Lock and Fence When Needed: State Space Exploration + Static Analysis = Improved Fence and Lock Insertion

Sander de Putter and Anton Wijs[0000−0002−2071−9624]

Eindhoven University of Technology, The Netherlands
{s.m.j.d.putter, a.j.wijs}@tue.nl

Abstract. When targeting modern parallel hardware architectures, constructing correct and high-performing software is complex and time-consuming. In particular, reorderings of memory accesses that violate intended sequentially consistent behaviour are a major source of bugs. Applying synchronisation mechanisms to repair these should be done sparingly, as they negatively impact performance.
In the past, both static analysis approaches and techniques based on explicit-state model checking have been proposed to identify where synchronisation fences have to be placed in a program. The former are fast, but the latter more precise, as they tend to insert fewer fences. Unfortunately, the model checking techniques suffer a form of state space explosion that is even worse than the traditional one.
In this work, we propose a technique using a combination of state space exploration and static analysis. This combination is in terms of precision comparable to purely model checking-based techniques, but it reduces the state space explosion problem to the one typically seen in model checking. Furthermore, experiments show that the combination frequently outperforms both purely model checking and static analysis techniques. In addition, we have added the capability to check for atomicity violations, which is another major source of bugs.

1 Introduction

When developing parallel software it is very challenging to guarantee the absence of bugs. Achieving the intended execution order of instructions while obtaining high performance is extremely hard. This is particularly the case on parallel hardware architectures where memory accesses may be reordered. Reorderings that break the intended sequential and atomic behaviour are a major source of bugs [31]. These can be avoided by appropriately using synchronisation mechanisms such as fences, semaphores, hardware-level atomic operations, and software/hardware transactional memory [37]. However, overusing these can cause contention, as experimentally demonstrated in [4], which negatively impacts performance and therefore defeats the purpose of using parallelism in the first place.

Sequential consistency is arguably the best understood concurrency model. An (execution) trace of a concurrent program is *Sequentially Consistent* (SC) iff all memory accesses are performed in program order, atomicity constraints are respected, and accesses of all threads are serviced as if from a single First In First Out queue [27]. As this model does not deviate from the software developers' specification it is very intuitive.

SC is very restrictive and does not benefit from modern compiler and processor optimisations. It is sufficient, however, that traces produce results that are *observably* equivalent to SC traces. Traces that do not do this, which we refer to as *non-SC* traces, produce results different from SC traces, i.e., they read and write combinations of values from/ to memory locations that an SC trace cannot produce. Such traces violate the behaviour intended by the software developer.

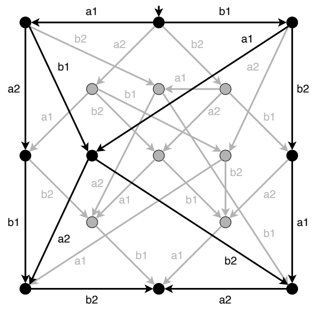

Fig. 1. Interleaving under weak memory models

In earlier work, either explicit-state model checking or static analysis was used to detect non-SC behaviour. Static analysis techniques [4–6, 16, 19, 28, 39], based on the seminal work of Shasha and Snir [36], estimate the possible SC behaviour, and derive which violations can occur when accesses are reordered according to what the memory model allows. These techniques are fundamentally limited regarding their accuracy. In particular, when pointers and guards (if-statements) are used, over-approximations cannot in general be avoided (are two pointers pointing to the same object? And when does the condition of an if-statement evaluate to **true**?).

An alternative is to use explicit-state model checking for fence insertion [1,2,8,14,23,25,29]. In those approaches, program specifications are extended to allow the model checker to systematically traverse *all* behaviour allowed by the memory model. For instance, to specify the memory store behaviour via thread-local caches, allowed by *Total Store Order* (TSO) [33,38] and weaker models, additional store buffers must be modelled. The benefit of these approaches is their accuracy. However, their main issue is scalability; exploring non-SC executions makes the state space explosion problem even worse. Not surprisingly, very weak memory models such as ARMv7 [7] and POWER [22] have not yet been considered. In Fig. 1, all possible interleavings are given for four memory accesses performed by two threads: the first should execute $a1$ followed by $a2$, the second should execute $b1$ followed by $b2$. Starting at the initial state (the one with the incoming arrow), the typical diamond of interleavings is indicated by the black states and transitions between them. This diamond represents all possible SC traces. If the memory model under consideration can also reorder $a2$ before $a1$ and reorder $b2$ before $b1$, then the grey states and transitions also need to be explored. Clearly, as the number of accesses and threads increases, the number of states grows even more rapidly than when only considering SC traces.

Contributions. We propose to combine the state space exploration approach used by techniques based on explicit-state model checking, with the core concept of static analysis approaches, the latter working as a *postprocessing* procedure, to keep the precision of model checking while restricting state space exploration to SC traces. First, we apply a model checker to explore the state space of a program specification. This specification describes all possible interleavings of program instructions, and its state space contains all possible SC executions, as is common for model checking.[1] This state space is used to extract an abstract event graph (AEG), which more accurately represents conflicts

[1]In order for this analysis to terminate, it is important that the system is *finite-state*, or at least has a finite-state quotient that can be derived prior to state space generation [32]. It may perform infinite executions, though, i.e., have cyclic behaviour between its states.

between the memory accesses of instructions than statically derived ones. Next, this graph is analysed using our postprocessing tool. Experiments show that constructing the AEG via state space exploration benefits the overall runtime performance, even compared to a state-of-the-art static analysis approach.

Besides this different workflow, compared to earlier work, we also support the specification of *atomic instructions*, thereby incorporating the atomicity checking originally addressed in [36]. When code does not enforce atomicity, non-SC behaviour can still occur, and only inserting fences does not suffice. In our approach, sets of accesses that are supposed to be executed atomically, i.e., without interruption by other threads, are marked for synchronisation if they are involved in non-SC behaviour. In the final code, these atomic instructions can be enforced by means of locks.

Our technique is the first *model checking-based* technique to support memory models as weak as ARMv7. We have implemented it by translating state machine specifications to mCRL2, such that the mCRL2 model checking toolset [15] can be used, and developed a new tool for the postprocessing. We demonstrate our technique for the memory models TSO, *Partial Store Order* (PSO) [38] and ARMv7/POWER, but it can be straightforwardly adapted to other memory models. During postprocessing, the technique reasons about the (un)safety of thread program paths by means of path rewrite rules.

Finally, it should be stressed that we reason about programs in which initially *no synchronisation primitives* are used. Therefore, the current work is *not* about reasoning about the semantics and SC guarantees of fences, atomic instructions, transactions, etc. When we talk about (atomic) instructions, we are referring to computation steps that are *specified* as atomic; whether they need to be implemented using some synchronisation primitive still needs to be determined. We acknowledge that in order to make an implementation correct, it is crucial to know the semantics and guarantees of those primitives; there are excellent studies on those for Java, C/C++11, and OpenCL [11, 12, 26], and various architectures [7, 33, 35], and applying their insights can be seen as the next step. Here, we focus on detecting the need for synchronisation. Earlier work on SC violation checking, such as [4, 6, 29], reasons about synchronisation primitives as well, but this is not really needed if one entirely relies on automatic insertion of synchronisation points.

This work is also not about optimally implementing synchronisation. An architecture may provide several types of fences, with various guarantees and performance penalties. We aim to optimise the number of places in which a program needs synchronisation, not how that should be implemented. Related work [4] provides methods for that.

Structure of the paper. In Section 2, we consider parallel program specifications, and recall the notions of SC, access conflicts and cycle detection in AEGs. We define the notion of (un)safety of thread program paths in Section 3, and present how we extract an AEG from a state space constructed by an explicit-state model checker in Section 4. The non-SC detection procedure is discussed in Section 5. Experimental results are discussed in Section 6. Finally, conclusions and pointers to future work are given in Section 7.

2 Preliminaries

Parallel program specifications. We assume that a parallel program involving shared variables is specified. Fig. 2 gives an overview of the concepts related to such a program used in this work. A program consists of a set of threads $\mathbb{T}$, each performing a (possibly infinite) sequence of *instructions* called a *thread program*. Such a specification can, for instance, be given as a PROMELA [21] or MCRL2 model [15], in which the processes represent the threads, and each instruction represents an atomic execution step. Each instruction is either of the form e, $e; x_1 = e_1; \ldots; x_n = e_n$ or $x_1 = e_1; \ldots; x_n = e_n$ ($n \geqslant 1$), with e and the e_i *expressions*, and the x_i memory locations. First, an optional *condition* e is checked (an expression evaluating to **true** or **false**). If the condition evaluates to **true**, or there is no condition, zero or more *assignments* can be performed that assign the outcome of interpreting e_i to x_i. For brevity, we do not define the form of expressions here (the usual logical and arithmetic operators can be used to combine variable references), and we assume that expressions are type correct. The data types we consider are Integer, Boolean, and arrays of Integers or Booleans. Extending this basic setup is not fundamentally relevant for the purpose of the current paper.

Program behaviour. To reason about SC behaviour of a concurrent program we consider its execution on, and its effects on the memory of, a multi-core machine. To this end we assume that the state of the machine is defined by the values stored in its storage locations. The set of storage *locations* of a machine is denoted by $\mathbb{L}$. A location may be a register or a memory location (associated to some variable). Fig. 2 presents that an instruction executes zero or more *accesses* to locations.

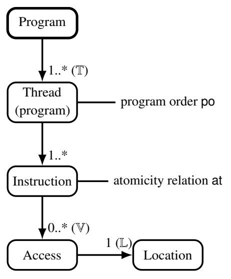

Fig. 2. Overview of a program

An *access* reads from, or writes to, a location $x \in \mathbb{L}$ and reads or writes a value. We write Rx and Wx to refer to a read and write access from/to x, respectively. When both the type (read or write) and the location are irrelevant, we use $a, b, \ldots$. An access is performed *atomically*, i.e., two accesses on the same location behave as if they occur in some serial order. The set of accesses of a thread $t \in \mathbb{T}$ is denoted by $\mathbb{V}_t$. The set of all accesses of a program is defined as $\mathbb{V} = \bigcup_{t \in \mathbb{T}} \mathbb{V}_t$.

For SC checking, only shared memory locations are relevant, i.e., those that can be accessed by multiple threads. Therefore, when we refer to memory locations, it is always implied that they are shared, and accesses always address shared locations.

The execution order of accesses in the program, or the *program order*, is defined by a per-thread total order $\mathsf{po} \subseteq \mathbb{V} \times \mathbb{V}$. Accesses of different threads are unrelated, hence po is the union of the (thread-local) total execution orders of accesses of all the threads.

The instructions of a specification are defined using an equivalence relation $\mathsf{at} \subseteq \mathbb{V} \times \mathbb{V}$ identifying classes of accesses that are to be performed (observably) atomically. Like po, at only relates accesses of the same thread. With $[\alpha]$, we refer to the (atomic) set of accesses associated with an instruction α. With $[e]$ consisting of read accesses for all locations referenced in expression e, we define $[\alpha]$ as $[e; x_1 = e_1; \ldots; x_n = e_n] = [e] \cup \bigcup_{1 \leqslant i \leqslant n} \{\mathrm{W}x_i\} \cup [e_i]$ and $[x_1 = e_1; \ldots; x_n = e_n] = \bigcup_{1 \leqslant i \leqslant n} \{\mathrm{W}x_i\} \cup [e_i]$. With $\langle\!\langle \alpha \rangle\!\rangle$, we refer to the set of read accesses performed to evaluate the condition of α: $\langle\!\langle e; x_1 = e_1; \ldots; x_n = e_n \rangle\!\rangle = [e]$ and $\langle\!\langle x_1 = e_1; \ldots; x_n = e_n \rangle\!\rangle = \emptyset$.

A *thread execution trace* π is a sequence of accesses $a \prec_t b \prec_t \ldots$, with $\prec_t$ a irreflexive, antisymmetric, non-transitive binary relation, describing the order in which accesses of thread t are (visibly) performed. The set of accesses in π is called $[\pi]$. A *program execution trace* is an interleaving of thread execution traces, ordered by $\prec$ (which is also, for convenience in Section 4, non-transitive).

Programmers *rely* on a programming paradigm where executions appear to be interleavings of instructions, and the instructions appear to be executed in programmed order without interruption. That is, they expect their program to be *sequentially consistent* [27]. The following definition is based on the one given by Shasha and Snir [36].

Definition 1 (Sequential Consistency). *A program execution trace $\pi = a{\prec}b{\prec}\ldots$ is* sequentially consistent (SC) *iff $\prec$ can be extended to a total order $\ll$ that satisfies:*

1. $\mathsf{po} \subseteq \ll$*, so that if a po b then $a{\ll}b$;*
2. at*-equivalent accesses occur in consecutive places in the sequence defined by $\ll$: if a at b but $\neg(a$ at $c)$, then either $c{\ll}a$ and $c{\ll}b$, or $a{\ll}c$ and $b{\ll}c$.*

By condition 2 of Def. 1, an SC trace retains atomicity of instructions.

In practice, it is sufficient that a trace is observably equivalent to an SC trace, which is the case if it computes the same values as an SC trace. The potential values of a location $x \in \mathbb{L}$ may depend on the order in which two accesses on x are executed. In this case, those accesses are said to be in conflict. In the literature, various types of conflicts have been formalised [6,7], but for fence insertion analysis, the symmetric *competing pairs* relation (cmp) [4], stemming from data race detection algorithms [24], typically suffices. We have a cmp b iff a and b access the same location and at least one of them is a write. A trace π is observably equivalent to an SC trace π' if we can obtain π' by commuting consecutive accesses in π that are not conflicting.

We recall the SC violation detection theory of Shasha and Snir [36], rephrased to be in line with recent work [4,6,7]. To detect non-SC behaviour, an *abstract event graph* (AEG) for a program can be constructed, in which the nodes are accesses and edges represent po and cmp. Shasha and Snir prove that cycles in the AEG with at least one cmp-edge and one po-edge represent all possibilities for non-SC executions if we run the program on an architecture without the guarantee that po is always respected (i.e., that has a weak memory model).

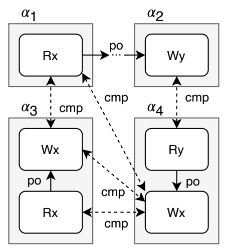

Fig. 3. Example cycles

Consider the cycle $\sigma_1 = \alpha_1 : \mathrm{R}x \xrightarrow{\mathsf{po}} \cdots \xrightarrow{\mathsf{po}} \alpha_2 : \mathrm{W}y \overset{\mathsf{cmp}}{\dashrightarrow} \alpha_4 : \mathrm{R}y \xrightarrow{\mathsf{po}} \alpha_4 : \mathrm{W}x \overset{\mathsf{cmp}}{\dashrightarrow} \alpha_1 : \mathrm{R}x$ in Fig. 3, with $\alpha : a$ indicating access a of instruction α. The grey boxes indicate the atomic instructions, i.e., they define at. Given the direction of σ_1, the involved accesses would lead to non-SC behaviour if either the trace $\pi_1 = \alpha_4 : \mathrm{W}x \prec \alpha_1 : \mathrm{R}x \prec \alpha_2 : \mathrm{W}y \prec \alpha_4 : \mathrm{R}y$ or the trace $\pi_2 = \alpha_2 : \mathrm{W}y \prec \alpha_4 : \mathrm{R}y \prec \alpha_4 : \mathrm{W}x \prec \alpha_1 : \mathrm{R}x$ is executed, since they contradict the po-order between $\alpha_4 : \mathrm{R}y$ and $\alpha_4 : \mathrm{W}x$, and between $\alpha_1 : \mathrm{R}x$ and $\alpha_2 : \mathrm{W}y$, respectively. That both traces are non-SC can be seen when trying to obtain an SC trace by commuting consecutive, non-conflicting accesses. For instance, in π_1, the accesses of α_4 would need to be commuted next to each other, but $\alpha_1 : \mathrm{R}x$ and $\alpha_2 : \mathrm{W}y$ prevent $\alpha_4 : \mathrm{W}x$ from moving to the right and $\alpha_4 : \mathrm{R}y$ from moving to the left, respectively, due to conflicts, unless we reorder $\alpha_1 : \mathrm{R}x$ and $\alpha_2 : \mathrm{W}y$, but then those instructions would be removed from each other and the trace would still not be SC. Traces π_1 and π_2 are not possible on architectures respecting po. However, as we discuss in Section 3, for some architectures, a compiler may ignore the po-order of some accesses, thereby breaking the cycle, and making non-SC traces possible. In [4,6,28], a po-path from access a to access b is called *unsafe* for an architecture if the latter's memory model allows b to be executed before a. The remedy to prevent non-SC behaviour is to *enforce* unsafe po-paths that are part of a cycle, by placing a delay, i.e., indicating that synchronisation is required, somewhere along the unsafe po-path.[2] The delays ensure the cycle remains.

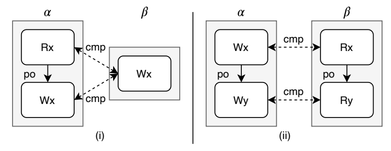

Fig. 4. Atomicity breaking examples

Shasha and Snir demonstrate that in the analysis, redundant work can be avoided by only considering cycles that are *simple* (they involve each vertex and edge at most once) and that have no chords. A chord is an edge between vertices in the cycle that are not each other's neighbour in the cycle. For instance, in Fig. 3, we also have cycles $\sigma_2 = \alpha_1 : \mathrm{R}x \xrightarrow{\text{po}} \cdots \xrightarrow{\text{po}} \alpha_2 : \mathrm{W}y \xrightarrow{\text{cmp}} \alpha_4 : \mathrm{R}y \xrightarrow{\text{po}} \alpha_4 : \mathrm{W}x \xrightarrow{\text{cmp}} \alpha_3 : \mathrm{W}x \xrightarrow{\text{cmp}} \alpha_1 : \mathrm{R}x$ and $\sigma_3 = \alpha_1 : \mathrm{R}x \xrightarrow{\text{po}} \cdots \xrightarrow{\text{po}} \alpha_2 : \mathrm{W}y \xrightarrow{\text{cmp}} \alpha_4 : \mathrm{R}y \xrightarrow{\text{po}} \alpha_4 : \mathrm{W}x \xrightarrow{\text{cmp}} \alpha_3 : \mathrm{R}x \xrightarrow{\text{po}} \alpha_3 : \mathrm{W}x \xrightarrow{\text{cmp}} \alpha_1 : \mathrm{R}x$, but both have the chord $\alpha_4 : \mathrm{W}x \xrightarrow{\text{cmp}} \alpha_1 : \mathrm{R}x$. The po-path $\alpha_3 : \mathrm{R}x \xrightarrow{\text{po}} \alpha_3 : \mathrm{W}x$ will be considered when the cycle $\sigma_4 = \alpha_4 : \mathrm{W}x \xrightarrow{\text{cmp}} \alpha_3 : \mathrm{R}x \xrightarrow{\text{po}} \alpha_3 : \mathrm{W}x \xrightarrow{\text{cmp}} \alpha_4 : \mathrm{W}x$ is analysed, hence analysis of σ_2 and σ_3 is redundant. We call a cycle *critical* if it is simple, has no chords, and contains at least one unsafe po-path.

Concerning atomicity, in Fig. 4, graph (i) represents the common situation of an instruction α reading and writing from/to the same location, for instance to increment a counter. A write to the same location can lead to an atomicity violation: If this write is performed between the read and write of the instruction, the effect of the former write will be lost. Shasha and Snir explain that for atomicity checking, it is even needed to go against the direction of po-edges inside instructions. By doing so, we get one cycle in graph (i), $\alpha : \mathrm{R}x \xrightarrow{\text{cmp}} \beta : \mathrm{W}x \xrightarrow{\text{cmp}} \alpha : \mathrm{W}x \xleftarrow{\text{po}} \alpha : \mathrm{R}x$, and two cycles in graph (ii), namely $\alpha : \mathrm{W}x \xrightarrow{\text{cmp}} \beta : \mathrm{R}x \xrightarrow{\text{po}} \beta : \mathrm{R}y \xrightarrow{\text{cmp}} \alpha : \mathrm{W}y \xleftarrow{\text{po}} \alpha : \mathrm{W}x$ and $\beta : \mathrm{R}x \xrightarrow{\text{cmp}} \alpha : \mathrm{W}x \xrightarrow{\text{po}} \alpha : \mathrm{W}y \xrightarrow{\text{cmp}} \beta : \mathrm{R}y \xleftarrow{\text{po}} \beta : \mathrm{R}x$. The cycles in graph (ii) represent the reading instruction retrieving an inconsistent state in which only one location has been updated. The remedy is to enforce po-edges against their direction, resulting in cyclic dependencies. These cannot be resolved in practice using fences, but require locks [36]. Recent work [1,2,4,6–8,14,16,19,23,25,28,29,39] does not address this, allowing non-SC behaviour due to atomicity violations.

Next, we consider unsafety of po-paths w.r.t. a given weak memory model. The information is obtained from [4,6,7,18]. The novelty is that we apply path rewriting.

3 Guarantees of Weak Memory Models

Over the years, various weak memory multiprocessor architectures have been developed, for instance with x86 [33], SPARC [38], ARMv7 [7], ARMv8 [20] and POWER [22]

[2] We use the term 'delay' here to refer to the *remedy* for non-SC behaviour [36], and not, as for instance later done in [4,6,7], to refer to the *problem*, i.e., the unsafe behaviour itself.

Table 1. Intra-thread safety guarantees under various memory models ($x \neq y$)

Case	Subtrace	SC	TSO	PSO	ARMv7
1	Wxj < Rxk	✓	✗	✗	✗
2	Wxj < Ryk	✓	✗	✗	✗
3	Rxj < Rxk	✓	✓	✓	✗
4	Rxj < Ryk	✓	✓	✓	✓ $\Longleftrightarrow$ Ryk addr Rxj
5	Wxj < Wxk	✓	✓	✓	✓
6	Wxj < Wyk	✓	✓	✗	✗
7	Rxj < Wxk	✓	✓	✓	✓ $\Longleftrightarrow$ Wxk addr Rxj $\vee$ Wxk dp Rxj
8	Rxj < Wyk	✓	✓	✓	✓ $\Longleftrightarrow$ Wyk addr Rxj $\vee$ Wyk dp Rxj

processors. For each, suitable memory models have been derived, to reason about the access reorderings they may apply [3,6,18]. For the x86 and some SPARC processors, the TSO memory model is applicable, while other SPARC architectures support the PSO model. The POWER and ARMv7 processors can apply many types of reordering, requiring a very weak memory model.

To reason about the order of accesses, we use a relation $<$, which is initially equal to po^-, the transitive reduction of po, i.e., $a \ \mathsf{po}^- \ b$ cannot be decomposed into multiple po-connections between accesses from a to b. With $<$, we define *paths* of accesses $a < b < \ldots$. Paths should not be confused with traces: a trace is a concrete execution of accesses, while a path indicates the order in which accesses may be executed. For accesses a and b, we say that $a < b$ is *safe* for a memory model iff any trace executing a and b executes a before b. This is formally expressed as $a \ \mathsf{ppo} \ b$, with ppo a safe subrelation of po [4,6].

Table 1 gives an overview of the guarantees provided by the aforementioned memory models for accesses performed by the same thread.[3] Case 5 expresses that the order of two writes to the same location is guaranteed under all memory models. For cases 4, 7 and 8 of ARMv7/POWER, a reordering is allowed if the latter access is not dependent on the former. We define the following (intra-thread) dependency relations [6,7]:

1. The *address* relation addr relates address dependent accesses. We have $b \ \mathsf{addr} \ a$ iff a is a read access and is done to (possibly via local variables) compute an address for access b (for instance an address of an array element). For example, in order to access in an array `v` element `v[i]`, first, the value of `i` must be retrieved.
2. The relation dp combines two dependency relations. We have $b \ \mathsf{dp} \ a$ iff either
   - b needs to write a value dependent on a (for example, to perform `x = y`, the value of `y` must be retrieved before assigning it to `x`), or
   - a is a read needed to evaluate a condition which must hold for the program branch containing b to be executed (for example, in `if (x==0) {y = z}`, the write to `y` and the read from `z` are both dp-dependent on the read from `x`).

Note that case 4 for ARMv7 requires only addr-dependency. A read access b may be performed even before a condition that guards b has been evaluated, unless b is address dependent on the read access(es) of the condition. For example, in `if (x==0) {y=z}`, the read access from `z` can be performed before the read from `x`. This is called

[3] We ignore rdw and detour dependencies between threads under ARMv7/POWER [7], since those cannot be checked thread-locally. The penalty is that we under-approximate the guarantees of those memory models, but the effect seems marginal, as experimentally observed in [7].

speculative execution. Cases 7 and 8 for ARMv7 imply that speculative writing is never allowed, i.e., in the example, the writing to `y` cannot be done before the read from `x`.

Next, we define safety of po-*paths*, i.e., paths of accesses with $< = \mathsf{po}^-$, in terms of string rewriting. Table 1 can be used to define rewrite rules for TSO, PSO and ARMv7/POWER: if a model does not guarantee the order of accesses a and b (i.e., $a < b$), then the rewrite rule $(a < b) \Rightarrow (b < a)$ is applicable. We refer to the set of rewrite rules for a model as Σ, and say that a po-path is safe w.r.t. Σ iff it is safe under a model with rewrite rules Σ. The transitive closure of $<$ is $<^+$.

Definition 2 (po-path safety). *Given a* po*-path q between accesses a and b ($a <^+ b$ with $< = \mathsf{po}^-$), and a set of rewrite rules Σ, we say that q is* safe w.r.t. Σ *iff it cannot be rewritten, using the rewrite rules in Σ, to a path q' with $b <^+ a$.*

In other words, safety of a po-path between accesses a and b w.r.t. Σ can be determined by applying a path rewriting algorithm that tries to reorder a and b.

We already covered the case of two writes related via cmp. For accesses a and b with at most one being a write, if we have both a po b and a cmp b, then order guarantee of $a < b$ does not depend on cmp under any of the memory models. For instance, if a and b are not related via either addr or dp, a is a write and b is a read, then b can use the result of a before a is globally visible. Concerning cmp-related accesses of *different* threads, ARMv7/POWER has a property called *store atomicity relaxation* (ARMv8 [20, 35] no longer allows this). Because of this, a cmp-edge in the AEG between a write a and a read b must be considered as unsafe in the direction from a to b [6]. To remedy this, a so-called *A-cumulative* fence must be placed along the po-path following the cmp-edge in the cycle [6]. The other memory models do not have this problem.

4 Deriving po and cmp via State Space Exploration

Given a formal parallel program specification M, containing a specification of the individual threads, i.e., which atomic instructions each thread performs in which SC order, and the variables (thread-local and program-global) each instruction accesses, we can construct its state space using an explicit-state model checker. As mentioned in Section 2, M can be expressed using a formal modelling language such as Promela [21] or mCRL2 [15]. In the next example, we use a state machine to specify a thread.

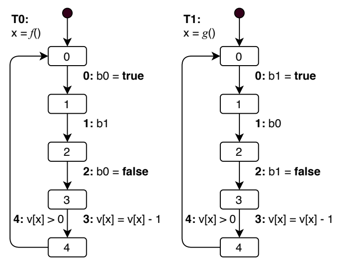

Fig. 5. State machines `T0` and `T1`

Example 1. Fig. 5 provides the specification of two threads `T0` and `T1`, each with a local variable `x`. The initial value of this variable is provided by function f and g, respectively. We assume that f and g are too complex to derive by means of static analysis whether or not they return the same value. From the initial states of `T0` and `T1`, i.e., 0, five steps can be repeatedly executed. First, `T0` sets the global boolean variable `b0` to **true** (initially it has the value **false**), after which it waits for global boolean variable `b1` to be **true** as well. Execution can continue as soon as this is the case; before that, execution is blocked (this has the same semantics as the condition statement in PROMELA [21], for instance). Next, `b0` is set to **false** again, and the value of element `x` in the global integer array `v` is decremented (all elements in `v` are initially set to some positive value). Finally, if `v[x]` is still bigger than `0`, the process is repeated. Thread `T1` works similarly to `T0`, except that `b0` and `b1` are swapped. Note that the use of `b0` and `b1` effectively works as a synchronisation mechanism; for each thread, execution can only proceed from state 1 to state 2 if the other thread has also proceeded to state 1. It is not guaranteed that when this has happened, both threads will proceed to state 2, as one thread may set one of the boolean variables back to **false** before the other thread has moved to state 2. However, this issue is not important for the example.

We define a state space as a tuple $\mathcal{G}_M = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \hat{s})$, with $\hat{s}$ the initial state, in which all thread specifications are in their initial state and all variables are set to their initial value, $\mathcal{S}$ the set of states reachable from $\hat{s}$ ($\hat{s} \in \mathcal{S}$), $\mathcal{A}$ a set of labels, and $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ the transition relation. With $s \xrightarrow{a} s'$, we express that from state s, a transition labelled a exists to a state s'. The set $out(s)$ is defined as $out(s) = \{a \mid \exists s' \in \mathcal{S}. s \xrightarrow{a} s'\}$.[4]

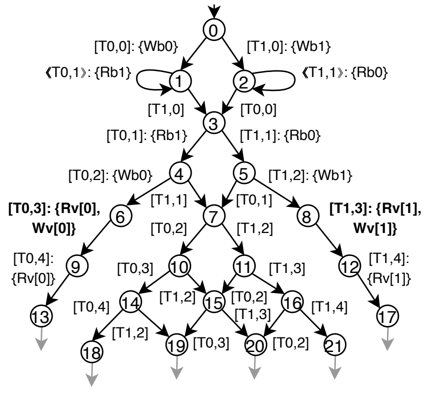

Fig. 6. A state space fragment

We construct the state space of a specification at the level of instructions, i.e., with each transition, exactly one instruction is associated. If M consists of n threads $t_1, \ldots, t_n$ in parallel composition, then every state s in $\mathcal{G}_M$ encodes, besides values for all variables in M, the local states $s_1, \ldots, s_n$ of the threads. For each enabled instruction α from s_i ($1 \leqslant i \leqslant n$) that locally leads to state s'_i in t_i, there is a transition $s \xrightarrow{t_i, \alpha, [\alpha]} s'$, with s' being the state in which t_i is in s'_i, all other threads t_k ($k \neq i$) are in s_k, and the variables have been updated as defined by α. Since we use $\mathcal{G}_M$ to analyse memory access behaviour, we need to know when read accesses are performed to evaluate a condition, also when it evaluates to **false**. For that reason, for each state s in $\mathcal{G}_M$ and each instruction α of t_i blocked in s, we have a self-loop $s \xrightarrow{t_i, \alpha, \langle\!\langle \alpha \rangle\!\rangle} s$.

Note that $\mathcal{G}_M$ allows us to precisely reason about accesses, even if pointers are used or dynamic accessing of array elements. For instance, in Fig. 5, the accessed location in `v[x]>0` is determined by the value of `x`. Because, of this, $[\alpha]$ depends on the current state s. In the example, in any system state in which `T0` is in state 2 and `x` has the value `0`, we have $[$`v[x]>0`$] = \{$`Rv[0]`$\}$. A model checker will make this explicit.

The state space will contain all possible interleavings of instructions in the specified order. All traces in $\mathcal{G}_M$ are therefore SC (Def. 1). Next, we wish to derive an AEG, as is

[4]Note that we define state spaces by means of *Labelled Transition Systems*, in which transitions are labelled with events. However, the technique we propose in this paper can be adapted to Kripke structures, by encoding via state predicates the events that are performed.

done, for instance, in [4, 11, 12, 26], but we derive it from the state space, as opposed to the specification (or the source code) of a program. Before we explain how to derive pr and cmp, consider the example state space fragment in Fig. 6, which is part of the state space of `T0` and `T1` (from Example 1) in parallel composition, when $f() = \mathtt{0}$ and $g() = \mathtt{1}$. The transition labels provide the sets of accesses, with instructions named by pairs 'thread, instruction identifier'. To make the figure more clear, only the sets of accesses are indicated, for instance $[\mathtt{T0},\mathtt{0}]$, and their definition is given only once, for instance $\{\mathrm{Wb}\mathtt{0}\}$ for $[\mathtt{T0},\mathtt{0}]$. For this example, this suffices, as `x` is never updated, and hence there is no instruction α for which $[\alpha]$ changes over time. The two outgoing transitions from initial state 0 are associated with instruction `0` of `T0` and `T1`. If we traverse the transition for `T0`,`0`, we enter state 1. State 1 has a self-loop for $\langle\!\langle \mathtt{T0}, 1\rangle\!\rangle$, representing the read access of `b1` to discover that `b1` evaluates to **false**. Beyond state 2, selfloops are not displayed, to simplify the figure.

For each thread t, instruction α of t and access $a \in [\alpha]$, we construct exactly one vertex $\langle t, \alpha, a\rangle$ in the AEG. The edges are derived from $\mathcal{G}_M$. As each access (of some instruction and thread) has one AEG vertex, the program order relation may appear cyclic if the thread specification contains cycles (as in Fig. 5). We model this with a *program relation* pr. It offers the benefit of being able to represent infinite behaviour, as long as the number of instructions is finite. Unsafety of pr-paths is similar to unsafety of po-paths. To define pr, we first consider pairs of accesses from the same instruction α of thread t. We have to define how these are ordered. We want to be as non-restrictive as possible, so we restrict pr only by addr and dp (see Section 3): for every instruction α and accesses $a, b \in [\alpha]$, we have $a \;\mathsf{pr}\; b$ iff $b \;\mathsf{addr}\; a$ or $b \;\mathsf{dp}\; a$, i.e., a only must precede b if b depends on a. In the AEG, if $a \;\mathsf{pr}\; b$, we add a pr-edge from $\langle t, \alpha, a\rangle$ to $\langle t, \alpha, b\rangle$.

Next, using this definition of pr, we define $\bot(A)$, with $A \subseteq [\alpha]$ of an instruction α, as $\{a \in A \mid \neg\exists b \in A. b \;\mathsf{pr}\; a\}$, i.e., the accesses in A that are first in the pr-order. With $\top(A)$, we refer to the accesses last in the pr-order: $\top(A) = \{a \in A \mid \neg\exists b \in A. a \;\mathsf{pr}\; b\}$.

Finally, we extend pr by relating accesses from different instructions executed by the same thread. If we can establish that the execution of an instruction α enables the execution of at least some accesses of instruction β, then we pr-relate the accesses of α executed last with the accesses of β executed first. In the example of Fig. 6, since the execution of access $[\mathtt{T0}, \mathtt{0}] = \{\mathrm{Wb}\mathtt{0}\}$ from state 0 leads to a state in which $\langle\!\langle \mathtt{T0}, 1\rangle\!\rangle$ is executable, i.e., $\mathrm{Rb}\mathtt{1}$, both accesses are performed by `T0`, and $\langle\!\langle \mathtt{T0}, 1\rangle\!\rangle$ was not enabled in state 0, the execution of $[\mathtt{T0}, \mathtt{0}]$ enables $\langle\!\langle \mathtt{T0}, 1\rangle\!\rangle$. Since $\mathrm{Wb}\mathtt{0} \in \top([\mathtt{T0}, \mathtt{0}])$ and $\mathrm{Rb}\mathtt{1} \in \bot(\langle\!\langle \mathtt{T0}, 1\rangle\!\rangle)$, we conclude that $\mathrm{Wb}\mathtt{0} \;\mathsf{pr}\; \mathrm{Rb}\mathtt{1}$ and $\langle \mathtt{T0}, \mathtt{0}, \mathrm{Wb}\mathtt{0}\rangle \xrightarrow{\mathsf{pr}} \langle \mathtt{T0}, 1, \mathrm{Rb}\mathtt{1}\rangle$. Similarly, when traversing the transition for `T0`,1 from state 3, we can conclude $\langle \mathtt{T0}, 1, \mathrm{Rb}\mathtt{1}\rangle \xrightarrow{\mathsf{pr}} \langle \mathtt{T0}, 2, \mathrm{Wb}\mathtt{0}\rangle$. In general, for any two instructions α, β of a thread t, we have:

$$\langle t, \alpha, a\rangle \xrightarrow{\mathsf{pr}} \langle t, \beta, b\rangle \iff \exists s, s' \in \mathcal{S}. \exists A \in \{[\alpha], \langle\!\langle\alpha\rangle\!\rangle\}, B \in \{[\beta], \langle\!\langle\beta\rangle\!\rangle\}. s \xrightarrow{t,\alpha,A} s' \\ \wedge (t, \beta, B) \in out(s') \backslash out(s) \wedge a \in \top(A) \wedge b \in \bot(B)$$

Note that the pr relation is non-transitive, corresponding closely with po^-.

Next, we need to derive cmp. A straightforward way is to relate every pair of accesses (a, b) performed by different threads that access the same location, if at least one access is a write, and that is essentially the approach taken by static analysis techniques. With state space exploration, we can define cmp more precisely. Note that the above condition is the same as the one for data races, except that there is a data race only if accesses execute *at the same time*. In terms of traces, this means that it must be possible to execute b immediately after a, and vice versa, i.e., we can have both $a \prec b$ and $b \prec a$. We observe that for AEG construction, this aspect is also relevant. Consider the situation that accesses a, b *cannot* be executed at the same time, i.e., no traces are possible in which either $a \prec b$ or $b \prec a$. In that case, if there would be a cmp-edge between a and b, it might be possible to construct cycles in which $a \xrightarrow{\mathsf{cmp}} b$ or $b \xrightarrow{\mathsf{cmp}} a$ is an edge, but that would represent an execution in which a is directly followed by b or b followed by a, respectively (recall the example in Fig. 3 of Section 2), behaviour that is not possible.

The following theorem addresses how to determine, by analysing the state space, whether two accesses of different threads can directly follow each other in an SC trace.

Theorem 1. *Given a state space $\mathcal{G}_M = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \hat{s})$, two accesses a, b and instructions α, β with $a \in [\alpha]$, $b \in [\beta]$, to be executed by threads t_1, t_2 of M, respectively ($t_1 \neq t_2$). There exists a trace π with $a \prec b$ or $b \prec a$ iff there exists a state $s \in \mathcal{S}$ with $(t_1, \alpha, A) \in out(s)$ and $(t_2, \beta, B) \in out(s)$, where $A \in \{[\alpha], \langle\!\langle\alpha\rangle\!\rangle\}$, $B \in \{[\beta], \langle\!\langle\beta\rangle\!\rangle\}$ and $a \in A$, $b \in B$.*[5]

It follows that cmp-edges can be added to the AEG by checking for each pair of outgoing transitions of every reachable state whether they are executed by different threads and have conflicting accesses. In Fig 6, state 1 shows that $\langle\!\langle \texttt{T0}, 1\rangle\!\rangle$ and $[\texttt{T1}, \texttt{0}]$ conflict. Therefore, we add $\langle \texttt{T0}, 1, \texttt{Rb1}\rangle \overset{\mathsf{cmp}}{\longleftrightarrow} \langle \texttt{T1}, \texttt{0}, \texttt{Wb1}\rangle$ to the AEG. In contrast, state 15 demonstrates that $[\texttt{T0}, 3] = \{\texttt{Rv[0]}, \texttt{Wv[0]}\}$ and $[\texttt{T1}, 3] = \{\texttt{Rv[1]}, \texttt{Wv[1]}\}$ can happen simultaneously, but they have no conflicting accesses.

The above procedure works for SC, but not yet for weaker memory models. For instance, if we have three instructions α, β, γ, two threads t, t', and states $s_0, \ldots, s_3 \in \mathcal{S}$, with $s_0 \xrightarrow{t,\alpha,[\alpha]} s_1 \xrightarrow{t,\beta,[\beta]} s_2$, $s_0 \xrightarrow{t',\gamma,[\gamma]} s_3$ and $s_1 \xrightarrow{t',\gamma,\langle\!\langle\gamma\rangle\!\rangle} s_1$, then there is no SC trace in which for any accesses $b \in [\beta]$, $c \in [\gamma] \backslash \langle\!\langle\gamma\rangle\!\rangle$ we have either $b \prec c$ or $c \prec b$, but if b can be reordered before some accesses of α, b and accesses of γ may suddenly conflict.

To both detect all cmp-edges for weak memory models *and* identify which pr-paths are unsafe, we first repeatedly apply access reordering on all accesses in the program, as explained in Section 3, with $<$ intially set to pr, and keep track for each instruction α which accesses of other instructions can be reordered to be executed at the same time α is executed. This leads to a set of accesses $[\alpha]^+ \supseteq [\alpha]$. The procedure is continued until a fix-point has been reached, i.e., the $[\alpha]^+$ have been identified. While the reordering is performed, we construct a relation ppr with $a \;\mathsf{ppr}^+\; b$ iff there exists no unsafe pr-path from a to b: if at any point we have $a < b$ and b cannot be reordered before a, we know that a ppr b. When a fix-point has been reached, we know there exists an unsafe pr-path from access a to access b if $\neg(a \;\mathsf{ppr}^+\; b)$. The cmp-relation can be constructed by comparing the accesses in $[\alpha]^+$ and $[\beta]^+$ of each two instructions α, β that are associated with different threads and are related to transitions with a common source state.

Example 2. Consider again Example 1 with threads T0, T1. Fig. 7 presents AEGs, for convenience at the instruction level, that can be constructed when using model checking (on the left) and when only using static analysis (on the right). For the moment, ignore the fences and the grey colouring. The vertices contain the instructions, solid edges represent

[5] A proof can be found at `http://www.win.tue.nl/~awijs/seqcon-analyser`.

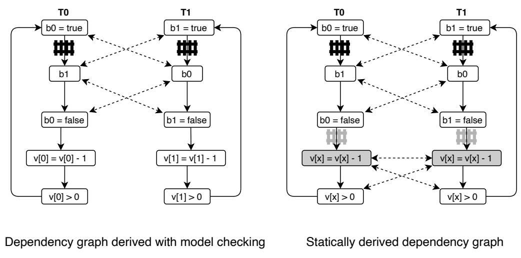

Fig. 7. AEGs (at instruction level) for `T0` and `T1` (Fig. 1), with $f()$=`0`, $g()$=`1`, obtained with (left) and without (right) model checking, including locks and fences for TSO.

pr-edges between accesses performed by those instructions, and dashed edges represent cmp-edges between some of those accesses. In the AEG on the left, the accesses to `v[0]` and `v[1]` have been distinguished, and it has been observed that `T0` and `T1` never conflict on accessing `v`. Hence, those accesses are not related by cmp. In the AEG on the right, the accesses to `v` of each thread have been symbolically represented by `v[x]`, and it is concluded that conflicts can happen, hence the extra cmp-edges.

5 Detecting and Ruling Out Non-SC Behaviour

Shasha and Snir provided a definition of critical cycle that allows efficient detection in an AEG [36]. We generalise this definition to use the non-transitive, cyclic pr, distinguish safe and unsafe pr-paths, and support unsafe cmp-edges.

Definition 3 (Critical cycle). *A cycle σ in a* pr $\cup$ cmp *AEG is* critical *iff:*

c1. σ contains for each thread t at most one pr*-path;*

c2. σ contains at consecutive start and end points of pr*-paths and/or* cmp*-edges up to three accesses to the same location;*

c3. For at least one thread t, its pr*-path in σ is unsafe, or there is an unsafe* cmp*-edge;*

c4. At least two threads are involved in σ;

c5. For each pr*-path $\alpha_0 : a_0 \xrightarrow{\mathsf{pr}} \cdots \xrightarrow{\mathsf{pr}} \alpha_n : a_n$ in σ, we have for all $0 \leqslant i < j \leqslant n$ that either $i = 0$ and $j = n$, or $\alpha_i : a_i \neq \alpha_j : a_j$.*

By c1, c3 and c4, a pr $\cup$ cmp cycle consists of at least one pr-path, connected at its start and end accesses via cmp-edges with at least one access of at least one other thread. Furthermore, for each pr-path, no proper subpath is a cycle (c5), but it may itself be a cycle. If some proper subpath is a cycle, it means that a shorter path can be constructed by removing that subpath. By c1, no chords exist in pr: if a thread t is involved in a cycle both with a pr-path $a \xrightarrow{\mathsf{pr}} \cdots \xrightarrow{\mathsf{pr}} b$ and a pr-path $c \xrightarrow{\mathsf{pr}} \cdots \xrightarrow{\mathsf{pr}} d$, then there are multiple chords, for instance $a \xrightarrow{\mathsf{pr}} \cdots \xrightarrow{\mathsf{pr}} d$. Finally, c2 ensures that no chords in cmp exist. If a location is involved more than three times in a cycle, such a chord must exist [36]. In Fig. 3, σ_2 and σ_3 violate c2, since x is involved four times, which points to the existence of the chord $\alpha_4 : \mathrm{W}x \xrightarrow{\mathsf{cmp}} \alpha_1 : \mathrm{R}x$.

When no atomicity checking is performed, there is actually an additional condition, namely that the cycle involves at least two locations [4]. The weak memory models we consider ensure SC *per location*, i.e., cycles in which all accesses access the same location do not represent non-SC behaviour. However, for atomicity checking, such cycles cannot be ignored, as explained in Section 2 (see Fig. 4).

Theorem 2. *A trace π of a program specification M is non-SC iff it traverses a* $\mathsf{pr} \cup \mathsf{cmp}$ *path through the AEG of M that contains a critical cycle.*[6]

Note that enforcing an unsafe pr-path p' results in enforcing any unsafe pr-path p containing p'. Any cycle σ satisfying c1 to c4 of Def. 3 contains a cycle σ' satisfying c1 to c5. If p is part of σ and p' is part of σ', then p is enforced due to p' being enforced. Therefore, it suffices to enforce unsafe pr-paths in critical cycles as defined by Def. 3.

For the detection of critical cycles, algorithms for finding elementary circuits can be used. We use Tarjan's algorithm [40], extended to detect cycles meeting c1-c5 of Def. 3. To support atomicity checking, all that is needed is to allow cycle detection to move against the pr-order between at-related accesses, as is explained in Section 2 and [36]. The ppr-relation allows us to identify the existence of unsafe pr-paths in constant time.

With each instruction in the program, we associate a counter. Each time a critical cycle is detected, we record the involved unsafe pr-paths and cmp-edges as sequences of the involved instructions, and increment their counters. If we do atomicity checking, we furthermore mark an instruction for locking if its execution in the unsafe path violates pr. Once all cycles have been detected, the sequences of instructions that include an instruction marked for locking are removed, since a lock will make an unsafe path safe. The counters of all involved instructions are decremented each time a sequence is removed. For the remaining sequences, we sort the instructions by the counter values, select the instruction α with the highest value, and place a delay after $\perp([\alpha])$, to ensure that any path involved α is enforced. We remove each sequence involving α, decrement the counters of the other instructions, and repeat until all counters have the value 0.

Although the constructed AEGs are more precise than statically derived ones, we do not claim that our fencing procedure is optimal. In fact, experiments demonstrate that it is not (see Section 6). Future work involves optimising this procedure as far as possible.

Example 3. Consider the fences and the grey colouring in Fig. 7, which result from analysing the system under TSO. In both AEGs, the critical cycle $\sigma_1 = \mathtt{T0} : \mathrm{Wb0} \xrightarrow{\mathsf{pr}} \mathtt{T0} : \mathrm{Rb1} \xrightarrow{\mathsf{cmp}} \mathtt{T1} : \mathrm{Wb1} \xrightarrow{\mathsf{pr}} \mathtt{T1} : \mathrm{Rb0} \xrightarrow{\mathsf{cmp}} \mathtt{T0} : \mathrm{Wb0}$ requires that the two black fences are placed. On the right, additional critical cycles are detected, due to the inaccuracy of the AEG. The grey nodes represent the instructions for which it is detected that locks are needed, when atomicity checking is performed. In other words, `v[x]=v[x]-1` can only be executed if a lock on `v[x]` has been acquired. The involved cycles are actually not directly visible, due to the AEG being given at the instruction level, but the two instructions `v[x]=v[x]-1` conflict with each other, forming two cycles $\sigma_2 = \mathtt{T0} : \mathrm{Wv}\mathtt{[x]} \xleftarrow{\mathsf{pr}} \mathtt{T0} : \mathrm{Rv}\mathtt{[x]} \xrightarrow{\mathsf{cmp}} \mathtt{T1} : \mathrm{Wv}\mathtt{[x]} \xrightarrow{\mathsf{cmp}} \mathtt{T0} : \mathrm{Wv}\mathtt{[x]}$ and $\sigma_3 = \mathtt{T1} : \mathrm{Wv}\mathtt{[x]} \xleftarrow{\mathsf{pr}} \mathtt{T1} : \mathrm{Rv}\mathtt{[x]} \xrightarrow{\mathsf{cmp}} \mathtt{T0} : \mathrm{Wv}\mathtt{[x]} \xrightarrow{\mathsf{cmp}} \mathtt{T1} : \mathrm{Wv}\mathtt{[x]}$, both going against the pr-direction. As locks strictly provide more guarantees than fences [5], placing locks means that no more unresolved violations exist. Alternatively, if no atomicity checking is performed, the cycles $\sigma_4 = \mathtt{T0} : \mathrm{Wb0} \xrightarrow{\mathsf{pr}} \mathtt{T0} : \mathrm{Rv}\mathtt{[x]} \xrightarrow{\mathsf{cmp}} \mathtt{T1} :$

[6]A proof sketch can be found at `http://www.win.tue.nl/~awijs/seqcon-analyser`.

$\mathtt{Wv[x]} \xrightarrow{\text{pr}} \cdots \xrightarrow{\text{pr}} \mathtt{T1} : \mathtt{Rb0} \xrightarrow{\text{cmp}} \mathtt{T0} : \mathtt{Wb0}$ and $\sigma_5 = \mathtt{T1} : \mathtt{Wb1} \xrightarrow{\text{pr}} \mathtt{T1} : \mathtt{Rv[x]} \xrightarrow{\text{cmp}} \mathtt{T0} : \mathtt{Wv[x]} \xrightarrow{\text{pr}} \cdots \xrightarrow{\text{pr}} \mathtt{T0} : \mathtt{Rb1} \xrightarrow{\text{cmp}} \mathtt{T1} : \mathtt{Wb1}$ require that the grey fences are placed. If the instructions `v[x]=v[x]-1` are locked, this is not required, as Wb0 in [`b0=false`] and Wb1 in [`b1=false`] are separated from the Rv`[x]`'s by those locks.

6 Experimental Results

We have implemented our technique using the mCRL2 toolset [15] for state space exploration. State machine models are first translated to mCRL2 specifications, from which a state space can be generated and written to disk. In addition, we have developed a new tool seqcon-analyser in C++, that reads the state space, constructs an AEG, and suggests where to place fences and locks, based on critical cycle detection.

We conducted experiments to compare seqcon-analyser with existing fence insertion tools, and to analyse their scalability. We decided to compare seqcon-analyser with the static analysis tool musketeer [4] and the model checking tool remmex [29]. These tools offer a good representation of the current state-of-the-art in static-analysis- and model checking-based approaches to automatic fence insertion, respectively. For instance, in [4], musketeer clearly outperforms the static analysis tools dfence [30], memorax [1], offence [6], trencher [14] and pensieve [39], using several instances of four of the models, Dekker, Peterson, Lamport, and Szymanski, that we also used in our experiments. Hence, we have not involved the other static analysis tools in our experiments.

Besides the four models already mentioned, we selected six additional models from the BEEM benchmark set [34], Anderson, Bakery, Elevator2, Leader_filters, Mcs, and Msmie, and manually translated several instances of each of those models, written in the DVE language [10], to suitable input for seqcon-analyser, musketeer and remmex.[7]

Table 2 presents the experimental results, comparing our technique without atomicity checking (s–a) with musketeer (m) and remmex (r), tools that do not support atomicity checking. In addition, we report the results obtained when performing atomicity checking with our tool (s+a). We report the number of delay insertions under TSO, PSO and ARMv7 whenever possible (remmex does not support ARMv7) in the form ('number of locks' / 'number of non-cumulative fences' / 'number of A-cumulative fences'). We acknowledge that ARMv8 is more recent, but the weaker ARMv7 is very suitable to demonstrate the efficiency of our technique when applied on very weak memory models.

We conducted our experiments on the DAS-5 cluster [9]. Each node runs CentOs 7.4, and has a 2.4 GHz Intel Haswell E5-2630-v3 CPU and 64 GB of memory. In the table, we list for each model the number of threads ($|\mathbb{T}|$), the number of instructions ($|I|$), and the number of memory locations ($|\mathbb{L}|$). For state space generation, we used version 201908.0 of the mCRL2 toolset, and report the number of states and the runtime. For our technique, on the one hand, the runtime of state space generation should be added to the runtime needed for critical cycle detection to obtain the overall time, but on the other hand, state space generation is only needed once per case, to perform cycle detection for all memory models, with and without atomicity checking. For the cycle detection phase, we have excluded the time needed to read the state space, as it obfuscates the time for

[7]See `http://www.win.tue.nl/~awijs/seqcon-analyser` for the models and our tool.

Table 2. Scalability results for (M)USKETEER, (R)EMMEX, and SEQCON-ANALYSER without (S–A) and with (S+A) atomicity checking. Runtimes in seconds, averages of ten executions. o.o.m.: out of memory (> 64 GB), o.o.t.: out of time (> 20 hrs.), *: executed with option –no-loop-duplication, -: no result possible due to lack of suitable error states.

				state space gen.		**TSO**				**PSO**				**ARMv7**		
case	$\lvert\mathbb{T}\rvert$	$\lvert I\rvert$	$\lvert\mathbb{L}\rvert$	**# states**	**time**	M	R	S–A	S+A	M	R	S–A	S+A	M	S–A	S+A
Anderson.1	2	12	3	350,119	41.89	0.03 (/6/)*	o.o.m.	**0.04 (/4/)**	0.05 (4/2/)	0.04 (/6/)*	o.o.m.	**0.05 (/4/)**	0.05 (4/2/)	0.06 (/2/6)*	**0.07 (//7)**	0.07 (4//5)
Anderson.4	4	24	5	9,956	2.56	o.o.m.	o.o.m.	**17.23 (/8/)**	17.36 (8/4/)	o.o.m.	o.o.m.	**39.61 (/12/)**	40.07 (8/8/)	o.o.m.	**54.68 (//12)**	55.68 (8//8)
Anderson.6	6	36	7	2,633,730	1,086.73	o.o.m.	o.o.m.	**855.68 (/12/)**	855.54 (12//)	o.o.m.	o.o.m.	**1,936.00 (/12/)**	1,910.83 (12//)	o.o.m.	**4,337.74 (//18)**	4,358.88 (12/0/12)
Anderson.8	7	42	8	50,799,077	25,322.92	o.o.m.	o.o.m.	**4,425.45 (/14/)**	4,340.08 (14//)	o.o.m.	o.o.m.	**9,524.06 (/14/)**	9,625.17 (14/7/)	o.o.m.	**24,017.29 (//21)**	23,694.51 (14//14)
Bakery.1	2	18	4	1,622	0.45	0.17 (/10/)	o.o.m.	**0.07 (/4/)**	0.07 (2/1/)	0.23 (/12/)	o.o.m.	**0.12 (/5/)**	0.09 (2/3/)	**0.34 (/2/9)**	0.32 (//12)	0.32 (2//10)
Bakery.3	3	27	6	41,185	14.52	o.o.m.	o.o.m.	**1.21 (/5/)**	1.37 (3/2/)	o.o.m.	o.o.m.	**1.55 (/7/)**	1.55 (3/4/)	o.o.m.	**3.98 (//18)**	4.01 (3//15)
Bakery.5	4	36	8	9,604,719	3,933.80	o.o.m.	o.o.m.	**10.65 (/7/)**	10.67 (4/3/)	o.o.m.	o.o.m.	**12.66 (/9/)**	12.70 (4/9/)	o.o.m.	**386.54 (/3/72)**	391.62 (4/0/20)
Dekker.1	2	16	3	20	0.02	0.01 (/4/)	1.20 (/2/)	**0.01 (/2/)**	0.01 (/2/)	0.01 (/8/)	**0.77 (/2/)**	0.02 (/3/)	0.01 (/3/)	0.02 (/8/8)	**0.02 (/3/2)**	0.02 (/3/2)
Dekker.2	3	24	4	62	0.03	0.04 (/6/)	o.o.m.	**0.04 (/3/)**	0.04 (/3/)	0.20 (/12/)	o.o.m.	**0.22 (/7/)**	0.22 (/7/)	0.64 (/11/10)	**0.32 (/5/7)**	0.28 (/5/7)
Dekker.3	4	32	5	212	0.06	2.55 (/8/)	o.o.m.	**0.23 (/4/)**	0.23 (/4/)	9.94 (/16/)	o.o.m.	**1.56 (/11/)**	1.52 (/11/)	59.26 (/18/13)	**2.44 (/7/9)**	2.46 (/7/9)
Dekker.5	6	48	7	3,104	0.83	0.04 (/11/)	o.o.m.	**2.62 (/6/)**	2.62 (/6/)	0.07 (/24/)	o.o.m.	**31.81 (/17/)**	31.79 (/17/)	0.08 (/24/46)	**54.57 (/11/13)**	54.59 (/11/13)
Elevator2.1	3	13	7	1,664	0.38	0.59 (/3/)	-	**0.04 (//)**	0.03 (//)	0.62 (/3/)	-	**0.06 (/2/)**	0.06 (5/1/)	0.81 (/8/5)	**0.17 (//6)**	0.18 (6//4)
Elevator2.2	3	26	13	174,080	38.85	o.o.m.	-	**0.15 (//)**	0.17 (//)	o.o.m.	-	**3.35 (/14/)**	3.58 (21/2/)	o.o.m.	**6.81 (/2/14)**	7.11 (21//5)
Elevator2.3	3	22	16	7,561,216	2,369.50	o.o.m.	-	**0.24 (//)**	0.27 (//)	o.o.m.	-	**0.56 (/2/)**	0.61 (14/1/)	o.o.m.	**4.31 (//6)**	4.53 (15//4)
Lamport.1	2	22	4	106	0.04	0.03 (/4/)	1.92 (/4/)	**0.01 (/2/)**	0.02 (/2/)	0.01 (/9/)	4.68 (/4/)	**0.06 (/4/)**	0.06 (/4/)	0.01 (/6/7)	**0.08 (//6)**	0.10 (//6)
Lamport.2	3	33	5	1,283	0.24	**0.01 (/6/)**	o.o.m.	0.23 (/6/)	0.18 (/6/)	0.01 (/13/)	o.o.m.	**0.80 (/6/)**	0.80 (/6/)	0.01 (/11/11)	**1.08 (//9)**	1.08 (//9)
Lamport.3	4	44	6	12,924	2.57	0.01 (/8/)	o.o.m.	**1.11 (/4/)**	1.11 (/4/)	0.02 (/17/)	o.o.m.	**5.64 (/8/)**	5.51 (/8/)	0.02 (/16/15)	**8.00 (//12)**	8.05 (//12)
Lamport.5	6	66	8	6,352,764	260.78	0.15 (/12/)	o.o.m.	**12.20 (/6/)**	12.48 (/6/)	0.92 (/25/)	o.o.m.	**64.57 (/12/)**	64.61 (/12/)	1.30 (/23/28)	**144.84 (//18)**	147.70 (/1/23)
Leader_filters.2	3	27	15	432	0.09	5.89 (/3/)	o.o.m.	**0.07 (/3/)**	0.07 (/3/)	**8.26 (/3/)**	o.o.m.	0.10 (/6/)	0.10 (/6/)	10.11 (/1/6)	**0.12 (/6/)**	0.11 (/6/)
Leader_filters.4	4	40	15	7,289	1.19	o.o.m.	o.o.m.	**0.88 (/4/)**	0.88 (/4/)	o.o.m.	o.o.m.	**1.03 (/4/)**	1.03 (/4/)	o.o.m.	**1.41 (//12)**	1.43 (//12)
Leader_filters.5	5	50	18	64,646	11.00	o.o.m.	o.o.m.	**3.70 (/10/)**	3.70 (/10/)	o.o.m.	o.o.m.	**4.29 (/10/)**	4.30 (/10/)	o.o.m.	**5.88 (//15)**	5.90 (//15)
Leader_filters.7	6	60	18	3,383,790	111.07	o.o.m.	o.o.m.	**11.49 (/6/)**	11.94 (/6/)	o.o.m.	o.o.m.	**13.17 (/6/)**	13.15 (/6/)	o.o.m.	**23.31 (//18)**	23.28 (//18)
Mcs.1	3	33	7	767	0.20	o.o.m.	o.o.m.	**3.27 (/3/)**	3.26 (6//)	o.o.m.	o.o.m.	**21.28 (/17/)**	21.49 (12/6/)	o.o.m.	**30.03 (/1/20)**	30.74 (12/1/9)
Mcs.3	4	44	9	11,599	2.84	o.o.m.	o.o.m.	**19.14 (/7/)**	19.73 (9/3/)	o.o.m.	o.o.m.	**78.54 (/19/)**	78.84 (13/8/)	o.o.m.	**117.30 (//28)**	118.71 (13//16)
Mcs.5	5	55	11	236,227	71.19	o.o.m.	o.o.m.	**108.22 (/8/)**	110.34 (11/4/)	o.o.m.	o.o.m.	**649.97 (/25/)**	697.02 (17/13/)	o.o.m.	**1,076.28 (//39)**	1,090.33 (17//27)
Msmie.1	5	79	6	2,334	1.18	o.o.m.	o.o.m.	**49.74 (/6/)**	54.19 (8/3/)	o.o.m.	o.o.m.	**4,559.18 (/27/)**	4,911.42 (38/5/)	o.o.m.	**7,180.24 (//61)**	7,518.20 (38//23)
Msmie.2	13	168	5	10,558	8.93	o.o.m.	o.o.m.	2,318.31 (/7/)	2,568.96 (19/16/)	o.o.m.	o.o.m.	o.o.t.	o.o.t.	o.o.m.	o.o.t.	o.o.t.
Peterson.1	2	8	3	14	0.02	0.01 (/3/)	0.81 (/2/)	**0.02 (/2/)**	0.02 (/2/)	0.01 (/6/)	1.07 (/4/)	**0.03 (/2/)**	0.03 (/2/)	0.01 (/1/5)	**0.04 (//2)**	0.04 (//2)
Peterson.2	3	12	4	62	0.02	0.01 (/6/)	10.50 (/3/)	**0.03 (/3/)**	0.02 (/3/)	0.01 (/12/)	68.12 (/6/)	**0.03 (/4/)**	0.03 (/4/)	0.01 (/1/10)	**0.04 (//3)**	0.04 (//3)
Peterson.3	4	16	5	1,264	0.06	0.01 (/13/)	3,754.04 (/4/)	**0.15 (/4/)**	0.13 (/4/)	0.08 (/20/)	o.o.m.	**0.17 (/4/)**	0.17 (/4/)	0.05 (/1/18)	**0.21 (//4)**	0.17 (//4)
Peterson.5	6	24	7	6,232	1.19	0.34 (/27/)	o.o.m.	**1.56 (/6/)**	1.56 (/6/)	11.53 (/42/)	o.o.m.	**1.87 (/6/)**	1.86 (/6/)	8.18 (/1/34)	**2.67 (//6)**	2.68 (//6)
Szymanski.1	2	20	2	52	0.02	**0.01 (/2/)**	2.33 (/2/)	0.03 (/4/)	0.03 (/4/)	0.01 (/9/)	o.o.m.	**0.44 (/4/)**	0.44 (/4/)	**0.02 (/2/2)**	0.80 (/2/8)	0.80 (/2/8)
Szymanski.2	3	30	3	354	0.10	**0.01 (/3/)**	o.o.m.	4.02 (/9/)	3.87 (/9/)	0.03 (/13/)	o.o.m.	**5.56 (/9/)**	**0.10 (/6/3)**	10.17 (/3/14)	10.27 (/3/14)	10.27 (/3/14)
Szymanski.3	4	40	4	2,376	0.77	**0.15 (/4/)**	o.o.m.	74.71 (/6/)	74.41 (/6/)	1.24 (/17/)	o.o.m.	**104.82 (/6/)**	104.88 (/6/)	**9.70 (/14/5)**	186.79 (//23)	189.70 (//23)
Szymanski.5	6	60	6	49,328	21.11	663.51 (/19/)	o.o.m.	1,070.38 (/13/)	1,072.20 (/13/)	o.o.m.	o.o.m.	**1,373.15 (/13/)**	1,377.78 (/13/)	o.o.m.	2,723.08 (/1/29)	2,730.82 (/1/29)

the actual computation. In the future, we plan to avoid storing and reading state spaces entirely. Translating a state machine model to mCRL2 can be done instantly.

In Table 2, the best results in each category, in terms of number of fences, and runtime in case of a tie, are written in bold. It should be mentioned that for the Anderson instances, MUSKETEER had to be run with the –no-loop-duplication option enabled, as the standard option reported (erroneously) that no fences were required.

Even though the models are relatively small, the positive effect of state space exploration is apparent. Constructing the state space helps to keep the AEG smaller, and hence to reduce the number of critical cycles. State spaces grow exponentially, but so does the number of critical cycles in the AEG. Frequently, state space exploration plus critical cycle detection with SEQCON-ANALYSER has an overall runtime that is not drastically worse than MUSKETEER, and in case of Dekker.3, the combined time is even shorter. It is important to note here that various techniques exist to speed up state space exploration considerably (for instance, by using symbolic [13, 17] or GPU exploration [42, 43]), whereas much fewer techniques exist to speed up the enumeration of elementary circuits in a graph. In other words, it is in practice beneficial to involve state space exploration, as the techniques above can be applied to further reduce the runtime.

The performance of REMMEX is as expected; since it has to explore all behaviour, SC and non-SC, it quickly runs out of memory. For the Elevator2 cases, no results could be obtained. REMMEX needs to be given an error state representing a violation of a safety property, after which it checks for the reachability of that state under the given memory model. However, for those cases, no suitable safety properties could be identified. With MUSKETEER, we also experienced out-of-memory frequently, which was not expected. The tool first constructs the entire set of critical cycles before deriving fences. This is not strictly needed, as alternatively, the output of Tarjan's algorithm [40], used by MUSKETEER, could be directly processed to store where fences are needed. If the implementation of MUSKETEER would be changed in this regard, the runtimes for the 'o.o.m.' cases of MUSKETEER would still be much higher than those of SEQCON-ANALYSER, as it always took several hours to fill the memory, and the number of fences would often be higher.

Finally, regarding the number of locks and fences, SEQCON-ANALYSER does not always identify the smallest number of fences, even though it works with AEGs that are in general more precise than those of MUSKETEER. This is due to the sub-optimal placement of fences to resolve all detected non-SC issues. In future work, we will continue on improving this aspect. The optimisation problem of resolving all non-SC behaviour with fences also frequently results in each tool suggesting different fence locations. For instance, the four fences suggested by SEQCON-ANALYSER for Anderson.1 are not suggested by MUSKETEER. Furthermore, it is interesting to note that in multiple cases, the number of locks is influenced by the memory model. Unfortunately, MUSKETEER does not support atomicity checking, so we cannot directly assess this, but with a pure static analysis technique, this effect would not be observed; if each pair of conflicting accesses is related by cmp, then for each memory model, the same atomicity issues would be detected.

7 Conclusions

We have proposed a technique for automatic delay insertion, combining state space generation and static analysis. It has the precision of model checking-based techniques, yet better scalability, and frequently even outperforms MUSKETEER, a state-of-the-art static analysis technique. Furthermore, it supports atomicity checking. We addressed TSO, PSO, and ARMv7, but an arbitrary set of intra-thread order guarantees can be specified. These may depend on relations such as addr and dp, but also on others. In the future, it will be interesting to support the CAT language [7], to make SEQCON-ANALYSER more configurable in this respect. Furthermore, we will investigate to what extent delay suggestions can be updated when a model is transformed, along the lines of [41].

References

1. Abdulla, P.A., Atig, M.F., Chen, Y.F., Leonardsson, C., Rezine, A.: Memorax, a precise and sound tool for automatic fence insertion under TSO. In: TACAS. LNCS, vol. 7795, pp. 530–536. Springer (2013)
2. Abdulla, P.A., Atig, M.F., Ngo, T.P.: The best of both worlds: Trading efficiency and optimality in fence insertion for TSO. In: ESOP. LNCS, vol. 9032, pp. 308–332. Springer (2015)
3. Adve, S., Gharachorloo, K.: Shared memory consistency models: A tutorial. Computer 29(12), 66–76 (1996)
4. Alglave, J., Kroening, D., Nimal, V., Poetzl, D.: Don't sit on the fence: a static analysis approach to automatic fence insertion. ACM Trans. on Progr. Lang. and Syst. 39(2), 6 (2017)
5. Alglave, J., Maranget, L.: Stability in weak memory models. In: CAV. LNCS, vol. 6806, pp. 50–66. Springer (2011)
6. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Fences in weak memory models. In: CAV. LNCS, vol. 6174, pp. 258–272. Springer (2010)
7. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. on Progr. Lang. and Syst. 36(2), 7:1–7:74 (2014)
8. Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in TSO analysis. In: CAV. LNCS, vol. 6806, pp. 99–115. Springer (2011)
9. Bal, H., Epema, D., de Laat, C., van Nieuwpoort, R., Romein, J., Seinstra, F., Snoek, C., Wijshoff, H.: A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term. IEEE Computer 49(5), 54–63 (May 2016)
10. Barnat, J., Brim, L., Havel, V., Havlíček, J., Kriho, J., Lenčo, M., Ročkai, P., Štill, V., Weiser, J.: DiVinE 3.0 - an explicit-state model checker for multithreaded C & C++ programs. In: CAV. LNCS, vol. 8044, pp. 863–868. Springer (2013)
11. Batty, M., Donaldson, A., Wickerson, J.: Overhauling SC atomics in C11 and OpenCL. In: POPL. pp. 634–648. ACM (2016)
12. Bender, J., Palsberg, J.: A formalization of Java's concurrent access modes. In: OOPSLA. pp. 142:1–142:28. ACM (2019)
13. Biere, A., Kroening, D.: SAT-based model checking. In: Handbook of Model Checking, chap. 10. Springer (2018)
14. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: ESOP. LNCS, vol. 7792, pp. 533–553. Springer (2013)
15. Bunte, O., Groote, J., Keiren, J., Laveaux, M., Neele, T., de Vink, E., Wesselink, W., Wijs, A., Willemse, T.: The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability. In: TACAS, Part II. LNCS, vol. 11428, pp. 21–39. Springer (2019)
16. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: CAV. LNCS, vol. 5123, pp. 107–120. Springer (2008)
17. Chaki, S., Gurfinkel, A.: BDD-based symbolic model checking. In: Handbook of Model Checking, chap. 8. Springer (2018)

18. Colvin, R., Smith, G.: A wide-spectrum language for verification of programs on weak memory models. In: FM. LNCS, vol. 10951, pp. 240–257. Springer (2018)
19. Fang, X., Lee, J., Midkiff, S.: Automatic fence insertion for shared memory multiprocessing. In: ICS. pp. 285–294. ACM Press (2003)
20. Flur, S., Gray, K., Pulte, C., Sarkar, S., Sezgin, A., Maranget, L., Deacon, W., Sewell, P.: Modelling the ARMv8 architecture, operationally: concurrency and ISA. In: POPL. pp. 608–621. ACM (2016)
21. Holzmann, G.: The SPIN model checker: Primer and reference manual. Addison-Wesley Professional (2003)
22. IBM: Power ISA Version 2.06 Revision B
23. Jonsson, B.: State-space exploration for concurrent algorithms under weak memory orderings. ACM SIGARCH Computer Architecture News 36, 65–71 (2009)
24. Kahlon, V., Sinha, N., Kruus, E., Zhang, Y.: Static data race detection for concurrent programs with asynchronous calls. In: FSE. pp. 13–22. ACM (2009)
25. Kuperstein, M., Vechev, M., Yahav, E.: Partial-coherence abstractions for relaxed memory models. In: PLDI. pp. 187–198. ACM Press (2011)
26. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.K., Dreyer, D.: Repairing sequential consistency in C/C++11. In: PLDI. pp. 618–632. ACM (2017)
27. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers 28(9), 690–691 (Sep 1979)
28. Lee, J., Padua, D.: Hiding relaxed memory consistency with a compiler. IEEE Trans. Comput. 50, 824–833 (2001)
29. Linden, A., Wolper, P.: A verification-based approach to memory fence insertion in PSO memory systems. In: TACAS. LNCS, vol. 7795, pp. 339–353. Springer (2013)
30. Liu, F., Nedev, N., Prisadnikov, N., Vechev, M., Yahav, E.: Dynamic synthesis for relaxed memory models. In: PLDI. pp. 429–440. ACM (2012)
31. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In: ASPLOS. pp. 329–339. ACM Press (2008)
32. Neele, T., Willemse, T., Groote, J.: Solving Parameterised Boolean Equation Systems with infinite data through quotienting. In: FACS. LNCS, vol. 11222, pp. 216–236. Springer (2018)
33. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: TPHOLs. LNCS, vol. 5674, pp. 391–407. Springer (2009)
34. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: SPIN'07. LNCS, vol. 4595, pp. 263–267. Springer (2007)
35. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8. In: POPL. pp. 19:1–19:29. ACM (2018)
36. Shasha, D., Snir, M.: Efficient and correct execution of parallel programs that share memory. ACM Transactions on Programming Languages and Systems 10(2), 282–312 (1988)
37. Shavit, N., Touitou, D.: Software Transactional Memory. Distr. Comp. 10(2), 99–116 (1997)
38. SPARC International, Inc.: The SPARC architecture manual (version 9) (1994)
39. Sura, Z., Fang, X., Wong, C.L., Midkiff, S., Lee, J., Padua, D.: Compiler techniques for high performance sequentially consistent Java programs. In: PPOPP. pp. 2–13. ACM Press (2005)
40. Tarjan, R.: Enumeration of the elementary circuits of a directed graph. SIAM Journal on Computing 2(3), 211–216 (1973)
41. Wijs, A.J., Engelen, L.J.P.: REFINER: Towards Formal Verification of Model Transformations. In: NFM. LNCS, vol. 8430, pp. 258–263. Springer (2014)
42. Wijs, A., Bošnački, D.: GPUexplore: Many-core on-the-fly state space exploration using gpus. In: TACAS. LNCS, vol. 8413, pp. 233–247. Springer (2014)
43. Wijs, A., Neele, T., Bošnački, D.: GPUexplore 2.0: Unleashing GPU Explicit-state Model Checking. In: FM 2016. LNCS, vol. 9995, pp. 694–701. Springer (2016)