Synthesizing Memory Models from Framework Sketches and Litmus Tests

James Bornholt Emina Torlak
University of Washington, USA
{bornholt, emina}@cs.washington.edu

Abstract

A memory consistency model 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 through substantial manual effort, but as new architectures emerge, there is a growing need for tools to aid these efforts.

This paper presents MemSynth, a synthesis-aided system for reasoning about axiomatic specifications of memory models. MemSynth takes as input a set of litmus tests and a *framework sketch* that defines a class of memory models. The sketch comprises a set of axioms with missing expressions (or *holes*). Given these inputs, MemSynth synthesizes a completion of the axioms—i.e., a memory model—that gives the desired outcome on all tests. The MemSynth engine employs 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. This design also enables it to solve new kinds of queries, such as checking if a set of litmus tests unambiguously defines a memory model within a framework sketch.

We show that MemSynth can synthesize specifications for x86 in under two seconds, and for PowerPC in 12 seconds from 768 litmus tests. Our ambiguity check identifies missing tests from both the Intel x86 documentation and the validation suite of a previous PowerPC formalization. We also used MemSynth to reproduce, debug, and automatically repair a paper on comparing memory models in just two days.

Categories and Subject Descriptors C.1.4 [*Processor Architectures*]: Parallel Architectures; D.1.2 [*Programming Techniques*]: Automatic Programming

Keywords program synthesis; weak memory models

PLDI '17 June 19–24, 2017, Barcelona, Spain

ISBN 978-1-4503-4988-8/17/06…$15.00

DOI: http://dx.doi.org/10.1145/3062341.3062353

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 imprecise definitions make reasoning about correctness difficult for both developers and tool builders. Researchers have therefore argued for formalizing memory models [49], and have recently created formal models for common architectures, including x86 [40] and PowerPC [30]. But each such formalization required several person-years of effort and several revisions (e.g., [5, 6, 35, 38, 39]).

These formalization efforts have been aided by tools for *verification* and *comparison* of memory models. Verification tools check whether a model allows a litmus test [6, 36, 45], while comparison tools synthesize litmus tests on which two models disagree [28, 47]. These tools provide verification and comparison queries for memory models within a given *axiomatic framework* (e.g., [8]). The framework supplies basic axioms that every memory model must follow, expressed as first-order constraints on relations that order memory events (such as reads and writes). The tools then answer queries about specific models from the framework with respect to a given litmus test (in the case of verification) or a space of litmus tests (in the case of comparison). But no existing tools can answer queries about the framework itself, e.g., whether it contains a memory model that satisfies a set of litmus tests.

This paper proposes using *program synthesis* to answer novel queries about memory models and their frameworks. The core idea behind our proposal is a *framework sketch*, which describes a class of memory models with a syntactic template. The template consists of a set of axioms with *holes* [41] (i.e., missing expressions) whose completion defines a memory model from the target class. The sketch is provided by the memory model designer and can capture domain-specific insights and assumptions, such as the use of scopes [9] to describe GPU memory models. Given a framework sketch, synthesis-based tools can answer a variety of new queries about memory models. For example, they can search for a memory model specification that satisfies a set of example litmus tests, automating a tedious development cycle currently performed by hand [32]. Synthesis also en-

ables more complex queries, such as determining whether a synthesized model is ambiguous by checking whether a second, semantically distinct model also explains the same example litmus tests.

We realize this proposal with MemSynth, a new system for synthesizing axiomatic specifications of memory models from framework sketches and litmus tests. MemSynth provides a language for writing framework sketches, 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 [24, 44] in Rosette [42, 43], a solver-aided host language that extends Racket [21, 37] 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 for memory models [6, 29, 45, 47]. The bounded version of the logic is decidable by reduction to boolean satisfiability, and existing relational solvers [24, 33, 44] 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 compiles the semantics of its guest languages to efficiently-solvable SMT constraints. MemSynth layers a specialized synthesis algorithm on top of this evaluator, scaling to produce specifications of real memory models in seconds.

The MemSynth synthesizer takes as input a framework sketch and a set of litmus tests. The sketch is a formula in relational logic with missing expressions (holes) over relations defined by the framework (e.g., happens-before [25]). Given these inputs, MemSynth completes the sketch by solving a synthesis query of the form $\exists \phi_M \in F. \bigwedge_{T \in \mathcal{T}_P} \exists I.\ [\![(\!| U_T; \mathcal{V}_T; \phi_M |\!)]\!] I \wedge \bigwedge_{T \in \mathcal{T}_N} \forall I.\ \neg [\![(\!| U_T; \mathcal{V}_T; \phi_M |\!)]\!] I$ where F is a framework sketch, 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 generic relational solvers [33] 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 [44]) that enable MemSynth's embedded engine to employ aggressive optimizations based on the structure of litmus tests and framework sketches.

But MemSynth's novel design offers advantages that go beyond scalable synthesis. Being embedded in Rosette, MemSynth provides a platform for rapid development of high-performance tools for reasoning about memory models. For example, we use MemSynth to implement the verification query in five lines of code, obtaining a tool that outperforms dedicated relational solvers [24, 33] and is comparable to existing hand-crafted verifiers [6, 29]. We also implement a novel *ambiguity query* for identifying ambiguities in the set of litmus tests with respect to a framework sketch. The ambiguity 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 using a framework sketch based on work by Alglave et al. [6]. Given this sketch, MemSynth synthesizes a specification for the notoriously relaxed PowerPC architecture from 768 litmus tests in under 12 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 [23]. In both cases, our ambiguity query finds that the given litmus tests do not uniquely define their intended memory model—several other models are also consistent with the set of tests. MemSynth synthesizes sets of missing tests from the validation suite of Alglave et al. [6] (for PowerPC) and the Intel manual (for x86) that resolve these ambiguities.

We evaluate MemSynth as a tool-building platform by reproducing results from an existing paper [29] on comparing memory models. In the process, we automatically synthesize a repair for a discrepancy between our framework sketch and the original work—due to a misprint in the paper—which we were unable to fix by hand. The repaired sketch of the paper's framework was developed in two days and achieves the same performance as the existing tool.

In summary, this paper makes the following contributions:

- We introduce MemSynth, a platform for automatically synthesizing memory model specifications from framework sketches and litmus tests. MemSynth's novel design, as an embedded logic in a solver-aided host language, enables it to synthesize complex memory models such as PowerPC from large sets of examples.
- We demonstrate that MemSynth can answer advanced queries about memory model specifications, such as ambiguity, that can aid memory model designers in refining their specifications. 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. Section 2 introduces the MemSynth language for relational logic. Section 3 defines framework sketches and litmus tests. Section 4 presents the queries that MemSynth can answer, and Section 5 describes the algorithms to answer these queries. Section 6 shows three case studies using MemSynth, including synthesizing and refining a specification of PowerPC and identifying ambiguities in x86 documentation. Section 7 describes related work, and Section 8 concludes.

2. MemSynth Language

MemSynth is a language and engine for automated reasoning about memory models. The language extends bounded relational logic [24, 44] with *expression holes*, which enable *sketching* of memory model frameworks. Thanks to its expressive underlying logic, MemSynth can host many existing frameworks for reasoning about classes of memory models. This section reviews the syntax and semantics of relational logic, and presents our extensions for synthesis problems.

2.1 Bounded Relational Logic

Relational logic [24] 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 memory model frameworks [6, 29, 45, 48] are expressed as first-order constraints on relations that order memory events. MemSynth is based on a new embedding of bounded relational logic [44] in the Rosette solver-aided language [42, 43], which extends Racket [21] with support for verification and synthesis. This embedding includes an explicit construct for sketching, and its engine offers optimizations for answering (satisfiability) queries about memory models orders of magnitude faster than general-purpose relational solvers [24, 33].

Syntax. Bounded relational logic (Figure 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.

Semantics. 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 [44], 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 Section 3.2, 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 does not satisfy $(\!|U;p,q;q.q \text{ in } q|\!)$.

2.2 Expression Holes

To support synthesis, we extend relational logic with *expression holes*, which define the search space for a synthesis query to explore [41]. An expression hole $\mathcal{G}(N,T,d,k)$ is a relational expression that evaluates non-deterministically to one of a finite set of concrete expressions. The set contains all expressions of arity k that can be produced with derivation trees of depth d from a context-free grammar with non-terminals N and terminals T, where the non-terminals are drawn from expression operators in relational logic. Expression holes are a key difference between MemSynth and other relational logic languages such as Kodkod [44] and Alloy* [33], which would 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 $X+X$, $\mathcal{G}(N,T,2,1)$ additionally contains $X+X+X$ and $X+X+X+X$, and $\mathcal{G}(N,T,1,2)$ contains Y, $Y+Y$, and $X \to X$.

2.3 Relational DSL

MemSynth is implemented (Figure 2) as a domain-specific language (DSL) in Rosette [42, 43]. The MemSynth interpreter INTERPRET(p,I) takes as input relational syntax p and an interpretation I, and executes the semantics in Figure 1. The interpreter represents relations of arity k in the standard way [24, 44], 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, 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 interpre-

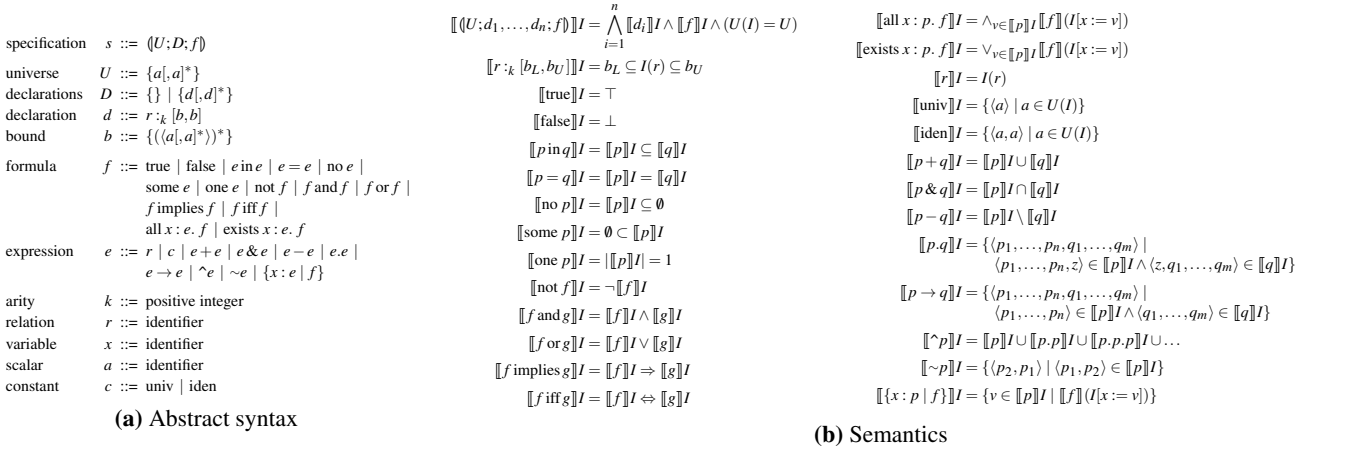


Figure 1. The syntax and semantics of bounded relational logic [44].

INTERPRET(p, I)
Inputs: Relational syntax p; interpretation I
Output: Encoding of the semantics of p (according to Figure 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.

tation $I \models s$, MemSynth simply evaluates 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 Figure 2). The result of 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 [19]. 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 [33] as we show in Section 6.

3. Framework Sketches

Like existing tools [8, 45, 47], MemSynth specifies memory models as axioms in relational logic that constrain the set of executions allowed for a concurrent program. But unlike existing tools, which take a complete memory model specification as input, MemSynth accepts a sketched specification in the form of a *framework sketch* provided by a memory model designer. Framework sketches are at the core of MemSynth's flexibility as a tool-building platform.

This section defines framework sketches in terms of relational logic, and introduces F_{Alglave}, an example sketch of the Alglave et al. [6] framework for memory models. We use F_{Alglave} to illustrate the automated reasoning queries (Section 4) supported by our engine (Section 5), and to demonstrate their scalability (Section 6).

3.1 Definitions

A framework sketch (Definition 1) consists of two components: a set of axioms ϕ that contain expression holes (Section 2.2), and a function ENC that encodes the syntax and semantics of a litmus test in bounded relational logic. Concurrent programs (without unbounded control flow) have a natural representation [45] in our logic: a program defines a finite universe of discourse U and a set of relations $\mathcal{V} = S \cup E$ over U that encode the test's syntax (S) and its candidate executions (E). For example, S often includes unary relations for each type of instruction in a concurrent program P (such as Read and Write), as well as the *program-order* relation po that relates instructions in the same thread. The relations in E encode possible executions of P by, for example, defining a *happens-before* ordering [25] on the instructions in P (see [6, 28, 45]). The holes in the axioms ϕ are specified over the relations $S \cup E$ emitted by ENC. A framework sketch (ϕ, ENC) thus defines a class of memory models (Definition 2) with respect to a framework-specific definition of a litmus test.

Definition 1 (Framework sketch)**.** *A* framework sketch *is a pair* (ϕ, ENC)*, where:*

- ϕ *is a relational formula containing zero or more expression holes. The relations in* ϕ *are partitioned into sets S and E, where relations in S characterize the syntax of a concurrent program, and relations in E characterize an execution of that program.*
- ENC *is a function that takes as input a concurrent program P, and returns a pair* $(U, \mathcal{V})$ *of a relational universe U and set of relation declarations* $\mathcal{V}$*, such that every relation in* $S \cup E$ *is bound by* $\mathcal{V}$*.*

We say that a framework sketch allows *a concurrent program P if there exists an interpretation I such that $I \models (\!| U; \mathcal{V}; \phi |\!)$, where $(U, \mathcal{V}) = \text{ENC}(P)$. Otherwise, the sketch* forbids *P.*

Definition 2 (Memory model)**.** *A* memory model *M is a framework sketch (ϕ_M, ENC) in which ϕ_M contains no expression holes. We say that M* belongs *to a framework sketch $F = (\phi_F, \text{ENC})$, written $M \in F$, if and only if ϕ_M can be obtained from ϕ_F by substituting every hole $h = \mathcal{G}(N, T, d, k)$ in ϕ_F with a relational expression $e \in h$.*

By not mandating a specific definition of a concurrent program P, MemSynth allows framework sketches to define instruction sets and other program structures (e.g., control flow) that are relevant to a given class of memory models. For example, a language memory model would include release/acquire operations [11]; an architectural model, such as F_{Alglave} below (Section 3.2), would include fences for a specific architecture (e.g., `mfence` on x86 or `sync` and `lwsync` on PowerPC); and a GPU memory model would include scopes on fence operations [9]. MemSynth requires only that the framework sketch separate the relations S defining a litmus test from the relations E defining an execution of that test, so that it can support a variety of automated reasoning queries in a framework-agnostic way (described in Section 4).

3.2 $\mathbf{F_{Alglave}}$

This section illustrates a framework sketch based on an axiomatic framework by Alglave et al. [6]. We call the corresponding framework sketch F_{Alglave}. We use F_{Alglave} for most of our experiments, although in Section 6.2 we construct a second framework sketch based on a different framework.

3.2.1 Litmus Tests

In F_{Alglave}, a *litmus test* is a small multi-threaded program together with a candidate outcome, expressed as a constraint on the program's final state. For example, the Intel Software Developer's Manual [23] includes the following litmus test to illustrate a surprising behavior allowed by the x86 memory model, where reads may be reordered with earlier writes:

Test `x86/3`

Thread 1	**Thread 2**
1: $\mathsf{X} \leftarrow 1$	3: $\mathsf{Y} \leftarrow 1$
2: $\mathsf{r1} \leftarrow \mathsf{Y}$	4: $\mathsf{r2} \leftarrow \mathsf{X}$

Outcome: $\mathsf{r1} = 0 \wedge \mathsf{r2} = 0$
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 $\mathsf{X} \leftarrow 1$ means that 1 is written to the memory location X, and $\mathsf{r1} \leftarrow \mathsf{Y}$ 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).

Given a litmus test, F_{Alglave}'s encoding function ENC_A constructs a universe of *memory events* (i.e., read, write, and fence instructions), *locations*, *threads*, and *values* that appear in the test (Definition 3). It also constructs the relations S that encode the syntax of the test, including, for example, unary relations (such as Read) for the types of each instruction of the test. The contents of the syntax relations S are known statically (i.e., the values observed by each read are known from the test's outcome predicate, and we do not handle control dependencies) and extracted automatically from the test.

Definition 3 (Litmus test)**.** *A* litmus test *in F_{Alglave} is a small concurrent program together with a postcondition constraint. Given a litmus test T, F_{Alglave}'s encoding function $\text{ENC}_A(T)$ returns a finite universe of discourse U and a set of relation declarations S over U, defined as follows:*

- *Every relation declaration in S 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 test `x86/3` above. ENC_A(`x86/3`) 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 $L = \{\mathsf{X}, \mathsf{Y}\}$, two threads $T = \{t_1, t_2\}$, and two values $V = \{0, 1\}$. Its relations $\mathcal{V}$ are:

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

3.2.2 Executions

F_{Alglave} uses two relations, rf and ws, to define the execution of a litmus test (Definition 4). 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.

$$\mathsf{ppo}_{SC} \triangleq \mathsf{po} \qquad \mathsf{grf}_{SC} \triangleq \mathsf{rf} \qquad \mathsf{fences}_{SC} \triangleq \emptyset$$

(a) Sequential consistency

$$\mathsf{ppo}_{TSO} \triangleq \mathsf{po} - (\mathsf{Write} \to \mathsf{Read}) \qquad \mathsf{grf}_{TSO} \triangleq \mathsf{rf} - (\mathsf{thd}.\sim\mathsf{thd}) \qquad \mathsf{fences}_{TSO} \triangleq \emptyset$$

(b) Total store order

Figure 3. Examples of common memory models defined by hand in the F_{Alglave} framework.

The encoding function ENC_A returns $\{\mathsf{rf}, \mathsf{ws}\}$ as the set of execution relations E for a litmus test T, and it specifies bounds on their contents by automatically extracting them from T.

Definition 4 (F_{Alglave} Execution)**.** *In F_{Alglave}, 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.*

3.2.3 Memory Model

F_{Alglave} defines a memory model as a relational formula ϕ_A that constructs a happens-before order and checks its acyclicity. F_{Alglave}'s memory model definition is parametric—many 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. Figure 3 shows examples of these relations for the common sequential consistency (SC) and total store order (TSO) models. The F_{Alglave} formula ϕ_A replaces these three relations with expression holes for use in synthesis.

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 Figure 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 (Definition 4), grf specifies the edges in rf that must be globally ordered. In Figure 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.

Fences. The *fences* relation fences defines which events are ordered by a memory fence. For example, the x86 architecture has an `mfence` instruction that serializes all reads and writes issued prior to it. The TSO example in Figure 3 already includes fences in ppo, and so fences is still empty. But some relaxed memory models, such as PowerPC and ARM, also have a notion of fence *cumulativity* [22], in which fence operations create orderings between events on other threads; F_{Alglave} uses fences to model cumulativity. The rules for cumulativity are subtle, but MemSynth correctly synthesizes them for PowerPC in under 12 seconds, as we show in Section 6.1.

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

(a) Auxiliary relations

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

(b) Axioms

Figure 4. The axioms of the F_{Alglave} framework extend those of Alglave et al. [6], with changes to remove initialization write events and support outcomes for memory locations.

Axioms Given the definitions of ppo, grf, and fences, F_{Alglave} uses the axioms in Figure 4 to specify the framework sketch's formula ϕ_A. The axioms follow Alglave et al. [6], with two changes for better solving performance. First, we omit initialization write events (events that initialize each memory location to 0) in favor of an Init axiom. 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 [8].

The first five axioms in Figure 4(b) define well-formedness of an execution E. The Execution axiom applies the rules in Definition 4 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*, F_{Alglave} constructs a *global happens-before* order ghb reflecting the orderings between events induced by the memory model. The Valid axiom allows a test 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, F_{Alglave}'s framework sketch (ϕ_A, ENC_A) defines $\phi_A \triangleq \mathsf{Valid}$.

4. Memory Model Queries

MemSynth is designed to efficiently answer four queries about memory models from a given framework sketch:

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

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

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

Ambiguity 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. Section 5 shows how to implement these queries to scale to hundreds of litmus tests and large specifications.

4.1 Verification

The verification query, determining whether a memory model allows a litmus test, is well-studied in the literature [6, 26, 29, 45, 48]. Given a litmus test T and memory model $M = (\phi, \textsc{Enc})$ (Definition 2), the verification query checks satisfiability of the formula

$$\exists I. [\![(\!| U; \mathcal{V}; \phi |\!)]\!] I$$

where $(U, \mathcal{V}) = \textsc{Enc}(T)$. If this formula is satisfiable, then M allows the test T (Definition 1). Otherwise, M forbids T. The verification query involves a straightforward satisfiability check that can be discharged with any relational solver, including MemSynth.

4.2 Synthesis

The synthesis query searches a framework sketch for 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, a set $\mathcal{T}_N$ of tests that should be forbidden, and a framework sketch $F = (\phi, \textsc{Enc})$, the synthesis task is to find a memory model $(\phi_M, \textsc{Enc}) \in F$ that allows all tests in $\mathcal{T}_P$ and forbids all tests in $\mathcal{T}_N$. This query amounts to solving the formula

$$\begin{aligned} \exists (\phi_M, \textsc{Enc}) \in F. \bigwedge_{T \in \mathcal{T}_P} \exists I. [\![(\!| U_T; \mathcal{V}_T; \phi_M |\!)]\!] I \\ \wedge \bigwedge_{T \in \mathcal{T}_N} \forall I. \neg [\![(\!| U_T; \mathcal{V}_T; \phi_M |\!)]\!] I \end{aligned} \tag{1}$$

where $(U_T, \mathcal{V}_T) = \textsc{Enc}(T)$.

The synthesis query involves higher-order universal quantification over the non-constant relations in $\mathcal{V}_T$ for forbidden tests $\mathcal{T}_N$ The recent Alloy* solver [33] 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 holes and bounds on the contents of declared relations. These limitations motivated our embedding of bounded relational logic in Rosette (Section 2). In Section 5.2, we present an algorithm for solving synthesis queries that scales to complex framework sketches and many litmus tests.

4.3 Equivalence

MemSynth can compare two memory models M_A and M_B from a framework F 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 [28, 47], 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 (Definition 5), in which some syntax relations S are not constant (in contrast to, e.g., Definition 3). A symbolic litmus test thus defines a set of concurrent programs rather than only one such program.

Definition 5 (Symbolic litmus test)**.** *A* symbolic litmus test $T_S = (\!| U; \mathcal{V}; f |\!)$ *for a framework sketch* $F = (\phi, \textsc{Enc})$ *is a relational specification in which*

- $\mathcal{V}$ *binds the relations* $S \cup E$ *in* ϕ *(as in Definition 1).*
- *The formula* f *is a* well-formedness *predicate for the litmus test, in which the only relations are those in* S*.*

Given a symbolic litmus test T_S and two memory models $M_A = (\phi_A, \textsc{Enc})$ and $M_B = (\phi_B, \textsc{Enc})$, the equivalence query solves for a distinguishing litmus test by checking the satisfiability of two formulas:

$$\begin{aligned} \exists I_T. [\![T_S]\!] I_T \wedge \exists I. [\![(\!| U; \mathcal{V}; \phi_A |\!)]\!] (I_T \cup I) \\ \wedge \forall I. \neg [\![(\!| U; \mathcal{V}; \phi_B |\!)]\!] (I_T \cup I) \end{aligned}$$

to find a test 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

$$\begin{aligned} \exists I_T. [\![T_S]\!] I_T \wedge \exists I. [\![(\!| U; \mathcal{V}; \phi_B |\!)]\!] (I_T \cup I) \\ \wedge \forall I. \neg [\![(\!| U; \mathcal{V}; \phi_A |\!)]\!] (I_T \cup I) \end{aligned}$$

for a test on which M_A is stronger than M_B. The symbolic litmus test $T_S = (\!| U; \mathcal{V}; f |\!)$ includes a well-formedness predicate f, a relational formula that ensures the resulting test is a syntactically valid program. If either formula is satisfiable, then $T_D = \textsc{Eval}(T_S, I_T)$ is a litmus test that distinguishes the two models M_A and M_B.[1] 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.

4.4 Ambiguity

The ambiguity query checks whether a memory model M is the only one within a framework sketch that gives the desired

[1] $\textsc{Eval}(T_S, I_T)$ substitutes each variable v in T_S with the value $I(v)$.

outcomes on 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_D such that M_S and M disagree on T_D 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 semantically distinct memory models that both explain the input tests $\mathcal{T}_P \cup \mathcal{T}_N$.

Given a memory model $M = (\phi_M, \text{ENC})$, a framework sketch $F = (\phi, \text{ENC})$, a symbolic litmus test $T_S = (\!|U_S; \mathcal{V}_S; f|\!)$, and sets of allowed and forbidden tests $\mathcal{T}_P$ and $\mathcal{T}_N$, detecting ambiguity involves checking the satisfiability of a formula that combines synthesis and equivalence:

$$\begin{aligned}\exists I_T.\exists(\phi_S, \text{ENC}) \in F.\ [\![T_S]\!]I_T \wedge &\bigwedge_{T\in\mathcal{T}_P} \exists I.\ [\![(\!|U_T; \mathcal{V}_T; \phi_S|\!)]\!]I \\ &\wedge \bigwedge_{T\in\mathcal{T}_N} \forall I.\ \neg[\![(\!|U_T; \mathcal{V}_T; \phi_S|\!)]\!]I \\ &\wedge \exists I.[\![(\!|U_S; \mathcal{V}_S; \phi_S|\!)]\!](I_T \cup I) \\ &\wedge \forall I.\neg[\![(\!|U_M; \mathcal{V}_M; \phi_M|\!)]\!](I_T \cup I)\end{aligned}$$

where $(U_T, \mathcal{V}_T) = \text{ENC}(T)$, 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 $M_S = (\phi_S, \text{ENC})$ 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 finite search space defined by the framework sketch F, and the finite search space for the symbolic litmus test T_S.

The ambiguity query identifies missing tests from the input sets, 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 Section 3.2, and $\mathcal{T}_N$ to be empty, then many distinct memory models within F_{Alglave} produce the desired outcomes (TSO, RMO, PowerPC, etc.). If we take M to be one such model, the ambiguity 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.

5. Reasoning Engine

This section presents MemSynth's engine for answering the queries in Section 4. We show the algorithms to implement these queries, and describe key optimizations to make them scale to real-world memory models.

```
1 function VERIFY(M = (φ_M, ENC), T)
2     (U, V) ← ENC(T)
3     I ← INSTANTIATE(V)
4     φ ← INTERPRET(φ_M, I)
5     return SOLVE(φ) = SAT
```

Figure 5. MemSynth's verification procedure VERIFY takes as input a memory model M and litmus test T and determines whether M allows T.

```
1 function ENC_A(T)
2     (U, V) ← ENCSYNTAX(T)                    ▷ Encode relations in Definition 3
3     I ← INSTANTIATE(V)                       ▷ Make an interpretation from V
4     B_u^rf ← INTERPRET((Write → Read) & (loc.~loc) & (val.~val), I)
5     B_u^ws ← INTERPRET((Write → Write) & (loc.~loc), I)        ▷ Figure 4
6     return (U, V ∪ {rf :2 [∅, B_u^rf], ws :2 [∅, B_u^ws]})
```

Figure 6. The ENC_A procedure computes relational bounds for an execution E in the F_{Alglave} framework.

5.1 Verification

The verification query (Section 4.1) determines whether a memory model M allows a litmus test T. The VERIFY procedure in Figure 5 takes as input a memory model $M = (\phi, \text{ENC})$ and litmus test T, and returns true iff M allows T. The VERIFY procedure first encodes the litmus test as a finite universe U and set of relation declarations $\mathcal{V}$ using the memory model's ENC function (Definition 1). Given these bounds, it then checks the satisfiability of the relational specification $(\!|U; \mathcal{V}; \phi|\!)$. The implementation of VERIFY is only four lines of code, demonstrating the utility of our relational DSL for reasoning about memory models.

Bounds Compaction. Figure 6 shows an example implementation of the ENC function. The ENC_A procedure computes bounds for the relations in a F_{Alglave} execution (Definition 4). 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 the solver query. For F_{Alglave}, an execution consists of two relations rf and ws that specify a reads-from and write serialization order, respectively. ENC_A computes upper bounds for each relation from the Execution axiom in Figure 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 27× on the PowerPC tests discussed in Section 6.1.

5.2 Synthesis

The synthesis query (Section 4.2) generates a memory model that gives the desired outcomes on a set of litmus tests. The space of candidate solutions is defined by a framework sketch $F = (\phi, \text{ENC})$ (Definition 1), which contains expression holes that define a candidate space of memory models.

Our synthesis procedure, SYNTHESIZE (Figure 7), takes as input a framework sketch $F = (\phi, \text{ENC})$, a set of allowed

```
1  function SYNTHESIZE(F = (φ, ENC), T_P, T_N)
2      S ← new IncrementalSMTSolver()
3      T_U ← {}                                   ▷ Set of used tests
4      φ_M ← false                                ▷ Model that forbids all outcomes
5      T ← NEXTTEST(φ_M, T_P, T_N, T_U)           ▷ Choose an initial test
6      while T ≠ ⊥ do
7          ADDTEST(S, F, 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         φ_M ← EVAL(φ, I_b)                     ▷ Use I_b to fill the holes in φ
13         T ← NEXTTEST(φ_M, ENC, T_P, T_N, T_U)  ▷ Choose the next test
14     return (φ_M, ENC)                          ▷ M gives the expected outcome on all tests in T_P ∪ T_N
```

(a) Main synthesis routine

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

(b) Test evaluation

```
1  function NEXTTEST(φ_M, ENC, T_P, T_N, T_U)
2      for T ∈ (T_P ∪ T_N) \ T_U do               ▷ Iterate over unused tests
3          if VERIFY((φ_M, ENC), 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 7. 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.

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 [19]. Because MemSynth represents relations as matrices of boolean values, these formulas quantify over boolean variables. We found the Z3 SMT solver [19] to be extremely effective at discharging these formulas—an average of 2–5× faster than our own specialized implementation of counterexample-guided inductive synthesis [41].

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 against the framework sketch predicate ϕ. Instead, tests are added to the synthesis query incrementally. The order in which tests are added influences synthesis performance; we use 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 Section 6.1, we show that only 16 of 768 tests were added to the query when synthesizing a model for PowerPC.

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

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

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

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

5.3 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 COMPARE(M_A, M_B, T_S) takes as input two memory models M_A and M_B, and a designer-provided symbolic litmus test T_S (Definition 5). Given these inputs, it returns either a litmus test T such that VERIFY$(M_A, T) \neq$ 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 Section 4.3 using the Z3 SMT solver (as with SYNTHESIZE), with two additional optimizations described next.

Symmetry Breaking. For most framework sketches, a naive specification of a symbolic litmus test will define a search space that contains many redundant candidate tests. For example, after checking a test T in F_{Alglave}, 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, our definition of T_S for F_{Alglave} applies lex-leader symmetry breaking [17] to rule out tests that differ only by a permutation of threads, addresses, or values, similar to existing work [28]. The well-formedness predicate f for T_S also adds assertions to rule out other uninteresting litmus tests, such as 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×, and generalize beyond F_{Alglave}.

Concretization with Metasketches. As another critical optimization, we express the symbolic litmus test T_S using a metasketch [13], which decomposes T_S into a set of *partially concretized* symbolic tests. In particular, T_S describes the set of all litmus tests with up to k threads and up to n instructions per thread. The corresponding metasketch describes the same search space using a *set* of symbolic tests of the form $T_S^{\langle k,(t_1,\ldots,t_k),(w_1,\ldots,w_k)\rangle}$, each of which encodes the space of all litmus tests with a concrete number of threads (k), instructions per thread ($t_1,\ldots,t_k$), and writes per thread ($w_1,\ldots,w_k$). For example, the set $T_S^{\langle 2,(2,3),(1,2)\rangle}$ contains all tests with two threads, with two instructions (one of which is a write) on the first thread, and three instructions (two of which are writes) on the second thread. This concretization enables each symbolic litmus test in the metasketch to use more compact bounds (e.g., the thread relation `thd` becomes

entirely concrete), which reduces the search space exponentially. Without this optimization, the equivalence queries in Section 6.3 are up to two orders of magnitude slower.

5.4 Ambiguity

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 ambiguity procedure $\textsc{Disambiguate}(M, \mathcal{T}_P, \mathcal{T}_N, F, T_S)$ takes as input a memory model M, sets of allowed tests $\mathcal{T}_P$ and forbidden tests $\mathcal{T}_N$, a framework sketch $F = (\phi, \textsc{Enc})$, and a symbolic litmus test T_S. It returns a new memory model M_S and test T_D, such that for all $T \in \mathcal{T}_P \cup \mathcal{T}_N$, $\textsc{Verify}(M,T) = \textsc{Verify}(M_S,T)$, but $\textsc{Verify}(M,T_D) \neq \textsc{Verify}(M_S,T_D)$. In other words, the set of tests $\mathcal{T}_P \cup \mathcal{T}_N$ is ambiguous, because both M and M_S satisfy every test in the set. Since the ambiguity query involves synthesizing a memory model M_S and litmus test T_D, the implementation of Disambiguate extends Synthesize (Figure 7) and benefits from the same optimizations as Compare, i.e., metasketches and symmetry breaking. Metasketches also enable solving in parallel [13], which Disambiguate exploits to gain up to 3× speedup on 8 threads in our experiments.

5.5 Discussion

Limitations. As with other tools based on syntax-guided synthesis [41], MemSynth's results are inherently bounded. Both framework sketches (for synthesis) and symbolic litmus tests (for equivalence and ambiguity) define large but finite search spaces, which MemSynth explores exhaustively with an SMT solver. While incomplete, such bounded reasoning provides useful results on real-world problems, as Section 6 shows; in most of those experiments, increasing the bounds yielded no meaningful difference in the results.

MemSynth's synthesis queries also face the potential for overfitting, like other example-based synthesis tools. The relational DSL (Section 2) reduces this risk by not including operators prone to overfitting (e.g., if-then-else expressions), and by offering control over the size of the search space for expression holes. The Compare and Disambiguate queries can also exploit metasketch support for cost functions [13] to minimize the size of the synthesized tests.

Integration. MemSynth queries read and write litmus tests in the common Herd format [8], allowing them to integrate with existing tools for memory models. MemSynth's relational DSL can also export models to Alloy* specifications, which are used by some memory model tools [45, 47]. MemSynth's relational DSL (Section 2) is similar to the `cat` language [3] for specifying memory models, and so MemSynth models could be exported for use by that toolchain. But `cat` includes fixpoint operations, while our DSL does not, so importing `cat` models into MemSynth would require more work (e.g., bounded unwinding of fixpoints [47]).

6. 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?

Methodology and Code. Experiments in this section were performed on a quad-core Intel Core i7-7700K CPU at 4.8 GHz, with 16 GB of RAM. We used Rosette [42, 43] version 2.2 and Z3 [19] version 4.5.0. Both MemSynth and our relational DSL, Ocelot, are open source and available from `http://memsynth.uwplse.org`, together with the synthesized models and tests from this section and a virtual machine artifact for reproducing the results.

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

This section uses MemSynth to synthesize specifications for the PowerPC [22] and x86 [23] memory models. The results (summarized in Figure 8) 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.

6.1.1 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 [5, 6, 30], making an automated process particularly appealing. To synthesize a specification for PowerPC, MemSynth uses a set of 768 litmus tests from Alglave et al. [4, 6], which they generated with their diy tool [7]. 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. [6] model to decide whether each test should be allowed, although we could use hardware observations instead, as discussed later.

We employ F_{Alglave} as the basis for the synthesis process. The framework sketch contains expression holes for the ppo, grf, and fences relations. All three holes use a grammar containing all relational expressions e in Figure 1 other than set comprehension and closure. For the barrier expression fences, we provide a sketch of the form $\text{fences} \triangleq \mathsf{F}_{\mathsf{Fence}} + \mathsf{F}_{\mathsf{LWFence}}$, where $\mathsf{F}_{\mathsf{Fence}}$ and $\mathsf{F}_{\mathsf{LWFence}}$ are expression holes containing Fence and LWFence, respectively, as terminals. This sketch expresses the high-level insight that PowerPC features two kinds of cumulative barriers (heavyweight `sync` fences and lightweight `lwsync` fences) that do not interact.

	Input Tests			Framework Sketch		
Arch.	$\lvert\mathcal{T}_P\rvert$	$\lvert\mathcal{T}_N\rvert$	Time	ppo/grf Depth	fences Depth	State Space
PPC	163	605	12 s	4	4	2^{1406}
x86	2	8	2 s	4	0	2^{624}

(a) Synthesis results

			Symbolic Litmus Test		
Arch.	New Tests	Time	Num. Threads	Num. Events	State Space
PPC	9	110 min	2–4	2–6	2^{165}
x86	4	67 min	2–4	2–6	2^{114}

(b) Ambiguity results

Figure 8. Results of real-world memory model synthesis and ambiguity experiments for PowerPC and x86. We describe the framework sketches and symbolic litmus tests both in terms of their parameters (e.g., expression hole depth) and the number of candidate solutions they contain (i.e., their state space). The ambiguity results (b) for a given architecture use the same framework sketch as the synthesis results (a) for that architecture.

Synthesis. MemSynth synthesizes a model, which we call PPC_0, that agrees with Alglave et al.'s hand-written model on all 768 tests. The synthesis takes 12 seconds, and due to its heuristics for test ordering, the incremental synthesis algorithm (Figure 7) uses only 16 of the 768 tests.

Ambiguity. 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 DISAMBIGUATE query (Section 4.4) 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 9 new tests to add to the set. The tests deal with the semantics of PowerPC barriers; for example:

Test `ppc/ambig/3`

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

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

After adding the 9 tests, the new synthesized model PPC_1 is equivalent to the Alglave et al. [6] 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 [6]: 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. [6] as an oracle, we could instead determine litmus test outcomes by manually consulting documentation or by hardware experiments. For example, Alglave et al. [4] 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_H in 13 seconds. The resulting model is not equivalent to PPC_0 (MemSynth synthesizes a distinguishing test with its equivalence query in 6 seconds) because some allowed outcomes were not observed on the hardware.

```
(define (hole depth arity non-terms terms)
  ...)  ; Expression hole (Section 2.2)

(define (F_Alglave ppo grf fences)
  ...)  ; Axioms from Figure 4

; Common components of memory model specifications
(define (SameAddr X) (& (-> X X) (join loc (~ loc))))
(define rfi (& rf (join thd (~ thd))))
(define rfe (- rf (join thd (~ thd))))

; Expression holes for F_Alglave model (Section 3.2)
(define ppo
  (hole 4 2 (list + - -> & SameAddr)
            (list po dep Event Read Write Fence Atomic)))
(define grf (hole 4 2 (list + - -> & SameAddr)
                      (list rf rfi rfe none univ)))
; x86 fences are not cumulative
(define fences (-> none none))

; Final sketch
(define x86-sketch (F_Alglave ppo grf fences))
```

(a) Framework sketch F_{Alglave}

```
; Before disambiguation
(define ppo_0
  (& po (- (-> Event (+ Write Read))
           (-> (- Write Atomic) Read))))
(define grf_0 (- rf (join thd (~ thd))))
(define TSO_0 (F_Alglave ppo_0 grf_0 fences))

; After resolving 4 ambiguities
(define ppo_4 (- po (-> (- Write Atomic) Read)))
(define grf_4 (- rf (join thd (~ thd))))
(define TSO_4 (F_Alglave ppo_4 grf_4 fences))
```

(b) Synthesized models TSO_0 and TSO_4

Figure 9. The framework sketch F_{Alglave} for synthesizing a memory model for the x86 architecture (a), and synthesized models TSO_0 and TSO_4 before and after resolving ambiguities (b). The expression holes for ppo and grf define a search space of size 2^{624}, as described in Figure 8. The fences relation is empty because x86 fences are not cumulative.

6.1.2 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 [23] 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 [14, 15, 38, 40].

We used MemSynth to synthesize a specification of the x86 memory model. To do so, we extended F_{Alglave} with support for atomic operations (adding a new unary Atomic

relation to Definition 3 to model x86's `xchg` instruction) and the `mfence` full memory fence (populating the Fence relation in Definition 3). MemSynth synthesizes a formalization TSO_0 that is correct on the Intel manual's 10 litmus tests in under two seconds. Figure 9 shows the framework sketch F_{Alglave} and the synthesized model TSO_0.

Ambiguity. But MemSynth's DISAMBIGUATE query (Section 4.4) determines that another weaker memory model, TSO_1, also satisfies all 10 tests, while disagreeing with TSO_0 on a new distinguishing test:

Test `x86/ambig/1`

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

Initially: `r3` = 1
Outcome: `r1` = 1 ∧ `r2` = 0

This test is a variant of the manual's example 8-1 [23], but with an atomic exchange instead of a plain write to `A`. The documentation indicates that x86 should forbid this outcome, as TSO_0 does but TSO_1 does not.

Repeating the ambiguity query after adding `x86/ambig/1` finds 3 more distinguishing tests that further examine the semantics of atomic operations and `mfence`. According to the documentation, the resulting tests should also be forbidden. After adding these tests to the synthesis process, MemSynth is able to prove that a new synthesized model TSO_4 is unique, up to the bounds in Figure 8 on the size of the model specification and distinguishing litmus test.

The TSO_4 model in Figure 9(b) correctly captures the intent of the x86 TSO model, and is similar to both Figure 3 and Alglave [2]. The synthesized $\texttt{ppo}_4$ allows writes to be reordered past later reads, while $\texttt{grf}_4$ allows a thread to read its own writes early. TSO_4 also models the semantics of the `xchg` and `mfence` instructions, both of which are included in $\texttt{ppo}_4$ and so prevent reordering.[2] We further validated the synthesized model by comparing it to the x86-TSO model of Sewell et al. [40] on the 24 litmus tests in their paper [34]; TSO_4 agrees with x86-TSO on all such tests. The 4 distinguishing tests synthesized by MemSynth are either single-fenced variants of Sewell et al.'s `amd5` litmus test, or `xchg`-based variants of other Intel tests, similar to their `n8` test.

Potential Redundancy. In the paper on their earlier x86-CC formalization of the x86 memory model, Sarkar et al. [38] 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." [23]

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.

[2] We model Atomic as a subset of Write, so the expression `(- Write Atomic)` allows only plain writes to be reordered.

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

While previous sections build on the F_{Alglave} framework sketch, based on the Alglave et al. [6] framework, MemSynth's engine generalizes to other framework sketches. In this section, we present F_{MH}, a framework sketch constructed from a framework developed by Mador-Haim et al. [28, 29]. The implementation took only two 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.

6.2.1 The F_{MH} Framework

Mador-Haim et al.'s memory model framework [28, 29] 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.

6.2.2 Repairing the Framework

After implementing F_{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`

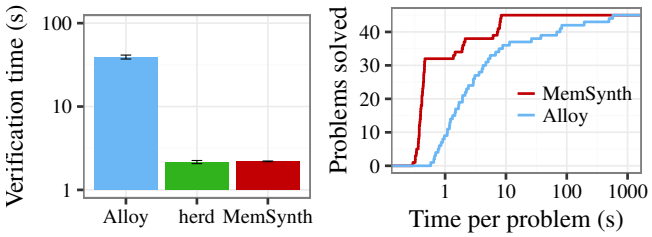
Outcome: `r1` = 2 ∧ `r2` = 0

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 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.



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

Figure 10. Performance comparisons between MemSynth and existing tools for (a) verification and (b) equivalence.

However, the paper's authors were unable to provide their implementation for us to compare against [10].

We first tried to fix the axiom by hand, but despite several attempts, a correct fix eluded us: our closest results identified 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 $(\sim \mathsf{po})$ & hb", where hb is the happens-before relation for an execution. To repair the axiom, we replaced $\sim$po with an expression hole of depth 3, and synthesized a completion that gave the correct outcomes on the 9 litmus tests from the original paper on both TSO and RMO memory models. We were able to synthesize the following repair in 4 seconds:

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

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.

6.3 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. Figure 10(a) shows the time for MemSynth, Alloy (`v4.2_2015-02-22`) [24, 44], and herd (v7.43) [8] to verify 768 PowerPC litmus tests from Section 6.1. The Alloy results use the PPC_1 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* (v0.2) [33] to perform a pairwise comparison of 10 different synthesized PowerPC models. Both MemSynth and Alloy* used a symbolic litmus test with up to 2 threads and 6 instructions. Figure 10(b) shows that MemSynth outperforms Alloy* on most of these queries: MemSynth can solve 3× more queries in under one second, and the hardest problem takes 8 s for MemSynth versus 10 min for Alloy*. With symmetry breaking and concretization (which cause the large steps in the MemSynth line in Figure 10(b)) disabled, MemSynth could not solve any of the comparisons in under an hour.

Synthesis. The synthesis query (Section 4.2) requires higher-order quantification, and so we compared MemSynth to Alloy* [33]. Because Alloy* does not support expression holes (Section 2), we designed a framework 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 $\mathcal{M} = \{\text{SC}, \text{TSO}, \text{PSO}\}$, MemSynth still returns in under a second, but Alloy* times out after one hour. This result suggests Alloy* would not be able to synthesize models from complex framework sketches.

7. 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 [46] and Alpha [16]), and so this task has fallen to researchers. A notable success is the x86-TSO model [40], which formalizes the memory model of the x86 architecture. This model was refined through several papers [35, 38], which revealed ambiguities in the x86 documentation. In Section 6.1.2, MemSynth's DISAMBIGUATE query automatically identified more such ambiguities.

Another effort has developed several formalizations of the PowerPC architecture [5, 6, 8, 30, 39]. 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 [22] required detailed experimentation to resolve. The PowerPC formalization effort also developed a suite of memory model experimentation tools, which we use in Section 6.1 and Section 6.3.

Formalization efforts have also brought clarity to emerging programming language memory models, particularly C11 and C++11 [11, 12]. 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 [31]) have already been formalized [45] in bounded relational logic. We therefore believe MemSynth could also be effective for language models, with appropriate design of a framework sketch.

Frameworks. Recent work has developed generic memory model frameworks that can be instantiated with different architectures. The Nemos framework [48] 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, 6, 8] developed an axiomatic framework for microprocessor memory models. It admits models for complex architectures such as PowerPC, and is the basis for our F_{Alglave} framework sketch (Section 3.2) and most experiments in Section 6. Mador-Haim et al. [29] developed a framework for store-atomic memory models, which we implement in Section 6.2. 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. [6] address this problem for PowerPC with a specification of the platform's barrier semantics, including cumulativity; we automatically synthesize this specification in Section 6.1. Another common application is verification of concurrent code under relaxed memory models, and several tools have been developed for this purpose (e.g., [18, 20]). All of them rely on formal specifications of memory models that can be synthesized with MemSynth.

MemSAT [45] is an automated tool that implements the verification query of Section 4.1 for axiomatic memory model specifications. MemSAT found several discrepancies in the formalization of the Java Memory Model [31]. MemSynth is similar to MemSAT in its use of relational logic, but focuses on hardware models and offers richer automated reasoning queries including synthesis. Wickerson et al. [47] use Alloy* [33] 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 ambiguity queries, and it is unclear how to adapt their quantifier elimination strategy ("deadness") to specification synthesis. Lustig et al. [27] use Alloy [24] to synthesize suites of litmus tests that examine a set of pre-defined memory ordering relaxations, which together compose a design space we could use as a framework sketch.

8. Conclusion

This paper presented MemSynth, a synthesis-aided system for reasoning about axiomatic specifications of memory consistency models. As the first of a new class of memory model tools, MemSynth can synthesize memory model formalizations based on a framework sketch provided by a designer. MemSynth's expressive specification language builds on an optimized bounded relational logic engine, which serves as a platform for developing novel automated reasoning queries. We showed that MemSynth can synthesize specifications for complex architectures, refine those specifications by identifying ambiguities, and support rapid development of 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.

Acknowledgments

We thank Dan Grossman, Xi Wang, Luis Ceze, John Wickerson, our shepherd Swarat Chaudhuri, and the anonymous reviewers for their feedback on this work. This work was supported in part by DARPA under contract FA8750-16-2-0032.

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. Modeling of Architectures. In *Advanced Lectures of the 15th International School on Formal Methods*, 2015.

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

[5] 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.

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

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

[8] 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.

[9] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson. GPU concurrency: Weak behaviors and programming assumptions. In *ASPLOS*, 2015.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[25] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), 1978.

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

[27] D. Lustig, A. Wright, A. Papakonstantinou, and O. Giroux. Automated synthesis of comprehensive memory model litmus test suites. In *ASPLOS*, 2017.

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

[29] 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.

[30] 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.

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

[32] P. E. McKenney. A Formal Model of Linux-Kernel Memory Ordering. Linux Plumbers Conference, 2016.

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

[34] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO (extended version). Technical Report UCAM-CL-TR-745, University of Cambridge, 2009.

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

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

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

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

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

[40] P. Sewell, S. Sarkar, S. Owens, F. Zappa 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.

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

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

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

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

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

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

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

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

[49] 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.