Counter-Example Guided Fence Insertion under TSO

Parosh Aziz Abdulla[1], Mohamed Faouzi Atig[1], Yu-Fang Chen[2], Carl Leonardsson[1], and Ahmed Rezine[3]

[1] Uppsala University, Sweden
[2] Academia Sinica, Taiwan
[3] Linköping University, Sweden

Abstract. We give a *sound* and *complete* fence insertion procedure for concurrent finite-state programs running under the classical TSO memory model. This model allows "write to read" relaxation corresponding to the addition of an unbounded store buffer between each processor and the main memory. We introduce a novel machine model, called the *Single-Buffer* (SB) semantics, and show that the reachability problem for a program under TSO can be reduced to the reachability problem under SB. We present a simple and effective backward reachability analysis algorithm for the latter, and propose a counter-example guided fence insertion procedure. The procedure is augmented by a *placement constraint* that allows the user to choose places inside the program where fences may be inserted. For a given placement constraint, we automatically infer all minimal sets of fences that ensure correctness. We have implemented a prototype and run it successfully on all standard benchmarks together with several challenging examples that are beyond the applicability of existing methods.

1 Introduction

Modern concurrent process architectures allow *weak* (*relaxed*) memory models, in which certain memory operations may overtake each other. The use of weak memory models makes reasoning about the behaviors of concurrent programs much more difficult and error-prone compared to the classical Sequentially Consistent (SC) memory model. In fact, several algorithms that are designed for the synchronization of concurrent processes, such as mutual exclusion and producer-consumer protocols, are not correct when run on weak memories [2]. One way to eliminate the non-desired behaviors resulting from the use of weak memory models is to insert memory *fence* instructions in the program code. In this work, a fence instruction forbids reordering between instructions issued by the same process. It does not allow any operation issued after the fence instruction to overtake an operation issued before it. Hence, a naive approach to correct a program running under a weak memory model is to insert a fence instruction after every operation. Adopting this approach results in significant performance degradation [13]. Therefore, it is important to optimize fence placement. A natural criterion is to provide *minimal* sets of fences whose insertion is sufficient for ensuring program correctness under the considered weak memory model (provided correctness under SC).

One of the most common relaxations corresponds to TSO (Total Store Ordering) that is adopted by Sun's SPARC multiprocessors. TSO is the kernel of many common

weak memory models [28, 31], and is the latest formalization of the x86 memory model. In TSO, read operations are allowed to overtake write operations of the same process if they concern different variables. In this paper, we use the usual formal model of TSO, developed in e.g. [28, 30], and assume it gives a faithful description of the actual hardware on which we run our programs. This model adds an unbounded FIFO buffer between each process and the main memory.

Our approach We present a sound and complete method for checking safety properties and for inserting fences in finite-state programs running on the TSO model. The procedure is parameterized by a fence placement constraint that allows to restrict the places inside the program where fences may be inserted. To cope with the unbounded store buffers in the case of TSO, we present a new semantics, called the *Single-Buffer (SB)* semantics, in which all the processes share one (unbounded) buffer. We show that the SB semantics is equivalent to the operational model of TSO (as defined in [30]). A crucial feature of the SB semantics is that it permits a natural ordering on the (infinite) set of configurations, and that the induced transition relation is monotonic wrt. this ordering. This allows to use general frameworks for *well quasi-ordered systems* [1] in order to derive verification algorithms for programs running on the SB model. In case the program fails to satisfy the specification with the current set of fences, our algorithm provides counter-examples (traces) that can be used to increase the set of fences in a systematic manner. Thus, we get a counter-example guided procedure for refining the sets of fences. We prove termination of the obtained procedure. Since each refinement step is performed based on an exact reachability analysis algorithm, the procedure will eventually return all minimal sets of fences (wrt. the given placement constraint) that ensure correctness of the program. Although we instantiate our framework to the case of TSO, the method can be extended to other memory models such as the PSO model.

Contribution We present the first *sound* and *complete* procedure for fence insertion for programs under TSO. The main ingredients of the framework are the following: (i) A new semantical model, the so called SB model, that allows efficient infinite state model checking. (ii) A simple and effective backward analysis algorithm for solving the reachability problem under the SB semantics. (iii) The algorithm uses finite-state automata as a symbolic representation for infinite sets of configurations, and returns a symbolic counter-example in case the program violates its specification. (iv) A counter-example guided procedure that automatically infers all minimal sets of fences sufficient for correctness under a given fence placement policy. (v) Based on the algorithm, we have implemented a prototype, and run it successfully on several challenging concurrent programs, including some that cannot be handled by existing methods.

Proofs, implementation details and experimental results are in the appendix.

Related Work To our knowledge, our approach is the first sound and complete automatic fence insertion method that discovers all minimal sets of fences for finite-state concurrent programs running under TSO. Since we are dealing with infinite-state verification, it is hard to provide methods that are both automatic and that return exact solutions. Existing approaches avoid solving the general problem by considering *under-approximations*, *over-approximations*, *restricted* classes of programs, *forbidding*

sequential inconsistent behavior, or by proposing exact algorithms for which termination is *not* guaranteed. Under-approximations of the program behavior can be achieved through testing [9], bounded model checking [7, 6], or by restricting the behavior of the program, e.g., through bounding the sizes of the buffers [18] or the number of switches [5]. Such techniques are useful in practice for finding errors. However, they are not able to check all possible traces and can therefore not tell whether the generated set of fences is sufficient for correctness. Recent techniques based on over-approximations [19] are valuable for showing correctness; however they are not complete and might not be able to prove correctness although the program satisfies its specification. Hence, the computed set of fences need not be minimal. Examples of restricted classes of programs include those that are free from different types of data races [27]. Considering only data-race free programs can be unrealistic since data races are very common in efficient implementations of concurrent algorithms. Another approach is to use monitors [3, 8, 10], compiler techniques [12], and explicit state model checking [16] to insert fences in order to remove all non-sequential consistent behaviors even if these will not violate the desired correctness properties. As a result, this approach cannot guarantee to generate minimal sets of fences to make programs correct because they also remove benign sequentially inconsistent behaviors. The method of [23] performs an exact search of the state space, combined with fixpoint acceleration techniques, to deal with the potentially infinite state space. However, in general, the approach does not guarantee termination. State reachability for TSO is shown to be non primitive recursive in [4] by reductions to/from lossy channel systems. The reductions involve nondeterministically guessing buffer contents, which introduces a serious state space explosion problem. The approach does not discuss fence insertion and cannot even verify the simplest examples. An important contribution of our work is the introduction of a single buffer semantics for avoiding the immediate state space explosion. In contrast to the above approaches, our method is efficient and performs *exact* analysis of the program on the given memory model. We show termination of the analysis. As a consequence, we are able to compute all *minimal* sets of fences required for correctness of the program.

2 Preliminaries

In this section we first introduce notations that we use throughout the paper, and then define a model for concurrent systems.

Notation We use $\mathbb{N}$ to denote the set of natural numbers. For sets A and B, we use $[A \mapsto B]$ to denote the set of all total functions from A to B and $f : A \mapsto B$ to denote that f is a total function that maps A to B. For $a \in A$ and $b \in B$, we use $f[a \hookleftarrow b]$ to denote the function f' defined as follows: $f'(a) = b$ and $f'(a') = f(a')$ for all $a' \neq a$.

Let Σ be a finite alphabet. We denote by Σ^* (resp. Σ^+) the set of all *words* (resp. non-empty words) over Σ, and by ε the empty word. The length of a word $w \in \Sigma^*$ is denoted by $|w|$; we assume that $|\varepsilon| = 0$. For every $i : 1 \leq i \leq |w|$, let $w(i)$ be the symbol at position i in w. For $a \in \Sigma$, we write $a \in w$ if a appears in w, i.e., $a = w(i)$ for some $i : 1 \leq i \leq |w|$. For words w_1, w_2, we use $w_1 \cdot w_2$ to denote the concatenation of w_1 and w_2. For a word $w \neq \varepsilon$ and $i : 0 \leq i \leq |w|$, we define $w \odot i$ to be the suffix of w we get by deleting the prefix of length i, i.e., the unique w_2 such that $w = w_1 \cdot w_2$ and $|w_1| = i$.

A transition system $\mathcal{T}$ is a triple $(\mathtt{C}, \mathtt{Init}, \rightarrow)$ where $\mathtt{C}$ is a (potentially infinite) set of *configurations*, $\mathtt{Init} \subseteq \mathtt{C}$ is the set of *initial configurations*, and $\rightarrow \subseteq \mathtt{C} \times \mathtt{C}$ is the *transition relation*. We write $c \rightarrow c'$ to denote that $(c, c') \in \rightarrow$, and $\xrightarrow{*}$ to denote the reflexive transitive closure of $\rightarrow$. A configuration c is said to be *reachable* if $c_0 \xrightarrow{*} c$ for some $c_0 \in \mathtt{Init}$; and a set C of configurations is said to be *reachable* if some $c \in C$ is reachable. A *run* π of $\mathcal{T}$ is of the form $c_0 \rightarrow c_1 \rightarrow \cdots \rightarrow c_n$, where $c_i \rightarrow c_{i+1}$ for all $i : 0 \leq i < n$. Then, we write $c_0 \xrightarrow{\pi} c_n$. We use *target* (π) to denote the configuration c_n. Notice that, for configurations c, c', we have that $c \xrightarrow{*} c'$ iff $c \xrightarrow{\pi} c'$ for some run π. The run π is said to be a *computation* if $c_0 \in \mathtt{Init}$. Two runs $\pi_1 = c_0 \rightarrow c_1 \rightarrow \cdots \rightarrow c_m$ and $\pi_2 = c_{m+1} \rightarrow c_{m+2} \rightarrow \cdots \rightarrow c_n$ are said to be *compatible* if $c_m = c_{m+1}$. Then, we write $\pi_1 \bullet \pi_2$ to denote the run $\pi = c_0 \rightarrow c_1 \rightarrow \cdots \rightarrow c_m \rightarrow c_{m+2} \rightarrow \cdots \rightarrow c_n$. Given an ordering $\sqsubseteq$ on $\mathtt{C}$, we say that $\rightarrow$ is *monotonic* wrt. $\sqsubseteq$ if whenever $c_1 \rightarrow c_1'$ and $c_1 \sqsubseteq c_2$, there exists a c_2' s.t. $c_2 \xrightarrow{*} c_2'$ and $c_1' \sqsubseteq c_2'$. We say that $\rightarrow$ is *effectively monotonic* wrt. $\sqsubseteq$ if, given configurations c_1, c_1', c_2 as above, we can compute c_2' and a run π s.t. $c_2 \xrightarrow{\pi} c_2'$.

Concurrent Programs We define *concurrent programs*, a model for representing shared-memory concurrent processes. A concurrent program $\mathcal{P}$ has a finite number of finite-state processes (threads), each with its own program code. Communication between processes is performed through a shared-memory that consists of a fixed number of shared variables (finite domains) to which all threads can read and write.

We assume a finite set X of *variables* ranging over a finite data domain V. A *concurrent program* is a pair $\mathcal{P} = (P, A)$ where P is a finite set of *processes* and $A = \{A_p \mid p \in P\}$ is a set of extended finite-state automata (one automaton A_p for each process $p \in P$). The automaton A_p is a triple $\left(Q_p, q_p^{init}, \Delta_p\right)$ where Q_p is a finite set of *local states*, $q_p^{init} \in Q_p$ is the *initial* local state, and Δ_p is a finite set of *transitions*. Each transition is a triple (q, op, q') where $q, q' \in Q_p$ and op is an *operation*. An operation is of one of the following five forms: (1) *"no operation"* nop, (2) *read operation* $\mathsf{r}(x, v)$, (3) *write operation* $\mathsf{w}(x, v)$, (4) *fence operation* fence, and (5) *atomic read-write operation* $\mathsf{arw}(x, v, v')$, where $x \in X$, and $v, v' \in V$. For a transition $t = (q, op, q')$, we use *source* (t), *operation* (t), and *target* (t) to denote q, op, and q' respectively. We define $Q := \cup_{p \in P} Q_p$ and $\Delta := \cup_{p \in P} \Delta_p$. A *local state definition* $\underline{q}$ is a mapping $P \mapsto Q$ such that $\underline{q}(p) \in Q_p$ for each $p \in P$.

3 TSO Semantics

We describe the TSO model formalized in [28, 30]. Conceptually, the model adds a FIFO buffer between each process and the main memory. The buffer is used to store the write operations performed by the process. Thus, a process executing a write instruction inserts it into its store buffer and immediately continues executing subsequent instructions. Memory updates are then performed by non-deterministically choosing a process and by executing the first write operation in its buffer (the left-most element in the buffer). A read operation by a process p on a variable x can overtake some write operations stored in its own buffer if all these operations concern variables that are different from x. Thus, if the buffer contains some write operations to x, then the read value must correspond to the value of the most recent write operation to x. Otherwise, the

value is fetched from the memory. A fence means that the buffer of the process must be flushed before the program can continue beyond the fence. Notice that the store buffers of the processes are *unbounded* since there is *a priori* no limit on the number of write operations that can be issued by a process before a memory update occurs. Below we define the transition system induced by a program running under the TSO semantics. To do that, we define the set of configurations and transition relation. We fix a concurrent program $\mathcal{P} = (P, A)$.

Formal Semantics A *TSO-configuration* c is a triple $(\underline{q}, \underline{b}, mem)$ where $\underline{q}$ is a local state definition, $\underline{b} : P \mapsto (X \times V)^*$, and $mem : X \mapsto V$. Intuitively, $\underline{q}(p)$ gives the local state of process p. The value of $\underline{b}(p)$ is the content of the buffer belonging to p. This buffer contains a sequence of write operations, where each write operation is defined by a pair, namely a variable x and a value v that is assigned to x. In our model, messages will be appended to the buffer from the right, and fetched from the left. Finally, *mem* defines the value of each variable in the memory. We use $\mathtt{C}_{TSO}$ to denote the set of TSO-configurations. We define the transition relation $\rightarrow_{TSO}$ on $\mathtt{C}_{TSO}$. The relation is induced by (1) members of Δ; and (2) a set $\Delta' := \{\mathtt{update}_p \mid p \in P\}$ where $\mathtt{update}_p$ is an operation that updates the memory using the first message in the buffer of process p. For configurations $c = (\underline{q}, \underline{b}, mem)$, $c' = (\underline{q}', \underline{b}', mem')$, a process $p \in P$, and $t \in \Delta_p \cup \{\mathtt{update}_p\}$, we write $c \xrightarrow{t}_{TSO} c'$ to denote that one of the following conditions is satisfied:

- Nop: $t = (q, \mathsf{nop}, q')$, $\underline{q}(p) = q$, $\underline{q}' = \underline{q}[p \hookleftarrow q']$, $\underline{b}' = \underline{b}$, and $mem' = mem$. The process changes its local state while buffer and memory contents remain unchanged.
- Write to store: $t = (q, \mathsf{w}(x,v), q')$, $\underline{q}(p) = q$, $\underline{q}' = \underline{q}[p \hookleftarrow q']$, $\underline{b}' = \underline{b}[p \hookleftarrow \underline{b}(p) \cdot (x,v)]$, and $mem' = mem$. The write operation is appended to the tail of the buffer.
- Update: $t = \mathtt{update}_p$, $\underline{q}' = \underline{q}$, $\underline{b} = \underline{b}' \left[p \hookleftarrow (x,v) \cdot \underline{b}'(p)\right]$, and $mem' = mem[x \hookleftarrow v]$. The write in the head of the buffer is removed and memory is updated accordingly.
- Read: $t = (q, \mathsf{r}(x,v), q')$, $\underline{q}(p) = q$, $\underline{q}' = \underline{q}[p \hookleftarrow q']$, $\underline{b}' = \underline{b}$, $mem' = mem$, and one of the following two conditions is satisfied:
  - Read own write: There is an $i : 1 \leq i \leq |\underline{b}(p)|$ such that $\underline{b}(p)(i) = (x,v)$, and $(x,v') \notin (\underline{b}(p) \odot i)$ for all $v' \in V$. If there is a write operation on x in the buffer of p then we consider the most recent of such a write operation (the right-most one in the buffer). This operation should assign v to x.
  - Read memory: $(x,v') \notin \underline{b}(p)$ for all $v' \in V$ and $mem(x) = v$. If there is no write operation on x in the buffer of p then the value v of x is fetched from memory.
- Fence: $t = (q, \mathsf{fence}, q')$, $\underline{q}(p) = q$, $\underline{q}' = \underline{q}[p \hookleftarrow q']$, $\underline{b}(p) = \varepsilon$, $\underline{b}' = \underline{b}$, and $mem' = mem$. A fence operation may be performed by a process only if its buffer is empty.
- ARW: $t = (q, \mathsf{arw}(x,v,v'), q')$, $\underline{q}(p) = q$, $\underline{q}' = \underline{q}[p \hookleftarrow q']$, $\underline{b}(p) = \varepsilon$, $\underline{b}' = \underline{b}$, $mem(x) = v$, and $mem' = mem[x \hookleftarrow v']$. The ARW operation corresponds to an atomic compare and swap (or test and set). It can be performed by a process only if its buffer is empty. The operation checks whether the value of x is v. In such a case, it changes its value to v'.

We use $c \rightarrow_{TSO} c'$ to denote that $c \xrightarrow{t}_{TSO} c'$ for some $t \in \Delta \cup \Delta'$. The set $\mathtt{Init}_{TSO}$ of *initial* TSO-configurations contains all configurations of the form $(\underline{q_{init}}, \underline{b_{init}}, mem_{init})$ where,

for all $p \in P$, we have that $\underline{q_{init}}(p) = q_p^{init}$ and $\underline{b_{init}}(p) = \varepsilon$. In other words, each process is in its initial local state and all the buffers are empty. On the other hand, the memory may have any initial value. The transition system induced by a concurrent system under the TSO semantics is then given by $(\mathtt{C}_{TSO}, \mathtt{Init}_{TSO}, \rightarrow_{TSO})$.

The TSO Reachability Problem Given a set `Target` of local state definitions, we use $\mathit{Reachable}(\mathit{TSO})(\mathcal{P})(\mathtt{Target})$ to be a predicate that indicates the reachability of the set $\{(\underline{q}, \underline{b}, \mathit{mem}) \mid \underline{q} \in \mathtt{Target}\}$, i.e., whether a configuration c, where the local state definition of c belongs to `Target`, is reachable. The reachability problem for TSO is to check, for a given `Target`, whether $\mathit{Reachable}(\mathit{TSO})(\mathcal{P})(\mathtt{Target})$ holds or not. Using standard techniques we can reduce checking safety properties to the reachability problem. More precisely, `Target` denotes "bad configurations" that we do not want to occur during the execution of the system. For instance, for mutual exclusion protocols, the bad configurations are those where the local states of two processes are both in the critical sections. We say that the "program is correct" to indicate that `Target` is not reachable.

4 Single-Buffer Semantics

The formal model of TSO [28, 30] is quite powerful since it uses *unbounded perfect* buffers. However, the reachability problem remains decidable [4]. Our goal is to exploit this to design a practically efficient verification algorithm. To do that, we introduce a new semantics model, called the *Single-Buffer (SB)* model that weaves the buffers of all processes into one unified buffer. The SB model satisfies two important properties (1) it is equivalent to the TSO semantics wrt. reachability, i.e., `Target` is reachable in the TSO semantics iff it is reachable in the SB semantics; (2) the induced transition system is "monotonic" wrt. some pre-order (on configurations) so that the classical infinite state model checking framework of [1] can be applied. Fix a concurrent system $\mathcal{P} = (P, A)$.

Formal Semantics A *SB-configuration* c is a triple $(\underline{q}, b, \underline{z})$ where $\underline{q}$ is (as in the case of TSO-semantics) a local state definition, $b \in ([X \mapsto V] \times P \times X)^+$, and $\underline{z} : P \mapsto \mathbb{N}$. Intuitively, the (only) buffer contains triples of the form (mem, p, x) where mem defines variable values (encoding a memory snapshot), x is the latest variable that has been written into, and p is the process that performed the write operation. Furthermore, $\underline{z}$ represents a set of *pointers* (one per process) where, from the point of view of p, the word $b \odot \underline{z}(p)$ is the sequence of write operations that have not yet been used for memory updates and the first element of the triple $b(\underline{z}(p))$ represents the memory content. As we shall see below, the buffer will never be empty, since it is not empty in an initial configuration, and since no messages are ever removed from it during a run of the system (in the SB semantics, the update operation moves a pointer to the right instead of removing a message from the buffer). This implies (among other things) that the invariant $\underline{z}(p) > 0$ is always maintained. We use $\mathtt{C}_{SB}$ to denote the set of SB-configurations.

Let $c = (\underline{q}, b, \underline{z})$ be an SB-configuration. For every $p \in P$ and $x \in X$, we use $\mathtt{LastWrite}(c, p, x)$ to denote the index of the most recent buffer message where p writes to x or the current memory of p if the aforementioned type of message does not exist in the buffer from the point of view of p. Formally, $\mathtt{LastWrite}(c, p, x)$ is the largest index i such that $i = \underline{z}(p)$ or $b(i) = (\mathit{mem}, p, x)$ for some mem.

We define the transition relation $\rightarrow_{SB}$ on the set of SB-configurations as follows. In a similar manner to the case of TSO, the relation is induced by members of $\Delta \cup \Delta'$. For configurations $c = (\underline{q}, b, \underline{z})$, $c' = (\underline{q}', b', \underline{z}')$, and $t \in \Delta_p \cup \{\mathtt{update}_p\}$, we write $c \xrightarrow{t}_{SB} c'$ to denote that one of the following conditions is satisfied:

- Nop: $t = (q, \mathsf{nop}, q')$, $\underline{q}(p) = q$, $\underline{q}' = \underline{q}[p \hookleftarrow q']$, $b' = b$ and $\underline{z}' = \underline{z}$. The operation changes only the local state of p.
- Write to store: $t = (q, \mathsf{w}(x,v), q')$, $\underline{q}(p) = q$, $\underline{q}' = \underline{q}[p \hookleftarrow q']$, $b(|b|)$ is of the form (mem_1, p_1, x_1), $b' = b \cdot (mem_1[x \hookleftarrow v], p, x)$, and $\underline{z}' = \underline{z}$. A new element is appended to the tail of the buffer. Values of variables in the new element are identical to those in the previous last element except that the value of x has been updated to v. Furthermore, we include the updating process p and the updated variable x.
- Update: $t = \mathtt{update}_p$, $\underline{q}' = \underline{q}$, $b' = b$, $\underline{z}(p) < |b|$ and $\underline{z}' = \underline{z}[p \hookleftarrow \underline{z}(p) + 1]$. An update operation (as seen by p) is simulated by moving the pointer of p one step to the right. This means that we remove the oldest write operation that is yet to be used for a memory update. The removed element will now represent the memory contents from the point of view of p.
- Read: $t = (q, \mathsf{r}(x,v), q')$, $\underline{q}(p) = q$, $\underline{q}' = \underline{q}[p \hookleftarrow q']$, $b' = b$, and $b(\mathtt{LastWrite}(c, p, x)) = (mem_1, p_1, x_1)$ for some mem_1, p_1, x_1 with $mem_1(x) = v$.
- Fence: $t = (q, \mathsf{fence}, q')$, $\underline{q}(p) = q$, $\underline{q}' = \underline{q}[p \hookleftarrow q']$, $\underline{z}(p) = |b|$, $b' = b$, and $\underline{z}' = \underline{z}$. The buffer should be empty from the point of view of p when the transition is performed. This is encoded by the equality $\underline{z}(p) = |b|$.
- ARW: $t = (q, \mathsf{arw}(x,v,v'), q')$, $\underline{q}(p) = q$, $\underline{q}' = \underline{q}[p \hookleftarrow q']$, $\underline{z}(p) = |b|$, $b(|b|)$ is of the form (mem_1, p_1, x_1), $mem_1(x) = v$, $b' = b \cdot (mem_1[x \hookleftarrow v'], p, x)$, and $\underline{z}' = \underline{z}[p \hookleftarrow \underline{z}(p) + 1]$. The fact that the buffer is empty from the point of view of p is encoded by the equality $\underline{z}(p) = |b|$. The content of the memory can then be fetched from the right-most element $b(|b|)$ in the buffer. To encode that the buffer is still empty after the operation (from the point of view of p) the pointer of p is moved one step to the right.

We use $c \rightarrow_{SB} c'$ to denote that $c \xrightarrow{t}_{SB} c'$ for some $t \in \Delta \cup \Delta'$. The set $\mathtt{Init}_{SB}$ of *initial* SB-configurations contains all configurations of the form $(\underline{q_{init}}, b_{init}, \underline{z_{init}})$ where $|b_{init}| = 1$, and for all $p \in P$, we have that $\underline{q_{init}}(p) = q_p^{init}$, and $\underline{z_{init}}(p) = 1$. In other words, each process is in its initial local state. The buffer contains a single message, say of the form $(mem_{init}, p_{init}, x_{init})$, where mem_{init} represents the initial value of the memory. The memory may have any initial value. Also, the values of p_{init} and x_{init} are not relevant since they will not be used in the computations of the system. The pointers of all the processes point to the first position in the buffer. According to our encoding, this indicates that their buffers are all empty. The transition system induced by a concurrent system under the SB semantics is then given by $(\mathtt{C}_{SB}, \mathtt{Init}_{SB}, \rightarrow_{SB})$.

The SB Reachability Problem We define the predicate $Reachable(SB)(\mathcal{P})(\mathtt{Target})$, and the reachability problem for the SB semantics, in a similar manner to TSO. The following theorem states equivalence of the reachability problems under TSO and SB semantics. Due to its technicality and lack of space, we leave the proof for the appendix.

Theorem 1. *For a concurrent program $\mathcal{P}$ and a local state definition* `Target`*, the reachability problems are equivalent under the TSO and SB semantics.*

5 The SB Reachability Algorithm

In this section, we present an algorithm for checking reachability of an (infinite) set of configurations characterized by a (finite) set `Target` of local state definitions. In addition to answering the reachability question, the algorithm also provides an "error trace" in case `Target` is reachable. First, we define an ordering $\sqsubseteq$ on the set of SB-configurations, and show that it satisfies two important properties, namely (i) it is a well quasi-ordering (wqo), i.e., for every infinite sequence $c_0, c_1, \ldots$ of SB-configurations, there are $i < j$ with $c_i \sqsubseteq c_j$; and (ii) the SB-transition relation $\rightarrow_{SB}$ is monotonic wrt. $\sqsubseteq$. The algorithm performs backward reachability analysis from the set of configurations with local state definitions that belong to `Target`. During each step of the search procedure, the algorithm takes the upward closure (wrt. $\sqsubseteq$) of the generated set of configurations. By monotonicity of $\sqsubseteq$ it follows that taking the upward closure preserves exactness of the analysis [1]. From the fact that we always work with upward closed sets and that $\sqsubseteq$ is a wqo it follows that the algorithm is guaranteed to terminate [1]. In the algorithm, we use a variant of finite-state automata, called *SB-automata*, to encode (potentially infinite) sets of SB-configurations.

Ordering For an SB-configuration $c = (\underline{q}, b, \underline{z})$ we define $\texttt{ActiveIndex}(c) := \min\{\underline{z}(p) \mid p \in P\}$. In other words, the part of b to the right of (and including) $\texttt{ActiveIndex}(c)$ is "active", while the part to the left is "dead" in the sense that all its content has already been used for memory updates. The left part is therefore not relevant for computations starting from c.

Let $c = (\underline{q}, b, \underline{z})$ and $c' = (\underline{q}', b', \underline{z}')$ be two SB-configurations. Define $j := \texttt{ActiveIndex}(c)$ and $j' := \texttt{ActiveIndex}(c')$. We write $c \sqsubseteq c'$ to denote that (i) $\underline{q} = \underline{q}'$ and that (ii) there is an injection $g : \{j, j+1, \ldots, |b|\} \mapsto \{j', j'+1, \ldots, |b'|\}$ such that the following conditions are satisfied. For every $i, i_1, i_2 \in \{j, \ldots, |b|\}$, (1) $i_1 < i_2$ implies $g(i_1) < g(i_2)$, (2) $b(i) = b'(g(i))$, (3) $\texttt{LastWrite}(c', p, x) = g(\texttt{LastWrite}(c, p, x))$ for all $p \in P$ and $x \in X$, and (4) $\underline{z}'(p) = g(\underline{z}(p))$ for all $p \in P$. The first condition means that g is strictly monotonic. The second condition corresponds to that the *active* part of b is a *sub-word* of the *active* part of b'. The third condition ensures the last write indices wrt. all processes and variables are consistent. The last condition ensures each process points to identical elements in b and b'.

We get the following lemma from the fact that (i) the sub-word relation is a well-quasi ordering on finite words [15], and that (ii) the number of states and messages (associated with last write operations and pointers) that should be equal, is finite.

Lemma 1. *The relation $\sqsubseteq$ is a well-quasi ordering on SB-configurations.*

The following lemma shows effective monotonicity (cf. Section 2) of the SB-transition relation wrt. $\sqsubseteq$. As we shall see below, this allows the reachability algorithm to only work with upward closed sets. Monotonicity is used in the termination of the reachability algorithm. The effectiveness aspect is used in the fence insertion algorithm (cf. Section 6).

Lemma 2. *$\rightarrow_{SB}$ is effectively monotonic wrt. $\sqsubseteq$.*

The *upward closure* of a set C is defined as $C{\uparrow} := \{c' \mid \exists c \in C, c \sqsubseteq c'\}$. A set C is *upward closed* if $C = C{\uparrow}$.

SB-Automata First we introduce an alphabet $\Sigma := ([X \mapsto V] \times P \times X) \times 2^P$. Each element $((mem, p, x), P') \in \Sigma$ represents a single position in the buffer of an SB-configuration. More precisely, the triple (mem, p, x) represents the message stored at that position and the set $P' \subseteq P$ gives the (possibly empty) set of processes whose pointers point to the given position. Consider a word $w = a_1 a_2 \cdots a_n \in \Sigma^*$, where a_i is of the form $((mem_i, p_i, x_i), P_i)$. We say that w is proper if, for each process $p \in P$, there is exactly one $i : 1 \leq i \leq n$ with $p \in P_i$. In other words, the pointer of each process is uniquely mapped to one position in w. A proper word w of the above form can be "decoded" into a (unique) pair $decoding(w) := (b, \underline{z})$, defined by (i) $|b| = n$, (ii) $b(i) = (mem_i, p_i, x_i)$ for all $i : 1 \leq i \leq n$, and (iii) $\underline{z}(p)$ is the unique integer $i : 1 \leq i \leq n$ such that $p \in P_i$ (the value of i is well-defined since w is proper). We extend the function to sets of words where $decoding(W) := \{decoding(w) \mid w \in W\}$.

An SB-automaton A is a tuple $\left(S, \Delta, S^{final}, h\right)$ where S is a finite set of *states*, $\Delta \subseteq S \times \Sigma \times S$ is a finite set of transitions, $S^{final} \subseteq S$ is the set of *final* states, and $h : (P \mapsto Q) \mapsto S$. The total function h defines a labeling of the states of A by the local state definitions of the concurrent program $\mathcal{P}$, such that each $\underline{q}$ is mapped to a state $h(\underline{q})$ in A. For a state $s \in S$, we define $L(A, s)$ to be the set of words of the form $w = a_1 a_2 \cdots a_n$ such that there are states $s_0, s_1, \ldots, s_n \in S$ satisfying the following conditions: (i) $s_0 = s$, (ii) $(s_i, a_{i+1}, s_{i+1}) \in \Delta$ for all $i : 0 \leq i < n$, (iii) $s_n \in S^{final}$, and (iv) w is proper. We define the *language* of A by $L(A) := \left\{ \left(\underline{q}, b, \underline{z}\right) \mid (b, \underline{z}) \in decoding\left(L\left(A, h(\underline{q})\right)\right) \right\}$. Thus, the language $L(A)$ characterizes a set of SB-configurations. More precisely, the configuration $\left(\underline{q}, b, \underline{z}\right)$ belongs to $L(A)$ if $(b, \underline{z})$ is the decoding of a word that is accepted by A when A is started from the state $h(\underline{q})$ (the state labeled by $\underline{q}$). A set C of SB-configurations is said to be *regular* if $C = L(A)$ for some SB-automaton A.

Operations on SB-Automata We show that we can compute the operations (union, intersection, test emptiness, compute predecessor, etc.) needed for the reachability algorithm. First, observe that regular sets of SB-configurations are closed under union and intersection. For SB-automata A_1, A_2, we use $A_1 \cap A_2$ to denote an automaton A such that $L(A) = L(A_1) \cap L(A_2)$. We define $A_1 \cup A_2$ in a similar manner. We use $A^{\emptyset}$ to denote an (arbitrary) automaton whose language is empty. We can construct SB-automata for the set of initial SB-configurations, and for sets of SB-configurations characterized by local state definitions.

Lemma 3. *We can compute an SB-automaton A^{init} such that $L\left(A^{init}\right) = \mathtt{Init}_{SB}$. For a set* `Target` *of local state definitions, we can compute an SB-automaton $A^{final}(\mathtt{Target})$ such that $L\left(A^{final}(\mathtt{Target})\right) := \left\{ \left(\underline{q}, b, \underline{z}\right) \mid \underline{q} \in \mathtt{Target} \right\}$.*

The following lemma tells us that regularity of a set is preserved by taking upward closure, and that we in fact can compute an automaton describing its upward closure.

Lemma 4. *For an SB-automaton A we can compute an SB-automaton $A{\uparrow}$ such that $L(A{\uparrow}) = L(A){\uparrow}$.*

We define the *predecessor function* as follows. Let $t \in \Delta \cup \Delta'$ and let C be a set of SB-configurations. We define $\mathtt{Pre}_t(C) := \{c \mid \exists c' \in C, c \xrightarrow{t}_{SB} c'\}$ to denote the set of immediate predecessor configurations of C w.r.t. the transition t. In other words,

$\texttt{Pre}_t(C)$ is the set of configurations that can reach a configuration in C through a single execution of t. The following lemma shows that Pre preserves regularity, and that in fact we can compute the automaton of the predecessor set.

Lemma 5. *For a transition t and an SB-automaton A, we can compute an SB-automaton $\texttt{Pre}_t(A)$ such that $L(\texttt{Pre}_t(A)) = \texttt{Pre}_t(L(A))$.*

Reachability Algorithm The algorithm performs a symbolic backward reachability analysis [1], where we use SB-automata for representing infinite sets of SB-configurations. In fact, the algorithm also provides *traces* that we will use to find places inside the code where to insert fences (see Section 6). For a set Target of local state definitions, a *trace* δ to Target is a sequence of the form $A_0 t_1 A_1 t_2 \cdots t_n A_n$ where $A_0, A_1, \ldots, A_n$ are SB-automata, $t_1, \ldots, t_n$ are transitions, and (i) $L(A_0) \cap \texttt{Init}_{SB} \neq \emptyset$; (ii) $A_i = \left(\texttt{Pre}_{t_{i+1}}(A_{i+1})\right)\uparrow$ for all $i : 0 \leq i < n$ (even if $L(A_{i+1})$ is upward-closed, it is still possible that $L\left(\texttt{Pre}_{t_{i+1}}(A_{i+1})\right)$ is not upward-closed; however due to monotonicity taking upward closure does not affect exactness of the analysis); and (iii) $A_n = A^{final}(\texttt{Target})$. In the following, we use $head(\delta)$ to denote the SB-automaton A_0. The algorithm inputs a finite set Target, and checks the predicate $Reachable(SB)(\mathcal{P})(\texttt{Target})$. If the predicate does not hold then Algorithm 1 simply answers *unreachable*; otherwise, it returns a *trace*. It maintains a *working* set $\mathcal{W}$ that contains a set of traces. Intuitively, in a trace $A_0 t_1 A_1 t_2 \cdots t_n A_n \in \mathcal{W}$, the automaton A_0 has been "detected" but not yet "analyzed", while the rest of the trace represents a sequence of transitions and SB-automata that has led to the generation of A_0. The algorithm also maintains an automaton $A^{\mathcal{V}}$ that encodes configurations that have already been analyzed.

Algorithm 1: Reachability

input : A concurrent program $\mathcal{P}$ and a finite set Target of local state definitions.
output: *"unreachable"* if $\neg Reachable(SB)(\mathcal{P})(\texttt{Target})$ holds. A trace to Target otherwise.

```
1  W ← {A^final(Target)};
2  A^V ← A^∅;
3  while W ≠ ∅ do
4      Pick and remove a trace δ from W;
5      A ← head(δ);
6      if L(A ∩ A^init) ≠ ∅ then return δ;
7      if L(A) ⊆ L(A^V) then discard A;
8      else
9          W ← {δ' ∈ W | L(head(δ')) ⊈ L(A)} ∪
               {(Pre_t(A))↑ · t · δ | t ∈ Δ ∪ Δ'};
10         A^V ← A^V ∪ A
11 return "unreachable";
```

Initially, $A^{\mathcal{V}}$ is an automaton recognizing the empty language, and $\mathcal{W}$ is the singleton $\{A^{final}(\texttt{Target})\}$. In other words, we start with a single trace containing the automaton representing configurations induced by Target (can be constructed by Lemma 3). At the beginning of each iteration, the algorithm picks and removes a trace δ (with head A) from the set $\mathcal{W}$. First it checks whether A intersects with A^{init} (can be constructed by Lemma 3). If yes, it returns the trace δ. If not, it checks whether A is covered by $A^{\mathcal{V}}$ (i.e., $L(A) \subseteq L\left(A^{\mathcal{V}}\right)$). If *yes* then A does not carry any new information and it (together with its trace) can be safely discarded. Otherwise, the algorithm performs the following operations: (i) it discards all elements of $\mathcal{W}$ that are covered by A; (ii) it adds A to $A^{\mathcal{V}}$; and (iii) for each transition t it adds a trace $A_1 \cdot t \cdot \delta$ to $\mathcal{W}$, where we

compute A_1 by taking the predecessor $\mathtt{Pre}_t(A)$ of A wrt. t, and then taking the upward closure (Lemmata 4 and 5). Notice that since we take the upward closure of the generated automata, and since $A^{final}(\mathtt{Target})$ accepts an upward closed set, then $A^{\mathcal{V}}$ and all the automata added to $\mathcal{W}$ accept upward closed sets. The algorithm terminates when $\mathcal{W}$ becomes empty.

Theorem 2. *The reachability algorithm always terminates with the correct answer.*

6 Fence Insertion

Our fence insertion algorithm is parameterized by a predefined *placement constraint* G where $G \subseteq Q$. The algorithm will place fences only after local states that belong to G. This gives the user the freedom to choose between the efficiency of the verification algorithm and the number of fences that are needed to ensure correctness of the program. The weakest placement constraint is defined by taking G to be the set of all local states of the processes, which means that a fence might be placed anywhere inside the program. On the other hand, one might want to place fences only after write operations, place them only before read operations, or avoid putting them within certain loops (e.g., loops that are known to be executed often during the runs of the program). For any given G, the algorithm finds the minimal sets of fences (if any) that are sufficient for correctness. First, we show how to use a trace δ to derive a *counter-example*: an SB-computation that reaches `Target`. From the counter example, we explain how to derive a set of fences in G such that the insertion of at least one element of the set is necessary in order to eliminate the counter-example. Finally, we introduce the fence insertion algorithm.

Fences We identify fences with local states. For a concurrent program $\mathcal{P} = (P,A)$ and a fence $f \in Q$, we use $\mathcal{P} \oplus f$ to denote the concurrent program we get by inserting a fence operation just after the local state f in $\mathcal{P}$. Formally, if $f \in Q_p$, for some $p \in P$, then $\mathcal{P} \oplus f := \left(P, \left\{A'_{p'} \mid p' \in P\right\}\right)$ where $A'_{p'} = A_{p'}$ if $p \neq p'$. Furthermore, if $A_p = \left(Q_p, q_p^{init}, \Delta_p\right)$, then we define $A'_p = \left(Q_p \cup \{q'\}, q_p^{init}, \Delta'_p\right)$ with $q' \notin Q_p$, and $\Delta'_p = \Delta_p \cup \{(f, \mathsf{fence}, q')\} \cup \{(q', op, q'') \mid (f, op, q'') \in \Delta_p\} \setminus \{(f, op, q'') \mid (f, op, q'') \in \Delta_p\}$. We say F is *minimal* wrt. a set `Target` of local state definitions and a placement constraint G if $F \subseteq G$ and $\mathit{Reachable}(SB)(\mathcal{P} \oplus F \setminus \{f\})(\mathtt{Target})$ holds for all $f \in F$ but not $\mathit{Reachable}(SB)(\mathcal{P} \oplus F)(\mathtt{Target})$. We use $F_{min}^{G}(\mathcal{P})(\mathtt{Target})$ to denote the set of minimal sets of fences in $\mathcal{P}$ wrt. `Target` that respect the placement constraint G.

Counter-Example Generation Consider a trace $\delta = A_0 t_1 A_1 t_2 \cdots t_n A_n$. We show how to derive a counter-example from δ. Formally, a counter-example is a run $c_0 \xrightarrow{t_1}_{SB} c_1 \xrightarrow{t_2}_{SB} \cdots \xrightarrow{t_m}_{SB} c_m$ of the transition system induced from $\mathcal{P}$ under the SB semantics, where $c_0 \in \mathtt{Init}_{SB}$ and $c_m \in \left\{(\underline{q}, b, \underline{z}) \mid \underline{q} \in \mathtt{Target}\right\}$. We assume a function *choose* that, for each automaton A, chooses a member of $L(A)$ (if $L(A) \neq \emptyset$), i.e., $\mathit{choose}(A) = w$ for some arbitrary but fixed $w \in L(A)$. We will define π using a sequence of configurations $c_0, \ldots, c_n$ where $c_i \in L(A_i)$ for $i : 0 \leq i \leq n$. Define

$c_0 := \mathit{choose}\left(A_0 \cap A^{\mathit{init}}\right)$. The first configuration c_0 in π is a member of the intersection of A_0 and A^{init} (this intersection is not empty by the definition of a trace). Suppose that we have computed c_i for some $i : 0 \leq i < n$. Since $A_i = \mathtt{Pre}_{t_{i+1}}(A_{i+1})\uparrow$ and $c_i \in L(A_i)$, there exist $c_i' \in \mathtt{Pre}_{t_{i+1}}(A_{i+1}) \subseteq L(A_i)$ and $d_{i+1} \in L(A_{i+1})$ such that $c_i' \sqsubseteq c_i$ and $c_i' \xrightarrow{t_{i+1}}_{SB} d_{i+1}$. Since there are only finitely many configurations that are smaller than c_i wrt. $\sqsubseteq$, we can indeed compute both c_i' and d_{i+1}. By Lemma 2, we know we can compute a configuration c_{i+1} and a run π_{i+1} such that $d_{i+1} \sqsubseteq c_{i+1}$ and $c_i \xrightarrow{\pi_{i+1}}_{SB} c_{i+1}$. Since $L(A_{i+1}\uparrow)$ is upward closed, we know that $c_{i+1} \in L(A_{i+1}\uparrow)$. We define $\pi := c_0 \bullet \pi_1 \bullet c_1 \bullet \pi_2 \bullet \cdots \bullet \pi_n \bullet c_n$. We use $\mathtt{CounterEx}(\delta)$ to denote such a π.

Fence Inference We will identify points along a counter-example $\pi = c_0 \xrightarrow{t_1}_{SB} c_1 \xrightarrow{t_2}_{SB} \cdots \xrightarrow{t_{n-1}}_{SB} c_{n-1} \xrightarrow{t_n}_{SB} c_n$ at which read operations overtake write operations and derive a set of fences such that any one of them forbids such an overtaking. We do this in several steps. Let c_i be of the form $\left(\underline{q_i}, b_i, \underline{z_i}\right)$. Define $n_i := |b_i|$. First, we define a sequence of functions $\alpha_0, \ldots, \alpha_n$ where α_i associates to each message in the buffer b_i the position in π of the write transition that gave rise to the message. Below we explain how to generate those α functions. The first message $b_i(1)$ in each buffer represents the initial state of memory. It has not been generated by any write transition, and therefore $\alpha_i(1)$ is undefined. Since b_0 contains exactly one message, $\alpha_0(j)$ is undefined for all j. If t_{i+1} is not a write transition then define $\alpha_{i+1} := \alpha_i$ (no new message is appended to the buffer, so all transitions associated to all messages have been defined). Otherwise, we define $\alpha_{i+1}(j) := \alpha_i(j)$ if $2 \leq j \leq n_i$ and define $\alpha_{i+1}(n_i+1) := i+1$. In other words, a new message will be appended to the end of the buffer (placed at position $n_{i+1} = n_i + 1$); and to this message we associate $i+1$ (the position in π of the write transition that generated the message).

Next, we identify the write transitions that have been overtaken by read operations. Concretely, we define a function $\mathtt{Overtaken}$ such that, for each $i : 1 \leq i \leq n$, if t_i is a read transition then the value $\mathtt{Overtaken}(\pi)(i)$ gives the positions of the write transitions in π that have been overtaken by the read operation. Formally, if t_i is not a read transition define $\mathtt{Overtaken}(\pi)(i) := \emptyset$. Otherwise, assume that $t_i = (q, \mathsf{r}(x,v), q') \in \Delta_p$ for some $p \in P$. We have $\mathtt{Overtaken}(\pi)(i) := \left\{\alpha_i(j) \middle| \, \mathtt{LastWrite}(c_i, p, x) < j \leq n_i \wedge t_{\alpha_i(j)} \in \Delta_p\right\}$. In other words, we consider the process p that has performed the transition t_i and the variable x whose value is read by p in t_i. We search for pending write operations issued by p on variables different from x. These are given by transitions that (i) belong to p and (ii) are associated with messages inside the buffer that belong to p and that are yet to be used for updating the memory (they are in the postfix of the buffer to the right of $\mathtt{LastWrite}(c_i, p, x)$).

Finally, we notice that, for each $i : 1 \leq i \leq n$ and each $j \in \mathtt{Overtaken}(\pi)(i)$, the pair (j,i) represents the position j of a write operation and the position i of a read operation that overtakes the write operation. Therefore, it is necessary to insert a fence at least in one position between such a pair in order to ensure that we eliminate at least one of the overtakings that occur along π. Furthermore, we are only interested in local states that belong to the placement constraint G. To reflect this, we define $\mathtt{Barrier}(G)(\pi) := \left\{\underline{q_k}(p) \middle| \, \exists i : 1 \leq i \leq n.\ \exists j \in \mathtt{Overtaken}(\pi)(i).\ j \leq k < i\right\} \cap G$.

Algorithm Our fence insertion algorithm (Algorithm 2) inputs a concurrent program $\mathcal{P}$, a placement constraint G, and a finite set `Target` of local state definitions, and returns all minimal sets of fences ($F^G_{min}(\mathcal{P})(\texttt{Target})$). If this set is empty then we conclude that the program cannot be made correct by placing fences in G. In this case, and if $G = Q$ (or indeed, if G includes sources of all read operations or destinations of all write operations), the program is not correct even under SC-semantics (hence no set of fences can make it correct).

Theorem 3. *For a concurrent program $\mathcal{P}$, a placement constraint G, and a finite set* `Target`*, Algorithm 2 terminates and returns* $F^G_{min}(\mathcal{P})(\texttt{Target})$.

Algorithm 2: Fence Inference

input : concurrent program $\mathcal{P}$, placement constraint G, local state definitions `Target`.
output: $F^G_{min}(\mathcal{P})(\texttt{Target})$.

$\mathcal{W} \leftarrow \{\emptyset\}$;
$\mathcal{C} \leftarrow \emptyset$;
while $\mathcal{W} \neq \emptyset$ **do**
  Pick and remove a set F from $\mathcal{W}$;
  if $Reachable(SB)(\mathcal{P} \oplus F)(\texttt{Target}) = \delta$ **then**
    $F_B \leftarrow \texttt{Barrier}(G)(\texttt{CounterEx}(\delta))$;
    if $F_B = \emptyset$ **then**
      return $\emptyset$
    else foreach $f \in F_B$ **do**
      $F' \leftarrow F \cup \{f\}$;
      if $\exists F'' \in \mathcal{C} \cup \mathcal{W}.\ F'' \subseteq F'$ **then**
        discard F'
      else $\mathcal{W} \leftarrow \mathcal{W} \cup \{F'\}$
  else
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{F\}$
return $\mathcal{C}$;

Remark 1. If only a smallest minimal set is of interest, then it is sufficient to implement $\mathcal{W}$ as a queue and to return the first added element to $\mathcal{C}$.

7 Experimental Results

We have evaluated our approach on several benchmark examples including some difficult problem sets that cannot be handled by *any previous approaches*. We have implemented Algorithm 2 in OCaml and run the experiments using a laptop computer with an Intel Core i3 2.26 GHz CPU and 4GB of memory. Table 1 summarizes our results. The placement constraint only allows fences immediately after write operations. The experiments were run in two modes: one until the first minimal set of fences is found, and one where all minimal sets of fences are found. For each concurrent program we give the program size (number of processes, number of states, variables and transitions), the total required time in seconds, the number of inserted fences in the smallest minimal fence set and the number of minimal fence sets.

Our implementation is able to verify all above examples. This is beyond the capabilities of previous approaches. In particular, none of our examples is data-race free. Furthermore, some of our examples may generate an arbitrary number of messages inside the buffers and they may have sequential inconsistent behaviors. To the best of our knowledge, only the approaches in [19] and in [22] are potentially able to handle such general classes of problems. However, the approach of [22] does not guarantee termination. The work in [19] abstracts away the *order* between buffer messages, and hence it cannot handle examples where the order of messages sent to the buffer is crucial (such as the "Increasing Sequence" example in the table). See the appendix for further details.

	Size Proc./States/Var./Trans	Total time seconds (one fence set)	Total time seconds (all fence sets)	Fences necessary (smallest set)	Number of minimal fence sets
1. Simple Dekker [31]	2/8/2/10	0.02	0.02	1 per process	1
2. Full Dekker [11]	2/14/3/18	0.28	0.28	1 per process	1
3. Peterson [29]	2/10/3/14	0.24	0.6	1 per process	1
4. Lamport Bakery [20]	2/22/4/32	52	5538	2 per process	4
5. Lamport Fast [21]	2/26/4/38	6.5	6.5	2 per process	1
6. CLH Queue Lock[25]	2/48/4/60	26	26	0	1
7. Sense Reversing Barrier [26]	2/16/2/24	1.1	1.1	0	1
8. Burns [24]	2/9/2/11	0.07	0.07	1 per process	1
9. Dijkstra [24]	2/14/3/24	9.5	10	1 per process	1
10. Tournament Barriers [14]	2/8/2/8	1.2	1.2	0	1
11. A Task Scheduling Algorithm	3/7/2/9	60	60	0	1
12. Increasing Sequence	2/26/1/44	25	27	0	1
13. Alternating Bit	2/8/2/12	0.2	0.2	0	1
14. Producer Consumer, v1, N=2	18/3/22	0.2	0.2	Erroneous	0
15. Producer Consumer, v1, N=3	22/4/28	4.5	4.5	Erroneous	0
16. Producer Consumer, v2, N=2	14/3/18	5.7	5.7	0	1
17. Producer Consumer, v2, N=3	16/4/22	580	583	0	1

Table 1. Analyzed concurrent programs

8 Conclusion

We have presented a sound and complete method for automatic fence insertion in finite-state programs running under the TSO memory model, based on a new (so called) SB-semantics. We have automatically verified several challenging examples, including some that cannot be handled by existing approaches. The design of the new SB semantics is not a trivial task. For instance, "obvious" variants such as simply making the buffer in TSO "lossy", or removing the pointers or storing less information inside the messages of the SB-buffer would fail, since they yield either over- or under-approximations (even wrt. reachability properties). Also the ordering we define on SB configurations cannot be "translated back" to an ordering on TSO configuration (this would make it possible to apply our method directly on TSO rather than on the SB semantics). The reason is that standard proofs that show reductions between different semantics (models), where each configuration in one model is shown to be in (bi-)simulation with a configuration in the other model cannot be used here. Given an SB-configuration, it is not obvious how to define an "equivalent" TSO configuration, and vice versa. However (crucially, as shown in the proof of Theorem 1) we show that each computation in one semantics violating/satisfying a given safety property is simulated by a (whole) computation that violates/satisfies the same safety property in the other. Our method can be carried over to other memory models such as PSO in a straightforward manner. In the future, we plan to apply our techniques to more memory models and to combine with predicate abstraction for handling programs with unbounded data.

References

1. P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *LICS*, 1996.
2. S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12), 1996.
3. J. Alglave and L. Maranget. Stability in weak memory models. In *CAV*, 2011.

4. M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In *POPL*, 2010.
5. M. F. Atig, A. Bouajjani, and G. Parlato. Getting rid of store-buffers in TSO analysis. In *CAV*, 2011.
6. S. Burckhardt, R. Alur, and M. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *PLDI*, 2007.
7. S. Burckhardt, R. Alur, and M. M. Martin. Bounded model checking of concurrent data types on relaxed memory models: A case study. In *CAV*, 2006.
8. S. Burckhardt, R. Alur, and M. Musuvathi. Effective program verification for relaxed memory models. In *CAV*, 2008.
9. J. Burnim, K. Sen, and C. Stergiou. Testing concurrent programs on relaxed memory models. Technical Report UCB/EECS-2010-32, UCB, 2010.
10. J. Burnim, K. Sen, and C. Stergiou. Sound and complete monitoring of sequential consistency in relaxed memory models. In *TACAS*, 2011.
11. E. W. Dijkstra. *Cooperating sequential processes*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.
12. X. Fang, J. Lee, and S. P. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *ICS*. ACM, 2003.
13. K. Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, 2004.
14. D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *IJPP*, 17, February 1988.
15. G. Higman. Ordering by divisibility in abstract algebras. *Proc. London Math. Soc. (3)*, 2(7), 1952.
16. T. Huynh and A. Roychoudhury. A memory model sensitive checker for C#. In *FM*, 2006.
17. I. Inc. Intel^{TM}64 and IA-32 Architectures Software Developer's Manuals.
18. M. Kuperstein, M. Vechev, and E. Yahav. Automatic inference of memory fences. In *FMCAD*, 2011.
19. M. Kuperstein, M. Vechev, and E. Yahav. Partial-coherence abstractions for relaxed memory models. In *PLDI*, 2011.
20. L. Lamport. A new solution of dijkstra's concurrent programming problem. *CACM*, 17, August 1974.
21. L. Lamport. A fast mutual exclusion algorithm, 1986.
22. A. Linden and P. Wolper. An automata-based symbolic approach for verifying programs on relaxed memory models. In *SPIN*, 2010.
23. A. Linden and P. Wolper. A verification-based approach to memory fence insertion in relaxed memory systems. In *SPIN*, 2011.
24. N. Lynch and B. Patt-Shamir. DISTRIBUTED ALGORITHMS , Lecture Notes for 6.852 FALL 1992. Technical report, MIT, Cambridge, MA, USA, 1993.
25. P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *IPPS*. IEEE Computer Society, 1994.
26. J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9, February 1991.
27. S. Owens. Reasoning about the implementation of concurrency abstractions on x86-tso. In *ECOOP*. 2010.
28. S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-tso. In *TPHOL*, 2009.
29. G. L. Peterson. Myths About the Mutual Exclusion Problem. *IPL*, 12(3), 1981.
30. P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-tso: A rigorous and usable programmer's model for x86 multiprocessors. *CACM*, 53, 2010.
31. D. Weaver and T. Germond, editors. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall, 1994.