Can We Efficiently Check Concurrent Programs Under Relaxed Memory Models in Maude?

Yehia Abd Alrahman[1], Marina Andric[1], Alessandro Beggiato[1], and Alberto Lluch Lafuente[1,2(✉)]

[1] IMT Institute for Advanced Studies Lucca, Lucca, Italy
[2] DTU Compute, Technical University of Denmark, Lyngby, Denmark
albl@dtu.dk

Abstract. Relaxed memory models offer suitable abstractions of the actual optimizations offered by multi-core architectures and by compilers of concurrent programming languages. Using such abstractions for verification purposes is challenging in part due to their inherent non-determinism which contributes to the state space explosion. Several techniques have been proposed to mitigate those problems so to make verification under relaxed memory models feasible. We discuss how to adopt some of those techniques in a Maude-based approach to language prototyping, and suggest the use of other techniques that have been shown successful for similar verification purposes.

1 Introduction

As we enter the so called *multi-core era*, electronic devices made of multiple computational units that work over shared memory are becoming more and more ubiquitous. The demand of performance on such systems is likewise increasing but, unfortunately, *the free lunch is over* [1,2], that is, it is getting harder and harder to develop more performant and energy efficient single computational units. This has lead compiler constructors and hardware designers to develop sophisticated optimization techniques that in some cases may affect the intended semantics of programs. A prominent example are optimizations that give up memory consistency to accelerate memory operations. Typically, such optimizations do not affect the meaning of sequential programs, but the situation is different for concurrent programs as different threads may have subtly different (inconsistent) views of the shared memory and thus their execution may result in an unexpected (non-sequentially consistent) behaviour.

As a motivating example, consider the pseudocode in Fig. 1 which may be seen as the initial part of Dekker's algorithm for mutual exclusion. There are two threads, 1 (left) and 2 (right), whose programs are symmetric. Initially, all variables are assumed to have value 0. When thread 1 tries to enter the critical section it sets `flag1` to 1, and checks the value of `flag2`. If the value for `flag2`

Research supported by the European projects IP 257414 ASCENS and STReP 600708 QUANTICOL, and the Italian PRIN 2010LHT4KM CINA.

S. Escobar (Ed.): WRLA 2014, LNCS 8663, pp. 21–41, 2014.
DOI: 10.1007/978-3-319-12904-4_2

```
[ flag1 := 1            ]   [ flag2 := 1            ]
[ if (flag2 = 0) then   ]   [ if (flag1 = 0) then   ]
[  //critical section   ]   [  //critical section   ]
```

Fig. 1. Can both threads enter the critical section?

is 0, it assumes that thread 2 has not yet attempted to enter the critical section and proceeds to enter it. Thread 2 proceeds similarly. Under the usual model of shared memory (i.e. sequential consistency) the algorithm guarantees mutual exclusion: either one thread enters the critical section or none of them does. However, under some of the mentioned relaxations of memory consistency, it is possible for both threads to enter the critical section. Indeed, as we shall see, it may be also the case that updates on memory are delayed, so that both threads execute their updates and then read the old value 0 on each other's flag, proceeding then together into the critical section. As a matter of fact, a direct implementation of the above algorithm on Intel or AMD x86 multiprocessors yields an incorrect program.

As we shall see, many authors have developed formal semantics for these optimizations which relax the standard sequential consistency of programs. Very often this has been done by defining appropriate abstract models of shared memory called *relaxed memory models*. Using such abstractions for verification purposes is challenging in part since they introduce yet another source of nondeterminism, thus contributing to the state space explosion.

As an example consider the graph in Fig. 2. The vertical axis presents the size of the state space in terms of number of states while in the horizontal axis we have the results obtained on 1-entry versions of four mutual exclusion algorithms (Dekker, Peterson, Lamport and Szymanski) and, for each of them, three cases: the algorithm under the usual sequential consistency memory model (SC), and two versions of the algorithms under a relaxed memory model (namely, TSO). The first of these relaxed versions is incorrect, while the second one is a correct variant obtained by adding some synchronization points referred as *fences*. The results provide evidence of the state space increase due to relaxed memory models, even in the case of correct algorithms. The situation is worse if one considers that even the simple program `while true do x:=0` has an infinite state space under some memory models.

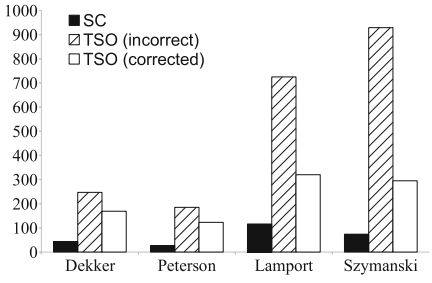


Fig. 2. State-Space Size: SC vs TSO

Several verification techniques that have been proposed to mitigate those problems in the last years aimed at making verification under relaxed memory models feasible. Some of them are described and discussed in Sects. 2 and 5. Unfortunately, those techniques are sometimes language- or model-specific and not directly applicable in the verification tasks typical of language design activities. We adopt in this work the perspective of a language designer who is willing to prototype a new language for concurrent programs under some relaxed memory model. We assume that the language designer has chosen Maude as a framework due to its suitability both as a *semantic framework* where different styles (SOS, CHAM, K, etc.) can be easily adopted [3] and as a *verification framework* featuring several tools (e.g. reachability analysis, LTL model checking, etc.). We assume that the language designer is interested in performing simple verification tasks using Maude's `search` command for the sake of testing the semantics of the language being prototyped by checking reachability properties of concurrent programs. We further assume that he is certainly not willing to modify Maude's engine for the sake of a more efficient verification and would rather resort to verification optimizations that can be realized in Maude itself. We assume that he is not willing to implement an existing technique before the language is mature enough for the development of sophisticated applications that will require state-of-the art verification techniques.

We discuss in this paper how to adopt in Maude some simple techniques to optimize the verification of concurrent programs under relaxed memory models. Some of the techniques are based or inspired by approaches to the verification of relaxed memory models or by other approaches that have been shown to be successful for similar verification purposes. We start the paper by providing in Sect. 2 a gentle introduction to relaxed memory models, mainly aimed at readers not familiar with this topic. We next introduce in Sect. 3 a running example consisting of the language Pimp, a simple language for concurrent programs, for which we provide a relaxed semantics. In Sect. 4 we discuss three families of techniques for mitigating the state space explosion due to relaxed memory models: approximations (Sect. 4.2), partial-orders (Sect. 4.3) and search strategies (Sect. 4.4). Last we discuss some of the most relevant verification techniques for relaxed memory models in Sect. 5 and draw some concluding remarks and future research in Sect. 6.

2 Relaxed Memory Models

A *memory consistency model* is a formal specification of the semantics of a shared memory, which can be a hardware-based shared-memory multiprocessor or a large-scale distributed storage. In what follows we will mainly refer to the former due to the focus on concurrent programing in our paper, but some of the inherent ideas apply to distributed settings as well. The simplest and, arguably, the most intuitive memory model is *sequential consistency* which can be seen as an extension of the uniprocessor model to multiple processors. As defined by Lamport [4], a multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some

sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. Such model imposes two requirements: (1) *write atomicity*, that is, memory operations must execute atomically with respect to each other and (2) *total program order*, which means that program order is maintained between operations from individual processors.

Sequential consistency provides a clear and well-understood model of shared memory to programmers but, on the other hand, it may affect the performance of concurrent programs since it constrains many common hardware and compiler optimizations. For instance, a common hardware optimization one can consider are write buffers with bypass capability which are used to mitigate the latency of write operations. The idea is that when a processor wants to perform a write operation, it inserts it into a write buffer and continues executing without waiting for the write to be completed. In a multiprocessor system each processor may buffer its write operations thus allowing subsequent read operations to bypass the write as long as the addresses being read differ from the address of any of the buffered writes. This clearly leads to a violation of total program order and write atomicity and hence the resulting programs are no more sequentially consistent.

Relaxed memory models provide an abstraction of the result of applying similar consistency-relaxing optimizations. If we let $X, Y \in \{Read, Write\}$, then *X-to-Y* denotes the relaxation that allows to violate the program order by performing a Y operation before an X operation that appears before in the program. The instances of this relaxation are *Write-to-Read*, *Write-to-Write*, *Read-to-Read* and *Read-to-Write*. Two common relaxations of write atomicity are *Read other's write early*, which allows a read operation to return the value of another processor's write before the write is made visible to all other processors, and *Read own's write early* which allows a read operation to return the value of its own previous write, before it is made visible to other processors.

Figure 3 depicts a hierarchy of memory models [5] and how they relate to each other based the relaxations they allow. The strictest model is sequential consistency (SC) which does not allow any relaxation. In the second category fall *total store order* (TSO), *processor consistency* (PC) and IBM-370 as they allow the *Write-to-Read* relaxation, all other program orders are maintained. The third category comprises of *partial store order* (PSO) that allows both *Write-to-Read* and *Write-to-Write* relaxations. The models at the bottom of the hierarchy also allow *Read-to-Read* and *Read-to-Write* reorderings and hence are the least strict.

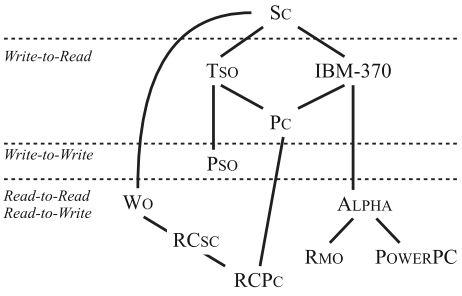


Fig. 3. Hierarchy of relaxations

The formalization of memory models has been mainly motivated by the fact that many processor vendors often do not provide clear architectural specifications of the underlying memory models [6]. Instead, the documentation is

typically given in (sometimes ambiguous) informal prose, which makes it hard to program above or to reason about. The formalization of memory models has been tackled by several authors in different styles: e.g. axiomatic and operational. Some examples are the axiomatic x86-TSO model [7], which is sound with respect to the Intel and AMD architectures, the axiomatic models for Java and C++ languages [8,9], and the operational models of relaxed memories for programming languages [10,11]. It is also worth to remark the efforts towards unifying frameworks to capture most memory models [12] and the theory of memory models of [13].

We focus in this work on the TSO model which we introduce first informally, following the usual operational description based on the architectural view depicted in Fig. 4: (i) each processor has a write buffer for storing write operations, and each processor corresponds to one thread; (ii) a thread that performs a read operation must read its most recent buffered write if there is one, otherwise reads are taken from shared memory; (iii) a thread can see its own writes before they are made visible to other threads by committing the pending writes to memory; (iv) delayed updates are committed from the buffer to the memory non-deterministically by the multiprocessor system, one-by-one and respecting their arrival order; (v) the programmer can use the `mfence` instruction to wait until a buffer is fully committed, so to enforce memory order between preceding and succeeding instructions. The next section will present a formal semantics of a language running under this memory model.

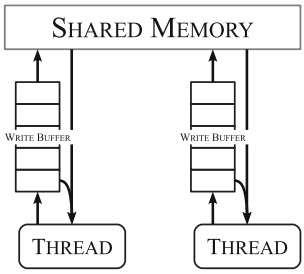


Fig. 4. TSO architecture

3 A Simple Language with Relaxed Concurrency

We introduce in this section a simple language called PIMP that we shall use as a case study and as a running example. Basically, PIMP is a simple imperative language reminiscent of the WHILE and IMP languages [14] enriched with some few concurrency features including shared memory communication, and blocking wait and fence operations. In few words, PIMP allows one to specify sequential threads that communicate over shared memory.

Definition 1 (threads). *The programs of* PIMP *threads are terms generated by S in the following grammar:*

$$S ::= \texttt{skip} \mid \texttt{x} := u \mid \texttt{mfence} \mid S \texttt{ ; } S' \mid \texttt{if } B \texttt{ then } S \texttt{ else } S' \mid \texttt{while } B \texttt{ do } S \mid \texttt{wait } B$$

were $\mathcal{X}$ is a set of variables, $\texttt{x} \in \mathcal{X}$, $u \in \mathbb{N}$ and B is a Boolean expression on $\mathcal{X}$.

Most of the syntactic constructs of the language are rather standard. We just mention here the `mfence` primitive (used to block a thread until its local view of memory is consistent) and the `wait` primitive (used to specify blocking guards). The construct `skip` is used to denote (immaterial) complete computations.

Definition 2 (programs). *The programs of* Pimp *are terms generated by* T *in the following grammar:*

$$\begin{array}{lll} P ::= \langle T, M \rangle & & T ::= [\![S \,/ N]\!]_i \mid T \parallel T' \\ N ::= \emptyset \mid \mathtt{x} \mapsto u \mid N \bullet N & & M ::= \emptyset \mid \mathtt{x} \mapsto u \mid M, M \end{array}$$

where $\boldsymbol{x} \in \mathcal{X}$*, and* $i, u \in \mathbb{N}$*.*

Programs are obtained by the parallel composition of sequential threads (denoted by juxtaposition). Each thread is indexed with a unique identifier i. Such identifier is later used to ease the presentation of some concepts but we often drop it when unnecessary. Each thread comes equipped with a (possibly empty) local memory N made of a composition of memory updates. In the case of Tso, N models a buffer. Programs are turned into *program configurations* (i.e. terms generated by P) by equipping them with a shared memory M, which we assume to denote a function $M \in (\mathcal{X} \rightharpoonup \mathbb{N})$ which may be partial on $\mathcal{X}$ but is certainly total on the variables of the program. In the definition above $\_,\_$ is considered to be associative, commutative and idempotent with $\emptyset$ as identity and with no two maps on the same variable. We shall also use the concept of *thread configuration*, i.e. tuples $\langle S, N, M \rangle$, where S is the program of the thread, N is its local memory (buffer), and M is the global (shared) memory. Thread configurations ease the presentation of the semantics, by allowing us to focus on individual threads thanks to the interleaving semantics of parallel composition.

Memory Views. An important concept that eases the presentation of the semantics is the *memory view*, which allow us to formalize the thread's local view on memory. More precisely, threads perceive memory as a particular composition of the shared memory M and their local memory N. We shall see that in the case of the Sc model there is in practice no local memory, and threads will only perceive M as the available memory. In the case of Tso, the thread's view on memory will be $M \circ M'$, where M' is obtained by rewriting N as a set (denoted $N \rightarrow_{\mathcal{F}} M'$) and $\circ$ is a composition operation defined by

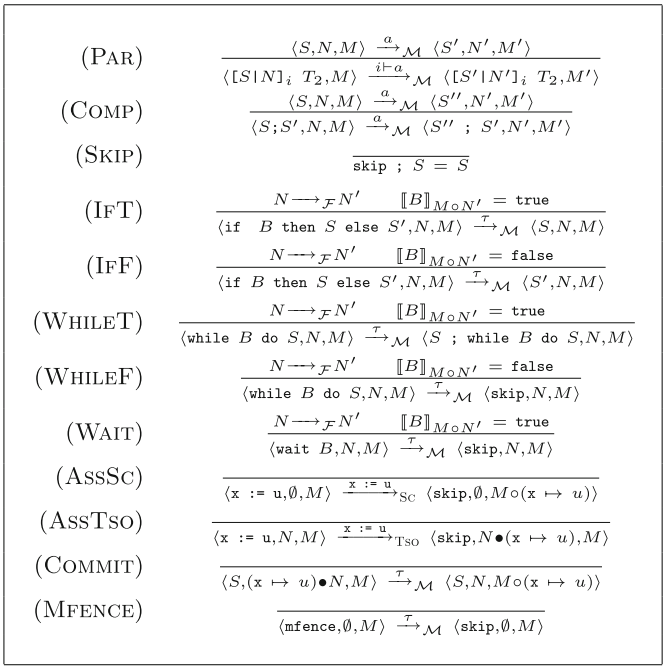
$$\begin{array}{rll} M \circ \emptyset = M & & \\ (M, \mathtt{x} \mapsto u) \circ (M', \mathtt{x} \mapsto v) = (M, \mathtt{x} \mapsto v) \circ M' & & \\ M \circ (M', \mathtt{x} \mapsto v) = (M, \mathtt{x} \mapsto v) \circ M' & \quad \mathtt{if}\ M(\mathtt{x}) = \bot & \end{array}$$

Rewriting a local memory N as a memory M is formalized by the following rule to be applied top-down

$$\frac{N \longrightarrow_{\mathcal{F}} M}{N \bullet \mathtt{x} \mapsto u \longrightarrow_{\mathcal{F}} M \circ \mathtt{x} \mapsto u}$$

Note that, in principle, $\longrightarrow_{\mathcal{F}}$ has a functional behavior, i.e. for a given local memory N there is only one possible rewrite $N \longrightarrow_{\mathcal{F}} M$. In this case we may use $\mathcal{F}(N)$ to denote M. Later, however, we shall consider variants of local memories where $\bullet$ obeys some axioms and since we consider terms up to structural

Table 1. Rules of the operational semantics of Pimp



equivalence $\longrightarrow_{\mathcal{F}}$ may have a non-functional behavior. An example of this will be used later to model non-deterministic evaluations of Boolean expressions for abstract local memories, where $\bullet$ enjoys the axioms of sets.

Pimp *Semantics.* The semantics of Pimp under a memory model $\mathcal{M} \in \{\text{Sc}, \text{Tso}\}$ is a labeled transition system whose states are program configurations and whose transitions $\rightarrow_{\mathcal{M}} \subseteq P \times A \times P$ are defined by the rules of Table 1. In the presented rules, program and thread configurations are to be intended up to a structural equivalence relation induced by the axiomatization of programs as multisets of threads (i.e. juxtaposition $\_\ \_$ is AC), memories as sets of updates (i.e. memory composition $\_,\_$ is ACI with identity $\emptyset$) and buffers as lists of updates (i.e. buffer composition $\_\bullet\_$ is A with identity $\emptyset$). The labels in A are used for the only purpose of decorating the semantics with information that will be useful in the verification techniques presented in later sections. At this point it is sufficient to understand that A contains labels of the form $i \vdash a$, where i is either a thread or buffer identifier and a is some action associated to the rules of the semantics, essentially used to record in some cases the statement associated to the transition. Label τ is used to denote some transition whose origin is not relevant. We sometimes drop labels from transitions.

Rule Par is the only rule for program configurations and specifies the interleaving semantics of the language. The rest of the rules specify then how individual threads evolve independently based on (and possibly modifying) the shared

memory M and their local buffer N. Rule Comp is defined as usual. It is however worth to remark that `skip` is treated as the left identity of sequential composition to get rid of completed executions. The semantics of control flow constructs is rather standard and defined by rules IfT, IfF, WhileT and WhileF. Such rules are defined in a big-step style, i.e. the evaluation of a binary expression B in some local view of memory M', denoted by $[\![B]\!]_{M'}$ is performed atomically as one thread transition. It is worth to remark that such evaluation is done based on the thread's view of the available memory (i.e. $M \circ N'$). Note that in the case of Sc, N' is always $\emptyset$ so that threads observe M directly. Rule Wait specifies the semantics of the `wait` primitive which blocks the execution until the binary condition B holds. The evaluation of the binary condition B is done in the same manner as for control flow constructs. The semantics of assignments depends on the memory model under consideration. We actually consider two cases defined respectively by rules AssSc (for Sc) and AssTso (for Tso). Rule AssignSc is as usual: an update is directly performed on the shared memory M. In the case of Tso, the story is totally different. Indeed, rule AssTso models the fact that updates are delayed by appending them to the thread's buffer N. The delayed updates in the buffers can be non-deterministically committed to memory in the order in which they arrived. This is specified by rule Commit. A memory commit consists in removing the update $(\mathtt{x} \mapsto u)$ at the beginning of the write buffer of any thread and updating the value of variable `x` in memory. Finally, rule Mfence specifies the semantics of the `mfence` primitive, which blocks the thread until its write buffer becomes empty.

We assume that the reader has some familiarity with the canonical approaches to encode operational semantic styles in rewriting logic and Maude are detailed [3]. We hence do not provide a detailed explanation on how to specify a Maude interpreter for Pimp. In few words, the main idea is to specify a rewrite theory $\mathcal{R}_{\text{Pimp}} = \langle \Sigma, E \cup A, R \rangle$ as a Maude module where (i) signature Σ models syntactic categories as sorts and contains all function symbols used in terms, (ii) equations and axioms $E \cup A$ model the above mentioned structural equivalence on terms, and (iii) rules R model the rules of the operational semantics. The obtained encoding is faithful in the sense that there is a one-to-one correspondence between transitions and 1-step rewrites of configuration-sorted terms.

4 Tackling the State Space Explosion

This section presents a number of techniques to tackle the state space explosion caused by relaxed memory models. More precisely, Sect. 4.2 deals with approximation techniques, focusing mostly in avoiding the generation of infinite state spaces due to the potentially unlimited growth of store buffers; Sect. 4.3 presents a partial order reduction technique aimed at reducing the number of interleavings introduced by the non-deterministic nature of relaxed memory models, and Sect. 4.4 discusses heuristic search strategies that can be adopted in order to detect bugs in a more efficient way by guiding the search towards error states and thus exploring a smaller portion of the state space.

4.1 Preliminaries

We introduce here some basic notation that we shall use in the rest of this section. First, we shall consider Kripke structures as semantic model for verification problems. These are obtained as usual, i.e. by enriching transition systems with observations on states, so to abstract away from concrete representation of states (as program configurations) and by restricting to reachable configurations only.

Definition 3 ($\mathcal{M}$-Kripke structure). *An* $\mathcal{M}$-Kripke structure *is a Kripke structure* $(S, s_0, \rightarrow, \mathcal{L}, AP, \mathcal{M})$ *where:* $S \subseteq P$ *is the set of* s_0*-reachable configurations, i.e.* $\{s \in P \mid s_0 \rightarrow^*_{\mathcal{M}} s\}$*;* $s_0 \in P$ *is the initial state;* $\rightarrow \subseteq S \times A \times S$ *is a transition relation defined as* $(P \times A \times P) \cap \rightarrow_{\mathcal{M}}$*, i.e. the restriction of* $\rightarrow_{\mathcal{M}}$ *to reachable states;* $\mathcal{L} : S \rightarrow 2^{AP}$ *a labeling function for the states;* AP *is a set of atomic propositions; and* $\mathcal{M} \in \{\textsc{Tso}, \textsc{Sc}\}$ *is a memory model.*

$\mathcal{M}$-Kripke structures are like ordinary Kripke structures with an explicit reference to the underlying memory model $\mathcal{M}$ and the corresponding transition system semantics $\rightarrow_{\mathcal{M}}$. In what follows we shall often fix $\mathcal{M}$ to be Tso unless stated otherwise. We shall also use the term *initial Kripke structure* for some program T to denote some Kripke structure whose initial state is an *initial configuration* $\langle T, M\rangle$, i.e. a configuration where M maps all variables of T to 0.

Some of the techniques we shall consider allow us to obtain for a given Kripke structure another (possibly smaller one) which is semantically related. We assume familiarity with the usual notions of state-based equivalences and preorders, such as weak/strong (bi)simulation and (stuttering) trace equivalence. Those semantic relations with respect to the observations on states specified by the labelling function $\mathcal{L}$ and the proposed techniques depend on the properties of $\mathcal{L}$. For a labelling function $\mathcal{L}$ we denote by $\equiv_{\mathcal{L}} \subseteq P \times P$ the equivalence relation on program configurations induced by labelling equality. Often, we shall require that $\mathcal{L}$ cannot distinguish states identified by some other equivalence relation R, i.e. that $\equiv_{\mathcal{L}} \supseteq R$. For example, consider the smallest congruence relation induced by axiom $[S, N] = [S, N']$, denoted $\equiv_{[S,N]=[S,N']}$, which identifies program configurations up to their local memories. Then requiring $\equiv_{\mathcal{L}} \supseteq \equiv_{[S,N]=[S,N']}$ amounts to require that $\mathcal{L}$ cannot observe local memories.

4.2 Approximations

Consider the simple sequential thread $p \triangleq$ `while true do x:=0` and the initial configuration $s = \langle [p \mid \emptyset], \mathtt{x} \mapsto 0\rangle$. Any initial Kripke structure $(S, s, \rightarrow, \mathcal{L}, AP, \textsc{Sc})$ is clearly finite-state and just composed of state s with a self loop $s \rightarrow s$. However, the same program under TSO has an infinite state space, i.e. Kripke structures $(S, s, \rightarrow, \mathcal{L}, AP, \textsc{Tso})$ have infinitely many states since it is possible to iterate the body of the `while` infinitely many times, each time adding an entry to the buffer: $s \rightarrow \langle [\mathtt{x:=0}; p \mid \emptyset], \mathtt{x} \mapsto 0\rangle \rightarrow \langle [p \mid \mathtt{x} \mapsto 0], \mathtt{x} \mapsto 0\rangle \rightarrow \langle [\mathtt{x:=0}; p \mid \mathtt{x} \mapsto 0], \mathtt{x} \mapsto 0\rangle \rightarrow \langle [p \mid \mathtt{x} \mapsto 0 \bullet \mathtt{x} \mapsto 0], \mathtt{x} \mapsto 0\rangle \rightarrow \ldots$ The unbounded growth of buffers is indeed one of the most challenging issues in the verification of concurrent programs under relaxed memory models and several approaches

have been proposed in the literature as we shall discuss later in Sect. 5. In this section we discuss one simple approach that one may easily adopt in Maude. For the sake of illustration we present as well some simple ideas that cannot be easily turned into useful sound approximations.

Simple Approximations. We shall consider in this section a simple approach to realize approximations based on equating program configurations. This kind of approximations can be easily implemented in Maude by language designers since they require minimal changes in the formal specification of the language, such as changing the equational attributes of some function symbols or introducing some equations. Moreover, the Maude literature offers approaches, such as equational abstractions [15] and c-reductions [16], to realize such kind of approximations in a disciplined way, and to use possibly tool-based proof techniques to prove their soundness and eventually correct them.

In this paper we will essentially follow an approach based on equational abstractions [15]. The main idea is to consider some axioms A of the form $t = t'$ where t, t' are terms denoting part of the program or thread configurations. Such laws will then be then used to specify a rewrite theory $R_{\text{PIMP/A}}$ which specifies the approximated semantics. This is realized in Maude by introducing the axioms of A as equational attributes or as equations in R_{PIMP}, the Maude specification of PIMP. The effect is that for a Kripke structure K we obtain an approximated Kripke structure K_{A} that, under some reasonable conditions on $\mathcal{L}$ (e.g. not being able to distinguish states identified by A), should simulate K. In some cases concrete transitions may not have an approximated counterpart, a situation that we can repair by introducing additional rules in the semantics. The final effect of the approximations is that more states will be identified thus resulting in smaller state spaces.

Buffers as Ordered Maps. Let us start considering the following simple approach to get rid of the unbounded growth of TSO buffers. The idea is to keep in the buffer just the latest update for each variable removing older updates. The rationale would be that by doing so we still preserve the order of write operations and all write operations will certainly have their chance to be committed to memory. We are just forbidding to delay updates on the same variable too much. This can be formalized by considering the following simple equation

$$\text{OM} \triangleq (x \mapsto u) \bullet N \bullet (x \mapsto v) = N \bullet (x \mapsto v)$$

The question is whether this would result in a useful approximation. We shall see that this is not the case. Let us consider first the PIMP program T in Fig. 5(a) and let K be some initial Kripke structure for it. It is easy to see that in K it is not possible to reach a state in which z takes the value 1, while this is possible in K_{OM}. The reason is, of course, that there exists an approximated execution in which the commit of $(x \mapsto 1)$ is discarded, and thus never visible to the second thread. For the same reason deadlock states can be introduced as well.

Of course, obtaining new *spurious* behaviors is usual when considering over-approximations. However, one would expect then that no concrete behavior is lost.

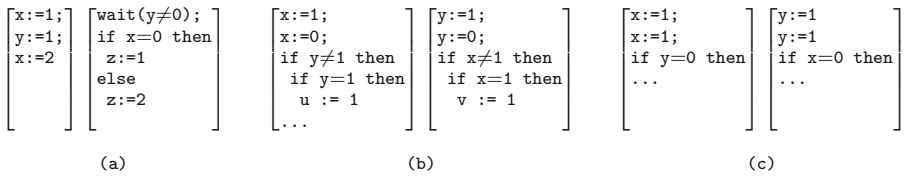


Fig. 5. Some concurrent programs

We can also observe that this is unfortunately not the case. Consider, a concurrent program T of the form in Fig. 5(b) and some initial Kripke structure K for it. It is easy to see that in K we can reach a configuration where both u and v have value 1. This can happen if both threads perform and delay their first two assignments, then enter the first branch of their first if statement, then commit their first pending write and finally enter the first branch of their second if statement thus proceeding to the update of u and v. Such behavior is however not possible in K_{OM}. Essentially, the considered approximation implies a loss of information that could only be recovered by considering an approximated semantics that would take into account all potentially (infinitely many) pending updates that could have been removed. Therefore, while simple, the idea of removing old updates from buffers is unlikely to provide a useful approximation.

Forced-commit Approximation. Let us consider now a similar idea, based on handling updates on the same variables in a slightly different way. Suppose we allow more than one update on the same variable with one exception: if the head of the buffer is an update $\mathtt{x} \mapsto u$ on a variable x for which there is another update $\mathtt{x} \mapsto v$ on the tail of the buffer, we remove the update $\mathtt{x} \mapsto u$ but this time we commit it to memory. This time the equation under consideration is

$$\textsc{Fc} \triangleq \langle P, (x \mapsto u) \bullet N \bullet (x \mapsto v), M \rangle = \langle P, N \bullet (x \mapsto v), M \circ (x \mapsto u) \rangle$$

Can we exploit Fc to build a useful approximation? It can be shown that under some reasonable constraints on $\mathcal{L}$ no observable behavior is introduced, but unfortunately, this approximation does not solve the problem of infinite state spaces as illustrated by the simple program `y:=0; while true do x:=0`. Clearly, there is an execution that delays and never commits the pending update $(\mathtt{y} \mapsto 0)$ while cumulating repeated updates $(\mathtt{x} \mapsto 0)$. Moreover, this approximation loses some behaviours. This can be seen by a program T such as the one in Fig. 5(c). In the initial Kripke structure K for T, it is possible for both threads to execute the then branch of the if statement. However, in the corresponding approximated Kripke structure K_{FC} it is no longer possible to reach a state in which both threads execute the then branch of the if statement.

Buffers as Set of Updates. What would then be a simple and still sound approximation? A simple idea to get rid of multiple copies of a same commit in a buffer is to approximate the buffer with a set of updates. This amounts to consider

those axioms Us that make the $\bullet$ operation become associative, commutative and idempotent, with identity $\emptyset$. Even if it may seem that we are again losing information, in particular about the order and multiplicity of pending updates, this is not exactly true. The reason is that now the evaluation of expressions automatically becomes non-deterministic due to the non-deterministic nature of $\rightarrow_{\mathcal{F}}$ that would consider all possible orderings among updates. What is left is the multiplicity of updates, which can be easily handled by introducing the rule

$$(\textsc{CommitUs})\quad \langle S, (\mathtt{x} \mapsto \mathtt{u}) \bullet N, M \rangle \xrightarrow{\tau}_{\textsc{Tso}} \langle S, (\mathtt{x} \mapsto \mathtt{u}) \bullet N, M \circ (\mathtt{x} \mapsto \mathtt{u}) \rangle$$

in the semantics. The rationale is that any update can be committed to memory but still kept in the buffer as it may represent an arbitrary number of copies of the same update. An advantage of this approximation is that it can be easily realized in Maude by changing the equational attributes of $\bullet$ and by adding a rule modeling rule CommitUs. The obtained approximation may exhibit "spurious" behaviours, but we are guaranteed to not miss any concrete behaviour provided that we do not want to observe too much information on states.

Proposition 1. *Let K be a* Tso*-Kripke structure whose labeling function $\mathcal{L}$ is such that $\equiv_{\mathcal{L}} \supseteq \equiv_{[S,N]=S[S,N']}$. Then $K_{\textsc{Us}}$ simulates K.*

Experiments. Figure 6 presents the results of some of our experiments. The vertical axis corresponds to the size of the state space in terms of number of states. In the horizontal axis we have our four mutual exclusion algorithms and, for each of them, the result obtained without (1st column) and with the above discussed approximations: Om (2nd column), Fc (3rd column), and Us (4th column). Clearly, not all explorations make sense since some of the approximations are unsound or incomplete but we included them here for a more comprehensive presentation of our experiments. The most relevant observation is that simple approximations such as Us, do provide finite state spaces but may enormously contribute to the state space explosion. This is evident in the considered mutual exclusion programs which are finite-state since they are 1-entry instances of the algorithms (i.e. with no loop). Section 5 discusses several sophisticated techniques that can provide more efficient approximations.

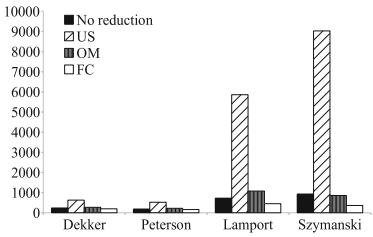


Fig. 6. Approximated state spaces

4.3 Partial-Order Reduction

As we have seen, relaxed memory models introduce a large amount of non-determinism in the state space of concurrent programs. In the case of the Tso, this is due to the introduction of buffers, which delay updates that are non-deterministically committed at any time. Such non-determinism may lead to

an increase of the interleaving of actions some of which may be equivalent. A popular and successful family of techniques to cope with this problem is *Partial Order Reduction* (POR) [17–19]. These techniques have been extended and implemented in several ways and are often part of the optimization features of verification tools such as model checkers, and they have been already successfully applied in the verification of programs under relaxed memory models [20,21].

POR in Maude. An easy way to adopt POR in Maude-based verification is to instantiate the generic language-independent approach described in [22], that we shall refer to as PorM. The method discharges relatively little requirements on a language designer: (i) a formal executable specification of the semantics of the programming language L under consideration (Pimp in our case) as a rewrite theory $\mathcal{R}_L$ satisfying some reasonable conditions explained below, and (ii) the specification of some properties of the language (e.g. an approximation of dependencies between actions). The latter, of course, may require some manual proof. The advantages of the method are that no additional proof is needed to guarantee the correctness of the approach, and that no change in the underlying verification capabilities of Maude are necessary.

We recall that the main idea underlying the ample set approach to POR [18], considered in PorM, is to prune *redundant* parts of the state space, avoiding the exploration of paths that do not bring additional information. This is done by considering at each state s a subset of its successors called *ample* set. For presentation purposes, we recall now some useful definitions. Let K be the Kripke structure under consideration. We denote with $enabled(s)$ the set of all the enabled transitions in state $s \in S$, i.e. $enabled(s) = \{s \xrightarrow{\alpha} s'\}$. We sometimes use the notation $t(s)$ to denote the target s' of transition $t = s \xrightarrow{\alpha} s'$. Two fundamental concepts in POR are those of *invisibility* of actions and *independence* between actions and between transitions.

Definition 4 (invisibility). *Let K be a Kripke structure. A transition $s \xrightarrow{\alpha} s'$ is* invisible *in K iff $\mathcal{L}(s) = \mathcal{L}(s')$. Similarly, an action α is invisible if all transitions $s \xrightarrow{\alpha} s'$ are invisible.*

Definition 5 (independence). *Two transition t_0, t_1 are* independent *if for each state s such that $t_0 \in enabled(s)$ and $t_1 \in enabled(s)$ it holds: $t_1 \in enabled(t_0(s))$, $t_0 \in enabled(t_1(s))$, and $t_0(t_1(s)) = t_1(t_0(s))$. We define the* independence relation *$\mathcal{I} \subseteq T \times T$ as $\{(t_0, t_1) \mid t_0$ and t_1 are independent$\}$.*

In words, independent transitions do not disable each other, and their execution commutes. If two transitions are not independent, we say that they are *dependent*. We let $\mathcal{D}$ be simply defined as $\mathcal{D} = (T \times T) \setminus \mathcal{I}$.

Instantiating PorM *to* Pimp. We recall that the PorM approach imposes some restrictions on the language under consideration as well as on the approximation of the dependency relation. The conditions on the language are: *"(1) In each program there are entities equivalent to threads, or processes, which can be uniquely identified by a thread identifier. The computation is performed as the combination of local computations inside individual threads, and communication between*

these threads through any possible discipline such as shared memory, synchronous and asynchronous message passing, and so on. (2) In any computational step (transition) a single thread is always involved. In other words, threads are the entities that carry out the computations in the system. (3) Each thread has at most one transition enabled at any moment." Clearly, PIMP satisfies those conditions by viewing buffers as independent computation entities (whose only actions are to commit updates to memory).

The strategies to compute ample sets discussed in [22] guarantee correctness given that the language designer provides a safe approximation of dependencies between transitions, and a correct specification of visibility. Regarding visibility, the idea we propose here for PIMP relies on the fact that, as long as the properties of interest do not concern the local memories or the program itself, all the transitions caused by assignments are invisible. This leads to the first lemma needed to ensure the correct instantiation of the PORM approach.

Lemma 1 (invisibility of assignments). *Let K be a Kripke structure. If $\mathcal{L}$ is such that $\equiv_{\mathcal{L}} \supseteq \equiv_{[S,N]=[S',N']}$ then all actions $\alpha = i \vdash x := u$ are invisible in K.*

Furthermore, it is easy to convince ourselves that the only way a transition could be dependent on an assignment transition, is to be generated by the execution of an instruction of the same thread following the assignment itself. Indeed, we hence define the following over-approximation of $\mathcal{D}$.

Definition 6 (dependency approximation). *Let K be a Kripke structure and let $F \subseteq A \times A$ be the relation on actions made of all pairs of actions (α, β) or (β, α) such that $\alpha = (i \vdash x := u)$, $\beta = (j \vdash a)$ and $i \neq j$. We define $D \subseteq T \times T$ as the set $(T \times T) \setminus \{(s \xrightarrow{\alpha} s', s'' \xrightarrow{\beta} s''') \mid (\alpha, \beta) \in F\}$.*

Lemma 2 (approximation of dependency). *Let K be a Kripke structure. We have $\mathcal{D} \subseteq D$.*

Of course, D is a very simple and coarse approximation but it serves well our illustrative purposes and can be easily implemented. Indeed, the simplest strategy of PORM consists of considering single transitions as candidates for ample sets. For a single transition t to be accepted as ample set it must be invisible (C2 in [22]), such that no other thread has a transition in the future that is dependent on t (C1' in [22]) and should not close a cycle in the state space (C3 in [22]). In our case, our approximation of dependency makes transitions corresponding to assignments obvious candidates. If we denote by $ample : P \to 2^P$ the function computing ample sets that the simplest strategy of PORM implements and if we let $K = (S, s_0, \to, \mathcal{L}, AP, \mathcal{M})$ be an *$\mathcal{M}$-Kripke structure*, then the PORM reduction of K is $K_{\text{PORM}} = (S, s_0, \to \cap \{(s, s') \mid s' \in ample(s)\}, \mathcal{L}, AP, \mathcal{M})$.

Proposition 2 (Soundness). *Let K be a Kripke structure whose labeling function $\mathcal{L}$ is such that $\equiv_{\mathcal{L}} \supseteq \equiv_{[S,N]=[S',N']}$, then K and K_{PORM} are stuttering bisimilar.*

The correctness of Proposition 2 trivially follows from Theorem 1 of [22] and Lemmas 1 and 2.

Experiments. Figure 7 presents the results of our experiments. The vertical axis corresponds to the size of the state space in terms of number of states. In the horizontal axis we have our four mutual exclusion algorithms and, for each of them, the result obtained without (left) and with (right) POR. The obtained result provides evidence of the advantages of applying POR even in the simple form presented here.

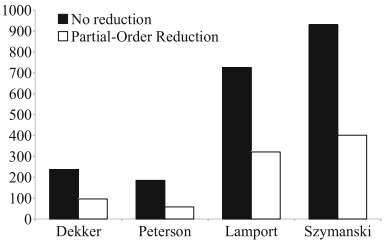


Fig. 7. Reduction with POR

4.4 Search Strategies

Verification tools based on explicit-state state space traversal often use quite simple but efficient search algorithms based on depth-first and breadth-first strategies. This is indeed the case of the standard verification capabilities of Maude: the `search` command performs a breadth-first search traversal of the state space of a rewrite theory, while the Maude LTL model checker [23] applies the usual nested depth-first search algorithm for checking emptiness of ω-regular languages. However, as many authors have noticed, using *smart* search strategies can provide better verification performances, both in the time and the memory consumed by the verification tool. The application of such techniques is often known in the model checking community by *directed model checking*, a term originally coined in [24], and made popular by its adoption in several model checkers such as SPIN [25] and Java Path Finder [26].

The main idea underlying such techniques is the use of search algorithms whose exploration strategy depends on some heuristics that aim at exploring a portion of the state space that is as small as possible for the required verification task. The archetypal example is the use of standard AI algorithms such as A* and best-first search in combination with heuristics that rank the states according to their likelihood to lead to a violation of the property being verified. Bug-finding, indeed, rather than verification, is the killer application of such techniques.

Search strategies are not novel in the Maude community. Indeed, they have been thoroughly investigated in [27]. In the proof-of-concept spirit of this work we have followed a very simple approach to give evidence of the advantages of using heuristically guided search algorithms in the verification of concurrent programs under relaxed memory models. In particular, we have implemented and evaluated the best-first search algorithm in combination with simple heuristics.

We recall that the best-first search algorithm works by maintaining two sets of states: a set of *closed* states (i.e. visited states whose transitions have been already explored) and a set of *open* states (i.e. visited states whose transitions are yet to be explored). The algorithm starts with an initially empty set of closed states and only the initial state in the open set, and iteratively selects one open state to be *expanded* and moved to the close set. Expanding a state means exploring the states immediately reachable through outgoing transitions

and putting them in the open set if they have been never visited before. The choice of the open state to be expanded depends on some heuristic function that ranks the states according to some rationale. Our implementation is rather canonical and exploits the reflective capabilities of the Maude language. Since Maude's meta-level module offers a `metaSearch` command to obtain the outgoing transitions of a state, the implementation of best-first search in a declarative way is almost straightforward.

Our heuristics are inspired by the work of Groce and Visser on so-called *structural* heuristics for directed model checking in the Java Path Finder model checker [26]. Such heuristics are based on inherently structural causes of errors in concurrent software. For instance, some of the heuristics tend to promote the exploration of executions with more context-switches or interleavings with the rationale that programmers tend to think sequentially and bugs such as race conditions often are due to unexpected interleavings. With this spirit we have designed three simple heuristics tailored for finding bugs in concurrent programs to be run under relaxed memory models, all of which map program configurations into natural numbers with the idea that program configurations with higher values are more likely to lead to a bug. The heuristics respectively count the number of non-empty buffers (NEB), the number of pending writes (PW), i.e. the sum of the sizes of all buffers, and the number of *inconsistent* pending writes (IPW), i.e. the number of pending writes $\mathtt{x} \mapsto u$ that map a variable `x` to a value u different from the value v assigned to `x` by the shared memory. These heuristics are measures of the level of memory inconsistency.

Figure 8 presents the results of our experiments. As usual, the vertical axis presents the number of states that were explored. In this case we were looking for violations of the mutual exclusion property and the verification stopped once the first violation was found. In the horizontal axis we have our four mutual exclusion algorithms and, for each of them, four cases: the usual breadth-first (BFS) search and best-first (BF) search in combination with the three heuristics. Without entering into details, the main observation is that the heuristically guided search for errors is in general more space efficient than the standard algorithm. Of course, there is a slight time overhead in our implementation since breadth-first search is implemented in Maude itself (using the meta-level) and the search command is provided by the Maude (C++) engine directly. However, our main point here is to show the potential of heuristically guided search strategies that may be prototyped using the meta-level (as we do here) and eventually implemented as extensions of the Maude engine if high time performance is needed.

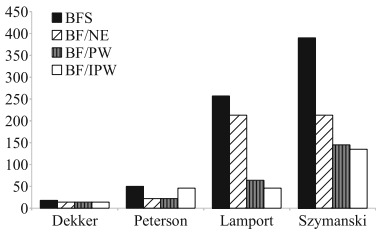


Fig. 8. Bug-finding with heuristics

5 Related Works

We discuss here some approaches, focusing the discussion on those that have inspired the techniques we have adopted in our case study and describing as well some archetypal examples of alternative techniques.

Partial order reduction techniques have been applied to the verification of concurrent programs under relaxed consistency by several authors. For instance, the authors of [20] use the SPIN model checker and exploit SPIN's POR based on the ample set approach [18], while the authors of [21] combine different techniques (some discussed below) which include an implementation of the persistent set approach to POR [17]. Those works should not be confused with the *partial order* models used [28], whose authors address the problem of program verification under relaxed memory (Tso, Pso, Rmo and Power) by using partial orders to model executions of finite-state programs. Those models are then analyzed using a SAT-based technique called *symbolic decision procedure for partial orders*, implemented in the *Bounded Model Checker for ANSI-C programs (CBMC)* [29]. The key idea is the partial order model, which is a graph whose nodes are the read/write operations of the program and whose directed arcs model the data and control dependency between operations. While the data dependency cannot be relaxed, the control dependency is relaxed according to the memory model under consideration. The absence of undesirable properties (e.g. possibility of reading certain values) is reduced to checking the presence of cycles in the graph.

Several approximation techniques for concurrent programs under relaxed consistency can be found in the literature (e.g. [30–32]). A representative example is described in [33], whose authors propose a verification approach for concurrent programs under Tso. The key idea is to approximate the (possibly unbounded) store buffers in a way that not only makes verification under Tso feasible, but also reduces the reachability problem under Tso to a reachability problem under Sc, thus enabling the use of off-the-shelf Sc analysis tools (as other authors do, e.g. [34]). The approach is based on *context-bounded analysis* [35]. A *context* is a computation segment where only one thread is active. All memory updates within a context are the result of committing delayed updates in the store buffer of the active thread. The authors prove that for every concurrent program P, it is possible to construct another concurrent program P' such that when P' runs under Sc, the reachable states of P' are exactly the same of the reachable states of P running under Tso with at most k context-switches for each thread. Their translation is done with a limited overhead, i.e. a polynomial increase in the size of the original program. The authors show that it is possible to use only a k-dependent fixed number of additional copies of the shared variables as local variables to simulate the store buffers, even if they are unbounded. The key assumption is that each store operation produced by some thread cannot stay in its store buffer for more than a bounded number of context switches of that thread. As a consequence, for a finite-state program, the context-bounded analysis of Tso programs is decidable. Such sort of bounded verification is proposed by other authors and there are also approaches that address the infinite state

space by resorting to predicate abstractions (e.g. [36]) or symbolic approaches (see e.g. [37–40])). A prominent example are *buffer automata* [41].

Apart from the already mentioned CBMC [29] several other verification tools have been conceived with the aim of supporting the development of correct and efficient C programs under relaxed memory models. For instance, *CheckFence* [42] is a tool that statically checks the consistency of a data type implementation for a given bounded test program and memory model (Tso or Rmo). The tool checks all concurrent executions of a given C program under relaxed consistency and produces a counterexample if it finds an execution that is not sequentially consistent. Another example is the tool *DFence* [43] which implements a technique that, given a C program, a safety property and a memory model (Tso or Pso), checks for violations of the property and infers fences to constrain problematic reorderings causing the violations. Finally, it is worth mention that the specification of KernelC in the K framework [44] includes as well a x86-Tso semantics of the memory models [45] which allow one to use the K tools (some of which are Maude-based) for verification purposes.

6 Conclusion

This paper addresses one of the problems that a language designer may encounter when prototyping a language for concurrent programs with a weak shared memory model, namely the state space explosion due to relaxed consistency. We have discussed how the flexibility of the Maude framework can be exploited to adopt some efficient verification techniques proposed in the literature. We have essentially focused on reachability analysis, since it plays an important role in the development of concurrent programming languages and programs, not only for verification purposes but also in techniques for automatically porting programs from sequential consistency memories to relaxed ones (e.g. by fence-insertion techniques [21,32,46,47]). The kind of verification techniques we have discussed are approximations, partial order reduction and heuristic search strategies. While approximations and partial order reduction have been proposed before, as far as we know, the use of directed model checking techniques in the domain of relaxed concurrency is a novelty. However, rather than proposing novel verification techniques, our aim was to provide evidence of the flexibility of Maude for adopting techniques that ease verification and language design task in presence of relaxed consistency. We believe that there are still many developments that can be carried out to provide language designers with a powerful verification-based framework, in particular in what regards the automatization of correctness proofs for the adopted verification techniques.

References

1. Sutter, H.: The free lunch is over: a fundamental turn toward concurrency in software. Dr. Dobbs J. **30**(3), 202–210 (2005)

2. Sutter, H., Larus, J.R.: Software and the concurrency revolution. ACM Queue **3**(7), 54–62 (2005)
3. Serbanuta, T.F., Rosu, G., Meseguer, J.: A rewriting logic approach to operational semantics. Inf. Comput. **207**(2), 305–340 (2009)
4. Lamport, L.: How to make a correct multiprocess program execute correctly on a multiprocessor. IEEE Trans. Comput. **46**(7), 779–782 (1997)
5. Memory consistency models, csc/ece 506 spring 2013/10c ks (2013). http://wiki.expertiza.ncsu.edu/index.php/CSC/ECE_506_Spring_2013/10c_ks
6. Zappa Nardelli, F., Sewell, P., Ševčík, J., Sarkar, S., Owens, S., Maranget, L., Batty, M., Alglave, J.: Relaxed memory models must be rigorous. In: Exploiting Concurrency Efficiently and Correctly, CAV 2009 Workshop, June 2009
7. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM **53**(7), 89–97 (2010)
8. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: Palsberg, J., Abadi, M. (eds.) POPL, pp. 378–391. ACM (2005)
9. Gupta, R., Amarasinghe, S.P. (eds.) Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation. ACM, Tucson, 7–13 June 2008
10. Boudol, G., Petri, G.: Relaxed memory models: an operational approach. In: Shao, Z., Pierce, B.C. (eds.) POPL, pp. 392–403. ACM (2009)
11. Petri, G.: Studying operational models of relaxed concurrency. In: Abadi, M., Lluch Lafuente, A. (eds.) TGC 2013. LNCS, vol. 8358, pp. 254–272. Springer, Heidelberg (2014)
12. Adve, S., Hill, M.D.: A unified formalization of four shared-memory models. IEEE Trans. Parallel Distrib. Syst. **4**, 613–624 (1993)
13. Saraswat, V.A., Jagadeesan, R., Michael, M.M., von Praun, C.: A theory of memory models. In: Yelick, K.A., Mellor-Crummey, J.M. (eds.) PPOPP, pp. 161–172. ACM (2007)
14. Nielson, H.R., Nielson, F.: Semantics with Applications: An Appetizer. Undergraduate Topics in Computer Science. Springer, London (2007)
15. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. Theor. Comput. Sci. **403**(2–3), 239–264 (2008)
16. Lluch Lafuente, A., Meseguer, J., Vandin, A.: State space c-reductions of concurrent systems in rewriting logic. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 430–446. Springer, Heidelberg (2012)
17. Godefroid, P., Wolper, P.: A partial approach to model checking. Inf. Comput. **110**(2), 305–326 (1994)
18. Peled, D.: Combining partial order reductions with on-the-fly model-checking. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 377–390. Springer, Heidelberg (1994)
19. Valmari, A.: A stubborn attack on state explosion. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 156–165. Springer, Heidelberg (1991)
20. Jonsson, B.: State-space exploration for concurrent algorithms under weak memory orderings: (preliminary version). SIGARCH Comput. Archit. News **36**(5), 65–71 (2008)
21. Linden, A., Wolper, P.: A verification-based approach to memory fence insertion in PSO memory systems. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 339–353. Springer, Heidelberg (2013)
22. Farzan, A., Meseguer, J.: Partial order reduction for rewriting semantics of programming languages. Electr. Notes Theor. Comput. Sci. **176**(4), 61–78 (2007)

23. Eker, S., Meseguer, J., Sridharanarayanan, A.: The maude ltl model checker. Electr. Notes Theor. Comput. Sci. **71**, 162–187 (2002)
24. Reffel, F., Edelkamp, S.: Error detection with directed symbolic model checking. In: Wing, J.M., Woodcock, J. (eds.) FM 1999. LNCS, vol. 1708, p. 195. Springer, Heidelberg (1999)
25. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed explicit-state model checking in the validation of communication protocols. STTT **5**(2–3), 247–267 (2004)
26. Groce, A., Visser, W.: Heuristics for model checking java programs. STTT **6**(4), 260–276 (2004)
27. Martí-Oliet, N., Meseguer, J., Verdejo, A.: A rewriting semantics for maude strategies. Electr. Notes Theor. Comput. Sci. **238**(3), 227–247 (2009)
28. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 141–157. Springer, Heidelberg (2013)
29. Clarke, E., Kroning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
30. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: Counter-example guided fence insertion under TSO. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 204–219. Springer, Heidelberg (2012)
31. Kuperstein, M., Vechev, M.T., Yahav, E.: Partial-coherence abstractions for relaxed memory models. In: Hall, M.W., Padua, D.A. (eds.) PLDI, 187–198. ACM (2011)
32. Kuperstein, M., Vechev, M.T., Yahav, E.: Automatic inference of memory fences. In:Bloem, R., Sharygina, N. (eds.) FMCAD, pp. 111–119. IEEE (2010)
33. Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in TSO analysis. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 99–115. Springer, Heidelberg (2011)
34. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems. LNCS, vol. 7792, pp. 512–532. Springer, Heidelberg (2013)
35. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. ACM SIGPLAN Not. **42**(6), 446–455 (2007)
36. Dan, A.M., Meshman, Y., Vechev, M., Yahav, E.: Predicate abstraction for relaxed memory models. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis. LNCS, vol. 7935, pp. 84–104. Springer, Heidelberg (2013)
37. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 107–120. Springer, Heidelberg (2008)
38. Burckhardt, S., Alur, R., Martin, M.M.K.: Bounded model checking of concurrent data types on relaxed memory models: a case study. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 489–502. Springer, Heidelberg (2006)
39. Gopalakrishnan, G.C., Yang, Y., Sivaraj, H.: QB or not QB: an efficient execution verification tool for memory orderings. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 401–413. Springer, Heidelberg (2004)
40. Burnim, J., Sen, K., Stergiou, C.: Sound and complete monitoring of sequential consistency for relaxed memory models. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 11–25. Springer, Heidelberg (2011)

41. Linden, A., Wolper, P.: An automata-based symbolic approach for verifying programs on relaxed memory models. In: van de Pol, J., Weber, M. (eds.) Model Checking Software. LNCS, vol. 6349, pp. 212–226. Springer, Heidelberg (2010)
42. Burckhardt, S., Alur, R., Martin, M.M.K.: Checkfence: checking consistency of concurrent data types on relaxed memory models. In: Ferrante, J., McKinley, K.S. (eds.) PLDI, pp. 12–21. ACM (2007)
43. Liu, F., Nedev, N., Prisadnikov, N., Vechev, M.T., Yahav, E.: Dynamic synthesis for relaxed memory models. In: Vitek, J., Lin, H., Tip, F. (eds.) PLDI, pp. 429–440. ACM (2012)
44. Rosu, G., Serbanuta, T.F.: An overview of the K semantic framework. J. Log. Algebr. Program. **79**(6), 397–434 (2010)
45. Şerbănuţă, T.F.: A Rewriting Approach to Concurrent Programming Language Design and Semantics. Ph.D. Thesis, University of Illinois at Urbana-Champaign, December 2010. https://www.ideals.illinois.edu/handle/2142/18252
46. Linden, A., Wolper, P.: A verification-based approach to memory fence insertion in relaxed memory systems. In: Groce, A., Musuvathi, M. (eds.) SPIN Workshops 2011. LNCS, vol. 6823, pp. 144–160. Springer, Heidelberg (2011)
47. Kuperstein, M., Vechev, M.T., Yahav, E.: Automatic inference of memory fences. SIGACT News **43**(2), 108–123 (2012)