The Virtues of Conflict: Analysing Modern Concurrency

Ganesh Narayanaswamy
University of Oxford
ganesh.narayanaswamy@cs.ox.ac.uk

Saurabh Joshi
IIT Guwahati
sbjoshi@iitg.ernet.in

Daniel Kroening
University of Oxford
kroening@cs.ox.ac.uk

Abstract

Modern shared memory multiprocessors permit reordering of memory operations for performance reasons. These reorderings are often a source of subtle bugs in programs written for such architectures. Traditional approaches to verify weak memory programs often rely on interleaving semantics, which is prone to state space explosion, and thus severely limits the scalability of the analysis. In recent times, there has been a renewed interest in modelling dynamic executions of weak memory programs using partial orders. However, such an approach typically requires ad-hoc mechanisms to correctly capture the data and control-flow choices/conflicts present in real-world programs. In this work, we propose a novel, conflict-aware, composable, truly concurrent semantics for programs written using C/C++ for modern weak memory architectures. We exploit our symbolic semantics based on general event structures to build an efficient decision procedure that detects assertion violations in bounded multi-threaded programs. Using a large, representative set of benchmarks, we show that our conflict-aware semantics outperforms the state-of-the-art partial-order based approaches.

Categories and Subject Descriptors [*F1.2*]: Modes of Computation—Parallelism and concurrency

General Terms Verification

Keywords Concurrency, weak consistency models, software, verification

1. Introduction

1.1 Problem Description

Modern multiprocessors employ a variety of caches, queues and buffers to improve performance. As a result, it is not uncommon for write operations from a thread to be not immediately visible to other threads in the system. Thus, writes from a thread, as seen by an external observer, may appear to have been reordered. The specifics of these processor-dependent reorderings are presented to programmers as a contract, called the *memory model*. A memory model dictates the order in which operations in a thread become visible to other threads [5]. Thus, given a memory model, a programmer can determine which values could be returned by a given read operation.

PPoPP '16 March 12-16, 2016, Barcelona, Spain

DOI: http://dx.doi.org/10.1145/2851141.2851165

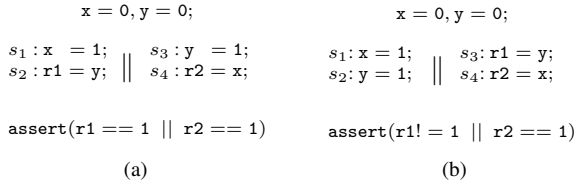


Figure 1: (a) Reordering in TSO (b) Reordering in PSO

While most developers are aware that instructions from two different threads could be interleaved arbitrarily, it is not atypical for a programmer to expect statements *within* one thread to be executed in the order in which they appear in the program text, the so called *program order* (PO). A memory model that guarantees that instructions from a thread are always executed in program order is said to offer sequential consistency (SC) [27]. However, none of the performant, modern multiprocessors offer SC: instead, they typically implement what are known as *relaxed* or *weak memory models* (R/WMM), by relaxing/weakening the program order for performance reasons. In general, the weaker the model, the better the opportunities for performance optimisations: the memory model alone could account for 10–40% of the processor performance in modern CPUs [40].

Such weakenings, however, not only increase performance, but also lead to intricate weak-memory artefacts that make writing correct multiprocessor programs non-intuitive and challenging. A key issue that compounds and exacerbates this difficulty is the fact that weak-memory bugs are usually non-deterministic: that is, weak memory defects manifest *only* under very specific, often rare, scenarios caused by a particular set of write orderings and buffer configurations. Although all modern architectures provide *memory barriers* or *fences* to prevent such relaxation from taking place around these barriers, the placement of fences remains a research topic [3, 4, 7, 24, 26, 29, 30] due to the inherent complexities involved caused by the intricate semantics of such architectures and fences. Thus, testing-based methods are of limited use in detecting weak memory defects, which suggests that a more systematic analysis is needed to locate these defects.

In this work, we present a novel, true-concurrency inspired investigation that leverages symbolic Bounded Model Checking (BMC) to locate defects in modern weak memory programs. We begin by introducing the problem of assertion checking in weak memory programs using pre-C11 programs as exemplar, and introduce the concerns that motivate our approach.

1.2 Example

Consider the program given in Fig. 1a. Let `x` and `y` be shared variables that are initialised with 0. Let the variables `r1` and `r2` be thread local. Statements s_1 and s_3 both perform write operations. However, owing to store-buffering, these writes may not be imme-

diately dispatched to the main memory. Next, after performing s_1 and s_3, both threads may now proceed to perform the read operations s_2 and s_4. Since the write operations might still not have hit the memory, stale values for `x` and `y` may be read into `r2` and `r1`, respectively. This may cause the assertion to fail. Such a behaviour is possible in a processor that implements *Total Store Ordering* (TSO), which permits *weakening* (or *relaxing*) the write-read ordering when the operations are performed on different memory locations. Note that on a hypothetical architecture that guarantees SC, this would never happen. However, due to store buffering, a global observer might witness that the statements were executed in the order $s_2; s_4; s_1; s_3$ which resulted in the said assertion failure. We say that the events inside the pairs (s_1, s_2) and (s_3, s_4) have been *reordered*.

Fig. 1b illustrates how the assertion might fail on architectures that implement *Partial Store Order* (PSO), which permits write-write and write-read reorderings when these operations are on different memory locations. If SC was honoured, one would expect to observe `r2` $==$ `1` if `r1` $==$ `1` has been observed. However, reordering of the write operations (s_1, s_2) (under PSO) would lead to the assertion failure.

In this work we would like to find assertion violations that occur in programs written for modern multiprocessors. Specifically, we will be focussing on C programs written for architectures that implement reordering-based memory models like TSO and PSO. We assume that the assertions to be checked are given as part of the program text.

1.3 Our Approach

Our approach differs from most existing research in the way we model concurrency. Most traditional work rely exclusively on interleaving semantics to reason about real-world multiprocessor programs. An interleaving semantics purports to reduce concurrent computations to their equivalent non-deterministic *sequential* computations. For instance, let P be a system with two concurrent events a and b; let's denote this fact as $P \triangleq a \| b$. Interleaving semantics then assigns the following meaning to P: $[\![P]\!] \triangleq a.b \mid b.a$ where '.' denotes sequential composition and '|' denotes non-deterministic choice. That is, a system in which a and b happens in parallel is indistinguishable from a system where a and b could happen in any order; we call $a.b$ (also $b.a$) a *schedule*. The set of all possible schedules is called the *schedule space* of P. As the input program size increases, the (interleaved) schedule space of the program may grow exponentially. This schedule space explosion severely limits the scalability of many analyses. The state space issues caused by interleaving semantics are only exacerbated under weak memory systems: weakening a memory model increases the number of admissible reorderings, and with it the degree of non-determinism.

In this paper, we propose an alternative approach: to directly capture the semantics of shared memory programs using *true concurrency* [1, 18]. There are two competing frameworks for constructing a truly concurrent semantics, one based on *event structures* [39] and the other on *pomsets* [32]. A recent paper [9], the work that is closest to ours, uses partial orders (pomsets) to capture the semantics of shared memory program. The main insight is that partial orders neatly capture the causality of events in the dynamic execution of weak memory programs. But such a model cannot directly capture the control and data flow choices present in the programs: the semantics of programs with multiple, conflicting (that is, mutually exclusive) dynamic executions is captured simply as a *set of* candidate executions [9]. In this work, we advocate integrating program choices *directly* into the true concurrency semantics: we show that this results in a more succinct, algebraic presentation, leading to a more efficient analysis. Our true concurrency semantics employs general event structures which, unlike partial orders, tightly integrate the branching/conflict structure of the program with its causality structure. Intuitively, such a conflict-aware truly concurrent meaning of P can be given as follows: $[\![P]\!] \triangleq \neg(a \# b) \wedge \neg(a < b)$; that is, the events a and b are said to be concurrent iff they are not conflicting (#) and are not related by a 'happens-before' ($<$) relation. This (logical) characterisation is strictly more expressive compared to interleaving-based characterisation[1]; in addition, such a semantics does not suffer from the schedule space explosion problem of a (more operationally defined) interleaving semantics. Intuitively, a true concurrency admits the phenomenon of concurrency to be a *true*, a first-class phenomenon that exists in the real-world and needed to modelled as such, as opposed to simply reducing it to a sequentialised choice.

Our event structure based semantics can naturally distinguish computations at a granularity finer than trace equivalence. For instance, consider threads t_1, t_2 and s, defined as follows: $t_1 = a.b \mid a.c$, $t_2 = a.(b|c)$ and $s = a.c$. Note than t_1 and t_2 are trace (and partial order) equivalent. Let $\|$ denote the (synchronous) parallel composition [34]. Then, $t_1 \| s$ can deadlock while $t_2 \| s$ cannot. Our semantics can distinguish $t1$ from $t2$, while current partial order based methods cannot. This is because partial order based methods capture the semantics as a set of *complete executions*, without ever specifying how the partial sub-executions that constitute a trace unfold. Thus, our semantics can be used to reason about deadlocks over partial computations involving, say, lock/unlock operations — not just assertion checking over complete computations involving read/write operations.

Although event structures offer an attractive means to capture the semantics of weak memory programs, we are not aware of any event structure based tool that verifies modern weak memory programs written using real-world languages like C/C++. In this work, we address this lacuna: by investigating the problem of assertion checking in weak memory programs using event structures. We first develop a novel true concurrency semantics based on general event structures [39] for a *bounded model checker*. We then formulate a succinct symbolic decision procedure and use this decision procedure to locate assertion violations in modern shared memory programs. Our tool correctly and efficiently analyses the large, representative set of programs in the SV-COMP 2015 [16] benchmark suite. It also successfully handles all the intricate tests in the widely used Litmus suite [10].

1.4 Contributions

Following are our contributions.

1. A compositional, symbolic (event structure based) true concurrency semantics for concurrent programs, and a characterisation of assertion violation over this semantics (Section 4).
2. A novel decision procedure based on the above semantics that locates assertion violations in multi-threaded programs written for weak memory architectures (Section 5).
3. A BMC-based tool that implements our ideas and a thorough performance evaluation on real-world C programs of our approach against the state-of-the-art research (Section 6).

We present our work in to three parts. The first part introduces the relevant background on weak memory models (Section 2) and true concurrency (Section 3). The second part defines an abstract, true concurrency based semantics for weak memory programs written in C (Section 4) and presents a novel decision procedure that exploits this abstract semantics to find assertion violations (Section 5). In the third part we discuss the specifics of our tool and

[1] For systems with a finite set of events, the expressibility of both the notions of concurrency coincides.

present a thorough performance evaluation (Section 6); we then discuss the related work in Section 7 and conclude.

2. Background

This section summarises the ideas behind three concepts: weak memory models, bounded model checking, and our intermediate program representation.

2.1 Weak Memory Models

We introduce three memory models — SC, TSO and PSO — and the necessary intuitions to understand them. We currently do not directly handle other memory models; please consult the related work section for further discussion on this.

SEQUENTIAL CONSISTENCY (SC): This is the simplest and the most intuitive memory model where executions strictly maintain the intra-thread program order (PO), while permitting arbitrary interleaving of instructions from other threads. Intuitively, one could view the processors/memory system that offers SC as a single-port memory system where the memory port is connected to a switch/-multiplexer, which is then connected to the processors. This switch can only commit one instruction from a thread/processor at a time, and it does so non-deterministically; this is depicted in Fig. 2a. The illustrations in Fig. 2 are from the SPARC architecture manual [36].

TOTAL STORE ORDER (TSO): In TSO, in addition to the behaviours permitted in SC, a write followed by a read to a different memory location may be reordered. Thus, the set of executions permissible under TSO is a strict superset of SC. This memory model is used in the widely deployed x86 architecture. Writes in x86 are first enqueued into the *store buffer*. These writes are later committed to memory in the order in which they are issued, i.e., the program order of writes is preserved. The store buffer also serves as a read cache for the future reads from the same processor. But any read to a variable that is not in the store buffer can be issued directly to memory and such reads could be completed before the pending, enqueued writes hit the memory. Well known mutual exclusion algorithms like Dekker, Peterson and Parker are all unsafe on x86. A TSO processor/memory system could be seen as one where the processor issues writes to a store buffer, but sends the reads directly to memory; this is depicted in Fig. 2b.

PARTIAL STORE ORDER (PSO): PSO is TSO with an additional relaxation: PSO guarantees that only writes to the *same location* are committed in the order in which they are issued, whereas writes to different memory locations may be committed to memory out of order. This is intuitively captured by a processor/memory system that employs separate store buffers for writes that write to different memory addresses, but the reads are still issued directly to the memory; this is depicted in Fig. 2c.

2.2 Bounded Model Checking

Bounded Model Checking (BMC) is a Model Checking technique that performs a depth-bounded exploration of the state space. This depth is often given in the form of an unwinding limit for the loops in the program. Bugs that require longer paths (deeper unwindings) are missed, but any bug found is indeed a real bug. Bounded Model Checkers typically employ SAT/SMT-based symbolic methods to explore the program behaviour exhaustively up to the given depth. As modern SAT solvers are able to solve propositional formulas that contain millions of variables, BMC is increasingly being used to verify real-world programs [20].

2.3 Program Representation

We rely on a Bounded Model Checker that uses a symbolic static single assignment form (SSA) to represent the input program. Specifically, we use CBMC [21], a well-known bounded model checker that supports C99 and most of the C11 standard. It can handle most compiler extensions provided by gcc and Visual Studio. Over this CBMC-generated symbolic SSA, we define two additional relations: a per-thread binary relation called *preserved program order* (PPO) and a system-wide n-ary relation called *potential matches* (POTMAT), where n is the total number of threads in the system. The triple — SSA, PPO, and POTMAT — is then used to define a truly concurrent semantics of the input program; one can see this triplet as an intermediate form that we use to represent all relevant aspects of the input program. These three components are discussed in more detail below.

STATIC SINGLE ASSIGNMENT FORM (SSA): The control and data flow of the input program is captured using guarded SSA form [22]. In a traditional SSA [31] or a concurrent SSA [28], a distict symbolic variable is introduced when a program variable appears on the left hand side (lhs) of an assignment, whereas ϕ and π function represents the possible values that may flow into an assignment. In the guarded SSA, each occurrence of the shared variable is treated as a distinct symbolic variable, and is given a unique SSA index, essentially allowing right hand side (rhs) symbols to remain unconstrained. In the guarded SSA, assignments are converted into equalities and conditionals act as guards to enforce these equalities. To restrict the values to only those as permitted by the underlying memory model, additional equality constraints specific to the memory model are then added. These constraints capture all possible interleavings and reordering of read and write operations as required by the model. Thus, the π functions of concurrent SSA are subsumed by these constraints. The details of the encoding are provided in Section 5.

We rely on the underlying BMC tool to supply the necessary (symbolic) variables and the constraints to cover the C constructs used in the input program. Each SSA assignment is decomposed into a pair of read and write events. These read/write events are augmented with guards over symbolic program variables: this guard is a disjunction (induced by path merging) of all conjunctions (induced by nested branch) of all paths that lead to the read/write events. For each uninitialised variable, we add a *initial write* that sets the variable to a non-deterministic, symbolic value. From now on, we will refer to our guarded SSA simply as SSA.

PRESERVED PROGRAM ORDER (PPO): This is a per-thread binary relation specific to the memory-model that is directed, irreflexive and acyclic. PPO captures the intra-thread ordering of read/write events. Given an input program in SSA form, different memory models produce different PPOs. Let TPPO be a binary relation over the read/write events where, for every event e_1 and e_2, $(e_1, e_2) \in$ TPPO iff the event e_1 cannot be relaxed after e_2. Note that TPPO is a partial order: it is transitive, anti-symmetric and reflexive. TPPO is collectively determined by the memory model under investigation and the fences present in the input program. We define PPO to be the (unique) transitive reduction [6] of the TPPO.

POTENTIAL MATCHES RELATION (POTMAT): While PPO models the intra-thread control and data flow, the potential matches relation aims at the inter-thread data flow. It is an n-ary relation with two kinds of tuples. Let m be a tuple and let $m(i)$ denote the i^{th} entry (for thread i) in the tuple. The first kind of tuple with one event — where $m(i) = e$ — captures the idea that the event e (in thread i) can happen by itself; the remaining tuple entries contain '*'. We

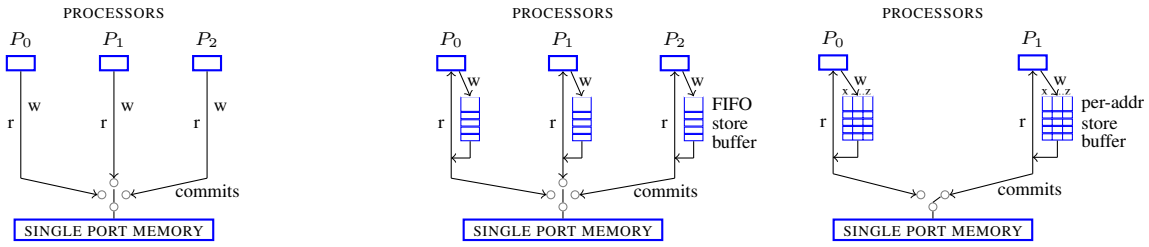


Figure 2: Execution Models: (a) SC (b) TSO (c) PSO

Thread 1:

```
x = 1;      // Wx3
if(y % 2)   // Ry3
   x = 3;   // Wx4
else
   x = 7;   // Wx5
y = x;      // Wy4, Rx6
```

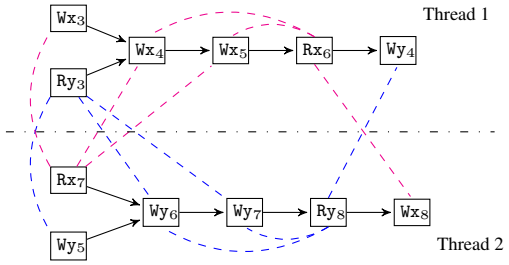
$\texttt{guard}_1 \triangleq \top$
$\texttt{guard}_1 \Rightarrow (\texttt{x}_3\texttt{=1})$
$\texttt{guard}_2 \triangleq (\texttt{y}_3\texttt{\%2=0})$
$\texttt{guard}_3 \triangleq (\texttt{guard}_1 \wedge \texttt{guard}_2)$
$\texttt{guard}_4 \triangleq (\texttt{guard}_1 \wedge \neg\texttt{guard}_2)$
$\texttt{guard}_4 \Rightarrow (\texttt{x}_4\texttt{=3})$
$\texttt{guard}_3 \Rightarrow (\texttt{x}_5\texttt{=7})$
$\texttt{guard}_1 \Rightarrow (\texttt{x}_{\phi 1}\texttt{=guard}_2\texttt{?x}_5\texttt{:x}_4)$
$\texttt{guard}_1 \Rightarrow (\texttt{y}_4\texttt{=x}_6)$

Thread 2:

```
y = 1;      // Wy5
if(x % 2)   // Rx7
   y = 3;   // Wy6
else
   y = 7;   // Wy7
x = y;      // Wx8, Ry8
```

$\texttt{guard}_5 \triangleq \top$
$\texttt{guard}_5 \Rightarrow (\texttt{y}_5\texttt{=1})$
$\texttt{guard}_6 \triangleq (\texttt{x}_7\texttt{\%2=0})$
$\texttt{guard}_7 \triangleq (\texttt{guard}_5 \wedge \texttt{guard}_6)$
$\texttt{guard}_8 \triangleq (\texttt{guard}_5 \wedge \neg\texttt{guard}_6)$
$\texttt{guard}_8 \Rightarrow (\texttt{y}_6\texttt{=3})$
$\texttt{guard}_7 \Rightarrow (\texttt{y}_7\texttt{=7})$
$\texttt{guard}_5 \Rightarrow (\texttt{y}_{\phi 1}\texttt{=guard}_6\texttt{?y}_7\texttt{:y}_6)$
$\texttt{guard}_5 \Rightarrow (\texttt{x}_8\texttt{=y}_8)$

(a)



(b)

Figure 3: A program and its corresponding intermediate form

say that such an e is a *free event*. Note that writes are free events, as they can happen by themselves.

Let i and j be two *distinct* thread indices that is, $i \neq j$ and — $0 \leq i, j < n$. The second kind of tuple, involving two events — where $m(i) = r$ and $m(j) = w$ — denotes a *potential* inter-thread communication where a read r (from thread i) has read the value written by the write w (from thread j); the rest of the tuple entries contain '*'. Such an r is called a *synchronisation event*. Reads are synchronisation events as they cannot happen by themselves: reads always happen in conjunction with a free (write) event. One should see synchronisation events as events that *consume* other events, thus always needing another (free) event to happen. As we will see later (Section 3), these two kinds of tuples/events are fundamental to our semantics.

Informally, POTMAT should be seen as a over-approximation of all possible inter-thread choices/non-determinism available to each shared read (and write) in a shared-memory program. We assume that for every shared read, there is at least one corresponding tuple (m) that *matches* the said read with a write: this corresponds to our intuition that every successful read must have read from *some* write. We do not demand the converse: there indeed could be writes that were not read by any of the reads. Also, any such m can only relate reads and writes that operate on the *same* memory location: that is, reads and write are related by the potential matches iff they operate on the same (original) program variable. A tuple in POTMAT is a potential inter-thread *match* — either containing an event that matches with itself, or a pair of events that could match with one another — hence the name potential matches. We sometimes denote the potential matches relation as $\mathbb{M}$. We currently construct $\mathbb{M}$ as a (minimally pruned) subset of the Cartesian product between per-thread events that share the same address[2]. This subset consists only of the two aforementioned types of tuples, and has at least one tuple for every read (containing two events) and write (containing one event).

EXAMPLE: Consider the two-thread program in Fig. 3a. The bottom half gives the corresponding SSA form. Both `x` and `y` are shared variables. The guards associated with each event can also be seen in the figure. As the distinct symbolic variables are introduced for every occurrance of a program variable, assignments can be converted to guarded equalities. Guards capture the condition that must hold true to enforce an equality. The symbols $\texttt{guard}_3$ and $\texttt{guard}_4$ illustrate how path conditions are conjucted as we go along branches. These symbols are also said to guard the events participating in the equality. For example, $\texttt{guard}_1 \Rightarrow (\texttt{y}_4\texttt{=x}_6)$ denotes not only that $\texttt{guard}_1$ implies the equality but also that it acts as guard to corresponding events $\texttt{Wy}_4$ and $\texttt{Rx}_6$: that is, *guard* (Wy_4) = *guard* (Rx_6) = $\texttt{guard}_1$. When the local paths merge, auxiliary variables (e.g., $\texttt{x}_{\phi 1}$) are introduced, which hold appropriate intra-thread values depending upon which path got executed. Note that `x6` is completely free in the constraints given in the figure. Later on, additional constraints are added, which restrict the value of `x6` to either an intra-thread value of $\texttt{x}_{\phi 1}$ or an inter-thread value of $\texttt{x}_8$. The corresponding TSO intermediate form is given in Fig. 3b: note that TSO relaxes the program order between $(\texttt{Wx}_3, \texttt{Ry}_3)$ and $(\texttt{Wy}_5, \texttt{Rx}_7)$. The (intra-thread) solid arrows depict the (intra-thread) preserved program order, and dashed lines depict the potential matches rela-

[2] More formally, $\mathbb{M}$ is proper subset of the n-ary fibred product between per-thread event sets (say, $E_i \cup$ '*') where the event labels agree. The label of '*' agrees with all the events in the system.

tion. The magenta lines show the matches involving x and the blue lines show the matches involving y. The initial writes are omited for brevity. The horizontal dash-dotted line demarcates the thread boundaries.

3. True Concurrency

Although most of the existing literature on event structures deal with prime or stable event structures [39], we will be using (a heavily modified) general event structure. General event structures are (strictly) more expressive compared to prime/stable event structures [37, 38]. In addition, the constructions we employ — parallel composition and restriction of event structures — have a considerably less complex presentation over general event structures.

We now present the concepts and definitions related to event structures. In each case, we give the formal definitions first, followed by an informal discussion as to what these definitions capture. Also, hereafter we will simply say 'event structures' to mean the modified general event structure defined by us.

A GENERAL EVENT STRUCTURE is a quadruple $(E, Con, \vdash, label)$[3], where:

- E is a countable set of *events*.
- $Con \triangleq \{X | X \subseteq_{finite} E, \forall e_1 \neq e_2 \in X \Rightarrow (e_1, e_2) \notin \#\}$. $\#$ is an irreflexive, symmetric relation on E, called the *conflict relation*. Intuitively, Con can be viewed as a collection of mutually consistent sets of events.
- $\vdash \subseteq Con \times E$ is an *enabling relation*.
- $label : E \to \Sigma$ is a labeling function and Σ is a set of labels.

such that:

- Con is consistent: $\forall X, Y \subseteq E$, $X \subseteq Y$, $Y \in Con \Rightarrow X \in Con$
- $\vdash$ is extensive: $\forall e \in E$, $\forall X, Y \in Con$, $X \vdash e$, $X \subseteq Y \Rightarrow Y \vdash e$

Let us now deconstruct the definition above. We would like to think of a thread as a countable set of events (E), which get executed in a particular fashion. Since we are interested only in finite computations, we require that all execution 'fragments' are finite. Additionally, for fragments involving conflicting events, we require that at most one of the conflicting events occurs in the execution fragment. The notion of computational conflict (or choice) is captured by the conflict relation (#). We call executions that abide by all the requirements above *consistent executions*; Con denotes the set of all such consistent executions. Thus, $Con \subseteq 2^E$ is the set of conflict-free, finite subsets of E. Since we want the 'prefixes' of executions to be executions themselves, we demand that Con is subset closed. Such execution fragments can be 'connected' to events using the enabling ($\vdash$) relation: $X \vdash e$ means that events of X enable the event e.

For example, in an SC architecture if there is a write event w followed by a read event r, then $(\{w\}, r) \in \vdash$ as w must happen before r could happen. In general, $\vdash$ allows us to capture the dependencies within events as dictated by the underlying memory model. Note that since the enabling relation connects the elements of Con with that of E, it is automatically branching/conflict aware. We do not require that a set X enabling e to be the minimal set (enabling e): extensiveness only requires that X contains a subset that enables e. The labeling function, $label(e)$, returns the label of the read/write event e. These labels are interpreted as addresses of the events. Finally, it is often useful to see E as a union of three disjoint sets R, W and IRW, where R corresponds to the set of reads, W to the set of writes and IRW correspond to the set of *local reads* (see Section 4).

CONFIGURATION: A *configuration* of event structure $(E, Con, \vdash)$ is a subset $C \subseteq E$ such that:

- C is conflict-free: $C \in Con$
- C is secured: $\forall e \in C$, $\exists e_0, \ldots, e_n \in C$, $e_n = e \wedge \forall i$ $0 \leq i \leq n$. $\{e_0, \ldots, e_{i-1}\} \vdash e_i$

A configuration $C \subseteq E$ is to be understood as a history of computation *up to* some computational state. This computational history cannot include conflicting events, thus we would like all finite subsets of C to be conflict free; this can also be ensured by requiring that all finite subsets of C be elements of Con. Securedness ensures that for any event e in a configuration, the configuration has as subsets a sequence of configurations $\emptyset, \{e_0\}, \ldots, \{e_0, \ldots, e_n\}$ — called a securing for e in C, such that one can build a 'chain of enablings' that will eventually enable e; all such chains must start from $\emptyset$.

Let the set of all configurations of the event structure $(E, Con, \vdash)$ be denoted by $\mathcal{F}(E)$. A *maximal configuration* is a configuration that cannot be extended further by adding more events.

COINCIDENCE FREE: Given an event structure $(E, Con, \vdash)$, we say that it is *coincidence free* iff $\forall X \in \mathcal{F}(E)$, $\forall e, e' \in X$, $e \neq e' \Rightarrow \exists Y \in \mathcal{F}(E), Y \subseteq X, (e \in Y \Leftrightarrow e' \notin Y)$. Intuitively, this property ensures that configurations add at most one event at a time: this in turn ensures that secured configurations track the enabling relation faithfully. We require our event structures to be coincidence free. This is a technical requirement that enables us to assign every event in a configuration a unique clock order (see below).

TRACE AND CLOCK ORDERS: Given an event e in configuration C and a securing up to e_k — that is, $\{e_{i=0}, e_{i=1}, \ldots, e_{i=k-1}\} \vdash e_{i=k}$ — we define the following injective map $trace_C(e) : E|_C \to \mathbb{N}^0$ as $trace_C(e) \triangleq i$. Informally, $trace_C(e)$ is the trace position of event e in C: $trace_C(e_1) < trace_C(e_2)$ implies that the event e_1 occurred before e_2 in the given securing of the configuration C. Given such a $trace_C$ map, we define a monotone map named *clock* as $clock_C(e) : e \to \mathbb{N}^0$ that is *consistent* with $trace_C$. That is, $\forall e_1, e_2 \in C$, $trace_C(e_1) < trace_C(e_2) \Rightarrow clock_C(e_1) < clock_C(e_2)$. Informally, the $clock_C$ map *relaxes* the $trace_C$ map monotonically so that $clock_C$ can accommodate events from other threads, while still respecting the ordering dictated by $trace_C$.

PARTIAL FUNCTIONS: As part of our event structure machinery, we will be working with partial functions on events, say $f : E_0 \to E_1$. The fact that f is undefined for a $e \in E_0$ is denoted by $f(e) = \bot$. As a notational shorthand, we assume that whenever $f(e)$ is used, it is indeed defined. For instance, statements like $f(e) = f(e')$ are always made in a context where both $f(e)$ and $f(e')$ are indeed defined. Also, for a $X \subseteq E_0$, $f(X) = \{f(e) \mid e \in X \text{ and } f(e) \text{ is defined}\}$.

MORPHISMS: A morphism between event structures is a structure-preserving function from one event structure to another. Let $\Gamma_0 = (E_0, Con_0, \vdash_0)$ and $\Gamma_1 = (E_1, Con_1, \vdash_1)$ be two stable event structures.

A *partially synchronous morphism* $f : \Gamma_0 \to \Gamma_1$ is a function f from read set (R_0) to write set (W_1) such that:

[3] Hereafter, for brevity, $(E, Con, \vdash)$ will stand for $(E, Con, \vdash, label)$: that is, every event structure is implicitly assumed to be equipped with a label set Σ and a labling function $label : E \to \Sigma$.

- f preserves consistency: $\forall X \in Con_0 \Rightarrow f(X) \in Con_1$.
- f preserves enabling: $\forall X \vdash_0 e, \texttt{def}(f(e))$[4] $\Rightarrow f(X) \vdash_1 f(e)$
- f preserves the labels: $f(e) = e' \Rightarrow label(e) = label(e')$
- f does not time travel: $X \in Con_0, Y \in Con_1$, $f(e) = e' \Rightarrow clock_X(e) > clock_Y(e')$
- f ensures freshness: $X \in Con_0, Y \in Con_1$, $f(e) = e'$, then $\forall e'' \in Y$ such that $label(e'') = label(e)$, $clock_Y(e'') < clock_X(e) \Rightarrow clock_Y(e'') < clock_Y(e')$

Such an f is called *synchronous morphism* if it is total.

A morphism should be seen as a way of synchronising reads of one event structure with the writes of another. We naturally require such a morphisms to be a function in the set theoretic sense: this ensures that a read always reads from exactly one write. Note that the requirement of f *being a function* introduces an implicit conflict between competing writes. Given a morphism $f : \Gamma_0 \to \Gamma_1$, $f(r_0) = w_1$ is to be understood as r_0 reading the value written by w_1 (or r_0 *synchronising* with w_1). Thus, the requirement that f is a function will disallow (or will 'conflict' with) $f(r_0) = w_2$. Such a morphism need not be total over E_0. The events for which f is defined are called the *synchronisation events*; thus, reads are synchronisation events. Recall that synchronisation events are to be seen as events that *consume* other (free) events. The events for which f is undefined are called *free events*. Writes are free events as they can happen freely without having to synchronise with events from another event structure. We do *not* require these morphisms to be injective: this allows for multiple reads to read from the same write. We require such a morphism to be consistency preserving: that is, morphisms map consistent histories in Con_1 to consistent histories in Con_2. We require that the morphisms preserve the $\vdash$ relation as well.

The next three requirements capture the idiosyncrasies of shared memory. First, we require that a morphism preserves labels. The labels are understood to be as addresses of program variables: this ensures that read and write operations can synchronise if and only if they are performed on the same address/label. Second, we demand that a morphism never reads a value that is not written: that is, any write that a read reads must have happened before the read. The final requirement ensures that a read always reads the latest write.

PRODUCT $\times$: Let $\Gamma_0 = (E_0, Con_0, \vdash_0)$ and $\Gamma_1 = (E_1, Con_1, \vdash_1)$ be two stable event structures. The *product* $\Gamma = (E, Con, \vdash)$, denoted $\mathbf{\Gamma_0 \times \Gamma_1}$, is defined as follows:

- $E \triangleq \{(e_0, *) \mid e_0 \in E_0\} \bigcup \{(*, e_1) \mid e_1 \in E_1\} \bigcup \{(e_0, e_1) \mid e_0 \in E_0, e_1 \in E_1, label(e_0) = label(e_1)\}$
- Let the *projection morphisms* $\pi_i : E \to E_i$ be defined as $\pi_i(e_0, e_1) = e_i$, for $i = 0, 1$. Using these projection morphisms, let us now define the Con of the product event structure as follows: for $X \subseteq E$, we have $X \in Con$ when
  - $\{X \mid X \subseteq_{finite} E\}$
  - $\pi_0 X \in Con_0$, $\pi_1 X \in Con_1$
  - Read events in X *form a function*: $\forall e, e' \in X$, $\big((\pi_0(e) = \pi_0(e') \neq *) \wedge (\pi_0(e) \in R_0)\big) \vee \big((\pi_1(e) = \pi_1(e') \neq *) \wedge (\pi_1(e) \in R_1)\big) \Rightarrow e = e'$
  - events in X *do not time travel*: that is, $\forall e \in X$, $\big((\pi_0(e) \in W_0 \wedge \pi_1(e) \in R_1) \Rightarrow clock_{\pi_0 X}(\pi_0(e))$[5] $< clock_{\pi_1 X}(\pi_1(e))\big)$ $\wedge$ $\big((\pi_0(e) \in R_0 \wedge \pi_1(e) \in W_1) \Rightarrow clock_{\pi_1 X}(\pi_1(e)) < clock_{\pi_0 X}(\pi_0(e))\big)$
  - read events in X *read the latest write*: $\forall e \in X$, $\pi_0(e) \in W_0 \wedge \pi_1(e) \in R_1$, $\forall w_i \in W_0(label(\pi_0(e)))$[6] $\setminus \pi_0(e)$, $clock_{\pi_1 X}(\pi_1(e)) > clock_{\pi_i X}(w_i) \Rightarrow clock_{\pi_0 X}(\pi_0(e)) > clock_{\pi_i X}(w_i)$[7]
  - in any given X, all the write events to the same address are *totally ordered*. Let Σ be a finite, label set, denoting the set of addresses/variables in the program. Then, $\forall l \in \bigcup_i \Sigma_i, i \in \{0, 1\}, \forall w, w' \in W(l)$, $(clock_{\pi_i X}(w) < clock_{\pi_i X}(w')) \vee (clock_{\pi_i X}(w') < clock_{\pi_i X}(w))$
- $X \vdash e \triangleq \forall X \in Con, \forall e \in E$, $0 \leq i, j \leq 1$, $i \neq j$, $e_i = \pi_i(e)$, $e_j = \pi_j(e)$, $\big(e_i \in R_i \Rightarrow e_j \neq *\big) \wedge \big(e_i = * \wedge e_j \in W_j \Rightarrow \pi_j X \vdash_j e_j\big) \wedge \big(e_i = * \wedge e_j \in IRW_j$[8] $\Rightarrow \pi_j X \vdash_j e_j\big) \wedge \big(e_i \in R_i \wedge e_j \in W_j \Rightarrow \pi_i X \vdash_i e_i \wedge \pi_j X \vdash_j e_j\big)$

Products are a means to build larger event structures from components. The event set of product event structure has all the combinations of the constituent per-thread events to account for all possible states of the system. A product should also accommodate the case where events in a thread do not synchronise with any event in other threads. This is ensured by introducing the dummy event '$*$'.

We next demand that admissible executions in the product event structure yield admissible executions in the constituent, per-thread event structures. This is ensured by introducing projection morphisms that 'project' executions of the product event structure to their respective, per-thread ones: we require these projected, per-thread executions to be consistent executions. Next, we forbid any read in an execution to match with more than one write, ensure that a read's clock is greater than that of the corresponding write's clock, and that a read always reads the latest write. We also demand that the writes to an address are always totally ordered. Finally, we demand that the enabling relation of product reflects all the per-thread enabling relations. This is ensured by requiring any product-wise enabling yields a valid per-thread enabling. It is important to note that every event in the product event structure is a free event, and product events do not synchronise with any other event.

RESTRICTION $\lceil$: Let $\Gamma = (E, Con, \vdash)$ be an event structure. Let $A \subseteq E$. We define the *restriction* of Γ to A, denoted $\mathbf{\Gamma\lceil_A} \triangleq (E_A, Con_A, \vdash_A)$, as follows.

- $E_A \triangleq A$
- $X \in Con_A \Leftrightarrow X \subseteq A, X \in Con$
- $X \vdash_A e \triangleq X \subseteq A, e \in A, X \vdash e$

Restriction builds a new event structure containing only events named in the restriction set: it restricts the set of events to A, isolates consistent sets involving events in A, and ensures that events of A are enabled appropriately.

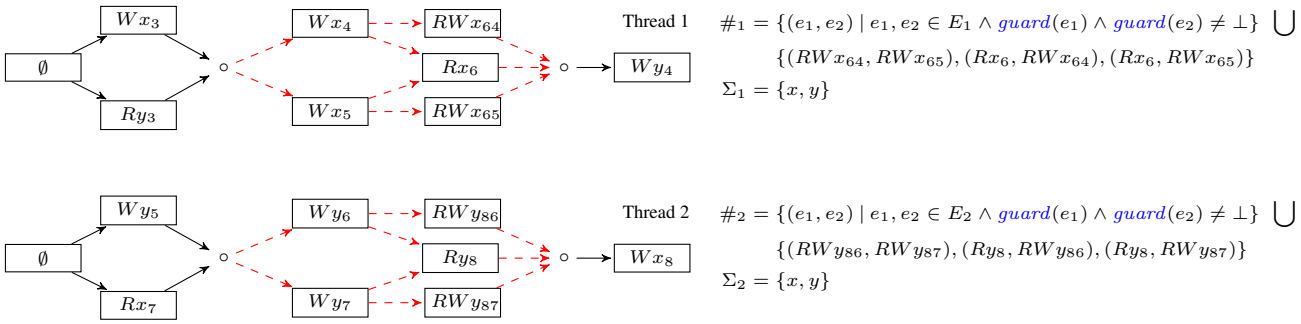
[4] The $\texttt{def}(f(e))$ predicate returns true if $f(e)$ is defined.

[5] $clock_{\pi_i X}(w_i)$ denotes the clock value of the event w_i in $\pi_i X$; i denotes the index of the process/thread that issued w_i. Note that our clock constraints only restrict the clocks of per-thread events, and the clock values of the product events are left 'free'.

[6] $W_0(label(\pi_0(e)))$ denotes the set of write events in thread 0 that share the same address/label as $\pi_0(e)$.

[7] The dual of this requirement, where we swap 0 and 1, is also assumed; we omit stating it for brevity.

[8] The set $IRW_j \subseteq E_j$ denotes the set of internal/local reads in thread j: $IRW_j = \{RW_{lm} \mid r_l \in R_j, w_m \in W_j, label(r_l) = label(w_m)\}$.



(a) The per-thread event structures

$$E = \{(Ry_3, Wy_5), (Ry_3, Wy_6), (Ry_3, Wy_7), (RWx_{64}, *), (RWx_{65}, *), (Rx_6, Wx_8), (Wx_3, Rx_7), (Wx_4, Rx_7), (Wx_5, Rx_7), (*, RWy_{86}), (*, Ry_{87}), (Wy_4, Ry_8), \} \qquad \Sigma = \Sigma_1 \cup \Sigma_2 = \{x, y\}$$
$$\# = \{((R{-_i}, W{-_j}), (R{-_i}, W{-_k})) \mid (R{-_i}, W{-_j}), (R{-_i}, W{-_k}) \in E, - \in \Sigma\}$$

(b) Semantics of the shared memory program

Figure 4: Event structure constructions for the example given in Fig. 3

4. Semantics for weak memory

Let P be a shared memory program with n threads. Each thread is modelled by an event structure $\Gamma_i = (E_i, Con_i, \vdash_i)$, which we call a *per-thread event structure*. The per-thread event structures are constructed using our per-thread PPOs and the guards associated with the read/write events. The computed guards naturally carry the control and data flow choices of a thread into the conflict relation of the corresponding per-thread event structure: two events are conflicting if their guards are conflicting; conflicting guards are those that cannot together be set to true.

As we build our E_i from PPO_i, in addition to all the read/write events in the PPO_i, we also add the set of local reads (IRW_i) (of thread i) into E_i as free events; we call a read event RWx_{kl} a *local* or *internal read* if it reads from a write Wx_l from the same thread. Note that all possible write events that can feed a value to a given local read can be statically computed (e.g., using def-use chains). Such local reads (say RWx_{kl}) are added as free events in E_i: in doing so, we require that the guards of the constituent events (*guard* (RWx_{kl}) and *guard* (Wx_l)) do not conflict, and that functoriality/freshness of reads/writes is guaranteed in all per-thread $X \in Con_i$ involving them. The intuition is that reads reading from local writes are free to do so without 'synchronising' with any other thread. Our POTMAT relation is constructed after adding such free, local reads. Since a read event can either read a local write or a (external) write from another thread, local reads are considered to be in conflict with the external reads that has to 'synchronise' with other threads. This conflict captures the fact that at runtime only one of these events will happen.

Let us also denote the system-wide POTMAT as $\mathbb{M}$. We are now in a position to define our *truly concurrent semantics for the shared memory program* P: the system of the n-threaded program P is over-approximated by $[\![P]\!] \triangleq \Gamma_P = (E, Con, \vdash) \triangleq \left(\prod_{i=0}^{n-1} (E_i, Con_i, \vdash_i)\right)\Big\lceil_{\mathbb{M}}$. This compositional, conflict-aware, truly concurrent semantics for multi-threaded shared memory programs, written for modern weak memory architectures is a novel contribution. Our symbolic product event structure faithfully captures the (abstract) semantics of the multi-threaded shared memory program: since it is conflict-aware, this semantics can also distinguish systems at least up to 'failure equivalence' [33], whereas coarser partial order based semantics like [9] can only distinguish systems up to trace equivalence.

AN EXAMPLE: Fig. 4 depicts the event structure related constructs for the example given in Fig. 3. The top row, Fig. 4a, gives the per-thread event structures. The nodes depict the events and the arrows depict the pre-thread enabling relation; the dummy node '$\circ$' is added only to aid the presentation. The solid, black lines depict non-conflicting enablings while the dotted, red lines show the enablings that are conflicting: for instance, in Thread 1, $\emptyset$ enables *both* events Wx_3 and Ry_3[9], while $\{\emptyset, Wx_3, Ry_3\}$ enables only one of Wx_4 or Wx_6. Note that the added local read events RWx_{64} and RWx_{65} are mutually conflicting, and these local reads in turn conflict with the Rx_6 event that could be satisfied externally. The conflict relation for both the threads is given on the right hand side of the diagrams; the symmetries are assumed. The label set is given by (base) names of the SSA variables: that is, $\Sigma = x, y$; the labeling function is a natural one taking the SSA variables to their base name, forgetting the indices. The bottom row, Fig. 4b, gives the event set and conflict relation for our semantics. That is, it gives $\Gamma_P = (E, Con, \vdash) \triangleq \left(\prod_{i=0}^{n-1} (E_i, Con_i, \vdash_i)\right)\Big\lceil_{\mathbb{M}}$. Note that E has only those product events that are present in $\mathbb{M}$; we omit the write events of $\mathbb{M}$ for brevity. In this slightly modified, but equivalent, presentation we included the functoriality condition of the product into the conflict relation. That is, for every variable (as given by the label set Σ), we demand that any read event synchronising with some write event conflicts with the same read event synchronising with any other write event. We conspicuously omit presenting Con as it is an exponential object (in number of events): the elements of Con, apart from being conflict free, are required to read the latest write, and that the reads do not time travel. The enabling relation of the final event structure relates an element C of Con with an element e of E if the per-thread projections of C themselves enable all the participating per-thread events in e.

[9] We are omitting the corresponding internal read RWy_{30} for brevity, which may capture the potential read from initial write Wy_0, capturing a read from an uninitialized value (Ref. section 2.3) on SSA.

4.1 Reachability in Weak Memory Programs

Having defined the semantics of weak memory programs, we now proceed to show how we exploit this semantics to reason about valid executions/reachability of shared memory programs. Let $P = (P_i)$, $0 \leq i < n$ is a shared memory system with n threads. Let $[\![P]\!] \triangleq \Gamma_P = (E, Con, \vdash) = \left(\prod_{i=0}^{n-1} (E_i, Con_i, \vdash_i)\right)\Big\lceil_{\mathbb{M}}$ be an event structure.

Let $\Gamma_{\mathbb{N}} = (E_{\mathbb{N}}, Con_{\mathbb{N}}, \vdash_{\mathbb{N}})$ be an event structure over natural numbers: $E_{\mathbb{N}} \triangleq \mathbb{N}^0$; $Con_{\mathbb{N}} \triangleq \{\emptyset, \{\emptyset, 0\}, \{\emptyset, 0, 1\}, \cdots\}$; $\vdash \triangleq \forall i \in \mathbb{N}^0.\{\emptyset, \cdots, i-1\} \vdash_{\mathbb{N}} i$. We call this event structure a *clock structure*. We would like to exploit the linear ordering provided by the clock structure to 'linearise' events in the product event structure; this linearisation correspond to an execution trace of the system. Naturally, we would like this linearisation to respect the original event enabling order. This requirement is captured using a partial synchronous morphism from Γ_P into $\Gamma_{\mathbb{N}}$. Let $\tau : \Gamma_P \to \Gamma_{\mathbb{N}}$ be such a partial synchronous morphism. Intuitively, such a τ yields a 'linear' execution that honours $\vdash$ and Con. In other words, every match event is mapped (linearised) to an integer clock position in the clock structure. Each such τ yields a valid execution of Γ_P.

Given such a τ, let us now define a per-thread $\tau_i : E_i \to E_{\mathbb{N}}$ as follows: $\forall e \in E, \tau_i(e_i) = \tau(e)$, where $e_i = \pi_i(e)$ if $\texttt{def}(\pi_i(e))$; $\tau_i(e_i)$ is undefined otherwise. By construction each such τ_i yields a valid execution for the thread i. Any per-thread assertion in the original program can then be reasoned using such τ_is: that is, a thread i violates an assertion iff we have a τ_i (from a τ) in which that assertion is, first reached, and then violated. In general, any (finite) reachability related question can be answered using our semantics.

We say that a product event structure *violates an assertion* whenever in (at least) one of its secured maximal configurations, (at least) one of its per-thread event structure's (projected) maximal configuration includes the negation of the said assertion in that (secured) maximal configuration. The following theorem formalises this.

THEOREM 1. *The input program P has an assertion violation iff there exists a maximal $C \in \mathcal{F}(E)$ such that at least one of $\tau_i\big|_C \in \mathcal{F}(E_i)$, $0 \leq i < n$ contains the negation of an assertion, under the specified memory model.*

The proof of the first part of the theorem (sans the memory model) follows directly from construction. Next, we present a proof sketch that addresses memory model specific aspects. Here we focus on TSO; suitable strengthening/weakening (as dictated by PPO) will yield a proof for SC/PSO.

A shared memory execution is said to be a valid TSO execution if it satisfies the following (informally specified) axioms [35, 36]. An execution is TSO iff:

1. *coherence*: Write serialisation is respected.
2. *atomicity*: The atomic load-store blocks behave atomically w.r.t. other stores.
3. *termination*: All stores and atomic load-stores terminate.
4. *value*: Loads always return the latest write.
5. *storeOrd*: Inter-store program order is respected.
6. *loadOrd*: Operations issued after a load always follow the said load in the memory order.

Our semantics omits *termination*. Recall that *clock* denotes the clock order. Intuitively, the clock order represents the memory order. Also, recall that an execution corresponds to a (secured) maximal configuration. We now refer to the product definition (see Fig. 4b), and show how the maximal configuration construction more or less directly corresponds to these axioms.

The *coherence* requirement is a direct consequence of demanding that writes (to the same memory location) are totally ordered with respect to each other. The *atomicity* axiom is enforced by assigning each atomic load-store block the same clock value: this is done by making the atomic block elements as incomparable/equal under PPO. The *value* requirement is taken as is, with time travel restrictions in the definition. The *storeOrd* and *loadOrd* requirements are enforced by PPO, and are captured by the enabling relation. This shows that any valid maximal configuration respects TSO. The converse holds as the product simply uses the Cartesian product of participating per-thread event sets and prunes it to the TSO specification.

This completes our sketch of the proof for TSO. Strengthening of *loadOrd* and weakening of *storeOrd* — via PPO — yield SC and PSO, respectively.

5. Encoding

Let $[\![P]\!] \triangleq (E, Con, \vdash) = \left(\prod_{i=0}^{n-1} (E_i, Con_i, \vdash_i)\right)\Big\lceil_{\mathbb{M}}$ be an event structure. We build a propositional formula Φ that is satisfiable iff the event structure (hence the input program) has an assertion violation; Φ is unsatisfiable otherwise.

The formula Φ will contain the following variables: A Boolean variable X_e (for every $e \in E$), a set of bit-vector variables V_x (for every program variable x) and a set of clock variables $C_{e_{ij}}$ (for every $e_{ij} \in E_i$). Given a per-thread event structure $\Gamma_i = (E_i, Con_i, \vdash_i)$, the (conflict-free) *covering relation* [38] of Γ_i and the PPO_i coincide: informally, given a event structure over E, an event e_1 is covered by another event e_2 if $e_1 \neq e_2$ and no other event can be 'inserted' between them in any configuration. Let us denote this covering relation of the event structure Γ_i as $\vdash_i^i$. Intuitively, $\vdash_i^i$ captures the per-thread *immediate* enabling, aka PPO_i. Let $guard(e)$ denote the guard of the event e.

Let $assert$ be the set of program assertions whose violation we would like to find; these are encoded as constraints over the V_x variables. Each element of $assert$ can be seen as a set of reads, with a set of values expected from those reads, where the guards of all these reads evaluate to true. Equipped with $\vdash_i^i$, $guard(e)$, three types of variables (X_e/V_e/C_e), and a set of $assert$s, we now proceed to define the formula Φ as follows:

$$\Phi \triangleq ssa \wedge ext \wedge succ \wedge m2clk \wedge unique \wedge \neg(\bigwedge_i assert_i)$$

The constituents of Φ are discussed as below.

1. **ssa (ssa):** These constraints include the intra-thread SSA data/-control flow constraints; we rely on our underlying, modified bounded model checker to generate them.
2. **extension (ext):** A match can happen as soon as its $guard$s are set to true, its reads are not reused, and the read has read the $latestw$ write. The $funct$ constraint ensures that once a read is matched, it cannot be reused in any other match; that is, it makes $f : R \to W$ a function. Thus, the ext formula uses the enabling and conflict relation to define left-closed, consistent configurations.
3. **successors ($succ$):** We require that the clocks respect the per-thread immediate enabling relation. This is the first step in ensuring that configurations are correctly enabled and secured.
4. **match2clock ($m2clk$):** A match forces the clock values of a write to precede that of the read (for non-local reads). This ensures that any write that a read has read from has already happened. A match also performs the necessary data-flow between

$$
\begin{array}{rcll}
\mathit{ext} & \triangleq & \bigwedge_{m=(r,w)\in \text{POTMAT};e\in E(r)\cap E(w)} & \Big(X_e \Leftrightarrow \Big(\mathit{latestw}(r,w) \;\wedge\; \mathit{funct}(r,e) \;\wedge\; \big(\mathit{guard}(r)\wedge \mathit{guard}(w)\big)\Big)\Big) \\
\mathit{succ} & \triangleq & \bigwedge_{p\in E_i;p\vdash_i^i q;} & \Big(\mathit{isBefore}(C_p, C_q)\Big) \\
\mathit{m2clk} & \triangleq & \bigwedge_{m=(r,w)\in \text{POTMAT};e\in E(r)\cap E(w)} & \Big(X_e \Rightarrow \;\mathit{isBefore}(C_w, C_r) \;\wedge\; \mathit{isEqual}(V_{[r]}, V_{[w]})\Big) \\
\mathit{latestw}(r,w) & \triangleq & \bigwedge_{(r,w')\in \text{POTMAT};w\neq w'} & \Big((\neg \mathit{isBefore}(C_r, C_{w'}) \wedge \mathit{guard}(w')) \Rightarrow \neg \mathit{isBefore}(C_w, C_{w'})\Big) \\
\mathit{funct}(q,m) & \triangleq & \bigwedge_{q\in R_i;e\in E(q)\setminus m} & \neg X_e
\end{array}
$$

A note on notations: Empty conjunctions are interpreted as 'true' and empty disjunctions as 'false'. We read ';' as *such that*: that is, '$e \in F; e \in E_i$' should be read as '$e \in F$ *such that* $e \in E_i$'. We use the shorthand (r, w) for the unique n-tuple $(\cdots, r, \cdots, w, \cdots)$ in $\mathbb{M}$.

the reads and writes involved: that is, a read indeed picks up the value written by the write. The constraint *m2clk*, together with *succ*, ensures that the *latestw* has the expected, system-wide meaning; they together also lift the per-thread enablings and securings to system-wide enablings and securings.

5. **uniqueness (*unique*):** We require that the clocks of writes that write to the same location are distinct. Since the clock ordering is total, this trivially ensures write serialisation.
6. **clock predicates (*isBefore* and *isEqual*):** The constraints *isBefore* and *isEqual* define the usual $<$ and $=$ over the integers. We use bit-vectors to represent bounded integers.

Let $k \triangleq |R| + |W|$, where $|R|$ and $|W|$ denote the total number of reads and writes in the system. That is, k is the total number of shared read/write events in the input program. The *worst case cost* of our encoding is exactly $\frac{1}{4}k^2 + k \cdot \log k$ Boolean variables; this follows directly from the observation that each entry in the potential-matches relation can be seen as an edge in the bipartite graph with vertex set $E = R \cup W$. Maximising for the edges (matches) in such a bipartite graph yields the $\frac{1}{4}k^2$ component. The $k \log k$ arises from the fact that we need k bit-vectors, each with $\log k$ bits to model the clock variables.

5.1 Soundness and Completeness: $\Phi \leftrightharpoons \Gamma_P$

The following theorems establish the soundness and completeness of the encoding with respect to the assertion checking characterisation introduced in Section 4.1. Note that any imprecision in computing the potential-matches relation will not yield any false positives in Φ, as long as the PPO is exact. This is so even if POTMAT is simply the Cartesian product of reads and writes.

THEOREM 2. *[Completeness: $\Gamma_P \Rightarrow \Phi$]*
For every assertion-violating event structure $\Gamma_P = (E, Con, \vdash) = \Big(\prod_{i=0}^{n-1} (E_i, Con_i, \vdash_i)\Big)\Big\lceil_{\mathbb{M}}$, there exists a satisfying assignment for Φ that yields the required assertion violation.

THEOREM 3. *[Soundness: $\Phi \Rightarrow \Gamma_P$]*
For every satisfying assignment for Φ, there is a corresponding assertion violation in the event structure $\Gamma_P = (E, Con, \vdash) = \Big(\prod_{i=0}^{n-1} (E_i, Con_i, \vdash_i)\Big)\Big\lceil_{\mathbb{M}}$.

We omit the proofs owing to lack of space. Instead, we here provide the intuition behind our encoding. It is easy to see that *ext* and *succ* together 'compute' secured, consistent, per-thread configurations whose initial events are enabled by the empty set: *funct* guarantees every read reads exactly one write and *latestw* ensures that reads always pick up the latest write (as ordered by *succ*) locally. *assert* picks out all secured, consistent configurations that violates any of the assertions. But an assignment satisfying these three constraints needs not satisfy the system-wide *latest write* requirement. This is addressed by the *m2clk* constraint: this constraint 'lifts' the per-thread orderings to a valid inter-thread total ordering; it also performs the necessary value assignments. Equipped with *m2clk*, the *latestw* now can pick the writes that are indeed latest across the system. The transitivity of *isBefore* (in *succ*, *latestw*, *m2clk*) correctly 'completes' the per-thread orderings to an arbitrary number of threads, involving arbitrary configurations of arbitrary matches.

6. Evaluation

We have implemented our approach in a tool named WALCYRIE[10] using CBMC [21] version 5.0 as a base. The tool currently supports the SC, TSO and PSO memory models. WALCYRIE takes a C/C++ program and a loop bound k as input, and transforms the input program into a loop-free program where every loop is unrolled at least k times. From this transformed input program we extract the SSA and the PPO and POTMAT relations. Using these relations, and the implicit event structure they constitute, we build the propositional representation of the input program. This propositional formula is then fed to the MiniSAT back-end to determine whether the input program violates any of the assertions specified in the program. If a satisfying assignment is found, we report assertion violation and present the violating trace; otherwise, we certify the program to be bug free, up to the specified loop bound k.

We use two large, well established, widely used benchmark suites to evaluate the efficacy of WALCYRIE: the Litmus tests from [10] and SV-COMP 2015 [16] benchmarks. We compare our work against the state of the art tool to verify real-world weak memory programs, [9]; hereafter we refer it as CBMC-PO. We remark that ([9], page 3) employs the term "event structures" to mean the per-processor total order of events, as dictated by PO. This usage is unrelated to *our* event structures; footnote #4 of [9] clarifies this point. We run all our tests with *six* as the unrolling bound and 900 s as the timeout. Our experiments were conducted on a 64-bit 3.07 GHz Intel Xeon machine with 48 GB of memory running Linux. Out tool is available at `https://github.com/gan237/walcyrie`; the URL provides all the sources, benchmarks and automation scripts needed to reproduce our results.

Litmus tests [10] are small (60 LOC) programs written in a toy shared memory language. These tests capture an extensive range

[10] A hark back to the conflict-friendly Norse decision makers; also an anagram of **W**eak memory **A**na**L**ysis using **C**onfl**I**ct aware t**R**u**E** concurrenc**Y**.

of subtle behaviours that result from non-intuitive weak memory interactions. We translated these tests into C and used WALCYRIE to verify the resulting C code. The Litmus suite contains 5804 tests and we were able to correctly analyse all of them in under 5 s.

The SV-COMP's concurrency suite [16] contains a range of weak-memory programs that exercise many aspects of different memory models, via the pthread library. These include crafted benchmarks as well as benchmarks that are derived from real-world programs: including device drivers, and code fragments from Linux, Solaris, NetBSD and FreeBSD. The benchmarks include non-trivial features of C such as bitwise operations, variable/function pointers, dynamic memory allocation, structures and unions. The SC/TSO/PSO part of the suite has 600 programs; please refer to [16] for the details. WALCYRIE found a handful of misclassifications (that is, programs with defects that are classified as defect-free) among the benchmarks; there were no misclassifications in the other direction, that is, all programs classified by the developers as defective are indeed defective. Such misclassifications are a strong indication that spotting untoward weak memory interactions is tricky even for experts. We have reported these misclassifications to the SV-COMP organisers.

The work that is closest to us is CBMC-PO [9]: like us, they use BMC based symbolic execution to find assertion violations in C programs. The key insight here is that executions in weak memory systems can be seen as partial orders (where pair of events relaxed are incomparable). Based on this, they developed a partial order based decision procedure. Like us, they rely on SAT solvers to find the program defects. But our semantics is conflict-aware, consequently the resulting decision procedure is also different; the formula generation complexity for both approaches is cubic and both of us generate quadratic number of Boolean variables. The original implementation provided with [9] handled thread creation incorrectly. We modified CBMC-PO to fix this, and we use this corrected version in all our experiments. Though the worst case complexity of both approaches is the same, our true concurrency based encoding is more compact: WALCYRIE often produced nearly half the number of Boolean variables and about 5% fewer constraints compared to CBMC-PO (after 3NF reduction).

Our results are presented as scatter plots, comparing the total execution times of WALCYRIE and CBMC-PO: this includes parsing, constraint generation and SAT solving times; the smaller the time, the better the approach. The x axis depicts the execution times for WALCYRIE and the y axis depicts the same for CBMC-PO. The SAT instances are marked by a triangle (▲) and the UNSAT instances are marked by a cross ($\times$). The first set of plots (Fig. 5a, Fig. 5b and Fig. 5c) presents the data for the Litmus tests. There are three plots and each corresponds to the memory model against which we tested the benchmarks. Both WALCYRIE and CBMC-PO solve these small problem instances fairly quickly, typically in under 5 s; recall that individual Litmus tests are made of only tens of LOC. But CBMC-PO appears to have a slight advantage: we investigated this and found that CBMC-PO's formula generation was quicker while the actual SAT solving times were comparable. WALCYRIE's current implementation has significant scope for improvement: for instance, the *latestw* and *funct* constraint generation could be integrated into one loop; also, the match generation could be merged with the constraint generation.

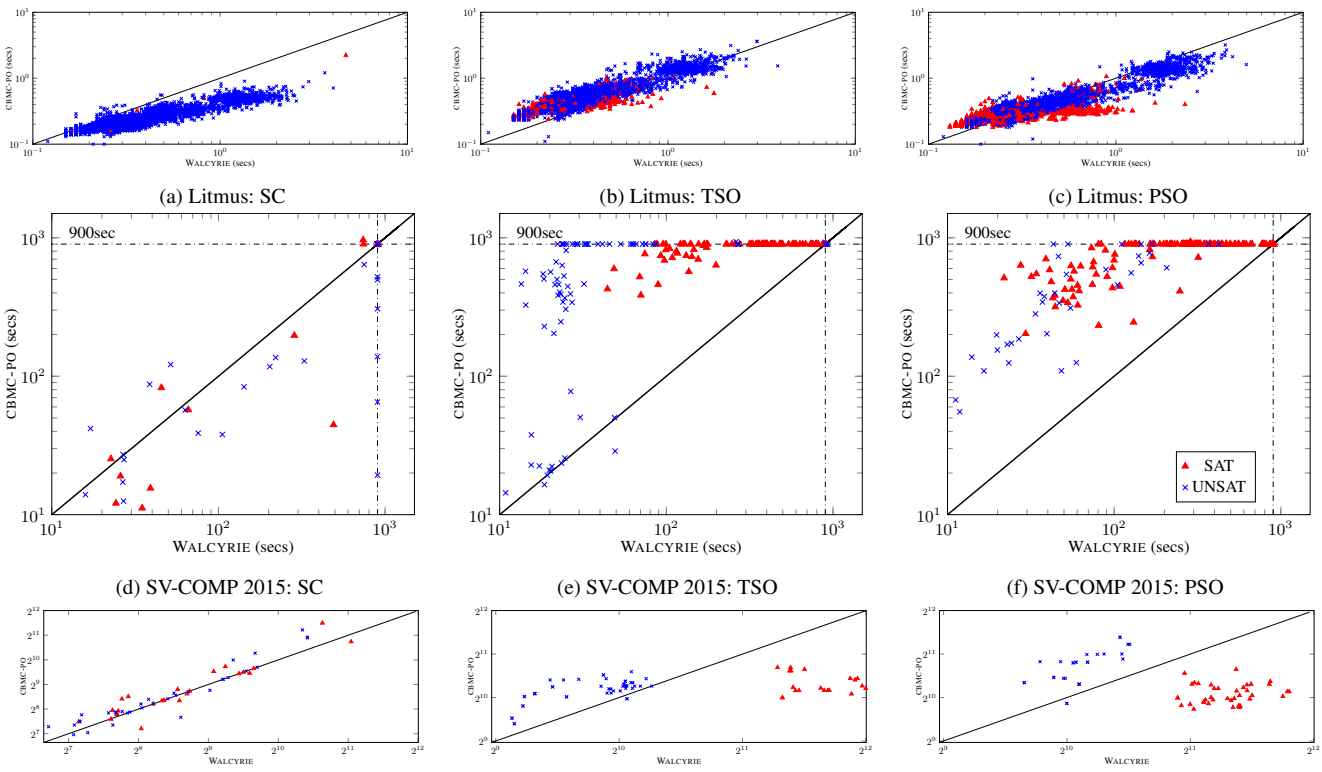
The scatter plot Fig. 5d compares the runtimes of SC benchmarks. Under SC, the performance of CBMC-PO and WALCYRIE appears to be mixed: there were seven (out of 125) UNSAT instances where WALCYRIE times out while CBMC-PO completes the analysis between 10 and 800 s. In the majority of the cases, the performance is comparable and the measurements cluster around the diagonal. Note that *no* modern multiprocessor offers SC, and the SC results are presented for sake of completeness.

Fig. 5e presents the results of SV-COMP's TSO benchmarks. The situation here is markedly different, compared to the Litmus tests and SC. Here, WALCYRIE clearly outperforms CBMC-PO, as indicated by the densely populated upper triangle. This is contrary to usual intuition: TSO, being weaker than SC, usually yields a larger state space, and the conventional wisdom is that the larger the state space, the slower the analysis. Our results appear to contradict this intuition. The same trend is observed in the PSO section (Fig. 5f) of the suite as well. In fact, WALCYRIE outperforms CBMC-PO over *all* inputs.

We investigated this further by looking deeper into the inner-workings of the SAT solver. SAT solvers are complex engineering artefacts and a full description of their internals is beyond the scope of this article; interested readers could consult [23, 25]. Briefly, SAT solvers explore the state space by making decisions (on the truth value of the Boolean variables) and then propagating these decisions to other variables to cut down the search space. If a propagation results in a conflict, the solver backtracks and explores a different decision. There is a direct trade-off between propagations and conflicts, and a good encoding balances these concerns judicially. To this end, we introduce a metric called *exploration efficacy*, ρ, defined as the ratio between the total number of propagations ($prop$) to the total number of conflicts ($conf$). That is, $\rho \triangleq prop/conf$. The numbers $prop$ and $conf$ are gathered only for those benchmarks where both the tools provided a definite SAT/UNSAT answer. Intuitively, one would expect SAT instances to have a higher ρ, while UNSAT instances are expected to have a lower ρ. To find a satisfying assignment, one needs to propagate effectively (without much backtracking) and to prove unsatisfiability, one should run into as many conflicts as early as possible. Thus, for SAT instances, a higher ρ is indicative of an effective encoding; the converse holds true for UNSAT instances.

Figs. 5g to 5i present the scatter plots for ρ for CBMC-PO and WALCYRIE, for three of our memory models. For SC, the ρ values are basically the same. This explains why we observed a very similar performance under SC. The situation changes with TSO and PSO. The clustering of ρ values on the either side of the diagonal hints at the reason behind the superior performance of our conflict-aware encoding. In both TSO and PSO, our ρ values for the SAT instances are two to four times higher; our ρ values for the UNSAT instances are one to two times lower. In PSO, the $prop$ values increase (as the state space grows with weakening) and the number of conflicts $conf$ also grow in tandem, unlike CBMC-PO. We conjecture that this is the reason for the performance gain as we move from TSO to PSO using WALCYRIE.

At the encoding level, the ρ values can be explained by the way WALCYRIE exploits the control and data-flow conflicts in the program. Since CBMC-PO is based on partial orders (which lack the notion of conflict), their encoding relies heavily on SAT solver *eventually* detecting a conflict. That is, CBMC-PO resolves the control and data conflicts *lazily*. By contrast, WALCYRIE exploits the conflict-awareness of general event structures to develop an encoding that handles conflicts *eagerly*: branching time objects like event structures are able to tightly integrate the control/data choices, resulting in faster conflict detection and faster state space pruning. For instance, our *funct* constraint (stemming from the the requirement that morphisms be functions) ensures that once a read (r) is satisfied by a write w (that is, when X_{rw} is set to true; equally, $f(r) = w$), all other matches involving the r (say, $X_{rw'}$) are invalidated immediately (via unit propagation). This, along with the equality in *ext*, ensures that any conflicts resulting from *guard* and *latestw* are also readily addressed simply by unit propagation. In CBMC-PO, this conflict (that $X_{rw'}$ cannot happen together with X_{rw}) is not immediately resolved/learnt and the SAT solver is let to explore infeasible paths until it learns the conflict sometime in the



(a) Litmus: SC (b) Litmus: TSO (c) Litmus: PSO

(d) SV-COMP 2015: SC (e) SV-COMP 2015: TSO (f) SV-COMP 2015: PSO

(g) SC: exploration efficacy (ρ) (h) TSO: exploration efficacy (ρ) (i) PSO: exploration efficacy (ρ)

Figure 5: Evaluating WALCYRIE against CBMC-PO

future. Thus, our true concurrency based, conflict-aware semantics naturally provides a compact, highly effective decision procedure.

7. Related Work

We give a brief overview of work on program verification under weak memory models with particular focus on assertion checking. Finding defects in shared memory programs is known to be a hard problem. It is non-primitive recursive for TSO and it is undecidable if read-write or read-read pairs can be reordered [12]. Avoiding causal loops restores decidability but relaxing write atomicity makes the problem undecidable again [13].

Verifiers for weak memory broadly come in two flavours: the "operational approach", in which buffers are modelled concretely [3, 17, 26, 29, 30], and the "axiomatic approach", in which the observable effects of buffers are modelled indirectly by (axiomatically) constraining the order of instruction executions [2, 9, 11, 15]. The former often relies on interleaving semantics and employs transition systems as the underlying mathematical framework. The later relies on independence models and employs partial orders as the mathematical foundation. The axiomatic method, by abstracting away the underlying complexity of the hardware, has been shown to enable the verification of realistic programs. Although we do *not* use partial orders, our true concurrency based approach falls under the axiomatic approach.

These two approaches have been used to solve two distinct, but related problems in weak memory. The first one is finding assertion violations that arise due to the intricate semantics of weak memory; this is the problem we address as well. The other is the problem of fence insertion. Fence insertion presents two further sub-problems: the first is to find a (preferably minimal, or small enough) set of fences that needs to be inserted into a weak memory program to make it sequentially consistent [7, 11, 17]; the second is to find a set of fences that prevents any assertion violations caused by weak memory artefacts [4, 19, 24, 29].

There are three works — [2, 8, 9] — that are very close to ours. The closest to our work, [9], was discussed in Section 6. Nidhugg [2] is promising but can only handle programs without data nondeterminism. The work in [8] uses code transformations to transform the weak memory program into an equivalent SC program, and uses SC-based to tools to verify the original program.

There are further, more complex memory models. Our approach can be used directly to model RMO. However, we currently cannot handle POWER and ARM without additional formalisms. Recent work [14] studies the difficulty of devising an axiomatic memory model that is consistent with the standard compiler optimizations for C11/C++11. Such fine-grained handling of desirable/undesirable thin-air executions is outside of the scope of our work.

8. Conclusion

We presented a bounded static analysis that exploits a conflict-aware true concurrency semantics to efficiently find assertion violations in modern shared memory programs written in real-world languages like C. We believe that our approach offers a promising line of research: exploiting event structure based, truly concurrent semantics to model and analyse real-world programs. In the future, we plan to investigate more succinct intermediate forms like Shasha-Snir traces to cover the Java or C++11 memory model and to study other match-related problems such as lock/unlock or malloc/free.

Acknowledgements We would like to thank the reviewers and Michael Emmi for their constructive input that significantly improved the final draft. Ganesh Narayanaswamy is a Commonwealth Scholar, funded by the UK government. This work is supported by ERC project 280053.

References

[1] Debate'90: An electronic discussion on true concurrency. In Vaughan Pratt, Doron A. Peled, and Gerard J. Holzmann, editors, *DIMACS Workshop on Partial Order Methods in Verification*, 1997.

[2] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos F. Sagonas. Stateless model checking for TSO and PSO. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2015.

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Counter-example guided fence insertion under TSO. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2012.

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Memorax, a precise and sound tool for automatic fence insertion under TSO. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2013.

[5] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 1996.

[6] Alfred V. Aho, M. R. Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM Journal of Computing*, 1972.

[7] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence – A static analysis approach to automatic fence insertion. In *International Conference on Computer Aided Verification (CAV)*, 2014.

[8] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. Software verification for weak memory via program transformation. In *European Conference on Programming Languages and Systems (ESOP)*, 2012.

[9] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *International Conference on Computer Aided Verification (CAV)*, 2013.

[10] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: Running tests against hardware. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2011.

[11] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models (extended version). *Formal Methods in System Design*, 40(2), 2012.

[12] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. In *Symposium on Principles of Programming Languages (POPL)*, 2010.

[13] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. What's decidable about weak memory models? In *European Conference on Programming Languages and Systems (ESOP)*, 2012.

[14] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The problem of programming language concurrency semantics. In *European Conference on Programming Languages and Systems (ESOP)*, 2015.

[15] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *Symposium on Principles of Programming Languages (POPL)*, January 2011.

[16] Dirk Beyer. Software verification and verifiable witnesses (report on SV-COMP 2015). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2015.

[17] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against TSO. In *European Conference on Programming Languages and Systems (ESOP)*, 2013.

[18] Howard Bowman and Rodolfo Gomez. *Concurrency Theory: Calculi an Automata for Modelling Untimed and Timed Concurrent Systems*. 2005.

[19] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *Programming Language Design and Implementation (PLDI)*, 2007.

[20] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, July 2001.

[21] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2004.

[22] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Design Automation Conference*, 2003.

[23] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Chapter 2, satisfiability solvers. In *Handbook of Knowledge Representation*. 2008.

[24] Saurabh Joshi and Daniel Kroening. Property-driven fence insertion using reorder bounded model checking. In *International Symposium on Formal Methods (FM)*, LNCS, 2015.

[25] Hadi Katebi, Karem A. Sakallah, and João P. Marques-Silva. Empirical study of the anatomy of modern SAT solvers. In *Theory and Application of Satisfiability Testing (SAT)*, 2011.

[26] Michael Kuperstein, Martin Vechev, and Eran Yahav. Partial-coherence abstractions for relaxed memory models. *SIGPLAN Notices*, June 2011.

[27] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transaction on Computing*, 1979.

[28] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *Languages and Compilers for Parallel Computing*, 1997.

[29] Alexander Linden and Pierre Wolper. A verification-based approach to memory fence insertion in PSO memory systems. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2013.

[30] Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. In *Programming Language Design and Implementation (PLDI)*, 2012.

[31] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA, 1997.

[32] Vaughan Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Program*, (1), February 1986.

[33] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall International Series in Computer Science. 1998.

[34] A.W. Roscoe. *Understanding Concurrent Systems*. 1st edition, 2010.

[35] Pradeep Sindhu, Michel Cekleov, and Jean-Marc Frailong. Formal specification of memory models. Technical Report CSL-91-11, Xerox, 1991.

[36] SPARC International, Inc. *The SPARC Architecture Manual: Version 8*. Upper Saddle River, NJ, USA, 1992.

[37] Rob J. van Glabbeek and Frits W. Vaandrager. Bundle event structures and CCSP. In *International Conference on Concurrency Theory (CONCUR)*, 2003.

[38] Glynn Winskel. Event structure semantics for CCS and related languages. In *International Colloquium on Automata, Languages and Programming (ICALP)*, 1982.

[39] Glynn Winskel. Event structures. In *Advances in Petri Nets*, 1986.

[40] Richard N. Zucker and Jean-Loup Baer. A performance study of memory consistency models. In *International Symposium on Computer Architecture*, 1992.