BMC with Memory Models as Modules

Hernán Ponce-de-León
fortiss GmbH
Germany
ponce@fortiss.org

Florian Furbach
TU Braunschweig
Germany
f.furbach@cs.tu-bs.de

Keijo Heljanko
University of Helsinki, Aalto University, and HIIT
Finland
keijo.heljanko@iki.fi

Roland Meyer
TU Braunschweig
Germany
roland.meyer@tu-bs.de

***Abstract*—This paper reports progress in verification tool engineering for weak memory models. We present two bounded model checking tools for concurrent programs. Their distinguishing feature is modularity: Besides a program, they expect as input a module describing the hardware architecture for which the program should be verified. DARTAGNAN verifies state reachability under the given memory model using a novel SMT encoding. PORTHOS checks state equivalence under two given memory models using a guided search strategy. We have performed experiments to compare our tools against other memory model-aware verifiers and find them very competitive, despite the modularity offered by our approach.**

Keywords: Memory models, CAT, concurrent programs, bounded model checking, SMT encodings.

I. INTRODUCTION

The semantics of concurrent programs depends on the memory model of the underlying hardware architecture. This has recently seen considerable interest [2], [6], [11], [15], [16], [21], [23], [27], [28], [46], [48]. A key insight is that, for verification purposes, the semantics is best formulated in an axiomatic style. The memory model is given in terms of assertions that constrain a set of candidate executions. A considerable achievement in this line of research is a specification language, CAT [7], [9], [15], in which basically all memory models of interest can be expressed. CAT is made for rapid prototyping. New models are easy to write so that the developer is able to quickly, yet precisely, assess the behavior of the program of interest on the corresponding hardware.

While CAT is successful as a modeling language, the tool support is lagging behind. Memory model-aware verification tools are still being developed for specific memory models. NIDHUGG [2], [6] implements stateless model checking for TSO, POWER, and a version of ARM. CBMC [11] is a bounded model checker for TSO. The RCMC tool [32] targets the C11 programming language. Other verification problems (e.g. fence insertion to restore sequential consistency) are tackled by MEMORAX [3], [4], [5], OFFENCE [13], FENDER [33], and DFENCE [35]. These tools support TSO and similar models, such as PSO or RMO, but cannot handle POWER or ARM.

What is missing are verification tools that are *modular* in the following sense: Besides the program, they should take a memory model as an input and then perform the analysis relative to that model. The HERD tool [15] accompanying CAT satisfies this requirement. Unfortunately, it is designed for litmus tests and limited to small programs.

thread t_0	thread t_1
`x.store(rx, 1)`	`y.store(rx, 1)`
thread t_2	thread t_3
$r_1 =$ `x.load(rx);`	$r_3 =$ `y.load(rx);`
$r_2 =$ `y.load(rx)`	$r_4 =$ `x.load(rx)`

Fig. 1: Program **IRIW**.

We set out to address the need for modular verification and developed two tools. DARTAGNAN is a safety verification engine that checks *reachability* of a (bad) state. It is modular and can handle memory models written in the core subset of the CAT language (see Fig. 4). PORTHOS employs this engine as a back-end and checks *equivalence* of the reachable states under two given memory models.

The following example illustrates how the hardware architecture influences the semantics of a concurrent program in subtle ways and motivates the verification problems. Consider the program **IRIW** given in Fig. 1 which is written in C11. Variables are initially set to 0. The memory order tag `rx` (for relaxed) indicates that an operation provides minimal guarantees w.r.t. the ordering of memory accesses. On X86-TSO [42], each thread has a store buffer of pending stores. When a store is propagated from a buffer to the memory, it becomes visible to all threads simultaneously. POWER, on the other hand, does not guarantee that stores become visible to all threads at the same point in time. With these architectures in mind, consider the following execution: Thread t_2 reads $x = 1, y = 0$ and t_3 reads $x = 0$, $y = 1$. Since under TSO every execution has a unique global view of all store operations, this execution is impossible and a state with $r_1 = 1, r_2 = 0$ and $r_3 = 1, r_4 = 0$ is not reachable. Under POWER, this is possible. The program thus behaves differently under the two memory models.

DARTAGNAN helps programmers find bugs due to unexpected executions. It checks whether a specified (undesirable) state can be reached in the program — relative to a given memory model. Reachability is analyzed with an efficient SMT-based bounded model checking algorithm [17], [24]. The tool computes an acyclic unwinding of the program and translates it, together with the module of the memory model and the specification of the state, into an SMT query. If the query is

satisfiable, the state is reachable. Otherwise it is not.

The challenge is to deal with modularity. It requires us to give an efficient encoding of all operations defined by CAT. Notably, we have to compute — in SMT — least fixpoints. They are used in prominent memory models like POWER and ARM [15]. A naive approach would implement the Kleene iteration in SAT by introducing copies of the variables for each iteration step. In [40], we showed that such an explicit iteration can be avoided by moving to an encoding based on SAT + integer difference logic.

In this paper, we present another improvement to the fixpoint encoding. For reachability, we show it is sound to encode any fixpoint, not necessarily the least one. This is the first technical contribution and implies the encoding from [40] can be simplified. DARTAGNAN implements the idea.

PORTHOS supports programmers in porting code from one architecture (for which it has been thoroughly validated) to another. The portability problem asks whether no new (potentially unsafe) states are introduced and whether all reachable states can still be reached (no functionality has been lost). PORTHOS checks this equivalence for two memory models that are given as modules. If equivalence does not hold, it reports a counterexample execution leading to a reachable state allowed by only one architecture. Equivalence checking is useful when programming performance-critical code for different architectures. Operating System kernel developers and library designers can use equivalence checks to understand whether a programming idiom, an algorithm, or a data structure that is known to work under one memory model can also be used under another.

Note that the assembly versions of the program will be different for the two architectures of interest. We address this by incorporating compiler mappings into our analysis. We return to this when we have our assembly language at hand.

State equivalence is checked in the form of inclusions in both directions. Due to the alternation of quantifiers, inclusion is notoriously difficult to check [49]: For every state reachable in one architecture we have to find an execution in the other that leads to the same state. In [40], we solved the trace inclusion problem and showed that it is easier to solve (in terms of complexity) than state inclusion. Despite that theoretical result, this paper shows that state inclusion can be solved practically using a guided search strategy.

The idea is to be pessimistic and try to disprove the inclusion. The analysis looks for a state that is reachable in one but not in the other model (like the one in the **IRIW** example above). To find states that may disprove the inclusion, PORTHOS invokes an oracle function. This oracle proposes a series of candidate states for which it gives the following guarantees.

(Progress) The series does not contain the same state twice.

(Soundness) If the oracle has no more states to propose, then the inclusion indeed holds.

Progress is certainly desirable and soundness is indispensable for verification. The interesting thing to note is that soundness leaves it to the oracle to terminate early if it finds out, by whatever reasoning, that the inclusion holds.

Our second technical contribution is the implementation of an oracle in SMT which makes progress, is sound, and may terminate early. The idea is to look for so-called *delta executions*: Executions that are inconsistent with one memory model but consistent with the other. Finding a delta execution corresponds to solving the trace inclusion problem. As we showed in [40], this does not require a quantifier alternation and can be done by suitably extending the reachability procedure of DARTAGNAN. A state resulting from a delta execution is clearly a candidate to violate the inclusion. Moreover, if there are no more states resulting from delta executions, the oracle can conclude that the inclusion holds — even if not all reachable states have been considered.

We evaluated the performance of both DARTAGNAN and PORTHOS on a benchmark suite of mutual exclusion algorithms and compared it against several other memory model-aware verification tools. Experiments show that our tools scale significantly better for larger programs.

Contributions: We report progress in memory modular verification in the form of new encoding techniques and oracle heuristics with SMT queries. In particular:

- We present two bounded model checkers for concurrent programs. Both tools are modular: They expect memory models as inputs rather than implementing the analysis for a fixed memory model.
- DARTAGNAN is a reachability checker. It simplifies our previous encoding by admitting arbitrary fixpoints. Its current implementation is an order of magnitude faster than the earlier prototype from [40]. It can be used as a back-end engine for other memory model-aware tools.
- PORTHOS is a portability checker. It implements a new method for checking state inclusion. The algorithm is an oracle-guided search that employs DARTAGNAN as a back-end. The oracle is driven by delta executions. In our experiments it requires only few iterations.
- We perform an exhaustive evaluation of DARTAGNAN and PORTHOS w.r.t. other memory model-aware tools, often observing significant speed ups. This shows the benefits of an SMT-based approach.

Outline: The remainder of the paper is structured as follows. In Section II we describe the user interface of the tools. Section III discusses the BMC for reachability. The guided search for inclusion is described in Section IV. Section V gives the experimental results. The related work is discussed in Section VI.

II. USER INTERFACE

We present our tools from a user's perspective. We examine the verification problems they solve together with the required inputs and their formats. Two verification tasks are supported: Reachability and state equivalence. The solid lines in Fig. 2

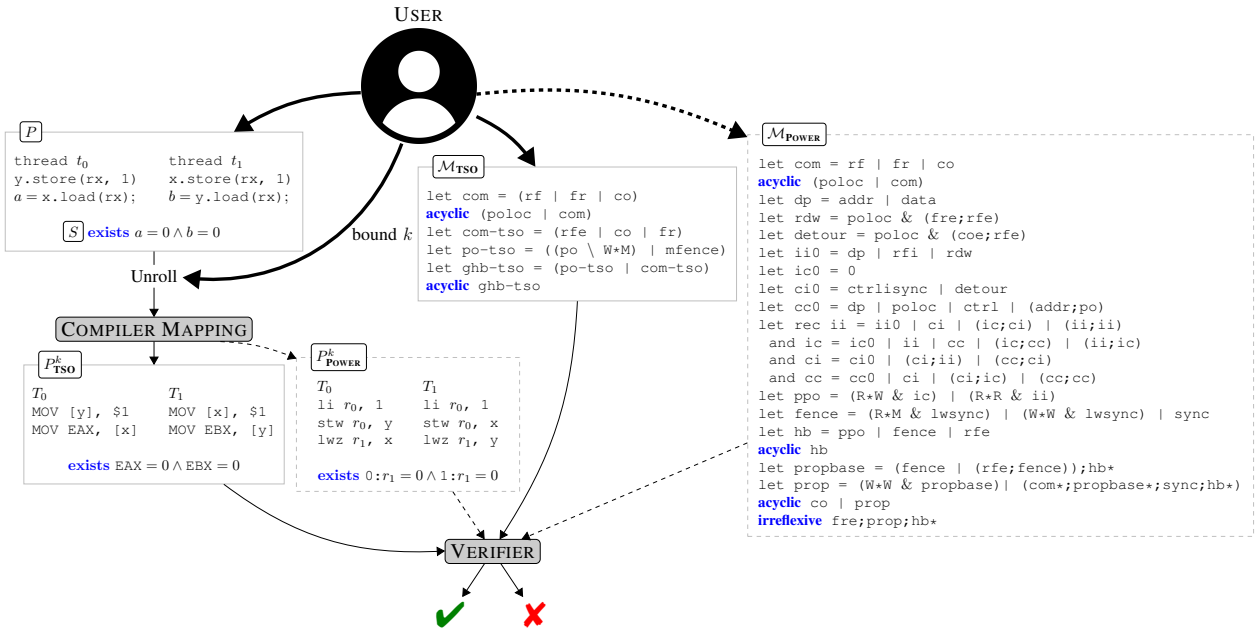

Fig. 2: DARTAGNAN (full arrows) and PORTHOS (full and dotted arrows) from the user's perspective.

illustrate the artifacts that are required for or produced by DARTAGNAN for checking reachability. The complete figure refers to testing for state equivalence with PORTHOS.

Verification Tasks: DARTAGNAN expects a program P annotated with a reachability condition S, a memory model $\mathcal{M}$ of the target architecture, and an unrolling bound k for the bounded model checking. It recursively unwinds all loops in P up to the bound k. The unwound program and the reachability condition are then mapped to the assembly dialect of the target architecture (we elaborate on compiler mappings below). The resulting acyclic and annotated assembly program is handed over to the analysis. In Fig. 2, program P is a simplified mutex algorithm which is mapped to X86 (P^k_{TSO}) using the compiler mapping in Table I. DARTAGNAN then verifies whether $\texttt{EAX} = 0 \wedge \texttt{EBX} = 0$ is reachable when running P^k_{TSO} under TSO. The definition of reachability will be given when we define memory models. In Fig. 2, we verify the mutex algorithm by checking whether both threads can read value 0 and thus enter their critical sections. Under TSO, this is possible.

For checking *equivalence*, PORTHOS expects as input the program P, two memory models $\mathcal{M}_S$ and $\mathcal{M}_T$, and an unrolling bound k. The tool checks whether the reachable states under $\mathcal{M}_T$ are the same as under $\mathcal{M}_S$. This analysis is performed on the unrolled and mapped programs. In Fig. 2, we check if the states reachable by P^k_{POWER} under POWER are the same as the ones reachable by P^k_{TSO} under TSO (which is the case). We process state equivalence queries with two inclusion checks. These queries compare the reachable states of two assembly versions of the same program running under different memory models.

Programs: Both DARTAGNAN and PORTHOS take as input programs written in a C11-like language with support for C11-atomics. Its grammar is given in Fig. 3. Programs consist of a finite number of threads. Each thread contains a sequence of operations such as while and if statements, computations on local variables, and accesses to the shared memory. We currently support Boolean and integer variables in the guards and expressions.

$$
\begin{aligned}
\langle prog \rangle &::= \texttt{program}\ \langle thrd \rangle^* \\
\langle thrd \rangle &::= \texttt{thread}\ \langle tid \rangle\ \langle inst \rangle^+ \\
\langle inst \rangle &::= \langle var \rangle \leftarrow \langle exp \rangle \mid \langle inst \rangle; \langle inst \rangle \\
&\quad \mid \langle var \rangle = \texttt{load}(\langle mem \rangle, \langle atom \rangle) \\
&\quad \mid \langle mem \rangle = \texttt{store}(\langle var \rangle, \langle atom \rangle) \\
&\quad \mid \texttt{while}\ \langle pred \rangle\ \langle inst \rangle \\
&\quad \mid \texttt{if}\ \langle pred \rangle\ \texttt{then}\ \langle inst \rangle\ \texttt{else}\ \langle inst \rangle \\
\langle atom \rangle &::= \texttt{sc} \mid \texttt{rel} \mid \texttt{acq} \mid \texttt{con} \mid \texttt{rx}
\end{aligned}
$$

Fig. 3: Programming language.

Load and store operations are annotated by memory order tags that define their ordering guarantees. The `sc` tag guarantees a sequentially consistent semantics for the access; `rel`/`acq` and `rel`/`con` implement the message-passing idiom; the `rx` (relaxed) tag maps directly to hardware accesses giving minimal guarantees on how those accesses are performed. Weaker guarantees yield higher performance but they usually allow additional program behavior that is hard to predict.

Although the input program is written in a C11-like language, the analysis is performed at the assembly level. The

C11	X86	POWER	ARMv7
Load `rx`	`MOV`	lwz	ldr
Load `con`	`MOV`	lwz; lwsync	ldr; dmb ish
Load `acq`	`MOV`	lwz; lwsync	ldr; dmb ish
Load `sc`	`MOV`	sync; lwz; lwsync	ldr; dmb ish
Store `rx`	`MOV`	stw	str
Store `rel`	`MOV`	lwsync; stw	dmb ish; str
Store `sc`	`MOV; mfence`	sync; stw	dmb ish; str; dmb ish

TABLE I. Compiler mappings for X86, POWER and ARMv7.

program is converted to hardware specific assembly code according to a given compiler mapping. The compiler mapping replaces load and store operations with their corresponding assembly memory accesses and adds fences to enforce the ordering guarantees provided by the memory model tag. Each compiler uses its own mapping. Our tools currently implement the mappings given in Table I, which are the ones used by the LLVM 4.0 compiler [38]. Other mappings, like the one from [1], can be easily added. For the method presented in Section IV to work, the only requirement is that the mapping of each atomic operation contains a single memory access.

It is worth noting that we assume the compiler does not perform any optimization; the program to be verified has already been optimized. Compiler optimizations under weak memory models are an active topic of research [34], [37], [47], [49], but they are out of the scope of this paper.

Memory Models: Informally, a memory model defines when store operations executed by one thread become visible to other threads. This means a memory model determines the semantics of a program on a hardware architecture. The semantics is defined in terms of so-called executions. It contains those executions that are (in a precise sense) consistent with the memory model [7], [36]. We elaborate on the notion of executions and how they define reachability. Afterwards we introduce memory models and consistency.

An execution $(\mathbb{X}, rf, co)$ consists of memory events executed by the program of interest and relations between these events [7], [49]. Set $\mathbb{X}$ states which events have been executed in each thread. This forms the control flow of the program. The reads-from relation rf specifies from which store each load gets its value. The coherence order co is the order in which stores to a variable take effect.

A *state* consists of the values of local and global variables. A state *reached* by a given execution is defined as follows. The value of a global variable is given by the last store operation according to the co relation. The value of a local variable depends on the last executed event (according to the control flow) loading to the local variable.

Memory models define a consistency predicate on executions. The semantics of a program on that memory model is then given by the executions of the program that satisfy the predicate [7], [11], [36]. We use the language CAT [9] to define memory models, the core of which is shown in Fig. 4. There are functional programming features in CAT that we do not support since they are not needed to define the hardware

$$
\begin{aligned}
\langle MCM \rangle &::= \langle assert \rangle \mid \langle rel \rangle \mid \langle MCM \rangle \wedge \langle MCM \rangle \\
\langle assert \rangle &::= acyclic(\langle r \rangle) \mid irreflexive(\langle r \rangle) \mid empty(\langle r \rangle) \\
\langle r \rangle &::= \langle b \rangle \mid \langle r \rangle \cup \langle r \rangle \mid \langle r \rangle \cap \langle r \rangle \mid \langle r \rangle \setminus \langle r \rangle \\
&\quad \mid \langle r \rangle^{-1} \mid \langle r \rangle^{+} \mid \langle r \rangle^{*} \mid \langle r \rangle; \langle r \rangle \\
\langle b \rangle &::= po \mid rf \mid co \mid ad \mid dd \mid cd \mid sthd \mid sloc \\
&\quad \mid mfence \mid sync \mid lwsync \mid isync \mid isb \mid ish \\
&\quad \mid id(\langle set \rangle) \mid \langle set \rangle \times \langle set \rangle \mid \langle name \rangle \\
\langle set \rangle &::= \mathbb{E} \mid \mathbb{W} \mid \mathbb{R} \\
\langle rel \rangle &::= \langle name \rangle := \langle r \rangle
\end{aligned}
$$

Fig. 4: The CAT language [9].

architectures of interest. In CAT, memory models define relations over the events in executions. The program order po and relations rf and co from above are common to all memory models, and typically referred to as base relations. Base relations also include, e.g., address, data and control dependences. Further so-called derived relations are defined using operations on relations such as transitive closure, union, intersection, and composition.

Importantly, CAT allows to define derived relations as least solutions to a system of equations. The semantics of such recursive definitions is well defined only if they behave monotonously [9]. Almost all of CAT is already monotonous, the only non-monotonous construct is the right hand side of the "$\setminus$"-operator. We disallow recursive definitions in the right side of it to ensure well defined semantics in a syntactic manner.

To define the notion of consistency for executions, a memory model requires a number of assertions to hold over its relations. These assertions are acyclicity, irreflexivity and emptiness guarantees. An execution is defined to be consistent with the memory model if it satisfies all assertions.

III. CHECKING REACHABILITY

DARTAGNAN encodes the reachability problem into an SMT formula which is constructed as follows. Formulas ϕ_{CF} and ϕ_{DF} encode the control flow and data flow of the program. The memory model dependent condition $\phi_{\mathcal{M}}$ ensures that the executions are *consistent* with the given model. Finally, ϕ_S is satisfied only if the final state reached by the program satisfies the predicate S. The overall BMC encoding is:

$$\phi_{CF} \wedge \phi_{DF} \wedge \phi_{\mathcal{M}} \wedge \phi_S.$$

Each loop in the program is unrolled up to a user defined depth k. The program is compiled using a given mapping and then converted into its single static assignment (SSA) form. This results in a directed acyclic graph presenting all possible control flows of the program up to the unrolling depth. As the program is now acyclic and in the SSA form, each statement and variable assignment can be executed at most once.

The main idea of the BMC encoding is to guess an execution, which consists of *executed events* and the rf and

co relations. Guessing the executed events fully specifies the control flow of the candidate execution, while guessing rf and co specifies the data-flow of the candidate execution. It is easy to see that this is basically the encoding of the weakest possible memory model expressible in CAT. All widely used models are additional restrictions of this.

The part of the encoding that is not dependent on the memory model is very similar to established BMC encodings of concurrent programs [25]. We recently introduced in [40] the encodings for the memory model specific parts, especially the ones for recursively defined relations with least fixpoint semantics (needed for POWER and ARM).

Encoding Control and Data Flow: Recall that the basic idea for the control flow is to guess the set of executed events. We encode this with a Boolean variable for each event, which is satisfied if the event is executed. We ensure that every load gets its value from one store on the same variable and that the stores to a variable form a total order in co. Relations are encoded as follows. For any pair of events $e_1, e_2 \in \mathbb{E}$ and relation $r \subseteq \mathbb{E} \times \mathbb{E}$ we use a Boolean variable $\mathtt{r}(e_1, e_2)$ representing the fact that $e_1 \xrightarrow{r} e_2$ holds.

The rest of the encoding ensures that the guessed executed events are a valid control flow path through each one of the threads, and that data-flow follows the reads-from and coherence order relations in the shared variables. The encoding also checks that all executed guards are satisfied, and that all executed data manipulation statements are correctly evaluated. The data flow encoding additionally relates the final state of the unrolled compiled program to the original program, allowing the state predicate formula ϕ_S to be expressed in terms of the variables of the original unrolled program before the SSA conversion. Thus, we ensure candidate executions that obey both the control flow and the data flow of the programs. The details of the encodings can be found in [39].

Encoding Memory Models: A memory model defined in the CAT language (see Fig. 4) is a constraint system over so-called derived relations together with some assertions. The language defines a number of base relations. Their encodings can be obtained directly from the source code of the program (e.g., the program order po), from statements corresponding to the synchronization primitives of the used architecture (e.g., memory fences $mfence$ on TSO) or they are part of the execution (the rf and co relations). Derived relations are built from relations using operators such as union, intersection, difference, composition, transitive closure, etc. We similarly use new Boolean variables to represent the derived relations. Most of the operators can be encoded in SMT in a fairly straightforward manner.

An execution is consistent with a memory model if all its assertions are satisfied. We encode acyclicity of a relation in a compact way using IDL by ensuring that a relation implies a partial ordering. We assign each event a numerical variable and require that if an event e is related to e' then the numerical value assigned to e is less than the value assigned to e'.

Encoding Recursive Relations: CAT additionally supports recursive definitions. The semantics of such recursively defined relations are the least fixpoint solution to this system of monotone equations on relations. We argue that for reachability, it is sufficient to encode any fixpoint, not necessarily the least one. The assertions of the memory model (acyclicity, irreflexivity and emptiness) are monotone in the following sense: If a relation fulfills an assertion, all of its subsets will also fulfill the assertion. The CAT operators on relations are also monotone (except set difference which is not applied to recursive relations): Consider $r := (r; r) \cup r_0$, where the operator ";" represents relation composition. If relation r_0 is enlarged or reduced, then so is r.

These observations allow us to apply the Knaster-Tarski Theorem [44]. This is a key contribution of the paper; we use it to simplify the SMT encoding of CAT models. We can freely pick any fixpoint that satisfies all the assertions, as it always contains the least fixpoint, which also satisfies all the assertions. It removes the need to encode the least fixpoints of the CAT language exactly. We call this the relaxed encoding. The encoding of r is simply:

$$\mathtt{r}(e_1, e_2) \Leftrightarrow \mathtt{r;r}(e_1, e_2) \vee \mathtt{r}_0(e_1, e_2).$$

We argue that for reachability queries, this encoding is still correct. Assume a least fixpoint encoding of a reachability query has a satisfying assignment. Naturally, the least fixpoint also satisfies the relaxed encoding as it is a fixpoint. If the least fixpoint encoding is unsatisfiable, every execution violates some assertion. Any violated acyclicity assertion implies a cycle. Since larger fixpoints only add dependencies to relations, the cycle remains for all larger fixpoints. The assertion remains violated with the relaxed encoding. Hence, the relaxed encoding is also unsatisfiable. Similar reasoning also holds for irreflexivity and emptiness violations.

IV. Checking Inclusion

We show how to efficiently check state inclusion. The inclusion requires that for all states reachable in the target memory model $\mathcal{M}_T$ there has to be an execution in the source memory model $\mathcal{M}_S$ reaching the same state. Such a $\forall\exists$-alternation of quantifiers is notoriously difficult to handle for verification tools [49]. A naive approach would iterate over all reachable states. We propose to use an oracle guiding the search by providing relevant candidate states. We present an implementation of the oracle that iterates over far fewer states but preserves completeness. The key observation is that new states always correspond to new executions. Therefore we only need to consider states coming from executions consistent with the target but inconsistent with the source memory model.

The main procedure is described by Algorithm 1. It takes as input a program, two memory models $\mathcal{M}_S, \mathcal{M}_T$[1], and a bound k. The program is first unrolled up to the bound k and converted to to the acyclic assembly programs P_S^k and P_T^k

[1]The latter is needed to implement a concrete oracle. However in Algorithm 1 we consider the oracle a black box object.

Algorithm 1 Incremental SMT Solving for State Inclusion

```
1: procedure PORTHOS(Program P, MCM M_S, M_T, Int k)
2:     φ_RCH ← φ_CF(P_S^k) ∧ φ_DF(P_S^k) ∧ φ_{M_S}(P_S^k)
3:     while ORACLE().hasState() do
4:         s ← ORACLE().getState()
5:         if φ_RCH ∧ φ_s is UNSAT then
6:             return false
7:     return true
```

using the mappings from Table I. The procedure might perform several reachability queries for $\mathcal{M}_S$. Therefore, we construct a formula defining its consistent executions in Line 2. The formulas ϕ_{CF}, ϕ_{DF} and $\phi_{\mathcal{M}_S}$ are the ones from Section III.

The algorithm then enters a loop iterating over a sequence of states which can be thought of as candidates for violating inclusion. These candidate states are provided by an oracle, a black box providing two functions. Function *hasState()* returns a Boolean judging whether there is still a candidate state to consider. If so, function *getState()* provides the candidate. The oracle has to meet the following specification.

(O1) If *hasState()* returns false, then state inclusion holds.
(O2) If *hasState()* returns true, an invocation of *getState()* returns a state.
(O3) Function *getState()* never returns the same state twice.
(O4) Every state returned by *getState()* is reachable in $\mathcal{M}_T$.

When the oracle provides a new candidate, the algorithm checks whether it is reachable in $\mathcal{M}_S$. If the state is not reachable, state inclusion does not hold and the procedure returns false at Line 6. If it is reachable, the check is repeated with a different state. If every state provided by the oracle is reachable under $\mathcal{M}_S$, state inclusion holds by (O1) and the procedure returns true at Line 7.

A correct but naive implementation of an oracle would list all states reachable under $\mathcal{M}_T$. A more efficient exploration is guaranteed by the following idea.

An Oracle for Efficient Exploration: We present an oracle that lists good candidates likely to violate state inclusion. Moreover, the oracle may be able to guarantee state inclusion early. Finally, the computation of candidate states itself is based on SMT-solving and quite efficient. The idea is to find all executions consistent with $\mathcal{M}_T$ but not $\mathcal{M}_S$, and extract their reachable states. This guarantees (O1) and (O4): When *hasState()* returns false, all states that may violate inclusion have been considered and thus state inclusion holds. Our implementation encodes the oracle as follows:

$$\phi_{\text{ORA}} = \phi_{\text{EQ}}(P_S^k, P_T^k) \wedge \phi_{CF}(P_T^k) \wedge \phi_{DF}(P_T^k) \wedge \phi_{\mathcal{M}_T}(P_T^k) \\ \wedge \phi_{CF}(P_S^k) \wedge \phi_{DF}(P_S^k) \wedge \phi_{\neg\mathcal{M}_S}(P_S^k).$$

Function *hasState()* denotes whether the formula ϕ_{ORA} is satisfiable. In this case, *getState()* extracts a state s from a satisfying assignment and returns it. This guarantees (O2). To ensure (O3), the same state is not returned twice, the formula is iteratively updated to $\phi_{\text{ORA}} := \phi_{\text{ORA}} \wedge \neg\phi_s$.

The formula ϕ_{EQ} relates the executions of both assembly programs by ensuring that they represent the same execution of P^k. This formula will be explained below. The next three formulas encode consistent executions in $\mathcal{M}_T$ as defined in Section III. The remaining formulas encode executions inconsistent with $\mathcal{M}_S$.

We encode acyclicity violations by guessing a cycle. For every event e, a Boolean variable $\mathtt{C}_\mathtt{r}(e)$ represents its presence in the cycle. We ensure that every event in the cycle has an incoming and an outgoing edge in the cycle. A more detailed description of the cycle encoding is given in [40].

Encoding Least Fixpoints: When using the relaxed encoding in the oracle, a larger fixpoint could be chosen with more dependencies between events and thus new cycles could be created. This implies that the oracle could propose additional candidate states and more iterations might be required. For this reason, we encode exact least fixpoints for PORTHOS.

Least fixpoints of recursively defined relations can be computed with the standard Kleene iteration [43], which starts from the empty relation and iterates until the least fixpoint is reached. A naive encoding approach would implement the Kleene iteration in SAT by introducing a Boolean variable for each pair of events and each iteration step. This naive encoding is too inefficient, as the number of iterations needed is basically the joint size of the involved relations.

We recently proposed in [40] a much more efficient SMT-encoding that uses Integer Difference Logic [26]. Instead of having a Boolean variable for each iteration step, it only uses one Boolean variable $\mathtt{r}(e_1, e_2)$ (representing if the relation holds) and one numerical variable $\Phi^{\mathtt{r}}_{e_1,e_2}$ representing the iteration in which the pair was added to the relation. Given a relation $r := (r; r) \cup r_0$, for events e_1, e_2 we construct the formula:

$$\mathtt{r}(e_1,e_2) \quad \Leftrightarrow \quad (\mathtt{r;r}(e_1,e_2) \wedge (\Phi^{\mathtt{r}}_{e_1,e_2} > \Phi^{\mathtt{r;r}}_{e_1,e_2})) \\ \vee (\mathtt{r}_0(e_1,e_2) \wedge (\Phi^{\mathtt{r}}_{e_1,e_2} > \Phi^{\mathtt{r}_0}_{e_1,e_2})).$$

The first part of the disjunction specifies that (e_1, e_2) can be added to r if the pair belongs to $r; r$ (i.e. variable $\mathtt{r;r}(e_1, e_2)$ is true) and it was added to $r; r$ at some previous iteration step (i.e. $\Phi^{\mathtt{r}}_{e_1,e_2} > \Phi^{\mathtt{r;r}}_{e_1,e_2}$). The second part is analogous.

Note that this only encodes *at most* the least fixpoint: A satisfying assignment could also set a value for $\Phi^{\mathtt{r}}_{e_1,e_2}$ that is too small and thus not add the pair. We combine the formula above with the relaxed encoding to get exactly the least fixpoint.

Encoding Common Executions: We look for an execution consistent with $\mathcal{M}_T$ and inconsistent with $\mathcal{M}_S$. However, we execute two different assembly programs P_S^k and P_T^k. This means we need a way to compare their executions. Intuitively, two executions are equivalent if they represent the same execution of the program P^k. Since the compilation scheme of Table I implements each atomic memory operation using a single low-level memory access, a one-to-one mapping $\pi : \mathbb{E}_T \to \mathbb{E}_S$ between the events of P_S^k and P_T^k can be

Benchmark	#Executions	TSO				C11	#Executions	POWER			ARM		
	TSO/C11	HERD	NIDHUGG	CBMC	DARTAGNAN	RCMC	POWER/ARM	HERD	NIDHUGG	DARTAGNAN	HERD	NIDHUGG	DARTAGNAN
PARKER	11	0.08	**0.01**	0.29	0.76	**0.08**	14	0.07	**0.01**	1.32	0.08	**0.02**	1.29
DEKKER	24	T/O	**0.02**	0.48	4.29	**0.05**	24	T/O	**0.05**	34.86	T/O	**0.04**	36.88
PETERSON	24	4.98	**0.03**	0.32	0.94	**0.07**	24	4.89	**0.04**	4.29	4.85	**0.03**	4.13
BURNS	47	284.90	**0.02**	0.29	1.10	**0.04**	47	316.33	**0.03**	4.10	289.66	**0.04**	4.05
BAKERY	12492	T/O	2.60	**0.41**	4.64	**0.07**	84760	T/O	141.56	**40.06**	T/O	140.25	**41.83**
LAMPORT	-	T/O	T/O	**0.38**	4.56	T/O	-	T/O	T/O	**72.03**	T/O	T/O	**70.64**
SZYMANSKI	4227148	T/O	966.71	**0.84**	18.98	**409.79**	-	T/O	T/O	**259.56**	T/O	T/O	**241.34**

TABLE II. Reachability of mutual exclusion algorithm under TSO, C11, POWER, and ARM.

defined. Given two events e_S and e_T representing instructions accessing memory in the assembly programs, $\pi(e_T) = e_S$ holds if they both represent the same high-level instruction. Note that such a mapping π can always be defined as long as the compiler implements atomic memory operations with a single memory access. The following encoding relates the executions of both assembly programs:

$$\begin{aligned}\phi_{\text{EQ}} = & \bigwedge_{e\in\mathbb{E}_T} e \in \mathbb{X}_T \Leftrightarrow \pi(e) \in \mathbb{X}_S \\ \wedge & \bigwedge_{e_1,e_2\in\mathbb{E}_T} \mathtt{rf}(e_1,e_2) \Leftrightarrow \mathtt{rf}(\pi(e_1),\pi(e_2)) \\ \wedge & \bigwedge_{e_1,e_2\in\mathbb{E}_T} \mathtt{co}(e_1,e_2) \Leftrightarrow \mathtt{co}(\pi(e_1),\pi(e_2)).\end{aligned}$$

V. EXPERIMENTAL EVALUATION

We implemented the algorithms from Sections III and IV in the DARTAGNAN and PORTHOS tools which use Z3 [29] as the backend SMT solver. Both tools are available from:

https://github.com/hernanponcedeleon/Dat3M.

The tools include the following memory models: SC, TSO, PSO, RMO, ALPHA, POWER, and ARM (v7). Others can be defined in the CAT language.

We compare their performance against several memory model-aware tools. HERD [12] is a tool designed for litmus tests (small programs). It takes CAT files as an input (and thus supports all memory models used in this section). It enumerates all candidate executions and then filters those accepted by the memory model. NIDHUGG [2], [6] performs stateless model checking. It supports TSO, POWER and a simplified version of ARM. CBMC [11] is a Bounded Model Checker with an encoding similar to ours, but it cannot handle recursive definitions efficiently and only supports TSO. For the sake of completeness, we also report results on reachability for C11 using the RCMC tool [32]. This is the memory model of a programming language instead of a hardware architecture and introduces new types of events. Therefore we cannot directly apply our approach to C11. However, the number of executions on C11 coincides with TSO for all programs and we expect our encoding to perform similar to the TSO case.

The tools listed above are designed to test reachability. They allow to reason about one memory model at a time and therefore cannot directly be used to test state inclusion. However, HERD returns information about all final states. We check state inclusion with HERD by computing the reachable states separately for both models (i.e. we run the tool twice) and comparing them afterwards.

Our benchmark suite consists of mutual exclusion algorithms. We unrolled loops twice ($k = 2$) which is sufficient to show that our approach scales better than the other tools for programs with several executions. Programs contains either two or three threads. However their size is reported in terms of the number of consistent executions since the performance of the tools strongly depends on this. The execution times are given in seconds. We set a timeout of 1800 secs for each call to the tools (3600 secs for HERD in the case of inclusion since the tool is run twice). For entries marked as T/O, the timeout was reached.

We performed two sets of experiments: *(i)* Reachability under TSO, C11, POWER and ARM; and *(ii)* the inclusions TSO $\subseteq$ SC, POWER $\subseteq$ TSO, and ARM $\subseteq$ TSO. Inclusion in the other direction (necessary for equivalence) holds by the definition of the memory models. E.g., every state reachable under TSO is also reachable under the weaker models POWER and ARM.

The results on reachability are given in Table II. We present the analysis for unreachable states since it forces all tools to perform a complete exploration and provides the worst case scenario. For TSO, the best results are obtained by NIDHUGG in benchmarks with small number of executions and by CBMC as soon as this number grows. Even though CBMC outperforms DARTAGNAN for TSO, our tool can be at least two orders of magnitude faster than stateless model checking techniques when the number of executions is in the order of millions. See, e.g., LAMPORT which DARTAGNAN solves in less than 5 secs while NIDHUGG and RCMC timeout. For both POWER and ARM, NIDHUGG again outperforms all tools when the number of executions is small. However for benchmarks with a big number of executions (above 80K), DARTAGNAN performs better. For the LAMPORT and SZYMANSKI benchmarks, our tool outperforms NIDHUGG by at least one order of magnitude. Table II suggests that approaches based on SAT/SMT encodings have a lot of potential for large programs. DARTAGNAN can currently handle four million executions in less than 20 secs while NIDHUGG and RCMC need 15 and 6 minutes respectively.

The results on state inclusion are given in Table III. The SAT column reports whether a counterexample to inclusion was found (✔) or not (✘). When HERD returns a result, we report on the number of delta executions (Δ). This corresponds to an upper bound on the maximal number of iterations

Benchmark	TSO ⊆ SC					
	SAT	HERD	PORTHOS	Δ	IT	S.U.
PARKER	✔	**0.15**	0.70	3	1	0.21
DEKKER	✔	T/O	**12.31**	-	1	**>292.44**
PETERSON	✔	9.96	**1.31**	12	1	**7.60**
BURNS	✔	610.65	**2.00**	53	1	**305.32**
BAKERY	✔	T/O	**10.78**	-	2	**>333.95**
LAMPORT	✔	T/O	**10.64**	-	3	**>338.34**
SZYMANSKI	✔	T/O	**101.32**	-	1	**>35.53**
Benchmark	POWER ⊆ TSO					
	SAT	HERD	PORTHOS	Δ	IT	S.U.
PARKER	✔	**0.15**	2.46	3	2	0.06
DEKKER	✘	T/O	**108.89**	-	0	**>33.06**
PETERSON	✘	9.94	**6.33**	0	0	**1.57**
BURNS	✘	578.55	**6.12**	18	1	**94.53**
BAKERY	✘	T/O	**836.44**	-	43	**>4.30**
LAMPORT	-	T/O	T/O	-	-	-
SZYMANSKI	✘	T/O	**940.75**	-	0	**>3.82**
Benchmark	ARM ⊆ TSO					
	SAT	HERD	PORTHOS	Δ	IT	S.U.
PARKER	✔	**0.15**	1.90	3	1	0.07
DEKKER	✘	T/O	**134.43**	-	0	**>26.77**
PETERSON	✘	10.28	**6.51**	0	0	**1.57**
BURNS	✘	546.90	**7.89**	18	1	**69.31**
BAKERY	-	T/O	T/O	-	-	-
LAMPORT	-	T/O	T/O	-	-	-
SZYMANSKI	✘	T/O	**850.44**	-	0	**>4.23**

TABLE III. State inclusion of mutual exclusion algorithms.

PORTHOS might perform. As it can be seen from Table II, in general this number is several orders of magnitude smaller than the total number of executions. The cases reporting zero iterations correspond to the set of executions coinciding for both memory models. For most of the cases, PORTHOS is at least one order of magnitude faster than HERD. For TSO, the speed-up (S.U. column) can be up-to two orders of magnitude.

VI. Related Work

The influence of memory models on the semantics of concurrent programs has been studied at least since 2007. Initially, hardware architectures have been addressed [7], [15], [22], [31], [36], [41], [42], followed by programming languages, in particular C11 and C++11 [18], [19], [34]. Recently, an axiomatic memory model for the Linux kernel has been introduced [14]. These semantic studies form the basis for the development of verification tools.

As of today, none of the following tools (except HERD) consider the description of the memory model as an input. They all implement (at best few) concrete models. NITPICK [20], SATCHECK [30], NEMOSFINDER [50], and MEMSAT [45] use SMT solvers. CBMC had been extended to support TSO and POWER [11] but POWER is no longer supported. CPPMEM [19] and HERD enumerate all executions, making them less scalable. More efficient but technically involved and hard to generalize are Stateless Model Checkers, available for TSO, PSO, POWER, ARM [2], [6] and C11 [32]. TRENCHER [21] looks for trace inclusion bugs between SC and TSO; it underapproximates state inclusion. It can also synthesize fences to enforce SC behaviors. MEMORAX shares this functionality and is complete for reachability under TSO [3], [4], [5]. Trace inclusion can be enforced not only for TSO but also for weaker memory models. The OFFENCE tool [13] does this, although it is limited to restoring SC behaviors of litmus tests. Another fence insertion tool is MUSKETEER [10]. It scales to large programs, but is also restricted to ensuring SC. The FENDER and DFENCE tools [33], [35] use fence insertion to guarantee safety properties. They support TSO, PSO, and RMO.

A modular proof technique has been introduced recently [8]. It uses invariants to verify programs under a model given in CAT. Another tool based on CAT synthesizes programs differentiating two memory models [49]. However, this tool is of interest to memory model designers and not made for verification.

PORTHOS was originally designed to check trace inclusion. In [40], we showed that state inclusion has a higher complexity than trace inclusion. As a consequence, there is no polynomial encoding that reduces inclusion to a single SAT query. However, the experiments in Section V show that our oracle-based heuristic still performs well in programs where an exhaustive state exploration does not scale.

VII. Conclusion and Outlook

We have presented DARTAGNAN and PORTHOS, two modular Bounded Model Checkers for concurrent programs. The tools can check reachability and state equivalence under any (pair of) memory model(s) defined in the CAT language. Our method reduces reachability to satisfiability of a SMT formula using novel encoding techniques. Equivalence is tested using a guided search. We propose to use an oracle to find relevant candidate states, and show how to implement an efficient oracle based on SMT queries. We have performed experiments to compare our tools to several memory model-aware tools, and find them at least one order of magnitude faster for large programs.

We are currently developing methods to synthesize memory models from reachability results using our encoding techniques. The techniques include compact representations of relations by predicates as well as approximations of operations that are not precise but still sound.

Other verification tasks, such as synthesizing programs to compare memory models, could in principle also be solved by reducing them to SMT queries. We would like to explore this in the future.

Modern compilers perform various optimizations when mapping high-level code to assembly instructions. We plan to investigate whether such compiler mappings can be extracted from the compilation process, at least approximately.

Acknowledgements: We thank Natalia Gavrilenko for constructive feedback on the manuscript and the tool implementation.

References

[1] C/C++11 mappings to processors. https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html. Accessed: 23.04.2018.

[2] Parosh A. Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos F. Sagonas. Stateless model checking for TSO and PSO. In *TACAS*, volume 9035 of *LNCS*, pages 353–367. Springer, 2015.

[3] Parosh A. Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Automatic fence insertion in integer programs via predicate abstraction. In *SAS*, volume 7460 of *LNCS*, pages 164–180. Springer, 2012.

[4] Parosh A. Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Counter-example guided fence insertion under TSO. In *TACAS*, volume 7214 of *LNCS*, pages 204–219. Springer, 2012.

[5] Parosh A. Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Memorax, a precise and sound tool for automatic fence insertion under TSO. In *TACAS*, volume 7795 of *LNCS*, pages 530–536. Springer, 2013.

[6] Parosh A. Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. Stateless model checking for POWER. In *CAV*, volume 9780 of *LNCS*, pages 134–156. Springer, 2016.

[7] Jade Alglave. *A Shared Memory Poetics*. Thèse de doctorat, L'université Paris Denis Diderot, 2010.

[8] Jade Alglave. Simulation and invariance for weak consistency. In *SAS*, pages 3–22, 2016.

[9] Jade Alglave, Patrick Cousot, and Luc Maranget. Syntax and semantics of the weak consistency model specification language CAT. *CoRR*, abs/1608.07531, 2016.

[10] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence - A static analysis approach to automatic fence insertion. In *CAV*, volume 8559 of *LNCS*, pages 508–524. Springer, 2014.

[11] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *CAV*, volume 8044 of *LNCS*, pages 141–157. Springer, 2013.

[12] Jade Alglave and Luc Maranget. The diy7 tool suite. http://diy.inria.fr/.

[13] Jade Alglave and Luc Maranget. Stability in weak memory models. In *CAV*, volume 6806 of *LNCS*, pages 50–66. Springer, 2011.

[14] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan S. Stern. Frightening small children and disconcerting grown-ups: Concurrency in the linux kernel. In *ASPLOS*, pages 405–418. ACM, 2018.

[15] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.

[16] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. In *POPL*, pages 7–18. ACM, 2010.

[17] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[18] Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and OpenCL. In *POPL*, pages 634–648. ACM, 2016.

[19] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *POPL*, pages 55–66. ACM, 2011.

[20] Jasmin Christian Blanchette, Tjark Weber, Mark Batty, Scott Owens, and Susmit Sarkar. Nitpicking C++ concurrency. In *PPDP*, pages 113–124, 2011.

[21] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against TSO. In *ESOP*, volume 7792 of *LNCS*, pages 533–553. Springer, 2013.

[22] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *PLDI*, pages 12–21. ACM, 2007.

[23] Sebastian Burckhardt and Madanlal Musuvathi. Effective program verification for relaxed memory models. In *CAV*, volume 5123 of *LNCS*, pages 107–120. Springer, 2008.

[24] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[25] Hélène Collavizza and Michel Rueher. Exploration of the capabilities of constraint programming for software verification. In *TACAS*, volume 3920 of *LNCS*, pages 182–196. Springer, 2006.

[26] Scott Cotton, Eugene Asarin, Oded Maler, and Peter Niebert. Some progress in satisfiability checking for difference logic. In *FORMATS*, volume 3253 of *LNCS*, pages 263–276. Springer, 2004.

[27] Andrei M. Dan, Yuri Meshman, Martin T. Vechev, and Eran Yahav. Predicate abstraction for relaxed memory models. In *SAS*, volume 7935 of *LNCS*, pages 84–104. Springer, 2013.

[28] Andrei M. Dan, Yuri Meshman, Martin T. Vechev, and Eran Yahav. Effective abstractions for verification under relaxed memory models. In *VMCAI*, volume 8931 of *LNCS*, pages 449–466. Springer, 2015.

[29] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[30] Brian Demsky and Patrick Lam. Satcheck: Sat-directed stateless model checking for sc and tso. In *OOPSLA*, pages 20–36, 2015.

[31] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the ARMv8 architecture, operationally: Concurrency and ISA. In *POPL*, pages 608–621. ACM, 2016.

[32] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Effective stateless model checking for C/C++ concurrency. *PACMPL*, 2(POPL):17:1–17:32, 2018.

[33] Michael Kuperstein, Martin T. Vechev, and Eran Yahav. Automatic inference of memory fences. *SIGACT News*, 43(2):108–123, 2012.

[34] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *PLDI*, pages 618–632. ACM, 2017.

[35] Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin T. Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. In *PLDI*, pages 429–440. ACM, 2012.

[36] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for POWER multiprocessors. In *CAV*, volume 7358 of *LNCS*, pages 495–512. Springer, 2012.

[37] Yatin A. Manerkar, Caroline Trippel, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Counterexamples and proof loophole for the C/C++ to POWER and armv7 trailing-sync compiler mappings. *CoRR*, abs/1611.01507, 2016.

[38] Robin Morisset and Francesco Zappa Nardelli. Partially redundant fence elimination for x86, ARM, and Power processors. In *CC*, pages 1–10. ACM, 2017.

[39] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. Portability analysis for axiomatic memory models. PORTHOS: One tool for all models. *CoRR*, abs/1702.06704, 2017. Extended version of [40].

[40] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. Portability analysis for weak memory models. PORTHOS: One tool for all models. In *SAS*, volume 10422 of *Lecture Notes in Computer Science*, pages 299–320. Springer, 2017.

[41] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *PLDI*, pages 175–186. ACM, 2011.

[42] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *POPL*, pages 379–391. ACM, 2009.

[43] Viggo Stoltenberg-Hansen, Edward R. Griffor, and Ingrid Lindstrom. *Mathematical Theory of Domains*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1994.

[44] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955.

[45] Emina Torlak, Mandana Vaziri, and Julian Dolby. MemSAT: Checking axiomatic specifications of memory models. In *PLDI*, pages 341–350. ACM, 2010.

[46] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*, pages 691–707. ACM, 2014.

[47] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *POPL*, pages 209–220. ACM, 2015.

[48] Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA*, pages 867–884. ACM, 2013.

[49] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. Automatically comparing memory consistency models. In *POPL*, pages 190–204. ACM, 2017.

[50] Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *IPDPS*. IEEE Computer Society, 2004.