Automatically Comparing Memory Consistency Models

John Wickerson
Imperial College London, UK
j.wickerson@imperial.ac.uk

Mark Batty
University of Kent, UK
m.j.batty@kent.ac.uk

Tyler Sorensen
Imperial College London, UK
t.sorensen15@imperial.ac.uk

George A. Constantinides
Imperial College London, UK
g.constantinides@imperial.ac.uk

Abstract

A *memory consistency model* (MCM) is the part of a programming language or computer architecture specification that defines which values can legally be read from shared memory locations. Because MCMs take into account various optimisations employed by architectures and compilers, they are often complex and counterintuitive, which makes them challenging to design and to understand.

We identify four tasks involved in designing and understanding MCMs: generating conformance tests, distinguishing two MCMs, checking compiler optimisations, and checking compiler mappings. We show that all four tasks are instances of a general constraint-satisfaction problem to which the solution is either a program or a pair of programs. Although this problem is intractable for automatic solvers when phrased over programs directly, we show how to solve analogous constraints over program *executions*, and then construct programs that satisfy the original constraints.

Our technique, which is implemented in the Alloy modelling framework, is illustrated on several software- and architecture-level MCMs, both axiomatically and operationally defined. We automatically recreate several known results, often in a simpler form, including: distinctions between variants of the C11 MCM; a failure of the 'SC-DRF guarantee' in an early C11 draft; that x86 is 'multi-copy atomic' and Power is not; bugs in common C11 compiler optimisations; and bugs in a compiler mapping from OpenCL to AMD-style GPUs. We also use our technique to develop and validate a new MCM for NVIDIA GPUs that supports a natural mapping from OpenCL.

Categories and Subject Descriptors C.1.4 [*Processor Architectures*]: Parallel processors; D.3.4 [*Programming Languages*]: Compilers; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

Keywords C/C++, constraint solving, graphics processor (GPU), model checking, OpenCL, program synthesis, shared memory concurrency, weak memory models

1. Introduction

In the specification of a concurrent programming language or a parallel architecture, the *memory consistency model* (MCM) defines which values can legally be read from shared memory locations [4]. MCMs have to be general enough to enable portability, but specific enough to enable efficient implementations. They must also admit optimisations employed by architectures (such as store buffering and instruction reordering [30]) and by compilers (such as common subexpression elimination and constant propagation [7]). This profusion of design goals has led to MCMs for languages (such as C11 [38] and OpenCL 2.0 [42]), for CPU architectures (such as x86, ARM, and IBM Power), and for GPU architectures (such as AMD and NVIDIA), that are complicated and counterintuitive. In particular, all of these MCMs permit executions that are not *sequentially consistent* (SC), which means that they do not correspond to a simple interleaving of concurrent instructions [45]. As a result, designing and reasoning about MCMs is extremely challenging.

Responding to this challenge, researchers have built numerous automatic tools (see §8). These typically address the question of whether a program P, executed under an MCM M, can reach the final state σ. Put another way: can the *litmus test* (P, σ) *pass* under M? While useful, there are several other questions whose answers are valuable for MCM reasoning and development. Four that have appeared frequently in the literature are:

Q1 Which programs can be run to test whether a compiler or machine conforms to a given MCM? [11, 24]

Q2 Is one MCM more permissive than another? That is, is there a litmus test that can pass under one but must fail under the other? [11, 14, 43, 48, 50, 58, 63]

Q3 Can 'strengthening' a program (syntactically) ever enable additional behaviours? For instance, can we take a litmus test that must fail, impose additional sequencing or dependencies between its instructions (or, in the C11 case, give an atomic operation a stronger 'memory order' [38 (§7.17.3)]), and thereby allow it to pass? [21, 22, 55, 77, 79, 80]

Q4 Is a given software/architecture compiler mapping correct? Or is there a litmus test that must fail under the software-level MCM, but which, when compiled, can pass under the architecture-level MCM? [15, 16, 46, 47, 81, 82]

1.1 Key Idea 1: Generalising the Question

Our first key idea is the observation that all four of the questions listed above can be answered, sometimes positively and sometimes negatively, by exhibiting programs P and Q and state σ such that the litmus test (P, σ) *must fail* under a given MCM M, P and Q

$$\begin{tikzcd}
 & (P,\sigma) \arrow[rr, "\blacktriangleright"] \arrow[dl] \arrow[d] & & (Q,\sigma) \arrow[dl] \arrow[d] \\
X' & X \notin \mathit{consistent}_M \arrow[rr, bend left, "\triangleright"] & Y' & Y \in \mathit{consistent}_N
\end{tikzcd}$$

Figure 1. Diagram for explaining Key Idea 2

are related by a given binary relation $\blacktriangleright$ on programs, and (Q, σ) *can pass* under a given MCM N. That is, each corresponds to finding inhabitants of the following set.

Definition 1 (General problem). $g(M, N, \blacktriangleright) \overset{\text{def}}{=}$

$$\{(P, Q, \sigma) \mid \sigma \notin \mathbf{obs}_M(P) \wedge P \blacktriangleright Q \wedge \sigma \in \mathbf{obs}_N(Q)\}$$

where **obs** returns the set of final states that can be observed after executing a given program under a given MCM.[1]

Our four questions correspond to the following specialisations of g's parameters:

Q1 $g(M, 0, id)$ consists of tests that check conformance to M, where 0 is the MCM that allows all executions and id is the identity relation;

Q2 $g(M, N, id)$ consists of tests that cannot pass under M but can under N;

Q3 $g(M, M, \text{'is weaker than'})$ consists of monotonicity violations in M, given a relation that holds when one program is (syntactically) 'weaker than' another; and

Q4 $g(M, N, \text{'compiles to'})$ consists of bugs in a compiler mapping from M to N, given a relation that holds when one program 'compiles to' another.

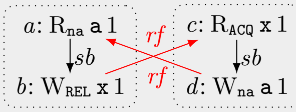
1.2 Key Idea 2: Constraining Executions, not Programs

Having captured in Def. 1 our four questions, we now ask whether answers can be generated using an automatic constraint solver. Unfortunately, Def. 1 is problematic because showing that a litmus test must fail under M requires universal quantification over executions. That is, we must show that there *exists* a litmus test (P, σ) such that *all* possible executions of P under M do not reach σ. As Milicevic et al. observe, 'this "$\exists\forall$" quantifier pattern is a particularly difficult instance of higher-order quantification' [52], and our pilot experiments confirmed it to be intractable in practice.

Our second 'key idea', then, involves rephrasing the constraints to be not over *programs*, but over *program executions*, where they become much cheaper to solve (see Def. 2, below), and then to recover litmus tests from these executions. We explain how this is possible with reference to Fig. 1.

We start by finding individual executions X and Y, such that X is inconsistent under M, Y is consistent under N, and $X \triangleright Y$. From X, we construct a litmus test (P, σ) that can behave like X, and from Y, we construct a litmus test (Q, σ) that can behave like Y. The $\triangleright$ relation between X and Y ensures that these litmus tests will have the same final state σ and that $P \blacktriangleright Q$. We now have a litmus test that *can fail* under M and another that *can pass* under N. But g requires a litmus test that *must fail* under M, and perhaps (P, σ) can still pass by taking a different execution X'. Example 1 illustrates how such a situation can arise.

Example 1. The diagram below (left) depicts a C11 execution. Dotted rectangles group events into threads; W and R denote write and read events; `REL` and `ACQ` denote atomic 'release' and 'acquire' accesses; `na` denotes a non-atomic memory access; rf is the 'reads from' relation; and sb means 'sequenced before'. We extract a litmus test from this executions (below right), by mapping write events to store instructions, reads to loads, and having the final state enforce the desired rf relation.



```
int a=0; atomic_int x=0;
  r0=a; x.store(1,REL);

  r1=x.load(ACQ); a=1;
r0==1 && r1==1
```

The execution is deemed inconsistent in C11. This is because the successful release/acquire synchronisation implies that b 'happens-before' c, hence that a happens-before d, and hence that it is impossible for a to read from d. As a result, it is tempting to use the litmus test derived from this execution for conformance testing (**Q1**). In fact, the litmus test is not useful for this purpose because it has a data race: the non-atomic store to `a` goes ahead even if the release/acquire synchronisation on `x` fails. Racy C11 programs have undefined semantics, so non-conformance cannot be deduced from the test passing.

To guard against situations like the one above, we require that X is also 'dead'. Semantically, X is dead if whenever X is inconsistent, and (P, σ) is a 'minimal' litmus test constructed from X, then no execution of P that leads to σ is allowed (which implies, in particular, that P is race-free). This ensures that P not only *can fail*, but *must fail*, as required. We obtain semantic deadness via a syntactic approximation (dead_M), which simply involves a few MCM-specific constraints on the shape of X. For instance, we require that whenever two non-atomic accesses are prevented from racing by release/acquire synchronisation (as a and d are in Example 1), one of the accesses must have a control dependency on the acquire event. That is, if we add a control dependency edge from c to d, then this execution would be in dead_M. This ensures semantic deadness because the resultant litmus test, now having `if(r1==1) a=1` rather than just `a=1`, is now race-free. It therefore becomes a useful conformance test.

Formally, we reduce our general constaint-solving problem to finding inhabitants of the following set.

Definition 2 (General problem over executions). $\tilde{g}(M, N, \triangleright) \overset{\text{def}}{=}$

$$\{(X, Y) \in \mathbb{X}^2 \mid X \notin \mathit{consistent}_M \wedge \\ X \in \mathit{dead}_M \wedge X \triangleright Y \wedge Y \in \mathit{consistent}_N\}.$$

Analogies to our four specific problems, **Q1**–**Q4**, can be obtained by specialising $\tilde{g}$'s parameters like we did in §1.1.

We remark that deadness ensures the *soundness* of our solving strategy, but because we obtain semantic deadness via a syntactic approximation, it may spoil its *completeness*. Nonetheless, although our technique is incomplete, we demonstrate in the following subsection that it is *useful*.

1.3 Applications

We have implemented our technique in the Alloy modelling framework [39]. An Alloy model comprises a set of classes plus a set of constraints that relate objects and fields in those classes. If further provided with upper bounds on the number of objects in each class, Alloy can compile the constraints down to a SAT query, and then invoke a SAT solver to search for a satisfying instance.

We have applied our technique to a range of MCMs, including both software-level and architecture-level MCMs, both CPU and GPU varieties, and both operationally-defined and axiomatically-defined. Our results fall into two categories: automatic recreations of results that have previously been manually generated, and new results.

[1] Elements of $g(M, N, \blacktriangleright)$ can also be seen as counterexamples to $\blacktriangleright$ implying observational refinement [33]: $P \blacktriangleright Q \not\Rightarrow \mathbf{obs}_N(Q) \subseteq \mathbf{obs}_M(P)$.

Recreated results We have rediscovered litmus tests that witness:

- the impact of three proposed changes to the C11 axioms (§4.1, §4.2, §4.3) – and our distinguishing litmus tests are substantially simpler than the originals in two cases;
- a violation of the supposedly guaranteed sequentially-consistent semantics for data-race-free programs (the 'SC-DRF guarantee' [5]) in a early draft of the C11 standard (§4.5) – similar to that reported by Batty et al. [16];
- that x86 is 'multi-copy atomic' [23, 72] but Power is not (§4.4);
- the C11 MCM behaving non-monotonically, by allowing tests to pass only if sequencing is added or a memory order is strengthened (§5) – and in the second case, our litmus test is simpler than that found manually by Vafeiadis et al. [77]; and
- two bugs in a published compiler mapping from OpenCL to AMD-style GPUs [62] (§6.1) – one of which is substantially simpler than the original found by Wickerson et al. [82].

New results Our main new result (§6.2) concerns the mapping of OpenCL to PTX, an assembly-like language for NVIDIA GPUs [61]. We first use **Q4** to show that a 'natural' OpenCL/PTX compiler mapping is unsound for an existing formalisation of the PTX MCM by Alglave et al. [9], but sound for a stronger PTX MCM that we propose. We then use **Q2** to generate litmus tests that distinguish the two PTX MCMs, which we use to validate our stronger MCM experimentally against actual NVIDIA GPUs.

In summary, we make the following contributions.

1. We show that four frequently-asked questions about MCMs can be viewed as instances of one general formula (g, Def. 1).
2. We rephrase the formula to constrain executions rather than litmus tests ($\tilde{g}$, Def. 2), so that it can be tractably explored using a constraint-solving tool.
3. We implement our approach in Alloy, and use it to automatically reproduce several results obtained manually in previous work, often finding simpler examples.
4. We present a new, experimentally-validated MCM for PTX, and an OpenCL/PTX compiler mapping, and use Alloy to validate the mapping against the PTX MCM.

Our supplementary material [1] contains our Alloy models and our PTX testing results.

2. Executions

The semantics of a program is a set of *executions*. This section describes our formalisation of executions, both in general (§2.1) and specifically for C11, OpenCL, and PTX (§2.2), and then explains how MCMs decide which executions are allowed (§2.3).

Program executions are composed of *events*, each representing the execution of a program instruction. Most existing MCM frameworks embed several pieces of information within each event, such as the location it accesses, the thread it belongs to, and the value it reads and/or writes (e.g., [5, 11, 16, 55, 68]). In our work, events are *pure*, in Needham's sense [57]: they are given meaning simply by their membership of, for instance, the 'read events' set, or the 'same location' relation. This formulation brings three benefits. First, it means that we can easily build a hierarchy of executions to unify several levels of abstraction. For instance, starting with a top-level 'execution' class, we can obtain a class of 'C11 executions' just by adding extra fields such as 'the set of events with acquire semantics'; we need not modify the type of events. Second, it means that the same events can appear (with different meanings) in two executions, thus reducing the total number of events needed in our search space. Third, we avoid the need to define (and therefore, in a bounded search query, set the number of) locations, threads, and values.

E	$\subseteq \mathbb{E}$	all events in the execution
$R,\ W$	$\subseteq \mathbb{E}$	events that read (resp. write) a location
F	$\subseteq \mathbb{E}$	fence events
nal	$\subseteq \mathbb{E}$	events that access a non-atomic location
sb	$\subseteq \mathbb{E}^2$	sequenced-before
$ad,\ cd,\ dd$	$\subseteq \mathbb{E}^2$	address/control/data dependency
$sthd,\ sloc$	$\subseteq \mathbb{E}^2$	same thread, same location
$rf,\ co$	$\subseteq \mathbb{E}^2$	reads-from, coherence order
rfe	$\subseteq \mathbb{E}^2$	$\overset{\text{def}}{=} rf - sthd$

❶ $R \cup W \cup F \cup nal \subseteq E$ ❷ $(R \cup W) \not\cap F$ ❸ $sb \subseteq sthd$

❹ $strict\text{-}po(sb)$ ❺ $ad \subseteq [R] \,;\, sb \,;\, [R \cup W]$

❻ $dd \subseteq [R] \,;\, sb \,;\, [W]$ ❼ $cd \subseteq [R] \,;\, sb$ ❽ $equiv(sthd, E)$

❾ $equiv(sloc, R \cup W)$ ❿ $nal \,;\, sloc = nal$

⓫ $rf \subseteq (W \times R) \cap sloc$ ⓬ $rf \,;\, rf^{-1} \subseteq id$ ⓭ $strict\text{-}po(co)$

⓮ $co \cup co^{-1} = (W - nal)^2 \cap sloc - id$

Figure 2. Basic executions, $\mathbb{X}$

Notation Our MCMs are written in Alloy's modelling language, but in this paper we opt for more conventional set-theoretic notation. For a binary relation r, $\overline{r}$ is its complement, r^{-1} is its inverse, $r^?$ is its reflexive closure, r^+ is its transitive closure, and r^* is its reflexive, transitive closure. The $strict\text{-}po$ predicate holds of binary relations that are acyclic and transitive, and $equiv(r, s)$ holds when r is a subset of s^2, reflexive over s, symmetric, and transitive. We compose an m-ary relation r_1 with an n-ary relation r_2 (where $m, n \geq 1$) via $r_1 \,;\, r_2 \overset{\text{def}}{=} \{(x_1, \ldots, x_{m-1}, z_1, \ldots, z_{n-1}) \mid \exists y.\, (x_1, \ldots, x_{m-1}, y) \in r_1 \wedge (y, z_1, \ldots, z_{n-1}) \in r_2\}$, and we lift a set to a subset of the identity relation via $[s] \overset{\text{def}}{=} \{(e, e) \mid e \in s\}$. We write $imm(r)$ for $r - (r \,;\, r^+)$, and $s_1 \not\cap s_2$ for $s_1 \cap s_2 = \emptyset$.

2.1 Basic Executions

Let $\mathbb{E}$ be a set of events.

Definition 3 (Basic executions)**.** The set $\mathbb{X}$ of basic executions comprises structures with fourteen fields, governed by well-formedness constraints (Fig. 2). We write f_X for the field f of execution X, omitting the subscript when it is clear from the context. The constraints can be understood as follows. The subsets R, W, F, and nal are all drawn from the events E that appear in the execution ❶. In particular, compound read-modify-write (RMW) events belong to *both* R and W. Fences are distinct from reads and writes ❷. *Sequenced before* is an intra-thread strict partial order ❸ ❹. (We allow sb within a thread to be partial because in C-like languages, the evaluation order of certain components, such as the operands of the +-operator, is not specified.) Address dependencies are either read-to-read or read-to-write ❺, data dependencies are read-to-write ❻, and control dependencies originate from reads ❼. The $sthd$ relation forms an equivalence among all events ❽, while $sloc$ forms an equivalence among reads and writes ❾. We allow a distinction between 'atomic' and 'non-atomic' locations; MCMs that do not make this distinction (such as architecture-level MCMs) simply constrain the set of non-atomic locations to be empty. The nal set consists only of complete $sloc$-classes ❿. A relation rf is a well-formed *reads-from* if it relates writes to reads at the same location ⓫ and is injective ⓬. The inter-thread reads-from (rfe) is derived from rf. A relation co is a well-formed *coherence order* if when restricted to writes on a single atomic location it forms a

$A \subseteq \mathbb{E}$ atomic events
$acq, rel \subseteq \mathbb{E}$ events that have acquire (resp. release) semantics
$sc \subseteq \mathbb{E}$ events that have SC semantics

⓯ $acq \cup rel \cup sc \cup (R \cap W) \cup F \subseteq A \subseteq E$

⓰ $R \cap sc \subseteq acq \subseteq R \cup F$ ⓱ $W \cap sc \subseteq rel \subseteq W \cup F$

⓲ $F \cap sc \subseteq acq \cap rel$ ⓳ $R - A \subseteq nal \subseteq E - A$

Figure 3. C11 executions, $\mathbb{X}_{\text{C11}}$ (extending $\mathbb{X}$)

$dv \subseteq \mathbb{E}$ events that have whole-device scope
$swg \subseteq \mathbb{E}^2$ same workgroup

⓴ $sthd \subseteq swg$ ㉑ $equiv(swg, E)$ ㉒ $dv \subseteq A$

Figure 4. OpenCL executions, $\mathbb{X}_{\text{OpenCL}}$ (extending $\mathbb{X}_{\text{C11}}$)

strict total order; that is, it is acyclic and transitive ⓭, and it relates every pair of distinct writes to the same atomic location ⓮.

Remark 4. We emphasise that elements of $\mathbb{E}$ have no intrinsic structure, only identity. Nevertheless, when drawing executions, we tag events with their type: R for elements of $R - W$, W for elements of $W - R$, and C (for 'compound') for elements of $R \cap W$. We indicate $sthd$ equivalence classes with dotted rectangles, and $sloc$ equivalence classes using named representatives, e.g. `x` and `y`. We also tag events with the values read/written, but this is purely for readability.

Remark 5 (Initial writes)**.** Like some prior MCM formalisations [50, 69], but unlike most (e.g. [10, 11, 16, 43]), our executions exclude initial writes. We found that Alloy's performance degrades rapidly as the upper bound on $|\mathbb{E}|$ increases; by avoiding initial writes, this bound can be lowered. Removing initial writes makes rf^{-1} a *partial* function, which complicates the definition of *from-read*, as described below.

Definition 6. *From-read* relates each read to all of the writes that are co-later than the write that the read observed [6]:

$$fr \overset{\text{def}}{=} ((rf^{-1} ; co) \cup fr_{\text{init}}) - id$$

where $fr_{\text{init}} \overset{\text{def}}{=} ([R] - (rf^{-1} ; rf)) ; sloc ; [W]$. In the absence of initial writes, fr_{init} handles reads that observe the initial value.

2.2 Language-Specific Executions

We can obtain executions for various languages as subclasses of $\mathbb{X}$.

Definition 7 (C11 executions)**.** C11 executions ($\mathbb{X}_{\text{C11}}$) are structures that inherit all the fields and well-formedness conditions from basic executions, and add those listed in Fig. 3. The new fields originate from the 'memory orders' that are attached to atomic operations in C11 [38 (§7.17.3)]. Acquire events, release events, SC events, RMWs, and fences are all atomic ⓯. Atomic events that are neither acquires nor releases correspond to C11's 'relaxed' memory order. Acquire semantics is given to *all* SC reads and *only* reads and fences ⓰. Release semantics is given to *all* SC writes and *only* writes and fences ⓱. SC fences have both acquire and release semantics ⓲. Non-atomic reads access only non-atomic locations, and atomic operations never access non-atomic locations ⓳.

Definition 8 (OpenCL executions)**.** OpenCL [42] extends C11 with hierarchical models of execution and memory that reflect the GPU architectures it was primarily designed to target. Accordingly, OpenCL executions ($\mathbb{X}_{\text{OpenCL}}$, Fig. 4) extend C11 executions first by partitioning threads into one or more *workgroups* via the swg equivalence ⓴ ㉑, and second by allowing some atomic operations to be tagged as 'device scope' ㉒, which ensures they are visible throughout the device. Other atomics (i.e., with only 'workgroup scope') can efficiently synchronise threads within a workgroup but are unsuitable for inter-workgroup synchronisation. We do not consider OpenCL's local memory in this work, and we restrict our attention to the single-device case.

$dv \subseteq \mathbb{E}$ events that have whole-device scope
$swg \subseteq \mathbb{E}^2$ same workgroup ('co-operative thread array')

㉓ $sthd \subseteq swg$ ㉔ $equiv(swg, E)$ ㉕ $dv \subseteq F$ ㉖ $nal = \emptyset$

㉗ $sthd - id \subseteq sb \cup sb^{-1}$ ㉘ $R \not\pitchfork W$

Figure 5. PTX executions, $\mathbb{X}_{\text{PTX}}$ (extending $\mathbb{X}$)

Definition 9 (PTX executions)**.** Like OpenCL executions, PTX executions ($\mathbb{X}_{\text{PTX}}$, Fig. 5) gather threads into groups ㉓ ㉔. Some PTX fences (`membar.gl`) have whole-device scope ㉕, and others (`membar.cta`) have only workgroup scope. There are no non-atomic locations ㉖, sequencing is total within each thread ㉗, and we do not consider RMWs ㉘.

Remark 10. When drawing C11 executions, we identify the sets A, acq, rel and sc by tagging events in $E - A$ with `na`, those in $A - acq - rel$ with `RLX`, those in $acq - rel - sc$ with `ACQ`, those in $rel - acq - sc$ with `REL`, those in $acq \cap rel - sc$ with `AR`, and those in sc with `SC`. In OpenCL or PTX executions, we tag events in $A - dv$ with `WG`, and those in dv with `DV`.

2.3 Consistent, Race-Free, and Allowed Executions

Each MCM M defines sets $consistent_M$ and $racefree_M$ of executions. (For architecture-level MCMs, which do not define races, the latter contains all executions.) The sets work together to define the executions allowed under M, as follows.

Definition 11 (Allowed executions)**.** The executions of a program P that are allowed under an MCM M are

$$[\![P]\!]_M \overset{\text{def}}{=} \begin{cases} [\![P]\!]_0 \cap consistent_M \\ \qquad \text{if } [\![P]\!]_0 \cap consistent_M \subseteq racefree_M \\ \top \qquad \text{otherwise} \end{cases}$$

where $\top$ stands for an appropriate universal set. Here, $[\![P]\!]_0$ is the set of P's *candidate executions*. These can be thought of as the executions allowed under an MCM that imposes no constraints, and are discussed separately (see Def. 16). The equation above says that the allowed executions of P are its consistent candidates, unless a consistent candidate is racy, in which case *any* execution is allowed. (This is sometimes called 'catch-fire' semantics [19].)

The consistency and race-freedom axioms for C11 and OpenCL (minus 'consume' atomics) are defined in Figure 6 and explained below. We have included some recently-proposed simplifications. In particular, the simpler release sequence (proposed by Vafeiadis et al. [77]) makes deadness easier to define (§3.4), and omitting the total order 'S' over SC events (as proposed by Batty et al. [14]) avoids having to iterate over all total orders when showing an execution to be inconsistent.

Happens before (hb) edges are created by sequencing and by a release *synchronising with* (sw) an acquire in a different thread. Synchronisation begins with a release write (or a release fence that precedes an atomic write) and ends with an acquire read (or an acquire fence that follows an atomic read) and the read observes either that write or a member of the write's *release sequence* (rs). An event's release sequence comprises all the writes to the same location that are sequenced after the event, as well as the RMWs (which may be in another thread) that can be reached from one of those writes via a chain of rf edges [77 (§4.3)]. In OpenCL, *consistent*$_{\mathsf{C11}}$

$$㉙\ \mathit{acyclic}((\mathit{hb} \cap \mathit{sloc}) \cup \mathit{rf} \cup \mathit{co} \cup \mathit{fr})$$

$$㉚\ \mathit{rf}\ ;[\mathit{nal}] \subseteq \mathit{imm}([W]\ ;(\mathit{hb} \cap \mathit{sloc}))$$

$$㉛\ \mathit{acyclic}((\mathit{Fsb}^{?}\ ;(\mathit{co} \cup \mathit{fr} \cup \mathit{hb})\ ;\mathit{sbF}^{?}) \cap \mathit{sc}^2 \boxed{\cap\ \mathit{incl}})$$

racefree$_{\mathsf{C11}}$

$$㉜\ \mathit{cnf} - A^2 - \mathit{sthd} \subseteq \mathit{hb} \cup \mathit{hb}^{-1}$$

$$㉝\ \mathit{cnf} \cap \mathit{sthd} \subseteq \mathit{sb} \cup \mathit{sb}^{-1} \quad \boxed{㉞\ \mathit{cnf} - \mathit{incl} \subseteq \mathit{hb} \cup \mathit{hb}^{-1}}$$

$$\mathit{Fsb} \overset{\text{def}}{=} [F]\ ;\mathit{sb} \qquad \mathit{sbF} \overset{\text{def}}{=} \mathit{sb}\ ;[F] \qquad \mathit{rs} \overset{\text{def}}{=} (\mathit{sb} \cap \mathit{sloc})^*\ ;\mathit{rf}^*$$

$$\mathit{sw} \overset{\text{def}}{=} [\mathit{rel}]\ ;\mathit{Fsb}^{?}\ ;[W \cap A]\ ;\mathit{rs}\ ;\mathit{rf}\ ;[R \cap A]\ ;\mathit{sbF}^{?}\ ;[\mathit{acq}]$$

$$\boxed{\mathit{incl} \overset{\text{def}}{=} \mathit{dv}^2 \cup \mathit{swg}} \qquad \mathit{hb} \overset{\text{def}}{=} ((\mathit{sw} \boxed{\cap\ \mathit{incl}} - \mathit{sthd}) \cup \mathit{sb})^+$$

$$\mathit{cnf} \overset{\text{def}}{=} ((W \times W) \cup (R \times W) \cup (W \times R)) \cap \mathit{sloc} - \mathit{id}$$

Figure 6. The C11 and OpenCL MCMs

synchronisation only occurs between events with *inclusive scopes* (incl), which means that if the events are in different workgroups they must both be annotated with 'device' scope. Happens-before edges between events accessing the same location, together with rf, co, and fr edges, must not form cycles ㉙ [77 (§5.3)]. A read of a non-atomic location must observe a write that is still *visible* (vis) ㉚. The *sequential consistency* (SC) axiom ㉛ requires, essentially, that the co, hb and fr edges between sc events do not form cycles [14 (§3.2)] An execution has a *data race* unless every pair of conflicting (cnf) events in different threads, not both atomic, is in hb ㉜. It has an *unsequenced race* unless every pair of conflicting events in the same thread is in sb ㉝. An OpenCL execution has a *heterogeneous race* [34] unless every pair of conflicting events with non-inclusive scopes is in hb ㉞.

3. Obtaining Litmus Tests

If we find executions that solve $\tilde{g}$ (Def. 2), we need to 'lift' these executions up to the level of programs in order to obtain a solution to the original problem, g (Def. 1).

We begin this section by defining a language for these generated programs (§3.1). The language is designed to be sufficiently expressive that for any discovered execution X, there exists a program in the language that can behave like X. In particular, the program must be able to create arbitrary sequencing patterns and dependencies. Beyond this, we keep the language as small as possible to keep code generation simple. The language is also generic, so that it can be used to generate both assembly and high-level language tests.

We go on to define a function that obtains litmus tests from executions (§3.2), and show that, under an additional assumption about executions, the function is total (§3.3). We then define our deadness constraint on executions (§3.4), and conclude with a discussion of some of the practical aspects of generating litmus tests (§3.5).

3.1 Programming Language

The syntax of our generic programming language is defined in Fig. 7. We postulate a set $\mathbb{V}$ of values (containing zero), a set $\mathbb{R}$ of registers, and a set $\mathbb{L}$ of (shared) locations, which is subdivided into atomic ($\mathbb{L}_{\mathrm{a}}$) and non-atomic ($\mathbb{L}_{\mathrm{na}}$) locations. Every register and location is implicitly initialised to zero. Let $\mathbb{A}_s$ denote a language of *expressions* over a set s. Observe that since the only construction is the addition of a register multiplied by zero, these expressions only evaluate to elements of s. We use them to create artificial dependencies (i.e. *syntactic* but not *semantic* dependencies). The set $\mathbb{I}$ of *instructions* includes stores, loads, compare-and-swaps (CAS's), and fences. The $\mathbf{cas}(x, v, v')$ instruction compares x to v (and if the comparison succeeds, sets x to v'), returning the observed value of x in either case. Observe that, artificial dependencies notwithstanding, stores and CAS's write only constant values to locations. Let $\mathbb{C}$ denote a set of *components*, each formed from sequenced (;) or unsequenced (+) composition, or one-armed conditionals that test for a register having a constant value. We have no need for loops because the executions we generate are finite. Each component has a globally-unique label, ℓ. Let $\mathbb{P}$ be the set of *programs*, each a parallel composition of components. The top-level components in a program are called *threads*.

$$\begin{aligned}
\mathbb{A}_s &::= s \mid \mathbb{A}_s + 0 \times \mathbb{R} \\
\mathbb{I} &::= \mathbf{st}(\mathbb{A}_{\mathbb{L}}, \mathbb{A}_{\mathbb{V}}) \mid \mathbf{ld}(\mathbb{R}, \mathbb{A}_{\mathbb{L}}) \mid \mathbf{cas}(\mathbb{A}_{\mathbb{L}}, \mathbb{V}, \mathbb{A}_{\mathbb{V}}) \mid \mathbf{fe} \\
\mathbb{C} &::= \mathbb{I}^{\ell} \mid \mathbb{C} +^{\ell} \mathbb{C} \mid \mathbb{C} \,;^{\ell} \mathbb{C} \mid \mathbf{if}^{\ell}\ \mathbb{R} = \mathbb{V}\ \mathbf{then}\ \mathbb{C} \\
\mathbb{P} &::= \mathbb{C} \parallel \ldots \parallel \mathbb{C}
\end{aligned}$$

Figure 7. A programming language

Some languages attach extra information to instructions, such as their atomicity and memory order (in C11), their memory scope (in OpenCL), or whether they are 'locked' (in x86). Accordingly, when using the generic language in Fig. 7 to represent one of these languages, we introduce an additional function that stores extra information for each instruction label.

We ensure that the programs we generate are *well-formed* – this is necessary for ensuring that they provide valid solutions to g.

Definition 12 (Well-formed programs)**.** A program is well-formed if: (1) different stores/CAS's to the same location store different values, (2) each register is written at most once, (3) stores/CAS's never store zero, (4) if-statements never test for zero, and (5) whenever an if-statement tests register r, there is a load/CAS into r sequenced earlier in the same thread.

3.2 From Executions to Litmus Tests

Our strategy for solving g, as outlined in §1.2, is summarised by the following 'proof rule':

$$\begin{array}{ll}
\text{Step 1:} & (X, Y) \in \tilde{g}(M, N, \triangleright) \\
\text{Step 2:} & (P, \sigma) \in \textbf{\textit{lit}}^{\min}(X) \quad (Q, \sigma') \in \textbf{\textit{lit}}(Y) \quad \sigma = \sigma' \quad P \blacktriangleright Q \\
\hline
\text{Result:} & (P, Q, \sigma) \in g(M, N, \blacktriangleright)
\end{array} \qquad (1)$$

The purpose of this subsection is to define the ***lit*** and $\textbf{\textit{lit}}^{\min}$ functions. We begin by defining a more general constraint, ***lit***$'$.

Definition 13. The predicate $\textbf{\textit{lit}}'(X, P, \sigma, \mathit{disabled}, \mathit{failures})$ serves to connect executions X with litmus tests (P, σ). The $\mathit{disabled}$ argument is a subset of P's components, which we interpret as those that do not actually execute when creating the execution X (because they are guarded by an if-statement whose test failed). The $\mathit{failures}$ argument is a subset of P's CAS instructions, which we interpret as those that fail to carry out the 'swap' when creating execution X. (Characterising executions by which instructions fail and which are disabled is only sensible because of our restriction to loop-free programs.) The predicate $\textbf{\textit{lit}}'(X, P, \sigma, \mathit{disabled}, \mathit{failures})$ holds whenever there exists a bijection μ between P's enabled instructions and X's events such that the following conditions all hold (in which we abbreviate $\mu(i)$ as μ_i):

Conditionals For an if-statement with condition '$r = v$', the body is enabled iff both the if-statement is enabled and $\sigma(r) = v$.

Disabled loads If a load/CAS into register r is disabled, then $\sigma(r) = 0$.

Unguarded components Any component not guarded by an if-statement is enabled.

CAS failures For every enabled CAS i in P: i is in *failures* iff there is no j with $(\mu_j, \mu_i) \in rf$ that writes the value i expects.

Instruction types For every enabled instruction i in P: i is a store iff $\mu_i \in W - R$; i is a load or a failed CAS iff $\mu_i \in R - W$; i is a successful CAS iff $\mu_i \in R \cap W$; and i is a fence iff $\mu_i \in F$.

Non-atomic locations If i is an enabled non-fence instruction in P then $\mu_i \in nal$ iff i's location is in $\mathbb{L}_{\mathrm{na}}$.

Threading, locations, and dependencies For every enabled instructions i and j in P: $(\mu_i, \mu_j) \in sthd$ iff i and j are in the same thread in P; $(\mu_i, \mu_j) \in sloc$ iff i and j both access the same location; $(\mu_i, \mu_j) \in cd$ iff there is an enabled if-statement, say T, such that i is sequenced before T, i writes to the register T tests, and j is in T's body; $(\mu_i, \mu_j) \in ad$ iff the expression for j's location depends (syntactically) on the register written by i; and $(\mu_i, \mu_j) \in dd$ iff j writes an expression that depends (syntactically) on the register i writes.

Sequenced composition For every enabled ;-operator in P: if i and j are enabled instructions in the left and right operands (respectively), then $(\mu_i, \mu_j) \in sb$.

Unsequenced composition For every enabled $+$-operator in P: if i and j are enabled instructions in the left and right operands (respectively), then $\{(\mu_i, \mu_j), (\mu_j, \mu_i)\} \not\cap sb$.

Final registers For every enabled load/CAS j in P on location x into register r: either (1) there exists an enabled store/CAS i that writes v to x, $(\mu_i, \mu_j) \in rf$ and $\sigma(r) = v$, or (2) $\sigma(r) = 0$ and μ_j is not in the range of rf.

Final locations For every location x: either (1) there is an enabled store/CAS i that writes v to x, μ_i has no successor in co, and $\sigma(x) = v$, or (2) x is never written and $\sigma(x) = 0$.

Definition 14 (Obtaining litmus tests)**.** Let ***lit***(X) be the set of all litmus tests (P, σ) for which ***lit***$'(X, P, \sigma, \mathit{disabled}, \mathit{failures})$ holds for some instantiation of *disabled* and *failures*.

Definition 15 (Obtaining minimal litmus tests)**.** Let ***lit***$^{\min}(X)$ be the set of all litmus tests (P, σ) for which ***lit***$'(X, P, \sigma, \emptyset, \emptyset)$ holds. That is, when P behaves like X, all of its instructions are executed and all of its CAS's succeed. We say that the litmus test (P, σ) is *minimal* for X, or, dually, X is a *maximal* execution of P.

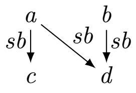
Definition 16 (Candidate executions)**.** We define P's candidate executions by inverting ***lit***: $[\![P]\!]_0 \overset{\text{def}}{=} \{X \mid \exists \sigma.\, (P, \sigma) \in \textbf{\textit{lit}}(X)\}$.

Definition 17 (Observable final states)**.** We can now formally define the notion of observation we employed in Def. 1: $\mathbf{obs}_M(P)$ is the set of final states σ for which $(P, \sigma) \in \textbf{\textit{lit}}(X)$ for some $X \in [\![P]\!]_M$.

3.3 Totality of *lit*$^{\min}$

We now explain why the ***lit***$^{\min}$ function is currently not total (that is: there exist well-formed executions X for which ***lit***$^{\min}(X)$ is empty), and how we can impose an additional restriction on executions to make it become total.

The problem is: our programming language cannot express all of the sequencing patterns that an execution can capture in a valid sb relation. For instance, it is not possible to write a program that generates exactly the following sb edges:



(2)

$$\boxed{\begin{array}{l} \mathit{dead} \\ ㉟\ \mathrm{domain}(cd) \subseteq \mathrm{range}(rf) \\ ㊱\ imm(co)\,;imm(co)\,;imm(co^{-1}) \subseteq rf^{?}\,;(sb\,;(rf^{-1})^{?})^{?} \end{array}}$$

$$\boxed{\begin{array}{l} \mathit{dead}_{\mathrm{C11}} \\ ㊲\ \mathit{dead} \qquad ㊳\ cnf \cap sthd \subseteq sb \cup sb^{-1} \\ ㊴\ pdr \subseteq dhb \cup dhb^{-1} \cup (narf\ ;\ ssc) \cup (ssc\ ;\ narf^{-1}) \end{array}}$$

$$pdr \overset{\text{def}}{=} cnf - A^2 \quad cde \overset{\text{def}}{=} (rfe \cup cd)^*\,;cd \quad drs \overset{\text{def}}{=} rs - ([R]\,;\overline{cde})$$

$$dsw \overset{\text{def}}{=} sw \cap (((Fsb^{?}\ ;[rel]\ ;drs^{?}) - (cd^{-1}\ ;\overline{cde}))\ ;rf)$$

$$dhb \overset{\text{def}}{=} sb^{?}\,;(dsw\,;cd)^* \quad ssc \overset{\text{def}}{=} id \cap cde \quad narf \overset{\text{def}}{=} rf \cap nal^2 - hb$$

Figure 8. The *dead* constraint and its specialisation for C11

This is because, armed only with sequenced (';') and unsequenced ('$+$') composition (see Fig. 7), it is only possible to produce sb relations that are in the set of *series–parallel partial orders* [53].[2] Helpfully, series–parallel partial orders are characterised exactly by a simple check: that they do not contain the 'N'-shaped subgraph shown in (2) [78]. Accordingly, we impose one further well-formedness constraint on executions:

$$\begin{array}{l} \nexists a, b, c, d \in E. \\ \quad \{(a,c),(a,d),(b,c)\} \in sb \wedge \{(a,b),(b,c),(c,d)\} \not\cap sb^{?}. \end{array}$$

Our ***lit***$^{\min}$ function now becomes total (and hence ***lit*** too). Indeed, it is straightforward to determinise the constraints listed in Def. 13 into an algorithm for *constructing* litmus tests from executions (and we have implemented this algorithm, in Java).

3.4 Dead Executions

When searching for inconsistent executions, we restrict our search to those that are also *dead*.

Definition 18 (Semantic deadness)**.** The set of executions that are (semantically) dead under MCM M is given by $\mathit{semdead}_M \overset{\text{def}}{=}$

$$\begin{array}{l} \{X \in \mathbb{X} \mid X \notin \mathit{consistent}_M \Rightarrow \forall P, \sigma, X', \sigma'. \\ \quad ((P,\sigma) \in \textbf{\textit{lit}}^{\min}(X) \wedge (P,\sigma') \in \textbf{\textit{lit}}(X') \wedge X' \in \mathit{consistent}_M) \\ \qquad \Rightarrow (X' \in \mathit{racefree}_M \wedge \sigma' \neq \sigma)\}. \end{array}$$

That is, for any minimal litmus test (P, σ) for X, no consistent candidate execution of P is racy or reaches σ. In other words: (P, σ) must fail under M.

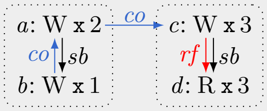
We employ syntactic approximations of semantic deadness. Figure 8 shows how this approximation is defined for any architecture-level MCM that enforces coherence (*dead*), and how it is strengthened to handle the C11 MCM ($\mathit{dead}_{\mathrm{C11}}$).

At the architecture level, we need not worry about races, so ensuring deadness is straightforward. In what follows, let X be an inconsistent execution, (P, σ) be a minimal litmus test of X, and X' be another candidate execution of P. First, we require every event that is the source of a control dependency to read from a non-initial write ㉟. This ensures that P need not contain '**if** `r = 0`'. Such programs are problematic because we cannot tell whether the condition holds because `r` was set to zero or because `r` was never assigned. Second, we require that every co edge (except the last) is justified by an sb edge ㊱. This condition ensures that X' cannot be made consistent simply by inverting one or more of X's co edges.

[2] That said, if we extended our language to support fork/join parallelism, then the execution in (2) would become possible: `t1=fork(`b`);` a`;` `t2=fork(`c`); join(t1);` d`; join(t2).`

The construction $imm(co)$ obtains event pairs (e_1, e_2) that are consecutive in co; composing this with '$imm(co)\ ;\ imm(co^{-1})$' restricts our attention to those pairs for which it is possible to take a further co step from e_2. The last co edge does not need justifying with an sb edge because it is directly observable in σ.

Example 2. The basic execution below (left) is inconsistent in any MCM that imposes coherence (because co is contradicting sb), but the litmus test obtained from it (below right) does not necessarily fail because its final state can be obtained via a consistent execution that simply reverses the co edge from (b, a) to (a, b).

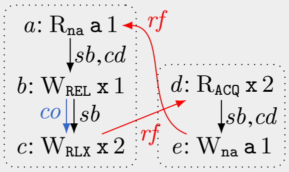


```
int x=0;
  x=2; x=1;
  ═════════
  x=3; r0=x;
x==3 && r0==3
```

At the software level, we must also worry about races. First, we forbid all unsequenced races ㊳, because if X does not have an unsequenced race, then neither will X', because unsequenced races are not affected by runtime synchronisation behaviour. Second, condition ㊴ is concerned with *potential data races* (pdr): events that conflict and are not both atomic. Although X is inconsistent (which renders any races in X irrelevant) we worry that X' might be consistent and racy. So, we require pdr-linked events to be also in the *dependable happens-before* (dhb) relation, or to exhibit a *self-satisfying cycle* (ssc), both of which we explain below.

Dependable happens-before This is a restriction of ordinary happens-before (hb). Essentially: if e_1 and e_2 are in dhb in X, and they map to instructions i_1 and i_2 respectively in P, then if i_1 and i_2 are represented by events in X', those events are guaranteed to be related by happens-before.

Example 3. The execution below (left) has fixed the shortcoming in Example 1 by adding control dependencies, but it has introduced the C11 *release sequence* as a further complexity.



```
int a=0; atomic_int x=0;
  r0=a;
  if(r0==1) x.store(1,REL);
  x.store(2,RLX);
  ═════════════════════════
  r1=x.load(ACQ);
  if(r1==2) a=1;
r0==1 && r1==1
```

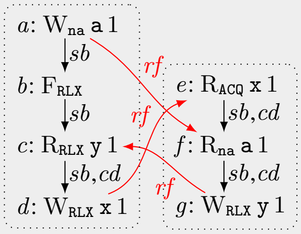
Although event d synchronises with b, it actually obtains its value from c, in b's release sequence. The execution is not semantically dead because its litmus test (above right) is racy: if `r0` is not assigned 1, then the release store is not executed; this means that `r1` can read 2 without synchronisation having occurred; this leads to a race on `a`. By subtracting $(cd^{-1}\,;\,\overline{cde})$ in the definition of dsw, we ensure that whatever controls b also controls c, and this rules out undesirable executions like the one above.

Moreover, the $([R]\,;\,\overline{cde})$ in the definition of drs ensures that if b is an RMW, it controls the execution of c. The effect is that c is not executed if the CAS corresponding to the RMW fails.

Self-satisfying cycles An event occurs in a self-satisfying cycle (ssc) if it is connected to itself via a chain of cd and rf edges that ends with a control dependency. The if-statements that create these dependencies are constructed such that their bodies are only executed if the desired rf edges are present (cf. Def. 13).

A potential race between e_1 and e_2 is deemed dead if both access a non-atomic location, e_1 is observed by e_2 but does not happen before it, and e_2 is in a self-satisfying cycle. The reasoning is as follows. First, note that the execution is inconsistent, because reads of non-atomic locations cannot observe writes that do not happen before them (c.f. ㉚). Second, the self-satisfying cycle ensures that e_2 reads from e_1 in *every* candidate execution that includes those events. Therefore, every candidate execution is inconsistent and any data races can be safely ignored. We illustrate this reasoning in the following example.

Example 4. The execution below (left) is inconsistent because f reads a non-atomic location from a write (a) that does not happen before f. It is dead because f is in a self-satisfying cycle. Its litmus test (below right) is not racy because the conditionals ensure that the right-hand thread's load of `a` is only executed if it obtains the value 1, which means that it reads from the left-hand thread's store to `a`, which means that the execution is inconsistent and hence that any races can be ignored.



```
int a=0; atomic_int x,y=0;
  a=1; fence(RLX);
  r0=y.load(RLX);
  if(r0==1) x.store(1,RLX);
  ═════════════════════════
  r1=x.load(ACQ);
  if(r1==1) r2=a;
  if(r2==1) y.store(1,RLX);
r0==1 && r1==1 && r2==1
```

Checking deadness The constraints that define $dead_M$ are quite subtle, particularly for complex MCMs like C11. Fortunately, we can use Alloy to check that these constraints imply semantic deadness, by seeking elements of $dead_M - semdead_M$. That is: we search for an execution X that is deemed dead, but which gives rise to a litmus test (P, σ) that has, among its candidates, a consistent execution X' that is either racy or reaches σ. We were able to check that $dead_M \subseteq semdead_M$ holds for all executions with no more than 8 events; beyond this, Alloy timed out (after four hours).

Remark 19. We are using Alloy here to search all candidate executions of a program, yet in §1.2 we argued that it is impractical to phrase constraints over programs because to do so would necessitate an expensive search over all candidate executions of a program. We emphasise that this is not a contradiction. The formula to which we objected had a problematic $\exists\forall$ pattern: 'show that *there exists* a program such that *for all* of its candidate executions, a certain property holds', whereas satisfying $dead_M - semdead_M$ requires only existential quantification.

3.5 Practical Considerations

We have found that Step 2 of our proof rule (1) can still be hard for Alloy to solve, often leading to timeouts, particularly when X and Y are large. The search space is constrained quite tightly by the values of X and Y, but there are still many variables involved. One of the degrees of flexibility in choosing P and Q is in the handling of if-statements. For instance, both ㊵ and ㊶ below give rise to the same executions because of our well-formedness restrictions on programs (Def. 12):

㊵ **if** b **then** $(C_1\ ;\ C_2)$ ㊶ (**if** b **then** C_1) ; (**if** b **then** C_2).

In response to these difficulties, we designed (and implemented in Java) a *deterministic* algorithm for $\boldsymbol{lit}^{\min}$ (as mentioned in §3.3), called LIT. In particular, it always chooses option ㊶ over option ㊵. In practice, we find it quicker to obtain (P, Q, σ) by constructing $(P, \sigma) = \textrm{LIT}(X)$ and $(Q, \sigma') = \textrm{LIT}(Y)$, rather than by solving the four constraints in Step 2 of (1). This approach satisfies the third constraint of Step 2 ($\sigma = \sigma'$) because all of the $\triangleright$ relations that we consider in this work ensure that X and Y have the same co and rf edges and hence reach the same final state. It does not, however, guarantee $P \blacktriangleright Q$. In particular, if a compiler mapping sends '`A`' to ‘`B ; C`’, then our algorithm would suggest, unrealistically, that ‘**if** b **then** `A`’ can compile to ‘(**if** b **then** `B`) ; (**if** b **then** `C`)’. In our experience, the generated P’s and Q’s are sufficiently close to being related by ▶ that the discrepancy does not matter.

In fact, we do not even prove that (1) is guaranteed to provide a valid solution to g, nor that a solution even necessarily exists. Such a proof would be highly fragile, being dependent on intricacies of the deadness constraint, which in turn depends on intricacies of various MCMs, many of which may be revised in the future. Instead, we follow the ‘lightweight, automatic’ approach extolled elsewhere in this paper. Besides using Alloy to check the definition of *dead* (as described in §3.4), we also implemented (in Java) a basic MCM simulator to enumerate the candidate executions of each litmus test we generate, to ensure that must-fail litmus tests really must fail. We would have preferred to have used an existing simulator like herd [12] or CppMem [16], but we found both tools to be unsuitable because of restrictions they impose on litmus tests: herd cannot handle sb being partial within a thread, and CppMem cannot handle if-statements that test for particular values.

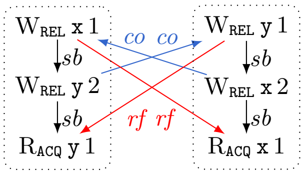
4. Application: Comparing MCMs (Q2)

In this section, we use Alloy to generate litmus tests that distinguish three proposed variants of the C11 MCM (§4.1, §4.2, §4.3). Such distinguishing tests, particularly the *simplest* distinguishing tests, are difficult to find by hand, but are very useful for illustrating the proposed changes. We go on to check two generic properties of MCMs – multi-copy atomicity (§4.4) and SC-DRF (§4.5) – by encoding the properties as MCMs themselves and comparing them against software- and architecture-level MCMs.

4.1 Strong Release/Acquire Semantics in C11

Lahav et al. have proposed a stronger semantics for release/acquire atomics [43]. For the release/acquire fragment of C11 (no non-atomics and no relaxed or SC atomics), their semantics amounts to adding the axiom $\mathit{acyclic}(\mathit{sb} \cup \mathit{co} \cup \mathit{rf})$. We used Alloy to compare their MCM, which we call C11-SRA, to C11 over the release/acquire fragment.

Lahav et al. provide a 10-event (and 4-location) execution that distinguishes the MCMs [43 (Fig. 5)]. Alloy, on the other hand, found a execution that requires just 6 events (and 2 locations) and serves the same purpose.[3]

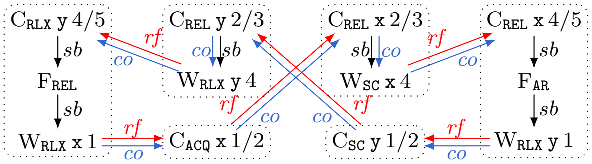


```
atomic_int x=0,y=0;
  x.store(1,REL);
  y.store(2,REL);
  r0=y.load(ACQ);

  y.store(1,REL);
  x.store(2,REL);
  r1=x.load(ACQ);
r0==1 && r1==1
```

4.2 Forbidding Reading/Synchronisation Cycles in C11

Nienhuis et al. [58 (Fig. 12)] have suggested strengthening the C11 MCM with the axiom $\mathit{acyclic}(\mathit{sw} \cup \mathit{rf})$. Let us call that MCM C11-SwRf. We used Alloy to search for litmus tests that could witness such a change, and discovered a solution requiring 12 events and 6 threads that is virtually identical to the one provided by Nienhuis et al., if a little less symmetrical. We sought smaller solutions, and found none with fewer than 8 events that could distinguish C11-SwRf from C11. For executions with 8 to 11 events, the SAT solver could not return a result in a reasonable time.



consistent$_{\text{MCA}}(\mathit{ppo})$

㊷ $\mathit{acyclic}((\mathit{sb} \cap \mathit{sloc}) \cup \mathit{co})$ ㊸ $\mathit{acyclic}(\mathit{wo}(\mathit{ppo}))$

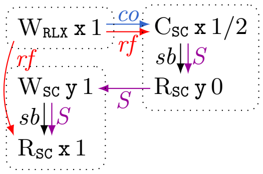
$$\mathit{wo}(\mathit{ppo}) \stackrel{\text{def}}{=} (((\mathit{rfe} ; \mathit{ppo} ; \mathit{rfe}^{-1}) - \mathit{id}) ; \mathit{co}) \cup (\mathit{rfe} ; \mathit{ppo} ; \mathit{fr}_{\text{init}})$$

Figure 9. Multi-copy atomicity as an MCM

4.3 Simplifying the SC Axioms in C11

Batty et al. have proposed a change to the C11 consistency axioms that enables them to be simplified, and also avoids the need for a total order, S, over all SC events [14]. Having already incorporated their proposal in our Fig. 6, let us call the original MCM C11-Orig.

Batty et al. present a litmus test to distinguish C11 from C11-Orig, which uses 7 instructions across 4 threads [14 (Example 1)]. Alloy, on the other hand, found one (below) that needs only 5 instructions and 3 threads.



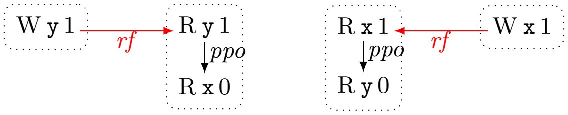
```
atomic_int x=0,y=0;
  x.store(1,RLX);

  r0=x.cas(1,2,SC,RLX);
  r1=y.load(SC);

  y.store(1,SC);
  r2=x.load(SC);
r0==true && r1==0 && r2==1
```

4.4 Multi-copy atomicity

The property of *multi-copy atomicity*[4] (MCA) [23] ensures that, in the absence of thread-local reordering, different threads cannot observe writes in conflicting orders – i.e., there is a single copy of the memory that serialises all writes. The canonical MCA violation is given by the IRIW test [19] (below) where thread-local reordering has been disabled by inducing ‘preserved program order’ (ppo) edges, perhaps using dependencies or fences. That the final reads observe 0 betrays a disagreement in the order the writes occurred:



(3)

We formalise MCA – for the first time in the axiomatic style – in Fig. 9. The model comprises write/write coherence ㊷ [72] and the acyclicity of the *write order* relation, wo ㊸. Write order captures the intuition that if two reads, say r_1 and r_2 (with $(r_1, r_2) \in \mathit{ppo}$), observe two writes, say w_1 and w_2 respectively, then any write co-later than w_2 must follow w_1 in the single copy of memory. The wo relation is the union of two cases: in the first, both reads observe write events, and in the second, one reads from the initial value

[3] Lahav et al. impose an additional technical requirement that postconditions should not need to refer to shared locations (only to registers), which rules out the even-simpler ‘2+2W’ litmus test [67].

[4] This is also known as *store atomicity* [72] or *write atomicity* [4].

$consistent_{\mathsf{SC}}$

$$(44)\ acyclic(rf \cup co \cup fr \cup sb)$$

Figure 10. The SC MCM (following Shasha et al. [70])

$strengthen(X, Y)$

$$(45)\ E_X = E_Y \quad (46)\ R_X = R_Y \quad (47)\ W_X = W_Y$$
$$(48)\ F_X = F_Y \quad (49)\ nal_X = nal_Y \quad (50)\ sb_X \subseteq sb_Y$$
$$(51)\ ad_X \subseteq ad_Y \quad (52)\ cd_X \subseteq cd_Y \quad (53)\ dd_X \subseteq dd_Y$$
$$(54)\ sthd_X \subseteq sthd_Y \quad (55)\ sloc_X = sloc_Y \quad (56)\ rf_X = rf_Y$$
$$(57)\ co_X = co_Y$$

Figure 11. 'Strengthening' an execution

(reusing our definition of fr_{init} from Def. 6). Note that MCA is parameterised by the given model's *ppo*.[5]

With this formal definition of MCA, we can seek executions allowed by a given MCM but disallowed by MCA. We tested x86 and Power, and as expected, Power does not guarantee MCA (Alloy finds a counterexample similar to (3)) but x86 does.

4.5 Seeking SC-DRF Violations

We used Alloy to seek violations of the SC-DRF guarantee [5] in an early draft of the C11 MCM [37]. The SC-DRF guarantee, in the C11 context, states that if a program is race-free and contains no non-SC atomic operations, then it has only SC semantics.

We sought an execution X that is dead (so that its corresponding litmus test is race-free), uses no non-SC atomics, and is consistent under the (draft) C11 MCM but inconsistent under SC:

$$A_X \subseteq sc_X \wedge X \in dead_{\mathsf{C11}} \cap consistent_{\mathsf{C11\text{-}Draft}} - consistent_{\mathsf{SC}}$$

where the SC MCM is characterised in Fig. 10 and C11-Draft has all the axioms listed by Batty et al. [14 (Def. 11)], minus their 'S4' axiom. Alloy found an example similar to that reported by Batty et al. [16 (§4, *Sequential consistency for SC atomics*)]. The non-SC execution is consistent under C11-Draft because no rule forbids SC reads to observe initial writes. The 'S4' axiom was added [16 (§2.7)] to fix exactly this issue.

5. Application: Checking Monotonicity (Q3)

In this section, we rediscover two monotonicity violations in the C11 MCM, whereby a new behaviour is enabled either by sequentialisation (§5.1) or by strengthening a memory order (§5.2).

Checking monotonicity requires the relation defined in Fig. 11, which holds when one execution is 'stronger' than another. There is almost an isomorphism between X and Y, except that Y may have extra *sb* and dependency edges (50) (51) (52) (53), and it may interleave multiple threads together (54).

5.1 Monotonicity of C11 w.r.t. Sequencing

One way to extend the *strengthen* relation to the C11 setting is to add the following constraints:

$$A_X = A_Y \qquad acq_X = acq_Y \qquad rel_X = rel_Y \qquad sc_X = sc_Y.$$

With this notion of strengthening, Alloy found a pair of 6-event executions that witness a monotonicity violation in C11. They are almost identical to those given by Vafeiadis et al. [77 (Fig. 1)], though slightly less elegant: Alloy chose one of the writes to be SC when a relaxed write would have sufficed.

$compile(\pi, X, Y)$

$$(58)\ [E_X] = \pi \mathbin{;} \pi^{-1} \qquad (59)\ [E_Y] \subseteq \pi^{-1} \mathbin{;} \pi$$
$$(60)\ \pi^{-1} \mathbin{;} sb_X^{?} \mathbin{;} \pi = sb_Y^{?} \qquad (61)\ ad_X = \pi \mathbin{;} ad_Y \mathbin{;} \pi^{-1}$$
$$(62)\ dd_X = \pi \mathbin{;} dd_Y \mathbin{;} \pi^{-1} \qquad (63)\ cd_X \mathbin{;} \pi = \pi \mathbin{;} cd_Y$$
$$(64)\ sloc_X = \pi \mathbin{;} sloc_Y \mathbin{;} \pi^{-1} \qquad (65)\ \pi^{-1} \mathbin{;} sthd_X \mathbin{;} \pi = sthd_Y$$
$$(66)\ rf_X = \pi \mathbin{;} rf_Y \mathbin{;} \pi^{-1} \qquad (67)\ co_X = \pi \mathbin{;} co_Y \mathbin{;} \pi^{-1}$$

Figure 12. Compiling an execution

5.2 Monotonicity of C11 w.r.t. Memory Orders

Another way to extend the *strengthen* relation is to add

$$A_X = A_Y \qquad acq_X \subseteq acq_Y \qquad rel_X \subseteq rel_Y \qquad sc_X \subseteq sc_Y$$
$$sb_X = sb_Y \qquad ad_X = ad_Y \qquad cd_X = cd_Y \qquad dd_X = dd_Y$$
$$sthd_X = sthd_Y$$

which allows memory orders to be strengthened but forbids changes to sequencing or threading. Using this notion of strengthening, Alloy found a 7-event monotonicity violation in C11: the execution previously given in Example 4. That execution is inconsistent, but if the relaxed fence is strengthened to a release fence, it becomes consistent. This example is similar to one due to Vafeiadis et al. [77 (§3, *Strengthening is Unsound*)], but it is simpler: where theirs requires 8 events, 4 locations, and 3 threads, our Alloy-generated example requires only 7 events, 3 locations, and 2 threads.

6. Application: Checking Compiler Mappings (Q4)

We now report on our use of Alloy to investigate OpenCL compiler mappings for AMD-style GPUs (§6.1) and NVIDIA GPUs (§6.2), and a C11 compiler mapping for Power (§6.3).

Checking compiler mappings requires a relation that holds when one execution 'compiles' to another. Of course, real compilers do not compile on a 'per-execution' level, but from a pair of executions related in this way, we can obtain a pair of programs that are related by the compiler mapping at the level of program code. Compiler mappings are more complicated than the strengthenings considered in §5, because they may introduce additional events, such as fences. For this reason, we introduce an additional relation, π, to connect each original event to its compiled event(s). (In §5, π is the identity relation.)

We are able to handle 'straight line' mappings like C11/x86 [16], OpenCL/AMD (§6.1) and OpenCL/PTX (§6.2). We can also handle more complicated mappings like C11/Power [15, 66], which includes RMWs and introduces control dependencies, but our examples are limited to the execution level, because at the code level, C11 RMWs map to Power loops, which our loop-free output programming language cannot express.

Execution X maps to execution Y if there exists a π for which $compile(\pi, X, Y)$ holds, as defined in Fig. 12. Here, π is an injective, surjective, multivalued function from X's events to Y's events (58) (59), that preserves sequencing (60), dependencies (61) (62) (63), locations (64), threads (65), the reads-from relation (66), and the coherence order (67). The location-preservation and thread-preservation constraints have different shapes because, for instance, when a mapping sends source event e to a set $\{e'_i\}_i$ of target events, it must ensure that *one* e'_i has e's location (the other events will be fences), but that *every* e'_i is in the same thread.

[5] We have an alternative formulation for happens-before-based models.

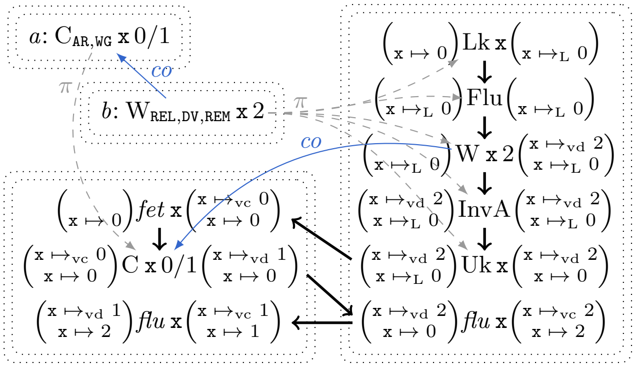


Figure 13. RMW atomicity bug in the OpenCL/AMD mapping

6.1 Compiling OpenCL to AMD-Style GPUs

Orr et al. [62] describe a compiler mapping from OpenCL to an AMD-style GPU architecture. Actually, they support OpenCL extended with 'remote-scope promotion', in which a workgroup-scoped event a can synchronise with an event b in a different workgroup if b is annotated as 'remote'. Wickerson et al. [82] report on two bugs in this scheme: a failure to implement message-passing, and a failure of RMW atomicity.

We have formalised Orr et al.'s architecture-level MCM and compiler mapping in Alloy [1], following the formalisation by Wickerson et al., and then used Alloy to search for bugs – essentially automating the task that Wickerson et al. conducted manually. The architecture-level MCM is *operational*, which means that consistent executions are obtained constructively, not merely characterised by axioms. This means that the MCM is more complex to express in Alloy. Essentially, the MCM involves a single global memory in which entries can be temporarily locked, several processing elements partitioned into compute units, and a write-back write-allocate cache per compute unit.

Fig. 13 depicts the RMW atomicity bug discovered by Alloy. The top left of the figure shows a 2-event execution that is inconsistent (and dead) in OpenCL, and the right shows a corresponding 9-event execution that is observable on hardware. (Wickerson et al.'s bug is similar, but requires an extra 'fetch' event.) All events are grouped by thread (inner dotted rectangles) and by workgroup (outer dotted rectangles). Events in the architecture-level execution are totally ordered (thick black arrows). We track the local and global state before and after each event, writing $\left(\begin{smallmatrix}\sigma_l\\ \sigma_g\end{smallmatrix}\right)$ for local state σ_l and global state σ_g. The global state comprises global memory entries, some of which are locked (L). The local state comprises the compute unit's cache entries, each either valid (v) or invalid (i), and either clean (c) or dirty (d). The π edges show how the software-level events are mapped to architecture-level events. The RMW at workgroup scope (a) maps to a single RMW, while the remote (`REM`) write to `x` (b) is implemented by first flushing all dirty local cache entries (Flu), then doing the write (W), then invalidating all entries in all caches (InvA), all while preventing concurrent access to `x` in the global memory (Lk, Uk). Architecture-level executions also include the actions of the 'environment' fetching (*fet*) and flushing (*flu*) entries to and from global memory, as well as derived *rf* and *co* relations. The undesirable but observable execution, in which $\mathtt{x} = 1$ in the final state, arises because the mapping fails to propagate $\mathtt{x} = 2$ to the global memory before releasing the lock on `x`. The fix is to flush immediately after the write.

To make Alloy's search for OpenCL/AMD compiling bugs tractable, we found it necessary to make several simplifications: we deleted the QuickRelease buffers [29]; we allowed multiple locations to be fetched and flushed in a single action (which reduces the total number of actions required); we hard-coded the system to have exactly two workgroups with one thread each; and we maximised sharing between global and local memory entry objects. These changes are not necessarily semantics-preserving, but any bogus solutions found using the simplified MCM can be simply tested manually against the full one.

consistent$_{\mathsf{PTX1}}$

(68) $acyclic((sb \cap sloc - R^2) \cup rf \cup co \cup fr)$

(69) $acyclic(hb_{\mathrm{wg}})$ (70) $acyclic(hb_{\mathrm{dv}})$

$$rmo \overset{\text{def}}{=} rfe \cup co \cup fr \quad f_{\mathrm{dv}} \overset{\text{def}}{=} sb\,;[F \cap dv]\,;sb \quad f_{\mathrm{any}} \overset{\text{def}}{=} sb\,;[F]\,;sb$$

$$hb_{\mathrm{wg}} \overset{\text{def}}{=} (rmo \cup f_{\mathrm{any}}) \cap swg \qquad hb_{\mathrm{dv}} \overset{\text{def}}{=} rmo \cup f_{\mathrm{dv}}$$

Figure 14. The PTX1 MCM

(71)	`store(na|RLX,` s`)`	⇝	`st.cg`	
(72)	`store(REL,` s`)`	⇝	$\mathtt{F}_s$; `st.cg`	
(73)	`store(SC,` s`)`	⇝	$\mathtt{F}_s$; `st.cg` ; $\mathtt{F}_s$	where
(74)	`load(na,` s`)`	⇝	`ld.cg`	$\mathtt{F_{WG}} \overset{\text{def}}{=}$ `membar.cta`
(75)	`load(RLX|ACQ,` s`)`	⇝	`ld.cg` ; $\mathtt{F}_s$	$\mathtt{F_{DV}} \overset{\text{def}}{=}$ `membar.gl`
(76)	`load(SC,` s`)`	⇝	$\mathtt{F}_s$; `ld.cg` ; $\mathtt{F}_s$	
(77)	`fence(`s`)`	⇝	$\mathtt{F}_s$	

Table 1. OpenCL/PTX compiler mapping, program code level

6.2 Compiling OpenCL to PTX

In this subsection, we develop and check a compiler mapping from OpenCL to PTX. First, using **Q4**, we show that a natural mapping is invalid for an existing formalisation of the PTX MCM (PTX1) [9], but valid for a stronger model that we develop (PTX2). Then, we use **Q2** to generate litmus tests that distinguish PTX2 from PTX1, which we use to confirm that our PTX2 remains empirically sound for NVIDIA GPUs.

Figure 14 defines the PTX1 MCM. The model enforces coherence (but not between reads, à la Sparc RMO [8]) (68), allows any fence to restore SC within a workgroup (69), and allows device-scoped fences to restore SC throughout the device (70). We omit dependencies because they are difficult to test: the PTX compiler often removes our artificially-inserted dependencies.

Table 1 defines the OpenCL/PTX mapping we use. OpenCL stores are mapped to PTX stores (`st.cg`) (71) (72) (73), and OpenCL loads are mapped to PTX loads (`ld.cg`) (74) (75), with fences (`membar`) placed before and/or after the memory access depending on whether the OpenCL instruction is non-atomic, relaxed, acquire, release, or SC. PTX fences are required even for relaxed OpenCL loads because PTX allows consecutive loads from the same address to be re-ordered (see (68)). The PTX fences are scoped to match the scope s of the OpenCL instruction. In line with prior work on the PTX MCM [9], we exclude RMWs, local memory, and multi-GPU interactions.

Figure 15 shows how the code-level mappings in (the first two rows of) Tab. 1 are encoded at the execution level. OpenCL workgroups correspond to PTX workgroups (79). An OpenCL write without release semantics (e) maps to a single PTX write (e') (80), while an OpenCL workgroup-scoped release write (e) maps to a workgroup-scoped fence (e'_1) sequenced before a PTX write (e'_2) (81). The other rows are handled similarly.

We used Alloy to check this mapping against PTX1, and found an execution that is disallowed by OpenCL (Fig. 16, top) but allowed, after compilation, by PTX1 (Fig. 16, bottom). Note that the outer dotted rectangles group threads by workgroup. The crux

$compile_{\mathrm{OpenCL/PTX}}(\pi, X, Y)$

❼❽ $compile(\pi, X, Y)$ ❼❾ $swg_X = \pi;swg_Y;\pi^{-1}$

❽⓿ $\forall e \in W_X - rel_X.\ \exists e' \in W_Y.\ \{e\};\pi = \{e'\}$

❽❶ $\forall e \in W_X \cap rel_X - sc_X - dv_X.\ \exists e'_1 \in F_Y - dv_Y.\ \exists e'_2 \in W_Y.\ (e'_1, e'_2) \in imm(sb_Y) \wedge \{e\};\pi = \{e'_1\} \uplus \{e'_2\}$

Figure 15. OpenCL/PTX mapping, execution level (excerpt)

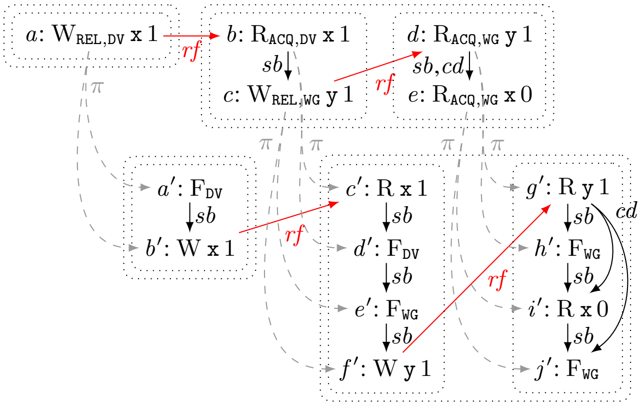


Figure 16. A WRC bug in the OpenCL/PTX1 mapping

here is *cumulative* synchronisation across scopes [36 (§1.7.1)]. The left thread synchronises at workgroup scope with the middle thread (via a and b), which synchronises at device scope with the right thread (via c and d). If PTX1 supported cumulative synchronisation across scopes, the left and right threads would now be synchronised, and the execution above, in which e observes a stale y, would be forbidden, just as it is in OpenCL.

We could fix the mapping for PTX1 by upgrading all the PTX fences in Tab. 1 to device scope. However, because widely-scoped fences incur high performance overhead on NVIDIA GPUs [71], we opt instead to strengthen PTX1 to support our mapping, by enforcing cumulative synchronisation across scopes (obtaining PTX2). This entails changing the definition of hb_{dv} (Fig. 14) to

$$hb_{\mathrm{dv}} \overset{\mathrm{def}}{=} rmo \cup (hb^*_{\mathrm{wg}} \,;\, f_{\mathrm{dv}} \,;\, hb^*_{\mathrm{wg}})$$

so that device-scoped fences (f_{dv}) can restore SC throughout the device even if they are preceded or followed by workgroup-scoped synchronisation (hb_{wg}). After this change, Alloy finds no bugs in our mapping with up to 5 software-level events. It times out (after four hours) when checking larger executions.

It remains to show that PTX2 is sound w.r.t. empirical GPU testing data. To do this, we use **Q2** to find litmus tests (P, σ) that are allowed under PTX1 but disallowed under PTX2. We wish to find not just a single litmus test (as in §4), but as many as possible, so we run Alloy iteratively, each time disallowing the execution shape found previously, until it is unable to find more. We then check testing results (or if results do not exist for a given test, we run new tests) to confirm that (P, σ) cannot pass on actual GPUs.

Using this method, Alloy finds all 14 distinguishing 7-event tests (e.g. WRC [50]), plus 12 of the distinguishing 8-event tests (e.g. IRIW [19]) before timing out. We are able to query Alglave et al.'s experimental results [9] for 22 of these 26 tests. The rest are single-address tests (which arise because PTX does not guarantee coherence in general). These we ran on an NVIDIA GTX Titan using the GPU-litmus tool [9]. We found no behaviours that are allowed by PTX1, disallowed by PTX2, and empirically observable on a GPU [1]. Thus, subject to the available testing data, strengthening PTX1 to PTX2 is sound, and thus the natural OpenCL/PTX compiler mapping can be used.

	Task	$\lvert\mathbb{E}\rvert$	t_{enc}/s	t_{sol}/s		
1	**Q2** C11-SRA vs. C11 (§4.1)	6	0.7	0.6	G	✓
2	**Q2** C11-SwRf vs. C11 (§4.2)	7	0.8	625	G	✗
3	**Q2** C11-SwRf vs. C11 (§4.2)	12	2	214	P	✓
4	**Q2** C11 vs. C11-Orig (§4.3)	5	0.4	0.3	G	✓
5	**Q2** MCA vs. x86 (§4.4)	9	0.8	607	P	✗
6	**Q2** MCA vs. Power (§4.4)	6	2	0.06	G	✓
7	**Q2** SC vs. C11-Draft (§4.5)	4	0.4	0.04	G	✓
8	**Q2** PTX2 vs. PTX1 (§6.2)	7	0.7	4	G	✓
9	**Q3** C11 (sequencing) (§5.1)	5	0.5	163	M	✗
10	**Q3** C11 (sequencing) (§5.1)	6	0.7	5	P	✓
11	**Q3** C11 (mem. orders) (§5.2)	7	0.9	51	G	✓
12	**Q4** C11 / x86	$5+5$	0.7	13029	P	✗
13	**Q4** C11 / Power (§6.3)	$6+8$	8	91	P	✓
14	**Q4** OpenCL / AMD (§6.1)	$2+9$	6	1355	G	✓
15	**Q4** OpenCL / AMD (§6.1)	$4+10$	16	4743	P	✓
16	**Q4** OpenCL / PTX1 (§6.2)	$5+8$	2	11	P	✓
17	**Q4** OpenCL / PTX2 (§6.2)	$5+15$	4	9719	P	✗

Table 2. Summary of tasks. We record the number ($\lvert\mathbb{E}\rvert$) of events in the search space, the time taken (in seconds) to encode (t_{enc}) and then solve (t_{sol}) the SAT query, whether the fastest solver was MiniSat (M), Glucose (G), or Plingeling (P), and whether a solution was found (✓) or not (✗). For **Q4** tasks, $m+n$ means that $\mathbb{E}$ is partitioned into m software-level and n architecture-level events.

6.3 Compiling C11 to Power

Work in progress by Lahav et al. [44] has uncovered a bug in the C11/Power mapping previously thought to have been proven sound by Batty et al. [15].[6] Before becoming aware of their work, we had used Alloy to verify the soundness of the mapping for up to 6 software-level events and up to 6 architecture-level events. Incrementing these bounds any further resulted in intractable solving times, which explains why the bug, which requires 6 software-level events and 13 architecture-level events, had not previously been found by Alloy. In order to recreate Lahav et al.'s result, we modified the C11/Power mapping so as not to place fences at the start or the end of a thread. Removing these redundant fences allows the bug to be expressed using just 8 architecture-level events, and found automatically by Alloy in a reasonable time (see Tab. 2 in the next section).

7. Performance Evaluation

In this section, we report on an empirical investigation of how our design decisions affect Alloy's SAT-solving performance.

Table 2 summarises the tasks on which we have evaluated our technique. We used a machine with four 16-core 2.1 GHz AMD Opteron processors and 128 GB of RAM.

7.1 Choice of SAT Solver

Figure 17 summarises the performance of three SAT solvers on our tasks: Glucose (version 2.1) [13], MiniSat (version 2.2) [26], and Plingeling [17]. Each bar shows the mean solve time over 4 runs, plus the minimum and maximum time.[7] Plingeling is able

[6] Concurrent work by Manerkar et al. [51] has shown the C11/ARMv7 mapping to be similarly flawed.

[7] A more thorough comparison of SAT solvers would control for the order of clauses, which greatly influences performance [59].

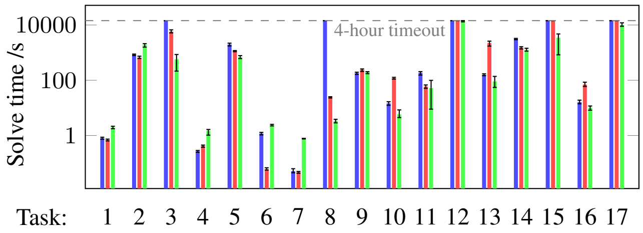


Figure 17. Comparing MiniSat (🟦), Glucose (🟥) and Plingeling (🟩)

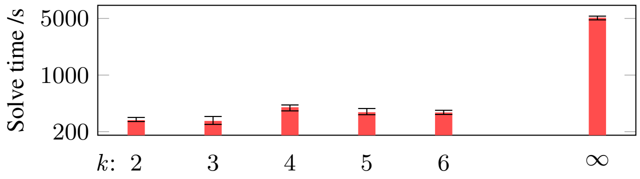


Figure 18. Performance of Task 13 with fixpoints unrolled k times

to complete all tasks, whereas Glucose and MiniSat time out on three and five of them respectively. On the tasks all three solvers complete, MiniSat takes an average of 9 minutes, Glucose takes 8, and Plingeling takes 6. Plingeling's relatively high startup overhead is evident on the quicker tasks.

7.2 Combined vs. Separate Event Sets

For **Q4** tasks (§6), we can choose to draw software-level and architecture-level events either from a single set $\mathbb{E}$ or from disjoint partitions $\mathbb{E}_{\text{sw}}$ and $\mathbb{E}_{\text{arch}}$. The former approach is attractive because it gives the user one fewer parameter to control, and it minimises the total number of events required to find solutions because a single element of $\mathbb{E}$ can represent both a software-level event and an architecture-level event. However, as shown in Tab. 2, we choose the latter approach. To see why, consider Task 17. To validate the OpenCL/PTX mapping for OpenCL executions of up to 5 events, we must consider PTX executions of up to 15 events (since each OpenCL event can map to up to three PTX events). We found that setting $|\mathbb{E}_{\text{sw}}| = 5$ and $|\mathbb{E}_{\text{arch}}| = 15$ led to a tractable constraint-solving problem for Alloy, but that merely setting $|\mathbb{E}| = 15$, which involves fewer events in total but includes OpenCL executions with more than 5 events, rendered the problem insoluble.

7.3 Recursive Definitions vs. Fixed Unrolling

Alglave et al.'s formalisation of the Power MCM in the `.cat` format involves several recursively-defined relations, but Alloy does not support definitions of the form '`let rec` $r = f(r)$'. Accordingly, in our formalisation of the Power MCM [1], we expand the recursive construction explicitly, by requiring the existence of an r satisfying $f(r) \subseteq r$ and $\forall r'.\, f(r') \subseteq r' \Rightarrow r \subseteq r'$. The latter constraint involves universal quantification over relations and hence requires the higher-order AlloyStar solver [52]. A first-order alternative is simply to unroll the recursive definitions a few times; that is, to set $r \overset{\text{def}}{=} f^k(\emptyset)$ for a fixed k. We found, for the small search scopes involved in our work, that $k \geq 2$ is sufficient for avoiding false positives when checking a compiler mapping from C11 (Task 13). Figure 18 shows that the proper fixpoint construction (i.e., $k = \infty$) is much more expensive than a fixed unrolling.

8. Related Work

Existing tools for MCM reasoning typically take a concurrent program as input, and produce all of the executions allowed under a given MCM. Some rely on SAT solvers to discover executions [18, 25, 75, 83], while others enumerate them explicitly [10, 12, 16, 60, 67, 68]. Our work tackles, in a sense, the 'inverse' problem: we start with executions that witness interesting MCM properties, and go on to produce programs that can give rise to these executions.

***Other works tackling* Q1–Q4** Regarding **Q1**, Darbari et al. [24] have automatically generated conformance tests for HSA [35] from an Event-B specification [2] of its MCM. Alglave et al.'s diy tool [11] generates conformance tests for a range of MCMs based on *critical cycles* [70]. We find that our Alloy-based approach has several advantages over diy when generating conformance tests. First, we can straightforwardly handle custom MCM constructs (e.g., C11 memory orders) while diy currently does not. Second, we generate only tests needed for conformance, while diy generates many more. Third, diy can also miss some tests required to distinguish two MCMs (such as the single-address tests we saw in §6.2), if not guided carefully through user-supplied critical cycles.

Regarding **Q2**, there is a long history of unifying frameworks for comparing MCMs [3, 23, 28, 31, 32, 70], and of manual proofs that different formulations of the same MCM are equivalent [50, 63]. On the automation side, Mador-Haim et al. [48] have, like ourselves, used a SAT solver as part of a tool for generating litmus tests that distinguish MCMs. However, where we use the solver to *generate* tests, they just use it to *check* the behaviour of pre-generated tests. Given that generating all 6-instruction litmus tests takes them 'a few minutes' (in 2010), we expect that their approach of explicit enumeration would not scale to find the 12-instruction litmus tests that are sometimes necessary to distinguish MCMs (see §4.2), and which Alloy is able to find in just 4 minutes.

Prior work has proved (manually) the validity of compiler optimisations in non-SC MCMs [79, 80]. Since optimisations should not introduce new behaviours, this problem is related to monotonicity (**Q3**). On the automation side, the Traver tool [21] uses an automated theorem prover to verify/falsify a given program transformation against a non-SC MCM. Unlike our work, Traver does not support multi-threaded optimisations such as linearisation. Chakraborty et al. have built a tool [22] that automatically verifies that LLVM optimisations preserve C11 semantics. Like Traver, their tool only checks specific *instances* of an optimisation, while our approach is able to check optimisations themselves. Morisset et al. [55] use random testing to validate optimisations (albeit those not involving atomic operations) against the C11 MCM; Vafeiadis et al. [77] then show (manually) that several of these optimisations are invalid when atomic operations are involved. Our work, in turn, shows how several of Vafeiadis et al.'s results can be recreated automatically, often in a simpler form (§5).

Regarding **Q4**, prior work has proved (manually) the correctness of both compiler mappings [15, 16, 82] and full compilers [81] in a non-SC context. These proofs all involve intensive manual effort, in contrast to our lightweight automatic checking, though our checking can of course only guarantee the absence of bugs within Alloy's search scope. On the automation side, Lustig et al. have built tools for finding and verifying semantics-preserving translations, but where we focus on language/architecture translation, they focus on architecture/microarchitecture [46] and architecture/architecture [47] translation. Very recent work by Trippel et al. [76] has produced a framework that can check language/architecture mappings; this works by enumerating standard litmus tests and then simulating each one against both the language-level MCM and, after compilation, the architecture-level MCM.

Reflections on Alloy Alloy is a mature, open-source, and widely-used modelling application, and its trio of features – a modular and object-oriented modelling language, an automatic constraint solver, and a graphical visualiser – makes it ideal for MCM development.

Although Tab. 2 shows several lengthy solving times, those figures are obtained once the search space has been set as large as computational feasibility allows. Given smaller search spaces, as would be appropriate during MCM prototyping, Alloy is suitable for interactive use.

When does Alloy's failure to distinguish two MCMs become a proof that they are equivalent? Mador-Haim et al. [49] prove that 6 events are enough, but their result applies only to multi-copy atomic, architecture-level MCMs (see §4.4). Momtahan [54] gives a result for general Alloy models, but imposes strong restrictions on quantifiers that our models do not meet.

Ivy [64] defines a relational modelling language similar to Alloy's. Where Alloy ensures the decidability of instance-finding by restricting to a finite search scope (which limits its usefulness for verification), Ivy instead restricts formulas to the form $\exists \bar{x}. \forall \bar{y}. \phi$ (for quantifier-free ϕ). If our models can be rephrased to fit into Ivy's restricted language, there is the potential not just to 'debug' MCM properties, but to *verify* them. Another language related to Alloy, Rosette [74], is used in very recent work by Bornholt et al. [20] to solve the problem of synthesising a MCM from a set of desired litmus test outcomes.

We find that Alloy has several advantages over other frameworks that have been used to reason about MCMs, such as Isabelle (e.g., [16]), Lem [56] (e.g., [15]), Coq (e.g., [77]), and `.cat` [12] (e.g., [14]). A key advantage is that the entire memory modelling process can be conducted within Alloy: the Alloy modelling language can express programming languages, compiler mappings, MCMs, and properties to test, the Alloy Analyzer can discover solutions, and the Alloy Visualizer can display solutions using theming customised for the model at hand. Alglave et al.'s `.cat` framework, like Alloy, allows MCM axioms to be expressed in the concise propositional relation calculus [73], but Alloy also supports the more powerful predicate calculus as a fallback. As such, Alloy is expressive enough to capture both axiomatic and operational MCMs, while remaining sufficiently restrictive that fully-automatic property checking is computationally tractable.

9. Conclusion

By solving relational constraints between executions and then lifting solutions to litmus tests, Alloy can generate conformance tests, compare MCMs (against each other and against general properties like SC-DRF and multi-copy atomicity), and check monotonicity and compiler mappings. As such, we believe that Alloy should become an essential component of the memory modeller's toolbox. Indeed, we are already working with two large processor vendors to apply our technique to their recent and upcoming architectures and languages. Other future work includes applying our technique to more recent MCMs that are defined in a non-axiomatic style [27, 40, 41, 65].

Although Alloy's lightweight, automatic approach cannot give the same universal assurance as fully mechanised theorems, we have found it invaluable in practice, because even (and perhaps *especially*) in the complex and counterintuitive world of non-SC MCMs, Jackson's maxim [39] holds true: *Most bugs have small counterexamples*.

Acknowledgements

We thank Alastair Donaldson, Carsten Fuhs, Daniel Jackson, Luc Maranget, and Matthew Parkinson for their valuable feedback. We acknowledge the support of a Research Fellowship from the Royal Academy of Engineering and the Lloyd's Register Foundation (Batty), a Research Chair from the Royal Academy of Engineering and Imagination Technologies (Constantinides), an EPSRC Impact Acceleration Award, EPSRC grants EP/I020357/1, EP/K034448/1, and EP/K015168/1, and a gift from Intel Corporation.

References

[1] Supplementary material for this paper is available in the ACM digital library, and in the following GitHub repository. URL: `https://johnwickerson.github.io/memalloy`.

[2] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. "Rodin: an open toolset for modelling and reasoning in Event-B". In: *Int. J. Softw. Tools Technol. Transfer* 12 (2010).

[3] Sarita V. Adve. "Designing Memory Consistency Models For Shared-Memory Multiprocessors". PhD thesis. University of Wisconsin-Madison, 1993.

[4] Sarita V. Adve and Kourosh Gharachorloo. "Shared memory consistency models: A tutorial". In: *IEEE Computer* 29.12 (1996).

[5] Sarita V. Adve and Mark D. Hill. "Weak Ordering - A New Definition". In: *Int. Symp. on Computer Architecture (ISCA)*. 1990.

[6] Mustaque Ahamad, Rida A. Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. "The Power of Processor Consistency". In: *ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*. 1993.

[7] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Second edition. Addison-Wesley, 2006.

[8] Jade Alglave. "A formal hierarchy of weak memory models". In: *Formal Methods in System Design* 41 (2012).

[9] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. "GPU concurrency: weak behaviours and programming assumptions". In: *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2015.

[10] Jade Alglave, Daniel Kroening, and Michael Tautschnig. "Partial Orders for Efficient Bounded Model Checking of Concurrent Software". In: *Computer Aided Verification (CAV)*. 2013.

[11] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. "Fences in Weak Memory Models". In: *Computer Aided Verification (CAV)*. 2010.

[12] Jade Alglave, Luc Maranget, and Michael Tautschnig. "Herding cats: modelling, simulation, testing, and data-mining for weak memory". In: *ACM Trans. on Programming Languages and Systems (TOPLAS)* 36.2 (2014).

[13] Gilles Audemard and Laurent Simon. "Predicting Learnt Clauses Quality in Modern SAT Solvers". In: *Int. Joint Conf. on Artificial Intelligence (IJCAI)*. 2009.

[14] Mark Batty, Alastair F. Donaldson, and John Wickerson. "Overhauling SC atomics in C11 and OpenCL". In: *ACM Symp. on Principles of Programming Languages (POPL)*. 2016.

[15] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. "Clarifying and Compiling C/C++ Concurrency: from C++11 to POWER". In: *ACM Symp. on Principles of Programming Languages (POPL)*. 2012.

[16] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. "Mathematizing C++ Concurrency". In: *ACM Symp. on Principles of Programming Languages (POPL)*. 2011.

[17] Armin Biere. *Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010*. Tech. rep. 10/1. Institute for Formal Models and Verification, Johannes Kepler University, 2010.

[18] Jasmin Christian Blanchette, Tjark Weber, Mark Batty, Scott Owens, and Susmit Sarkar. "Nitpicking C++ Concurrency". In: *Int. Symp. on Principles and Practice of Declarative Programming (PPDP)*. 2011.

[19] Hans-J. Boehm and Sarita V. Adve. "Foundations of the C++ Concurrency Memory Model". In: *ACM Conf. on Programming Language Design and Implementation (PLDI)*. 2008.

[20] James Bornholt and Emina Torlak. *Synthesizing Memory Models from Litmus Tests*. Tech. rep. UW-CSE-16-10-01. University of Washington, 2016.

[21] Sebastian Burckhardt, Madanlal Musuvathi, and Vasu Singh. "Verifying Local Transformations on Relaxed Memory Models". In: *Int. Conf. on Compiler Construction (CC)*. 2010.

[22] Soham Chakraborty and Viktor Vafeiadis. "Validating Optimizations of Concurrent C/C++ Programs". In: *Int. Symp. on Code Generation and Optimization (CGO)*. 2016.

[23] William W. Collier. *Reasoning about Parallel Architectures*. Prentice Hall, 1992.

[24] Ashish Darbari, Iain Singleton, Michael Butler, and John Colley. "Formal Modelling, Testing and Verification of HSA Memory Models using Event-B". Draft. 2016. URL: `http://arxiv.org/pdf/1605.04744v1.pdf`.

[25] Brian Demsky and Patrick Lam. "SATCheck: SAT-Directed Stateless Model Checking for SC and TSO". In: *ACM Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2015.

[26] Niklas Eén and Niklas Sörensson. "An Extensible SAT-solver". In: *Theory and Applications of Satisfiability Testing (SAT)*. 2003.

[27] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. "Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA". In: *ACM Symp. on Principles of Programming Languages (POPL)*. 2016.

[28] Kourosh Gharachorloo. "Memory Consistency Models for Shared-Memory Multiprocessors". PhD thesis. Stanford University, 1995.

[29] Blake A. Hechtman, Shuai Che, Derek R. Hower, Yingying Tian, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. "QuickRelease: A Throughput-oriented Approach to Release Consistency on GPUs". In: *IEEE Int. Symp. on High Performance Computer Architecture (HPCA)*. 2014.

[30] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Fifth edition. Morgan Kaufmann, 2012.

[31] Lisa Higham, LillAnne Jackson, and Jalal Kawash. "Specifying Memory Consistency of Write Buffer Multiprocessors". In: *ACM Trans. on Programming Languages and Systems (TOPLAS)* 25.1 (2007).

[32] Lisa Higham, Jalal Kawash, and Nathaly Verwaal. "Defining and Comparing Memory Consistency Models". In: *Int. Conf. on Parallel and Distributed Computing Systems (PDCS)*. 1997.

[33] C. A. R. Hoare. "Proof of Correctness of Data Representations". In: *Acta Informatica* 1 (1972).

[34] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. "Heterogeneous-race-free Memory Models". In: *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2014.

[35] HSA Foundation. *HSA Platform System Architecture Specification*. Version 1.0, 2015. URL: `http://www.hsafoundation.com/standards/`.

[36] IBM. *Power ISA (Version 2.06B)*. 2010.

[37] ISO/IEC. *Programming languages – C++*. Draft N3092, 2010. URL: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf`.

[38] ISO/IEC. *Programming languages – C*. International standard 9899:2011, 2011.

[39] Daniel Jackson. *Software Abstractions – Logic, Language, and Analysis*. Revised edition. MIT Press, 2012.

[40] Alan Jeffrey and James Riely. "On Thin Air Reads: Towards an Event Structures Model of Relaxed Memory". In: *ACM/IEEE Symp. on Logic in Computer Science (LICS)*. 2016.

[41] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. "A Promising Semantics for Relaxed-Memory Concurrency". In: *ACM Symp. on Principles of Programming Languages (POPL)*. 2017.

[42] Khronos Group. *The OpenCL Specification*. Version 2.0, 2013. URL: `https://www.khronos.org/registry/cl/`.

[43] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. "Taming Release-Acquire Consistency". In: *ACM Symp. on Principles of Programming Languages (POPL)*. 2016.

[44] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. *Repairing Sequential Consistency in C/C++11*. Tech. rep. MPI-SWS-2016-011. MPI-SWS, 2016.

[45] Leslie Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". In: *IEEE Transactions on Computers* C-28.9 (1979).

[46] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. "PipeCheck: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models". In: *Int. Symp. on Microarchitecture (MICRO)*. 2014.

[47] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. "ArMOR: Defending Against Memory Consistency Model Mismatches in Heterogeneous Architectures". In: *Int. Symp. on Computer Architecture (ISCA)*. 2015.

[48] Sela Mador-Haim, Rajeev Alur, and Milo M. K. Martin. "Generating Litmus Tests for Contrasting Memory Consistency Models". In: *Computer Aided Verification (CAV)*. 2010.

[49] Sela Mador-Haim, Rajeev Alur, and Milo M. K. Martin. "Litmus Tests for Comparing Memory Consistency Models: How Long Do They Need to Be?" In: *Design Automation Conference (DAC)*. 2011.

[50] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. "An Axiomatic Memory Model for POWER Multiprocessors". In: *Computer Aided Verification (CAV)*. 2012.

[51] Yatin A. Manerkar, Caroline Trippel, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. "Counterexamples and Proof Loophole for the C/C++ to POWER and ARMv7 Trailing-Sync Compiler Mappings". 2016. URL: `http://arxiv.org/pdf/1611.01507v1.pdf`.

[52] Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, and Daniel Jackson. "Alloy*: A General-Purpose Higher-Order Relational Constraint Solver". In: *Int. Conf. on Software Engineering (ICSE)*. 2015.

[53] Rolf H. Möhring. "Computationally tractable classes of ordered sets". In: *Algorithms and Order*. Ed. by Ivan Rival. Springer, 1989.

[54] Lee Momtahan. "Towards a Small Model Theorem for Data Independent Systems in Alloy". In: *Electronic Notes in Theoretical Computer Science* 128 (2005).

[55] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. "Compiler Testing via a Theory of Sound Optimisations in the C11/C++11 Memory Model". In: *ACM Conf. on Programming Language Design and Implementation (PLDI)*. 2013.

[56] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. "Lem: reusable engineering of real-world semantics". In: *ACM Int. Conf. on Functional Programming (ICFP)*. 2014.

[57] Roger M. Needham. "Names". In: *Distributed Systems*. Ed. by Sape Mullender. ACM Press, 1989.

[58] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. "An Operational Semantics for C/C++11 Concurrency". In: *ACM Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2016.

[59] Mladen Nikolić. "Statistical Methodology for Comparison of SAT Solvers". In: *Theory and Applications of Satisfiability Testing (SAT)*. 2010.

[60] Brian Norris and Brian Demsky. "CDSChecker: Checking Concurrent Data Structures Written with C/C++ Atomics". In: *ACM Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2013.

[61] NVIDIA. *Parallel Thread Execution ISA, version 4.3*. 2015. URL: `http://docs.nvidia.com/cuda/pdf/ptx_isa_4.3.pdf`.

[62] Marc S. Orr, Shuai Che, Ayse Yilmazer, Bradford M. Beckmann, Mark D. Hill, and David A. Wood. "Synchronization Using Remote-Scope Promotion". In: *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2015.

[63] Scott Owens, Susmit Sarkar, and Peter Sewell. "A Better x86 Memory Model: x86-TSO". In: *Theorem Proving in Higher Order Logics (TPHOLs)*. 2009.

[64] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. "Ivy: Safety Verification by Interactive Generalization". In: *ACM Conf. on Programming Language Design and Implementation (PLDI)*. 2016.

[65] Jean Pichon-Pharabod and Peter Sewell. "A Concurrency Semantics for Relaxed Atomics that Permits Optimisation and Avoids Thin-Air Executions". In: *ACM Symp. on Principles of Programming Languages (POPL)*. 2016.

[66] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. "Synchronising C/C++ and POWER". In: *ACM Conf. on Programming Language Design and Implementation (PLDI)*. 2012.

[67] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. "Understanding POWER Multiprocessors". In: *ACM Conf. on Programming Language Design and Implementation (PLDI)*. 2011.

[68] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. "The Semantics of x86-CC Multiprocessor Machine Code". In: *ACM Symp. on Principles of Programming Languages (POPL)*. 2009.

[69] Ali Sezgin. "Formalization and Verification of Shared Memory". PhD thesis. University of Utah, 2004.

[70] Dennis Shasha and Marc Snir. "Efficient and Correct Execution of Parallel Programs that Share Memory". In: *ACM Trans. on Programming Languages and Systems (TOPLAS)* 10.2 (1988).

[71] Tyler Sorensen and Alastair F. Donaldson. "Exposing Errors Related to Weak Memory in GPU Applications". In: *ACM Conf. on Programming Language Design and Implementation (PLDI)*. 2016.

[72] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Ed. by Mark D. Hill. Vol. 16. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2011.

[73] Alfred Tarski. "On the Calculus of Relations". In: *Journal of Symbolic Logic* 6.3 (1941), pp. 73–89.

[74] Emina Torlak and Rastislav Bodik. "Growing Solver-Aided Languages with Rosette". In: *Onward!* 2013.

[75] Emina Torlak, Mandana Vaziri, and Julian Dolby. "MemSAT: Checking Axiomatic Specifications of Memory Models". In: *ACM Conf. on Programming Language Design and Implementation (PLDI)*. 2010.

[76] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. "Exploring the Trisection of Software, Hardware, and ISA in Memory Model Design". Draft. 2016. URL: `http://arxiv.org/pdf/1608.07547v1.pdf`.

[77] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. "Common compiler optimisations are invalid in the C11 memory model and what we can do about it". In: *ACM Symp. on Principles of Programming Languages (POPL)*. 2015.

[78] Jacobo Valdes, Robert E. Tarjan, and Eugene L. Lawler. "The recognition of Series Parallel digraphs". In: *ACM Symp. on Theory of Computing (STOC)*. 1979.

[79] Jaroslav Ševčík. "Safe Optimisations for Shared-Memory Concurrent Programs". In: *ACM Conf. on Programming Language Design and Implementation (PLDI)*. 2011.

[80] Jaroslav Ševčík and David Aspinall. "On Validity of Program Transformations in the Java Memory Model". In: *Europ. Conf. on Object-Oriented Programming (ECOOP)*. 2008.

[81] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. "Relaxed-Memory Concurrency and Verified Compilation". In: *ACM Symp. on Principles of Programming Languages (POPL)*. 2011.

[82] John Wickerson, Mark Batty, Alastair F. Donaldson, and Bradford M. Beckmann. "Remote-scope promotion: clarified, rectified, and verified". In: *ACM Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2015.

[83] Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. "Nemos: A Framework for Axiomatic and Executable Specifications of Memory Consistency Models". In: *Int. Parallel and Distributed Processing Symp. (IPDPS)*. 2004.