# Automated Synthesis of Comprehensive Memory Model Litmus Test Suites

Daniel Lustig
NVIDIA
dlustig@nvidia.com

Andrew Wright
MIT
acwright@mit.edu

Alexandros Papakonstantinou
NVIDIA
alexandrosp@nvidia.com

Olivier Giroux
NVIDIA
ogiroux@nvidia.com

## Abstract

The memory consistency model is a fundamental part of any shared memory architecture or programming model. Modern weak memory models are notoriously difficult to define and to implement correctly. Most real-world programming languages, compilers, and (micro)architectures therefore rely heavily on black-box testing methodologies. The success of such techniques requires that the suite of litmus tests used to perform the testing be comprehensive—it should ideally stress all obscure corner cases of the model and of its implementation. Most litmus test suites today are generated from some combination of manual effort and randomization; however, the complex and subtle nature of contemporary memory models means that manual effort is both error-prone and subject to incomplete coverage.

This paper presents a methodology for synthesizing comprehensive litmus test suites directly from a memory model specification. By construction, these suites contain all tests satisfying a minimality criterion: that no synchronization mechanism in the test can be weakened without causing new behaviors to become observable. We formalize this notion using the Alloy modeling language, and we apply it to a number of existing and newly-proposed memory models. Our results show not only that this synthesis technique can automatically reproduce all manually-generated tests from existing suites, but also that it discovers new tests that are not as well studied.

***CCS Concepts*** • **Computer systems organization** → **Multicore architectures**; • **Hardware** → **Coverage metrics**; • **Software and its engineering** → **Synchronization**

***Keywords*** memory consistency models, litmus tests, synchronization, synthesis

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

*ASPLOS '17* April 8–12, 2017, Xi'an, China.
© 2017 ACM. ISBN 978-1-4503-4465-4/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3037697.3037723

## 1. Introduction

Memory consistency models specify the set of values that can be legally returned by memory loads. For threads or programs to communicate through shared memory, it is imperative that the memory model be well defined and properly implemented. Unfortunately, both tasks have proven extremely difficult; nearly every major processor vendor has fallen victim to recall-caliber memory consistency bugs (Alglave et al. 2015a; AMD 2012; ARM 2011; Intel 2016a).

Today, memory model verification methodologies rely heavily on testing. The basic unit of testing for a memory model is a *litmus test*, a small, stylized program designed to stress certain behaviors of the model. An example is shown in Figure 1. Litmus tests will generally have a number of different possible executions. The *outcome* of a litmus test is the set of values returned by the loads during execution plus the set of values stored in memory at the end of the execution. The role of a memory model is therefore to declare which test outcomes are legal and which are not.

A suite of litmus tests is fully useful only if it is comprehensive: if it stresses every possible behavior of the model and/or implementation. Unfortunately, it is impractical to exhaustively consider all possible programs. Memory models are becoming increasingly sophisticated, with different ordering strength annotations (e.g., in C/C++: relaxed, acquire, release, sequentially consistent), different scopes (e.g., in OpenCL: work-item, work-group, kernel), different fences (e.g., on Power: lightweight `lwsync` vs. heavyweight `sync`), and so on. These features provide important (and mostly orthogonal) performance benefits and hence are unlikely to disappear. However, the set of all possible tests generally grows exponentially both with the number of features and with the test size bound. Memory model testing methodologies must therefore be fully capable of scaling up to these increasingly large design spaces.

Existing litmus test generation techniques cannot provide any guarantee of comprehensiveness, and many struggle even to attain non-trivial coverage of the design space. Most existing test generation methodologies combine manual effort with randomization, but manual effort is error-prone, and neither technique can guarantee completeness of coverage. Symmetry-based filtering does help, but alone it is

```
St [data], 1             || Ld.acquire r1 = [flag]
St.release [flag], 1     || Ld r2 = [data]
```
Legal: (r1=0,r2=0),(r1=0,r2=1),(r1=1,r2=1)
Illegal: (r1=1,r2=0)

Figure 1: Message Passing (MP) litmus test

insufficient even for simpler memory models (Mador-Haim et al. 2010). Making matters worse is the fact that certain bugs only appear in rare circumstances, and so external stressors may need to be applied to induce the buggy behaviors (Alglave et al. 2010; Sorensen and Donaldson 2016). Stressors add yet another dimension to the design space.

This paper presents a methodology for automatically generating comprehensive-by-construction litmus test suites specific to any axiomatically-specified memory model. We define a *litmus test minimality criterion* which states that no synchronization mechanism in a test can be weakened without allowing new outcomes to become observable. Using the Alloy model finder (Jackson 2002), we then automatically synthesize all tests which satisfy the criterion with respect to the rules of a given memory model. These tests can then be fed into any existing testing infrastructure.

We applied our test generation technique to a number of popular real-world memory models. As our results show, our technique is able both to prune out redundant tests as well as to identify tests that were omitted from the existing suites. By focusing only on tests which satisfy the minimality criterion, our synthesis technique therefore quantifiably increases the effective coverage while simultaneously minimizing the time spent executing redundant tests and mitigating the cost of human error.

## 2. Background and Related Work

### 2.1 Litmus Tests

Litmus testing is a well-established means of stress testing memory models and their implementations. Existing testing infrastructures have uncovered numerous bugs in compilers, architecture specifications, and microarchitectures, even when certain outcomes only appear on the order of once every billion executions (Alglave et al. 2015a, 2014; Hangal et al. 2004; Manovit et al. 2006; Owens et al. 2009; Sarkar et al. 2011b). Previous work has also identified external stressors which make uncommon behaviors appear more reliably (Sorensen and Donaldson 2016). The focus of this paper is on improving the litmus test suites themselves rather than on improving the rest of the testing infrastructure.

Traditionally, litmus tests have been derived from three sources. First, they may be manually generated by a designer who reasoned intuitively about some particular memory model feature. Second, they may be automatically generated by a random test generator (that may use guided randomness). Third, tests deemed interesting from either of the previous two sources may be gathered into a repository and forwarded along to future suites. These approaches have been extremely useful in identifying many well-known patterns common across a range of models, but they do not help in automatically identifying new corner cases that can arise due to new features of new models. Our techniques aim at complementing these existing approaches by identifying interesting tests which manual analysis and/or random generation may have missed.

The `diy` tool generates litmus tests by considering specific relaxations of sequential consistency (Alglave et al. 2010). This approach ties the synthesis to the particular phrasing of the memory model, and it requires the relaxations to be provided as input. In contrast, our technique automatically enumerates the complete set of interesting relaxations, and it ties the synthesis to ISA-specific instruction properties rather than to patterns specific to one particular formalization of a memory model.

### 2.2 Axiomatic Memory Model Specifications

Although various approaches have been used to define memory consistency models over the years, we follow the axiomatic approach in this paper. Axiomatic memory models define legal program executions as those which satisfy some set of constraints on specific relations between the instructions in the program. This style has successfully been used to define a wide range of memory models, including those used by C/C++, OpenCL, x86, Power, and ARM (Alglave et al. 2014; Batty et al. 2016; International Organization for Standardization (ISO) 2011a,b; Mador-Haim et al. 2012).

While the specifics can and do vary by formulation, the most fundamental relations have become more or less standardized. "Program order" (`po`) relates each instruction with later instructions in the same thread. The "reads-from" (`rf`) relation exists between a source write and any read returning its value (in one particular execution). Coherence (`co`) establishes a total order (per execution) over all stores from all threads to each address. Lastly, "from-reads" ($\texttt{fr}=\texttt{rf}^{-1};\texttt{co}$), sometimes called "reads-before", is the inverse of `rf` followed by `co`, and `po_loc` relates accesses with accesses later in the same thread and to the same memory location. Relations between different threads are suffixed with "e" for external (e.g., `rf` between threads becomes `rfe`).

Axioms are defined as constraints on the relations of the model. For example, the "Sequential Consistency (SC) per Location" axiom is defined as follows:

$$\texttt{acyclic(rf} \cup \texttt{co} \cup \texttt{fr} \cup \texttt{po_loc)}$$

These axioms are what vary from model to model.

Many tools have been built for exploring axiomatic models for the software, the architecture, and the microarchitecture spaces (Alglave et al. 2014; Lustig et al. 2016; Mador-Haim et al. 2012). In this paper, we use the more general Alloy framework (Jackson 2002), as explained in Section 4.

| Thread 0 | Thread 1 |
|---|---|
| `St.release [data], 1` | `Ld.acquire r1 = [flag]` |
| `St.release [flag], 1` | `Ld.acquire r2 = [data]` |

Legal: (r1=0,r2=0),(r1=0,r2=1),(r1=1,r2=1)
Illegal: (r1=1,r2=0)

Figure 2: A flavor of MP with two releases and two acquires. The extra synchronization as compared to Figure 1 does not prevent any outcomes not already forbidden in the original.

## 3. Instruction Relaxations

Although it would be prohibitively expensive to exhaustively cover the set of all possible litmus tests, it is unnecessary to do so as such a suite would contain a large amount of redundancy. Tests may use overly-strong synchronization, may contain operations which have no effect, or may not sufficiently stress the memory system. Alternatively, a test may simply duplicate a pattern already covered by another test in the suite. Conversely, a test is in fact useful if it covers some pattern not already tested within the suite. Most interesting are the tests that use the least amount of synchronization necessary to prevent some particular outcome: with fewer constraints, new (and possibly buggy) behaviors would be more likely to appear.

Consider the variant of MP in Figure 2. This test contains extraneous synchronization: the first store and/or second load could be demoted into a relaxed store or load, respectively, without causing any changes in the set of legal vs. illegal outcomes. In this sense it is redundant with the test in Figure 1, and the coverage of a test suite would not be increased by including both.

Of course, the characterization of a litmus test as being minimally-synchronized is dependent on the rules of the memory model being analyzed. For example, in a model based strictly on acquire-release synchronization, the variant of MP in Figure 1 would be considered minimally synchronized. On the other hand, it would not be minimal under the ARM or Power memory models. ARM and Power CPUs guarantee to respect orderings imposed by address and data dependencies, and since either would be cheaper than an acquire operation, neither Figure 1 nor Figure 2 would be minimal with respect to those models.

### 3.1 Instruction Relaxation Basics

We formally define our litmus test minimality criterion in terms of *instruction relaxations*, which transform an input test into an output test which is almost identical, but in which some instruction has weaker synchronization semantics than it did in the input.

**Definition** (Minimality Criterion)**.** *A litmus test satisfies the minimality criterion with respect to a particular memory model if and only if that test has at least one forbidden outcome that becomes observable under every instruction relaxation that can be applied to the test.*

| Thread 0 | Thread 1 |
|---|---|
| ~~`St [data], 1`~~ | `Ld.acquire r1 = [flag]` |
| `St.release [flag], 1` | `Ld r2 = [data]` |

Legal: (r1=1, r2=0)

(a) Removing the first store

| Thread 0 | Thread 1 |
|---|---|
| `St [data], 1` | ~~`Ld.acquire r1 = [flag]`~~ |
| `St.release [flag], 1` | `Ld r2 = [data]` |

Legal: r2=0 (matches (r1=1, r2=0) with r1 removed)

(b) Removing the first load

| Thread 0 | Thread 1 |
|---|---|
| `St [data], 1` | `Ld.acquire r1 = [flag]` |
| `St.release [flag], 1` | ~~`Ld r2 = [data]`~~ |

Legal: r1=1 (matches (r1=1, r2=0) with r2 removed)

(c) Removing the second load

| Thread 0 | Thread 1 |
|---|---|
| `St [data], 1` | `Ld.acquire r1 = [flag]` |
| ~~`St.release [flag], 1`~~ | `Ld r2 = [data]` |

Legal: r2=0 (matches (r1=1, r2=0) with r1 removed)

(d) Removing the second store

Figure 3: Applying **RI** to the MP litmus test. In each case, there exists at least one legal execution producing the subset of outcome (r1=1, r2=0) that is unaffected by **RI**.

To see how instruction relaxations work, consider how the **Remove Instruction (RI)** relaxation would affect MP from Figure 1. As its name suggests, **RI** simply removes one instruction from the input test. To check whether MP satisfies the minimality criterion with respect to **RI**, we must find an outcome that is forbidden by the input test but which is observable after applying RI to each instruction. The only plausible possibility[1] in this case is (r1=1,r2=0).

If **RI** were to remove the store to [data], then the output (r1=1,r2=0) would be easily observable, even under sequential consistency (Figure 3a). If **RI** were to remove either load from MP, then all remaining parts of the outcome (r1=1,r2=0) would suddenly also become observable (Figures 3b and 3c). The last case is more interesting: if **RI** were to remove the store to [flag], the load of [flag] would suddenly become orphaned and hence free to choose any other value to read. In this case, we can simply choose to once again match (r1=1,r2=0) (Figure 3d). Since the otherwise-illegal outcome (r1=1,r2=0) becomes visible when applying **RI** to any instruction in the test, the minimality criterion is satisfied.

### 3.2 Other Instruction Relaxations

Besides **RI**, we consider the following additional instruction relaxations:

**DMO** (Demote Memory Order) and **DF** (Demote Fence): weaken the memory ordering strength parameter of a memory access or fence. We derive the term from its usage in

[1] Any other possibility would require some load to return a value neither written by some store nor taken from the implicit initial condition.

| | | |
|---|---|---|
| | `memory_order_seq_cst` | |
| | `memory_order_acq_rel` | |
| `memory_order_release` | | `memory_order_acquire` |
| | | `memory_order_consume` |
| | `memory_order_relaxed` | |

Table 1: Memory order parameters in C11/C++11, in order of decreasing strength (International Organization for Standardization (ISO) 2011a,b)

the C11/C++11 memory model (Table 1), but we use it in a more general sense (e.g., also for demoting ARMv8 `LDAR` load-acquire opcodes into `LDR` load-relaxed opcodes). In some memory models, there may be multiple variants of **DMO** and **DF**. For example, in C11/C++11, there may one variant which demotes `memory_order_acq_rel` into `memory_order_acquire`, and another which demotes it into `memory_order_release`.

**DRMW** (Decompose Atomic Read-Modify-Write): break an atomic read-modify-write operation or a load-linked/ store-conditional pair into a non-atomic or non-linked read-write pair, respectively. The `po_loc` and data dependencies between the load and the store remain in effect.

**RD** (Remove Dependency): discard any dependencies originating from the targeted instruction. Some memory models (notably ARM and Power) explicitly use address, control, and data dependencies to form lightweight ordering enforcement primitives, and removing dependencies is therefore equivalent to weakening a particular type of synchronization. In other models, dependencies are used to define *out-of-thin-air* executions (Boehm and Demsky 2014), and so removing dependencies could affect the legality of specific outcomes even if dependencies are not directly used for synchronization.

**DS** (Demote Scope): lowers the scope of an instruction. Memory models such as OpenCL allow users to synchronize within an explicit scope (i.e., set of threads), with the idea that smaller scopes will allow for faster synchronization (Khronos Group 2015). However, if the scopes are made too narrow, the synchronization will be insufficient.

### 3.3 Applicability to Different Memory Models

Table 2 shows how the different instruction relaxations of Section 3.2 apply to a range of different memory models. Besides Streamlined Causal Consistency (SCC), a model we introduce in this paper (Section 6.3), the models in the table are all well-studied. In relatively simple models such as Total Store Ordering (TSO) (Owens et al. 2009; SPARC International 1993), there are fewer degrees of freedom within the model, and hence relatively few instruction relaxations are applicable. On the other hand, in sophisticated models such as the Heterogeneous System Architecture (HSA) memory model (Alglave and Maranget 2016), all of the discussed instruction relaxations are relevant.

In between, there are cases in which instruction relaxations should be useful in theory, but which turn out not to be in practice. Sometimes the formalization may be missing features present in the full ISA specification. For example, the Itanium memory model was defined before the out-of-thin-air executions were characterized, and therefore it does not address them. Likewise, features such as `eieio` on Power or `dmb.st` on ARM are still not axiomatically formalized due to a lack of clarity in the architecture manuals (Alglave et al. 2014). For the sake of completeness, Table 2 attempts to list all instruction relaxations, but our experiments only use those which are directly applicable within the available formalizations.

Other times, there may be deeper fundamental issues involved. For example, formalization of "no out-of-thin-air behavior" axioms is notoriously difficult and a major open problem in the study of software memory models (Boehm and Demsky 2014). Formalizations of models such as C/C++ and OpenCL therefore often simply do not attempt to axiomatize it. In such situations, until the out-of-thin-air specification problem is resolved, instruction relaxations relevant only to non-axiomatized rules must likewise be skipped.

## 4. Formalizing the Minimality Criterion

We put our minimality condition into practice using the Alloy relational model finder (Jackson 2002). The details are described below.

### 4.1 Defining Memory Models Using Alloy

Alloy provides a domain-specific language for defining relational models: models consisting of a set of atoms (or, for our purposes, nodes in a graph), and relations (edges in a graph). The relational approach makes it a natural fit for describing axiomatic memory models (Wickerson et al. 2017). Alloy feeds its models into the Kodkod relational model finder, which in turn uses off-the-shelf SAT solvers to search for instances of the given model (Torlak and Jackson 2007). This flow provides a convenient front end suited for asking questions about memory models and/or about particular litmus tests. The basic Alloy syntax is summarized in Table 3.

Figure 4 summarizes how the TSO memory model can be defined using Alloy. This formulation adds atomic read-modify-write operations to the formulation of Alglave et al. (Alglave et al. 2014). Our modeling approach is very similar to the approach used by `herd` and its `cats` language; however, Alloy provides a much more programmable interface for re-defining features hard-coded into `herd`. It is this flexibility that allows us to define the litmus test minimality criterion in the way described below.

At the top of the model, a number of type signatures ("sigs") are defined in a hierarchical manner. Each sig defines the set of relations that originate from atoms of that type; each relation defines its own range as appropriate. Below the sig definitions are the basic memory model con-

|  | RI | DRMW | DF | DMO | RD | DS |  |
|---|---|---|---|---|---|---|---|
| SC | ✓ | ✓ | — | — | — | — | (Lamport 1979) |
| TSO | ✓ | ✓ | — | — | — | — | (Owens et al. 2009; SPARC International 1993) |
| Power | ✓ | —[1] | ✓ | — | ✓ | — | (Alglave et al. 2014; Power.org 2013) |
| ARMv7 | ✓ | —[1] | —[1] | — | ✓ | — | (Alglave et al. 2014; ARM Holdings 2016) |
| ARMv8 | ✓ | —[1] | —[1] | ✓ | ✓ | — | (ARM Holdings 2016) |
| Itanium | ✓ | ✓ | — | ✓ | —[1] | — | (Intel 2002) |
| SCC | ✓ | ✓ | ✓ | ✓ | ✓[2] | — | [§6.3] |
| HSA | ✓ | ✓ | ✓ | ✓ | ✓[2] | ✓ | (Alglave and Maranget 2016) |
| C/C++ | ✓ | ✓ | ✓ | ✓ | —[1] | — | (Batty et al. 2016; ISO 2011a; ISO 2011b) |
| OpenCL | ✓ | ✓ | ✓ | ✓ | —[1] | ✓ | (Batty et al. 2016; Khronos Group 2015) |

[1] Would apply if model formalizations filled in the missing features.
[2] Dependencies not used directly for synchronization; **RD** applies to no-thin-air axioms only.

Table 2: Applicability of different instruction relaxations to different hardware and software memory models.

| Symbol | Meaning |
|---|---|
| `+` | Union |
| `&` | Intersection |
| `-` | Set Difference |
| `^` | Transitive Closure (1 or more) |
| `*` | Reflexive Transitive Closure (0 or more) |
| `.` | Relational Join |
| `a[b]` | Relational Join (= `b.a`) or function call |
| `~` | Inverse Relation (a→b becomes b→a) |
| `->` | Cross Product (all edges from src set to dst set) |
| `<:` | Domain restriction |
| `:>` | Range restriction |
| `=>` | Implication |

Table 3: Alloy syntax key. Alloy uses prefix notation.

straints ("facts"). Any valid instance of the model is required to satisfy all of the listed facts to be considered well-formed, regardless of whether it is considered valid according to the memory model itself. Alloy also allows for the definition of functions ("fun"), which return relations of the specified types, and predicates ("pred"), which return boolean truth values. In the context of memory models, there is generally a top-level predicate (e.g., "`tso`" in Figure 4) which specifies the properties that must hold true for a well-formed execution to also be considered a valid execution.

### 4.2 Formalizing the Minimality Criterion in Alloy

The mathematical phrasing of the minimality criterion defined in Section 3.1 is shown in Figures 5a and 5b. This phrasing makes a precise distinction between the following three concepts. A litmus test consists of the basic static properties defining the test: the set of events, `po`, `address`, `dep`, `rmw`, and so on. An outcome of that test is the set of dynamic values that are directly observable at the conclusion of one execution of the test: generally, `rf` and the final value at each memory location. An execution consists of the outcome as well as any auxiliary relations (such as `co`) which are not directly observable as part of an outcome[2].

```
abstract sig Event {
  po: set Event
}
abstract sig MemoryEvent {
  address: one Address
}
sig Write extends MemoryEvent {
  rf: set Read, co: set Write
}
sig Read extends MemoryEvent {
  rmw: lone Write // lone = zero or one
}
// Alternative to fr=~rf.co that accounts for
// implicit initial writes as well
fun fr : Read->Write {
  (Read <: address.~address :> Write) - ~rf.*~co
}
fact { // atomic Read/Write pairs must be adjacent
  rmw in po - po.po
}
sig Fence extends Event {}
fact { rf + co + fr in address.~address }
fact { acyclic[po] }
fun po_loc : MemoryEvent->MemoryEvent {
  po & (address.~address)
}
fun ppo : Event->Event { // preserved prog. order
  po - (Write->Read)
}
fun fence : Event->Event { (po :> Fence).po }
pred acyclic[r: Event->Event] { some iden & *r }
pred tso {
  acyclic[rf + co + fr + po_loc]//SC per Location
  no fre.coe & rmw               // RMW Atomicity
  acyclic[rfe + co + fr + ppo + fence]//Causality
}
```

Figure 4: Defining TSO Using Alloy

[2] It may be possible to infer relations such as `co` by working backwards from some particular outcome, but the fact remains that `co` itself cannot be directly observed without, say, a hardware widget probing the cache coherence protocol.

```
run generate { minimal[axiom] } for 5
```

(a) Alloy searches the set of all tests within the given test size bound for a test satisfying the minimality criterion.

```
let minimal[axiom] {
  // no execution produces the outcome
  (all x: execution | not axiom[x, no_r])
  and
  // under all relaxations, there exists some
  // execution which produces the outcome
  (all r: RTag | all e: Event |
    relaxation_applies[r, e] =>
    exists x': execution | model[x', r->e])
}
```

(b) The general statement of the minimality criterion relies on a higher-order "exists-forall" quantification that first-order solvers such as Alloy+Kodkod cannot directly solve.

```
let minimal[axiom] {
  // the execution is forbidden
  not axiom[no_r]
  and
  // under all relaxations, there exists some
  // execution which produces the outcome
  (all r: RTag | all e: Event |
    relaxation_applies[r, e] => model[r->e])
}
```

(c) Equating outcomes with executions removes the "exists-forall", but at the cost of possibly introducing false negatives.

Figure 5: Defining the minimality criterion in Alloy.

In terms of the three definitions above, the minimality criterion states that given some outcome, no valid execution produces that outcome, but every instruction relaxation application does enable some execution to produce that outcome. Unfortunately, the first portion of this statement contains a higher-order "exists-forall" pattern of the kind which is known to be computationally intractable in general, and which Alloy cannot analyze without the help of experimental additions such as Alloy* (Milicevic et al. 2015).

As an alternative, we can avoid the exists-forall quantification if we eliminate the distinction between executions and outcomes. In other words, if we treat all dynamic relations as being part of the directly observable outcome, then the code of Figure 5b reduces to the code of Figure 5c, which does not suffer from the same "exists-forall" problem. Unfortunately, while pragmatic, this approach is not strictly sound. It effectively moves the quantification over executions before the quantification over relaxations, meaning that relaxation candidates are no longer as free to search for legal executions, and this limitation can lead to false negatives.

Fortunately, for two reasons, false negatives are largely only a theoretical concern in our experiments. First, for `co` in particular, since the `co`-final value is already in fact part of the observable test outcome, it would take a test with at least three writes to the same address for `co` to actually be ambiguous. Within the set of tests as small as the 6-7 instruction bound that we found tractable (see Section 6), such tests are uncommon. Second, for models with other types of auxiliary relations, we can use other workarounds which remain practical and sufficient for small test size bounds. An example of such a workaround is discussed in Section 6.3. We therefore use the code of Figure 5c for our experiments, and we leave the full resolution of this false negative problem for future work.

```
abstract sig RTag {}       // Relaxation Tag
one sig RI extends RTag {} // Remove Instruction
one sig RD extends RTag {} // Remove Dependency
fun rf_p[r: RTag->Event] {
  // rf, but with the domain restricted to
  // non-RI'ed Writes, and with the range
  // restricted to non-RI'ed Reads
  (Write - r[RI]) <: rf :> (Read - r[RI])
}
fun rmw_p[r: RTag->Event] {
  // rmw, but with the domain restricted to
  // non-RI'ed and non-RD'ed Reads, and with
  // the range restricted to non-RI'ed Writes
  (Read - r[RI+RD]) <: rmw :> (Write - r[RI])
}
fun po_p[r: RTag->Event] {
  (Event - r[RI]) <: po :> (Event - r[RI])
}
// ...and so on for the other relations...
pred tso_causality_p[r: RTag->Event] {
  acyclic[(rf_p[r]) + (co_p[r]) +
      (fr_p[r]) + (po_loc_p[r])]
}
```

Figure 6: Implementing Instruction Relaxations in Alloy

### 4.3 Formalizing Instruction Relaxations in Alloy

We now describe how instruction relaxations themselves can be defined within Alloy. Figure 6 shows our approach. We define an abstract base sig "RTag" from which all instruction relaxation definitions are derived. We then define a perturbed version of each primitive relation in the base model. We label these with the suffix "`_p`". Perturbed relations take as input the application of an instruction relaxation to an instruction in the test. The body of each perturbed relation specifies how the given applied instruction relaxation affects instances of the relation which start or end at the chosen instruction.

For example, consider the `rf` relation and the **RI** instruction relaxation. The relation `rf_p`, the perturbed version of `rf`, is defined to include all original `rf` edges, except those whose domain and/or range include instructions which have been removed by **RI**. The `rmw_p` relation is defined similarly, except that it also excludes `rmw` edges whose source load was subject to **RD**.

Note that when **RI**'ing the store sourcing a load, we choose not to explicitly enumerate the set of all possible replacements when forming `rf_p`, as doing so would have introduced a non-trivial performance cost and little actual benefit. Instead, we simply leave the return value of that load unconstrained. Doing so results in relaxed tests which may be slightly underconstrained, but the downside of this is

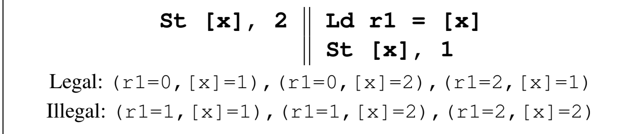



Figure 7: The CoRW litmus test

```
fun co_p[r: RTag->univ] {
  (Write - r[RI]) <: ^co :> (Write - r[RI])
}
```

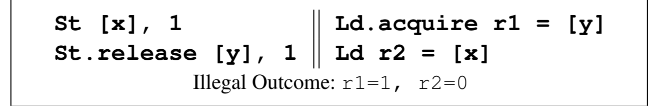
Figure 8: Repairing a perturbed `co_p` relation by taking the transitive closure prior to performing the relaxation

only a slight risk of producing false positives (non-minimal tests which are nevertheless emitted by the generation flow). We considered this worthwhile, as false positives are mostly harmless; they would at worst cause a few cycles to be wasted running a test which is not quite technically minimal (but which may nevertheless be interesting anyway).

To see the effect of leaving perturbed relations unconstrained, consider CoRW (Figure 7). Outcome (`r1=2`, `[x]=2`) is forbidden under any coherent memory model, sometimes even by definition, but it can be made to produce that outcome when applying **RI** to each instruction, and hence it is minimal. Most interestingly, consider what happens when applying **RI** to the store of 2 to `[x]`. If the load either chooses to instead return the value of the initial condition or is simply left unconstrained, then the (`[x]=2`) portion of the outcome becomes valid and the requirement is satisfied. However, suppose instead the load were simply reassigned to source from the `co`-predecessor (i.e., the store of 1 to `[x]`) of the **RI**'ed store, as might be a natural thought. In that case, however, the load would be sourced from a future store in the same thread, a condition which is always illegal. Hence the test would no longer pass the minimality criterion, and it would become a false negative.

To avoid false negatives from appearing as described above, we "repair" perturbed relations only when it is abundantly clear that doing so will not overconstrain the search space. Suppose, for example, `co` were defined to be non-transitive[3]. In that case, **RI**'ing an instruction in the middle of a long chain of `co` edges would leave the events in the first part unrelated to events in the second part. However, `co` is inherently a transitively closed relation when actually used, and so the removal of one event in the middle of a chain should not conceptually affect the rest of the transitive closure. Since there is no risk of introducing false negatives by simply restoring transitivity to perturbed relations in this category, such relations are the only ones we make an extra effort to explicitly "repair".

[3] Although it may seem strange theoretically, it is often convenient: making relations such as `po` and `co` non-transitive removes a significant amount of clutter from graphical representations of the generated tests.



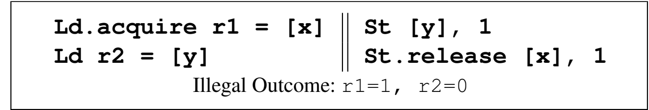




Figure 9: Only one test in each symmetry class needs to be emitted. In these two tests, the addresses and the threads are swapped, but the overall test structure is the same.

## 5. Synthesising Test Suites

Our basic synthesis methodology is straightforward: we simply have Alloy generate all model instances (i.e., all litmus test executions) which satisfy the minimality criterion as applied to some axiom of the model. However, we also take into account the practical concerns described below.

### 5.1 Symmetry Reduction

As shown in previous work, naive enumeration of litmus tests produces a large amount of redundancy. The same is true here: by default, Alloy emits a large number of redundant copies of each test satisfying the minimality criterion. For example, Figure 9 shows two tests that appear different on the surface but which are in fact symmetric and therefore redundant. Alloy does have some built-in symmetry reduction through its use of symmetry-breaking predicates (Torlak and Jackson 2007), but this is insufficient to capture the higher-level notions of symmetry such as those eliminated by Mador-Haim et al. (2010).

We therefore built a post-processor which collects and canonicalizes the tests emitted by Alloy. Our canonicalizer simply augments the approach of Mador-Haim et al. by incorporating new instruction features such as memory order parameters. Threads in a test are hashed and sorted alphabetically, and addresses are reassigned in sorted-sequential order to produce a single canonical form of every test. Only one test producing any given hash is emitted.

### 5.2 Experimental Methodology

To demonstrate the value of our synthesis methodology, we apply it to a number of memory models. We generate one test suite for each axiom of each memory model by applying the instruction relaxations of Table 2, and we record the number of tests emitted and the CPU time it took to generate each suite. We also generate one "union" test suite per model. The union suite includes all tests that satisfy the minimality criterion for at least one axiom in the model[4].

[4] Generating this suite directly with Alloy often took significantly longer than it would have taken to simply generate the results for each axiom separately and then merge the suites at the end. We nevertheless include it as an interesting comparison point.

```
Ld r1 = [x]  || Ld r2 = [x]
St [x], 2    || St [x], 1
```
Illegal Outcome: r1=1,r2=2,[x]=2

Figure 10: n5/CoLB is in "Owens" but not "causality". However, n5/CoLB contains as a subset CoRW (Figure 7), which is in "causality".

If a litmus test satisfies the minimality criterion for more than one axiom, we count it only once in the overall per-model "union" count. Therefore, due to overlap, some per-model "union" test count totals may be less than the sum of the per-axiom test count totals.

In contrast with Mador-Haim et al. (2010), we count all instructions. Specifically, we include fences in the instruction counts, and we count atomic read-modify-write instructions however they are formalized (load-store pairs connected by an `rmw` relation count as two instructions, while atomic RMW primitives count as a one instruction). Therefore, instruction counts for seemingly identical litmus tests might differ slightly from model to model.

We perform our synthesis in Alloy 4 with the MiniSAT backend on a server farm composed of varying Xeon-class processors. We include all suites that could be generated in no more than 16GB of memory and within 48 hours of runtime. The results are shown in the following section.

## 6. Case Studies

### 6.1 Total Store Order (TSO)

The TSO memory model was first introduced for SPARC, but it is best known today as the memory model adopted by x86 processors (Advanced Micro Devices (AMD) 2016; Intel 2016b; SPARC International 1993; Owens et al. 2009). We use the axiomatic formulation described earlier in Figure 4. As a baseline for comparison, we use the suite of x86-TSO litmus tests gathered by Owens et al. (2009). We refer to this as the "Owens" suite for brevity. At the time, x86 had not settled definitively on TSO, and this suite aimed at reconciling differences between competing definitions of the x86 memory model. They collected litmus tests from Intel ISA manuals, AMD ISA manuals, and academic papers, and they added various tests of their own creation. The complete suite contains 24 tests, and 15 specify forbidden outcomes. Our aim is therefore to reproduce (at least) these 15 tests.

Table 4 compares the "causality" suite to the "Owens" suite. At first glance, it appears that some tests in "causality" were not present in "Owens", and vice versa. Further examination, however, shows that every test in "Owens" but not in "causality" contains inside of it a test which is in fact present in "causality". Figure 10 shows an example: test n5/CoLB is in "Owens", but it is not in "causality" because it does not satisfy the minimality criterion for **RI** applied to the load in Thread 0. However, n5/CoLB contains within it test CoRW

| #Insts | "Owens" only | Both | "causality" Only |
|---|---|---|---|
| 2 | — | — | CoWW |
| 3 | — | — | CoRR; CoRW |
| 4 | n5/CoLB (CoRW) | MP; LB | S; 2+2W; CoMP; W+WRR; W+W+RR |
| 5 | iwp2.8.b (MP) | WRC; n6 | WWC; R+f; MP+st; (3 more) |
| 6 | iwp2.6/CoIRIW (W+W+RR); n4 (n5/CoLB) | amd5/SB+mfences; amd6/IRIW; iwp2.8.a; RWC+mfence | (many more) |
| 7 | — | — | (many more) |
| 8 | amd10 (amd5/ SB+mfences); iwp2.7/amd7 (amd6/IRIW) | — — — — | (many more) |
| 9 | n3 (amd6/IRIW) | — | (many more) |

Table 4: Comparing the "Owens" suite to the "causality" suite. Each non-minimal test in "Owens" contains as a subtest a test present in "TSO-union" (shown in parentheses).

```
Ld [x]   | St [x]   | St [x]   | St [x] || St [x]
St [x]   | Ld [x]   | St [x]   |        || Ld [x]
         |          | Ld [x]   |
```

Figure 11: The four tests in "sc_per_loc" but in neither "causality" nor "Owens"

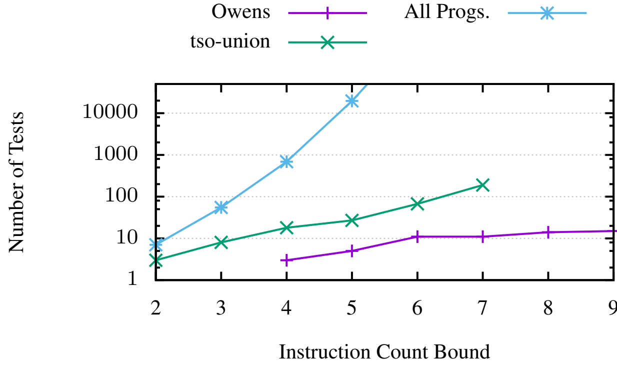
```
St [x] || RMW [x]        St [x] || St [x] || RMW [x]

St [x] || RMW [x]        St [x] || St [x]
St [x] ||                       || RMW [x]
```

Figure 12: The four TSO tests in "rmw_atomicity"

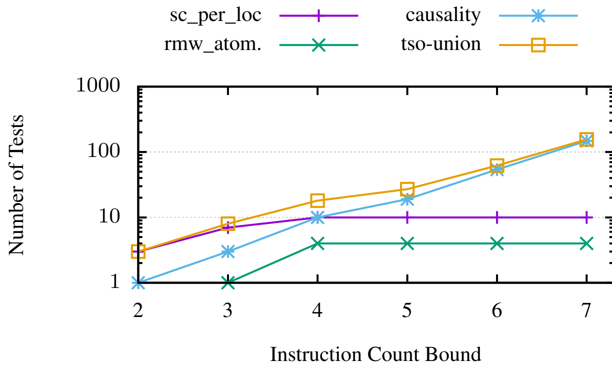
(Figure 7), and CoRW is in fact in "causality". In this sense, "causality" reproduces the entirety of "Owens", while also adding new tests that "Owens" did not include.

In addition to "causality" by itself entirely subsuming "Owens", "sc_per_loc" and "rmw_atomicity" introduce additional tests that neither "causality" nor "Owens" include (Figures 11 and 12). "rmw_atomicity" contains four tests, while "sc_per_loc" contains ten tests, but six overlap with "causality". That "Owens" contains no primitive tests from these two axioms is not entirely unexpected, as "Owens" was instead to distinguish between different possibilities for what we refer to as the `causality` axiom, and as such it simply took the basic behaviors of coherence and read-modify-write atomicity for granted.

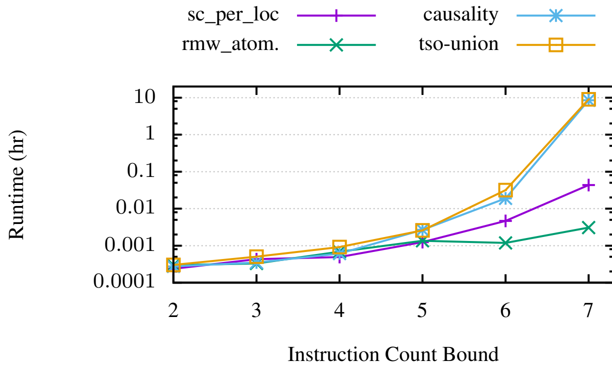
Overall, the results for TSO are summarized in Figure 13. As Figure 13a shows, for each bound, our synthesis tech-




(a) Number of forbidden tests from each source.




(b) Number of tests in the suite generated for each axiom. The total is less than the sum of the parts due to overlap.




(c) Runtime to generate each suite.

Figure 13: TSO memory model results

nique produces an order of magnitude more tests than are in "Owens", while simultaneously remaining tractable as compared to the set of all possible litmus tests. Figure 13b shows how these tests are generated: the "sc_per_loc" and "rmw_atomicity" saturate at ten and four tests, respectively, while the "causality" suite continues to grow without bound. Unfortunately, the runtime is currently super-exponential with the test size bound (Figure 13c); see Section 7 for a discussion. Nevertheless, these results provide strong empirical evidence of the effectiveness of our synthesis technique

```
St [x], 2 || Ld r1 = [x] || Ld r2 = [y]
          || St [y], 1   || St [x], 1
```

Illegal Outcome: `r1=2,r2=1,[x]=2,[y]=1`

Figure 14: Litmus test WWC

```
pred power {
  acyclic[rf + co + fr + po_loc] // SC per Loc.
  acyclic[ppo + fences + rfe]    // No Thin-Air
  irreflex[fre.prop.*(ppo+fences+rfe)]  // Obs.
  acyclic[co + prop]             // Propagation
}
// Write propagation order; mostly due to fences
fun prop : Event->Event { /* see Alglave */ }
// Preserved program order, due to dependencies,
// po_loc, and other subtleties.  Fixed-point of
// four mutually-recursive helper relations.
fun ppo : Event->Event { /* see Alglave */ }
```

Figure 15: Top-level Power memory model axioms (Alglave et al. 2014)

in generating litmus tests for TSO. In addition to reproducing every basic pattern in "Owens", our synthesis technique generates numerous additional tests, including some that are studied in the context of non-TSO models, and others which are less well-known overall.

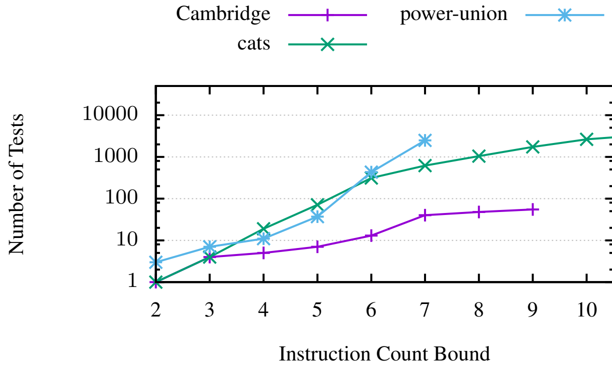
We note one minor redundancy: our symmetry reduction technique (adapted from Mador-Haim et al. (2010)) was unable to detect the symmetry between two different variants of litmus test WWC. As shown in Figure 14, WWC has two threads with identical load-store patterns, and so the canonicalization algorithm could not detect the symmetry between the two variants with the first two threads swapped. We plan to enhance our canonicalizer to address this, but even without it, the only cost is a small amount of extra redundancy in the test suite.

### 6.2 Power (and ARMv7)

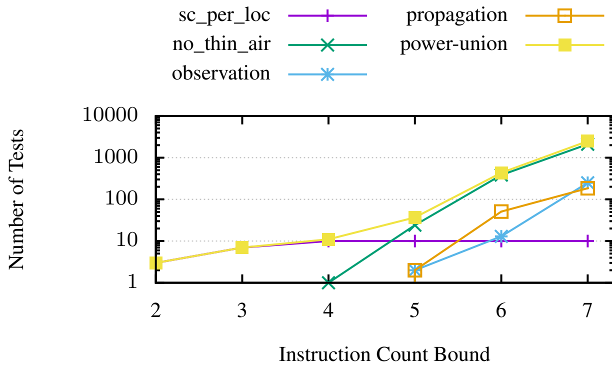
The ARM and Power memory models together make for another interesting case study (ARM Holdings 2016; Power.org 2013). Both are significantly more complex than TSO, and yet both have been studied heavily in previous work. This makes them good baseline models for comparison.

For our study, we use the axiomatic formulation of the Power memory model from Alglave et al. (2014). The basic rules are shown in Figure 15. ARMv7 is broadly similar to Power, but differs in some of the details (e.g., ARM has no equivalent of the Power `lwsync` lightweight fence). However, according to Alglave et al., some subtleties of the ARMv7 model remain uncertain. ARMv8 adds explicit load-acquire and store-release opcodes (ARM Holdings 2016), but to our knowledge ARMv8 has not been formalized axiomatically.

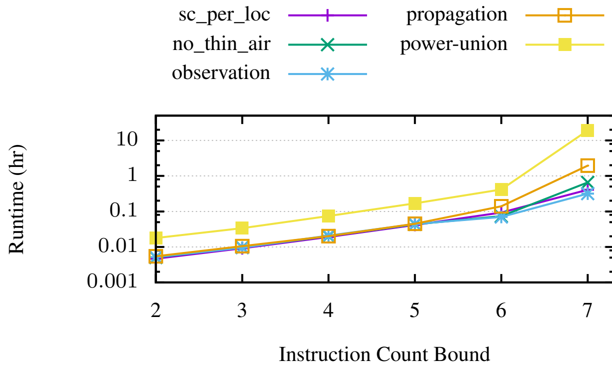
We synthesized the "power-union" suite consisting of tests which satisfy the minimality criterion for at least one




(a) Number of forbidden tests from each source




(b) Number of tests in the suite generated for each axiom.




(c) Runtime to generate each suite.

Figure 16: Power memory model results

of the four axioms. We then compare "power-union" against two baseline suites: the "Cambridge" 55-test summary of the Power and ARM memory models (Sarkar et al. 2011a,b), and the "cats" suite of over eight thousand tests studied by Alglave et al. as part of their formalization process (Alglave et al. 2014, 2015b). Just as with "Owens" suite for TSO, these suites consist of hand-written tests, tests extracted from industry manuals, and randomly-generated variants thereof (Alglave et al. 2010).

Figure 16 summarizes the results. As with our comparison to TSO, our synthesis technique reproduces all of the tests in "Cambridge", with one exception: test PPOAA is presented using a full `sync` fence, but in reality it only requires a lightweight `lwsync` fence, and hence it is not minimal as presented in "Cambridge". Fortunately, the `lwsync` variant of PPOAA is included in "power-union".

As shown in Figure 16a, there is no clear trend in relative test counts between "power-union" and "cats". Figure 16b shows that "power-union" contains a large number of "no_thin_air" tests due largely to the variety of ways different dependency types can be chained together to form `ppo`. "cats" contains a large number of randomized tests exploring a similar space. We did not directly analyze all tests in "cats" suite due to their being written in the incompatible `.litmus` format. However, after manual empirical analysis, we again find some tests in "cats" which do not satisfy the minimality criterion, and we find other tests in "power-union" which are not present in "cats", including again the first three tests of Figure 11. We did verify that our synthesis technique was able to reproduce some particularly interesting tests such as lb+addrs+ww, which identifies the difference in strength between address and data dependencies present in this formalization.

Figure 16c shows that Alglave's formalization of the Power memory model was taxing on our infrastructure. The runtime is still super-exponential with respect to the test size bound, but the constant factor is also much larger than it was for TSO. We identify two major causes of this. First, the presence of three separate types of dependency (address, control, data) with subtly different semantics means that each basic test shape has a huge number of subtle dependency variants that must be considered independently. Second, the axiomatic formulation that we use calculates preserved program order (`ppo`) by finding the fixed point of a set of four mutually-recursive definitions. This adds substantial computational complexity to the process. As always, there is a tradeoff between having simpler, easier-to-analyze models and having models which squeeze out every ounce of the deliverable performance from a given (micro)architecture.

In the next case study, we compare these results to those for a streamlined version of the ARM and Power models.

### 6.3 Streamlined Causal Consistency (SCC)

The Streamlined Causal Consistency (SCC) model is a CPU-like memory model aimed at simplifying away the complexities and corner cases of the ARM and Power memory model while preserving similar relaxed behaviors. We introduce SCC for two purposes: to compare the suite generation runtime of simple vs. complex models, and to analyze additional hardware memory model features not already covered by TSO and Power. The core of the model is presented in Figure 17. SCC adds explicit Acquire and Release instructions, similar to ARMv8 (ARM Holdings 2016), but it eliminates dependencies and the complex `ppo` entirely. It also defines the `sc` relation to be a total order over all heavyweight

```
sig Acquire extends Read {}
sig Release extends Write {}
sig FenceAcqRel extends Fence {}
sig FenceSC extends Fence { sc: set FenceSC }
fact { total[sc, FenceSC] } // FenceSC total order
pred scc {
  acyclic[rf + co + fr + po_loc]    // SC per Loc.
  acyclic[rf + dep]             // No Thin-Air values
  no fr.co & rmw                     // RMW Atomicity
  irreflexive[*(rf + co + fr).^cause]// Causality
}
fun prefix : Event->Event {
  iden + (Fence <: po) + (Release <: po_loc)
}
fun suffix : Event->Event {
  iden + (po :> Fence) + (po_loc :> Acquire)
}
fun sync : Event->Event {
  Releasers <: prefix.^(rf+rmw).suffix :> Acquirers
}
fun cause : Event->Event { *po.(sc + sync).*po }
```

Figure 17: Top-level SCC Axioms

| | |
|---|---|
| `St [x], 1` | `St [y], 1` |
| `FenceSC` | `FenceSC` |
| `Ld r1 = [y]` | `Ld r2 = [x]` |

Legal Outcomes: `(r1=1,r2=1)`, `(r1=1,r2=0)`, `(r1=0,r2=1)`
Illegal Outcomes: `(r1=0,r2=0)`

(a) The store buffering (SB) litmus test

| | |
|---|---|
| `St [x], 1` | ~~`St [y], 1`~~ |
| `FenceSC` | `FenceSC` |
| ~~`Ld r1 = [y]`~~ | `Ld r2 = [x]` |

Legal Outcomes: `(r1=1,r2=1)`, `(r1=1,r2=0)`, `(r1=0,r2=1)`
Illegal Outcomes: `(r1=0,r2=0)`

(b) If `sc` places the Thread 0 fence before the Thread 1 fence, then even if the accesses to [y] are removed, `(r1=0,r2=0)` remains illegal.

Figure 18: If `sc` is chosen before applying **RI**, test fails the statement of the minimality criterion from Figure 5c.

`FenceSC` instructions (International Organization for Standardization (ISO) 2011a,b; Mador-Haim et al. 2012).

The inclusion of `sc` required us to make one modification to our basic modeling approach. As we noted in Section 4.3, there are cases in which our practical formalization of the minimality criterion is an under-approximation of the proper definition. Unfortunately, `sc` falls exactly into this category.

Consider the "store buffering" (SB) litmus test of Figure 18a. This test satisfies the minimality criterion: the outcome `(r1=0,r2=0)` is illegal, but applying any instruction weakening to the test causes it to become observable. However, according to the code of Figure 5c, the `sc` edge is implicitly chosen before applying the instruction relaxation. Unfortunately, when the operations are performed in

```
fact { lone sc } // zero or one sc edges
pred causality_wa {
  irreflexive[(rf + co + fr).^cause] or
  irreflexive[(rf + co + fr).^cause_wa]
}
fun cause_wa : Event->Event {
  (*po.(~sc + sync).*po)
}
```

Figure 19: Augmenting SCC to work around (wa) our infrastructure limitations

that order, one store and one load can in fact be removed by **RI** without causing the outcome to become observable (Figure 18b). In this case, SB would be a false negative of the kind described in Section 4.3.

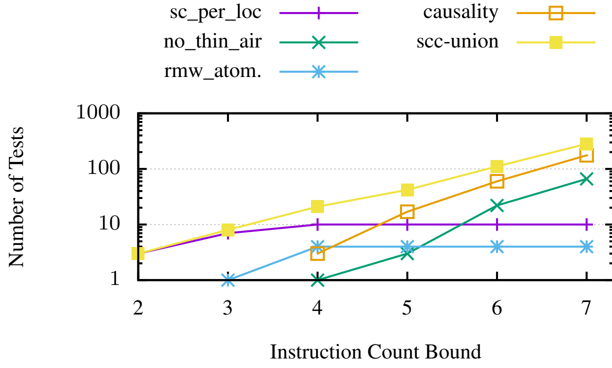
To work around this limitation, we use the code shown in Figure 19. If there are zero or one `sc` edges, then we can manually emulate the enumeration over all `sc` edges by simply considering both the relation and its reversal. A test with three non-trivial FenceSC instructions would need at least 8 instructions overall since a fence at the start or end of a thread is irrelevant. As our results currently do not scale past eight instructions anyway, this workaround was sufficient for our current study. A general long-term solution would require a technique such as the one discussed in Section 4.3.

The SCC results generated using the workaround above are shown in Figure 20. Figure 20a shows the number of tests generated. "sc_per_location" and "rmw_atomicity" saturate as they did before, while the tests generated from the other axioms continue to grow with the test size bound. Notably, most per-axiom numbers are larger, since SCC provides more ways to synchronize (e.g., acquire/release vs. fences) than did the previous models. However, with only one type of depenedency, there are much fewer "no_thin_air" axioms than there were for Power. The suite also includes tests such as SB which require `FenceSC` instructions and `sc` edges, confirming that the approach of Figure 19 worked as intended. Finally, Figure 20b shows a similar super-exponential runtime trend to the previous models.

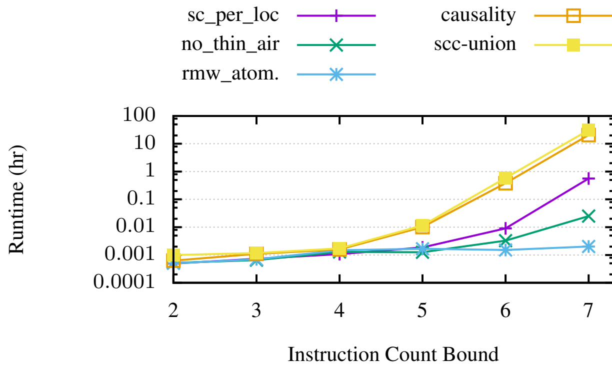
### 6.4 C and C++

Lastly, we analyze the C11/C++11 memory model (International Organization for Standardization (ISO) 2011a,b). Although this model was finalized six years ago, it still contains various surprises and unresolved corner cases (Batty et al. 2016; Manerkar et al. 2016; Vafeiadis et al. 2015). It also serves as a good case study of how synthesizing test suites for software memory models differs from the process of generating suites for hardware models. We use the formulation of Batty et al. (2016) for our experiments (Figure 21), but we eliminate initialization events in order to scale more easily to larger tests.

A new wrinkle is added by the fact that C11/C++11 (like many other software models) gives no semantics at all to




(a) Number of tests in the suite generated for each axiom.



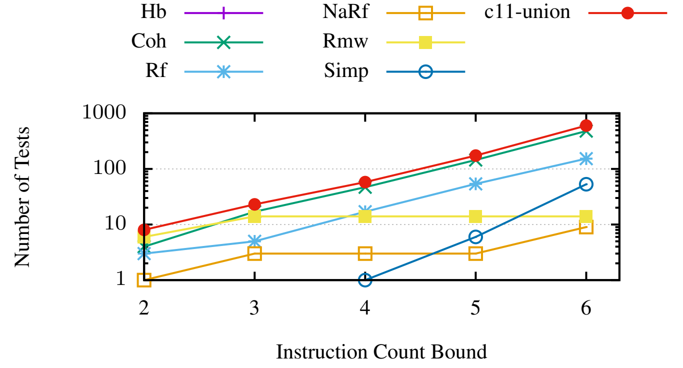

(b) Runtime to generate each suite.

Figure 20: SCC memory model results

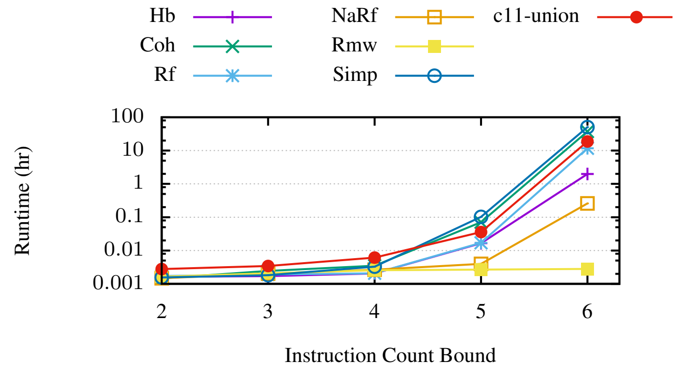
```
pred c11 {
  irreflex[hb]                          // Hb
  irreflex[(?(~rf)).mo.(?rf).hb]        // Coh
  irreflex[rf.hb]                       // Rf
  no (rf :> NonAtomicEvent) - vis       // NaRf
  irreflex[rf + (mo.mo.~rf) + (mo.rf)] // Rmw
  acyclic[(SCEvent->SCEvent - iden) &   // Simp
    (?Fsb.(hb+fr+mo).?sbF)]
}
fun hb : Event->Event { sb + sw }
// sequenced-before: analogous to program order
// synchronizes-with: loosely, release->acquire
//     synchronization.  See Batty for details.
```

Figure 21: Top-level C11 memory model axioms (Batty et al. 2016). As a convenient shorthand, the figure uses a “zero or one” operator “?” even though Alloy has no such operator.

a program with the possibility of a data race (Adve and Hill 1990; Boehm and Adve 2008; International Organization for Standardization (ISO) 2011a,b). This raises an interesting question: if application of an instruction relaxation to a properly-synchronized race-free program results in a racy execution, should such tests be considered valid new outcomes with respect to the minimality criterion? The distinction is non-trivial; just as in Section 4.3, more tests are produced if racy outcomes are included than if they are excluded. However, in this case, the potentially racy relaxed tests represent buggy executions rather than malformed programs, and so we do not consider them false positives. In any case, we would rather err on the side of producing more tests rather than fewer.




(a) Number of tests in the suite generated for each axiom. The total is less than the sum of the parts due to overlap. No tests were generated for “Hb”.




(b) Runtime to generate each suite.

Figure 22: C11/C++11 memory model results

Also worth noting is that C11/C++11 (as well as Java) currently have some counterintuitive situations in which increasing the synchronization can cause *more* outcomes to become visible (Ševčík and Aspinall 2008; Vafeiadis et al. 2015). While one could in theory consider instruction relaxations which increase the strength of the synchronization, we do not consider any in this paper, as this non-monotonicity is generally not intentional anyway.

The C11 results are shown in Figure 22. Of immediate note from Figure 22a is the fact that no interesting tests are generated for the “Hb” axiom. This is because applying **DMO** to an instruction with an `hb` relation demotes the `hb` into `*hb.rf.*hb`, and the original `irreflexive[hb]` statement becomes `irreflexive[(*hb.rf.*hb).hb] = irreflexive[rf.hb]`, where the latter is exactly the “Rf” axiom. Hence **DMO** cannot produce a new outcome which satisfies “Rf”, and no “Hb” test can satisfy the

(implicit initial condition: `[x]=0`)

```
rmw.add [x], 1 || rmw.add [x], 1
```

Illegal Outcome: `[x]=1`

(a) In models which expose each atomic RMW event as a single instruction, this test is minimal

(implicit initial condition: `[x]=0`)

| Thread 0 | Thread 1 |
|---|---|
| ~~`ld.rmw r1, [x]`~~ | `ld.rmw r2, [x]` |
| ~~`add r1, r1, 1`~~ | `add r2, r2, 1` |
| `st.`~~`rmw`~~` [x], 1` | `st.rmw [x], r2` |

Illegal Outcome: `[x]=1`

(b) In models where atomic RMW operations are exposed as paired loads and stores, the same test is no longer minimal: applying **RI** to the first load produces still-illegal CoRW

Figure 23: The choice of basic primitives can affect whether a test is considered minimal

minimality criterion. Additionally, "Hb", "Coh", "Rf", and "Rmw" can be collectively rewritten in the simpler form `acyclic[*com.hb]` (Vafeiadis et al. 2015). We confirmed empirically that the suite generated for this combined axiom produces the same set of litmus tests as the combined suites for the original four[5]. This shows that the emptiness of the "Hb" suite is merely a property of the formalization than a shortcoming of our synthesis technique; all minimal tests are present in the final overall suite. Finally, Figure 22b shows that C11 took the longest to analyze of the models we considered in this paper.

Our C11 results demonstrate that our synthesis technique works well on both software and hardware models. We hope to extend our method to more diverse and more sophisticated models (e.g., OpenCL) in the future.

## 7. Discussion

There are various ways in which our synthesis method could be extended in the future. For example, the litmus test suites of Section 6.2 both consider test families such as PPO[000-999] in which there is a fixed basic pattern (here, MP), but in which one or more of the relations in the test are randomized (here, the load-load ordering). It would be straightforward to extend our techniques to generate minimized instances of particular relations to reproduce such families, and more broadly to take advantage of combining randomization techniques with ours.

There may also be tests which are not strictly minimal, but which nevertheless may be interesting to include in a test suite. For example, the test of Figure 23 is not minimal in models where atomic read-modify-writes are exposed as two paired instructions (such as the model in Figure 4): the load half of one of the atomics can be removed without allowing any new outcomes. This is analogous to n5/CoLB being excluded in Section 6.1 due to its containing CoRW as a subcomponent. However, even in such models, the test of Figure 23 is perfectly reasonable to include, as atomics may behave differently from non-atomics in practice. It would be interesting to classify all tests by counting how many instruction relaxations it would take to make the test produce a new outcome, and then to produce test suites of varying "degrees of minimality".

Alternatively, it would also be useful to extend a suite of minimal tests by feeding it into a flow that randomly strengthens or perturbs those tests. For example, the test of Figure 23 could be re-derived as a random and/or guided strengthening of CoRW. This approach would tie in nicely with a stressing or fuzzing flow of the kind that has been shown to encourage weak memory behaviors to appear in practice (Sorensen and Donaldson 2016).

Of course, one practical problem to solve is the fact that runtime is currently super-exponential with respect to the test size bound. Fortunately, generation of a litmus test suite is a one-time cost for any given model. We also believe that a more sophisticated use of symmetry reduction within the SAT solver itself could provide significant benefits. The risk would be that doing so would come at the expense of generality or usability, which through Alloy are what made our exploration possible in the first place. We hope to further explore this tradeoff as future work as well.

## 8. Conclusion

Memory model verification is becoming increasingly important as hardware and software models continue to evolve in increasingly complex directions. Unfortunately, existing testing infrastructures are prone to human error and incomplete coverage. To mitigate this, we presented a new methodology for automatically synthesizing comprehensive litmus test suites directly from axiomatic memory model formalizations. Our synthesis technique was able not only to reproduce but also to fill in gaps in existing test suites for various important real-world memory models, including x86-TSO, IBM Power, and C11/C++11. We also showed how easily our technique was able to adopt to the SCC memory model newly proposed in this paper. Finally, our results reinforce the need for and benefits of formalism when defining and analyzing memory models. To aid in this effort, our models are open-sourced and freely available at the following URL: `http://github.com/nvlabs/litmustestgen`


## Acknowledgments

This research was developed, in part, with funding from the United States Department of Energy. The views, opinions, and/or findings contained in this article are those of the authors and should not be interpreted as representing the official views or policies of the Department of Energy or the U.S. Government.


[5] Batty et al. intentionally focused on producing axioms corresponding to text in the standard, rather than on finding the most compact representations.

## References


Advanced Micro Devices (AMD). AMD64 architecture programmer's manual. Technical report, 2016. URL: `http://developer.amd.com/resources/developer-guides-manuals`.

S. V. Adve and M. D. Hill. Weak ordering—a new definition. In *17th Annual International Symposium on Computer Architecture (ISCA)*, 1990.

J. Alglave and L. Maranget. Towards a formalization of the HSA memory model in the cat language. Technical report, 2016. HSA Foundation Specification Version 1.1. URL: `http://www.hsafoundation.com/?ddownload=5381`.

J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *22nd International Conference on Computer Aided Verification (CAV)*, 2010.

J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transanctions on Programming Languages and Systems (TOPLAS)*, 36(2):7:1–7:74, July 2014. ISSN 0164-0925.

J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In *20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015a.

J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory, companion material, Power litmus tests. 2015b. URL: `http://diy.inria.fr/cats/showlogs/power-tests.tgz`.

AMD. Revision guide for AMD family 10h processors. Technical report, 2012. Bug 254. URL: `http://support.amd.com/TechDocs/41322_10h_Rev_Gd.pdf`.

ARM. Cortex-A9 MPCore™, programmer advice notice, read-after-read hazards. Technical report, 2011. URL: `http://infocenter.arm.com/help/topic/com.arm.doc.uan0004a/UAN0004A_a9_read_read.pdf`.

ARM Holdings. ARM architecture reference manuals. Technical report, 2016. URL: `http://infocenter.arm.com/help/topic/com.arm.doc.set.architecture`.

M. Batty, A. F. Donaldson, and J. Wickerson. Overhauling sc atomics in c11 and opencl. In *43rd Annual Symposium on Principles of Programming Languages (POPL)*, 2016.

H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *29th Annual Conference on Programming Language Design and Implementation (PLDI)*, 2008.

H.-J. Boehm and B. Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *Workshop on Memory Systems Performance and Correctness (MSPC)*, 2014.

S. Hangal, D. Vahia, C. Manovit, and J.-Y. J. Lu. TSOtool: A program for verifying memory systems using the memory consistency model. In *31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

Intel. A formal specification of Intel Itanium processor family memory ordering. Technical report, 2002. URL: `ftp://download.intel.com/design/Itanium/Downloads/25142901.pdf`.

Intel. Intel Xeon processor E5 v3 product family, processor specification update. Technical report, 2016a. Bug HSE44. URL: `http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e5-v3-spec-update.pdf`.

Intel. Intel®64 and IA-32 architectures software developer manuals. Technical report, 2016b. URL: `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

International Organization for Standardization (ISO). Information technology – programming languages – C, ISO/IEC 9899:2011. Technical report, Dec. 2011a.

International Organization for Standardization (ISO). Information technology – programming languages – C++, ISO/IEC 14882:2011. Technical report, Sept. 2011b.

D. Jackson. Alloy: A lightweight object modelling notation. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*, volume 11, Apr. 2002. URL: `http://alloy.mit.edu`.

Khronos Group. The OpenCL specification, version 2.1. Technical report, 2015. URL: `https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf`.

L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28, 1979.

D. Lustig, G. Sethi, M. Martonosi, and A. Bhattacharjee. COATCheck: Verifying memory ordering at the hardware-OS interface. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

S. Mador-Haim, R. Alur, and M. M. K. Martin. Generating litmus tests for contrasting memory consistency models. In *22nd International Conference on Computer Aided Verification (CAV)*, 2010.

S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams. An axiomatic memory model for power multiprocessors. In *24th International Conference on Computer Aided Verification (CAV)*, 2012.

Y. A. Manerkar, C. Trippel, D. Lustig, M. Pellauer, and M. Martonosi. Counterexamples and proof loophole for the C/C++ to POWER and ARMv7 trailing-sync compiler mappings. *arXiv*, 1611.01507v2, Nov 2016.

C. Manovit, S. Hangal, H. Chafi, A. McDonald, C. Kozyrakis, and K. Olukotun. Testing implementations of transactional memory. In *15th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2006.

A. Milicevic, J. P. Near, E. Kang, and D. Jackson. Alloy*: A higher-order relational constraint solver. In *37th International Conference on Software Engineering (ICSE)*, 2015.

S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2009.

Power.org. Power ISA™ version 2.07. Technical report, 2013. URL: https://www.power.org/wp-content/uploads/2013/05/PowerISA_V2.07_PUBLIC.pdf.

S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors, companion material, POWER and ARM litmus tests. Technical report, 2011a. URL: https://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test6.pdf.

S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *32nd Conference on Programming Language Design and Implementation (PLDI)*, 2011b.

J. Ševčík and D. Aspinall. On validity of program transformations in the java memory model. In *22nd European Conference on Object-Oriented Programming (ECOOP)*, 2008.

T. Sorensen and A. F. Donaldson. Exposing errors related to weak memory in GPU applications. In *37th Conference on Programming Language Design and Implementation (PLDI)*, 2016.

SPARC International. The SPARC architecture manual, version 9. Technical report, 1993.

E. Torlak and D. Jackson. Kodkod: A relational model finder. In *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2007.

V. Vafeiadis, T. Balabonski, S. Chakraborty, R. Morisset, and F. Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *42nd Symposium on Principles of Programming Languages (POPL)*, 2015.

J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides. Automatically comparing memory consistency models. *44th Symposium on Principles of Programming Languages (POPL)*, 2017.