

PrideMM: Second Order Model Checking for Memory Consistency Models

Simon Cooksey¹, Sarah Harris¹, Mark Batty¹, Radu Grigore¹, and Mikoláš Janota²

¹ University of Kent, Canterbury
{sjc205, seh53, mjb211, rg399}@kent.ac.uk
² IST/INESC-ID, University of Lisbon

Abstract. We present PrideMM, an efficient model checker for second-order logic enabled by recent breakthroughs in quantified satisfiability solvers. We argue that second-order logic sits at a sweet spot: constrained enough to enable practical solving, yet expressive enough to cover an important class of problems not amenable to (non-quantified) satisfiability solvers. To the best of our knowledge PrideMM is the first automated model checker for second-order logic formulae.

We demonstrate the usefulness of PrideMM by applying it to problems drawn from recent work on memory specifications, which define the allowed executions of concurrent programs. For traditional memory specifications, program executions can be evaluated using a satisfiability solver or using equally powerful ad hoc techniques. However, such techniques are insufficient for handling some emerging memory specifications.

We evaluate PrideMM by implementing a variety of memory specifications, including one that cannot be handled by satisfiability solvers. In this problem domain, PrideMM provides usable automation, enabling a modify-execute-evaluate pattern of development where previously manual proof was required.

1 Introduction

This paper presents PrideMM, an efficient model checker for second-order (SO) logic. PrideMM is used to automatically evaluate tests under the intricate memory specifications³ of aggressively optimised *concurrent* languages, where no automated solution currently exists, and it is compared to existing tools over a simpler class of memory specifications.

We argue that SO logic is a sweet spot: restrictive enough to enable efficient solving, yet expressive enough to extend automation to a new class of memory specifications that seek to solve open problems in concurrent language design. PrideMM enables a modify-execute-evaluate pattern of memory-specification development, where changes are quickly implemented and automatically tested.

³ The paper uses the term ‘memory specification’ instead of ‘memory (consistency) model’, and reserves the word ‘model’ for its meaning from logic.

Memory specifications define what values may be read in a concurrent system. Current evaluators rely on ad hoc algorithms [3,6,14] or satisfiability (SAT) solvers [40]. However, flaws in existing language memory specifications [5] — where one must account for executions introduced through aggressive optimisation — have led to a new class of memory specifications [22,20] that cannot be practically solved using existing ad hoc or SAT techniques.

Many memory specifications are definable in $\exists\text{SO}$ in a natural way and one can simulate them using SAT solvers. We demonstrate this facility of PrideMM for a realistic C++ memory specification [24], reproducing previous results [40,39]. But, some memory specifications are naturally formulated in higher-order logic. For example, the Jeffrey-Riely specification (J+R) comes with a formalisation, in the proof assistant Agda [11], that clearly uses higher-order features [20]. We observed that the problem of checking whether a program execution is allowed by J+R can be reduced to the model checking problem for SO. From a program execution, one obtains an SO structure \mathfrak{A} on an universe of size n , and then one asks whether $\mathfrak{A} \models \text{JR}_n$, where

$$\begin{aligned} \text{JR}_n &:= \exists X (\text{TC}_n(\text{AeJ}_n)(\emptyset, X) \wedge F(X)) \\ \text{AeJ}_n(P, Q) &:= \begin{cases} \text{sub}^1(P, Q) \wedge V(P) \wedge V(Q) \wedge \\ \forall X (\text{TC}_n(\text{AJ})(P, X) \rightarrow \exists Y (\text{TC}_n(\text{AJ})(X, Y) \wedge \text{J}(Y, Q))) \end{cases} \end{aligned}$$

We will define precisely these formulae later (§5.4). For now, observe that the formula JR_n is in $\exists\forall\exists\text{SO}$. In practice, this means that it is not possible to use SAT solvers, as that would involve an exponential explosion. That motivates our development of an SO model checker. It is known that SO captures the polynomial hierarchy [27, Corollary 9.9], and the canonical problem for the polynomial hierarchy is quantified satisfiability. Hence, we built our SO model checker on top of a quantified satisfiability solver (QBF solver), QFUN [17].

The contributions of our work are as follows:

1. we present a model checker for SO, built on top of QBF solvers;
2. we reproduce known simulation results for traditional memory specifications;
3. we simulate a memory specification (J+R) that is a representative of a class of memory specifications that are out of the reach of traditional simulation techniques.

2 Overview

Figure 1 shows the architecture of our memory-specification simulator. The input is a litmus test written in the LISA language, and the output is a boolean result. LISA is a programming language that was designed for studying memory specifications [1]. We use LISA for its compatibility with the state-of-the-art memory-specification checker Herd7 [3]. We transform the input program into an event structure [41]. The memory-specification generator (MSG) produces an SO formula. We have a few interchangeable MSGs (§5). For some memory

specifications (§ 5.1, § 5.2, § 5.3), which Herd7 can handle as well, the formula is in fact fixed and does not depend at all on the event structure. For other memory specifications (such as § 5.4), the MSG might need to look at certain characteristics of the structure (such as its size). Finally, both the second-order structure and the second-order formula are fed into a solver, giving a verdict for the litmus test.

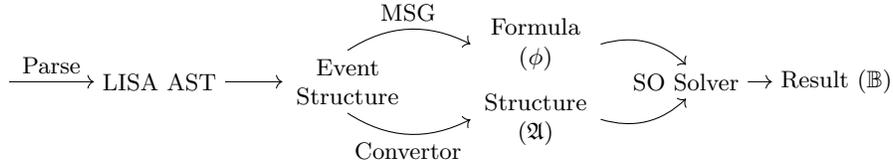


Fig. 1. From a LISA test case to a Y/N answer, given by the SO solver.

We are able to do so because of a key insight: relational second-order logic represents a sweet-spot in the design space. On the one hand, it is expressive enough such that encoding memory specifications is natural. On the other hand, it is simple enough such that it can be solved efficiently, using emerging QBF technology.

2.1 Memory Specifications

A *memory specification* describes the executions allowed by a shared-memory concurrent system; for example, under *sequential consistency* (SC) [25] memory accesses from all threads are interleaved and reads take their value from the most recent write of the same variable. Processor speculation, memory-subsystem reordering and compiler optimisations lead mainstream languages and processors to violate SC, and we say such systems exhibit *relaxed concurrency*. Relaxed concurrency is commonly described in an *axiomatic* specification (e.g. SC, ARM, Power, x86, C++ specifications [3]), where each program execution is represented as a graph with memory accesses as vertices, and edges representing program structure and dynamic memory behaviour. A set of axioms permit some execution graphs and forbid others.

Figure 2 presents a *litmus test* — a succinct pseudocode program designed to probe for a particular relaxed behaviour — together with an execution graph and an axiom. We shall discuss each in turn.

The test, called *LB+ctrl*, starts with x and y initialised to 0, then two threads concurrently read and conditionally write 1 back to their respective variables. The outcome $r_1 = 1 \wedge r_2 = 1$ (1/1) is unintuitive, and it cannot result from SC: there is no interleaving that agrees with the program order and places the writes of 1 before the reads for both x and y .

In an axiomatic specification, the outcome specified by the test corresponds to the execution graph shown in Figure 2. Initialisation is elided, but the read

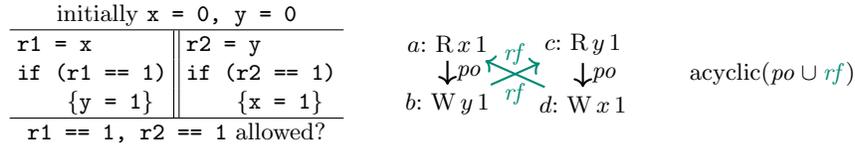


Fig. 2. LB+ctrl, an axiomatic execution of it, and an axiom that forbids it.

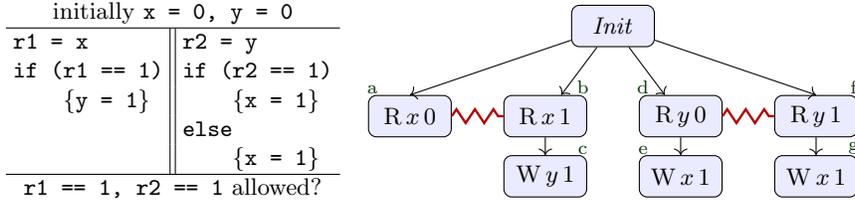


Fig. 3. LB+false-dep and the corresponding event structure.

and write of each thread is shown with po edges reflecting program order and rf edges linking writes to reads that *read from* them. The axiom of Figure 2 forbids the outcome 1/1 as the corresponding execution contains a cycle in $po \cup rf$. The SC, x86, Power and ARM specifications each include a variant of this axiom, all forbidding 1/1, whereas the C++ standard omits it [6] and allows 1/1.

MemSAT [39] and Herd7 [3] automatically solve litmus tests for axiomatic specifications using a SAT solver and ad hoc solving respectively, but not all memory specifications fit the axiomatic paradigm.

Axiomatic specifications do not fit optimised languages. Languages like C++ and Java perform dependency-removing optimisations that complicate their memory specifications. For example, the second thread of the LB+false-dep test in Figure 3 can be optimised using common subexpression elimination to $r2=y; x=1;$. On ARM and Power, this optimised code may be reordered, permitting the relaxed outcome 1/1, whereas the syntactic control dependency of the original would make 1/1 forbidden. It is common practice to use syntactic dependencies to enforce ordering on hardware, but at the language level the optimiser removes these *false* dependencies.

The memory specification of the C++ standard [15] is flawed because its axiomatic specification cannot draw a distinction between the executions leading to outcome 1/1 in LB+ctrl and LB+false-dep: to see that the dependency is false, one must consider more than one execution path, but axiomatic specifications judge single executions only [5].

Event structures capture the necessary information. A new class of specifications aims to fix this by ordering only real dependencies [22,20,31,12]. With a notable exception [22], these specifications are based on *event structures*, where all paths of control flow are represented in a single graph. Figure 3 presents the event structure for LB+false-dep. Program order is represented by arrows (\rightarrow). Conflict (\rightsquigarrow)

links events where only one can occur in an execution (the same holds for their program-order successors). For example, on the left-hand thread, the load of x can result in a read of value 0 (event a) or a read of value 1 (event b), but not both. Conversely, two subgraphs unrelated by program-order or conflict, e.g. $\{a, b, c\}$ and $\{d, e, f, g\}$, represent two threads in parallel execution.

It should be clear from the event structure in Figure 3 that regardless of the value read from y in the right-hand thread, there is a write to x of value 1; that is, the apparent dependency from the load of y is false and could be optimised away. Memory specifications built above event structures can recognise this pattern and permit relaxed execution.

The Jeffrey and Riely specification. J+R is built above event structures and correctly identifies false dependencies [20]. Conceptually, the specification is related to the Java memory specification [29]: in both, one constructs an execution stepwise, adding only memory events that can be *justified* from the previous steps. The sequence captures a causal order that prevents cycles with real dependencies. While Java is too strong, J+R allows writes that have false dependencies on a read to be justified before that read. To do this, the specification recognises confluence in the program structure: regardless of the execution path, the write will always be made. This search across execution paths involves an alternation of quantification that current ad hoc and SAT-based tools cannot efficiently simulate. However, the problem is amenable to QBF solvers.

2.2 Developing SC in SO Logic

The SC memory specification can be expressed as an axiomatic model [3] using *coherence order*, a per-variable total order of write events. An execution is allowed if there exists a reads-from relation rf and a coherence order co such that the transitive closure of $rf \cup co \cup (rf^{-1}; co) \cup po$ is acyclic. Here, po is the (fixed) program-order relation, and it is understood that co and rf satisfy certain further axioms. In our setting, we describe the sequentially consistent specification as follows. We represent rf and co by existentially-quantified SO arity-2 variables Y_{rf} and Y_{co} , respectively. For example, to say $(x, y) \in co$, we use the formula $Y_{co}(x, y)$. The program order po is represented by an interpreted arity-2 symbol $<$. Then, the SO formula that represents $rf \cup co \cup (rf^{-1}; co) \cup po$ is

$$R(y, z) := Y_{rf}(y, z) \vee Y_{co}(y, z) \vee \exists x (Y_{rf}(x, z) \wedge Y_{co}(x, y)) \vee (y < z)$$

The definition from above should be interpreted as a macro expansion rule: the left-hand side $R(y, z)$ is a combinator that expands to the formula on right-hand side. To require that the transitive closure of R is acyclic we require that there exists a relation that includes R , is transitive, and irreflexive:

$$\exists Z (\text{sub}^2(R, Z) \wedge \text{trans}(Z) \wedge \text{irrefl}(Z))$$

The combinators sub^2 , trans , irrefl are defined as one would expect. For example, $\text{sub}^2(P, Q)$, which says that the arity-2 relation P is included in the arity-2 relation Q , is $\forall xy (P(x, y) \rightarrow Q(x, y))$. In short, the translation from the usual formulation of memory specifications into the SO logic encoding that we propose is natural and almost automatic.

To represent programs and their behaviours uniformly for all memory specifications in §5, we use event structures. These have the ability to represent an overlay of potential executions. Some memory specifications require reasoning about several executions at the same time: this is a salient feature of the J+R memory specification.

Once we have the program and its behaviour represented as a logic structure \mathfrak{A} and the memory specification represented as a logic formula ϕ , we ask whether the structure satisfies the formula, written $\mathfrak{A} \models \phi$. In other words, we have to solve a model-checking problem for second-order logic, which reduces to QBF solving because the structure \mathfrak{A} is finite.

3 Preliminaries

To introduce the necessary notation, we recall some standard definitions [27]. A (finite, relational) *vocabulary* σ is a finite collection of *constant symbols* $(1, \dots, \mathfrak{n})$ together with a finite collection of *relation symbols* $(\mathfrak{q}, \mathfrak{r}, \dots)$. A (finite, relational) *structure* \mathfrak{A} over vocabulary σ is a tuple $\langle A, Q, R, \dots \rangle$ where $A = \{1, \dots, n\}$ is a finite set called *universe* with several distinguished relations Q, R, \dots . We assume a countable set of *first-order variables* (x, y, \dots) , and a countable set of *second-order variables* (X, Y, \dots) . A *variable* α is a first-order variable or a second-order variable; a *term* t is a first-order variable or a constant symbol; a *predicate* P is a second-order variable or a relation symbol. A (second-order) *formula* ϕ is defined inductively: (a) if P is a predicate and t_1, \dots, t_k are terms, then $P(t_1, \dots, t_k)$ is a formula⁴; (b) if ϕ_1 and ϕ_2 are formulae, then $\phi_1 \circ \phi_2$ is a formula, where \circ is a boolean connective; and (c) if α is a variable and ϕ is a formula, then $\exists \alpha \phi$ and $\forall \alpha \phi$ are formulae. We assume the standard satisfaction relation \models between structures and formulae.

The logic defined so far is known as relational SO. If we require that all quantifiers over second-order variables are existentials, we obtain a fragment known as \exists SO. For example, the SC specification of §2.2 is in \exists SO.

The Model Checking Problem. Given a structure \mathfrak{A} and a formula ϕ , determine if $\mathfrak{A} \models \phi$. We assume that the relations of \mathfrak{A} are given by explicitly listing their elements. The formula ϕ uses the syntax defined above.

Combinators. We will build formulae using the combinators defined below. This simplifies the presentation, and directly corresponds to an API for building

⁴ we make the usual assumptions about arity

formulae within PrideMM.

$$\begin{aligned}
 \text{sub}^k(P^k, Q^k) &:= \forall \mathbf{x} (P^k(\mathbf{x}) \rightarrow Q^k(\mathbf{x})) & \text{id}(x, y) &:= (x = y) \\
 \text{eq}^k(P^k, Q^k) &:= \forall \mathbf{x} (P^k(\mathbf{x}) \leftrightarrow Q^k(\mathbf{x})) & \text{inv}(P^2)(x, y) &:= P^2(y, x) \\
 \text{seq}(P^2, Q^2)(x, z) &:= \exists y (P^2(x, y) \wedge Q^2(y, z)) & \text{irrefl}(P^2) &:= \forall x \neg P^2(x, x) \\
 \text{inj}(P^2) &:= \text{sub}^2(\text{seq}(P^2, \text{inv}(P^2)), \text{id}) & \text{or}(\mathbf{R}, \mathbf{S})(x, y) &:= \mathbf{R}(x, y) \vee \mathbf{S}(x, y) \\
 \text{trans}(P^2) &:= \text{sub}^2(\text{seq}(P^2, P^2), P^2) & \text{maybe}(\mathbf{R})(x, y) &:= \text{or}(\text{id}, \mathbf{R})(x, y) \\
 \\
 \text{acyclic}(P^2) &:= \exists X^2 (\text{sub}^2(P^2, X^2) \wedge \text{trans}(X^2) \wedge \text{irrefl}(X^2)) \\
 \text{TC}_0(\mathbf{R}) &:= \text{eq}^1 \\
 \text{TC}_{n+1}(\mathbf{R})(P^1, Q^1) &:= \text{eq}^1(P^1, Q^1) \vee \exists X^1 (\mathbf{R}(P^1, X^1) \wedge \text{TC}_n(\mathbf{R})(X^1, Q^1))
 \end{aligned}$$

By convention, all quantifiers that occur on the right-hand side of the definitions above are over fresh variables. Above, P^k and Q^k are arity- k predicates, x and y are first-order variables, and \mathbf{R} and \mathbf{S} are combinators.

Let us discuss two of the more interesting combinators: `acyclic` and `TC`. A relation P is acyclic if it is included in a relation that is transitive and irreflexive. We remark that the definition of `acyclic` is carefully chosen: even slight variations can have a strong influence on the runtime of solvers [18]. The combinator `TC` for bounded transitive closure is interesting for another reason: it is higher-order — applying an argument (\mathbf{R}) relation in each step of its expansion. By way of example, let us illustrate its application to the subset combinator `sub`¹.

$$\begin{aligned}
 &\text{TC}_1(\text{sub}^1)(P, Q) \\
 &= \text{eq}^1(P, Q) \vee \exists X (\text{sub}^1(P, X) \wedge \text{TC}_0(\text{sub}^1)(X, Q)) \\
 &= \begin{cases} \forall x_1 (P(x_1) \leftrightarrow Q(x_1)) \vee \\ \exists X (\forall x_2 (P(x_2) \rightarrow X(x_2)) \wedge \text{eq}^1(X, Q)) \end{cases} \\
 &= \begin{cases} \forall x_1 (P(x_1) \leftrightarrow Q(x_1)) \vee \\ \exists X (\forall x_2 (P(x_2) \rightarrow X(x_2)) \wedge \forall x_3 (X(x_3) \leftrightarrow Q(x_3))) \end{cases}
 \end{aligned}$$

In the calculation above, P , Q and X have arity 1.

4 SO Solving through QBF

From a reasoning perspective, SO model-checking is a highly non-trivial task due to quantifiers. In particular, quantifiers over relations, where the size of the search-space alone is daunting. For a universe of size n there are 2^{n^2} possible binary relations, and there are 2^{n^k} possible k -ary relations.⁵

A relation is uniquely characterised by a vector of Boolean values, each determining whether a certain tuple is in the relation or not. This insight lets us formulate a search for a relation as a SAT problem, where a fresh Boolean variable is introduced for any potential tuple in the relation. Even though the translation is exponential, it is a popular method in finite-model finding for first-order logic formulae [13,38,33].

However, in the setting of SO, a SAT solver is insufficient since the input formula may contain alternating quantifiers. We tackle this issue by translating

⁵ Finding constrained finite relations is NEXP-TIME complete [26].

to quantified Boolean formulae (QBF), rather than to plain SAT. The translation is carried out in three stages.

1. each interpreted relation is in-lined as a disjunction of conjunctions over the tuples where the relation holds;
2. first-order quantifiers are expanded into Boolean connectives over the elements of the universe, i.e. $\forall x\phi$ leads to one conjunct for each element of the universe and $\exists x\phi$ leads to one disjunct for each element of the universe;
3. all atoms now are ground and each atom is replaced by a fresh Boolean variable, which is inserted under the same type of quantifier as the atom.

For illustration, consider the formula $\exists X\forall Y\forall z (Y(z) \rightarrow X(z))$ and the universe $A = \{1, 2\}$. The formula requires a set X that is a superset of all sets. Inevitably, X has to be the whole domain. The QBF formulation is $\exists x_1x_2\forall y_1y_2 ((y_1 \rightarrow x_1) \wedge (y_2 \rightarrow x_2))$. Intuitively, rather than talking about a set, we focus on each element separately, which is enabled by the finiteness of the universe. Using QBF enables us to arbitrarily quantify over the sets' elements.

PrideMM enables exporting the QBF formulation into the QCIR format [21], which is supported by a bevy of QBF solvers. However, since most solvers only support prenex form, PrideMM, also additionally prenexes the formula, where it attempts to heuristically minimise the number of quantifier levels.

The experimental evaluation showed that the QFUN solver [17] performs the best on the considered instances, see §6. While the solver performs very well on the J+R litmus tests, a couple of instances were left unsolved. Encouraged by the success of QFUN, we built a dedicated solver that integrates the translation to QBF and the solving itself. The solver represents the formula in dedicated hash-consed data structures (the formulae grow in size considerably). The expansion of first-order variables is done directly on these data structures while also simplifying the formula on the go. The solver also directly supports non-prenex formulation (see [19] for non-prenex QBF solving). The solver applies several preprocessing techniques before expanding the first-order variables, such as elimination of relations that appear only in positive or only in negative positions in the formula.

5 Memory Specification Encodings

In this section, we show that many memory specifications can be expressed conveniently in second-order logic. We represent programs and their behaviours with event structures: this supports the expression of axiomatic specifications such as C++, but also the higher-order specification of J+R. For a given program, its event structure is constructed in a straightforward way: loads give rise to mutually conflicting read events and writes to write events [20]. We express the constraints over event structures with the following vocabulary, shared across all specifications.

Vocabulary. A memory specification decides if a program is allowed to have a certain behaviour. We pose this as a model checking problem, $\mathfrak{A} \models \phi$, where \mathfrak{A}

captures program behaviour and ϕ the memory specification. The vocabulary of \mathfrak{A} consists of the following symbols:

- arity 1: **read**, **write**, **final**
- arity 2: \leq , **conflict**, **justifies**, **sloc**, $=$

Sets **read** and **write** classify read and write events. The symbol **final**, another set of events, identifies the executions that exhibit final register states matching the outcome specified by the litmus test.

Events x and y are in program order, written $x \leq y$, if event x arises from an earlier statement than y in the program text. We have **conflict**(x, y) between events that cannot belong to the same execution; for example, a load statement gives rise to an event for each value it might read, but an execution chooses one particular value, and contains only the corresponding event. We write **justifies**(x, y) when x is a read and y is a write to the same memory location of the same value. We have **sloc**(x, y) when x and y access the same memory location. Identity on events, $\{(x, x) \mid x \in A\}$, is denoted by $=$.

Configurations and Executions. We distinguish two types of sets of events. A *configuration* is a set of events that contains no conflict and is downward closed with respect to \leq ; that is, X is a configuration when $\mathsf{V}(X)$ holds, where the V combinator is defined by

$$\mathsf{V}(X) := \begin{cases} \forall x \forall y \left((X(x) \wedge X(y)) \rightarrow \neg \mathbf{conflict}(x, y) \right) \\ \wedge \forall y \left(X(y) \rightarrow \forall x \left((x \leq y) \rightarrow X(x) \right) \right) \end{cases}$$

We say that a configuration X is an *execution of interest* when every final event is either in X or in conflict with an event in X ; that is, X is an execution of interest when $\mathsf{F}(X)$ holds, where the F combinator is defined by

$$\mathsf{F}(X) := \mathsf{V}(X) \wedge \forall x \left(\begin{array}{l} (\mathbf{final}(x) \wedge \neg X(x)) \rightarrow \\ \exists y (\mathbf{conflict}(x, y) \wedge \mathbf{final}(y) \wedge X(y)) \end{array} \right)$$

Intuitively, we shall put in **final** all the maximal events (according to \leq) for which registers have the desired values.

Notations. In the formulae below, X will stand for a configuration, which may be the execution of interest. Variables Y_{rf} , Y_{co} , Y_{hb} and so on are used to represent the relations that are typically denoted by *rf*, *co*, *hb*, ... Thus, X has arity 1, while Y_{rf} , Y_{co} , Y_{hb} , ... have arity 2.

In what follows, we present four memory specifications: sequential consistency (§ 5.1), release-acquire (§ 5.2), C++ (§ 5.3), and J+R (§ 5.4). The first three can be expressed in \exists SO (and in first-order logic). The last one uses both universal and existential quantification over sets. For each memory specification, we shall see their encoding in second-order logic.

5.1 Sequential Consistency

The SC specification allows all interleavings of threads, and nothing else. It is described by the following SO sentence:

$$\text{SC} := \exists X Y_{co} Y_{rf} (\mathbf{F}(X) \wedge \text{co}(X, Y_{co}) \wedge \text{rf}(X, Y_{rf}) \wedge \text{acyclic}(\mathbf{R}(Y_{co}, Y_{rf})))$$

Intuitively, we say that there exists a coherence order relation Y_{co} and a reads-from relation Y_{rf} which, when combined in a certain way, result in an acyclic relation $\mathbf{R}(Y_{co}, Y_{rf})$. The formula $\text{co}(X, Y_{co})$ says that Y_{co} satisfies the usual axioms of a coherence order with respect to the execution X ; and the formula $\text{rf}(X, Y_{rf})$ says that Y_{rf} satisfies the usual axioms of a reads-from relation with respect to the execution X . Moreover, the formula $\mathbf{F}(X)$ asks that X is an execution of interest, which results in registers having certain values.

$$\text{co}(X, Y_{co}) := \begin{cases} \text{trans}(Y_{co}) \wedge \\ \forall xy \left((X(x) \wedge X(y) \wedge \text{write}(x) \wedge \text{write}(y) \wedge \text{sloc}(x, y) \wedge (x \neq y)) \right. \\ \quad \left. \leftrightarrow (Y_{co}(x, y) \vee Y_{co}(y, x)) \right) \end{cases}$$

$$\text{rf}(X, Y_{rf}) := \begin{cases} \text{inj}(Y_{rf}) \wedge \text{sub}^2(Y_{rf}, \text{justifies}) \wedge \\ \forall y \left((\text{read}(y) \wedge X(y)) \rightarrow \exists x (\text{write}(x) \wedge X(x) \wedge Y_{rf}(x, y)) \right) \end{cases}$$

When X is a potential execution and Y_{co} is a potential coherence-order relation, the formula $\text{co}(X, Y_{co})$ requires that the writes in X for the same location include some total order. Because of the later condition that $\mathbf{R}(Y_{co}, Y_{rf})$ is acyclic, Y_{co} is in fact required to be a total order per location. When X is a potential execution and Y_{rf} is a potential reads-from relation, the formula $\text{rf}(X, Y_{rf})$ requires that Y_{rf} is injective, is a subset of justifies , and relates all the reads in X to some write in X .

The auxiliary relation $\mathbf{R}(Y_{co}, Y_{rf})$ is the union of strict program-order ($<$), reads-from (Y_{rf}), coherence-order (Y_{co}), and the from-reads relation:

$$\mathbf{R}(Y_{co}, Y_{rf})(y, z) := (y < z) \vee Y_{co}(y, z) \vee Y_{rf}(y, z) \vee \exists x (Y_{co}(x, z) \wedge Y_{rf}(x, y))$$

5.2 Release–Acquire

Release–Acquire is a simple relaxed memory specification, which is represented straightforwardly in SO logic. It is captured by the formula RA using the vocabulary established in the definition of SC:

$$\text{RA} := \exists X Y_{co} Y_{rf} \left(\begin{array}{l} \mathbf{F}(X) \wedge \text{co}(X, Y_{co}) \wedge \text{rf}(X, Y_{rf}) \wedge \text{acyclic}(Y_{co}) \\ \wedge \exists Y_{hb} \left(\begin{array}{l} \text{sub}^2(<, Y_{hb}) \wedge \text{sub}^2(Y_{rf}, Y_{hb}) \wedge \text{trans}(Y_{hb}) \\ \wedge \text{irrefl}(Y_{hb}) \wedge \text{irrefl}(\text{seq}(Y_{co}, Y_{hb})) \\ \wedge \text{irrefl}(\text{seq}(\text{inv}(Y_{rf}), \text{seq}(Y_{co}, Y_{hb}))) \end{array} \right) \end{array} \right)$$

The existential SO variable Y_{hb} over-approximates a relation traditionally called happens-before.

5.3 C++

To capture the C++ specification in SO logic, we follow the Herd7 specification of Lahav et al. [24]. Their work introduces necessary patches to the specification of the standard [6] but also includes fixes and adjustments from prior work [4,23]. The specification is more nuanced than the SC and RA specifications and requires additions to the vocabulary of \mathfrak{A} together with a reformulation for efficiency, but the key difference is more fundamental. C++ is a *catch-fire* semantics: programs that exhibit even a single execution with a data race are allowed to do anything — satisfying every expected outcome. This difference is neatly expressed in SO logic:

$$\text{CPP} := \exists X Y_{co} Y_{rf} Y_{\alpha\beta} \left(\begin{array}{l} \text{co}(X, Y_{co}) \wedge \text{rf}(X, Y_{rf}) \wedge \text{hb}(Y_{\alpha\beta}, Y_{rf}) \\ \wedge \text{M}(Y_{\alpha\beta}, Y_{co}, Y_{rf}) \wedge (\text{F}(X) \vee \text{C}(Y_{\alpha\beta}, Y_{rf})) \end{array} \right)$$

The formula reuses $\text{co}(X, Y_{co})$, $\text{rf}(X, Y_{rf})$ and $\text{F}(X)$ and includes three new combinators: $\text{hb}(Y_{\alpha\beta}, Y_{rf})$, $\text{M}(Y_{\alpha\beta}, Y_{co}, Y_{rf})$ and $\text{C}(Y_{\alpha\beta}, Y_{rf})$. $\text{hb}(Y_{\alpha\beta}, Y_{rf})$ constrains a new over-approximation, $Y_{\alpha\beta}$, used for building a transitive relation. $\text{M}(Y_{\alpha\beta}, Y_{co}, Y_{rf})$ captures the conditions imposed on a valid C++ execution, and is the analogue of the conditions applied in SC and RA. $\text{C}(Y_{\alpha\beta}, Y_{rf})$ holds if there is a race in the execution X . Note that the expected outcome is allowed if $\text{F}(X)$ is satisfied or if there is a race and $\text{C}(Y_{\alpha\beta}, Y_{rf})$ is true, matching the catch-fire semantics.

New vocabulary. C++ *Read-modify-write* operations load and store from memory in a single atomic step: a new **rmw** relation links the corresponding reads and writes. C++ *fence* operations introduce new events and the set **fences** identifies them. The programmer annotates each memory access and fence with a *memory order* parameter that sets the force of inter-thread synchronisation created by the access. For each choice, we add a new set: **na**, **rlx**, **acq**, **rel**, **acq-rel**, and **sc**.

Over-approximation in happens before. The validity condition, $\text{M}(Y_{\alpha\beta}, Y_{co}, Y_{rf})$, and races $\text{C}(Y_{\alpha\beta}, Y_{rf})$, hinge on a relation called *happens-before*. We over-approximate transitive closures in the SO logic for efficiency, but Lahav et al. [24] define happens-before with nested closures that do not perform well. Instead we over-approximate a reformulation of happens-before that flattens the nested closures into a single one (see Appendix A).

We define a combinator for happens-before, $\text{HB}(Y_{\alpha\beta}, Y_{rf})$, that is used in $\text{M}(Y_{\alpha\beta}, Y_{co}, Y_{rf})$ and $\text{C}(Y_{\alpha\beta}, Y_{rf})$. It takes as argument an over-approximation of the closure internal to the reformed definition of happens-before, $Y_{\alpha\beta} \cdot \text{hb}(Y_{\alpha\beta}, Y_{rf})$ constrains $Y_{\alpha\beta}$, requiring it to be transitive and to include the conjuncts of the

closure, α and β below.

$$\begin{aligned} \text{HB}(Y_{\alpha\beta}, Y_{rf}) &:= \text{or}(<, \text{seq}(\text{maybe}(<), \text{sw}_{\text{begin}}(Y_{rf}), Y_{\alpha\beta}, \text{sw}_{\text{end}}(Y_{rf}), \text{maybe}(<))) \\ \alpha(Y_{rf}) &:= \text{seq}(\text{sw}_{\text{end}}(Y_{rf}), \text{maybe}(<), \text{sw}_{\text{begin}}(Y_{rf})) \\ \beta(Y_{rf}) &:= \text{seq}(Y_{rf}, \text{rmw}) \\ \text{hb}(Y_{\alpha\beta}, Y_{rf}) &:= \begin{cases} \text{trans}(Y_{\alpha\beta}) \\ \wedge \text{sub}^2(\text{id}, Y_{\alpha\beta}) \wedge \text{sub}^2(\alpha(Y_{rf}), Y_{\alpha\beta}) \wedge \text{sub}^2(\beta(Y_{rf}), Y_{\alpha\beta}) \end{cases} \end{aligned}$$

5.4 Jeffrey–Riely

The J+R memory specification is captured by a sentence JR_n , parametrised by an integer n . Unlike the formulae we saw before, JR_n makes use of three levels of quantifiers ($\exists\forall\exists$), putting it on the third level of the polynomial hierarchy. We begin by lifting⁶ **justifies** from events to sets of events P and Q :

$$\begin{aligned} \text{J}(P, Q) &:= \forall y \left(\begin{array}{l} (\neg P(y) \wedge Q(y) \wedge \text{read}(y)) \\ \rightarrow \exists x (P(x) \wedge \text{write}(y) \wedge \text{justifies}(x, y)) \end{array} \right) \\ \text{AJ}(P, Q) &:= \text{J}(P, Q) \wedge \text{sub}^1(P, Q) \wedge \text{V}(P) \wedge \text{V}(Q) \end{aligned}$$

We read J as ‘justifies’, and AJ as ‘always justifies’. Next, we define what Jeffrey and Riely call ‘always eventually justify’

$$\text{AeJ}_n(P, Q) := \begin{cases} \text{sub}^1(P, Q) \wedge \text{V}(P) \wedge \text{V}(Q) \wedge \\ \forall X (\text{TC}_n(\text{AJ})(P, X) \rightarrow \exists Y (\text{TC}_n(\text{AJ})(X, Y) \wedge \text{J}(Y, Q))) \end{cases}$$

The size of the formula $\text{TC}_n(\text{AeJ}_m)(P, Q)$ we defined above is $\Theta(mn)$. In particular, it is bounded. Finally, we let⁷

$$\text{JR}_n := \exists X (\text{TC}_n(\text{AeJ}_n)(\emptyset, X) \wedge \text{F}(X))$$

and ask solve the model checking problem $\mathfrak{A} \models \text{JR}_n$. Since the formulae above are in MSO, it is sufficient to pick $n := 2^{|A|}$. Since all bounded transitive closures include the subset relation, they are monotonic, and it suffices, in fact, to pick $n := |A|$. For actual solving, we will use this observation.

6 Evaluation

We evaluate our tool in the context of Herd7 [3], which is a standard tool among memory specification researchers for building axiomatic memory specifications. No similar tool exists for higher-order event structure based memory specifications.

⁶ Our definition of J is different from the original one [20]: we require that only new reads are justified, by including the conjunct $\neg P(y)$. Without this modification, our solver’s results disagree with the hand-calculations reported by Jeffrey and Riely; with this modification, the results agree.

⁷ The symbol \emptyset denotes the empty unary relation, as expected.

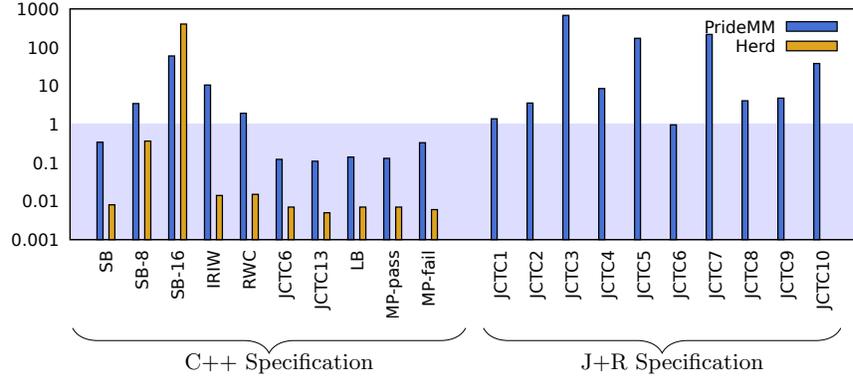


Fig. 4. Comparison of PrideMM’s performance in contrast to Herd7 [3].

6.1 Comparison to existing techniques

In figure Fig. 4 we compare the performance and capabilities of PrideMM to Herd7, the de facto standard tool for building axiomatic memory specifications. Herd7 and PrideMM were both executed on a machine equipped with an Intel i5-5250u CPU and 16 GB of memory. We choose not to compare our tool to MemSAT [39], as there are more memory specifications implemented for Herd7 in the CAT language [2] than there are for MemSAT.

Performance. Notably Herd7’s performance is very favourable in contrast to the performance of PrideMM, however there are some caveats. The performance of PrideMM is largely adequate, with most of the standard litmus tests taking less than 2 seconds to execute. $y \leq 1s$ is highlighted on the chart. We find that our QBF technique scales better than Herd7 with large programs. This is demonstrated in the SB-16 test, a variant of the “store buffering” litmus test with 16 threads. The large number of combinations for quantifying the existentially quantified relations which are explored naïvely by Herd7 cause it to take a long time to complete. In contrast, smarter SAT techniques handle these larger problems handily.

Expressiveness. We split the chart in figure Fig. 4 into 2 sections, the left-hand side of the chart displays a representative subset of common litmus tests showing PrideMM’s strength and weaknesses. These litmus tests are evaluated under the C++ memory specification. Note that these include tests with behaviour expected to be observable and unobservable, hence there being two MP bars. The C++ memory specification is within the domain of memory specifications that Herd7 can solve, as it requires only existentially quantified relations.

The right-hand half of the chart is the first 10 Java causality test cases run under the J+R specification, which are no longer expressible in Herd7. PrideMM solves these in reasonable time, with most tests solved in less than 10 minutes.

Prob.	SAT	caqe (s)	qfun (s)	qfm (s)	Prob.	SAT	caqe (s)	qfun (s)	qfm (s)
1	N	⊥	610	2	10	Y	⊥	36	10
2	N	⊥	23	2	11	Y	⊥	598	335
3	Y	⊥	⊥	222	13	Y	1	1	1
4	Y	⊥	2	5	14	Y	⊥	29	33
5	Y	⊥	78	51	15	Y	⊥	512	157
6	N	5	4	1	16	N	⊥	⊥	12
7	Y	⊥	280	56	17	N	⊥	39	311
8	N	⊥	2	2	18	N	⊥	359	190
9	N	⊥	2	1	#17		#2	#15	#17

Fig. 5. Solver approaches for PrideMM on Java Causality Test Cases. ⊥ represents timeout or mem-out.

Our J+R tests replicate the results found in the original paper, but where they use laborious manual proof in the Agda proof assistant, PrideMM validates the results automatically.

6.2 QBF vs SO Solver Performance

PrideMM enables emitting the SO logic formulae and structures directly for the SO solver, or we can convert to a QBF query (see § 4). This allows us to use our SO solver as well as QBF solvers. We find that the SO solver affords us a performance advantage over the QBF solver in most of the Java causality test cases, where performance optimisations for alternating quantification are applicable.

We include the performance of the QBF solvers CAQE and QFUN, the respective winners of the CNF and non-CNF tracks at 2017’s QBFEVAL competition [32]. Our QBF benchmarks were first produced in the circuit-like format QCIR [21], natively supported by QFUN. The inputs to CAQE were produced by converting to CNF through standard means, followed by a preprocessing step with Bloqqer [7].

We can also emit the structures and formulae as an Isabelle/HOL file, which can then be loaded into Nitpick [8] conveniently. We found that Nitpick cannot be run over the C++ specification or the J+R specification, timing out after 1 hr on all the litmus tests.

7 Related Work

We build on prior work from two different areas — relaxed memory specifications, and SAT/QBF solving: the LISA frontend comes from the Herd7 memory-specification simulator [3], the MSGs implement memory specifications that have been previously proposed [24,20], and the SO solver is based on a state-of-the-art QBF solver [17].

There is a large body of work on finite relational model finding in the context of memory specifications using Alloy [16]. Alloy has been used to compare memory specifications and find litmus tests which can distinguish two specifications [40], and has been used to synthesise comprehensive sets of tests for a specific memory

specification [28]. Applying SAT technology in the domain of evaluating memory specifications has been tried before, too. MemSAT [39] uses Kodkod [38], the same tool that Alloy relies on to do relational model finding. MemSynth [10] uses Ocelot [9] to embed relational logic into the Rosette [37] language. Our results are consistent with the findings of MemSAT and MemSynth: SAT appears to be a scalable and fast way to evaluate large memory specification questions. Despite this, SAT does not widen the class of specifications that can be efficiently simulated beyond ad hoc techniques.

There is work to produce a version of Alloy which can model higher-order constructions, called Alloy* [30], however this is limited in that each higher order set requires a new signature in the universe to represent it. Exponential expansion of the sets quantified in the J+R specification leaves model finding for J+R executions intractable in Alloy* too.

While Nitpick [8] can model higher order constructions, we found it could not generate counter examples in a reasonable length of time of the specifications we built. There is work to build a successor to Nitpick called Nunchaku [34], however, at present Nunchaku does not support higher order quantification. Once Nunchaku is more complete we intend to output to Nunchaku and evaluate its performance in comparison to our SO solver.

There is a bevy of work on finite model finding in various domains. SAT is a popular method for finite model finding in first-order logic formulae [13,33]. There are constraint satisfaction-based model finders, e.g. the SEM model finder [42], relying on dedicated symmetry and propagation. Reynolds et al. propose solutions for finite model finding in the context of SMT [35,36] (CVC4 is in fact used as backend to Nunchaku).

8 Conclusion

This paper presents PrideMM, a case study of the application of new solving techniques to a problem domain with active research. PrideMM allows memory specification researchers to build a new class of memory specifications with richer quantification, and still automatically evaluate these specifications over programs. In this sense we provide a Herd7-style modify-execute-evaluate pattern of development for higher-order memory specifications that were previously unsuitable for mechanised model finding.

References

1. Alglave, J., Cousot, P.: Syntax and analytic semantics of LISA. <https://arxiv.org/abs/1608.06583> (2016)
2. Alglave, J., Cousot, P., Maranget, L.: Syntax and analytic semantics of the weak consistency model specification language CAT. <https://arxiv.org/abs/1608.07531> (2016)
3. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.* **36**(2), 7:1–7:74 (2014). <https://doi.org/10.1145/2627752>, <http://doi.acm.org/10.1145/2627752>
4. Batty, M., Donaldson, A.F., Wickerson, J.: Overhauling SC atomics in C11 and openc11. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 634–648 (2016). <https://doi.org/10.1145/2837614.2837637>, <http://doi.acm.org/10.1145/2837614.2837637>
5. Batty, M., Memarian, K., Nienhuis, K., Pichon-Pharabod, J., Sewell, P.: The problem of programming language concurrency semantics. In: Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. pp. 283–307 (2015). https://doi.org/10.1007/978-3-662-46669-8_12, https://doi.org/10.1007/978-3-662-46669-8_12
6. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. pp. 55–66 (2011). <https://doi.org/10.1145/1926385.1926394>, <http://doi.acm.org/10.1145/1926385.1926394>
7. Biere, A., Lonsing, F., Seidl, M.: Blocked clause elimination for QBF. In: The 23rd International Conference on Automated Deduction CADE (2011)
8. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L.C. (eds.) *Interactive Theorem Proving*. pp. 131–146. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/https://doi.org/10.1007/978-3-642-14052-5_11
9. Bornholt, J., Torlak, E.: Ocelot: A solver-aided relational logic DSL (2017), <https://ocelot.memsynth.org/>
10. Bornholt, J., Torlak, E.: Synthesizing memory models from framework sketches and litmus tests. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 467–481 (2017). <https://doi.org/10.1145/3062341.3062353>, <http://doi.acm.org/10.1145/3062341.3062353>
11. Bove, A., Dybjer, P., Norell, U.: A brief overview of Agda - A functional language with dependent types. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) *Theorem Proving in Higher Order Logics*, 22nd International Conference, TPHOLS 2009, Munich, Germany, August 17-20, 2009. Proceedings. *Lecture Notes in Computer Science*, vol. 5674, pp. 73–78. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_6, https://doi.org/10.1007/978-3-642-03359-9_6
12. Chakraborty, S., Vafeiadis, V.: Grounding thin-air reads with event structures. *PACMPL* **3**(POPL), 70:1–70:28 (2019), <https://dl.acm.org/citation.cfm?id=3290383>

13. Claessen, K., Sörensson, N.: New techniques that improve MACE-style finite model finding. In: Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications (2003)
14. Gray, K.E., Kerneis, G., Mulligan, D.P., Pulte, C., Sarkar, S., Sewell, P.: An integrated concurrency and core-isa architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In: Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015. pp. 635–646 (2015). <https://doi.org/10.1145/2830772.2830775>, <http://doi.acm.org/10.1145/2830772.2830775>
15. ISO/IEC: Programming languages – C++. Draft N3092 (March 2010), <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf>
16. Jackson, D.: Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.* **11**(2), 256–290 (Apr 2002). <https://doi.org/10.1145/505145.505149>, <http://doi.acm.org/10.1145/505145.505149>
17. Janota, M.: Towards generalization in QBF solving via machine learning. In: AAAI Conference on Artificial Intelligence (2018)
18. Janota, M., Grigore, R., Manquinho, V.: On the quest for an acyclic graph. In: RCRA (2017)
19. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.: Solving QBF with counterexample guided refinement. *Artificial Intelligence* **234**, 1–25 (2016). <https://doi.org/http://dx.doi.org/10.1016/j.artint.2016.01.004>
20. Jeffrey, A., Riely, J.: On thin air reads towards an event structures model of relaxed memory. In: Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science. pp. 759–767. LICS '16, ACM, New York, NY, USA (2016). <https://doi.org/10.1145/2933575.2934536>, <http://doi.acm.org/10.1145/2933575.2934536>
21. Jordan, C., Klieber, W., Seidl, M.: Non-CNF QBF solving with QCIR. In: AAAI Workshop: Beyond NP. AAAI Workshops, vol. WS-16-05. AAAI Press (2016)
22. Kang, J., Hur, C., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 175–189 (2017), <http://dl.acm.org/citation.cfm?id=3009850>
23. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 649–662 (2016). <https://doi.org/10.1145/2837614.2837643>, <http://doi.acm.org/10.1145/2837614.2837643>
24. Lahav, O., Vafeiadis, V., Kang, J., Hur, C., Dreyer, D.: Repairing sequential consistency in C/C++11. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 618–632 (2017). <https://doi.org/10.1145/3062341.3062352>, <http://doi.acm.org/10.1145/3062341.3062352>
25. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers* **28**(9), 690–691 (1979). <https://doi.org/10.1109/TC.1979.1675439>, <https://doi.org/10.1109/TC.1979.1675439>
26. Lewis, H.R.: Complexity results for classes of quantificational formulas. *Journal of Computer and System Sciences* **21**(3), 317–353 (1980). [https://doi.org/https://doi.org/10.1016/0022-0000\(80\)90027-6](https://doi.org/https://doi.org/10.1016/0022-0000(80)90027-6), <http://www.sciencedirect.com/science/article/pii/0022000080900276>

27. Libkin, L.: *Elements of Finite Model Theory*. Springer (2004)
28. Lustig, D., Wright, A., Papakonstantinou, A., Giroux, O.: Automated synthesis of comprehensive memory model litmus test suites. In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. pp. 661–675. ASPLOS '17, ACM, New York, NY, USA (2017). <https://doi.org/10.1145/3037697.3037723>, <http://doi.acm.org/10.1145/3037697.3037723>
29. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. pp. 378–391 (2005). <https://doi.org/10.1145/1040305.1040336>
30. Milicevic, A., Near, J.P., Kang, E., Jackson, D.: Alloy*: A general-purpose higher-order relational constraint solver. In: *ICSE (2015)*
31. Pichon-Pharabod, J., Sewell, P.: A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. pp. 622–633 (2016). <https://doi.org/10.1145/2837614.2837616>
32. QBF Eval 2017, http://www.qbflib.org/event_page.php?year=2017
33. Reger, G., Suda, M., Voronkov, A.: Finding finite models in multi-sorted first-order logic. In: Creignou, N., Berre, D.L. (eds.) *Theory and Applications of Satisfiability Testing - SAT*. *Lecture Notes in Computer Science*, vol. 9710, pp. 323–341. Springer (2016). https://doi.org/10.1007/978-3-319-40970-2_20
34. Reynolds, A., Blanchette, J.C., Cruanes, S., Tinelli, C.: Model finding for recursive functions in SMT. In: *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*. pp. 133–151 (2016). https://doi.org/10.1007/978-3-319-40229-1_10
35. Reynolds, A., Tinelli, C., Goel, A., Krstić, S.: Finite model finding in SMT. In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013, Proceedings*. pp. 640–655 (2013). https://doi.org/10.1007/978-3-642-39799-8_42, https://doi.org/10.1007/978-3-642-39799-8_42
36. Reynolds, A., Tinelli, C., Goel, A., Krstić, S., Deters, M., Barrett, C.: Quantifier instantiation techniques for finite model finding in SMT. In: Bonacina, M.P. (ed.) *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013, Proceedings*. *Lecture Notes in Computer Science*, vol. 7898, pp. 377–391. Springer (2013). https://doi.org/10.1007/978-3-642-38574-2_26, https://doi.org/10.1007/978-3-642-38574-2_26
37. Torlak, E., Bodik, R.: A lightweight symbolic virtual machine for solver-aided host languages. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. pp. 530–541. PLDI '14, ACM, New York, NY, USA (2014). <https://doi.org/10.1145/2594291.2594340>, <http://doi.acm.org/10.1145/2594291.2594340>
38. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*. *Lecture Notes in Computer Science*, vol. 4424, pp. 632–647. Springer (2007). https://doi.org/10.1007/978-3-540-71209-1_49

39. Torlak, E., Vaziri, M., Dolby, J.: MemSAT: Checking axiomatic specifications of memory models. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 341–350. PLDI '10, ACM, New York, NY, USA (2010). <https://doi.org/10.1145/1806596.1806635>
40. Wickerson, J., Batty, M., Sorensen, T., Constantinides, G.A.: Automatically comparing memory consistency models. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18–20, 2017. pp. 190–204 (2017), <http://dl.acm.org/citation.cfm?id=3009838>
41. Winskel, G.: Event structures, pp. 325–392. Springer Berlin Heidelberg, Berlin, Heidelberg (1987). https://doi.org/10.1007/3-540-17906-2_31
42. Zhang, J., Zhang, H.: SEM: a system for enumerating models. In: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI. pp. 298–303. Morgan Kaufmann (1995), <http://ijcai.org/Proceedings/95-1/Papers/039.pdf>

Appendix A Reformulation of happens before

Lahav et al. [24] define happens before, **hb**, in terms of *sequenced before* **sb**, the C++ name for program order, and *synchronises with*, **sw**, inter-thread synchronisation. Their **rf** and **rmw** relations match Y_{rf} and **rmw** in our vocabulary. Fixed sequences of memory events initiate and conclude synchronisation, and these are captured by **sw_{begin}** and **sw_{end}**. In the definition below, semicolon represents forward relation composition.

$$\begin{aligned} \mathbf{sw} &:= \mathbf{sw}_{begin}; (\mathbf{rf}; \mathbf{rmw})^*; \mathbf{sw}_{end} \\ \mathbf{hb} &:= (\mathbf{sb} \cup \mathbf{sw})^+ \end{aligned}$$

For efficiency we over-approximate transitive closures in the SO logic, but the nesting over-approximation that follows from the structure of **hb** does not perform well. Instead we over-approximate a reformulation of **hb**.

$$\mathbf{hb}' := \mathbf{sb} \cup (\mathbf{sb}^?; \mathbf{sw}_{begin}; ((\mathbf{sw}_{end}; \mathbf{sb}^?; \mathbf{sw}_{begin})) \cup (\mathbf{rf}; \mathbf{rmw}))^*; \mathbf{sw}_{end}; \mathbf{sb}^?)$$

By unpacking the definition of **sw**, the reformulation flattens the nested closures into a single one. The closure combines fragments of happens before where at the start and end of the fragment, a synchronisation edge has been initiated but not concluded. Within the closure, the synchronisation edge can be concluded and a new one opened, or some number of read-modify-writes can be chained together with **rf**.

We explain the definition of **hb'** by considering the number of **sw** edges that constitute a particular **hb** edge. If a **hb** edge contains no **sw** edge, then because **sb** is transitive, the **hb** edge must be a single **sb** edge. Otherwise, the **hb** edge is made up of a sequence of one or more **sw** edges with **sb** edges before, between and after some of the **sw** edges. The first **sw** edge is itself a sequence of edges starting with **sw_{begin}**. This is followed by any number of **rf; rmw** edges. At the end of the **sw** edge there are two possibilities: this edge was the final **sw** edge, or there is another in the sequence to be initiated next. The first conjunct of the closure, **sw_{end}; sb[?]; sw_{begin}** captures the closing and opening of **sw** edges, the second captures the chaining of read-modify-writes. The end of the definition closes the final **sw** edge with **sw_{end}**.