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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI’10, June 5–10, 2010, Toronto, Ontario, Canada.
Copyright © 2010 ACM 978-1-4503-0019/10/06...$10.00
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, we used the first version of the specification, which was clearly stated. For the other memory 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 operators, transitive closure, and bitwise 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).

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

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.

\[
\begin{align*}
1 & \forall i,j:A [i \neq j \Rightarrow (\text{ord}[i,j] \lor \text{ord}[j,i])] \\
2 & \forall i,j,k:A [\text{ord}[i,j] \lor \text{ord}[j,i]] \Rightarrow \neg \text{ord}[i,k] \\
3 & \forall i,j,k:A [\text{ord}[i,j] \land \text{ord}[j,k]] \Rightarrow \text{ord}[i,k] \\
4 & \forall i,j,k:A [\text{t}[i]=t[j] \land \text{ord}^+[i,j]] \Rightarrow \text{ord}[i,j] \\
5 & \forall i,j,k:A [\text{t}[i] \neq \text{t}[j] \land \text{ord}^+[i,j]] \Rightarrow \neg \text{ord}[i,j] \\
6 & \forall k:A \cap \text{Read} \lor \text{Write}[W[k] \land V[k]] \subseteq (A \cap \text{Write}) \\
7 & \forall k:A \cap \text{Read} \lor \text{Write}[I[k]=I[W[k]]] \\
8 & \forall k:A \cap \text{Read} \lor \text{Write}[\neg \text{ord}[k,W[k]]] \\
9 & \forall k:A \cap \text{Read},j:A \cap \text{Write}[\neg (I[j]=I[k] \land \text{ord}[W[k],j] \land \text{ord}[j,k])] \\
\end{align*}
\]

Figure 2. Sequential consistency in relational logic.

We define sequential consistency in terms of the execution structure \( E = \{A,W,V,I,m,\text{ord}\} \) and program constants co, to, t, Read and Write. The variable \( \text{ord} \) models the ordering of the executed actions. The constant \( \text{t} \) maps each action in a program to the thread that executes it; the constant \( \text{t} \) denotes the partial execution order among threads; and \( \text{Read} \) and \( \text{Write} \) model all actions in a program whose action kind is a read or a write, respectively. The first three formulas in Fig. 2 constrain \( \text{ord} \) 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 \( \text{ord} \); a read \( k \) cannot see a write that follows it in the \( \text{ord} \) relation, and no write to \( I[k] \) is ordered (by \( \text{ord} \)) 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 “computing 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) transitiveely 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 Sevick and Aspinal [25]. A JMM execution \( E = \{A,W,V,I,m,\text{po,so,hb,sw}\} \) includes four ordering relations:

\[
\begin{align*}
\text{po} \ & : \ a_1 \rightarrow a_2 \\
\text{so} \ & : \ a_1 \rightarrow a_2 \\
\text{hb} \ & : \ a_1 \rightarrow a_2 \\
\text{sw} \ & : \ a_1 \rightarrow a_2
\end{align*}
\]

The relation \( \text{po} \) 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); \( \text{sw} \) consists of the tuples \((a,b)\) in \( s \) 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 \( \text{hb} \) is the transitive closure of \( \text{po} \cup \text{sw} \). An execution is well-formed, denoted by \( \text{well-formed}(E) \), 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 \( 0 \leq i \leq k \), and a finite sequence of well-formed executions \( E_i = \{A_i,W_i,V_i,I_i,m_i,\text{po},\text{so},\text{hb},\text{sw}\} \), where \( 1 \leq i \leq k \), 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 \( \triangleleft \) denotes domain restriction, and all other symbols have their previously defined or standard meaning.

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

\[
F(P,E) \land F_{\text{po}}(P,E) \land \bigvee_{i=1}^{k} F(E_i) \land M_P(E,E_1,\ldots,E_k),
\]

where \( k \geq 0 \); \( F(P,E) \) is true only if the execution \( E \) respects the intra-thread semantics of \( P \); \( F_{\text{po}}(P,E) \) 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 \( E_1,\ldots,E_k \), 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 \( A_1 \)) 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 \( \text{:: \text{read}(x,0)} \) on the action \( a_1 \) in the set \( A_1 \) means that \( a_1 \) was a read of the value 0 from the field \( x \) (i.e., \( V_1[W_1[a_1]] = 0 \) and \( l_1[a_1] = x \)). The values assigned to orderings such as \( hh \) are
shown partially; we only display the tuples in their transitive closure.

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 \( t_1 \) and the read of \( y \) by \( t_2 \) see the initial writes of 0 to these locations. The execution \( E_1 \) has two data races that can be committed: \( a_11 \) and \( a_22 \) form a data race on \( x \), and \( a_12 \) and \( a_21 \) form a data race on \( y \). MEMSAT chooses to commit both, setting \( C_1 \) to \( \{ a_12, a_22 \} \). The next execution, \( E_2 \), performs the committed writes: \( a_12, a_22 \in A_2 \), 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 (finite) 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[x_{01}] = 0 Fig. 1(b), line 2
V[x_{02}] = 0 Fig. 1(b), line 3
V[W[x_{11}]] = 1 Fig. 1(b), line 9
V[W[x_{21}]] = 1 Fig. 1(b), line 15
∀i, j: \( A \upharpoonright i \neq j \Rightarrow (\text{ord}[i,j] \lor \text{ord}[j,i]) \) Fig. 2, line 1
∀i, j: \( A \upharpoonright i \neq j \Rightarrow (\text{ord}[i,j] \lor \text{ord}[j,i]) \) Fig. 2, line 3
∀i, j: \( A \upharpoonright i \neq j \Rightarrow (\text{ord}[i,j] \lor \text{ord}[j,i]) \) Fig. 2, line 4
∀k: \( A \cap \text{Read}[k] \Rightarrow \text{ord}[k, W[k]] \) 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 \( c_0 \) (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 \( I(P) \);
2. Translating \( I(P) \) into a relational representation \( R(P) \);
3. Combining \( R(P) \) 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 (\( \cdot \)), product (\( \cdot \cdot \)), union (\( \cup \)), intersection (\( \cap \)), difference (\( \setminus \)), and transitive closure (\( + \)). The join (\( \cdot \)) of two relations is the pairwise join of their tuples, where \( (a_0, \ldots, a_k), (a_{k+1}, \ldots, a_n) \) yields
Figure 6. Relational logic.

\[(a_0, \cdots, a_k, a_{k+1}, \cdots, a_n)\]. 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 \((a_0, \cdots, a_k) \rightarrow (a_0, \cdots, a_k) = (a_0, \cdots, a_0, a_{k+1}, \cdots, a_n)\). The formulas \(\text{one} \, \text{Expr}\) and one \(\text{Expr}\) 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; \(\text{bits}(r)\) computes the sum of atoms representing integers in the set \(r\) as a bitvector; and \(\text{Bits}(v)\), where \(v\) is the bitvector \(b_0 \cdots b_j\), evaluates to the set of integer atoms \(2^j\) 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 finitesizes \(P\)'s code by unwinding all loops and inlining all method calls. The finitized code is then transformed into an intermediate structure \(I(P) = \langle \text{egf}, \text{guard}, \text{ptsTo}, \text{maySec} \rangle\), in which \(\text{egf}\) denotes the extended control flow graph of \(P\); \(\text{guard}\) maps each instruction in \(\text{egf}\) to the control conditions that guard its execution; \(\text{ptsTo}\) maps each variable to the heap objects, if any, that it may point to at runtime; and \(\text{maySec}\) maps each read in \(\text{egf}\) to the set of writes that it may observe. All four components of \(I(P)\) are computed using standard WALA analyses [29].

An example of \(I(P)\) 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 \(\phi\) 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 \(\text{guard}(s, v_i)\), where \(s\) is \(v_j = (\cdots, v_i, \cdots)\), is the condition under which \(s\) assigns \(v_i\) to \(v_j\). For other statements, \(\text{guard}(s)\) evaluates to the conjunction of control conditions that must be true for \(s\) to execute [9].

\[J\text{Stmt} := \text{start} | \text{end} | \text{branch}(\text{JExpr}) | \text{assert}(\text{JExpr}) | v_i = JExpr | v_i = (\cdots, v_j, \cdots) | \text{lock}(v_i) | \text{unlock}(v_i) | v_i = \text{read}([v_{rref}], \text{Field}) | \text{write}(v_{wref}, \text{Field}, JLeaf)\]

\[J\text{Expr} := \text{JLeaf} | \text{new}(\text{Class}) | J\text{Expr} \oplus J\text{Expr} | J\text{Expr}\]

\[J\text{Leaf} := v_i | \text{null} | \text{true} | \text{false} | 0 | 1 | -1 | 2 | \cdots \]

\[\text{Field} := \text{identifier} | \text{expression}\]

\[\text{Class} := \text{identifier}\]

Figure 8. Syntax for statements in the intermediate representation.

3.3 Translation

The translation of a preprocessed program \(I(P)\) to its relational representation \(R(P)\) relies on a translation function \(T : I\text{Expr} \rightarrow J\text{Expr}\) (Fig. 9) that takes a WALA expression and returns a relational expression. Unlike prior relational encodings (e.g., [9]) for sequential programs, the function \(T\) does not interpret heap accesses. That is, if a variable \(v_i\) is defined by a read statement, \(T[v_i]\) is an unconstrained unary relation \(p_{v_i}\), 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; \(T\) 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.

\[T[v_i] := \begin{cases} \{ & \\
\bigcup_{j=1}^m T'[\text{guard}(\text{def}(v_i), v_j)] \wedge T[v_j] : v_j = \text{read}(s) \bigcup_{j=1}^m T'[\text{guard}(\text{def}(v_i), v_j)] \wedge T[v_j] : v_j = \phi(v_1, \cdots, v_n) \\
\{ & \\
T[e_1 \& e_2] := E[T[e_1] \wedge T[e_2]] & T[e_1] : \text{True} \\
T[e_1 \& e_2] := E[T[e_1] \wedge T[e_2]] & T[e_1] : \text{False} \\
E[f] := f & T[f] : \text{True} \\
E[f] := f & T[f] : \text{False} \\
E[f] := \text{bits}(e) & B[e] : e \subseteq \text{True} \\
E[f] := \text{bits}(e) & B[e] : e \subseteq \text{False} \\
E[f] := \text{write}(e, f) & \text{Field} : \text{Field} \wedge T[v_{rref}] \wedge T[v_{wref}] \\
E[f] := \text{write}(e, f) & \text{Field} : \text{Field} \wedge T[v_{rref}] \wedge T[v_{wref}] \\
E[f] := \text{write}(e, f) & \text{Field} : \text{Field} \wedge T[v_{rref}] \wedge T[v_{wref}] \\
E[f] := \text{write}(e, f) & \text{Field} : \text{Field} \wedge T[v_{rref}] \wedge T[v_{wref}] \\
E[f] := \text{write}(e, f) & \text{Field} : \text{Field} \wedge T[v_{rref}] \wedge T[v_{wref}] \\
E[f] := \text{write}(e, f) & \text{Field} : \text{Field} \wedge T[v_{rref}] \wedge T[v_{wref}] \\
\end{cases}\]

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

\(^3\)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 $E$ converts formulas and bitvectors to expressions, and $P$ and $B$ do the reverse. All integer and boolean operations are translated using their corresponding operators in relational logic.

The relational representation $R(P)$ is a structure $(L(P), E, V, G)$, which captures the semantics of the program $P$ with the partial functions $E$, $V$, and $G$. The function $L$ 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 $f$, $L[s]$ yields the constant relation $f$ whose value. (4), consists of the atom that represents the field $f$. If $s$ reads or writes an instance field $f$, $L[s]$ yields $f \cup T[v_{ref}]$, 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, $L[s]$ produces an expression that evaluates to the object that is locked or unlocked by $s$. The function $V$ maps variables and assertions to relational expressions of the values that they write or assert. The function $G$ takes each statement $s$ in its domain to a relational formula that represents the guard of $s$. Figure 10 shows $L$, $V$, and $G$ for the program in Fig. 7.

### 3.4 Constraint assembly

Our tool encodes the legality of a program $R(P)$ 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 $R(P)$ and a memory model specification $M_P(E, E_1, \ldots, E_k)$ and produces the legality formula $F(P, M)$. The base step, $F(s, E_1)$, allocates a fresh unary relation $a_i$ for each statement $s$ and execution $E_1 \in \{E_1, E_2, \ldots, E_k\}$ to represent the action that $E_1$ performs if it executes $s$. The function $\sigma(fe, E_1)$ replaces all placeholder relations $\rho_k$ in the formula or expression $fe$ with $V_i[W_i(F(def(v), E_1))]$, which is the value observed by the read that defines the variable $v$ in the context of $E_1$. In other words, the application of $\sigma$ replaces the placeholders generated in the translation stage with the values specified by the memory model.

The recursive step $F(R(P), E_1)$ constrains the execution $E_1$ to respect the semantics of $R(P)$ by generating the following formulas: (1) a statement executed by $E_1$ can perform at most one action; (2) a statement performs an action if and only if its guard is true in the context of $E_1$; (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 $E_1$ are consistent with the corresponding values given by $V_i$ and $L_i$; and (5) the set $A_i$ of all actions executed by $E_1$ is the union of the actions performed by the executed statements. The step $F$ 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 $B_u(I)$ specifies the tuples that the relation $I$ may contain in a model of the formula. The lower bound $B_l(I) \subseteq B_u(I)$ designates the tuples that $r$ must contain, if any. The total number of unknown tuples—i.e., $\sum_i |B_u(I_i) \setminus B_l(I_i)|$—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 $F(R(P), M_P(E, E_1, \ldots, E_k))$. Both the universe and the bounds are defined in terms of the auxiliary function $acts$, discussed below, which maps each memory-related statement $s \in ops(\mathcal{P}(P))$ 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 preorder. For example, the upper bound on $a_1$ 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 $a_1$ are the same; i.e., every $E_1$ is guaranteed to ex-
execute $s$ and, therefore, to perform the action $acts(s)$. Other bounds are derived from $acts$ and $T$ in a similar fashion.

The procedure \texttt{COMPUTE-ACTS} for computing the \texttt{acts} function is presented in Fig. 13. It works, thread by thread, as follows. Given a thread $t_i$, we use the function \texttt{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 $C$, \texttt{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 $C$, \texttt{REPRESENTATIVE-ATOMS} creates a unique atom $aij$ 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 \texttt{COMPUTE-ACTS} use the representatives map to compute $acts(s)$ for all $s$ in $t_i$. In particular, $acts(s)$ contains the atom $aij$ 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 $acts$ sets contains at least one atom; and
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. 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 memory operations each, has roughly $n!^3$ candidate executions under a relaxed memory model [24]. As a result, even tests as small as these are too difficult for manual analysis.

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.”

<table>
<thead>
<tr>
<th>Sequential Consistency</th>
<th>Coherence</th>
<th>PRAM</th>
<th>Causal Consistency</th>
<th>Processor Consistency</th>
</tr>
</thead>
<tbody>
<tr>
<td>test</td>
<td>w.exp.</td>
<td>w.fnd.</td>
<td>sec</td>
<td>vars</td>
</tr>
<tr>
<td>1</td>
<td>n/y</td>
<td>2</td>
<td>363</td>
<td>584</td>
</tr>
<tr>
<td>2</td>
<td>n/n</td>
<td>2</td>
<td>800</td>
<td>1,305</td>
</tr>
<tr>
<td>3</td>
<td>y/y</td>
<td>3</td>
<td>36,846</td>
<td>69,896</td>
</tr>
<tr>
<td>4</td>
<td>n/n</td>
<td>1</td>
<td>1,305</td>
<td>2,226</td>
</tr>
<tr>
<td>5</td>
<td>n/n</td>
<td>2</td>
<td>1,028</td>
<td>1,795</td>
</tr>
<tr>
<td>6</td>
<td>n/n</td>
<td>1</td>
<td>716</td>
<td>1,186</td>
</tr>
<tr>
<td>7</td>
<td>n/n</td>
<td>2</td>
<td>2,164</td>
<td>3,777</td>
</tr>
<tr>
<td>8</td>
<td>y/y</td>
<td>2</td>
<td>1,298</td>
<td>4,026</td>
</tr>
<tr>
<td>9</td>
<td>n/n</td>
<td>2</td>
<td>7,592</td>
<td>13,890</td>
</tr>
<tr>
<td>10</td>
<td>n/n</td>
<td>1</td>
<td>800</td>
<td>1,334</td>
</tr>
<tr>
<td>11</td>
<td>n/n</td>
<td>1</td>
<td>800</td>
<td>1,333</td>
</tr>
<tr>
<td>12</td>
<td>n/n</td>
<td>2</td>
<td>1,849</td>
<td>3,210</td>
</tr>
<tr>
<td>13</td>
<td>n/n</td>
<td>2</td>
<td>3,547</td>
<td>6,526</td>
</tr>
<tr>
<td>14</td>
<td>n/n</td>
<td>1</td>
<td>1,317</td>
<td>2,244</td>
</tr>
<tr>
<td>15</td>
<td>n/n</td>
<td>1</td>
<td>2,203</td>
<td>3,842</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Test</th>
<th>Original JMM</th>
<th>Revised JMM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>w.exp.</td>
<td>w.fnd. sec</td>
</tr>
<tr>
<td>1</td>
<td>y/y</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>y/y</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>y/y</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>n/n</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>n/n</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>y/y</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>y/y</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>y/y</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>y/y</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>n/n</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>y/y</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>n/n</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>n/n</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>n/n</td>
<td>35</td>
</tr>
<tr>
<td>15</td>
<td>n/n</td>
<td>96</td>
</tr>
<tr>
<td>16</td>
<td>y/y</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>y/y</td>
<td>2</td>
</tr>
<tr>
<td>18</td>
<td>y/y</td>
<td>2</td>
</tr>
<tr>
<td>19</td>
<td>y/y</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>y/n</td>
<td>2</td>
</tr>
<tr>
<td>T1</td>
<td>n/n</td>
<td>189</td>
</tr>
<tr>
<td>T1.t</td>
<td>y/y</td>
<td>23</td>
</tr>
<tr>
<td>T2</td>
<td>n/n</td>
<td>3</td>
</tr>
<tr>
<td>T2.t</td>
<td>y/y</td>
<td>1</td>
</tr>
<tr>
<td>T3</td>
<td>n/n</td>
<td>26</td>
</tr>
<tr>
<td>T3.t</td>
<td>y/y</td>
<td>34</td>
</tr>
<tr>
<td>T4</td>
<td>n/n</td>
<td>3</td>
</tr>
<tr>
<td>T4.t</td>
<td>y/y</td>
<td>2</td>
</tr>
<tr>
<td>T5</td>
<td>n/n</td>
<td>2</td>
</tr>
<tr>
<td>T5.t</td>
<td>y/y</td>
<td>1</td>
</tr>
</tbody>
</table>

5. Related Work

Many memory models have been proposed both at the hardware interface (see [1] for a 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 tool 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 as traces—i.e., sequences of reads and writes. 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**


