmarks from https://svn.sosy-lab.org/software/sv-benchmarks/trunk/c/ to compare the efficiency of the partial order based approach to existing tools. We mirror the competition settings, with a time-out of 15 minutes and a memory bound of 15 GB. Fig. 6 (left) depicts the overall performance of SatAbs (7 solved correctly w.r.t. the rules of SV-COMP), CSeq (12), ESBMC (15), Threader (29), and CBMC, which solves all 34 instances correctly. The programs where CBMC takes more than a few seconds have a large number of shared memory array operations, which challenge the underlying SAT solver even for the SSA part. We primarily compare the run-time with ESMBC, as it also performs bounded model checking, but analyses interleavings (total orders). Fig. 6 (right, logarithmically scaled) shows that CBMC outperforms ESBMC on all examples. Comparing to Threader is less meaningful, as Threader is abstraction-based and does not impose loop bounds. We note that Threader wrongly marks one benchmark (qrcu) as safe, whereas CBMC correctly

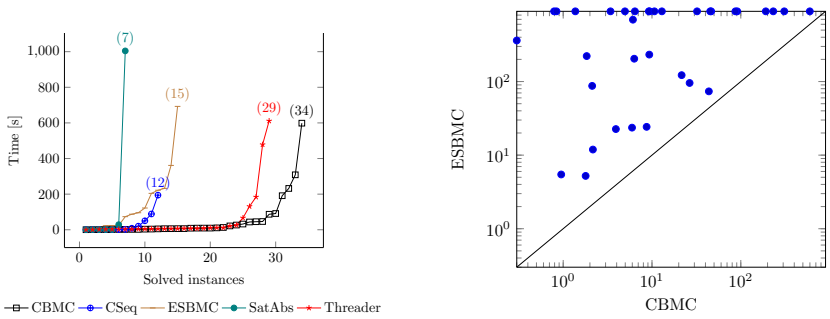

Fig. 6. Comparison of efficiency on SV-COMP'13 benchmarks

reports a counterexample. No other tool had been able to analyse the program, thus Threader's result had been deemed correct for SV-COMP'13. This was raised with the competition organiser and the developers of Threader.

5.2 Weak Memory Models: Litmus Tests

We analyse 5803 tests exposing weak memory artefacts, e.g., instruction reordering, store buffering, store atomicity relaxation. These tests are assembly programs with a non-SC final state, but reachable on a weaker model, generated by the diy tool [4]. For example, **iriw** (Fig. 2) 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 51 lines on average, involving 2 to 6 threads. Despite the small size of the tests, they prove challenging to verify: as we showed in [2], most tools, except Blender [22], SatAbs and CBMC, give wrong results or fail in other ways on a vast majority of tests, even for SC, when run for up to 15 minutes. CBMC, however, takes 0.21 s on average to correctly compute the result for each of the memory models SC, TSO, PSO, RMO, Alpha, and Power. No test requires more than 0.7 s, with the exception of the test CO-IRIW, which takes up to 3.7 s (it yields 2450 partial order constraints). CheckFence reported violated properties on all tests, even on SC (where all properties of these tests hold). Blender, which supports only PSO, took 0.6 s on average, and at most 9.7 s. With [2] we can transform C programs to analyse them under weak memory model semantics with SC-only tools. For these transformed programs, SatAbs took 87.8 s on average, Poirot 364.1 s, and ESBMC 723.1 s (all three tools also timed out on several instances). Analysing the transformed programs with CBMC, and SC as memory model, took 6.7 s on average and 305 s at most.

5.3 Real-World Systems Code

We study key components of software widely deployed in server systems. Other tools, including ESBMC and Threader, largely fail to even parse the code.

PostgreSQL Developers observed that a regression test failed on a PowerPC machine,[5] and later identified the memory model as possible culprit: the processor

[5] http://archives.postgresql.org/pgsql-hackers/2011-08/msg00330.php

could delay a write by a thread until after a token signalling the end of this thread's work had been set. A detailed description of the problem is in [2]. Our tool confirms the bug, and proves a patch we proposed to fix the problem. For each memory model, and both with and without the fix, CBMC takes 3 s.

Read-Copy-Update (RCU) is a synchronisation mechanism of the Linux kernel. 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. The old component is cleaned up when there is no process reading.

Thus readers can rely on lightweight (hence fast) lock-free synchronisation. Protection of reads against concurrent writes is fence-free on x86, and uses only a lightweight fence (`lwsync`) on Power. We verify the original implementation of the 3.2.21 kernel for x86 and Power 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 via 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 show that hand-over guarantees consistency of the payload data passed to the worker. The architecture-dependent code is only verified by CBMC, in less than 70 seconds.

6 Related Work and Conclusion

We broadly survey verification for concurrent programs in [3]. Here, we focus on closely related methods for software verification, and weak memory models.

Most existing work for weak memory models supports assembly or toy languages only [18, 35, 26, 9], except for [5, 20, 2] and [10]. Yet [5] bounds the number of context switches, is restricted to TSO, and is not automated. The work of [20] implements an explicit-state analysis for C#. In our prior work [2] we use program transformation to verify C programs w.r.t. weak memory model semantics *using existing SC model checkers*. We discuss [10], which has been successfully applied to non-trivial algorithms, in detail below.

Our work relates the most to [10, 16, 33, 34], which use axiomatic specifications of SC to compose the distinct threads and a similar SSA encoding per thread. The size of the encodings of [10, 33, 34] are $\mathcal{O}(N^3)$ for N shared memory accesses *to any address*, as we detail below; [16] 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 $\mathcal{O}(M^3)$ (due to fr and ab, others are quadratic only), with M the maximal number of events for a *single address*.

In Sec. 5, N on average was 5.7 times larger than M. This extrapolates to a difference of more than *two orders of magnitude* in the size of the formula.

CheckFence [10] encodes total orders over memory accesses. In contrast to our clock variables, [10] uses a Boolean variable M_{xy} per pair (x, y), such that M_{xy} places x and y in a total order: either x before y, or y before x. Furthermore, transitive closure constraints are required; their number is at least cubic in the number of variables M_{xy}. We only consider relations per address, except for program order and fences, and do not build transitive closures. As noted above, the constraints for fr and ab are cubic in the worst case; all others are quadratic.

Sinha and Wang [33, 34] use partial orders like us; they note redundancies in their constraints and then develop pruning [33] and abstractions [34] to reduce these. Initially, [33, 34] quantify over all events regardless of their address, whereas we mostly build constraints per address, based on our formal framework. As said above, this results in two orders of magnitude lower formula size.

Conclusion We developed a symbolic encoding of partial orders to perform bounded model checking of concurrent software. We generalise [33, 34] to weak memory, also unsupported by [16]. We prove suitability and scalability of our tool to systems code, which could not be processed by CheckFence; implementations of [16, 33, 34] are not available. We furthermore showed superior performance on benchmarks of SV-COMP, comparing favourably to all participants.

Acknowledgements We thank Matthew Hague, Alex Horn, Lihao Liang, Vincent Nimal, Peter O'Hearn and Georg Weissenbacher for invaluable discussions and comments.

References

1. Adve, S.V., Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial. IEEE Computer (1995)
2. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In: ESOP. Springer (2013)
3. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient BMC of concurrent software. CoRR abs/1301.1629 (2013)
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. Springer (2011)
6. Ben-Asher, Y., Farchi, E.: Using True Concurrency to Model Execution of Parallel Programs. In: IJPP (1994)
7. Beyer, D.: Second competition on software verification. In: TACAS. Springer (2013)
8. Boehm, H.J., Adve, S.V.: Foundations of the C++ concurrency memory model. In: PLDI (2008)
9. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking robustness against TSO. In: ESOP. Springer (2013)
10. Burckhardt, S., Alur, R., Martin, M.: CheckFence: Checking consistency of concurrent data types on relaxed memory models. In: PLDI (2007)
11. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: TACAS (2005)

12. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. Springer (2004)
13. Cordeiro, L., Fischer, B.: Verifying multi-threaded software using SMT-based context-bounded model checking. In: ICSE (2011)
14. D'Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. TCAD (2008)
15. Fischer, B., Inverso, O., Parlato, G.: CSeq: A sequentialization tool for C (competition contribution). In: TACAS. Springer (2013)
16. Ganai, M., Gupta, A.: Efficient modeling of concurrent systems in BMC. In: SPIN. Springer (2008)
17. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer (1996)
18. Gopalakrishnan, G., Yang, Y., Sivaraj, H.: QB or not QB: An efficient execution verification tool for memory orderings. In: CAV. Springer (2004)
19. Gupta, A., Popeea, C., Rybalchenko, A.: Threader: A constraint-based verifier for multi-threaded programs. In: CAV. Springer (2011)
20. Huynh, Q., Roychoudhury, A.: A memory sensitive checker for C#. In: FM (2006)
21. Kroening, D., Clarke, E., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: DAC (2003)
22. Kuperstein, M., Vechev, M., Yahav, E.: Automatic inference of memory fences. In: FMCAD (2010)
23. Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. In: FMSD (2009)
24. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. CACM (1978)
25. Lamport, L.: How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. IEEE Trans. Comput. (1979)
26. 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. Springer (2012)
27. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: POPL (2005)
28. Owens, S., Sarkar, S., Sewell, P.: A better x86 model: x86-TSO. In: TPHOL (2009)
29. Peled, D.: All from one, one for all. In: CAV (1993)
30. Pratt, V.: Modeling Concurrency with Partial Orders. In: International Journal of Parallel Programming (1986)
31. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: TACAS. Springer (2005)
32. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding Power Multiprocessors. In: PLDI (2011)
33. Sinha, N., Wang, C.: Staged concurrent program analysis. In: FSE (2010)
34. Sinha, N., Wang, C.: On interference abstractions. In: POPL (2011)
35. Torlak, E., Vaziri, M., Dolby, J.: MemSAT: Checking axiomatic specifications of memory models. In: PLDI (2010)
36. Winskel, G.: Event structures. In: Advances in Petri Nets (1986)