PrideMM: A Solver for Relaxed Memory Models

Simon Cooksey[1], Sarah Harris[1], Mark Batty[1], Radu Grigore[1], and Mikoláš Janota[2]

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

Abstract. Relaxed memory models are notoriously delicate. To ease their study, several ad hoc simulators have been developed for axiomatic memory models. We show how axiomatic memory models can be simulated using a solver for ∃SO. Further, we show how memory models based on event structures can be simulated using a solver for MSO. Finally, we present a solver for SO, built on top of QBF solvers.

1 Introduction

Understanding processor and language concurrency is an essential step in building reliable systems. Formal modelling and simulation have exposed flaws [4,39,51,52] and led to refinements [1,10] in the official descriptions of concurrency in key languages and processors. Current simulators rely on ad hoc algorithms [5,10,19] or SAT solvers [53]. However, flaws in existing language concurrency models [9] – where one must account for behaviour introduced through aggressive optimisation – have led to a new class of models [27,29] that cannot be simulated with previous ad hoc methods and fit awkwardly in the limited language of SAT, making simulation unworkable.

This paper presents PrideMM, a tool that both simulates the more intricate models of aggressively optimised concurrent languages and replicates the functionality of previous tools. PrideMM identifies second order (SO) logic as expressive enough to capture the wider set of concurrency models, while restrictive enough to enable automatic solving. PrideMM uses a new checker, built above rapidly improving *quantified boolean formula* (QBF) solvers, that solves SO logic formulas directly.

The following contributions underpin PrideMM:

1. we demonstrate simulation of existing models using a solver for ∃SO,
2. we present a model checker for SO, built on top of QBF solvers, and
3. we simulate the Jeffrey and Riely model – one of a new class of concurrency models for optimised concurrent languages – using a solver for SO.

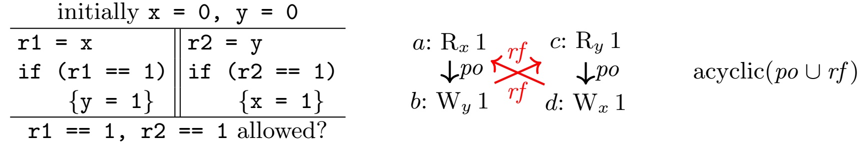

Fig. 1: LB+ctrl, an axiomatic execution of it, and an axiom that forbids it.

1.1 Modelling Relaxed Memory Models

Processor speculation, memory-subsystem reordering and compiler optimisations lead mainstream languages and processors to violate *sequential consistency*, a model of memory where accesses are simply interleaved [33]. We say such systems exhibit *relaxed concurrency.* Relaxed concurrency is commonly described in an *axiomatic* model, where each program behaviour is represented as graph of memory accesses, and a set of axioms filters forbidden execution graphs.

Herd is a simulator of axiomatic models that has been used extensively to model processor, GPU, and language concurrency [5]. In Herd, the model is expressed as a predicate on execution graphs, written in the propositional relation calculus, and recorded in a `.cat` file. Figure 1 presents *load buffering with control dependencies* (LB+ctrl), a small program called a *litmus test* constructed to probe for a single relaxed behaviour, together with an execution graph and an axiom as it would appear in a `.cat` file. LB+ctrl consists of two parallel threads that read `x` (or `y`) and then conditionally write `y` (or `x`), with `x` and `y` initialised to 0. The outcome 1/1 represents a relaxed behaviour, and is allowed in particular by the current C++ standard, but forbidden under the SC, x86, Power and ARM models. The graph of Figure 1 presents the execution in question, with memory reads and writes as vertices (eliding the initialisation) and edges representing program order (*po*) and the writes that each read reads from (*rf*). The axiom of Figure 1 forbids the outcome 1/1 as the corresponding execution contains a cycle in $po \cup rf$. The SC, x86, Power and ARM models each include a variant of this axiom, all forbidding 1/1.

Herd uses an ad hoc algorithm for judging whether an execution is allowed. Its performance is surpassed by the Memalloy [53] tool built above SAT-based Alloy, so it is clear that the judgement of axiomatic models can be expressed as a SAT problem. Unfortunately, not all memory models fit the axiomatic paradigm.

Axiomatic models do not fit optimised languages. Languages like C++ and Java perform dependency-removing optimisations that complicate their memory models. For example, the second thread of the LB+false-dep test in Figure 2 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 dependency of the original would make 1/1 forbidden. It is common practice to use syntactic dependencies to enforce

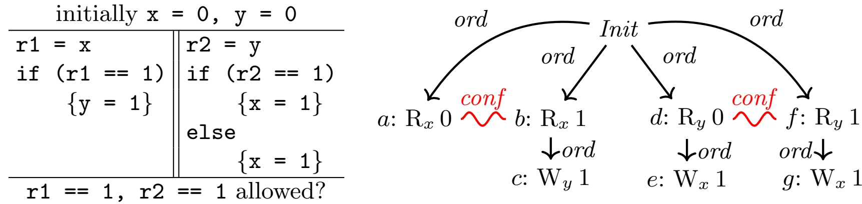

Fig. 2: LB+false-ctrl and the corresponding event structure.

ordering on hardware, but at the language level the optimiser removes these *fake* dependencies.

The C++ standard is flawed because it describes an axiomatic language model that cannot draw a distinction between the executions leading to outcome 1/1 in LB+dep and LB+false-dep: the details of other branches of control flow have been stripped and they have precisely the same vertices and edges [9]

Event structures capture the necessary information. A new class of models aims to fix this by ordering only real dependencies [27, 29, 42]. With a notable exception [29], these models are based on *event structures*, where all paths of control flow are represented in a single graph. Figure 2 presents the event structure for LB+false-deps. Program order is captured by the *ord* relation. *Conflict*, the *conf* edge, links events where only one can occur in an execution (the same holds for their *ord*-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 *ord* or *conf*, 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 2 that regardless of the value read from `y` in the right-hand thread, there is a write to `x` of value 1, i.e. the apparent dependency from the load of `y` is fake and could be optimised away. Memory models built above event structures can recognise this pattern and permit relaxed execution.

The Jeffrey and Riely model. Jeffrey and Riely proposed a concurrency model (henceforth referred to as J+R) built above event structures that correctly identifies fake dependencies [27]. Conceptually, the model is related to the Java memory model [36]: 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 and prevents cycles that could lead to thin-air values. While Java is too strong, the J+R model allows writes that have fake dependencies on a read to be justified before that read. To do this, the model recognises confluence in the program structure: regardless of the execution path, the write will always be made. This search across execution paths involves alternation of quantification that current ad hoc and SAT-based tools cannot efficiently simulate. The problem is amenable to the new breed of QBF solvers.

1.2 Solvers

The late 90's brought about a surge of practical applications of *SAT solvers* [11,38]. QBF provides a more expressive language and therefore less burden on the modeler but it is also inherently harder. Indeed, QBF is PSPACE-complete, whereas SAT is "only" NP-complete. Initially QBF solving mainly focused on adapting SAT techniques to quantifiers [56]. In the last decade, however, there has been a prolific activity in the field leading to several independent paradigms. There has been a remarkable progress in the area almost each year [7, 17, 18, 20, 25, 26, 30, 35, 41, 45–47, 49, 50]. This evolution has also been traced by the yearly QBF competitions [43], see also [37]. These improvements suggest that it may be beneficial to integrate modern QBF technology into formal verification tools.

We highlight the algorithm *RAReQS* [20,24] with its recent improvements [22]. The algorithm has exhibited highly competitive performance in formulas coming from practical applications and with small number of quantifier levels. Hence, RAReQS is a natural candidate for the problems targeted in this paper. Nevertheless, other solvers are also included in the evaluation (see § 6).

From practical perspective, it is important to mention the input format to QBF solvers. Unlike in SAT, CNF input has been observed as extremely harmful to QBF solving [6, 21, 55]. This has been reflected by recent efforts to promote solvers that accept a circuit-like format *QCIR* [28]. Hence, QBF solvers can be classified according to which of the two inputs they support. During the experimental evaluation we have observed that the circuit-based solvers dramatically outperform the CNF-based ones (see § 6).

We should mention that there are other tools dedicated to automated solving in higher-order logic. Namely *higher order model finders* [13] or *automated higher order theorem provers* [15]. Even though one could encode the problems considered in this paper into those tools, their ultimate focus are mainly mathematical theorems. Hence, applying these tools to our problems would likely lead to poor performance: a scenario of a using a sledgehammer to crack a nut.

2 Overview

Figure 3 shows the architecture of our memory-model simulator. The input is LISA code, and the output is a yes/no answer. LISA is a programming language that has been designed for studying memory models [2]. This language enables writing multi-threaded programs and ask questions about whether certain behaviours are allowed. Using LISA as our input format enables a comparison with the state-of-the-art memory-model simulator Herd [5]. The LISA frontend produces an event structure [54]. Any event structure is trivially representable as a SO logic structure, so the conversion is simple. The MM generator (memory-model generator) produces a SO formula. We have a few interchangeable MM generators (§ 4). For some memory models (§ 4.1, § 4.2, § 4.3), which Herd can handle as well, the formula is in fact fixed and does not depend at all on the event structure. For other memory models (such as § 4.4), the MM generator might need to look at

certain characteristics of the event structure (such as its size). Finally, both the second-order structure and the second-order formula are fed into a solver, which effectively simulates the program under the memory-model, and gives a verdict.

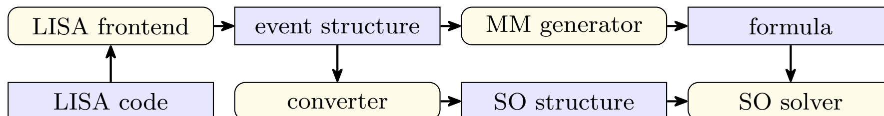

Fig. 3: From a LISA test case to a Y/N answer, given by the SO solver.

We build on prior work from two different areas – relaxed memory models, and SAT/QBF solving: the LISA frontend comes from the Herd memory-model simulator [5], the MM generators implement memory models that have been previously proposed [27, 32], and the SO solver is based on a state-of-the-art QBF solver [22]. Our main contribution to the area of relaxed memory models is that we widen the class of memory models that can be efficiently simulated. Our main contribution to SAT/QBF solving is that we widen the applicability of such tools.

Applying SAT technology to simulate memory models has been tried before [53]. But, although it did lead to performance improvements, it did not widen the class of models that can be efficiently simulated. 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 models is natural. On the other hand, it is simple enough such that it can be solved efficiently, using emerging QBF technology.

Consider for example the sequentially consistent memory model. It is often described by saying that there exists a reads-from relation rf and a coherence order co such that the transitive closure of $\mathit{rf} \cup \mathit{co} \cup (\mathit{rf}^{-1}; \mathit{co}) \cup \mathit{po}$ 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 model as follows. We represent rf and co by existentially-quantified SO arity-2 variables Y_{co} and Y_{rf}, respectively. For example, to say $(x, y) \in \mathit{co}$, we use the formula $Y_{rf}(x, y)$. The program order po is represented by an interpreted arity-2 symbol $<$. Then, the SO formula that represents $\mathit{rf} \cup \mathit{co} \cup (\mathit{rf}^{-1}; \mathit{co}) \cup \mathit{po}$ is

$$\mathsf{R}(y, z) \;\coloneqq\; Y_{rf}(y, z) \vee Y_{co}(y, z) \vee \exists x \left(Y_{rf}(x, z) \wedge Y_{co}(x, y) \right) \vee (y < z) \tag{1}$$

The definition from above should be interpreted as a macro expansion rule: the left-hand side $\mathsf{R}(y, z)$ is a macro 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:

$$\exists Z \left(\mathsf{sub}^2(\mathsf{R}, Z) \wedge \mathsf{trans}(Z) \wedge \mathsf{irrefl}(Z) \right) \tag{2}$$

The macros sub^2, trans, irrefl are defined as one would expect. For example, $\mathsf{sub}^2(P, Q)$, which says that the arity-2 relation P is included in the arity-2

relation Q, is $\forall xy\,\big(P(x,y) \wedge Q(x,y)\big)$. In short, the translation from the usual formulation of memory-models into the SO logic encoding that we propose is natural and almost automatic. In § 4, we describe this translation in detail for 4 memory models. One of these models (§ 4.4) illustrates that the translation is not entirely automatic: some care is required to skirt exponential blowup.

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

Once we have the program and its behaviour represented as a logic structure $\mathfrak{A}$ and the memory model 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. As a foretaste, consider the SO formula

$$\exists X \begin{pmatrix} \forall xy\,\big((\mathtt{ord}(x,y) \wedge X(y)) \to X(x)\big) \wedge \\ \quad \forall xy\,\big((X(x) \wedge X(y)) \to \neg\mathtt{conflict}(x,y)\big) \end{pmatrix} \tag{3}$$

which asks if there exists an execution X that is downward closed with respect to the order *ord* and does not contain conflicting events. We wish to evaluate this formula on a structure $\mathfrak{A}$ defined by

$$A = \{1,2,3\} \qquad \mathtt{ord}^{\mathfrak{A}} \coloneqq \{(1,2),\,(1,3)\} \qquad \mathtt{conflict}^{\mathfrak{A}} \coloneqq \{(2,3)\} \tag{4}$$

This structure contains three events that are partially ordered (with 1 coming first). Events 2 and 3 are conflicting. The QBF question we ask is the following:

$$\forall \mathsf{x}_1\mathsf{x}_2\mathsf{x}_3\,\big((\mathsf{x}_2 \to \mathsf{x}_1) \wedge (\mathsf{x}_3 \to \mathsf{x}_1) \wedge \neg(\mathsf{x}_2 \wedge \mathsf{x}_3)\big) \tag{5}$$

To represent the arity-1 second-order variable X we introduced 3 (Boolean) QBF variables $\mathsf{x}_1, \mathsf{x}_3, \mathsf{x}_3$. In general, an arity-$k$ second-order variable is encoded into $|A|^k$ QBF variables. The first order quantifiers $(\forall xy)$ disappeared altogether, because they were expanded. The relation names `ord` and `conflict` do not appear anymore either, because, once we fixed x and y, we could replace them by true/false constants that were simplified away. For example, $\mathtt{ord}(2,3)$ was replaced by 'false', which was simplified away; and $\mathtt{ord}(1,2)$ was replaced by 'true', which was also simplified away.

Observe that (5) is in fact a SAT instance. This is because the formula (3) does not contain universal second-order variables. When such universal variables are present, they give rise naturally to universal QBF variables.

It is well known that SO finite model-checking can be reduced to QSAT. However, in practice it is important to know how the reduction is done. We give the details of our reduction in § 5. We have implemented this reduction twice, independently. One implementation is built-in the SO solver and optimised; the other implementation is an optional backend for the MM generators. Having two implementations that agree increases our confidence that they are correct.

Further, the MM generator backend produces formulas in the QCIR format, which can be solved using multiple QBF solvers.

We illustrate the generality of our approach by implementing the MM generator component for 4 memory models, including one that cannot be handled by existing simulators. These 4 MM generators are implemented on top of an OCaml API that provides combinators such as sub^2, trans, and irrefl. Since this API has 4 users, we believe it is reusable.

Three of the four memory models we described could be described in the CAT language [3], but not J+R. As future work, we aim to extend the CAT language, and implement a generic MM generator that can handle this extended CAT.

3 Preliminaries

The standard problem solved by simulators is to decide whether a given program behaviour is allowed by a given memory model. The standard model checking problem is to decide whether a given structure $\mathfrak{A}$ satisfies a formula ϕ, written $\mathfrak{A} \models \phi$. We will describe program behaviours by relational structures $\mathfrak{A}$, and memory models by second-order formulas ϕ.

We now recall standard definitions [34]. A (finite, relational) *vocabulary* σ is a finite collection of *constant symbols* (a, b, $\ldots$) together with a finite collection of *relation symbols* (Q, R, $\ldots$). A (finite, relational) *structure* $\mathfrak{A}$ *over vocabulary* σ is a tuple $\langle A, b^{\mathfrak{A}}, c^{\mathfrak{A}}, \ldots, Q^{\mathfrak{A}}, R^{\mathfrak{A}}, \ldots\rangle$ where A is a finite set called *universe* with several distinguished elements $a^{\mathfrak{A}}, b^{\mathfrak{A}}, \ldots$ and relations $Q^{\mathfrak{A}}, R^{\mathfrak{A}}, \ldots$ To simplify the presentation, we will assume that the universe A is $\{a_1^{\mathfrak{A}}, \ldots, a_n^{\mathfrak{A}}\}$, and that the constant symbols include $a_1, \ldots, a_n$, which denote the elements of the universe. For each distinguished relation such as $Q^{\mathfrak{A}}$, there is a k such that $Q^{\mathfrak{A}} \subseteq A^k$; we say that k is the *arity* of $Q^{\mathfrak{A}}$. We assume a countable set of *first-order variables* (x, y, $\ldots$); for each arity $k > 0$, we assume a countable set of *second-order variables* (X^k, Y^k, $\ldots$). In particular, we think of the arity as being part of the variable name, and we single it out only when necessary. 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^k is a second-order variable or a relation symbol. A (second-order) *formula* ϕ is defined inductively: (a) if P^k is a predicate and $t_1, \ldots, t_k$ are terms, then $P^k(t_1, \ldots, t_k)$ is a formula; (b) if ϕ_1 and ϕ_2 are formulas, then $\phi_1 \barwedge \phi_2$ is a formula; (c) if α is a variable and ϕ is a formula, then $\forall\alpha\, \phi$ and $\exists\alpha\, \phi$ are formulas. Other boolean connectives can be desugared into logical-not-and $\barwedge$.

Assume a structure $\mathfrak{A}$ over universe A, a formula ϕ, an environment γ that binds the free first-order variables of ϕ to elements of A, and an environment Γ that binds the free SO variables of ϕ to subsets of A^k, where k is the arity. We use the notation $\gamma[x \mapsto a^{\mathfrak{A}}]$ and $\Gamma[x \mapsto R^{\mathfrak{A}}]$ to extend environments, which we define as $\gamma[x \mapsto a^{\mathfrak{A}}](y) := a^{\mathfrak{A}}$ when $y = x$ and $\gamma(y)$ otherwise. Similar for Γ.

We let the first-order empty environment ϵ map constant symbols to their respective constants $\epsilon(a) := a^{\mathfrak{A}}$, and we let the second-order empty environment E map relation symbols to their respective relations $E(R) := R^{\mathfrak{A}}$. With these

$$\begin{aligned}
\mathsf{sub}^k(P^k,Q^k) &\coloneqq \forall\vec{x}\,\big(P^k(\vec{x}) \to Q^k(\vec{x})\big) & \mathsf{id}(x,y) &\coloneqq (x=y)\\
\mathsf{eq}^k(P^k,Q^k) &\coloneqq \forall\vec{x}\,\big(P^k(\vec{x}) \leftrightarrow Q^k(\vec{x})\big) & \mathsf{inj}(P) &\coloneqq \mathsf{sub}^2\big(\mathsf{seq}(P,\mathsf{inv}(P),\mathsf{id})\big)\\
\mathsf{irrefl}(P) &\coloneqq \forall x\,\neg P(x,x) & \mathsf{seq}(P,Q)(x,z) &\coloneqq \exists y\,\big(P(x,y)\wedge Q(y,z)\big)\\
\mathsf{inv}(P)(x,y) &\coloneqq P(y,x) & \mathsf{trans}(P) &\coloneqq \mathsf{sub}^2\big(\mathsf{seq}(P,P),P\big)
\end{aligned}$$

$$\begin{aligned}
\mathsf{acyclic}(P) &\coloneqq \exists X\,\big(\mathsf{sub}^2(P,X)\wedge\mathsf{trans}(X)\wedge\mathsf{irrefl}(X)\big)\\
\mathsf{TC}_0(\mathsf{R}) &\coloneqq \mathsf{eq}^1\\
\mathsf{TC}_{n+1}(\mathsf{R})(P^1,Q^1) &\coloneqq \mathsf{eq}^1(P^1,Q^1)\vee\exists X^1\,\big(\mathsf{R}(P^1,X^1)\wedge\mathsf{TC}_n(\mathsf{R})(X^1,Q^1)\big)
\end{aligned}$$

Fig. 4: Combinators used to build SO formulas. By convention, all quantifiers that occur on the right-hand side of the definitions above are over fresh variables. Above, P and Q are arity-2 predicates, P^k and Q^k are arity-k predicates, x and y are first-order variables, and R is a combinator.

conventions, we interpret formulas over structures by defining the judgement $\mathfrak{A}\models\phi[\gamma,\varGamma]$ as follows:

$$\begin{aligned}
&\mathfrak{A}\models P(t_1,\ldots,t_k)[\gamma,\varGamma] && \text{iff} && \big(\gamma(t_1),\ldots,\gamma(t_k)\big)\in\varGamma(P)\\
&\mathfrak{A}\models(\phi_1\barwedge\phi_2)[\gamma,\varGamma] && \text{iff} && \text{not both } \mathfrak{A}\models\phi_1[\gamma,\varGamma] \text{ and } \mathfrak{A}\models\phi_2[\gamma,\varGamma]\\
&\mathfrak{A}\models(\forall x\,\phi)[\gamma,\varGamma] && \text{iff} && \mathfrak{A}\models\phi\big[\gamma[x\mapsto a^{\mathfrak{A}}],\varGamma\big] \text{ for all } a^{\mathfrak{A}}\in A\\
&\mathfrak{A}\models(\exists x\,\phi)[\gamma,\varGamma] && \text{iff} && \mathfrak{A}\models\phi\big[\gamma[x\mapsto a^{\mathfrak{A}}],\varGamma\big] \text{ for some } a^{\mathfrak{A}}\in A\\
&\mathfrak{A}\models(\forall X^k\,\phi)[\gamma,\varGamma] && \text{iff} && \mathfrak{A}\models\phi\big[\gamma,\varGamma[X^k\mapsto R^{\mathfrak{A}}]\big] \text{ for all } R^{\mathfrak{A}}\subseteq A^k\\
&\mathfrak{A}\models(\exists X^k\,\phi)[\gamma,\varGamma] && \text{iff} && \mathfrak{A}\models\phi\big[\gamma,\varGamma[X^k\mapsto R^{\mathfrak{A}}]\big] \text{ for some } R^{\mathfrak{A}}\subseteq A^k
\end{aligned}$$

The notation $\mathfrak{A}\models\phi$ is a shorthand for $\mathfrak{A}\models\phi[\epsilon,E]$. A formula with no free variables is called a *sentence*. For a formula ϕ whose free variables are $\vec{\alpha}$, both $\exists\vec{\alpha}\,\phi$ and $\forall\vec{\alpha}\,\phi$ are sentences. We say that ϕ is *satisfiable* when there exists a structure $\mathfrak{A}$ such that $\mathfrak{A}\models\exists\vec{\alpha}\,\phi$; we say that ϕ is *valid* when for all structures $\mathfrak{A}$ we have $\mathfrak{A}\models\forall\vec{\alpha}\,\phi$.

The logic defined so far is known as SO. If we require that all quantifiers over second-order variables are existentials, we obtain a fragment known as ∃SO (existential second-order). If we require that all second-order variables have arity 1, we obtain a fragment known as MSO (monadic second-order). If we make both requirements, the fragment is called ∃MSO.

Combinators. In what follows, we shall be describing some rather large SO formulas. To do so concisely, we shall utilise the combinators from Figure 4. All combinators are typeset in sf-fonts.

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 [23]. The combinator TC for bounded transitive closure is interesting for another reason: it is higher-order. By

way of example, let us illustrate its application to the subset combinator sub^1.

$$\begin{aligned}
&\mathsf{TC}_1(\mathsf{sub}^1)(P,Q)\\
&\quad = \mathsf{eq}^1(P,Q) \lor \exists X \big(\mathsf{sub}^1(P,X) \land \mathsf{TC}_0(\mathsf{sub}^1)(X,Q)\big)\\
&\quad = \begin{cases} \forall x_1 \big(P(x_1) \leftrightarrow Q(x_1)\big) \lor \\ \quad \exists X \big(\forall x_2 \big(P(x_2) \to X(x_2)\big) \land \mathsf{eq}^1(X,Q)\big) \end{cases}\\
&\quad = \begin{cases} \forall x_1 \big(P(x_1) \leftrightarrow Q(x_1)\big) \lor \\ \quad \exists X \big(\forall x_2 \big(P(x_2) \to X(x_2)\big) \land \forall x_3 \big(X(x_3) \leftrightarrow Q(x_3)\big)\big) \end{cases}
\end{aligned}$$

In the calculation above, P, Q and X have arity 1. In what follows, we freely use the combinators from Figure 4 and, occasionally, we define some that are specific to a memory model.

4 Memory Models

In this section, we show that many memory models can be expressed conveniently in second-order logic. Before diving into the details of memory models, let us first discuss briefly the representation we use for programs and their behaviours; namely, event structures. We first describe the theory of event structures (vocabulary and axioms), followed by some useful definitions and notational conventions. We do not describe how event structures are obtained from programs; for that, we refer the reader to [27].

Vocabulary. A memory model decides if a program is allowed to have a certain behaviour. We shall formulate this question as a model checking question, $\mathfrak{A} \models \phi$. The vocabulary of $\mathfrak{A}$ consists of the following symbols:

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

The symbol $=$ always denotes the identity relation on events, $\{\,(x,x) \mid x \in A\,\}$. The symbol $\leq$ corresponds to program order; we have $x \leq y$ when events x and y come from program statements that are ordered in the program text. We have `justifies`(x,y) when x reads the value that y wrote, to the same memory location. We have `conflict`(x,y) when events x and y cannot belong to the same execution; for example, events x and y may model the same read-statement but for different values that are being read. The sets `read` and `write` classify events in the obvious way. We have `sloc`(x,y) when x and y are access the same memory location.

The symbol `final` is not a standard component of event structures. We will make use of it to identify the set of executions that exhibit a behaviour of interest.

Axioms. The theory of event structures is defined by the following axioms:

$$\mathfrak{A} \models \forall x \left(\neg\texttt{read}(x) \vee \neg\texttt{write}(x)\right) \tag{6}$$

$$\mathfrak{A} \models \forall xy \left(\texttt{justifies}(x,y) \to \left(\texttt{write}(x) \wedge \texttt{read}(y)\right)\right) \tag{7}$$

$$\mathfrak{A} \models \forall xy \left(\texttt{conflict}(x,y) \leftrightarrow \texttt{conflict}(y,x)\right) \tag{8}$$

$$\mathfrak{A} \models \forall x \, \neg\texttt{conflict}(x,x) \tag{9}$$

$$\mathfrak{A} \models \forall xyz \left(\left(\texttt{conflict}(x,y) \wedge (y \le z)\right) \to \texttt{conflict}(x,z)\right) \tag{10}$$

$$\mathfrak{A} \models \forall xyz \left(\left(\texttt{conflict}(x,y) \wedge (z < y)\right) \to (z < x)\right) \tag{11}$$

$$\mathfrak{A} \models \forall xyz \begin{pmatrix} \left(\texttt{conflict}(x,y) \wedge \texttt{conflict}(y,z)\right) \\ \to \left(\texttt{conflict}(x,z) \vee (x = z)\right) \end{pmatrix} \tag{12}$$

Intuitively, conflicts can first occur when an event x is immediately followed in program-order by two events y_1 and y_2 which are incomparable to each-other; and once a conflict occurs it propagates to subsequent events. Furthermore, conflict is irreflexive, and becomes transitive when unioned with the identity relation.

Currently, our SO solver has no knowledge of the theory of event structures, so it does not exploit the axioms from above. But, it can check that the structures $\mathfrak{A}$ we produce satisfy the axioms, as they should.

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 $\le$; that is, X is a configuration when $\mathsf{V}(X)$ holds, where the V combinator is defined by

$$\mathsf{V}(X) \;\coloneqq\; \begin{cases} \forall x \forall y \left(\left(X(x) \wedge X(y)\right) \to \neg\texttt{conflict}(x,y)\right) \\ \quad \wedge \, \forall y \left(X(y) \to \forall x \left((x \le y) \to X(x)\right)\right) \end{cases} \tag{13}$$

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

$$\mathsf{F}(X) \;\coloneqq\; \mathsf{V}(X) \wedge \forall x \begin{pmatrix} \left(\texttt{final}(x) \wedge \neg X(x)\right) \to \\ \quad \exists y \left(\texttt{conflict}(x,y) \wedge \texttt{final}(y) \wedge X(y)\right) \end{pmatrix} \tag{14}$$

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

Notations. In the formulas 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 $Y_{rf}, Y_{co}, \ldots$ have arity 2.

In what follows, we present four memory models: sequential consistency (§ 4.1), release–acquire (§ 4.2), C++ (§ 4.3), and J+R (§ 4.4). The first three can be

expressed in ∃SO (and in first-order logic). The last one uses both universal and existential quantification over sets. For each memory model, we shall see their encoding in second-order logic.

4.1 Sequential Consistency

The sequential consistency memory model is the oldest and the least relaxed we consider. Intuitively, this model allows all interleavings of threads, and nothing else. It is described by the following SO sentence:

$$\mathsf{SC} \coloneqq \exists X\, Y_{co}\, Y_{rf}\, \big(\mathsf{F}(X) \wedge \mathsf{co}(X, Y_{co}) \wedge \mathsf{rf}(X, Y_{rf}) \wedge \mathsf{acyclic}(\mathsf{R}(Y_{co}, Y_{rf}))\big)$$

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 $\mathsf{R}(Y_{co}, Y_{rf})$. The formula $\mathsf{co}(X, Y_{co})$ says that Y_{co} satisfies the usual axioms of a coherence order with respect to the execution X; and the formula $\mathsf{rf}(X, Y_{rf})$ says that Y_{rf} satisfies the usual axioms of a reads-from relation with respect to the execution X. Moreover, the formula $\mathsf{F}(X)$ asks that X is an execution of interest, which results in registers having certain values.

$$\mathsf{co}(X, Y_{co}) \coloneqq \forall xy \begin{pmatrix} \big(X(x) \wedge X(y) \wedge \mathtt{write}(x) \wedge \mathtt{write}(y) \wedge \mathtt{sloc}(x,y) \wedge (x \neq y)\big) \\ \leftrightarrow \big(Y_{co}(x,y) \vee Y_{co}(y,x)\big) \end{pmatrix} \tag{15}$$

$$\mathsf{rf}(X, Y_{rf}) \coloneqq \begin{cases} \mathsf{inj}(Y_{rf}) \wedge \mathsf{sub}^2(Y_{rf}, \mathtt{justifies}) \wedge \\ \quad \forall y \Big(\big(\mathtt{read}(y) \wedge X(y)\big) \rightarrow \exists x \,\big(\mathtt{write}(x) \wedge X(x) \wedge Y_{rf}(x,y)\big)\Big) \end{cases} \tag{16}$$

When X is a potential execution and Y_{co} is a potential coherence-order relation, the formula $\mathsf{co}(X, Y_{co})$ requires that the writes in X for the same location includes some total order. Because of the later condition that $\mathsf{R}(Y_{co}, Y_{rf})$ 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 $\mathsf{rf}(X, Y_{rf})$ 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 $\mathsf{R}(Y_{co}, Y_{rf})$ is the union of strict program-order ($<$), reads-from (Y_{rf}), coherence-order (Y_{co}), and the from-reads relation:

$$\mathsf{R}(Y_{co}, Y_{rf})(y,z) \;\coloneqq\; (y < z) \vee Y_{co}(y,z) \vee Y_{rf}(y,z) \vee \exists x \,\big(Y_{co}(x,z) \wedge Y_{rf}(x,y)\big) \tag{17}$$

4.2 Release–Acquire

The Release–Acquire memory model is similar to sequential consistency but more relaxed. The structure it operates has the same vocabulary, and the memory

model is captured by the formula RA, defined as follows:

$$\mathsf{RA} \;\coloneqq\; \exists X\, Y_{co}\, Y_{rf} \left(\begin{array}{l} \mathsf{F}(X) \wedge \mathsf{co}(X, Y_{co}) \wedge \mathsf{rf}(X, Y_{rf}) \wedge \mathsf{acyclic}(Y_{co}) \\ \quad \wedge\, \exists Y_{hb} \left(\begin{array}{l} \mathsf{sub}^2(<, Y_{hb}) \wedge \mathsf{sub}^2(Y_{rf}, Y_{hb}) \wedge \mathsf{trans}(Y_{hb}) \\ \quad \wedge\, \mathsf{irrefl}(Y_{hb}) \wedge \mathsf{irrefl}(\mathsf{seq}(Y_{co}, Y_{hb})) \\ \quad \wedge\, \mathsf{irrefl}(\mathsf{seq}(\mathsf{inv}(Y_{rf}), \mathsf{seq}(Y_{co}, Y_{hb}))) \end{array} \right) \end{array} \right) \tag{18}$$

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

4.3 C++

To capture the C++ model in SO logic, we follow the `.cat` model of Lahav et al. [32]. Their work introduces necessary patches to the model of the standard [10] but also includes fixes and adjustments from prior work [8,31]. The model is more nuanced than the SC and RA models and requires additions to the vocabulary of $\mathfrak{A}$, 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 at all, even burst into flames, and this means that they satisfy every expected outcome. This difference is neatly expressed in SO logic:

$$\mathsf{CPP} \;\coloneqq\; \exists X\, Y_{co}\, Y_{rf} \left(\begin{array}{l} \mathsf{co}(X, Y_{co}) \wedge \mathsf{rf}(X, Y_{rf}) \wedge \mathsf{M}(Y_{co}, Y_{rf}) \\ \quad \wedge\, (\mathsf{F}(X) \vee \mathsf{C}(Y_{co}, Y_{rf})) \end{array} \right) \tag{19}$$

The formula reuses co, rf and $\mathsf{F}(X)$ and includes two new macros: $\mathsf{M}(Y_{co}, Y_{rf})$ and $\mathsf{C}(Y_{co}, Y_{rf})$. $\mathsf{M}(Y_{co}, Y_{rf})$ captures the conditions imposed on a valid C++ execution, and is the analogue of the conditions applied in SC and RA. $\mathsf{C}(Y_{co}, Y_{rf})$ holds if there is a race in the execution X. Note that the expected outcome is allowed if $\mathsf{F}(X)$ is satisfied or if there is a race and $\mathsf{C}(Y_{co}, Y_{rf})$ is true.

4.4 Jeffrey–Riely

The J+R memory model is captured by a sentence JR_n, parametrised by an integer n. Unlike the formulas 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 lifting[3] `justifies` from events to sets of events P and Q:

$$\mathsf{J}(P,Q) \coloneqq \forall y \left(\begin{array}{l} \big(\neg P(y) \wedge Q(y) \wedge \mathtt{read}(y)\big) \\ \quad \rightarrow \exists x \,\big(P(x) \wedge \mathtt{write}(y) \wedge \mathtt{justifies}(x,y)\big) \end{array} \right) \tag{20}$$

$$\mathsf{AJ}(P,Q) \coloneqq \mathsf{J}(P,Q) \wedge \mathsf{sub}^1(P,Q) \wedge \mathsf{V}(P) \wedge \mathsf{V}(Q) \tag{21}$$

[3] Our definition of J is different from the original one [27]: 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.

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

$$\mathsf{AeJ}_n(P,Q) \coloneqq \begin{cases} \mathsf{sub}^1(P,Q) \wedge \mathsf{V}(P) \wedge \mathsf{V}(Q) \wedge \\ \forall X \left(\mathsf{TC}_n(\mathsf{AJ})(P,X) \to \exists Y \left(\mathsf{TC}_n(\mathsf{AJ})(X,Y) \wedge \mathsf{J}(Y,Q)\right)\right) \end{cases} \tag{22}$$

The size of the formula $\mathsf{TC}_n(\mathsf{AeJ}_m)(P,Q)$ we defined above is $\Theta(mn)$. In particular, it is bounded. Finally, we let[4]

$$\mathsf{JR}_n \coloneqq \exists X \left(\mathsf{TC}_n(\mathsf{AeJ}_n)(\emptyset,X) \wedge \mathsf{F}(X)\right) \tag{23}$$

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

5 Encoding in QBF

In the previous section, we saw that deciding whether a given program behaviour is allowed by a given memory model can often be expressed naturally as a model checking problem $\mathfrak{A} \models \phi$ in second-order logic. Now we want to solve such problems. We do not use existing model finders and solvers [13, 15, 16]: we find those for first-order logic are efficient, but not expressive enough; whereas those for higher-order logic are expressive but not efficient. As a middle road, we reduce the model-checking problem in second-order logic to checking the validity of a QBF. This reduction is simple and natural, and it lets us profit from the recent improvements in QBF solving. We first define QBF (§ 5.1) and then present the translation from SO to QBF (§ 5.2).

5.1 Quantified Boolean Formulas

QBF can be seen as a restriction of second-order logic: (i) we banish second-order quantifiers from formulas; and (ii) we fix the structure. The universe contains two elements, $0^{\mathfrak{A}}$ and $1^{\mathfrak{A}}$, denoted by the constant symbols 0 and 1, respectively. There is a unique relational symbol T which denotes the relation $\{1^{\mathfrak{A}}\}$. We denote this fixed structure by $\mathfrak{A}_{\mathsf{qbf}}$. Instead of writing $T(0)$ and $T(1)$ we abuse notation, as is common, and write 0 and 1.

[4] The symbol $\emptyset$ denotes the empty unary relation, as expected.

5.2 Translation from SO to QBF

Given a structure $\mathfrak{A}$ and an SO sentence ϕ, we will construct a QBF sentence $[\![\mathfrak{A} \models \phi]\!]$ such that $\mathfrak{A} \models \phi$ holds if and only if $\mathfrak{A}_{\mathsf{qbf}} \models [\![\mathfrak{A} \models \phi]\!]$ holds:

$$[\![\mathfrak{A} \models P(t_1, \ldots, t_k)]\!]_{\gamma,\Gamma} \coloneqq \Gamma(P)(\gamma(t_1), \ldots, \gamma(t_k)) \tag{24}$$

$$[\![\mathfrak{A} \models \phi_1 \barwedge \phi_2]\!]_{\gamma,\Gamma} \coloneqq [\![\mathfrak{A} \models \phi_1]\!]_{\gamma,\Gamma} \barwedge [\![\mathfrak{A} \models \phi_2]\!]_{\gamma,\Gamma} \tag{25}$$

$$[\![\mathfrak{A} \models \forall x\, \phi]\!]_{\gamma,\Gamma} \coloneqq \bigwedge_{i=1}^{n} [\![\mathfrak{A} \models \phi]\!]_{\gamma[x \mapsto a_i^{\mathfrak{A}}],\Gamma} \tag{26}$$

$$[\![\mathfrak{A} \models \exists x\, \phi]\!]_{\gamma,\Gamma} \coloneqq \bigvee_{i=1}^{n} [\![\mathfrak{A} \models \phi]\!]_{\gamma[x \mapsto a_i^{\mathfrak{A}}],\Gamma} \tag{27}$$

$$\left[\!\!\left[\mathfrak{A} \models \forall X^k\, \phi\right]\!\!\right]_{\gamma,\Gamma} \coloneqq \forall \vec{\mathsf{x}}\, [\![\mathfrak{A} \models \phi]\!]_{\gamma,\Gamma[X^k \mapsto \vec{\mathsf{x}}]} \tag{28}$$

$$\left[\!\!\left[\mathfrak{A} \models \exists X^k\, \phi\right]\!\!\right]_{\gamma,\Gamma} \coloneqq \exists \vec{\mathsf{x}}\, [\![\mathfrak{A} \models \phi]\!]_{\gamma,\Gamma[X^k \mapsto \vec{\mathsf{x}}]} \tag{29}$$

As before, γ maps first-order variables to universe elements. Unlike before, Γ maps SO variables X^k to (total) functions from A^k to QBF terms. For example, $\Gamma(X^2)(a_1^{\mathfrak{A}}, a_2^{\mathfrak{A}})$ is a QBF term. As before, we make the convention that the empty first-order environment maps constants to the elements they denote: $\epsilon(a) \coloneqq a^{\mathfrak{A}}$. For the SO environment, we make the following convention:

$$E(R)(\vec{a}) \coloneqq \begin{cases} 0 & \text{if } \vec{a} \in R^{\mathfrak{A}} \\ 1 & \text{if } \vec{a} \notin R^{\mathfrak{A}} \end{cases} \tag{30}$$

Above, 0 and 1 are QBF constants, and $\vec{a} \in A^k$ where k is the arity of R. The notation $[\![\mathfrak{A} \models \phi]\!]$ is shorthand for $[\![\mathfrak{A} \models \phi]\!]_{\epsilon,E}$.

In (28) and (29), SO quantifiers are handled by introducing $|A|^k$ QBF variables $\vec{\mathsf{x}}$, where k is the arity. The SO environment Γ is extended with a binding from the SO variable X^k to a bijective function from A^k to the fresh variables. In (24), this function is extracted from the environment and applied. Intuitively, the QBF variable $\vec{\mathsf{x}}(a_1^{\mathfrak{A}}, \ldots, a_k^{\mathfrak{A}})$ tracks whether $(a_1^{\mathfrak{A}}, \ldots, a_k^{\mathfrak{A}})$ belongs to X^k.

In (26) and (27), first-order quantifiers are handled by simply expanding them into corresponding boolean connectives. This eager expansion is a potential target for optimisation in the future.

6 Evaluation

The evaluation aims to analyse the performance and correctness of the developed tool. To this end we included "tricky" benchmarks that are studied in the literature and benchmarks for scaling. Additionally, various beckends to the presented tool PrideMM are considered.

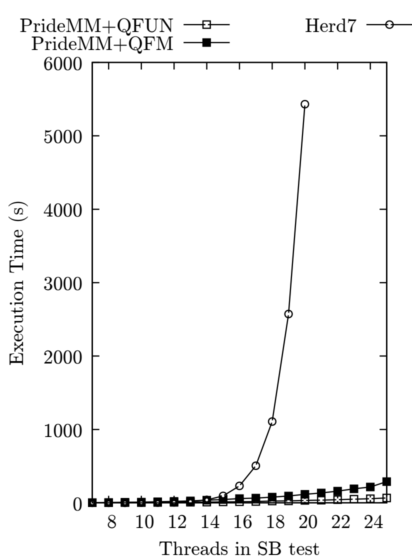

Fig. 5: Comparison between Herd and PrideMM on the store buffer problem.

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

Table 1: CPU time for solving the litmus tests with J+R model; ⊥ represents time/mem-out.

Solvers. We evaluate the QBF approach using off-the-shelf solvers CAQE [46] and QFUN [22], the respective winners of the CNF and non-CNF tracks at 2017's QBFEVAL competition [44]. Our QBF benchmarks were first produced in the circuit-like format QCIR [28], natively supported by QFUN. The inputs to CAQE were produced by converting to CNF through standard means, followed by a preprocessing step with bloqqer [12].

Encouraged by the results of the QBF approach, we have started the development of a dedicated solver for SO model checking. The solver is called *QFM* and it accepts as input a structure and an SO formula. Currently the solver expands all first-order quantifications, following a similar approach to the translation of Section 5.2. The QBF problem is then solved using the non-prenex version of the RAReQS algorithm [24]. A dedicated SO solver is able to use specialised techniques, e.g. lazily expanding quantifiers. Such techniques present a particular advantage for universes with large number of elements: the inherent exponential characteristic of the expansion step will eventually lead to issues in the translation to QBF.

Instances and memory models. In our first set of instances, we simulate a series of n-threaded store-buffering tests (Figure 6) over sequential consistency [33], and compare the performance of PrideMM and Herd7 [5]. The results of this comparison are shown in Figure 5. In a second set of instances, we simulate the J+R model on the Java causality tests [36]. There are no other tools to

initially $\mathtt{x_1} = 0, \mathtt{x_2} = 0, \ldots, \mathtt{x_n} = 0$

$\mathtt{x_1} = 1$	$\mathtt{x_2} = 1$	...	$\mathtt{x_{n-1}} = 1$	$\mathtt{x_n} = 1$
$\mathtt{r_1} = \mathtt{x_n}$	$\mathtt{r_2} = \mathtt{x_1}$	...	$\mathtt{r_{n-1}} = \mathtt{x_{n-2}}$	$\mathtt{r_n} = \mathtt{x_{n-1}}$

$\mathtt{r_1} == 0 \wedge \mathtt{r_2} == 0 \wedge \ldots \wedge \mathtt{r_{n-1}} == 0 \wedge \mathtt{r_n} == 0$ allowed?

Fig. 6: The store-buffer problem.

benchmark against; ours is the only simulator for this model. Instead, we provide a comparative evaluation between our QBF and QFM backends. In a final set of instances, we simulate a collection of standard tests taken from the literature on axiomatic memory models [48]. Each of these completes in under 6s.

Discussion of the results. Figure 5 indicates a stark contrast in the scalability of the store-buffering problem on PrideMM when compared with Herd7. PrideMM enables the practical simulation of far larger tests: 25-thread SB – with 100 events – solves in 1 minute. Axiomatic tests reduce to SAT problems, so one might expect similar performance from QFUN and QFM, but QFUN has the more mature implementation.

Table 1 demonstrates the viability of our approach to simulating the J+R model. QFUN solves all but two instances, whereas QFM solves all of them, taking no longer than 6 min on any instance. We found the CNF-based QBF solver CAQE to be inadequate for these problems. The timeout was set to 30 minutes, and the memory available was 32GB. The dedicated SO solver QFM performs better than the off-the-shelf QBF solver QFUN – even though they implement the same algorithm. We attribute this to a more efficient implementation of the expansion of first-order logic quantifiers (e.g. repetition of subformulas is avoided by hash-consing already during expansion). Additionally, QFM supports non-prenex input, while QFUN operates on prenex form. The satisfiability of each instance matches the expected results [27].

7 Related Work

Our evaluation was limited to 4 memory models: SC, RA, C++ and J+R. Although we have covered a breadth of axiomatic models, there are several others that fall into the class of the J+R model that we have not covered, i.e. the promising model of Kang et al. [29], or the model of Pichon–Pharabod and Sewell (P+S) [42]. It is clear that Promising and P+S are definable in higher-order logic and hence in second-order logic, by the standard encoding of higher-order in second-order (over finite structures). Moreover, for J+R, we do not show that the model definable directly as a second-order logic formula ϕ, but instead describe it as a sequence $\{\phi_n\}_{n\geq 0}$ of formulas, one for each universe size. Thus, our decision to stay in second-order logic and use parametrised formulas does not prevent us from representing other models, and experimental validation indicates that we have found a pragmatic sweet-spot for simulating this new class of models.

We use Herd as a performance benchmark because it is the predominant weak-memory modelling tool, but there are others. CDSChecker [40] is a model checker entirely specialised to the axiomatic model of C++. Memalloy [53] uses SAT solvers to model a range of models, but cannot model the J+R model efficiently.

There are other weak-memory questions that one might seek to answer automatically beyond simulation: Memalloy [53] can compare axiomatic memory models to find programs that act as differentiating counterexamples, with executions allowed by one and not the other. Bornholt and Torlak's MemSynth [14] can synthesise axiomatic memory models from sets of litmus tests. We choose synthesis as our task because it is a good starting point with clear utility.

8 Conclusion

This paper presents PrideMM, a tool that vastly exceeds the performance of Herd, a state-of-the-art simulator for axiomatic concurrency models, and that simulates one of a new class of models for which previous techniques do not apply. We argue that for weak-memory model simulation, SO logic provides a useful balance of expressiveness and performance when combined with state-of-the-art solvers.

References

1. Alglave, J., Batty, M., Donaldson, A.F., Gopalakrishnan, G., Ketema, J., Poetzl, D., Sorensen, T., Wickerson, J.: GPU concurrency: Weak behaviours and programming assumptions. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015. pp. 577–591 (2015), http://doi.acm.org/10.1145/2694344.2694391
2. Alglave, J., Cousot, P.: Syntax and analytic semantics of LISA. https://arxiv.org/abs/1608.06583 (2016)
3. 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)
4. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Fences in weak memory models (extended version). Formal Methods in System Design 40(2), 170–205 (2012), https://doi.org/10.1007/s10703-011-0135-z
5. 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), http://doi.acm.org/10.1145/2627752
6. Ansótegui, C., Gomes, C.P., Selman, B.: The Achilles' heel of QBF. In: AAAI. pp. 275–281 (2005)
7. Balabanov, V., Jiang, J.R., Mishchenko, A., Scholl, C.: Clauses versus gates in CEGAR-Based 2QBF solving. In: Beyond NP, AAAI Workshop (2016)
8. Batty, M., Donaldson, A.F., Wickerson, J.: Overhauling SC atomics in C11 and opencl. 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), http://doi.acm.org/10.1145/2837614.2837637
9. 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
10. 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), http://doi.acm.org/10.1145/1926385.1926394
11. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
12. Biere, A., Lonsing, F., Seidl, M.: Blocked clause elimination for QBF. In: The 23rd International Conference on Automated Deduction CADE (2011)
13. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Interactive Theorem Proving, First International Conference (ITP). pp. 131–146 (2010), https://doi.org/10.1007/978-3-642-14052-5_11
14. 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), http://doi.acm.org/10.1145/3062341.3062353
15. Brown, C.E.: Satallax: An automatic higher-order prover. In: Automated Reasoning - 6th International Joint Conference (IJCAR). pp. 111–117 (2012), https://doi.org/10.1007/978-3-642-31365-3_11
16. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In: TACAS (2014)
17. Goultiaeva, A., Bacchus, F.: Exploiting QBF duality on a circuit representation. In: AAAI (2010)
18. Goultiaeva, A., Seidl, M., Biere, A.: Bridging the gap between dual propagation and CNF-based QBF solving. In: DATE. pp. 811–814 (2013)
19. 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), http://doi.acm.org/10.1145/2830772.2830775
20. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.M.: Solving QBF with counterexample guided refinement. In: SAT. pp. 114–128 (2012)
21. Janota, M., Marques-Silva, J.: An Achilles' heel of term-resolution. In: EPIA Conference on Artificial Intelligence. pp. 670–680 (2017)
22. Janota, M.: Towards generalization in QBF solving via machine learning. In: AAAI Conference on Artificial Intelligence (2018)
23. Janota, M., Grigore, R., Manquinho, V.: On the quest for an acyclic graph. In: RCRA (2017)
24. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.: Solving QBF with counterexample guided refinement. Artificial Intelligence 234, 1–25 (2016)
25. Janota, M., Marques-Silva, J.: Abstraction-based algorithm for 2QBF. In: SAT. pp. 230–244 (2011)

26. Janota, M., Marques-Silva, J.: Solving QBF by clause selection. In: International Joint Conference on Artificial Intelligence (IJCAI) (2015)
27. 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), http://doi.acm.org/10.1145/2933575.2934536
28. 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)
29. 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
30. Klieber, W., Sapra, S., Gao, S., Clarke, E.M.: A non-prenex, non-clausal QBF solver with game-state learning. In: SAT (2010)
31. 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), http://doi.acm.org/10.1145/2837614.2837643
32. 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), http://doi.acm.org/10.1145/3062341.3062352
33. 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
34. Libkin, L.: Elements of Finite Model Theory. Springer (2004)
35. Lonsing, F., Egly, U., Seidl, M.: Q-resolution with generalized axioms. In: Theory and Applications of Satisfiability Testing - SAT. pp. 435–452 (2016)
36. 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), http://doi.acm.org/10.1145/1040305.1040336
37. Marin, P., Narizzano, M., Pulina, L., Tacchella, A., Giunchiglia, E.: Twelve years of QBF evaluations: QSAT is PSPACE-hard and it shows. Fundam. Inform. 149(1-2), 133–158 (2016), https://doi.org/10.3233/FI-2016-1445
38. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers 48(5), 506–521 (1999)
39. Morisset, R., Pawan, P., Nardelli, F.Z.: Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013. pp. 187–196 (2013), http://doi.acm.org/10.1145/2491956.2491967
40. Norris, B., Demsky, B.: A practical approach for model checking c/c++11 code. ACM Trans. Program. Lang. Syst. 38(3), 10:1–10:51 (May 2016), http://doi.acm.org/10.1145/2806886
41. Peitl, T., Slivovsky, F., Szeider, S.: Dependency learning for QBF. In: Theory and Applications of Satisfiability Testing - (SAT). pp. 298–313 (2017), https://doi.org/10.1007/978-3-319-66263-3_19
42. 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), http://doi.acm.org/10.1145/2837614.2837616
43. QBF Eval, http://www.qbflib.org/index_eval.php
44. QBF Eval 2017, http://www.qbflib.org/event_page.php?year=2017
45. Rabe, M.N., Seshia, S.A.: Incremental determinization. In: Theory and Applications of Satisfiability Testing - SAT. pp. 375–392 (2016)
46. Rabe, M.N., Tentrup, L.: CAQE: A certifying QBF solver. In: Formal Methods in Computer-Aided Design, FMCAD. pp. 136–143 (2015)
47. Ranjan, D.P., Tang, D., Malik, S.: A comparative study of 2QBF algorithms. In: SAT. pp. 292–305 (2004)
48. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011. pp. 175–186 (2011), http://doi.acm.org/10.1145/1993498.1993520
49. Tentrup, L.: Non-prenex QBF solving using abstraction. In: Theory and Applications of Satisfiability Testing (SAT). pp. 393–401 (2016)
50. Van Gelder, A.: Primal and dual encoding from applications into quantified boolean formulas. In: CP. pp. 694–707 (2013)
51. Ševčík, J., Aspinall, D.: On validity of program transformations in the Java memory model. In: ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings. pp. 27–51 (2008), https://doi.org/10.1007/978-3-540-70592-5_3
52. Wickerson, J., Batty, M., Beckmann, B.M., Donaldson, A.F.: Remote-scope promotion: clarified, rectified, and verified. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015. pp. 731–747 (2015), http://doi.acm.org/10.1145/2814270.2814283
53. 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
54. Winskel, G.: Event structures, pp. 325–392. Springer Berlin Heidelberg, Berlin, Heidelberg (1987), https://doi.org/10.1007/3-540-17906-2_31
55. Zhang, L.: Solving QBF by combining conjunctive and disjunctive normal forms. In: AAAI (2006)
56. Zhang, L., Malik, S.: Conflict driven learning in a quantified Boolean satisfiability solver. In: International Conference On Computer Aided Design (ICCAD). pp. 442–449 (2002)