Synthesizing Memory Models from Litmus Tests

Technical Report UW-CSE-16-10-01

James Bornholt Emina Torlak

University of Washington

{bornholt, emina}@cs.washington.edu

Abstract

Reasoning about concurrent code requires a memory consistency model that specifies which writes to shared memory a given read may see. Ambiguities or errors in these specifications can lead to bugs in both compilers and applications. Yet architectures usually define their memory models with prose and *litmus tests*—small concurrent programs that demonstrate allowed and forbidden outcomes. Recent work has formalized the memory models of common architectures, but each formalization required substantial effort and several revisions. As new architectures emerge, there is a growing need for tools that can aid these efforts.

This paper presents MemSynth, the first system for automatic synthesis of axiomatic memory model specifications from litmus tests. MemSynth provides an expressive language for specifying the axioms that define a class of memory models, sketching a memory model within that class, and writing litmus tests. Its synthesis engine takes the axioms, sketch, and tests as input, and searches for a model (if any) that gives the desired outcomes on all tests. The MemSynth engine is based on a novel embedding of bounded relational logic in a solver-aided programming language, which enables it to tackle complex synthesis queries intractable to existing relational solvers. Thanks to this design, MemSynth can also solve new kinds of queries, such as a uniqueness check to discover ambiguities in memory model specifications.

We show that MemSynth can synthesize specifications for Intel's x86 in under a second, and for the PowerPC architecture in 16 seconds from 768 litmus tests. Our uniqueness check demonstrates several ambiguities in the documentation for the x86 memory model. We also use MemSynth to reproduce, debug, and automatically repair a prior paper on comparing memory models—all in just 2 days.

1. Introduction

Reasoning about concurrent code requires a *memory consistency model* that specifies the memory reordering behaviors the hardware will expose. Architectures typically define their memory consistency model with prose and *litmus tests*, small programs that illustrate allowed and forbidden outcomes. These ambiguous definitions make reasoning about correctness difficult for both developers and tool builders. Researchers have therefore argued for formalizing memory models [43], and have recently formalized models for common architectures, including x86 [34] and PowerPC [26]. But each such formalization required several person-years of effort and several revisions (e.g., [4, 5, 29, 32, 33]).

This paper addresses the problem of synthesizing specifications of memory models from litmus tests. The core challenge we tackle is one of language design and implementation—the target specification language must be rich enough to express practical memory models and their litmus tests, while supporting efficient automation. Prior work [2, 5, 25, 42] has developed many axiomatic frameworks for reasoning about memory models. These frameworks supply basic axioms that every memory model in the framework must follow, expressed as first-order constraints on relations that order memory events (such as reads and writes). But different frameworks capture different classes of memory models, and they evolve over time [2, 5, 7], so no single framework is itself a future-proof target for synthesis.

We address this challenge with MemSynth, a new metaframework for synthesizing axiomatic specifications of memory models from litmus tests. MemSynth provides a language for implementing memory model frameworks, and an efficient engine for synthesizing models in those frameworks.

The language and the engine are both based on a deep embedding of bounded relational logic [22, 38] in Rosette [36, 37], a solver-aided host language that extends Racket [19, 31] with support for verification and synthesis. Relational logic combines first-order logic with relational algebra and transitive closure, providing an expressive semantics that subsumes many recent frameworks (e.g., [5, 25, 39, 42]). The bounded version of the logic is decidable by reduction to boolean satisfiability, and existing relational solvers [22, 28, 38] are based on such a reduction. MemSynth takes a radically simpler approach—it delegates the reduction to its host language. Rosette includes a symbolic evaluator that can compile the semantics of its guest languages to efficiently-solvable SMT constraints. The MemSynth engine layers a domain-specific synthesis algorithm on top of this evaluator, scaling to produce specifications of real memory models in seconds.

The MemSynth synthesizer takes as input a sketch [35] of a memory model expressed in some framework, along with a set of litmus tests. The sketch is a formula in relational logic, with missing expressions (called *holes*) over relations defined by the framework. The holes define a finite space of candidate specifications, which the synthesizer searches for a memory model that gives the desired outcome on all provided tests. This search involves solving a synthesis query of the form $\exists M.\ \bigwedge_{T\in\mathcal{T}_P}(\exists E.\,\mathrm{Allow}(M,T,E)) \wedge \bigwedge_{T\in\mathcal{T}_N}(\forall E.\ \neg\,\mathrm{Allow}(M,T,E))$, where M is a memory model specification, E consists of relations that encode litmus test executions, and $\mathcal{T}_P$ and $\mathcal{T}_N$ contain litmus tests that demonstrate allowed and forbidden behaviors, respectively. In principle, such a query can be discharged by relational solvers [28] that support higher-order quantification (over the relations E). In practice, however, our queries are intractable for these solvers: their languages lack the constructs (such as sketches and partial interpretations [38]) that enable MemSynth's embedded engine to aggressively exploit domain-specific knowledge, extracted from litmus tests and memory model frameworks, for search space reduction and symmetry breaking.

But MemSynth's novel design offers advantages that go beyond scalable synthesis. Being embedded in Rosette, MemSynth is a full-featured platform for rapid development of high-performance tools for reasoning about memory models. For example, we implement the classic verification query [30], which determines if a litmus test is allowed or forbidden by a memory model, in five lines of code. The resulting implementation outperforms dedicated relational solvers [22, 28] on these queries, and is comparable to hand-crafted verifiers for the frameworks we embed in MemSynth [5, 25]. We also implement a novel *uniqueness query* for refining synthesized memory model specifications by identifying ambiguities in the set of litmus tests used during synthesis. The uniqueness query checks whether a memory model uniquely explains a set of litmus tests, and if not, synthesizes another model along with a *distinguishing test* that illustrates the difference between the two models.

We evaluate the scalability and utility of MemSynth's queries by instantiating it with an expressive framework developed by Alglave et al. [5]. With this instantiation, MemSynth synthesizes a specification for the notoriously relaxed PowerPC architecture from 768 litmus tests in under 16 seconds. including definitions for the subtle cumulativity behavior of PowerPC fences. We also synthesize a specification for the total store ordering (TSO) memory model used by the x86 architecture in under two seconds, using the litmus tests from the Intel Software Developer's Manual [21]. These tests do not uniquely define TSO, however: our uniqueness query finds a second model and a distinguishing test that demonstrates the ambiguity.

We evaluate MemSynth as a tool-building platform by reproducing results from an existing paper [25] on comparing memory models. In the process, we automatically synthesize a repair for a discrepancy between our implementation and the original paper—due to a misprint in the paper—which we were unable to fix by hand. The repaired instantiation of the paper's framework was developed in 2 days by one of the authors and achieves the same performance as the existing tool.

In summary, this paper makes the following contributions:

- We introduce MemSynth, a meta-framework for automatically synthesizing memory model specifications from litmus tests. MemSynth's novel design, as an embedded logic in a solver-aided host language, enables it to scale to complex memory models such as PowerPC, and to synthesize models from large sets of examples.
- We demonstrate that MemSynth can be used to answer advanced queries about memory model specifications, such as uniqueness, that can aid specification writers and architecture designers in refining their models. To our knowledge, MemSynth is the first tool to provide this form of analysis for memory model designs.
- We show MemSynth's utility for rapid development of automated memory model frameworks by constructing several tools that outperform existing counterparts.

The remainder of this paper is organized as follows. Sec. 2 introduces the MemSynth language for memory models and litmus tests. Sec. 3 presents the queries that MemSynth can answer, and Sec. 4 describes the algorithms to answer these queries. Sec. 5 shows three case studies using MemSynth, including synthesizing and refining a specification of PowerPC and identifying ambiguities in x86 documentation. Sec. 6 describes related work, and Sec. 7 concludes.

2. MemSynth Language

MemSynth is a language and an engine for automated reasoning about memory models. The language extends bounded relational logic [22, 38] with generic support for sketching [35] and parametric support [2, 42] for specifying litmus tests and memory models. Thanks to its parameterized design and expressive underlying logic, MemSynth can host many existing frameworks for reasoning about specific classes of memory models. This section reviews the syntax and semantics of relational logic; presents our extensions for expressing sketches, litmus tests, and memory models; and describes $\mathrm{MemSynth}_A$, an embedding of the Alglave et al. [5] framework for memory models. We use $\mathrm{MemSynth}_A$ to illustrate the automated reasoning queries (Sec. 3) supported by our engine (Sec. 4), and to demonstrate their scalability (Sec. 5).

2.1 Bounded Relational Logic with Sketches

Relational logic [22] extends classic first-order logic with transitive closure and relational algebra. The inclusion of closure and relations makes this logic ideally suited for reasoning about memory models. In fact, many recent axiomatic frameworks for memory models (e.g., [5, 25, 39, 42]) are expressed as first-order constraints on relations that order memory

events. MemSynth is based on a new embedding of bounded relational logic [38] in the Rosette solver-aided language [36, 37], which extends Racket [19] with support for verification and synthesis. This embedding includes an explicit construct for sketching, and its engine is optimized for answering (satisfiability) queries about memory models orders of magnitude faster than general-purpose relational solvers [22, 28].

Bounded Relational Logic. Bounded relational logic (Fig. 1) includes the standard connectives and quantifiers of first-order logic, along with the standard operators of relational algebra. A *specification* $(\!|U; D; f|\!)$ in this logic consists of a *universe* of discourse U, a set of *relation declarations* D, and a *formula* f. The universe U is a finite, non-empty set of uninterpreted symbols. A relation declaration $r :_k [R_l, R_u]$ introduces a free variable r (in essence, a Skolem constant), which denotes a *relation* of arity k. Each tuple in this relation consists of k elements drawn from the universe U. The relations R_l and R_u are called the *lower* and *upper* bound on r, and specify the tuples that r must and may contain, respectively. The formula f may refer to the variables r declared in D, but it may not include any other free (unquantified) variables.

We define the meaning of a relational specification $s = (\!|U; D; f|\!)$ with respect to an *interpretation* as follows. An interpretation I consists of a universe $U(I)$ and a map of variables to relations drawn from $U(I)$. We say that I satisfies the specification s, written as $I \models s$, if I and s have the same universe of discourse (i.e., $U(I) = U$), if $R_l \subseteq I(r) \subseteq R_u$ for each $r :_k [R_l, R_u]$ in D, and if the formula f evaluates to 'true' in the environment defined by I, i.e., $[\![f]\!]I = \top$.

The semantics of formulas and expressions are standard [38], but we review the most relevant constructs next. The constant univ denotes the universal relation $\{\langle a\rangle \mid a \in U\}$, and iden is the identity relation $\{\langle a,a\rangle \mid a \in U\}$. The multiplicity predicates no, some, and one constrain their argument to contain zero, at least one, and exactly one tuple, respectively. The cross product $X \to Y$ of two relations is the Cartesian product of their tuples. The join $X.Y$ of two relations is the pairwise join of their tuples, omitting the last column of X and first column of Y, on which the two relations are matched. As we will see in Sec. 2.3, memory model specifications make heavy use of these constructs.

Example 1. Let the universe be $U = \{a,b,c,d\}$, $X = \{\langle a\rangle, \langle c\rangle\}$ a relation of arity 1 with two tuples, and $Y = \{\langle a,b\rangle, \langle b,d\rangle\}$ a relation of arity 2 with two tuples. We can take the cross product, join, and transitive closure of these relations as follows: $X \to Y = \{\langle a,a,b\rangle, \langle a,b,d\rangle, \langle c,a,b\rangle, \langle c,b,d\rangle\}$, $X.Y = \{\langle b\rangle\}$, $Y.Y = \{\langle a,d\rangle\}$, and $\hat{}Y = \{\langle a,b\rangle, \langle b,d\rangle, \langle a,d\rangle\}$. If we provide the declarations $p :_1 [\{\}, \{\langle a\rangle, \langle c\rangle, \langle d\rangle\}]$ and $q :_2 [\{\langle a,b\rangle\}, \{\langle a,b\rangle, \langle b,d\rangle\}]$, then the interpretation $I = \{p \mapsto X, q \mapsto Y\}$ satisfies the specification $(\!|U; p, q; \text{no } q.p|\!)$ but it does not satisfy $(\!|U; p, q; q.q \text{ in } q|\!)$.

Expression Sketches. To support synthesis, we extend relational logic with *expression sketches*, which define the search space for a synthesis query to explore [35]. An expression sketch $\mathcal{G}(N,T,d,k)$ is a finite set of expressions in relational logic, each of which evaluates to a relation of arity k. The set contains all expressions of arity k that can be produced by up to d applications of the production rules of a context-free grammar with non-terminals N and terminals T, where the non-terminals are drawn from expression operators in relational logic. Expression sketches are a key difference between MemSynth and other relational logic languages such as Kodkod [38] and Alloy* [28], which require another layer of embedding—building an interpreter for relational logic inside relational logic—to achieve the same result.

Example 2. Let X be a relation of arity 1, Y a relation of arity 2, $T = \{X, Y\}$, and $N = \{+, \to\}$. Then $\mathcal{G}(N,T,1,1)$ contains only the expressions X and $X + X$, $\mathcal{G}(N,T,2,1)$ contains X, $X + X$, and $X + X + X$, and $\mathcal{G}(N,T,1,2)$ contains Y, $Y + Y$, and $X \to X$.

Relational DSL. MemSynth is implemented (Fig. 2) as a domain-specific language (DSL) in Rosette [36, 37]. The MemSynth interpreter $\text{INTERPRET}(p,I)$ takes as input relational syntax p and an interpretation I, and executes the semantics in Fig. 1. The interpreter represents relations of arity k in the standard way [22, 38], as boolean matrices of size $|U|^k$, with each cell denoting the presence or absence of a given k-tuple. Relational expressions are then interpreted as matrix operations and formulas as constraints over matrix entries; e.g., relational join becomes matrix multiplication.

Being embedded in Rosette [36, 37], MemSynth is both an interpreter for bounded relational logic and an engine for answering *relational satisfiability queries*—such as finding an interpretation I that satisfies a specification s, if one exists. We obtain this engine for free by exploiting Rosette's symbolic evaluation facilities. To search for a satisfying interpretation $I \models s$, MemSynth simply evaluates $\text{INTERPRET}(s,I)$ against an interpretation I that binds the free variables in s to matrices populated with *symbolic boolean values* (using the INSTANTIATE function in Fig. 2). The result of $\text{INTERPRET}(s,I)$ is a symbolic encoding of the semantics of s, which is then checked for satisfiability with an off-the-shelf SMT solver [17]. This lifted evaluation works both on symbolic interpretations and on specifications that are made symbolic by the inclusion of expression sketches. This evaluation strategy also offers precise state space control: by exploiting domain-specific knowledge to reduce the number of symbolic values in I, MemSynth outperforms state-of-the-art relational solvers [28] as we show in Sec. 5.

2.2 Litmus Tests and Memory Models

A *litmus test* is a small multi-threaded program together with a candidate outcome, expressed as a constraint on the program's final state. A *memory model* determines whether the outcome is *allowed* or *forbidden* for the program. For example, the Intel Software Developer's Manual [21] includes the following litmus test to illustrate a surprising behavior

specification	s	::= $(\!\|U; D; f\|\!)$
universe	U	::= $\{a[,a]^*\}$
declarations	D	::= $\{\} \mid \{d[,d]^*\}$
declaration	d	::= $r :_k [b,b]$
bound	b	::= $\{(\langle a[,a]^*\rangle)^*\}$
formula	f	::= true $\mid$ false $\mid$ e in e $\mid$ $e = e$ $\mid$ no e $\mid$ some e $\mid$ one e $\mid$ not f $\mid$ f and f $\mid$ f or f $\mid$ f implies f $\mid$ f iff f $\mid$ all $x : e.\ f$ $\mid$ exists $x : e.\ f$
expression	e	::= $r \mid c \mid e + e \mid e \,\&\, e \mid e - e \mid e.e \mid e \to e \mid \hat{}e \mid {\sim}e \mid \{x : e \mid f\}$
arity	k	::= positive integer
relation	r	::= identifier
variable	x	::= identifier
scalar	a	::= identifier
constant	c	::= univ $\mid$ iden

(a) Abstract syntax

$$[\![(\!|U; d_1, \ldots, d_n; f|\!)]\!]I = \bigwedge_{i=1}^{n} [\![d_i]\!]I \wedge [\![f]\!]I \wedge (U(I) = U)$$
$$[\![r :_k [b_L, b_U]]\!]I = b_L \subseteq I(r) \subseteq b_U$$
$$[\![\text{true}]\!]I = \top$$
$$[\![\text{false}]\!]I = \bot$$
$$[\![p \text{ in } q]\!]I = [\![p]\!]I \subseteq [\![q]\!]I$$
$$[\![p = q]\!]I = [\![p]\!]I = [\![q]\!]I$$
$$[\![\text{no } p]\!]I = [\![p]\!]I \subseteq \emptyset$$
$$[\![\text{some } p]\!]I = \emptyset \subset [\![p]\!]I$$
$$[\![\text{one } p]\!]I = |[\![p]\!]I| = 1$$
$$[\![\text{not } f]\!]I = \neg [\![f]\!]I$$
$$[\![f \text{ and } g]\!]I = [\![f]\!]I \wedge [\![g]\!]I$$
$$[\![f \text{ or } g]\!]I = [\![f]\!]I \vee [\![g]\!]I$$
$$[\![f \text{ implies } g]\!]I = [\![f]\!]I \Rightarrow [\![g]\!]I$$
$$[\![f \text{ iff } g]\!]I = [\![f]\!]I \Leftrightarrow [\![g]\!]I$$

$$[\![\text{all } x : p.\ f]\!]I = \wedge_{v \in [\![p]\!]I} [\![f]\!](I[x := v])$$
$$[\![\text{exists } x : p.\ f]\!]I = \vee_{v \in [\![p]\!]I} [\![f]\!](I[x := v])$$
$$[\![r]\!]I = I(r)$$
$$[\![\text{univ}]\!]I = \{\langle a \rangle \mid a \in U(I)\}$$
$$[\![\text{iden}]\!]I = \{\langle a, a \rangle \mid a \in U(I)\}$$
$$[\![p + q]\!]I = [\![p]\!]I \cup [\![q]\!]I$$
$$[\![p \,\&\, q]\!]I = [\![p]\!]I \cap [\![q]\!]I$$
$$[\![p - q]\!]I = [\![p]\!]I \setminus [\![q]\!]I$$
$$[\![p.q]\!]I = \{\langle p_1, \ldots, p_n, q_1, \ldots, q_m \rangle \mid \langle p_1, \ldots, p_n, z \rangle \in [\![p]\!]I \wedge \langle z, q_1, \ldots, q_m \rangle \in [\![q]\!]I\}$$
$$[\![p \to q]\!]I = \{\langle p_1, \ldots, p_n, q_1, \ldots, q_m \rangle \mid \langle p_1, \ldots, p_n \rangle \in [\![p]\!]I \wedge \langle q_1, \ldots, q_m \rangle \in [\![q]\!]I\}$$
$$[\![\hat{}p]\!]I = [\![p]\!]I \cup [\![p.p]\!]I \cup [\![p.p.p]\!]I \cup \ldots$$
$$[\![{\sim}p]\!]I = \{\langle p_2, p_1 \rangle \mid \langle p_1, p_2 \rangle \in [\![p]\!]I\}$$
$$[\![\{x : p \mid f\}]\!]I = \{v \in [\![p]\!]I \mid [\![f]\!](I[x := v])\}$$

(b) Semantics

Figure 1. The syntax and semantics of bounded relational logic [38].

INTERPRET(p, I)
Inputs: Relational syntax p; interpretation I
Output: Encoding of the semantics of p (according to Fig. 1) with respect to (possibly symbolic) bindings in I

INSTANTIATE(D)
Input: Set of relation declarations $D = \{d_1, \ldots, d_n\}$
Output: Interpretation I that binds each decl. $r :_k [R_l, R_u]$ in D to a matrix with entries

$$m[i_1, \ldots, i_k] = \begin{cases} \top & \langle u_{i_1}, \ldots, u_{i_k} \rangle \in R_l \\ \text{freshSymBool}() & \langle u_{i_1}, \ldots, u_{i_k} \rangle \in R_u \setminus R_l \\ \bot & \text{otherwise} \end{cases}$$

Figure 2. Functions provided by the MemSynth DSL for interpreting relational formulas.

allowed by the x86 memory model, where reads may be reordered with earlier writes:

Test `x86/3`

Thread 1	**Thread 2**
1: X ← 1	3: Y ← 1
2: r1 ← Y	4: r2 ← X

Outcome: $\texttt{r1} = 0 \wedge \texttt{r2} = 0$
x86: allowed

We assume that all memory locations (denoted by capital letters) and registers (denoted by r1, r2, etc.) initially hold the value 0 unless stated otherwise. The instruction X ← 1 means that 1 is written to the memory location X, and r1 ← Y means that the value at memory location Y is read into register r1. The outcome is a conjunction of equalities that specify final values of memory (optional) and registers (mandatory).

Litmus Tests as Relations. Litmus tests (Def. 1) have a natural representation [39] in bounded relational logic: a test defines a finite universe of discourse U and a set of relations $\mathcal{V}$ over that universe. The universe consists of *memory events* (i.e., read, write, and fence instructions), *locations*, *threads*, and *values* that appear in the test. The relations $\mathcal{V}$ encode the test's syntax and candidate outcome. The contents of each relation are known statically (i.e., the values observed by each read are known from the test's outcome predicate), and MemSynth extracts them automatically from the test.

Definition 1 (Litmus test)**.** *A* litmus test *is a specification* $T = (\!|U; \mathcal{V}; \text{true}|\!)$*, where* U *is a finite universe of discourse and* $\mathcal{V}$ *is a set of relation declarations over* U *that encode the test's syntax and candidate outcome as follows:*

- *Every relation declaration in* $\mathcal{V}$ *takes the form* $r :_k [R, R]$*. That is,* $I(r) = R$ *for all interpretations* I*, and we say that* r *is* constant*.*
- *Unary relations* Event*,* Thread*,* Location*, and* Value *partition the universe* U *into memory events, threads, locations, and values.* Value *always includes the distinguished value* 0*.* Event *is partitioned by* Read*,* Write*,* Fence*, and* LWFence *relations, which contain reads, writes, heavyweight fences, and lightweight fences, respectively.*
- *The* thd *relation is a function from* Event *to* Thread*.*
- loc *and* val *map each event* $e \in$ Read + Write *to the* Location *and* Value*, respectively, that they read or write.*
- *The program order relation* po *is a strict partial order over* Event *(i.e., irreflexive, transitive, and asymmetric); if* $(e_1, e_2) \in$ po*, then events* e_1 *and* e_2 *share a thread (i.e.,* e_1.thd $= e_2$.thd*) and event* e_1 *executes before event* e_2*.*
- *The dependencies relation* dep *is a subset of* po*; if* $(e_1, e_2) \in$ dep *then event* e_2 *depends on event* e_1*.*
- *The final value relation* final *is a partial function from* Location *to* Value*, specifying constraints on the final state of memory imposed by the test's candidate outcome.*

Example 3. Consider the litmus test `x86/3` above. This test defines a universe $U = E \cup L \cup T \cup V$ with four events $E = \{e_1, e_2, e_3, e_4\}$, two locations $L = \{\texttt{X}, \texttt{Y}\}$, two threads $T = \{t_1, t_2\}$, and two values $V = \{0, 1\}$. Its relations $\mathcal{V}$ are:

$\mathsf{Read} = \{\langle e_2\rangle, \langle e_4\rangle\}$ $\quad$ $\mathsf{Write} = \{\langle e_1\rangle, \langle e_3\rangle\}$
$\mathsf{Fence} = \{\}$ $\quad$ $\mathsf{Thread} = \{\langle t_1\rangle, \langle t_2\rangle\}$
$\mathsf{LWFence} = \{\}$ $\quad$ $\mathsf{Location} = \{\langle \mathsf{X}\rangle, \langle \mathsf{Y}\rangle\}$
$\mathsf{Value} = \{\langle 0\rangle, \langle 1\rangle\}$ $\quad$ $\mathsf{dep} = \{\}$
$\mathsf{po} = \{\langle e_1, e_2\rangle, \langle e_3, e_4\rangle\}$ $\quad$ $\mathsf{final} = \{\}$
$\mathsf{thd} = \{\langle e_1, t_1\rangle, \langle e_2, t_1\rangle, \langle e_3, t_2\rangle, \langle e_4, t_2\rangle\}$
$\mathsf{loc} = \{\langle e_1, \mathsf{X}\rangle, \langle e_2, \mathsf{Y}\rangle, \langle e_3, \mathsf{Y}\rangle, \langle e_4, \mathsf{X}\rangle\}$
$\mathsf{val} = \{\langle e_1, 1\rangle, \langle e_2, 0\rangle, \langle e_3, 1\rangle, \langle e_4, 0\rangle\}$

Memory Models as Constraints on Executions. A memory model consists of axioms that constrain the set of executions allowed for a concurrent program, such as a litmus test. An (arbitrary) execution E of a test is described by a set of relation declarations, drawn from the test's universe (Def. 2). The details of E vary for different memory model frameworks, so MemSynth is parametric in its definition of executions. Memory models (Def. 3) are defined abstractly as well. A model takes as input a test T and an execution E, and produces a relational formula that encodes its axioms. The resulting formula may refer to free variables declared by T and E, but no other free variables. Given these definitions, we formalize the notions of allowed and forbidden outcomes simply as relational satisfiability queries (Def. 4).

Definition 2 (Execution)**.** *An* execution $E = \textsc{Exec}(T)$ *of a litmus test* $T = (\!|U; \mathcal{V}; \text{true}|\!)$ *is a set of relation declarations* $r :_k [R_l, R_u]$ *such that no* r *appears in* $\mathcal{V}$ *and each* R_l, R_u *is drawn from* U.

Definition 3 (Memory model)**.** *A* memory model $M(T,E)$ *is a function that takes as input a litmus test* T *and execution* $E = \textsc{Exec}(T)$*, and returns a relational formula in which the only free variables are those declared in* T *and* E.

Definition 4 (Allowed Outcomes)**.** *Let* $T = (\!|U; \mathcal{V}; \text{true}|\!)$ *be a litmus test and* $E = \textsc{Exec}(T)$ *an arbitrary execution of* T. *The test* T *is* allowed *by a memory model* M *if there exists an interpretation* I *such that* $I \models (\!|U; \mathcal{V}, E; M(T,E)|\!)$. *Otherwise,* T *is* forbidden *by* M.

To simplify our definitions, in the rest of the paper we write

$$\text{Allow}(M,T,E) = (\!|U; \mathcal{V}, E; M(T,E)|\!)$$

to stand for the specification in Def. 4. A litmus test T is therefore allowed by a memory model M if there exists an I such that $I \models \text{Allow}(M, T, \textsc{Exec}(T))$.

2.3 MemSynth$_A$

This section illustrates the definitions of executions (Def. 2) and memory models (Def. 3) by instantiating them according to an axiomatic framework by Alglave et al. [5]. The framework uses a global-time model of memory. We call our instantiation of this framework MemSynth$_A$.

2.3.1 Executions

MemSynth$_A$ uses two relations, rf and ws, to define the execution of a litmus test (Def. 5). The *reads-from* relation rf maps each write event to the reads that observe it: if $(w,r) \in \mathsf{rf}$, then w and r are a write and a read, respectively, to the same address and with the same value. The *write serialization* relation ws places a total order on all writes to the same location.

$\mathsf{ppo}_{SC} \triangleq \mathsf{po}$	$\mathsf{ppo}_{TSO} \triangleq \mathsf{po} - (\mathsf{Write} \to \mathsf{Read})$
$\mathsf{grf}_{SC} \triangleq \mathsf{rf}$	$\mathsf{grf}_{TSO} \triangleq \mathsf{rf} - (\mathsf{thd}.\sim\mathsf{thd})$
$\mathsf{fences}_{SC} \triangleq \emptyset$	$\mathsf{fences}_{TSO} \triangleq \mathsf{po}.\mathsf{Fence}.\mathsf{po}$
(a) Sequential consistency	**(b)** Total store order

Figure 3. Examples of common memory models defined by hand in the MemSynth$_A$ framework.

Definition 5 (MemSynth$_A$ Execution)**.** *In* MemSynth$_A$*, an* execution *E of a litmus test T declares two relations:*

- *The* reads-from *relation* rf *is a subset of* $\mathsf{Write} \to \mathsf{Read}$*, such that if* $(w,r) \in \mathsf{rf}$ *then (1)* $w.\mathsf{loc} = r.\mathsf{loc}$ *and* $w.\mathsf{val} = r.\mathsf{val}$*, and (2) for all* $w' \in \mathsf{Write}$*, if* $w' \neq w$ *then* $(w',r) \notin \mathsf{rf}$.
- *The* write serialization *relation* ws *is a subset of* $\mathsf{Write} \to \mathsf{Write}$*, such that if* $(w_1, w_2) \in \mathsf{ws}$ *then* $w_1.\mathsf{loc} = w_2.\mathsf{loc}$*, and for every memory location* $l_i \in \mathsf{Location}$*, the relation* $\{(w_1, w_2) \in \mathsf{ws} \mid w_1.\mathsf{loc} = l_i\}$ *is a total order.*

2.3.2 Memory Model

MemSynth$_A$ defines a memory model $M(T,E)$ as a relational formula that constructs a happens-before order and checks its acyclicity. MemSynth$_A$ is parametric in the definition of a memory model—several different memory models can be defined within the same framework. This freedom is exposed through three relations $\langle \mathsf{ppo}, \mathsf{grf}, \mathsf{fences}\rangle$ that define the allowed intra-thread reorderings, inter-thread reorderings, and reorderings across fences, respectively. Fig. 3 shows examples of these relations for the common sequential consistency (SC) and total store order (TSO) models.

Preserved program order. The *preserved program order* relation ppo defines which thread-local reorderings are allowed by a memory model. Given the program order relation po of a litmus test, $\mathsf{ppo} \subseteq \mathsf{po}$ specifies the program-order edges in po that cannot be reordered. In Fig. 3, sequential consistency allows no thread-local reordering, while total store order (TSO) allows writes to be reordered beyond later reads by excluding write-to-read edges from ppo.

Global reads-from. The *global reads-from* relation grf defines which inter-thread communications create ordering requirements between events. Given the reads-from relation rf from an execution (Def. 2), grf specifies the edges in rf that must be globally ordered. In Fig. 3, sequential consistency allows no reordering, and so every edge in rf creates an ordering obligation. On the other hand, total store order (TSO) allows threads to read their own writes early, and so if a read observes a write on the same thread, it should not create an ordering obligation for other threads.

$$\mathsf{fr} \triangleq (\sim\mathsf{rf}.\mathsf{ws}) + \{\langle r,w\rangle : \mathsf{Read} \to \mathsf{Write} \mid (\text{no } \mathsf{rf}.r) \text{ and } (r.\mathsf{loc} = w.\mathsf{loc})\}$$
$$\mathsf{ghb} \triangleq \mathsf{ppo} + \mathsf{ws} + \mathsf{fr} + \mathsf{grf} + \mathsf{fences}$$

(a) Auxiliary relations

$$\begin{aligned}
\mathsf{Execution} \triangleq\ & \mathsf{rf} \text{ in } (\mathsf{Write} \to \mathsf{Read}) \,\&\, (\mathsf{loc}.\sim\mathsf{loc}) \,\&\, (\mathsf{val}.\sim\mathsf{val}) \\
& \text{and no } (\mathsf{rf}.\sim\mathsf{rf} - \mathsf{iden}) \\
& \text{and } \mathsf{ws} \text{ in } (\mathsf{Write} \to \mathsf{Write}) \,\&\, \mathsf{loc}.\sim\mathsf{loc} \\
& \text{and no iden} \,\&\, \mathsf{ws} \\
& \text{and } \mathsf{ws}.\mathsf{ws} \text{ in } \mathsf{ws} \\
& \text{and all } a : \mathsf{Write}.\ \text{all } b : \mathsf{Write}. \\
& \qquad (\text{not } (a = b) \text{ and } a.\mathsf{loc} = b.\mathsf{loc}) \\
& \qquad\quad \text{implies } (\langle a,b\rangle \text{ in } \mathsf{ws} \text{ or } \langle b,a\rangle \text{ in } \mathsf{ws}) \\
\mathsf{Init} \triangleq\ & \text{all } r : \mathsf{Read}.\ (\text{no } \mathsf{rf}.r) \text{ implies } r.\mathsf{val} = 0 \\
\mathsf{Uniproc} \triangleq\ & \text{no } \hat{}(\mathsf{rf} + \mathsf{ws} + \mathsf{fr} + (\mathsf{po} \,\&\, \mathsf{loc}.\sim\mathsf{loc})) \,\&\, \mathsf{iden} \\
\mathsf{Thin} \triangleq\ & \text{no } \hat{}(\mathsf{rf} + \mathsf{dep}) \,\&\, \mathsf{iden} \\
\mathsf{Final} \triangleq\ & \text{all } w : \mathsf{Write}.\ (w \text{ in } (\mathsf{univ}.\mathsf{ws} - \mathsf{ws}.\mathsf{univ}) \text{ and some } (w.\mathsf{loc}).\mathsf{final}) \\
& \qquad \text{implies } w.\mathsf{val} = w.\mathsf{loc}.\mathsf{final} \\
\mathsf{Acyclic} \triangleq\ & \text{no } \hat{}\mathsf{ghb} \,\&\, \mathsf{iden} \\
\mathsf{Valid} \triangleq\ & \mathsf{Execution} \text{ and } \mathsf{Init} \text{ and } \mathsf{Uniproc} \text{ and } \mathsf{Thin} \text{ and } \mathsf{Final} \text{ and } \mathsf{Acyclic}
\end{aligned}$$

(b) Axioms

Figure 4. The axioms of the MemSynth$_A$ framework extends those of Alglave et al. [5], with changes to remove initialization write events and support outcomes for memory locations.

Fences. The *fences* relation fences defines which events are ordered by a memory fence. For example, the x86 architecture has a fence instruction `mfence` that serializes all memory reads and writes issued prior to it. The TSO example in Fig. 3 therefore adds all memory events separated by a fence to the fences relation. Some relaxed memory models, such as PowerPC and ARM, also have a notion of fence *cumulativity* [20], in which fence operations create orderings between events on other threads. The rules for cumulativity are subtle, but MemSynth correctly synthesizes them for PowerPC in under 16 seconds, as we show in Sec. 5.1.

Axioms Given the definitions of ppo, grf, and fences, MemSynth$_A$ uses the axioms in Fig. 4 to specify the memory model $M(T,E)$. The axioms follow Alglave et al. [5], with two changes for better solving performance. First, we omit initialization write events (events that initialize each memory location to hold the value 0). Second, we use an explicit Final axiom to encode outcome constraints on memory locations, rather than simulating all possible memory states as Alglave et al.'s herd tool does [7].

The first five axioms in Fig. 4(b) define well-formedness of an execution E. The Execution axiom applies the rules in Def. 5 to the rf and ws relations. The *initialization* axiom Init states that reads absent from the reads-from relation rf observe the initial value 0. The *uniprocessor* axiom Uniproc requires executions to respect coherence at each memory location. The *thin-air* axiom Thin prevents executions that create values out of thin air (i.e., involve cyclic dependencies). Lastly, the *final value* axiom Final imposes the constraints defined by the final relation.

To define whether an execution is *allowed*, MemSynth$_A$ constructs a *global happens-before* order ghb reflecting the orderings between events induced by the memory model. The Valid axiom concludes that a litmus test T is allowed by a memory model M if there exists some valid execution for which the global happens-before relation is acyclic (i.e., no event is transitively reachable from itself). That is, MemSynth$_A$ defines

$$M(T,E) \triangleq \mathsf{Valid}$$

where the free variables in Fig. 4 are instantiated with the appropriate values from the test T and execution E.

3. Memory Model Queries

MemSynth is designed to efficiently answer four queries about memory models and their specifications:

Verification determines whether a litmus test is allowed or forbidden by a memory model;

Synthesis generates a memory model that produces desired outcomes on a set of litmus tests;

Equivalence determines whether two memory models are equivalent (within certain bounds); and

Uniqueness decides whether a memory model is the only one that explains the outcomes of a set of litmus tests.

This section defines the MemSynth queries and explains their utility in building and refining memory model specifications. Sec. 4 shows how to implement these queries to scale to hundreds of litmus tests and large specifications.

3.1 Verification

The verification query, determining whether a memory model allows a litmus test, is well-studied in the literature [5, 23, 25, 39, 42]. Given a litmus test $T = (\!|U;\mathcal{V};\text{true}|\!)$ (Def. 1) and memory model M (Def. 3), the verification query checks satisfiability of the formula

$$\exists I.[\![\text{Allow}(M,T,\textsc{Exec}(T))]\!]I$$

where $\text{Allow}(M,T,E) = (\!|U;\mathcal{V},E;M(T,E)|\!)$, as defined in Sec. 2.2. If this formula is satisfiable, then M allows the test T (Def. 4). Otherwise, M forbids T. The verification query involves a straightforward satisfiability check that can be discharged with any relational solver, including MemSynth.

3.2 Synthesis

The synthesis query generates a memory model that is consistent with the desired outcomes for a set of litmus tests. Given a set $\mathcal{T}_P$ of tests that should be allowed, and a set $\mathcal{T}_N$ of tests that should be forbidden, the synthesis task is to find a memory model M that allows all tests in $\mathcal{T}_P$ and forbids all tests in $\mathcal{T}_N$. This query amounts to solving the formula

$$\begin{aligned}
\exists M.\ & \bigwedge_{T\in\mathcal{T}_P} \exists I.\ [\![\text{Allow}(M,T,\textsc{Exec}(T))]\!]I \\
& \wedge \bigwedge_{T\in\mathcal{T}_N} \forall I.\ \neg[\![\text{Allow}(M,T,\textsc{Exec}(T))]\!]I
\end{aligned} \tag{1}$$

In general, M could be any function that implements Def. 3, but solving such a synthesis query would be intractable. Instead, MemSynth allows the formula returned by M to contain *holes* [35] which the synthesizer will complete. The set of candidate completions for each hole is defined by an expression sketch (Sec. 2.1). For example, MemSynth$_A$ defines a memory model using the ppo, grf, and fences relations (Sec. 2.3.2). To synthesize a memory model from a set of litmus tests, these three relations are replaced with expression sketches that define the search space to explore.

The synthesis query involves higher-order universal quantification over the execution relations E for forbidden tests $\mathcal{T}_N$. The recent Alloy* solver [28] supports finite model finding for relational formulas with higher-order quantifiers, and so could in principle solve the synthesis query. In practice, however, these queries are intractable for Alloy* because its language lacks crucial constructs for precisely specifying the size and shape of the search space: expression sketches and bounds on the contents of declared relations. These limitations motivated our embedding of bounded relational logic in Rosette (Sec. 2). In Sec. 4.2, we present a synthesis algorithm for solving synthesis queries that scales to large expression sketches and many litmus tests.

3.3 Equivalence

MemSynth can compare two memory models M_A and M_B for equivalence. If they are not equivalent, MemSynth generates a *distinguishing litmus test* T_D on which they disagree (i.e., one model allows T_D while the other forbids it). As with existing work on generating distinguishing tests [24], the equivalence check is bounded, proving two models equivalent only up to a bound on the size of the distinguishing test. These bounds are defined by a *symbolic* litmus test, in which some relations are not constant (in contrast to Def. 1).

Definition 6 (Symbolic litmus test)**.** *A* symbolic litmus test $T_S = (\!|U;\mathcal{V};f|\!)$ *is a litmus test (Def. 1) with two modifications:*

- *Relation declarations in $\mathcal{V}$ are not required to be constant: some declarations $r :_k [R_l, R_u]$ in $\mathcal{V}$ may not have $R_l = R_u$.*
- *The formula f is a* well-formedness *predicate for the litmus test, in which the only free variables are those declared in $\mathcal{V}$.*

The equivalence query solves for a distinguishing litmus test by checking the satisfiability of two formulas:

$$\exists I_T.\ [\![T_S]\!]I_T \wedge \exists I.[\![\text{Allow}(M_A, T_S, \text{EXEC}(T_S))]\!](I_T \cup I) \\ \wedge \forall I.\neg[\![\text{Allow}(M_B, T_S, \text{EXEC}(T_S))]\!](I_T \cup I)$$

to find a test T_S on which M_A is weaker than M_B (i.e., M_A allows a test that M_B forbids), and similarly the second formula

$$\exists I_T.\ [\![T_S]\!]I_T \wedge \exists I.[\![\text{Allow}(M_B, T_S, \text{EXEC}(T_S))]\!](I_T \cup I) \\ \wedge \forall I.\neg[\![\text{Allow}(M_A, T_S, \text{EXEC}(T_S))]\!](I_T \cup I)$$

for a test on which M_A is stronger than M_B. In both formulas, the symbolic litmus test $T_S = (\!|U;\mathcal{V};\mathsf{WFP}|\!)$ includes a well-formedness predicate WFP, a relational formula that ensures the resulting test is a valid program. If either formula is satisfiable, then $T_D = \text{EVAL}(T_S, I_T)$ is a litmus test that distinguishes the two models M_A and M_B. If both formulas are unsatisfiable, then M_A and M_B are equivalent on all valid tests in the search space defined by T_S.

3.4 Uniqueness

The uniqueness query checks whether a memory model M is the only one that gives the desired outcomes a set of allowed ($\mathcal{T}_P$) and forbidden ($\mathcal{T}_N$) litmus tests. To do so, the query attempts to synthesize a second memory model M_S and a distinguishing litmus test T_S such that M_S and M disagree on T_S but agree on all tests in $\mathcal{T}_P$ and $\mathcal{T}_N$. If such a model and test exist, the set of given tests is ambiguous: there are two distinct memory models that both explain the input tests $\mathcal{T}_P \cup \mathcal{T}_N$.

Establishing uniqueness involves checking the satisfiability of a formula that combines synthesis and equivalence:

$$\exists I_T, M_S.\ [\![T_S]\!]I_T \wedge \bigwedge_{T \in \mathcal{T}_P} \exists I.\ [\![\text{Allow}(M_S, T, \text{EXEC}(T))]\!]I \\ \wedge \bigwedge_{T \in \mathcal{T}_N} \forall I.\ \neg[\![\text{Allow}(M_S, T, \text{EXEC}(T))]\!]I \\ \wedge \exists I.[\![\text{Allow}(M_S, T_S, \text{EXEC}(T_S))]\!](I_T \cup I) \\ \wedge \forall I.\neg[\![\text{Allow}(M, T_S, \text{EXEC}(T_S))]\!](I_T \cup I)$$

and a second formula that swaps M_S and M in the final two conjuncts (akin to the two equivalence formulas). If either formula is satisfiable, then M_S is a second memory model that produces the desired outcomes on all tests in $\mathcal{T}_P$ and $\mathcal{T}_N$, and $T_D = \text{EVAL}(T_S, I_T)$ is a litmus test that distinguishes M and M_S. If both formulas are unsatisfiable, then M is the only memory model that produces the desired outcomes. This uniqueness result is with respect to two bounds: the search space for M_S (defined using sketches, as for the synthesis query), and the search space for the symbolic litmus test T_S.

The uniqueness query identifies ambiguities in the set of input litmus tests, and so can form the basis of a refinement loop to guide the development of a memory model specification. For example, if we take $\mathcal{T}_P$ to contain only the test `x86/3` from Sec. 2.2, and $\mathcal{T}_N$ to be empty, then many distinct memory models produce the desired outcomes (TSO, RMO, PowerPC, etc.). If we take M to be one such model, the uniqueness query will identify a second model that also allows test `x86/3`, and produce a new distinguishing litmus test T_D to resolve the ambiguity. By deciding the desired outcome for T_D and adding it to the appropriate set ($\mathcal{T}_P$ or $\mathcal{T}_N$), we can repeat the synthesis process to refine the memory model M. The user can decide on the desired outcome for T_D by inspecting documentation, executing the test on hardware, consulting with system architects, or otherwise.

4. Reasoning Engine

This section presents MemSynth's engine for answering the queries in Sec. 3. We show the algorithms to implement these

```
1  function VERIFY(M, T)
2      (|U; V; true|) ← T
3      E ← EXEC(T)
4      I ← INSTANTIATE(E ∪ V)
5      φ ← INTERPRET(Allow(M, T, E), I)
6      return SOLVE(φ) = SAT

7  function EXEC(T = (|U; V; true|))
8      I ← INSTANTIATE(V)                          ▷ Make an interpretation from V
9      B_u^rf ← INTERPRET((Write → Read) & (loc.~loc) & (val.~val), I)
10     B_u^ws ← INTERPRET((Write → Write) & (loc.~loc), I)        ▷ Fig. 4
11     return {rf :₂ [∅, B_u^rf], ws :₂ [∅, B_u^ws]}
```

Figure 5. MemSynth's verification procedure VERIFY takes as input a memory model M and litmus test T and returns whether the model allows the litmus test. The EXEC procedure computes relational bounds for an execution E.

queries, and describe key optimizations to make them scale to real-world memory models.

4.1 Verification

The verification query (Sec. 3.1) determines whether a memory model M allows a litmus test T. The VERIFY procedure in Fig. 5 takes as input a memory model M and litmus test T, and returns true iff M allows T. The VERIFY procedure first computes bounds for the relations in the execution E (Def. 2). Given these bounds, it then checks the satisfiability of the relational specification $\text{Allow}(M,T,E)$. The implementation of VERIFY is only five lines of code, demonstrating the utility of our relational DSL for reasoning about memory models.

Bounds Compaction. The EXEC procedure in Fig. 5 computes bounds for the free relations in a MemSynth$_A$ execution E. A naive bound that includes every tuple of the appropriate arity is sound, but tighter bounds can significantly improve performance, since the difference between the upper and lower bounds for each free relation defines the size of the search space. For MemSynth$_A$, an execution consists of two relations rf and ws, that specify a reads-from and write serialization order, respectively (Def. 5). EXEC computes an upper bound for both relations based on the Execution axioms in Fig. 4. The rf relation contains only tuples (w,r) where w is a write, r is a read, and both w and r access the same location with the same value. Likewise, the ws relation contains only tuples (w_1,w_2) where both entries are writes to the same location. Compared to naive upper bounds, this more compact search space improve verification time by an average of 24× on the PowerPC tests discussed in Sec. 5.1.

4.2 Synthesis

The synthesis query (Sec. 3.2) generates a memory model that gives the desired outcomes on a set of litmus tests. The space of candidate solutions is defined by a memory model sketch $\mathcal{M}$, which is a memory model (Def. 3) with some expressions replaced by associated expression sketches (Sec. 2.1).

Our synthesis procedure, SYNTHESIZE (Fig. 6), takes as input a memory model sketch $\mathcal{M}$, a set of allowed litmus tests $\mathcal{T}_P$, and a set of forbidden litmus tests $\mathcal{T}_N$. Given these inputs, it uses our relational DSL (embedded in Rosette) to generate and solve quantified formulas using an off-the-shelf SMT solver [17]. Because MemSynth represents relations as matrices of boolean values, these formulas quantify over boolean variables. We found the Z3 SMT solver [17] to be extremely effective at discharging these formulas—an average of 2–5× faster than our own heavily-optimized implementation of counterexample-guided inductive synthesis [35].

SYNTHESIZE does not try to find a correct model for all tests in $\mathcal{T}_P$ and $\mathcal{T}_N$ at once, since this would require encoding every test in $\mathcal{T}$ up front. Instead, tests are added to the synthesis query incrementally. The order in which tests are added can influence synthesis performance; we choose a simple heuristic that adds tests in increasing order of size, which optimizes for small search spaces. This incrementalization reduces the size of the synthesis query substantially: in Sec. 5.1, we show that only 18 of 768 tests were added to the query when synthesizing a model for PowerPC.

The SYNTHESIZE procedure is sound, and is complete with respect to the input sketch: if a correct model exists within the input sketch, SYNTHESIZE will return a solution.[1]

Theorem 1 (Soundness)**.** *If* $\text{SYNTHESIZE}(\mathcal{M},\mathcal{T}_P,\mathcal{T}_N)$ *returns a memory model M, then M satisfies Equation 1.*

Theorem 2 (Termination)**.** $\text{SYNTHESIZE}(\mathcal{M},\mathcal{T}_P,\mathcal{T}_N)$ *terminates when* $\mathcal{T}_P$ *and* $\mathcal{T}_N$ *are finite sets.*

Theorem 3 (Completeness)**.** *If there exists a model M in the sketch* $\mathcal{M}$ *that satisfies Equation 1, and* $\mathcal{T}_P$ *and* $\mathcal{T}_N$ *are finite sets, then* $\text{SYNTHESIZE}(\mathcal{M},\mathcal{T}_P,\mathcal{T}_N)$ *will return a model.*

4.3 Equivalence

MemSynth can determine if two memory models are equivalent (up to given bounds) by searching for a litmus test on which they disagree. Our equivalence-checking procedure $\text{COMPARE}(M_A,M_B,T_S)$ takes as input two memory models M_A and M_B, and a symbolic litmus test T_S (Def. 6), and returns either a litmus test T such that $\text{VERIFY}(M_A,T) \neq \text{VERIFY}(M_B,T)$, or $\perp$ if no such test exists within the bounds of T_S. To search for a distinguishing test T, COMPARE solves the two quantified boolean equivalence formulas shown in Sec. 3.3 using the Z3 SMT solver (as with SYNTHESIZE), with two additional optimizations described next.

Symmetry Breaking. The symbolic litmus test T_S defines a search space that contains many redundant candidate tests. For example, after checking a test T, there is no need to also check a test T' that differs from T by a permutation of the used memory locations (e.g., T' swaps all instances of X and Y in the loc relation of T). To improve query performance, the bounds computation $\text{EXEC}(T_S)$ applies lex-leader symmetry breaking [15] to rule out tests that differ only by a permutation of threads, addresses, or values, similar to existing work [24]. The well-formedness predicate $\text{WFP}(T_S)$ adds several assertions to rule out uninteresting litmus tests

[1] Proofs are provided in supplementary material.

```
1  function SYNTHESIZE(M, T_P, T_N)
2      S ← new IncrementalSMTSolver()
3      T_U ← {}                                  ▷ Set of used tests
4      M ← false                                 ▷ Model that forbids all outcomes
5      T ← NEXTTEST(M, T_P, T_N, T_U)            ▷ Choose an initial test
6      while T ≠ ⊥ do
7          ADDTEST(S, M, T, T_P)                 ▷ Add encoding of T to S
8          T_U ← T_U ∪ T
9          I_b ← SOLVE(S)                        ▷ Boolean interpretation or UNSAT
10         if I_b = UNSAT then                   ▷ No model exists
11             return UNSAT
12         M ← EVAL(M, I_b)                      ▷ Use I_b to fill the holes in M
13         T ← NEXTTEST(M, T_P, T_N, T_U)        ▷ Choose the next test
14     return M          ▷ M gives the expected outcome on all tests in T_P ∪ T_N
```

(a) Main synthesis routine

```
1  function ADDTEST(S, M, T = (|U; V; true|), T_P)
2      E ← EXEC(T)
3      I ← INSTANTIATE(E ∪ V)                    ▷ Symbolic relational interpretation I
4      φ ← INTERPRET(Allow(M, T, E), I)          ▷ Boolean encoding
5      if T ∈ T_P then
6          ASSERT(S, φ)                          ▷ Add an allowed test
7      else
8          X ← SYMBOLICS(I, E)                   ▷ All symbolic booleans from E in I
9          ASSERT(S, ∀X. ¬φ)                     ▷ Add a forbidden test
```

(b) Test evaluation

```
1  function NEXTTEST(M, T_P, T_N, T_U)
2      for T ∈ (T_P ∪ T_N) \ T_U do              ▷ Iterate over unused tests
3          if VERIFY(M, T) ≠ (T ∈ T_P) then
4              return T                          ▷ M gives the wrong outcome on T
5      return ⊥          ▷ M gives the expected outcome on all unused tests
```

(c) Test selection

Figure 6. MemSynth's synthesis procedure SYNTHESIZE takes as input a memory model sketch $\mathcal{M}$, a set $\mathcal{T}_P$ of allowed litmus tests, and a set $\mathcal{T}_N$ of forbidden litmus tests, and returns a memory model that produces the given outcomes on all tests.

(e.g., ruling out tests that refer to a memory location exactly once, which has no visible effect on inter-thread memory reorderings). These optimizations reduce the run time of equivalence queries by 2–10×.

Concretization. MemSynth partially concretizes the symbolic litmus test T_S. In particular, the query in Sec. 4.3 is solved with respect to a concrete *topology*, which fixes the number of threads and instructions per thread (e.g., a single query may check only tests with 2 threads of 3 instructions each). The COMPARE procedure then implements a top-level search over all topologies using a metasketch [11]. The concrete topology allows $\text{EXEC}(T_S)$ to compute more compact bounds (e.g., the thread relation thd becomes entirely concrete), which reduces the search space exponentially.

4.4 Uniqueness

The final MemSynth query checks whether a memory model is unique for a set of allowed tests $\mathcal{T}_P$ and forbidden tests $\mathcal{T}_N$. The uniqueness procedure $\text{DISAMBIGUATE}(M, \mathcal{T}_P, \mathcal{T}_N, \mathcal{M}, T_S)$ takes as input a memory model M, sets of allowed tests $\mathcal{T}_P$ and forbidden tests $\mathcal{T}_N$, a memory model sketch $\mathcal{M}$, and a symbolic litmus test T_S. It returns a new memory model M_S and test T_D, such that for all $T \in \mathcal{T}_P \cup \mathcal{T}_N$, $\text{VERIFY}(M, T) = \text{VERIFY}(M_S, T)$, but for T_D, $\text{VERIFY}(M, T_D) \neq \text{VERIFY}(M_S, T_D)$. Since the uniqueness query involves synthesizing a memory model M_S and litmus test T_D, the implementation of DISAMBIGUATE extends SYNTHESIZE (Fig. 6) and benefits from the same optimizations as COMPARE.

5. Case Studies

To demonstrate that MemSynth is an effective approach to reasoning about memory models, we sought to answer three research questions:

- Can MemSynth scale to real-world memory models such as PowerPC and x86?
- Does MemSynth provide a basis for rapidly building useful automated memory model tools?
- Does MemSynth outperform existing relational solvers and memory model tools?

5.1 Can MemSynth scale to real-world memory models such as PowerPC and x86?

This section uses MemSynth to synthesize specifications for the PowerPC [20] and x86 [21] memory models. The results (summarized in Fig. 7) show that MemSynth scales to complex real-world models, and that its queries can aid in the design of memory model specifications by identifying ambiguities and redundancies in tests and documentation.

5.1.1 Synthesizing a PowerPC Model

The PowerPC architecture is well-known for relaxed memory behaviors that have proven difficult to formalize. Existing formalization efforts have identified subtle mis-specifications [4, 5, 26], making an automated process particularly appealing. To synthesize a specification for PowerPC, MemSynth uses a set of 768 litmus tests from Alglave et al. [3, 5], which they generated with their diy tool [6]. These tests vary from 6–24 instructions across 2–5 threads, and while they examine most aspects of the PowerPC memory model, they are not intended to be exhaustive. We use the Alglave et al. [5] model to decide whether each test should be allowed, although we could use hardware observations instead, as discussed later.

We employ MemSynth_A as the basis for the synthesis process. The memory model sketch replaces the ppo, grf, and fences relations (Sec. 2.3.2) with expression sketches for the synthesizer to complete. All three sketches use a grammar containing all relational expressions e in Fig. 1 other than set comprehension. For the barrier expression fences, we provide a sketch of the form $\text{fences} \triangleq \text{F}_{\text{Fence}} + \text{F}_{\text{LWFence}}$, where F_{Fence} and $\text{F}_{\text{LWFence}}$ are expression sketches containing Fence and LWFence, respectively, as terminals. This sketch expresses the high-level insight that PowerPC features two kinds of barriers (heavyweight `sync` fences and lightweight `lwsync` fences) that do not interact.

Synthesis. MemSynth synthesizes a model, which we call PPC_0, that agrees with the hand-written model by Alglave et al. on all 768 tests. The synthesis takes 16 seconds, and due to its heuristics for test ordering, the incremental synthesis algorithm (Fig. 6) uses only 18 of the 768 tests to find PPC_0.

	Input Tests			Memory Model Sketch		
Arch.	$\lvert\mathcal{T}_P\rvert$	$\lvert\mathcal{T}_N\rvert$	Time	ppo/grf Depth	fences Depth	State Space
PPC	163	605	16 s	4	4	2^{1406}
x86	2	8	2 s	4	0	2^{560}

(a) Synthesis results

			Litmus Test Sketch		
Arch.	New Tests	Time	Num. Threads	Num. Events	State Space
PPC	10	5.2 hrs	2–4	2–6	2^{165}
x86	3	3.4 hrs	2–4	2–6	2^{114}

(b) Uniqueness results

Figure 7. Results of real-world memory model synthesis and uniqueness experiments for PowerPC and x86. We describe the memory model and litmus test sketches both in terms of sketch-specific parameters (e.g., expression sketch depth) and the number of candidate solutions they contain (i.e., their state space). The uniqueness results (b) for a given architecture use the same memory model sketch as the synthesis results (a) for that architecture. Synthesized models and tests are provided in supplementary material.

Uniqueness. While the 768 tests described above cover much of the semantics of PowerPC, they do not identify a unique model. To resolve this ambiguity, we apply MemSynth's uniqueness query (Sec. 3.4) to enlarge the set until it identifies a single model. We use the Alglave et al. model as an oracle to decide the correct outcome for the generated distinguishing tests.

MemSynth finds 10 new tests to add to the set. The tests deal with the semantics of PowerPC barriers; for example:

Test `ppc/unique/5`

Thread 1	**Thread 2**
1: r1 ← B	4: r2 ← A
2: lwsync	5: sync
3: A ← 1	6: B ← 2

Outcome: $\texttt{r1} = 2 \wedge \texttt{r2} = 1$
PowerPC: forbidden

After adding the 10 tests, the synthesized model is equivalent to the Alglave et al. [5] model on all tests up to 6 instructions across 4 threads, and is the only model (within our sketch) that produces the given outcomes on all tests.

Discussion. MemSynth is complementary to test-generation tools such as diy [5]: these tools can seed the synthesis process with initial tests, and MemSynth can then identify ambiguities and synthesize new tests to resolve them. While our experiments use the hand-written model of Alglave et al. [5] as an oracle, we could instead determine litmus test outcomes by manually consulting documentation or by hardware experiments. For example, Alglave et al. [3] also ran their 768 tests on PowerPC hardware and observed whether each behavior occurred. MemSynth is able use the results of these experiments as an oracle, and synthesizes a new model PPC_1 in 22 seconds; the resulting model is not equivalent to PPC_0.

5.1.2 x86 Ambiguity and Redundancy

The x86 architecture specifies a variant of *total store ordering* (TSO) as its memory model. The x86 TSO memory model is defined in the Intel Software Developer's Manual [21] with prose and a set of 10 litmus tests. Though TSO is one of the simplest memory models, formalizing the subtleties of its x86 variant has been challenging [12, 13, 32, 34].

We used MemSynth to synthesize a specification of the x86 memory model. After extending MemSynth_A with support for atomic operations (x86's `xchg` instruction), MemSynth synthesizes a formalization TSO_0 that is correct on the Intel manual's 10 litmus tests in under two seconds.

Ambiguity. But MemSynth's uniqueness query (Sec. 3.4) determines that at least one other memory model, TSO_1, also satisfies all 10 tests, while disagreeing with TSO_0 on a new distinguishing test:

Test `x86/unique/2`

Thread 1	**Thread 2**
1: r1 ← A	3: B ← 1
2: r2 ← B	4: xchg(A, r3)

Initially: $\texttt{r1} = \texttt{r2} = 0 \wedge \texttt{r3} = 1$
Outcome: $\texttt{r1} = 1 \wedge \texttt{r2} = 0$

This test is a variant of the manual's example 8-1 [21], but with an atomic exchange instead of a plain write to location A. The documentation indicates that x86 should forbid this outcome, as TSO_0 does but TSO_1 does not.

Repeating the uniqueness query after adding this new test finds 2 more distinguishing tests that further examine the semantics of atomic operations. According to the documentation, these tests should also be forbidden. After adding these tests to the synthesis process, MemSynth is able to prove that a new synthesized model TSO_2 is unique, up to the bounds in Fig. 7 on the size of the model specification and distinguishing litmus test.

Potential Redundancy. In the paper on their x86-CC formalization of the x86 memory model, Sarkar et al. [32] write that "P8 may be redundant," where P8 is a principle from the Intel manual about which reorderings are allowed:

> "§8.2.3.9: Loads and stores are not reordered with locked instructions." [21]

The manual section describing this principle includes two litmus tests demonstrating forbidden reorderings. We found that if we omit these two tests from the synthesis process, the ambiguity experiment above re-discovers them, suggesting they are needed to uniquely identify x86's memory model.

5.2 Does MemSynth provide a basis for rapidly building useful automated memory model tools?

While previous sections build on MemSynth$_A$ and the Alglave et al. [5] framework, MemSynth is agnostic to the memory model framework with which it is instantiated. In this section, we present MemSynth$_{MH}$, an instantiation of MemSynth with a framework developed by Mador-Haim et al. [24, 25]. The implementation took only 2 days of work by one of this paper's authors. Moreover, we use MemSynth to automatically rectify a discrepancy between our implementation and the paper's results that we could not resolve by hand.

5.2.1 The MemSynth$_{\text{MH}}$ Framework

Mador-Haim et al.'s memory model framework [24, 25] was developed to *contrast* memory model specifications by generating a distinguishing litmus test on which two models disagree (as MemSynth's equivalence query does). A memory model is defined by a "must-not-reorder" function $F(x,y)$ that determines whether two instructions x and y can be reordered. The framework places syntactic restrictions on F such that it admits only 90 models. The authors prove that the size of litmus test needed to distinguish models in this set is bounded, and that only 82 of the 90 models are semantically distinct.

5.2.2 Repairing the Framework

After implementing MemSynth$_{MH}$, we found that our results differed from those in the original paper. The paper states there should be 82 distinct models, but our implementation found only 12 distinct models. Moreover, the paper identifies the following as a distinguishing litmus test (i.e., some models allow it while others forbid it):

Test `mh/L2`

Thread 1	**Thread 2**
1: X ← 1	3: r1 ← X
2: X ← 2	4: r2 ← X

Outcome: $\texttt{r1} = 2 \wedge \texttt{r2} = 0$

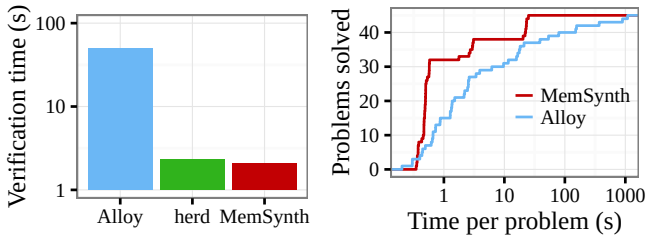
Yet our implementation reported this test (which contains a load-load coherence violation allowed by SPARC's RMO model) to be disallowed by all 90 memory models.

Our manual investigation implicated one of the paper's axioms for the happens-before relations:

> 5. **Ignore local**: If x is after y in program order, then x cannot happen before y.

Omitting this axiom from our implementation gave 86 distinct models, not 82 as expected, and so we hypothesized that the axiom was necessary but too strong. Since the paper correctly reports that `mh/L2` is allowed by RMO, we believe the paper's results are correct but this axiom was misprinted in the paper. However, the paper's authors were unable to provide their implementation for us to compare against [8].

We first tried to fix the axiom by hand, but despite several attempts, a correct fix eluded us: our closest results identified



(a) Verification query **(b)** Equivalence query

Figure 8. Performance comparisons between MemSynth and existing tools for (a) verification and (b) equivalence queries.

78 or 86 distinct models rather than 82. Instead, we used MemSynth's relational logic DSL to synthesize a repair. In relational logic, axiom 5 is written as "no $(\sim\texttt{po})\,\&\,\texttt{hb}$", where hb is the happens-before relation for an execution. To repair the axiom, we replaced ~po with an expression sketch of depth 3, and synthesized a completion that gave the correct outcomes on 9 litmus tests from the original paper on both TSO and RMO memory models. We were able to synthesize the following repair in 15 seconds:

$$\text{no}\,(\sim((\texttt{po} - \texttt{rf})\,\&\,(\texttt{Write} \rightarrow \texttt{Read})))\,\&\,\texttt{hb}$$

In prose:

> 5a. **Ignore local**: If x is after y in program order, x is a read, y is a write, and x does not read the value written by y, then x cannot happen before y.

The repaired axiom allows reads to see local writes early without affecting the happens-before relation. We believe it is intended to allow models such as TSO to observe their own writes early by ignoring the happens-before order. With the repaired axiom, our pairwise comparison results produce 82 distinct models, identical to the original paper.

5.3 Does MemSynth outperform existing relational solvers and memory model tools?

This section compares MemSynth to existing relational engines and memory model tools on verification, equivalence, and synthesis queries.

Verification. Fig. 8(a) shows the time for MemSynth, Alloy [22, 38], and herd [7] to verify 768 PowerPC litmus tests from Sec. 5.1. The Alloy results use the PPC_0 specification synthesized by MemSynth, while herd (configured in "speed check" mode) already supports PowerPC. The results show that MemSynth outperforms Alloy by 10×, and is comparable to herd's custom decision procedure for memory models.

Equivalence. We used MemSynth and Alloy* [28] to perform a pairwise comparison of 10 different synthesized PowerPC models. Fig. 8(b) shows that MemSynth outperforms Alloy* on most of these queries: MemSynth can solve twice as many queries in under one second, and the hardest problem takes 25 s for MemSynth versus 18 min for Alloy*. With symmetry breaking and concretization (which cause the large

steps in the MemSynth line in Fig. 8(b)) disabled, MemSynth could not solve any of the comparisons in under an hour.

Synthesis. The synthesis query (Sec. 3.2) requires higher-order quantification, and so we compared MemSynth to Alloy* [28]. Because Alloy* does not support expression sketches (Sec. 2), we designed a sketch $\mathcal{M}$ that simply chooses between hard-coded memory models. When given $\mathcal{M} = \{\text{SC}, \text{TSO}\}$, both MemSynth and Alloy* return in under a second. However, when given $\mathcal{M} = \{\text{SC}, \text{TSO}, \text{RMO}\}$, MemSynth still returns in under a second, but Alloy* times out after one hour. This result suggests Alloy* could not synthesize models from complex expression sketches.

6. Related Work

MemSynth is, to our knowledge, the first tool to provide synthesis and other higher-order queries for memory model specifications. It builds on existing work in formalizing and reasoning about memory models, which this section reviews.

Formalization. Few architectures formalize their memory models (with the exception of SPARC [40] and Alpha [14]), and so this task has fallen to researchers. A notable success is the x86-TSO model [34], which formalizes the memory model of the x86 architecture. This model was refined through several papers [29, 32], which revealed ambiguities in the x86 documentation. In Sec. 5.1.2, MemSynth's uniqueness query automatically identified more such ambiguities.

Another effort has developed several formalizations of the PowerPC architecture [4, 5, 7, 26, 33]. The PowerPC memory model allows many more reorderings than x86, and features cumulative barriers to restore stronger behavior. The specification for PowerPC is complex, and several ambiguities in the PowerPC manual [20] required detailed experimentation to resolve. The PowerPC formalization effort also developed a suite of memory model experimentation tools, which we use in Sec. 5.1 and Sec. 5.3.

Formalization efforts have also brought clarity to emerging programming language memory models, particularly C11 and C++11 [9, 10]. These efforts have helped check that the target models provide basic guarantees about important classes of programs—for example, that all data-race-free programs have sequentially consistent memory ordering [1]. Like hardware memory models, language memory models are also relational, and some (e.g., the Java Memory Model [27]) have already been formalized [39] in bounded relational logic. We therefore believe the MemSynth engine can be extended to language memory models with appropriate selection of a framework and new structures for litmus tests (Def. 1).

Frameworks. Recent work has developed generic memory model frameworks that can be instantiated with different architectures. The Nemos framework [42] offers axiomatic specifications for a variety of models, such as causal consistency, but (to our knowledge) cannot express microprocessor models such as TSO. Alglave et al. [2, 5, 7] developed an axiomatic framework for microprocessor memory models. It admits models for complex architectures such as PowerPC, and is the basis for our MemSynth_A framework (Sec. 2.3) and most experiments in Sec. 5. Mador-Haim et al. [25] developed a framework for expressing store-atomic memory models, which we implement in Sec. 5.2. It captures common models such as TSO, but is restricted enough to prove upper bounds on the size of distinguishing litmus tests.

Automated Reasoning. One common application of formal memory models is inserting synchronization instructions that restore sequential consistency in a concurrent program. Alglave et al. [5] address this problem for PowerPC with a specification of the platform's barrier semantics, including cumulativity; we automatically synthesize this specification in Sec. 5.1. Another common application is verification of concurrent code under relaxed memory models, and several tools have been developed for this purpose (e.g., [16, 18]). All of them rely on formal specifications of memory models that can be synthesized with MemSynth.

MemSAT [39] is an automated tool that implements the verification query of Sec. 3.1 for axiomatic memory model specifications. MemSAT found several discrepancies in the formalization of the Java Memory Model [27]. MemSynth is similar to MemSAT in its use of relational logic, but focuses on hardware memory models and offers richer automated reasoning queries including synthesis. Wickerson et al. [41] used Alloy* [28] to implement a tool for automatically comparing memory consistency models, similar to MemSynth's equivalence query. They show results for both processor and language memory models, but their tool does not support MemSynth's synthesis and uniqueness queries.

7. Conclusion

This paper presented MemSynth, a system for synthesizing axiomatic specifications of memory consistency models. MemSynth addresses the challenge of producing memory model formalizations, which are crucial for reasoning about concurrent code, from example litmus tests. MemSynth's expressive specification language builds on an optimized bounded relational logic engine, which serves as a platform for developing other novel automated queries, such as memory model uniqueness. We showed that MemSynth can synthesize specifications for complex architectures, refine those specifications by identifying ambiguities, and support rapid development of automated memory model tools that outperform hand-crafted versions. As new parallel architectures continue to emerge, MemSynth can help formalize their memory models rapidly and precisely.

References

[1] S. V. Adve and M. D. Hill. Weak ordering - a new definition. In *ISCA*, 1990.

[2] J. Alglave. A formal hierarchy of weak memory models. *Form. Methods Syst. Des.*, 41(2), 2012.

[3] J. Alglave and L. Maranget. The Phat Experiment. `http://diy.inria.fr/phat/`, 2010.

[4] J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In *DAMP*, 2009.

[5] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *CAV*, 2010.

[6] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hardware. In *TACAS*, 2011.

[7] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2), 2014.

[8] R. Alur and M. M. K. Martin. Personal communication, July 2016.

[9] M. Batty, S. Owens, S. Sarkar, P. Sewell, and W. Weber. Mathematizing C++ concurrency. In *POPL*, 2011.

[10] M. Batty, A. F. Donaldson, and J. Wickerson. Overhauling sc atomics in C11 and OpenCL. In *POPL*, 2016.

[11] J. Bornholt, E. Torlak, D. Grossman, and L. Ceze. Optimizing synthesis with metasketches. In *POPL*, 2016.

[12] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *CAV*, 2008.

[13] J. Burnim, K. Sen, and C. Stergiou. Sound and complete monitoring of sequential consistency for relaxed memory models. In *TACAS*, 2011.

[14] Compaq. *Alpha Architecture Reference Manual*. 4th edition, 2002.

[15] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *KR*, 1996.

[16] A. Dan, Y. Meshman, M. Vechev, and E. Yahav. Effective abstractions for verification under relaxed memory models. In *VMCAI*, 2015.

[17] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[18] B. Demsky and P. Lam. SATCheck: SAT-directed stateless model checking for SC and TSO. In *OOPSLA*, 2015.

[19] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010.

[20] IBM. *Power ISA Version 2.06 Revision B*. IBM, 2010.

[21] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, 2015. Revision 53.

[22] D. Jackson. *Software Abstractions: logic, language, and analysis*. MIT Press, 2nd edition, 2009.

[23] D. Lustig, M. Pellauer, and M. Martonosi. PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models. In *MICRO*, 2014.

[24] S. Mador-Haim, R. Alur, and M. M. K. Martin. Generating litmus tests for contrasting memory consistency models. In *CAV*, 2010.

[25] S. Mador-Haim, R. Alur, and M. M. K. Martin. Litmus tests for comparing memory consistency models: How long do they need to be? In *DAC*, 2011.

[26] 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 *CAV*, 2012.

[27] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL*, 2005.

[28] A. Milicevic, J. P. Near, E. Kang, and D. Jackson. Alloy*: A general-purpose higher-order relational constraint solver. In *ICSE*, 2015.

[29] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOLs*, 2009.

[30] S. Park and D. L. Dill. An executable specification, analyzer and verifier for RMO (Relaxed Memory Order). In *SPAA*, 1995.

[31] Racket. The Racket programming language. http://racket-lang.org.

[32] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *POPL*, 2009.

[33] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *PLDI*, 2011.

[34] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, July 2010.

[35] A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[36] E. Torlak and R. Bodik. Growing solver-aided languages with Rosette. In *Onward!*, 2013.

[37] E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, 2014.

[38] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *TACAS*, 2007.

[39] E. Torlak, M. Vaziri, and J. Dolby. MemSAT: Checking axiomatic specifications of memory models. In *PLDI*, 2010.

[40] D. L. Weaver and T. Germond. *The SPARC architecture manual (version 9)*. SPARC International, 1994.

[41] J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides. Automatically comparing memory consistency models. In *POPL*, 2017.

[42] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *IPDPS*, 2004.

[43] F. Zappa Nardelli, P. Sewell, J. Ševčík, S. Sarkar, S. Owens, L. Maranget, M. Batty, and J. Alglave. Relaxed memory models must be rigorous. In *EC*^{2}, 2009.