MemSAT: Checking Axiomatic Specifications of Memory Models

Emina Torlak Mandana Vaziri Julian Dolby
IBM T. J. Watson Research Center, Hawthorne, NY, USA
{etorlak, mvaziri, dolby}@us.ibm.com

Abstract

Memory models are hard to reason about due to their complexity, which stems from the need to strike a balance between ease-of-programming and allowing compiler and hardware optimizations. In this paper, we present an automated tool, MEMSAT, that helps in debugging and reasoning about memory models. Given an axiomatic specification of a memory model and a multi-threaded test program containing assertions, MEMSAT outputs a trace of the program in which both the assertions and the memory model axioms are satisfied, if one can be found. The tool is fully automatic and is based on a SAT solver. If it cannot find a trace, it outputs a minimal subset of the memory model and program constraints that are unsatisfiable. We used MEMSAT to check several existing memory models against their published test cases, including the current Java Memory Model by Manson et al. and a revised version of it by Sevcik and Aspinall. We found subtle discrepancies between what was expected and the actual results of test programs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Formal Methods, Model Checking; D.1.3 [*Programming Techniques*]: Concurrent Programming; F.3.1 [*Logics and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs—Assertions, Mechanical Verification

General Terms Algorithms, Design, Languages, Verification

Keywords Memory Models, Axiomatic Specifications, Bounded Model Checking, SAT

1. Introduction

In a multi-threaded shared-memory system, a *memory (consistency) model* specifies how concurrent accesses to shared memory are permitted to behave. In particular, a memory model determines which writes to a given location any read of that location may observe. The most intuitive memory model is *sequential consistency* [16], which requires simply that all accesses appear to execute one at a time, respecting the program order of each thread. But this simplicity comes at a price: the strict ordering requirements imposed by sequential consistency disallow many compiler and hardware optimizations that reorder instructions. As a result, most systems exhibit *relaxed memory models* (e.g., [1, 19]), which enable common instruction optimizations by stipulating weaker, and more complex, ordering rules. To make such rules accessible to programmers, formal specifications of memory models are usually supplemented with small programs, called *litmus tests*, that illustrate how the rules work.

Litmus tests elucidate a memory model insofar as they are consistent with its formal specification. Many automated techniques have therefore been developed for checking litmus tests and other micro-benchmarks against relaxed memory models. Tools based on model checking [7, 31], constraint solving [4, 11, 33], and custom search [24] have been successfully applied to axiomatic specifications of hardware memory models, such as the Intel Itanium and x86-CC Model, and to operational approximations of the Java Memory Model (JMM) [19]. But the JMM itself, which is given axiomatically, has so far eluded automatic checking despite a recognized need for it [3]. The large state space induced by the model's committing semantics has often been cited as prohibitive for model checking [2].

This paper presents MEMSAT, the first automated tool for debugging and reasoning about memory models that can handle the full axiomatic specification of the JMM. MEMSAT takes as input a memory model described by a set of constraints and a litmus test containing assertions. It then searches for a trace of the program in which both the assertions and the memory model constraints are satisfied. Assertions provide a convenient means of encoding the expected outcome of a litmus test, in that they express whether the computation of a given value at a given program point is permitted by the memory model. Constraints provide a natural means of expressing memory models. As argued in prior work [34], axiomatic specifications are easier to write and to understand than operational specifications, since they naturally decompose into the constituent rules of a memory model. Operational specifications, on the other hand, tend to be monolithic and coupled to specific data structures.

MEMSAT is fully automatic, requiring no user guidance, and is based on the Kodkod constraint solver [28], which works by reduction to SAT. The test program is translated into constraints that are assembled with the constraints describing the memory model. We then employ an optimization based on Kodkod's ability to complete a partial model to reduce the search space. If the combined result is satisfiable, our tool outputs a concrete execution of the program, called a *witness*, that is permitted by the memory model. Otherwise, it outputs a minimal subset of the constraints that are collectively unsatisfiable, called a *minimal unsatisfiable core*. Either a witness or a minimal core can help in debugging the test program or the memory model: a witness provides a concrete trace of memory operations that the user can examine as to whether it is intended, and a core shows which constraints—i.e., which rules in the memory model—prevent the assertions in the program from being satisfied.

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.

We used MEMSAT to check several existing memory models against their published test cases. We considered the Java Memory Model by Manson et al. [19] and a revised version of it by Sevcik and Aspinall [25]. We also studied five well-known memory models with existing axiomatic descriptions [34], including processor and causal consistency. For the JMM, our experiments confirmed that two causality test cases [6] did not behave as expected for the original model and that the revised version fixes these. We also discovered that the original JMM [19] correctly allows two other causality test cases that Aspinall and Sevcik report as forbidden [3]. For the other models, we found several discrepancies between the expected and actual results of tests.

Our case studies indicate that MEMSAT can be used to quickly and easily run litmus tests against different memory models. For litmus tests that contain no loops, the result of MEMSAT is sound and complete, meaning that there are no spurious witnesses, and if a witness exists, it is found. Otherwise, MEMSAT uses an under-approximation, and may miss witnesses, so it is sound but not complete. In practice, however, we have found that most litmus tests for memory models at the programming language level do not contain loops.

2. Overview

MEMSAT is designed as an extensible framework for specifying, testing and debugging memory models. It takes as inputs a multi-threaded *test program* with one or more assertions; a specification of a *memory model* in relational logic [14]; and a set of code *finitization parameters*, such as the number of times to unwind loops and the length of bitvectors used to represent integers. The test program is then finitized (by unwinding loops and bounding recursive method calls) and translated to relational logic. The resulting constraints are combined with the memory model constraints and checked with a SAT-based constraint solver. If the combined constraints are satisfiable, the program is said to be *legal* with respect to the memory model, and the output of the tool is a concrete *witness* of legality, expressed in terms of the relations constrained by the memory model. Otherwise, the program is said to be illegal, and the output is a proof of illegality, expressed as a *minimal unsatisfiable core* of the combined constraints.

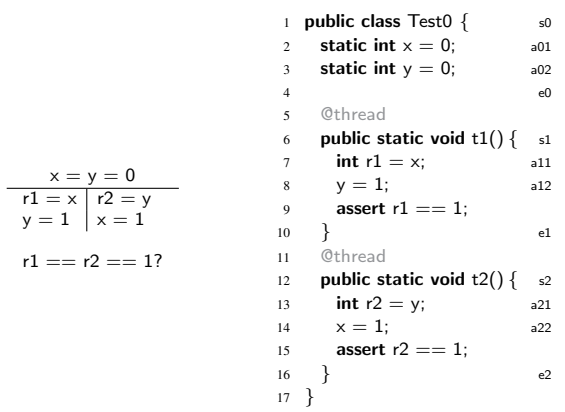
2.1 Test program

A test program consists of an (implicit) initialization thread and two or more user threads. The initialization thread executes first, writing initial values to all shared memory locations referenced in the program. The user threads execute after initialization has completed, running either in parallel or in a partial order specified by the user. Programs are encoded in a subset of Java that includes control flow constructs, the synchronized construct (which generates lock and unlock instructions on a given monitor), method calls, field and array accesses, integer and boolean operations, and assertions.

Figure 1 shows a test program from Manson et al. [19] both in a standard schematic notation [19, 25] and as an annotated Java program accepted by MEMSAT. The program consists of an initialization thread and two user threads that execute in parallel. In the schematic notation (Fig. 1(a)), threads that run in parallel are separated by a vertical bar; threads whose execution is partially ordered are separated by horizontal bars. In the Java encoding (Fig. 1(b)), the implicit static initialization method holds the code for the initialization thread, while the methods annotated with "@thread" indicate user threads. The (static) variables x and y refer to shared memory locations, and r1 and r2 refer to thread-local registers.

2.2 Memory model specification

A memory model is specified as a set of constraints in relational logic (§3)—that is, first order logic with quantifiers, set-theoretic



(a) Schematic notation (b) Annotated Java encoding

Figure 1. A sample litmus test for memory consistency models.

operators, transitive closure, and bitvector arithmetic.[1] These constraints are given over *relations* that describe properties of test programs (such as control flow) and their executions (such as the value written by a given write instruction). Each MEMSAT relation is defined in terms of memory-related operations, or *actions*, that are performed when a test program is executed. Actions are partitioned into thread start, thread end, (volatile) read, (volatile) write, lock, unlock, and special operations. Read, write, lock and unlock actions are generated by executing read, write and synchronize instructions. Special actions are generated by calls to methods that are designated as "special" in the definition of a given memory model (e.g., I/O methods in the Java Memory Model). Thread start and end actions mark the beginning and termination of a thread and do not correspond to any instructions.

Our framework provides two kinds of relations for specifying memory models: *relational constants* and *relational variables*. Relational constants capture static program properties, and MEMSAT infers their values directly from the program text. For example, the relative ordering of actions within a program's control flow graph is represented by the constant binary relation co. The value of co for the program in Fig. 1 is the set $\{\langle \texttt{s0}, \texttt{a01}\rangle, \langle \texttt{a01}, \texttt{a02}\rangle, \langle \texttt{a02}, \texttt{e0}\rangle, \langle \texttt{s1}, \texttt{a11}\rangle, \langle \texttt{a11}, \texttt{a12}\rangle, \langle \texttt{a12}, \texttt{e1}\rangle, \langle \texttt{s2}, \texttt{a21}\rangle, \langle \texttt{a21}, \texttt{a22}\rangle, \langle \texttt{a22}, \texttt{e2}\rangle\}$, where $\texttt{a}ij$ represents the action performed by the j^{th} memory-related instruction of the i^{th} thread, and $\texttt{s}i$ and $\texttt{e}i$ represent the start and end actions of the i^{th} thread. In this example, it suffices to think of an action as the memory operation performed by a given statement; however, the relationship can be more complex under the JMM, due to its speculative executions.

Relational variables capture execution properties. In our framework, an execution E is a structure $(\!|A, W, V, l, m, \mathcal{O}_1, \ldots, \mathcal{O}_n|\!)$ in which the relation A denotes the subset of the program's actions that are executed; the write-seen relation W maps each executed read to the write of which the value is seen by that read; the value-written relation V maps each write action to the value that is written; the location-accessed relation l maps each read and write to the memory location that it accesses; and the monitor-used relation m maps each lock and unlock to its associated monitor. The definition of an execution may also include any number of ordering relations $\mathcal{O}_i$ over A (e.g., happens-before) that are specific to a given memory model.

Example: Sequential Consistency Sequential consistency (SC) is perhaps the best known and most easily understood mem-

[1] Relational constraints are provided to MEMSAT using a Java API, rather than the illustrative syntax used in this paper.

ory model. It requires simply that all executed actions appear in a (weak) total order that is consistent with the program order. Figure 2 shows a relational specification of sequential consistency without synchronization, transcribed from Yang et al. [34]. Constants are displayed in the sans-serif font, logic keywords in the roman font, and variables in italics. The expression $r[x]$, where r is a binary relation and x is a scalar (or, in relational logic, a singleton unary relation), denotes the relational image of x under r; $r[x, y]$ denotes a formula that evaluates to true only if the relation r maps x to y; and r^+ denotes the transitive closure of r. The operator "one" constrains its argument relation to contain exactly one tuple.

1 $\forall i, j : A \mid i \neq j \implies (ord[i, j] \vee ord[j, i])$
2 $\forall i, j : A \mid ord[i, j] \implies \neg\, ord[j, i]$
3 $\forall i, j, k : A \mid (ord[i, j] \wedge ord[j, k]) \implies ord[i, k]$
4 $\forall i, j : A \mid (\mathsf{t}[i] = \mathsf{t}[j] \wedge \mathsf{co}^+[i, j]) \implies ord[i, j]$
5 $\forall i, j : A \mid (\mathsf{t}[i] \neq \mathsf{t}[j] \wedge \mathsf{to}^+[\mathsf{t}[i], \mathsf{t}[j]]) \implies ord[i, j]$
6 $\forall k : A \cap \mathsf{Read} \mid \mathrm{one}\ W[k] \wedge W[k] \subseteq (A \cap \mathsf{Write})$
7 $\forall k : A \cap \mathsf{Read} \mid l[k] = l[W[k]]$
8 $\forall k : A \cap \mathsf{Read} \mid \neg\, ord[k, W[k]]$
9 $\forall k : A \cap \mathsf{Read}, j : A \cap \mathsf{Write} \mid \neg(l[j] = l[k] \wedge ord[W[k], j] \wedge ord[j, k])$

Figure 2. Sequential consistency in relational logic.

We define sequential consistency in terms of the execution structure $E = (\!|A, W, V, l, m, ord|\!)$ and program constants co, to, t, Read and Write. The variable ord models the ordering of the executed actions A; the constant t maps each action in a program to the thread that executes it; the constant to denotes the partial execution order among threads; and Read and Write model all actions in a program whose action kind is a read or a write, respectively. The first three formulas in Fig. 2 constrain ord to be weakly total, asymmetric and transitive. The fourth and fifth formulas specify that it is consistent with the program order and the thread execution order. The sixth and seventh formulas constrain W to be a function from executed reads to executed writes and to be consistent with the location-accessed relation. The last two formulas require that W be consistent with ord: a read k cannot see a write that follows it in the ord relation, and no write to $l[k]$ is ordered (by ord) between $W[k]$ and k.

Example: Java Memory Model. The Java Memory Model (JMM) specifies what behavior is legal for a given program using a "committing semantics" [2, 3, 19, 25]. An execution is legal if it can be derived from a sequence of *speculative executions* of the program, constructed according to the following rules. The first execution in the sequence is "well-behaved" [3]: its reads can only see writes that *happen-before* them. The happens-before ordering (hb) transitively relates reads and writes in an execution according to the program order (po) and the synchronizes-with (sw) order implied by synchronization constructs. The remaining executions in the sequence are derived from the initial well-behaved execution by "committing" and executing data races. After each execution, one or more data races from that execution are chosen, and the reads and writes involved in those data races are remembered, or "committed." The committed data races are then executed in the next execution: each read in the execution must either be committed and see a committed write through a race, or it must see a write through the happens-before relation. The committed writes must also be executed, and they must write the committed values. Any execution reachable through this process is legal under the JMM.

Figure 3 presents a relational formalization of the revised JMM, transcribed from Sevcik and Aspinall [25]. A JMM execution $E = (\!|A, W, V, l, m, po, so, hb, sw|\!)$ includes four ordering relations: po, so, hb, and sw. The relation po models the program order, which is total over the actions of a single thread and which does not relate actions from different threads; so is a total order over all synchronization actions in A (i.e., the lock, unlock, thread start and thread end actions); sw consists of the tuples $\langle a, b\rangle$ in so such that a is an unlock and b is a lock on a given monitor, or a is a write and b is a read of a given volatile location in shared memory; and hb is the transitive closure of $po \cup sw$. An execution is *well-formed*, denoted by WELL-FORMED(E), if its constituent relations satisfy Definition 7 of the revised JMM [25], which we omit here for brevity. A well-formed execution E is *legal* if there is a finite sequence of sets C_i, where $0 \leq i \leq \mathsf{k}$, and a finite sequence of well-formed executions $E_i = (\!|A_i, W_i, V_i, l_i, m_i, po_i, so_i, hb_i, sw_i|\!)$, where $1 \leq i \leq \mathsf{k}$, that satisfy the constraints in Fig. 3. The upper bound on the number of speculative executions, denoted by k, can either be provided as an input to the tool or MEMSAT will compute a sound k from the program text. The symbol $\lhd$ denotes domain restriction, and all other symbols have their previously defined or standard meaning.

a WELL-FORMED$(E) \wedge \forall i : [1..\mathsf{k}] \mid$ WELL-FORMED(E_i)
b $A = \bigcup_{i=0}^{i \leq \mathsf{k}} C_i \wedge C_0 = \emptyset \wedge \forall i : [1..\mathsf{k}] \mid C_{i-1} \subseteq C_i$
c $\forall i : [1..\mathsf{k}] \mid C_i \lhd l_i = C_i \lhd l \wedge C_i \lhd m_i = C_i \lhd m$

1 $\forall i : [1..\mathsf{k}] \mid C_i \subseteq A_i$
2 $\forall i : [1..\mathsf{k}], r : C_i \cap \mathsf{Read} \mid (hb[W[r], r] \iff hb_i[W[r], r]) \wedge \neg hb_i[r, W[r]]$
3 $\forall i : [1..\mathsf{k}] \mid C_i \lhd V_i = C_i \lhd V$
4 $\forall i : [1..\mathsf{k}] \mid C_{i-1} \lhd W_i = C_{i-1} \lhd W$
5 $\forall i : [1..\mathsf{k}], r : (A_i \setminus C_i) \cap \mathsf{Read} \mid\ hb_i[W_i(r), r]$
6 $\forall i : [1..\mathsf{k}], r : (C_i \setminus C_{i-1}) \cap \mathsf{Read} \mid W[r] \subseteq C_{i-1}$
7 $\forall i : [1..\mathsf{k}], y : C_i, x : A_i \mid (y \subseteq \mathsf{Special} \wedge hb[x, y]) \implies x \subseteq C_i$

Figure 3. Revised JMM in relational logic.

2.3 Proof of legality (witness)

Given a test program P and a memory model M, MEMSAT generates a legality formula $F(P, M)$ of the form

$$F(P, E) \wedge F_\alpha(P, E) \wedge \textstyle\bigwedge_{i=1}^{i \leq k} F(P, E_i) \wedge M_P(E, E_1, \ldots, E_k),$$

where $k \geq 0$; $F(P, E)$ is true only if the execution E respects the intra-thread semantics of P; $F_\alpha(P, E)$ is true only if E satisfies all of the assertions in P; and $M_P(E, E_1, \ldots, E_k)$ is true only if the constraints that constitute M are satisfied with respect to the constants and variables that describe P and $E, E_1, \ldots, E_k$.

A *model* of the formula $F(P, M)$ is an assignment of relational values (i.e., sets of tuples) to the variables in $E, E_1, \ldots, E_k$ which makes each constraint in the formula true. This assignment, if it exists, is a concrete *witness* that at least one execution of P is both legal with respect to M and satisfies all the assertions in P. The tuples that comprise a MEMSAT model are drawn from a finite set, or *universe*, of symbolic values computed by the tool based on the program text and the values of the finitization parameters.

Figure 4 shows a witness that demonstrates the legality of the program in Fig. 1 with respect to the revised JMM (Fig. 3). For readability, MEMSAT displays formatted snippets of the model produced by the constraint solver rather than the complete assignment from variables to values. Each action in the set A (or A_i) of executed actions is annotated with its action kind. Read and write actions are additionally annotated with the location they access and the value they read or write. For example, the annotation "::read(x,0)" on the action a11 in the set A_1 means that a11 was a read of the value 0 from the field x (i.e. $V_1[W_1[\mathtt{a11}]] = 0$ and $l_1[\mathtt{a11}] = \mathtt{x}$). The values assigned to orderings such as hb are

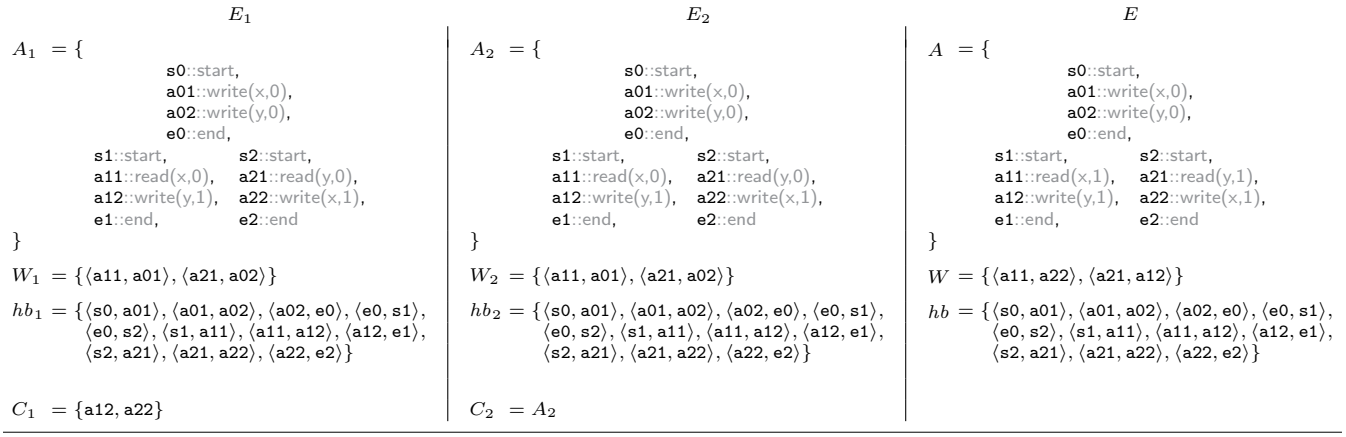


Figure 4. A witness for the program in Fig. 1 under the revised JMM (Fig. 3).

shown partially; we only display the tuples in their transitive reduction.

Operationally, the execution E in Fig. 4 is justified as follows. We start with the well-behaved execution E_1, in which each read sees the write that happens before it. Both the read of `x` by the thread `t1` and the read of `y` by `t2` see the initial writes of 0 to these locations. The execution E_1 has two data races that can be committed: `a11` and `a22` form a data race on `x`, and `a12` and `a21` form a data race on `y`. MEMSAT chooses to commit both, setting C_1 to $\{\texttt{a12}, \texttt{a22}\}$. The next execution, E_2, performs the committed writes: $\texttt{a12}, \texttt{a22} \in A_2$, and each writes 1 to its respective location. Note that the reads of E_2 still see the default writes since they have not been committed to C_1. We next commit the reads and all the remaining actions to C_2. The final execution, E, performs the actions from C_2, with each committed read seeing the write of 1 in the opposite thread.

2.4 Proof of illegality (minimal core)

A formula that has no models is said to be *unsatisfiable*. Unsatisfiability of a legality formula $F(P, M)$ means that the (finitized) program P has no executions that are legal with respect to M and that also satisfy all the assertions in P. To aid in the understanding of causes of illegality, MEMSAT produces a *minimal unsatisfiable core* of the formula $F(P, M)$. A minimal unsatisfiable core is an unsatisfiable subset of the formula's constraints that becomes satisfiable if any of its members are removed. In other words, a minimal core represents an irreducible cause of unsatisfiability.

constraint	source
$V[a_{01}] = 0$	Fig. 1(b), line 2
$V[a_{02}] = 0$	Fig. 1(b), line 3
$V[W[a_{11}]] = 1$	Fig. 1(b), line 9
$V[W[a_{21}]] = 1$	Fig. 1(b), line 15
$\forall i, j : A \mid i \neq j \implies (ord[i, j] \vee ord[j, i])$	Fig. 2, line 1
$\forall i, j, k : A \mid (ord[i, j] \wedge ord[j, k]) \implies ord[i, k]$	Fig. 2, line 3
$\forall i, j : A \mid (\mathsf{t}[i] = \mathsf{t}[j] \wedge \mathsf{co}^+[i, j]) \implies ord[i, j]$	Fig. 2, line 4
$\forall k : A \cap \mathsf{Read} \mid \neg\, ord[k, W[k]]$	Fig. 2, line 8

Figure 5. A minimal core for the program in Fig. 1 under sequential consistency (Fig. 2).

Figure 5 shows a minimal core that illustrates why the program in Fig. 1 has no sequentially consistent executions. The first two constraints[2] encode the meaning of the instructions on lines 2 and 3 of Fig. 1(b). The next two constraints encode the assertions on lines 9 and 15. The remaining constraints are drawn from the definition in Fig. 2. According to this core, the sample program is not sequentially consistent because all initial writes write 0; all assertions expect 1; all actions must be executed in a total order consistent with `co` (which, in this case, means that at least one of reads must occur before the non-initial write to the same location); and no read can observe an out-of-order write.

3. Approach

A MEMSAT analysis of a test program P and a memory model M involves five steps:

1. Converting P into an intermediate form $\mathcal{I}(P)$;
2. Translating $\mathcal{I}(P)$ into a relational representation $\mathcal{R}(P)$;
3. Combining $\mathcal{R}(P)$ and M into the legality formula $F(P, M)$;
4. Computing a set of bounds $B(P, M)$ on the space to be searched for a model or a core of $F(P, M)$; and
5. Finding a model or a core of the relational satisfiability problem defined by $F(P, M)$ and $B(P, M)$.

The first and last steps use off-the-shelf tools, the WALA [29] program analysis library and the Kodkod [26, 28] constraint solver. Translation, constraint assembly, and bounds computation comprise the core technical innovations of MEMSAT.

3.1 A brief introduction to relational logic

A key ingredient of our approach is the use of relational logic [14], which extends first-order logic with relational algebra and signed bitvector arithmetic. The basic concept in this logic is a *relation*: a set of tuples of equal length, drawn from a common universe of *atoms*. Atoms can denote integers or uninterpreted symbolic values. The *arity* of a relation, which can be any positive integer, determines the length of its tuples. We refer to unary relations (i.e., relations of arity 1) as "sets" and to singleton unary relations as "scalars."

The kernel of the logic, shown in Fig. 6, includes standard bitvector operators; connectives and quantifiers of first order logic; and the operators of relational algebra. The latter include relational join (.), product ($\rightarrow$), union ($\cup$), intersection ($\cap$), difference ($\setminus$), and transitive closure ($^+$). The join (.) of two relations is the pairwise join of their tuples, where $\langle a_0, \cdots, a_k \rangle.\langle a_k, \cdots, a_n \rangle$ yields

[2] As explained in the next section, MEMSAT encodes $F(P, E)$ and $F_\alpha(P, E)$ in terms of the variables a_{ij}, each of which is constrained to evaluate to the action (if any) generated by E while executing the j^{th} instruction of the i^{th} thread.

$$
\begin{array}{lll}
\mathit{Expr} & := & r \mid \emptyset \mid \mathit{Expr}^{+} \mid \mathit{Expr}\ \mathit{rop}\ \mathit{Expr} \mid \{\mathit{decl} \mid \mathit{Formula}\} \mid \\
& & \mathit{Formula}\ ?\ \mathit{Expr} : \mathit{Expr} \mid \mathrm{Bits}(\mathit{Bitvec}) \\
\mathit{Formula} & := & \mathit{Expr}\ \mathit{rcmp}\ \mathit{Expr} \mid \mathit{Bitvec}\ \mathit{bcmp}\ \mathit{Bitvec} \mid [\mathrm{l}]\mathrm{one}\ \mathit{Expr} \mid \\
& & \neg\, \mathit{Formula} \mid \mathit{Formula}\ \mathit{fop}\ \mathit{Formula} \mid \forall\, \mathit{decl} \mid \mathit{Formula} \mid \\
& & \exists\, \mathit{decl} \mid \mathit{Formula} \mid \mathit{true} \mid \mathit{false} \\
\mathit{Bitvec} & := & v \mid \neg\, \mathit{Bitvec} \mid \mathit{Bitvec}\ \mathit{bop}\ \mathit{Bitvec} \mid \mathrm{bits}(\mathit{Expr}) \mid |\mathit{Expr}|
\end{array}
$$

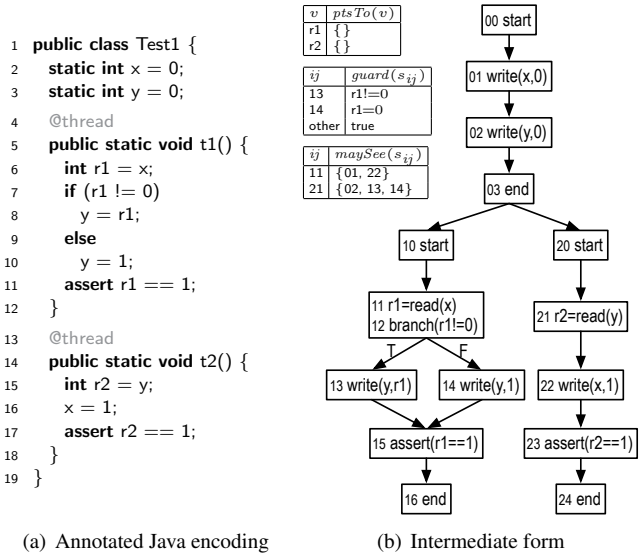
$$
\begin{array}{lll}
\mathsf{r} & := \text{relation} & \mathit{decl} := x\colon \mathit{Expr} \mid \mathit{decl}, \mathit{decl} \\
\emptyset & := \text{empty relation} & \mathit{rop} := . \mid \rightarrow \mid \cup \mid \cap \mid \setminus \\
\mathit{rcmp} & := \subseteq \mid \not\subseteq \mid = \mid \neq & \mathit{fop} := \wedge \mid \vee \mid \implies \mid \iff \\
\mathit{bcmp} & := < \mid \leq \mid = \mid > \mid \geq & \mathit{bop} := + \mid - \mid * \mid / \mid \% \mid \wedge \mid \vee \mid \oplus
\end{array}
$$

Figure 6. Relational logic.

$\langle a_0, \cdots, a_{k-1}, a_{k+1}, \cdots, a_n \rangle$. We use $e.r$ and $r[e]$ interchangeably to represent the join of e and r. The product ($\rightarrow$) of two relations is the pairwise product of their tuples, which is defined as $\langle a_0, \cdots, a_k \rangle \rightarrow \langle a_m, \cdots, a_n \rangle = \langle a_0, \cdots, a_k, a_m, \cdots, a_n \rangle$. The formulas $\mathrm{lone}\ \mathit{Expr}$ and $\mathrm{one}\ \mathit{Expr}$ are true for relations with at most one and exactly one tuple, respectively. The cardinality expression $|r|$ gives the number of tuples in r as a bitvector; $\mathrm{bits}(r)$ computes the sum of atoms representing integers in the set r as a bitvector; and $\mathrm{Bits}(v)$, where v is the bitvector $b_0 \ldots b_k$, evaluates to the set of integer atoms 2^i for which $b_i \neq 0$. All other expressions and formulas have their standard meaning.

3.2 Preprocessing

To translate a test program P to relational logic, MEMSAT first *finitizes* P's code by unwinding all loops and inlining all method calls. The finitized code is then transformed into an intermediate structure $\mathcal{I}(P) = (\!|efg, guard, ptsTo, maySee|\!)$, in which efg denotes the *extended control flow graph* of P; $guard$ maps each instruction in efg to the control conditions that guard its execution; $ptsTo$ maps each variable to the heap objects, if any, that it may point to at runtime; and $maySee$ maps each read in efg to the set of writes that it may observe. All four components of $\mathcal{I}(P)$ are computed using standard WALA analyses [29].



(a) Annotated Java encoding (b) Intermediate form

Figure 7. The intermediate form for a test program. The label ij for a statement s indicates that s is the j^{th} statement in the i^{th} thread.

An example of $\mathcal{I}(P)$ is shown in Fig. 7. The extended control flow graph of P is the union of the control flow graphs of P's threads, with additional edges between the exit and entry blocks of threads whose execution is partially ordered. The nodes of the graph are comprised of WALA statements[3] (Fig. 8) in the Static Single Assignment (SSA) form, which gives a new name to every new definition of a variable. Variable definitions are merged using ϕ statements, and heap accesses are expressed as explicit read and write statements. The synthetic start and end statements indicate the start and end of a thread. The value of $\mathit{guard}(s, v_i)$, where s is $v_j = \phi(\cdots, v_i, \cdots)$, is the condition under which s assigns v_i to v_j. For all other statements, $\mathit{guard}(s)$ evaluates to the conjunction of control conditions that must be true for s to execute [9].

$$
\begin{array}{l}
\mathit{JStmt} := \mathrm{start} \mid \mathrm{end} \mid \mathrm{branch}(\mathit{JExpr}) \mid \mathrm{assert}(\mathit{JExpr}) \mid v_i = \mathit{JExpr} \mid \\
\qquad v_j = \phi(\cdots, v_i, \cdots) \mid \mathrm{lock}(v_{ref}) \mid \mathrm{unlock}(v_{ref}) \mid \\
\qquad v_i = \mathrm{read}([v_{ref},]\ \mathit{Field}) \mid \mathrm{write}([v_{ref},]\ \mathit{Field}, \mathit{JLeaf}) \\
\mathit{JExpr} := \mathit{JLeaf} \mid \mathrm{new}(\mathit{Class}) \mid \mathit{JExpr}\ \mathit{op}\ \mathit{JExpr} \mid\, !\mathit{JExpr}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{JLeaf} & := v_i \mid \mathsf{null} \mid \mathsf{true} \mid \mathsf{false} \mid 0 \mid \text{-}1 \mid 1 \mid \text{-}2 \mid 2 \mid \ldots & \mathit{Field} := \text{identifier} \\
\mathit{op} & := + \mid - \mid * \mid / \mid \% \mid < \mid > \mid == \mid \&\& \mid \| & \mathit{Class} := \text{identifier}
\end{array}
$$

Figure 8. Syntax for statements in the intermediate representation.

3.3 Translation

The translation of a preprocessed program $\mathcal{I}(P)$ to its relational representation $\mathcal{R}(P)$ relies on a translation function $\mathcal{T} : \mathit{JExpr} \rightarrow \mathit{Expr}$ (Fig. 9) that takes a WALA expression and returns a relational expression. Unlike prior relational encodings (e.g., [9]) for sequential programs, the function $\mathcal{T}$ does not interpret heap accesses. That is, if a variable v_i is defined by a read statement s, $\mathcal{T}[\![v_i]\!]$ is an unconstrained unary relation ρ_{v_i}, which acts as a placeholder for the value read by s. In a sequential setting, a relational encoding for the value seen by a read can be computed directly from the program text. In a concurrent setting, however, these values are determined both by the program semantics and by the memory model. The placeholders are a key feature of our framework that allows us to separate the encoding of program semantics from the specification of the memory model: $\mathcal{T}$ encodes the program semantics in terms of the placeholders, which the constraint assembler (§3.4) then replaces with relational expressions dictated by the memory model.

$$
\mathcal{T}[\![v_i]\!] := \begin{cases} \rho_{v_i} & \text{if } \mathit{def}(v_i) \text{ is:} \\ \mathcal{T}[\![e]\!] & v_i = \mathrm{read}(*) \\ \bigcup_{j=1}^{n} \mathcal{F}[\![\mathcal{T}[\![\mathit{guard}(\mathit{def}(v_i), v_j)]\!]\ ?\ \mathcal{T}[\![v_j]\!] : \emptyset & v_i = e \\ & v_i = \phi(v_1, \ldots, v_n) \end{cases}
$$

$$
\begin{array}{lll}
\mathcal{T}[\![e_1 \,\&\&\, e_2]\!] := \mathcal{E}[\![\mathcal{F}[\![\mathcal{T}[\![e_1]\!]\!] \wedge \mathcal{F}[\![\mathcal{T}[\![e_2]\!]\!]\!] & & \mathcal{T}[\![\mathsf{true}]\!] := \mathsf{True} \\
\mathcal{T}[\![e_1 == e_2]\!] := \mathcal{E}[\![\mathcal{T}[\![e_1]\!] = \mathcal{T}[\![e_2]\!]\!] & & \mathcal{T}[\![\mathsf{false}]\!] := \mathsf{False} \\
\mathcal{T}[\![e_1 + e_2]\!] := \mathcal{E}[\![\mathcal{B}[\![\mathcal{T}[\![e_1]\!]\!] + \mathcal{B}[\![\mathcal{T}[\![e_2]\!]\!]\!] & &
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{E}[\![f]\!] := f\ ?\ \mathsf{True} : \mathsf{False} & \mathcal{B}[\![e]\!] := \mathrm{bits}(e) & \mathcal{F}[\![\mathcal{E}[\![f]\!]\!] := f \\
\mathcal{E}[\![b]\!] := \mathrm{Bits}(b) & \mathcal{E}[\![\mathcal{F}[\![e]\!]\!] := e & \mathcal{B}[\![\mathcal{E}[\![b]\!]\!] := b \\
\mathcal{F}[\![e]\!] := e \subseteq \mathsf{True} & \mathcal{E}[\![\mathcal{B}[\![e]\!]\!] := e &
\end{array}
$$

$$
\mathcal{L}[\![s]\!] := \begin{cases} \mathsf{f} & s \in \{\mathrm{read}(f), \mathrm{write}(f, e)\} \\ \mathsf{f} \cup \mathcal{T}[\![v_{ref}]\!] & s \in \{\mathrm{read}(v_{ref}, f), \mathrm{write}(v_{ref}, f, e)\} \\ \mathcal{T}[\![v_{ref}]\!] & s \in \{\mathrm{lock}(v_{ref}), \mathrm{unlock}(v_{ref})\} \end{cases}
$$

$$
\begin{array}{ll}
\mathcal{V}[\![s]\!] := & \mathcal{T}[\![e]\!] \qquad s \in \{\mathrm{write}(*, e), \mathrm{assert}(e)\} \\
\mathcal{G}[\![s]\!] := & \mathcal{F}[\![\mathcal{T}[\![\mathit{guard}(s)]\!]\!]
\end{array}
$$

Figure 9. Translation function $\mathcal{T}$ and representation functions $\mathcal{L}$, $\mathcal{V}$, and $\mathcal{G}$.

For expressions that are not defined by heap reads, the function $\mathcal{T}$ yields the same relational expressions as a prior encoding [9] for sequential programs. Figure 9 reproduces a representative sampling of those. The function def takes a variable in SSA form and returns the statement that defines its value. True and False are con-

[3] For the sake of brevity, we omit the formal description of arrays and method calls that correspond to special actions.

stant unary relations whose values are the atoms `true` and `false`, respectively. The function $\mathcal{E}$ converts formulas and bitvectors to expressions, and $\mathcal{F}$ and $\mathcal{B}$ do the reverse. All integer and boolean operations are translated using their corresponding operators in relational logic.

The relational representation $\mathcal{R}(P)$ is a structure $(\!|\mathcal{I}(P), \mathcal{L}, \mathcal{V}, \mathcal{G}|\!)$, which captures the semantics of the program $\mathcal{I}(P)$ with the partial functions $\mathcal{L}$, $\mathcal{V}$ and $\mathcal{G}$. The function $\mathcal{L}$ maps reads, writes, locks and unlocks to relational expressions that represent the heap locations or monitors accessed by these statements. If s is a read or a write of a static field f, $\mathcal{L}[\![s]\!]$ yields the constant relation f whose value, $\{\langle \mathtt{f} \rangle\}$, consists of the atom that represents the field f. If s reads or writes an instance field f, $\mathcal{L}[\![s]\!]$ yields $\mathsf{f} \cup \mathcal{T}[\![v_{ref}]\!]$, whose value is a set of two unary tuples, one of which represents the field f and the other the object referenced by v_{ref}. For monitor statements, $\mathcal{L}[\![s]\!]$ produces an expression that evaluates to the object that is locked or unlocked by s. The function $\mathcal{V}$ maps writes and assertions to relational encodings of the values that they write or assert. The function $\mathcal{G}$ takes each statement s in its domain to a relational formula that represents the guard of s. Figure 10 shows $\mathcal{L}$, $\mathcal{V}$, and $\mathcal{G}$ for the program in Fig. 7.

	s	$\mathcal{L}[\![s]\!]$	$\mathcal{V}[\![s]\!]$	$\mathcal{G}[\![s]\!]$	$acts(s)$
00	start			$true$	{a00}
01	write(x, 0)	x	Bits(0)	$true$	{a01}
02	write(y, 0)	y	Bits(0)	$true$	{a02}
03	end			$true$	{a03}
10	start			$true$	{a10}
11	r1 = read(x)	x		$true$	{a11}
13	write(y, r1)	y	ρ_{r1}	$\rho_{r1} \neq$ Bits(0)	{a13}
14	write(y, 1)	y	Bits(1)	$\rho_{r1} =$ Bits(0)	{a13}
15	assert(r1==1)		$\rho_{r1} =$ Bits(1)	$true$	
16	end			$true$	{a16}
20	start			$true$	{a20}
21	r2 = read(y)	y		$true$	{a21}
22	write(x, 1)	x	Bits(1)	$true$	{a22}
23	assert(r2==1)		$\rho_{r2} =$ Bits(1)	$true$	
24	end			$true$	{a24}

r	$B_u(r)$	$B_l(r)$
A_i	{⟨a00⟩, ⟨a01⟩, ⟨a02⟩, ⟨a03⟩, ⟨a10⟩, ⟨a11⟩, ⟨a13⟩, ⟨a16⟩, ⟨a20⟩, ⟨a21⟩, ⟨a22⟩, ⟨a24⟩}	$B_u(A_i) \setminus$ {⟨a13⟩}
W_i	{⟨a11, a01⟩, ⟨a11, a22⟩, ⟨a21, a02⟩, ⟨a21, a13⟩}	{ }
V_i	{a01, a02, a13, a22} × {-8, 1, 2, 4}	{ }
l_i	{a01, a11, a22} × {x} ∪ {a02, a13, a21} × {y}	$B_u(l_i)$
m_i	{ }	{ }

Figure 10. Relational representation functions, actions and sample bounds for the program in Fig. 7.

3.4 Constraint assembly

Our tool encodes the legality of a program $\mathcal{R}(P)$ with respect to a memory model $M_P(E, E_1, \ldots, E_k)$ using the recursive constraint assembly procedure defined in Fig. 11. The procedure F takes as input a relational representation $\mathcal{R}(P)$ and a memory model specification $M_P(E, E_1, \ldots, E_k)$ and produces the legality formula $F(P, M)$. The base step, $F(s, E_i)$, allocates a fresh unary relation a_s^i for each statement s and execution $E_i \in \{E, E_1, \ldots, E_k\}$ to represent the action that E_i performs if it executes s. The function $\sigma(fe, E_i)$ replaces all placeholder relations ρ_v in the formula or expression fe with $V_i[W_i[F(def(v), E_i)]]$, which is the value observed by the read that defines the variable v in the context of E_i. In other words, the application of σ replaces the placeholders generated in the translation stage with the values specified by the memory model.

The recursive step $F(\mathcal{R}(P), E_i)$ constrains the execution E_i to respect the semantics of $\mathcal{R}(P)$ by generating the following

$$F(\mathcal{R}(P), M_P(E, E_1, \ldots, E_k)) := F(\mathcal{R}(P), E) \wedge F_\alpha(\mathcal{R}(P), E) \wedge \left(\bigwedge_{j=1}^{k} F(\mathcal{R}(P), E_j)\right) \wedge M_P(E, E_1, \ldots, E_k)$$

$$\begin{aligned}
F(\mathcal{R}(P), E_i) \quad := & \left(\textstyle\bigwedge_{s \in ops(\mathcal{I}(P))} \text{lone } F(s, E_i)\right) \wedge && (1)\\
& \left(\textstyle\bigwedge_{s \in ops(\mathcal{I}(P))} \sigma(\mathcal{G}[\![s]\!], E_i) \iff F(s, E_i) \neq \emptyset\right) \wedge && (2)\\
& \left(\textstyle\bigwedge_{s, s' \in ops(\mathcal{I}(P)), s \neq s'} F(s, E_i) \cap F(s', E_i) = \emptyset\right) \wedge && (3)\\
& \left(\textstyle\bigwedge_{s \in ops(\mathcal{I}(P))} F(s, \mathcal{R}(P), E_i)\right) \wedge && (4)\\
& \left(A_i = \textstyle\bigcup_{s \in ops(\mathcal{I}(P))} F(s, E_i)\right) && (5)
\end{aligned}$$

$$F(s, \mathcal{R}(P), E_i) := \begin{cases} F(s, E_i).l_i = \sigma(\mathcal{L}[\![s]\!], E_i) & s \in \text{read}(*) \\ F(s, E_i).l_i = \sigma(\mathcal{L}[\![s]\!], E_i) \wedge \\ F(s, E_i).V_i = \sigma(\mathcal{V}[\![s]\!], E_i) & s \in \text{write}(*) \\ F(s, E_i).m_i = \sigma(\mathcal{L}[\![s]\!], E_i) & s \in \{\text{lock}(*), \text{unlock}(*)\} \end{cases}$$

$$\begin{aligned}
F(s, E_i) \quad & := a_s^i \\
F_\alpha(\mathcal{R}(P), E) \quad & := \textstyle\bigwedge_{s \in (\text{assert}(*) \cap \mathcal{I}(P))} \sigma(\mathcal{G}[\![s]\!] \wedge \mathcal{F}[\![\mathcal{V}[\![s]\!]]\!], E) \\
\sigma(fe, E_i) \quad & := fe \oplus \{\rho_v \mapsto V_i[W_i[F(def(v), E_i)]] \mid v \in vars(\mathcal{I}(P))\}
\end{aligned}$$

Figure 11. Constraint assembly function F. The auxiliary function ops yields all statements in $\mathcal{I}(P)$ that perform memory related operations, and $vars$ returns the defined variables. The operator $\oplus$ performs syntactic substitution: $fe \oplus \{x \mapsto y\}$ replaces all free occurrences of x in the formula or expression fe with y.

formuals: (1) a statement executed by E_i can perform at most one action; (2) a statement performs an action if and only if its guard is true in the context of E_i; (3) different statements, if executed, must perform different actions; (4) the value-written (V_i), location-accessed (l_i) and monitor-used (m_i) relations of E_i are consistent with the corresponding values given by $\mathcal{V}$ and $\mathcal{L}$; and (5) the set A_i of all actions executed by E_i is the union of the actions performed by the executed statements. The step F_α ensures that the main execution E satisfies all assertions in P.

3.5 Bounds assembly

The last phase of the analysis—finding a model or a core of the assembled legality formula—is delegated to the Kodkod constraint solver [26]. Kodkod takes as input a *relational satisfiability problem*, which is solved by reduction to boolean satisfiability and application of a SAT solver (e.g., [10]) to the resulting boolean constraints. A relational satisfiability problem consists of a formula in relational logic, a universe of atoms in which the formula is to be interpreted, and a *lower* and *upper bound* on the value of each relation in the formula. These bounds are given as sets of tuples drawn from the provided universe. The upper bound $B_u(r)$ specifies the tuples that the relation r *may* contain in a model of the formula. The lower bound $B_l(r) \subseteq B_u(r)$ designates the tuples that r *must* contain, if any. The total number of unknown tuples—i.e., $\sum_r |B_u(r) \setminus B_l(r)|$—determines the exponent in the size of the search space explored by Kodkod. We therefore use the algorithms in Figs. 12 and 13 to set the bounds judiciously, so that the resulting search space is both compact and includes all potential witnesses.

Figure 12 shows the MEMSAT functions for computing the universe and bounds for a legality formula $F(\mathcal{R}(P), M_P(E, E_1, \ldots, E_k))$. Both the universe and the bounds are defined in terms of the auxiliary function $acts$, discussed below, which maps each memory-related statement $s \in ops(\mathcal{I}(P))$ to a set of atoms representing the actions that the execution of s may generate. The universe consists of five kinds of symbolic values: 1) primitive values (bits, booleans, and null); 2) heap objects that may be allocated by P; 3) locations (fields) referenced within P; 4) threads that comprise P; and 5) memory actions that may be performed by P. The bounds are drawn from the universe based on the (sound) information computed by the preprocessor. For example, the upper bound on a_s^i is the set of all unary tuples drawn from $acts(s)$. Its lower bound is empty, unless the guard of s is the constant true and $acts(s)$ has exactly one action atom. In this case, the lower and upper bounds on a_s^i are the same; i.e., every E_i is guaranteed to ex-

$$\mathcal{U}(F(P,M)) := bits \cup bool \cup nil \cup objs \cup threads(\mathcal{I}(P)) \cup fields(\mathcal{I}(P)) \cup \bigcup_{s \in ops(\mathcal{I}(P))} acts(s)$$
$$B_u(F(P,M)) := \bigcup_{i \in \{\epsilon,1,\ldots,k\}} \{r \mapsto B_u(r, ops(\mathcal{I}(P))) \mid r \in E_i\} \cup \{a_s^i \mapsto B_u(s) \mid s \in ops(\mathcal{I}(P))\}$$
$$B_l(F(P,M)) := \bigcup_{i \in \{\epsilon,1,\ldots,k\}} \{r \mapsto B_l(r, ops(\mathcal{I}(P))) \mid r \in E_i\} \cup \{a_s^i \mapsto B_l(s) \mid s \in ops(\mathcal{I}(P))\}$$

...	$B_u(\ldots)$	precondition
r, S	$\bigcup_{s \in S} B_u(s)$ $\bigcup_{s \in S} acts(s) \times B_u(r,s)$	$arity(r) = 1$ $arity(r) = 2$
s	$\{\langle x \rangle \mid x \in acts(s)\}$	
W_i, s	$\bigcup_{x \in maySee(s)} acts(x)$	$s \in \text{read}(*)$
V_i, s	$bits$ $bool$ $ptsTo(v_{ref}) \cup nil$	$s \in \text{write}(*, e_{int})$ $s \in \text{write}(*, e_{bool})$ $s \in \text{write}(*, v_{ref})$
l_i, s	$\{\texttt{f}\}$ $\{\texttt{f}\} \cup ptsTo(v)$	$s \in rw(f)$ $s \in rw(f, v)$
m_i, s	$ptsTo(v)$	$s \in ul(v)$

...	$B_l(\ldots)$	precondition
r, S	$\bigcup_{s \in S} B_l(s)$ $\bigcup_{s \in S} acts(s) \times B_l(r,s)$	$arity(r) = 1$ $arity(r) = 2$
s	$B_u(s)$	$guard(s) = \textsf{true} \wedge$ $\lvert acts(s) \rvert = 1$
W_i, s	$\emptyset$	
V_i, s	$\emptyset$	
l_i, s	$\{\texttt{f}\}$ $\{\texttt{f}\}$ $\{\texttt{f}\} \cup ptsTo(v)$	$s \in rw(f)$ $s \in rw(f, v) \wedge \neg\, sole(s, v)$ $s \in rw(f, v) \wedge sole(s, v)$
m_i, s	$ptsTo(v)$	$ul(s, v) \wedge sole(s, v)$

$bits := \{-2^{b-1}, 1, \ldots, 2^{b-2}\}$ $\quad bool := \{\texttt{true}, \texttt{false}\}$ $\quad rw(f) := \{\text{read}(f), \text{write}(f, e)\}$ $\quad ul(v) := \{\text{lock}(v), \text{unlock}(v)\}$
$objs := \bigcup_{v \in vars(\mathcal{I}(P))} ptsTo(v)$ $\quad nil := \{\texttt{null}\}$ $\quad rw(f, v) := \{\text{read}(v, f), \text{write}(v, f, e)\}$ $\quad sole(s, v) := \lvert acts(s) \rvert = 1 \wedge \lvert ptsTo(v) \rvert = 1$

Figure 12. Computing the universe ($\mathcal{U}$) and bounds (B_l, B_u) for the legality formula $F(P, M) = F(\mathcal{R}(P), M_P(E, E_1, \ldots, E_k))$. All functions return $\emptyset$ if applied to arguments that violate their preconditions. The auxiliary function *threads* returns a set of objects that represent the threads in a given program, and *fields* yields the set of fields that are referenced in the program's read and write statements. Epsilon (ϵ) stands for the empty string, and b is the integer finitization parameter provided by the user.

```
COMPUTE-ACTS(efg, ptsTo)
1  acts ← map()
2  for 0 ≤ i < |threads(efg)| do
3    cfg_i ← restrict(efg, i)
4    R ← REPRESENTATIVE-ATOMS(cfg_i)
5    for s ∈ ops(cfg_i) do
6      S ← {}
7      for s' ∈ domain(R) do
8        if MAY-GEN-SAME-ACT(ptsTo, s, s') then
9          S ← S ∪ {R[s']}
10     acts[s] ← S
11 return acts

MAX-EXECUTABLE-SETS(cfg_i, s)
1  ms ← map()
2  for s' ∈ succs(cfg_i, s) do
3    ms' ← MAX-EXECUTABLE-SETS(cfg_i, s')
4    for k ∈ domain(ms') ∩ domain(ms) do
5      if |ms'[k]| > |ms[k]| then
6        ms[k] ← ms'[k]
7    for k ∈ domain(ms') \ domain(ms) do
8      ms[k] ← ms'[k]
9  k ← KEY(s)
10 ms[k] ← {s} ∪ (k ∈ domain(ms) ? ms[k] : {})
11 return ms
```

```
REPRESENTATIVE-ATOMS(cfg_i)
1  ms ← MAX-EXECUTABLE-SETS(cfg_i, s_i0)
2  ra ← map()
3  for k ∈ domain(ms) do
4    for s_ij ∈ ms[k] do
5      ra[s_ij] ← aij
6  return ra

MAY-GEN-SAME-ACT(ptsTo, s, s')
1  mayAlias ← λv.λv'. ptsTo(v) ∩ ptsTo(v') ≠ ∅
2  switch(s)
3    case start, end     : return s' = s
4    case read(f)        : return s' = read(f)
5    case write(f, e)    : return s' = write(f, e')
6    case read(v, f)     : return s' = read(v', f) ∧ mayAlias(v, v')
7    case write(v, f, e) : return s' = write(v', f, e') ∧ mayAlias(v, v')
8    case lock(v)        : return s' = lock(v') ∧ mayAlias(v, v')
9    case unlock(v)      : return s' = unlock(v') ∧ mayAlias(v, v')

KEY(ptsTo, s)
1  switch(s)
2    case start, end                : return list(kind(s))
3    case read(f), write(f, e)      : return list(kind(s), f)
4    case read(v, f), write(v, f, e) : return list(kind(s), f, ptsTo(v))
5    case lock(v), unlock(v)        : return list(kind(s), ptsTo(v))
```

Figure 13. COMPUTE-ACTS procedure for computing the assignment of actions to statements. The auxiliary function $restrict(efg, i)$ restricts a given *efg* to the control flow graph of the i^{th} thread; $domain(m)$ yields the set of all keys mapped by the map m; and $kind(s)$ returns the kind of the statement s as a string (e.g. "read", "write", etc.). Other auxiliary functions are self-explanatory.

ecute s and, therefore, to perform the action $acts(s)$. Other bounds are derived from $acts$ and $\mathcal{I}(P)$ in a similar fashion.

The procedure COMPUTE-ACTS for computing the $acts$ function is presented in Fig. 13. It works, thread by thread, as follows. Given a thread t_i, we use the function KEY to partition the statements of t_i into equivalence classes. For example, two reads of the same static field have equal keys and are in the same equivalence class. Then, for each class of statements C, MAX-EXECUTABLE-SETS finds the largest subset $C_{max} \subseteq C$ such that all elements of C_{max} appear on a single path through the CFG of t_i. We say that the statements in C_{max} are *representative* of C. Following the generation of C_{max} for each C, REPRESENTATIVE-ATOMS creates a unique atom $\texttt{a}ij$ for every representative statement s_{ij} and records the correspondence between the two in a map. The size of this map is an upper bound on the number of actions that any execution of t_i may generate, and it is bounded above by the total number of memory-related statements in t_i. The last few lines of COMPUTE-ACTS use the representatives map to compute $acts(s)$ for all s in t_i. In particular, $acts(s)$ contains the atom $\texttt{a}ij$ if s and s_{ij} may generate the same memory event (e.g., a read of the field f).

An example of the $acts$ mapping and of the resulting bounds is shown in Fig. 10. The sample mapping illustrates four key properties of $acts$, which ensure that our bounds are both compact and do not exclude any witnesses:

1. each statement s is mapped to at least one atom;
2. if s and s' may both be performed in some execution, the union of their $acts$ sets contains at least two atoms;
3. if s and s' may produce the same memory event, the intersection of their $acts$ contains at least one atom; and

	Sequential Consistency				Coherence				PRAM				Causal Consistency				Processor Consistency			
test	w.exp. /w.fnd.	sec	vars	clauses	w.exp. /w.fnd.	sec	vars	clauses	w.exp. /w.fnd.	sec	vars	clauses	w.exp. /w.fnd.	sec	vars	clauses	w.exp. /w.fnd.	sec	vars	clauses
1	y/y	2	363	584	y/y	2	87	62	y/y	2	835	1,298	y/y	2	1,030	1,649	y/y	2	996	1,528
2	n/n	2	800	1,335	y/y	2	249	128	y/y	2	1,628	2,584	n/n	2	2,262	3,756	y/y	2	1,902	2,966
3	y/y	3	36,848	69,896	y/y	2	4,380	1,786	y/y	10	143,328	256,041	y/y	12	326,930	616,481	y/y	14	149,986	267,130
4	n/n	1	1,305	2,226	y/y	1	429	242	y/y	1	2,803	4,613	y/y	1	3,727	6,337	y/y	1	3,203	5,181
5	n/n	2	1,028	1,735	y/y	1	332	185	n/n	2	2,138	3,449	n/n	2	2,922	4,906	n/n	2	2,469	3,914
6	n/n	1	716	1,186	n/n	1	238	251	y/y	1	1,404	2,193	y/y	1	1,990	3,265	n/n	2	1,639	2,519
7	n/n	2	2,164	3,777	y/y	1	758	274	y/y	2	4,932	8,404	y/y	2	6,274	10,940	n/n	2	5,542	9,281
8	y/y	1	2,298	4,026	y/y	1	589	336	y/y	2	5,679	9,368	y/y	1	8,817	15,329	y/y	2	6,330	10,331
9	n/n	2	7,592	13,890	y/y	1	2,061	370	y/y	2	23,472	41,122	y/y	2	37,171	67,671	y/y	3	25,302	43,988
10	n/n	1	800	1,334	y/y	1	249	127	n/n	1	1,644	2,610	n/n	1	2,265	3,757	n/n	1	1,918	2,992
11	n/n	1	800	1,333	y/y	2	249	126	y/y	1	1,628	2,582	n/y	1	2,262	3,754	y/y	1	1,902	2,964
12	n/n	2	1,849	3,210	y/y	1	432	218	y/y	1	4,486	7,277	n/n	2	7,095	12,215	y/y	2	5,038	8,088
13	n/n	2	3,547	6,328	y/y	1	651	308	y/y	2	9,791	16,216	n/y	2	17,184	30,385	y/y	2	10,753	17,684
14	n/n	1	1,317	2,244	y/y	1	449	268	y/y	1	2,109	3,244	n/y	1	3,732	6,275	y/y	1	2,445	3,682
15	n/n	1	2,203	3,842	y/y	1	788	379	y/y	1	3,887	6,334	n/y	1	6,316	10,921	n/n	2	4,417	7,049

Table 1. MEMSAT results for five classic memory models on Nemos test cases [20]. The column label "w.exp./w.fnd" stands for "witness expected / witness found."

4. if s and s' may never produce the same memory event, the intersection of their $acts$ sets is empty.

The first two properties ensure that witnesses are not missed because the search space excludes executions that perform particular statements or combinations of statements. For example, if $acts(s)$ is empty for some s, then both the lower and the upper bound on the relation a_s^i are also empty, which forces the solver to treat a_s^i as the constant relation $\emptyset$. As a result, the only way to satisfy the legality constraint $\sigma(\mathcal{G}[\![s]\!], E_i) \iff F(s, E_i) \neq \emptyset$ is to have the guard of s evaluate to false. An empty $acts$ set for s therefore rules out all witnesses that perform s. Similarly, if $acts(s) \cup acts(s')$ contains just one atom $\mathtt{a}ij$, then $B_u(a_s^i) = B_u(a_s^i) = \{\langle \mathtt{a}ij \rangle\}$. In this case, the only way to satisfy the legality constraint $F(s, E_i) \cap F(s', E_i) = \emptyset$ is to set either a_s^i or $a_{s'}^i$ (or both) to the empty set, thus ruling out all witnesses that execute both s and s'.

The third property ensures that witnesses are not missed because the memory model equates actions performed by different statements in the context of different executions. For example, the program in Fig. 7 is legal under the Java Memory Model. In its witness execution E, statement 11 reads the value 1 from x, which causes statement 13 to execute and write the value 1 to y; i.e. $a_{13} \subseteq A$. The execution E is justified by a speculative execution E_1, in which statement 11 reads the value 0 from x, causing statement 14 to execute and write the value 1 to y; i.e., $a_{14}^1 \subseteq A_1$. As a result, the only way to speculatively commit a write of 1 to y is to commit the result of executing a_{14}^1, but the only way to honor this commitment in E is by executing a_{13}. Hence, we must have $a_{13} = a_{14}^1$, which means that $B_u(a_{13}) \cap B_u(a_{14}^1)$ (and, by extension, $acts(s_{13}) \cap acts(s_{14})$) must be non-empty.

The fourth property ensures compactness of the search space. If s and s' may never perform the same event, then a commitment to perform an action generated by s in a speculative execution E_i can never be honored by executing s' in E_{i+1}. We can therefore leave $acts(s) \cap acts(s')$ empty to get a smaller search space without losing any witnesses.

4. Case Studies

We used MEMSAT to check seven existing memory models using their published test cases: the Java Memory Model (JMM) by Manson et al. [19], a revised version of it by Sevcik and Aspinall [25], and five classic memory models with existing axiomatic descriptions [34]. Our experiments revealed several discrepancies between what was expected and the actual results of the tests.

4.1 Classic memory models

Table 1 presents the results of applying MEMSAT to the classic memory models specified in the Nemos framework [34] and the test cases provided with the NemosFinder tool [20]. The test programs consist of 2-8 threads, each of which performs up to four memory accesses.[4] For every model and every test, the table shows whether a witness was expected and whether we found one; the total analysis time, rounded to the nearest second; and the size of the final SAT problem, as measured by the number of variables and clauses. All experiments were performed on a 2.4 GHz Intel Core 2 Duo machine with 4 GB of RAM.

We found that sequential consistency, coherence and PRAM behave as expected on all tests, while causal consistency allows witnesses for tests 10 and 12–15, against expectation. Our initial experiments also revealed that the Nemos definition of processor consistency (PC) behaves as sequential consistency on the given tests, due to an overconstraint in the `mapConstraints` predicate [34]. We isolated the faulty constraint from the minimal core generated for test 2. The PC results that are shown in Table 1 reflect our fix to `mapConstraints`, which involved replacing an equality with an implication. The analysis time for all tests was negligible.

4.2 Java Memory Model

The Java Memory Model (JMM) was first defined in the Java Language Specification [12]. A few years later, Pugh [21] showed that this initial definition was flawed, leading to a formal revision process [15]. Manson et al. [19] eventually re-defined the model in its current form, which we call the "original JMM." Sevcik and Aspinall [25] then formalized the original JMM in the Isabelle theorem prover [13] and discovered that it disallows causality tests 17–20 [6], contrary to expectation. They fixed this flaw in a revision of the model, which we call the "revised JMM," by weakening rules 2 and 7 of legality [19]. The revised model also omits rules 3 and 8; restricts executions to be finite; and modifies the handling of initial writes.

[4] Note that a program with t threads, which perform n memory operations each, has roughly $(n!)^t$ candidate executions under a relaxed memory model [24]. As a result, even tests as small as these are too difficult for manual analysis.

	Original JMM				Revised JMM			
test	w.exp. /w.fnd.	sec	vars	clauses	w.exp. /w.fnd.	sec	vars	clauses
1	y/y	2	3,044	5,725	y/y	2	1,764	2,767
2	y/y	2	7,311	18,872	y/y	2	4,250	6,352
3	y/y	2	9,579	23,179	y/y	2	5,776	8,993
4	n/n	2	2,924	5,552	n/n	2	1,709	2,807
5	n/n	3	15,029	28,869	n/n	2	9,263	15,795
6	y/y	2	6,506	15,457	y/y	2	3,822	3,591
7	y/y	2	5,459	10,410	y/y	2	2,949	4,859
8	y/y	2	3,715	7,112	y/y	2	2,192	3,662
9	y/y	2	7,505	14,384	y/y	2	4,581	7,716
10	n/n	2	15,486	29,850	n/n	2	9,480	16,000
11	y/y	2	9,106	17,484	y/y	2	4,645	7,763
12	n/n	4	22,038	65,378	n/n	3	10,757	18,213
13	n/n	2	3,066	5,826	n/n	2	1,781	2,851
14	n/n	35	78,481	256,684	n/n	13	34,393	71,345
15	n/n	96	332,130	1,246,708	n/n	56	112,230	249,352
16	y/y	1	1,930	3,539	y/y	1	1,152	1,809
17	y/y	2	15,348	42,259	y/y	2	8,873	15,784
18	y/y	2	15,357	42,297	y/y	1	8,882	15,822
19	y/n	2	5,693	11,351	y/y	1	3,182	5,380
20	y/n	2	8,335	16,727	y/y	2	4,552	7,702
T1	n/n	189	729,697	2,676,373	n/n	22	148,125	318,106
T1.t	y/y	23	492,152	1,773,208	y/y	4	112,569	238,649
T2	n/n	3	11,796	30,032	n/n	2	7,219	11,746
T2.t	y/y	1	5,564	11,066	y/y	2	3,650	6,665
T3	n/n	26	389,029	1,209,703	n/n	12	151,048	299,977
T3.t	y/y	34	459,861	1,483,288	y/y	8	152,963	307,337
T4	n/n	3	10,031	22,792	n/n	3	6,104	10,349
T4.t	y/y	2	12,162	28,091	y/y	2	7,415	11,801
T5	n/n	2	5,769	13,070	n/y	2	3,462	5,511
T5.t	y/y	1	5,393	11,581	y/y	1	3,326	5,522

Table 2. MEMSAT results for the original [19] and revised [25] definition of the JMM on standard causality tests [6] and program transformation tests [25].

Table 2 shows the results of applying MEMSAT to both the original and the revised JMM on two sets of test cases: the standard causality tests by Pugh et al. [6] and the program transformation tests by Sevcik and Aspinall [25]. Causality tests are labeled 1–20 and transformation tests are T1–T5.t. All tests consist of 2 to 4 threads, each with up to 10 lines of code. Two of the tests contain loops, which were unwound once. We initially found that none of the tests pass for the revised JMM, because its specification of the synchronizes-with relation fails to include edges from the end of the initialization thread to the start of the user threads. The omission was immediately apparent from the minimal cores, which showed that it was impossible for any reads to see the initial writes without these edges. We fixed the definition to include the missing edges and repeated the experiments to obtain the results in Table 2.

The highlighted entries show that the original JMM does not behave as expected on tests 19 and 20, confirming the findings by Aspinall and Sevcik [3]. The cores produced by MEMSAT for these tests consist of program constraints and rules 2 and 7 of the original JMM. The revised JMM, which weakens both rules, correctly validates tests 19 and 20.

Our findings disagree with those of Aspinall and Sevcik [3, 25] on the causality tests 17–18 and on the transform test T5. In particular, we found that the original JMM allows both tests 17 and 18. The witnesses for these tests would have been difficult to construct by hand, since each consists of an unusual justifying sequence that has two different statements, only one of which is guarded by a conditional, performing the same action. We also found that the revised JMM allows the transform test T5, against expectation. This test was designed to show that rule 9 of both the original and revised JMM forbids reordering of statements with special actions. The test indeed fails for the original model, and MEMSAT produces a minimal core that includes rule 9. Upon closer inspection of rule 9 in the revised model, we found that its specification is weaker than in the original model. The difference between the two rules is undocumented.

Most of our experiments on the JMM completed in a few seconds. The most notable exceptions—tests 15 and T1—had three or more user threads each and no witnesses. The resulting cores were hence large and relatively expensive to minimize. The core minimization feature [27] of Kodkod can be turned off for better performance. Without it, MEMSAT would simply output an unsatisfiable core that is not guaranteed to be minimal.

5. Related Work

Many memory models have been proposed both at the hardware interface (see [1] for a detailed survey) and at the programming language level [17, 19, 23]. The difficulty of reasoning about these proposals was recognized early, prompting the development of numerous techniques for formalizing and analyzing memory models.

Automated Analyses of the Java Memory Model. A number of automated analyses have been developed for various incarnations of the JMM. Yang et al. [31] present an analysis of the JMM CRF model [17]. In this work, the model is hard-coded in a tool based on the Murφ model checker [8], and test programs are given in the Murφ input language. Roychoudhury and Mitra [22] propose their own operational specification of the JMM that they devised from the Java Language Specification [12], based on guarded commands. They use XSB logic programming [30] for search-space exploration. For input test programs, the user needs to specify the paths taken in each thread. De et al. [7] developed OpMM, an operational under-approximation for the current version of the JMM [19]. They use a model checker similar to JavaPathFinder for state exploration, with the semantics of OpMM built into it. Aspinall and Sevcik [2] formalized the data race free guarantee of the current JMM and proved it using the Isabelle theorem prover [13]. Finally, Manson [18] developed a simulator for the current version of the JMM that enumerates all possible executions for a given program.

Since MEMSAT supports an axiomatic style for specifying memory models, the JMM specification can be provided directly as an input with no need for an operational approximation. Our tool is based on a SAT solver, so it suffers less from the state-explosion problem faced by explicit-state model checkers, and its test programs are given in Java. Unlike Manson's simulator [18], MEMSAT does not need to explicitly enumerate all executions, and it can be used to check memory models other than the JMM. Unlike techniques based on theorem proving, MEMSAT cannot be used to verify properties of memory models. Instead, it is intended as a debugging and rapid prototyping tool to help support reasoning about them.

Automated Analyses of Other Memory Models. Aside from the JMM, other memory models have been the focus of formalization and analysis in prior work. Sober [5] is a scalable model checking technique for detecting program executions that are not sequentially consistent due to store buffer relaxation. CheckFence [4] is a SAT-based tool for checking that a data structure, implemented in C, is sequentially consistent under a hard-coded relaxed memory model, which approximates several different hardware-level memory models. Unlike these tools, MEMSAT was developed for checking different memory models against small pro-

grams, rather than for checking larger programs against a specific low-level memory model. While both MEMSAT and CheckFence are based on SAT, their translation techniques are fundamentally different. MemSAT uses a template-based translation with placeholders, which enables it to handle different input memory models. MemSAT also incorporates optimizations that allow it to handle high-level memory models with speculative executions and arbitrary ordering relations.

Yang et al. [11, 33] devised a technique based on SAT for evaluating tests against the Itanium memory model. To encode transitivity constraints, the tool generates a functional program for each input test that, when executed, produces a set of propositional clauses to be analyzed by a SAT solver. The authors report debugging one of their test cases by examining the propositional clauses in a low-level core generated by the SAT solver. MEMSAT, by contrast, generates cores that are meaningful and minimal at the specification level. Moreover, our backend engine supports transitive closure naturally, simplifying the expression of transitivity constraints.

Sarkar et al. [24] used the HOL theorem prover to formalize a semantics for multiprocessor x86 programs with an integrated relaxed memory model. They also developed two automated tools written in OCaml, one to check litmus tests against the axiomatic memory model and one to run the tests on actual hardware. Unlike these tools, which are dedicated checkers for the x86-CC memory model, MEMSAT takes a memory model specification as input and targets Java rather than machine code.

Frameworks for Memory Model Analysis. Several frameworks for checking memory models have been proposed. The Unified Memory Model (UMM) [32] is a framework for describing operational memory models. Memory models in the UMM are given using guarded commands, with a transition table for specifying the behavior of instructions, and a bypass table for specifying allowed instruction reorderings. The framework is based on the Murφ model checker. Yang et al. also developed the Nemos framework [34], which is based on SAT and supports simple axiomatic specifications of low-level memory models. Litmus tests are provided as traces—i.e., sequences of reads and writes.

Our tool improves on the UMM and Nemos in several ways. First, unlike either of these tools, MEMSAT can provide proofs of illegality in the form of minimal cores. Second, Nemos encodes memory models with a limited set of predicates, which cannot express rules about locking or speculative executions. MemSAT handles memory models with both of these features. Finally, Nemos requires users to manually enumerate complete traces, which our tool does automatically based on the program and the assertions being checked.

6. Conclusions

We presented MEMSAT, a fully automated tool for debugging and reasoning about axiomatic specifications of memory models. We used MEMSAT to check the JMM [19], a revised version of it [2, 25], and several well-known memory models. Our experiments confirmed previously reported discrepancies in the expected behavior of test programs, and uncovered new ones. To the best of our knowledge, our tool is the first fully automated technique that can handle the current axiomatic specification of the JMM. In the future, we plan to investigate applying MEMSAT to hardware memory models.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[2] D. Aspinall and J. Sevcik. Formalising Java's data race free guarantee. In *TPHOLs '07*, pages 22–37, 2007.

[3] D. Aspinall and J. Sevcík. Java memory model examples: good, bad and ugly. In *VAMP '07*, Lisbon, Portugal, September 2007.

[4] S. Burckhardt, R. Alur, and M. M. K. Martin. CheckFence: checking consistency of concurrent data types on relaxed memory models. In *PLDI '07*, 2007.

[5] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *CAV '08*, 2008.

[6] Causality test cases for the Java Memory Model. http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html.

[7] A. De, A. Roychoudhury, and D. D'Souza. Java memory model aware software validation. In *PASTE '08*, 2008.

[8] D. Dill. The Murφ verification system. In *CAV '96*, 1996.

[9] J. Dolby, M. Vaziri, and F. Tip. Finding bugs efficiently with a SAT solver. In *FSE '07*, pages 195–204, 2007.

[10] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT'03*, 2003.

[11] G. Gopalakrishnan, Y. Yang, and H. Sivaraj. QB or Not QB: An efficient execution verification tool for memory orderings. In *CAV '04*, 2004.

[12] J. Gosling, B. Joy, and G. Steele. *The Java Specification Language*. Addison-Wesley, 1996.

[13] Isabelle Theorem Prover. http://isabelle.in.tum.de/.

[14] D. Jackson. *Software Abstractions: logic, language and analysis*. MIT Press, 2006.

[15] JSR-133: Java memory model and thread specification revision. http://www.cs.umd.edu/~pugh/java/memoryModel.

[16] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Trans. Comput.*, 28(9), 1979.

[17] J.-W. Maessen, Arvind, and X. Shen. Improving the Java Memory Model using CRF. In *OOPSLA '00*, 2000.

[18] J. Manson. *The Java memory model*. PhD thesis, University of Maryland, College Park, 2004.

[19] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL '05*, pages 378–391, New York, NY, USA, 2005. ACM.

[20] NemosFinder. http://www.cs.utah.edu/formal_verification/.

[21] W. Pugh. Fixing the Java memory model. In *Java Grande '99*, 1999.

[22] A. Roychoudhury and T. Mitra. Specifying multithreaded Java semantics for program verification. In *ICSE '02*, 2002.

[23] V. A. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A theory of memory models. In *PPoPP '07*, pages 161–172, New York, NY, USA, 2007. ACM.

[24] 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 '09*, 2009.

[25] J. Sevcík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP '08*, 2008.

[26] E. Torlak. *A constraint solver for software engineering: finding models and cores of large relational specifications*. PhD thesis, MIT, 2009.

[27] E. Torlak, F. S.-H. Chang, and D. Jackson. Finding minimal unsatisfiable cores of declarative specifications. In *FM '08*, 2008.

[28] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *TACAS '07*, 2007.

[29] Watson libraries for analysis (WALA). http://wala.sourceforge.net.

[30] The XSB logic programming system. http://xsb.sourceforge.net.

[31] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Analyzing the CRF Java memory model. In *APSEC '01*, 2001.

[32] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Specifying Java thread semantics using a uniform memory model. In *JGI '02*, 2002.

[33] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Analyzing the Intel Itanium memory ordering rules using logic programming and SAT. In *CHARME '03*, 2003.

[34] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: a framework for axiomatic and executable specifications of memory consistency models. In *IPDPS '04*, pages 26–30, 2004.