Counter-Example Guided Fence Insertion under Weak Memory Models

Parosh Aziz Abdulla
Uppsala University
parosh@it.uu.se

Mohamed Faouzi Atig
Uppsala University
mohamed_faouzi.atig@it.uu.se

Yu-Fang Chen
Academia Sinica
yfc@iis.sinica.edu.tw

Carl Leonardsson
Uppsala University
carl.leonardsson@it.uu.se

Ahmed Rezine
Uppsala University
rezine.ahmed@it.uu.se

Abstract

We give a *sound* and *complete* procedure for fence insertion 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 the places inside the program where fences may be inserted. For a given placement constraint, the method infers automatically all minimal sets of fences that ensure correctness of the program. 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.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification.

General Terms Verification, Theory, Reliability.

Keywords Program verification, Relaxed memory models, Infinite-state systems.

1. Introduction

Background The *Out-of-Order Execution (OoOE)* paradigm lets the scheduling of CPU instructions be governed by the availability of their operands rather than by the order in which they are issued [39]. This leads to an efficient use of clock cycles and hence an improvement in program execution times. The gain in efficiency has meant that the OoOE technology is present in most modern processor architectures. In the context of *sequential* programming, the technique is transparent to the programmer since she can still work under the Sequential Consistency (SC) model [26] in which the program behaves according to the classical interleaving semantics. However, this is not true any more once we consider concurrent processes that share the memory. In fact, several algorithms that are designed for the synchronization of concurrent processes, such as mutual exclusion and producer-consumer protocols, are not correct in the OoOE setting. The inadequacy of the interleaving semantics in the presence of OoOE has prompted researchers to introduce *weak (or relaxed) memory models* by allowing permutations between certain types of memory operations [2, 3, 14]. Weak memory models are used in all major CPU designs including Intel x86 [22, 38], SPARC [40], and PowerPC [21]. Since the weak memory semantics adds more behavior to a program, the program may violate its specification even if it runs correctly under the SC semantics. 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. A fence instruction, executed by a process, implies that no reordering is allowed between instructions issued before and after the fence instruction.

The most common relaxation corresponds to TSO (for Total Store Ordering) that is adopted by Sun's SPARC multiprocessors [40]. TSO is the kernel of many common weak memory models and is the latest formalization of the x86-tso memory model [35, 38].

Challenge Processor vendors do not provide formal definitions for the memory models of their products, but rather informal documents describing allowed/forbidden behaviors. This is not sufficient for the development of verification tools. Therefore, a substantial research effort has recently been devoted to this issue, resulting in formal (weak) memory models (both axiomatic and operational) for several different types of processors [35, 38]. In this paper, we describe how to insert sets of fences that are sufficient for making the program correct wrt. its specification. In doing this, we start from the premise that the formal models developed, e.g. in [35, 38], give faithful descriptions of the actual hardware on which we run our programs. The fence insertion procedure gives rise to two important challenges. First, we need to be able to perform program verification; and in particular to be able to verify the correctness of the program for a given set of fences. This is necessary in order to be able to decide whether the current set of fences is sufficient, or whether additional fences are needed to ensure correctness. Program verification here is even more complicated than usual, since the above mentioned operational models often provide a *store-buffer* semantics in which one or more buffers are associated with each processor. The store buffers may grow unboundedly during a run of the program, which results in an *infinite* state space even in the case where the program operates on a finite data domain. Second, we need to optimize the manner in which we place fences inside the program. We would like to avoid the naive approach where we insert a fence instruction after every write instruction, or before every read instruction. Adopting this approach would result in a significant performance degradation [17] as it would mean that we would get back to the SC model. A natural criterion is to provide *minimal* sets of fences whose insertion is sufficient for ensuring program correctness under the given weak memory model.

Existing Approaches 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, or by proposing exact algorithms for which termination is *not* guaranteed. Under-approximations of the program behavior can be achieved through testing [11], bounded model checking [8], or by restricting the behavior of the program, e.g., through bounding the sizes of the buffers [23]. Such techniques are useful in practice for finding errors in the program. However, they are not able to check all the possible traces of the program and therefore they cannot tell whether the generated set of fences is sufficient for the correctness of the program. Over-approximative techniques have recently been proposed based on abstraction [24]. Such methods 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 [34, 37]. Considering only data-race free programs can be unrealistic since data races are very common in efficient implementations of concurrent algorithms. The method of [29] 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. In contrast to these approaches, our method performs *exact* analysis of the program on the given memory model. Termination of the analysis is guaranteed. As a consequence, we are also able to compute all *minimal* sets of fences that are required for correctness of the program.

Our Approach We present a sound and complete method for fence insertion 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 [38]). 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, 16] 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. This procedure is guaranteed to terminate. 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 This paper gives for the first time a *sound* and *complete* procedure for fence insertion for programs running under TSO. The main ingredients of the framework are the following:

- A new semantical model, the so called SB model, that allows efficient infinite state model checking.
- A simple and effective backward analysis algorithm for solving the reachability problem under the SB semantics. 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.
- A counter-example guided fence insertion procedure that automatically infers the minimal set of fences necessary for the correctness of the program under a given fence placement policy.
- 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.

Outline In Section 2 we give an overview of our framework. In Section 3 we give some preliminaries. In Section 4 we introduce our model of concurrent programs, recall the formal model of TSO, and then introduce the SB semantics. In Section 5 we provide an algorithm for backward reachability analysis of programs under the SB semantics. We explain in Section 6 how we use the analysis to automatically derive all minimal sets of fences that ensure correctness of the program. In Section 7 we report on our experimental results. We make a detailed comparison with related work in Section 8. Finally, we give in Section 9 some conclusions and directions for future research. The proofs of the lemmas, details of the implementation, and the experimental results are in the appendix.

2. Overview

An Example Figure 1 shows the code for Burn's mutual exclusion protocol, instantiated for two processes. It consists of a concurrent program with two processes that repeatedly enter and exit their critical sections. We want the program to satisfy a safety property, namely that of *mutual exclusion*. Checking such a safety property can be reduced to solving a *reachability problem*: check whether the program will ever reach a *bad configuration*, i.e., a configuration in which the two processes are both in their critical sections. The program satisfies the specification under the SC semantics.

```
  // process[0]:
1 while true
2   store flag[0]=1;
3   fence;
4   load _flag=flag[1];
5   if _flag==1 goto 4;
6   //CS
7   store flag[0]=0;

  // process[1]:
1 while true
2   store flag[1]=0;
3   load _flag=flag[0];
4   if _flag==1 goto 2;
5   store flag[1]=1;
6   fence;
7   load _flag=flag[0];
8   if _flag==1 goto 2;
9   // CS
10  store flag[1] = 0;
```

Figure 1. Burn's Mutual Exclusion Algorithm. Local variables are prefixed with an underscore and the initial values of variables are 0. Notice that the definitions of the two processes are not symmetric.

However, as we will see below, the program fails to satisfy its specification under TSO, if any of the fences is removed.

Total Store Order (TSO) A memory model is defined by the order in which the read (load) and write (store) operations are performed.

The *Sequential Consistency (SC)* semantics is the classical interleaving semantics, in which a trace of the system is an interleaving of traces of the different processes. In particular a store operation is visible to all processes immediately after it has been performed. In the *Total Store Order (TSO)* semantics, read operations are allowed to overtake write operations of the same process if they concern different variables. More precisely, each process sees its own read and write operations exactly in the same order as it has issued them. However, other processes may see older values than the one that has been stored by a process. TSO is thus also referred to as the store $\rightarrow$ load order relaxation. Each possible execution of the program under the SC semantics is also possible under the TSO semantics. However, the converse is not true. For example, if we remove the fences in Figure 1, the program does not satisfy mutual exclusion any more. In fact, as described below, there are at least two runs of the systems that can reach a bad configuration. Let us for the time being ignore the fences at lines 3 resp. 6 in the definitions of the processes. *Run 1*: *process*[1] reorders the read at line 7 with the write at line 5 as they concern different variables. As a result, *process*[1] can execute lines 1-4, then line 7, and before executing line 5, *process*[0] proceeds to the critical section as $flag[1]$ is still 0. After that, *process*[1] can run lines 5 and 8 before also reaching the critical section and violating mutual exclusion. *Run 2* is obtained in a similar manner, now by letting *process*[0] reorder lines 2 and 4.

Fence Insertion To eliminate the errors that may arise due to OoOEs (such as the ones described above for Burn's protocol), processor vendors provide *fence operations* that give the programmer more control over the executions of the program. More precisely, a fence inserted somewhere inside the program, restricts the reordering of the operations before and after the fence: the operations before the fence must take global effect before the execution of the operations after the fence. There are different types of fences depending on the operations they control (e.g., store-store or store-load). In the context of TSO, the relevant fence type is that of a *full memory barrier* that prevents the reordering of all memory operations. In the example of Figure 1, a fence is needed at line 6 in *process*[1] to forbid *Run 1* (the reordering of lines 5 and 7); and a fence is also needed at lines 3 in *process*[0] to prevent *Run 2*. Executing programs with fences is more costly than executing them without, and in fact inserting large numbers of fences goes against the spirit of OoOE as the program will approach its SC behavior. Therefore, one reasonable criterion is to insert as few fences as possible (provided that the set of inserted fences guarantees correctness of the program).

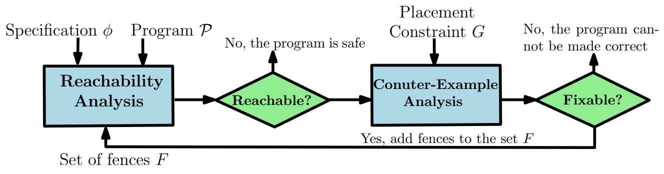
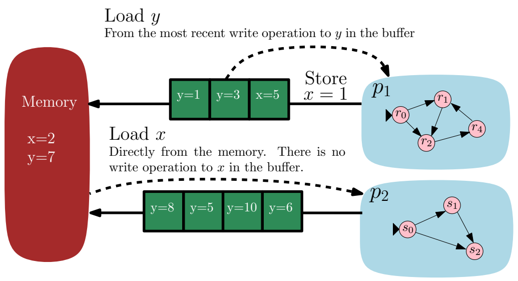


Figure 2. The flow of counter-example guided fence insertion.

Our Method Figure 2 gives an overview of our fence insertion procedure. Given a program $\mathcal{P}$ and a specification (a safety property ϕ), the procedure finds the set of minimal sets of fences necessary in order to make the $\mathcal{P}$ correct wrt. ϕ. The procedure performs a counter-example guided refinement of sets of fences. More precisely, it maintains a set of candidate sets of fences that it will refine continuously during the run of the algorithm. The procedure starts optimistically by only having the empty set as a candidate (i.e., assuming that no fences are needed for correctness). Given a candidate set F, the *Reachability Analysis* module checks whether $\mathcal{P}$, with F inserted, satisfies ϕ. This question is translated into the reachability of a set of *bad configurations*. If no bad configuration is reachable, we conclude that F is sufficient for correctness and proceed to the next candidate set. Otherwise, the module returns a counter-example which will be provided to the *Counter-Example Analysis* module. The module either generates new fences that need to be added to F or concludes that the program cannot be made correct, in which case the whole procedure terminates. The procedure also terminates in case there are no other candidate sets to consider. The algorithm is parameterized by a predefined *placement constraint* which is a subset G of all local states of the processes. 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 that are sufficient for correctness. Below, we explain in more detail the main ingredients of the procedure.

Operational TSO Semantics Program verification requires a formal model of the system under verification. In our setting, this implies that we need a formal description of the TSO memory model. For this purpose, we use the operational semantics defined in [38, 40]. Conceptually, the model adds a FIFO buffer between each process and the main memory (cf. Figure 3). The buffer is used



Load y
From the most recent write operation to y in the buffer
Memory
x=2
y=7
y=1
y=3
x=5
Store
$x = 1$
p_1
r_0
r_1
r_2
r_4
Load x
Directly from the memory. There is no write operation to x in the buffer.
y=8
y=5
y=10
y=6
p_2
s_0
s_1
s_2

Figure 3. Store buffers and the shared memory of a program under TSO. The size of the store buffer is unbounded.

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 such a write operation. 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. 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. For instance, in the program of Figure 1, the loop of lines 2–4 of process[1] may generate an unbounded number of write operations (issued at line 2), and hence create an unbounded number of elements inside the store buffer of process[1]. In fact, this still holds even after the insertion of fences since the inserted fences do not affect the operations inside the loop.

The SB Semantics The main obstacle in the design of the reachability algorithm is the fact the formal model of TSO [38] equips the processes with FIFO store buffers that are perfect and (potentially) unbounded. If one aims at ensuring correctness of the program regardless of the underlying (TSO) architecture, the sizes of the buffers cannot be pre-assumed. This gives rise to a difficult problem; it is well-known that the problem of checking safety properties for finite-state processes communicating through unbounded FIFO channels is undecidable [7]. However, the TSO model does not exploit the full power of perfect FIFO buffers. This is demonstrated, for instance, by the fact that the reachability problem for finite-state programs running on TSO is in fact decidable [5]. Our goal is to exploit this in order to first design a method for algorithmic verification of programs running under the TSO semantics, and then use it to develop a method for automatic fence insertion. Concretely, we will use the framework of *well quasi-ordered* transition systems [1, 16] in order to derive an algorithm for reachability analysis. The main challenge in using this framework is to define a pre-order $\preceq$ on the set of configurations, such that the transition relation is *monotonic* wrt. $\preceq$ (informally, monotonicity means that larger configurations can simulate smaller configurations). To derive such an ordering, we make the observation that a write operation sent by one process to the buffer may never be noticed by the other processes. This is true since a value in the memory might be overwritten by other write operations before any other process has had time to read it. This suggests that we should define our ordering $\preceq$ to reflect the sub-word relation on the contents of the buffer. (to be more concrete, the buffer of p_1 in Figure 3 can be viewed as the word $[y{=}1][y{=}3][x{=}5]$; and then $[y{=}1][x{=}5]$ is one of its subwords). The intuition would be that the extra write operations in the larger buffers of a process may after all never be noticed by the other processes, and hence a larger configuration should be able to simulate a smaller configuration. Unfortunately, as we will see in Section 4, the transition system induced by the TSO model is not monotonic wrt. $\preceq$. In order to circumvent this problem, we propose a new semantical model, namely the Single-Buffer (SB) semantics. We defer the technical details of the SB semantics until Section 4, where we also explain how it is derived as an alternative to TSO. Roughly speaking, a system in the SB semantics contains only one store buffer that is shared by all the processes (cf. Figure 5). On the other hand, each message inside the SB buffer contains a copy of the whole memory (rather than containing a write operation to a single variable, as in the case of TSO); together with a finite amount of "control data". The SB model satisfies the two needed properties: (i) it is equivalent to TSO (we can reduce he reachability problem under TSO to the one under SB); and (ii) its transition system is monotonic wrt. the sub-word relation $\preceq$ on the (single) store buffer.

Reachability Analysis Given the pre-order $\preceq$ on the set of SB-configurations, we use the framework of [1, 16] to design our reachability analysis algorithm. Concretely, the algorithm works on infinite sets of SB-configurations that are upward closed wrt. the ordering $\preceq$. The algorithm performs backward reachability starting from the set of bad configurations (those violating the safety property). The monotonicity of the transition relation implies that all the generated sets are upward closed. Furthermore, termination of the algorithm is guaranteed since $\preceq$ is a *well quasi-ordering*. In our instantiation of the algorithm, we use an automata-based formalism as a symbolic representation of upward closed sets of configurations. If the safety property is violated, the algorithm returns a symbolic representation of a set of traces from which the *Counter-Example Analysis* module can refine the set of fences.

Counter-Example Guided Fence Insertion A naive way to find the minimal fence sets is to simply try out all combinations. Obviously, such an algorithm would not work in practice due to the large number of possible combinations. Using the counter-examples provided by our reachability algorithm, the *Counter-Example Analysis* module either finds new fences (satisfying the placement constraint G) to be added to the program, or it concludes that the program cannot be made correct (regardless of the number of inserted fences from G). In the latter case, we can conclude that the program is incorrect even under the SC semantics (this holds provided that G has been chosen so that it includes sources of all read operations or destinations of all write operations).

The fence refinement procedure means that the framework, as a whole, amounts to a counter-example guided fence insertion procedure, that automatically infers the minimal set of fences (satisfying a given fence placement constraint) that ensures the correctness of the program. The algorithm can be made to run until it has found all minimal sets of fences, or stop after finding the first set.

3. Preliminaries

In this section we first introduce notations that we use through the paper, and then define the notion of transition 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 that we get by deleting the prefix of length i, i.e., the unique w_2 such that $w = w_1 \cdot w_2$ and $|w_1| = i$.

Transition Systems A transition system $\mathcal{T}$ is a triple $(C, \texttt{Init}, \rightarrow)$ where C is a (potentially infinite) set of *configurations*, $\texttt{Init} \subseteq C$ is the set of *initial configurations*, and $\rightarrow \subseteq C \times C$ 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 *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 $target(\pi)$ 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 $c_0 \in \texttt{Init}$. 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 $\pi_1 = c_0 \rightarrow c_1 \rightarrow \cdots \rightarrow c_m \rightarrow c_{m+2} \rightarrow \cdots \rightarrow c_n$. Given an ordering $\sqsubseteq$ on 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' such that $c_2 \xrightarrow{*} c_2'$ and $c_1' \sqsubseteq c_2'$. We say that $\rightarrow$ is *effectively monotonic* wrt. $\sqsubseteq$ if, given the configurations c_1, c_1', c_2 described above, we can compute c_2' and a run π such that $c_2 \xrightarrow{\pi} c_2'$.

4. 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 (with finite domains) to which all threads can read and write. First, we define the syntax we use for concurrent programs. Next, we introduce the TSO semantics including the transition systems it induces and its reachability problem. Finally, we describe informally the different features of the SB semantics, and the manner in which we derive them from the definition of TSO; and then we define formally its transition system and reachability problem.

4.1 Syntax

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) the *"no operation"* nop, (2) the *read operation* $\mathsf{r}(x,v)$, (3) the *write operation* $\mathsf{w}(x,v)$, (4) the *fence operation* fence, and (5) the *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 $p \in P$. It is straightforward to translate programs in the form of Figure 1 to this model. Figure 4 is an automaton for *process*[0] in Figure 1.

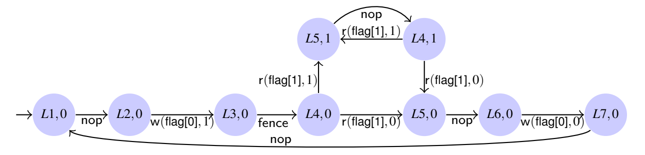


Figure 4. The automaton of *process*[0] in Figure 1. Local states encode program locations and the value of the local variable _flag.

4.2 TSO Semantics

We refer to Section 2 for an informal description TSO semantics.

Transition System 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. 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 state of the memory (defines the value of each variable in the memory). We use C_{TSO} to denote the set of TSO-configurations. In Figure 3, we have $\underline{b}(p_1) = [y = 1][y = 3][x = 5]$, $\underline{b}(p_2) = [y = 8][y = 5][y = 10][y = 6]$, $mem(x) = 2$, and $mem(y) = 7$ (to increase readability in the examples, we write the contents of the buffers in the form $[y = 1][y = 3][x = 5]$ instead of $(y,1)(y,3)(x,5)$). We define the transition relation $\rightarrow_{TSO}$ on C_{TSO}. The relation is induced by (1) members of Δ; and (2) a set $\Delta' := \{\mathsf{update}_p \mid p \in P\}$ where 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 $t \in \Delta_p \cup \{\mathsf{update}_p\}$, 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 state while the buffer contents and the memory 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. In Figure 3, executing a transition of the form $(q, \mathsf{w}(x,2), q') \in \Delta_{p_1}$ would give $\underline{b}'(p_1) = [y = 1][y = 3][x = 5][x = 2]$.
- Update: $t = \mathsf{update}_p$, $\underline{q}' = \underline{q}$, $\underline{b} = \underline{b}'\left[p \hookleftarrow (x,v) \cdot \underline{b}'(p)\right]$, and $mem' = mem[x \hookleftarrow v]$. The write in the head of the buffer is removed and the memory is updated accordingly. In Figure 3, update_{p_1} would give $\underline{b}'(p_1) = [y = 3][x = 5]$ and $mem'(y) = 1$.
- 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 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 on x in the buffer of p then we consider the most recent of such write operations (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 the memory.

  In Figure 3, p_1 can read the values $x = 5$ and $y = 3$, while p_2 can read the value $x = 2$ and $y = 6$.
- 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 is performed atomically. It may be performed by a process only if its buffer is empty. The operation checks whether the value of variable x is v. In such a case, it changes its value to v'. Note this operation permits to model instructions like locked writes under x86-tso [22, 38] or compare-and-swap or swap under SPARC [40].

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 $\left(\underline{q_{init}}, \underline{b_{init}}, mem_{init}\right)$ 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 $(C_{TSO}, \mathtt{Init}_{TSO}, \rightarrow_{TSO})$.

The TSO Reachability Problem Given a set $\mathtt{Target}$ of local state definitions, we use $Reachable(TSO)(\mathcal{P})(\mathtt{Target})$ to be a predicate that indicates the reachability of the set $\{(\underline{q}, \underline{b}, mem) \mid \underline{q} \in \mathtt{Target}\}$, i.e., whether a configuration c, where the local state definition of c belongs to $\mathtt{Target}$, is reachable. The reachability problem for TSO is to check, for a given $\mathtt{Target}$, whether $Reachable(TSO)(\mathcal{P})(\mathtt{Target})$ holds or not. Using standard techniques we can reduce checking safety properties to the reachability problem. More precisely, we use $\mathtt{Target}$ to denote "bad configurations" that we do not want to occur during the execution of the system. For instance, in Burn's protocol (Section 2), the bad configurations are those where the two processes are in line 6 resp. line 9 (corresponding to the critical sections of the two processes). Therefore, we often say that the "program is correct" to indicate that $\mathtt{Target}$ is not reachable.

4.3 Single-Buffer Semantics

As explained in Section 2, our goal is to derive a semantical model that is both equivalent to TSO and monotonic.

Informal Explanation Below, we motivate the different features of the SB semantics and explain how we derive them from the TSO semantics. To do that, we start from TSO and perform a number of steps where we define new semantics SB1, SB2, SB3, until we arrive at the final definition of SB. The semantics introduced in each step is derived from the one in the previous step, by adding new features that are used to solve certain problems. These problems are illustrated through examples of concrete runs of the system. First, we illustrate why the TSO semantics is not monotonic wrt. the sub-word relation on the buffer contents. Recall that a system is monotonic if all the behaviors of a smaller configuration can also occur from a larger configuration. We consider a program with two processes p_1, p_2, where the automaton of p_1 contains the following two transitions:

$$\begin{tikzcd} p_1 : & q_0 \arrow[r, "r(x,1)"] & q_1 \arrow[r, "r(y,0)"] & q_2 \end{tikzcd}$$

Consider a configuration c_1 where the local state of p_1 is q_0, the memory contains $x = 0, y = 0$, the buffer of p_1 is empty, and the buffer of p_2 contains the word $[x{=}1]$. Then, according to the TSO semantics, there is a run of the system from c_1 to a configuration where the local state of p_1 is q_2; the run simply updates the memory by $x = 1$, and then performs the two read transitions. Consider the configuration c_2 where the buffer of p_2 now contains the word $[y{=}1][x{=}1]$. The local state q_2 is not reachable any more; we have to update the memory by $x = 1$ (otherwise we cannot perform the read transition from q_0 to state q_1). However, this implies that we have already updated the memory by $y = 1$ which means that we cannot perform the read transition from q_1 to state q_2. This violates monotonicity: some behavior of a smaller configuration c_1 is not a behavior of a larger configuration c_2.

SB1. The problem with the above scenario is that we get a memory configuration $x = 1, y = 0$ in the smaller configuration c_1 that is inconsistent with those we get from c_2. To cure this problem, we define SB1, where we let the processes send entire memory snapshots to the buffer (rather than single write operations). In other words, each message in the buffer will now define the value of each variable in the program (rather than a single variable). The buffer contents of the configurations c_1 and c_2 in step 1 will be of the forms $[x{=}1,y{=}0]$ resp. $[x{=}0,y{=}1]$ $[x{=}1,y{=}1]$. Notice that the buffer contents are not related by the sub-word relation, and hence the configurations c_1, c_2 do not violate monotonicity anymore. The problem with SB1 is that it is not equivalent to TSO; some behavior in SB1 is not possible in TSO. To see this, we consider two processes p_1, p_2 whose automata contain the following transitions:

$$\begin{tikzcd} p_1 : & q_0 \arrow[r, "w(y,0)"] & q_1 & p_2 : & r_0 \arrow[r, "w(x,1)"] & r_1 \arrow[r, "r(x,0)"] & r_2 \end{tikzcd}$$

Consider a configuration c where the local states of p_1, p_2 are q_0, r_0, the memory contains $x = 0, y = 0$, and the buffers are empty. There is no run from c in TSO to a configuration where the local states are q_1, r_2 since p_2 can only fetch the value 1 from x once the operation $w(x,1)$ has been performed. However, such a configuration is reachable in SB1 as follows. The messages $[x{=}1,y{=}0]$ by p_2 and $[x{=}0,y{=}0]$ by p_1 are sent to the buffers and delivered to memory in that order. After these two operations, the memory value will be $x = 0, y = 0$, which allows the operation $r(x,0)$ to be performed, hence the system reaches the states q_1, r_2.

SB2. The problem in the above scenario is that the processes are not synchronized on memory updates, so a process may use some values that are not in the memory any more. In the above example, when p_1 sends the memory snapshot for $w(y,0)$ to its buffer, it should take the memory values contributed by other processes into consideration. Instead of sending $[x{=}0,y{=}0]$ that contains an old value of x, it should notice that p_2 has changed the memory value of x to 1 (by checking the most recent buffer message) and should hence send $[x{=}1,y{=}0]$. In SB2, we solve this problem by letting all the processes share a single buffer. Thus, the buffer contents in the above run just before performing the last operation $r(x,0)$ will be $[x{=}1,y{=}0][x{=}1,y{=}0]$ and the read statement from s_1 to s_2 is not enabled any more. Again SB2 is not equivalent to TSO; some behavior of TSO is not possible under SB2. To see this, consider four processes p_1, p_2, p_3, p_4 whose automata contain the following transitions:

$$\begin{tikzcd} p_1 : & q_0 \arrow[r, "w(x,1)"] & q_1 \arrow[r, "w(x,2)"] & q_2 & p_2 : & r_0 \arrow[r, "r(y,2)"] & r_1 \arrow[r, "r(y,1)"] & r_2 \\ p_3 : & s_0 \arrow[r, "w(y,1)"] & s_1 \arrow[r, "r(x,1)"] & s_2 & p_4 : & t_0 \arrow[r, "r(x,2)"] & t_1 \arrow[r, "w(y,2)"] & t_2 \end{tikzcd}$$

Consider a configuration c where the local states of p_1, p_2, p_3, p_4 are q_0, r_0, s_0, t_0, the memory contains $x = 0, y = 0$, and the buffers are all empty. There is a run from c in TSO to a configuration where the local states are q_2, r_2, s_2, t_2 as follows: (i) p_1 sends $[x{=}1]$ followed by $[x{=}2]$ to its buffer and moves to q_2; (ii) p_3 sends $[y{=}1]$ to its buffer; (iii) $[x{=}1]$ is updated to the memory from the buffer of p_1; (iv) p_3 reads $x = 1$ from the memory and moves to s_2; (v) $[x{=}2]$ is updated to the memory from the buffer of p_1; (vi) p_4 reads $x = 2$ from the memory and then sends $[y{=}2]$ to its buffer moving to state t_2; (vii) $[y{=}2]$ is updated to the memory from the buffer of p_4; (viii) p_2 reads $[y{=}2]$ from the memory; (ix) $[y{=}1]$ is updated to the memory from the buffer of p_3; (x) p_2 reads $y = 1$ from the memory and moves to r_2. Such a run is not possible in SB2 according to the following reasoning. The write operation $w(y,1)$ has to be sent to the buffer before the write operation $w(y,2)$; otherwise $r(x,2)$ is already performed and the value of x in the memory will be equal (and remain) to 2 when p_3 is in local state s_1. Hence p_3 cannot perform the read transition from s_1 to s_2. In SB2, the operations in the buffer will be delivered to the memory in the same order as they entered the buffer. So $w(y,1)$ will be delivered to the memory before the write operation $w(y,2)$. This means that p_2 cannot fetch the value $y = 2$ from the memory before it fetches the value $y = 1$, and hence p_2 will not be able to reach r_2.

SB3. The problem is that SB2 forces memory updates to be performed in the same order as the order of the corresponding write transitions even if these write transitions belong to different processes. For instance, in the above example, the write operation $w(y,1)$ was performed before the write operation $w(y,2)$. In the TSO semantics, the corresponding updates can be performed in the opposite order (since they belong to different buffers), while this is not allowed in SB2. In SB3, we remedy to this problem by providing the processes with a mechanism that allows them to update the memory independently of the other processes. More precisely, we add to each process a pointer to a position inside the buffer. From the point of view of the process, the buffer is divided by the pointer into three parts. The suffix of the buffer to the right of the pointer represents the sequence of write operations that have still not been used for memory updates. The position of the pointer itself represents the content of the memory, while the rest of the buffer (the prefix to the left of the pointer) is not relevant for the future behavior of the process (since it represents write operations that have already been used for updating the buffer). An update operation will then be simulated by moving the relevant pointer one step to the right. This adds the missing run mentioned above (cf. Figure 5). First, the write operations $x = 1$ and $x = 2$ are transferred to the buffer. Then, p_4 moves its pointer to the position of the buffer where $x = 2$ sending the write operation $y = 2$ to the buffer; and then p_3 moves its pointer to the position of the buffer where $x = 1$ sending the write operation $y = 1$ to the buffer. Now the write operations $y = 1$ and $y = 2$ are in the correct order ($y = 2$ before $y = 1$). Therefore, p_2 can first move its pointer to the position in the buffer where $y = 2$ after which it moves its pointer one step to the right in the buffer where $y = 1$. In this manner, it is able to perform the two read transitions in the correct order.

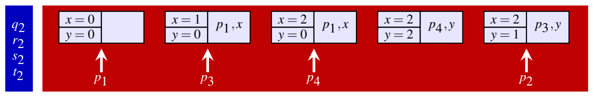


Figure 5. An SB-configuration.

SB. Finally, since the write operations of the different processes are now all mixed in the (single) buffer, we add also a mechanism that allows the processes to recognize the last write operations they have performed on each variable. Recall that in TSO, if p reads the value of x then it will fetch the value from the most recent message in its buffer that represents a write operation on x (if such a message exists), instead of getting the value of x directly from the main memory (cf. Figure 3). To do this in SB, we equip each message in the buffer with a process p and a variable x, which denotes "p writes to x". When a process p reads x, it takes the value from the most recent message with (p,x) in the buffer (or from the memory if the buffer does not have such a message). As an example, the SB-configuration at the end of the above described computation is shown in Figure 5. Notice that an SB-configuration does not have an explicit representation of the main memory, as this is replaced by the pointers that represent the local views of the processes.

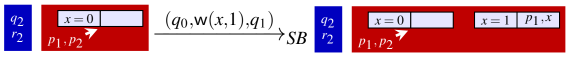
Finally, The ordering $\sqsubseteq$ on SB-configurations (formally defined in Section 5) is induced by the sub-word relation on the buffer contents. However, it will also reflect the last-write information on the variables, and the positions of the pointers inside the buffer.

Transition System Formally, an *SB-configuration* c is a triple $\left(\underline{q},b,\underline{z}\right)$ where $\underline{q}$ is (as in the case of the 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 the values of the variables (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 for each 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. We use C_{SB} to denote the set of SB-configurations. As an example, in the SB-configuration of Figure 5, the pointer of p_4 points to the message $[x{=}2,y{=}0,(p_1,x)]$. This means that the current view of p_4 is that the memory contains $x = 2$ and $y = 0$. From the point of view of p_4, the word $[x{=}2,y{=}2,(p_4,y)][x{=}2,y{=}1,(p_3,y)]$ represents those write operations that have not yet been delivered to the memory. 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 SB semantics, the update operation moves a pointer to the right instead of removing a message in the buffer). This implies (among other things) that the invariant $\underline{z}(p) > 0$ is always maintained.

Let $c = \left(\underline{q},b,\underline{z}\right)$ be an SB-configuration. For every $p \in P$ and $x \in X$, we use $\texttt{LastWrite}\,(c,p,x)$ to denote the index of the most recent buffer message where p writes to x or the message with the current memory of p if the aforementioned type of message does not exist in the buffer. For example, let c be the configuration in Figure 5, then $\texttt{LastWrite}\,(c,p_1,x) = 3$, $\texttt{LastWrite}\,(c,p_1,y) = 1$, $\texttt{LastWrite}\,(c,p_4,y) = 4$, and $\texttt{LastWrite}\,(c,p_3,x) = 2$. Formally, $\texttt{LastWrite}\,(c,p,x)$ is the largest index i such that $i = \underline{z}(p)$ or $b(i) = (mem,p,x)$ for some mem.

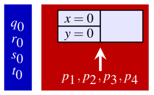
We define the transition relation $\rightarrow_{SB}$ on the set of SB-configurations as follows. In a similar manner to the case of TSO, the relation is induced by members of $\Delta \cup \Delta'$. For configurations $c = \left(\underline{q},b,\underline{z}\right)$, $c' = \left(\underline{q}',b',\underline{z}'\right)$,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

- Nop: $t = (q,\mathsf{nop},q')$, $\underline{q}(p) = q$, $\underline{q}' = \underline{q}\left[p \hookleftarrow q'\right]$, $b' = b$ and $\underline{z}' = \underline{z}$. The operation changes only local states.
- Write to store: $t{=}(q,\mathsf{w}(x,v),q')$, $\underline{q}(p){=}q$, $\underline{q}'{=}\underline{q}\left[p \hookleftarrow q'\right]$, $b(|b|)$ is of the form (mem_1,p_1,x_1), $b' = b \cdot (mem_1\left[x \hookleftarrow v\right],p,x)$, and $\underline{z}' = \underline{z}$. A new message is appended to the tail of the buffer. The values of the variables in the new message are identical to those in the previous last message except that the value of x has been updated to v. Moreover, we include the updating process p and the updated variable x. Below is an example of p_1 writes to x.



- Update: $t = \mathsf{update}_p$, $\underline{q}' = \underline{q}$, $b' = b$, $\underline{z}(p) < |b|$ and $\underline{z}' = \underline{z}\left[p \hookleftarrow \underline{z}(p)+1\right]$. An update operation performed by a process 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. For example, update_{p_4} moves the pointer of p_4 in Figure 5 from $[x{=}2,y{=}0,(p_1,x)]$ to $[x{=}2,y{=}2,(p_4,y)]$.
- Read: $t = (q,\mathsf{r}(x,v),q')$, $\underline{q}(p) = q$, $\underline{q}' = \underline{q}\left[p \hookleftarrow q'\right]$, $b' = b$, and $b(\texttt{LastWrite}\,(c,p,x)) = (mem_1,p_1,x_1)$ for some mem_1,p_1,x_1 with $mem_1(x) = v$. As an example, suppose that p_1 reads the variable x in the configuration of Figure 5. The message $b(\texttt{LastWrite}\,(c,p_1,x)) = [x{=}2,y{=}0,(p_1,x)]$. It follows that a transition with read operation $\mathsf{r}(x,v)$ is enabled only when $v = 2$.
- Fence: $t = (q,\mathsf{fence},q')$, $\underline{q}(p) = q$, $\underline{q}' = \underline{q}\left[p \hookleftarrow q'\right]$, $\underline{z}(p) = |b|$, $b' = b$, 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 $\underline{z}(p) = |b|$. In Figure 5, p_2 is the only process that can execute a fence operation (its buffer is empty).
- ARW: $t = (q,\mathsf{arw}(x,v,v'),q')$, $\underline{q}(p) = q$, $\underline{q}' = \underline{q}\left[p \hookleftarrow q'\right]$, $\underline{z}(p) = |b|$, $b(|b|)$ is of the form (mem_1,p_1,x_1), $mem_1(x) = v$, $b' = b \cdot (mem_1\left[x \hookleftarrow v'\right],p,x)$, and $\underline{z}' = \underline{z}\left[p \hookleftarrow \underline{z}(p)+1\right]$. 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. In Figure 5, p_2 is the only process that can execute this type of transition.

We use $c \rightarrow_{SB} c'$ to denote that $c \xrightarrow{t}_{SB} c'$ for some $t \in \Delta \cup \Delta'$.



The set $\texttt{Init}_{SB}$ of *initial* SB-configurations (the figure on the right is an example of an initial SB-configuration) contains all configurations of the form $\left(\underline{q_{init}},b_{init},\underline{z_{init}}\right)$ 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 $(C_{SB},\texttt{Init}_{SB},\rightarrow_{SB})$.

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

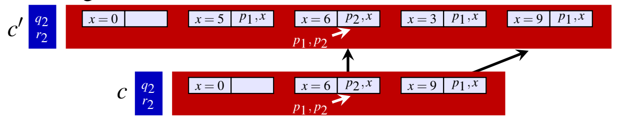
THEOREM 4.1. *For a concurrent program $\mathcal{P}$ and a local state definition* `Target`*, the reachability problems are equivalent under the TSO and SB semantics.*

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) that the SB-transition relation $\rightarrow_{SB}$ is monotonic wrt. $\sqsubseteq$. The algorithm performs backward reachability analysis from the set of target configurations. 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. 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. In the algorithm, we use a variant of finite-state automata, called *SB-automata*, as a symbolic representation of (potentially infinite) sets of SB-configurations. Assume a concurrent program $\mathcal{P} = (P, A)$.

Ordering For an SB-configuration $c = \left(\underline{q}, b, \underline{z}\right)$ we define $\texttt{ActiveIndex}(c) := \min\left\{\underline{z}(p) \mid p \in P\right\}$. 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. For example, in Figure 5, the active index is 1, i.e., all of its buffer messages are still alive.

Let $c = \left(\underline{q}, b, \underline{z}\right)$ and $c' = \left(\underline{q}', b', \underline{z}'\right)$ 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 the fact that the *active* part of b is a *sub-word* of the *active* part of b'. The third condition ensures that the last write indices wrt. all processes and variables are consistent. The last condition ensures that each process points to identical elements in b and b'. Below is an example of two configurations c and c' such that $c \sqsubseteq c'$.



We get the following lemma from the fact that (i) the subword relation is a well-quasi ordering on finite words [19], and that (ii) the number of states and messages (associated with last write operations and pointers) that should be equal, is finite.

LEMMA 5.1. *The relation $\sqsubseteq$ is a well-quasi ordering on SB-configurations.*

The following lemma shows effective monotonicity 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 (among other things) implies the termination of the reachability algorithm. The effectiveness aspect will be used in the fence insertion algorithm (cf. Section 6).

LEMMA 5.2. *$\rightarrow_{SB}$ is effectively monotonic wrt. $\sqsubseteq$.*

Recall that the term *effective monotonicity* is defined in Section 3. The *upward closure* of a set C of SB-configurations is defined as $C{\uparrow} := \{c' \mid \exists c \in C, c \sqsubseteq c'\}$. The 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 $decoding(W) := \{decoding(w) \mid w \in W\}$. Below is a proper word corresponding to the buffer and pointers in Figure 5.

$\boxed{[\mathbf{x{=}0,y{=}0,*}], \{\mathbf{p_1}\}}$ $\boxed{[\mathbf{x{=}1,y{=}0,p_1,x}], \{\mathbf{p_3}\}}$ $\boxed{[\mathbf{x{=}2,y{=}0,p_1,x}], \{\mathbf{p_4}\}}$ $\boxed{[\mathbf{x{=}2,y{=}2,p_4,y}], \emptyset}$ $\boxed{[\mathbf{x{=}2,y{=}1,p_3,y}], \{\mathbf{p_2}\}}$

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. Examples of SB-automata can be found in Figure 6. 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 that is 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 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 5.3. *We can compute an SB-automaton A^{init} such that $L\left(A^{init}\right) = \texttt{Init}_{SB}$. For a set* `Target` *of local state definitions, we can compute an SB-automaton $A(\texttt{Target})$ such that $L(A(\texttt{Target})) := \left\{\left(\underline{q}, b, \underline{z}\right) \mid \underline{q} \in \texttt{Target}\right\}$.*

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

LEMMA 5.4. *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 $\texttt{Pre}_t(C) := \{c \mid \exists c' \in C, c \xrightarrow{t}_{SB} c'\}$ to denote the set of immediate predecessor

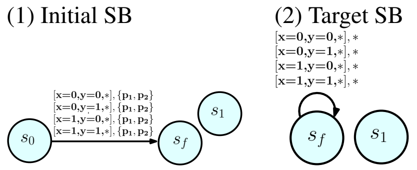


Figure 6. Typical examples of initial and target SB-automata. The system has processes p_1, p_2 and variables x, y ranging over $\{0,1\}$. In (1), h maps the initial local state definition to s_0 and others to s_1. In (2), h maps sets of local state definitions in `Target` to s_f and others to s_1. Properness ensures the correctness of the language of (2). Notice that the only final state s_f is not reachable from s_1. The symbol $*$ denotes any possible combination of a process/variable pair or set of processes in the system.

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 5.5. *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, 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) $A_i = (\texttt{Pre}_t(A_{i+1}))\uparrow$ for all $i : 1 \leq i < n$ (even if $L(A_{i+1})$ is upward-closed, it is still possible that $L(\texttt{Pre}_t(A_{i+1}))$ is not upward-closed; however due to monotonicity taking upward closure does not affect exactness of the analysis.); and (iii) $A_n = A(\texttt{Target})$. 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. Initially, $A^{\mathcal{V}}$ is an automaton recognizing the empty language, and $\mathcal{W}$ is the singleton $\{A(\texttt{Target})\}$. In other words, we start with a single trace containing the automaton representing configurations induced by `Target` (can be constructed by Lemma 5.3). 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 5.3). 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 $\texttt{Pre}_t(A)$ of A wrt. t, and then taking the upward closure (Lemmata 5.4 and 5.5). Notice that since we take the upward closure of the generated automata, and since $A(\texttt{Target})$ accepts an upward closed set, then $A^{\mathcal{V}}$ and all

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  $\mathcal{W} \leftarrow \{A(\texttt{Target})\}$;
2  $A^{\mathcal{V}} \leftarrow A^{\emptyset}$;
3  while $\mathcal{W} \neq \emptyset$ do
4      Pick and remove a trace $\delta$ from $\mathcal{W}$;
5      $A \leftarrow head(\delta)$;
6      if $L(A \cap A^{init}) \neq \emptyset$ then return $\delta$;
7      if $L(A) \subseteq L(A^{\mathcal{V}})$ then discard $A$;
8      else
9          $\mathcal{W} \leftarrow \{\delta' \in \mathcal{W} \mid L(head(\delta')) \not\subseteq L(A)\} \cup$
           $\{(\texttt{Pre}_t(A))\uparrow \cdot t \cdot \delta \mid t \in \Delta\}$;
10         $A^{\mathcal{V}} \leftarrow A^{\mathcal{V}} \cup A$
11 return "unreachable";
```

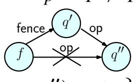
the automata added to $\mathcal{W}$ accept upward closed sets. The algorithm terminates when $\mathcal{W}$ becomes empty.

THEOREM 5.6. *The reachability algorithm always terminates returning the correct answer.*

6. Fence Insertion

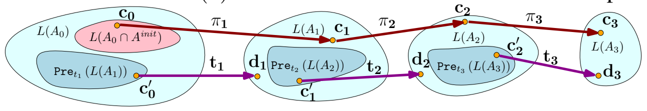
In this section we describe our fence insertion algorithm. The algorithm builds sets of fences successively using the counter-examples that are generated by the reachability algorithm. The algorithm is parameterized by a predefined *placement constraint* which is a subset G of Q, and will place fences only in local states that belong to G. For any given G, the algorithm finds the set of minimal sets of fences that are sufficient for correctness, if any. The algorithm uses different operations. First, we will show how to use a trace δ to derive a *counter-example*: an SB-computation that reaches `Target`. From the counter-example, we will 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 from the behavior of the program. Finally, we introduce the fence insertion algorithm. In the following, we fix a placement constraint G.

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 f in $\mathcal{P}$ (we insert a fence operation just after the local state f). 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 $\Delta'_p = \Delta_p \cup \{(f, \mathsf{fence}, q')\} \cup \{(q', op, q'') \mid (f, op, q'') \in \Delta_p\} \setminus \{(f, op, q'') \mid (f, op, q'') \in \Delta_p\}$. In other words, we add a new local state q', place a fence from f to q' and make all previously outgoing transitions from f start from q' instead. For a set $F = \{f_1, \ldots f_n\} \subseteq Q$ of local states, we let $\mathcal{P} \oplus F := \mathcal{P} \oplus f_1 \cdots \oplus f_n$. We say F is *minimal* wrt. a set `Target` of local state definitions and a placement constraint G if $F \subseteq G$ and $Reachable(SB)(\mathcal{P} \oplus F \setminus \{f\})(\texttt{Target})$ holds for all $f \in F$ but not $Reachable(SB)(\mathcal{P} \oplus F)(\texttt{Target})$. That is, all fences are in G, and the removal of any fence makes `Target` reachable. We use $F_{min}^{G}(\mathcal{P})(\texttt{Target})$ to denote the set of minimal sets of fences in $\mathcal{P}$ wrt. `Target` that respect the placement constraint G. Notice that $F_{min}^{G}(\mathcal{P})(\texttt{Target}) = \emptyset$ if $\texttt{Target}$ is reachable even if fences are placed at all locations in G.

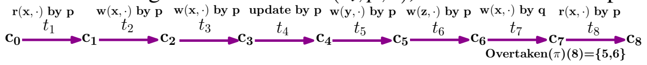
Counter-Example Generation Consider a trace $\delta = A_0 t_1 A_1 t_2 \cdots t_n A_n$. We show how to derive a counter-example from δ. Formally, a counter-example is a run $c_0 \xrightarrow{t_1}_{SB} c_1 \xrightarrow{t_2}_{SB} \cdots \xrightarrow{t_m}_{SB} c_m$ of the transition system induced from $\mathcal{P}$ under the SB semantics, where $c_0 \in \texttt{Init}_{SB}$ and $c_m \in \left\{ \left(\underline{q}, b, \underline{z}\right) \mid \underline{q} \in \texttt{Target} \right\}$. We assume a function *choose* that, for each automaton A, chooses a member of $L(A)$ (if $L(A) \neq \emptyset$), i.e., $choose(A) = w$ 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 $i : 1 \leq i \leq n$. Define $c_0 := choose\left(A_0 \cap A^{init}\right)$. 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 $i : 1 \leq i < n$. Since $A_i = \texttt{Pre}_{t_{i+1}}(A_{i+1})\uparrow$ and $c_i \in L(A_i)$, there exist $c_i' \in \texttt{Pre}_{t_{i+1}}(A_{i+1}) \subseteq L(A_i)$ 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 5.2, we know that 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 $\texttt{CounterEx}(\delta)$ to denote such a π. Below is an example.



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 then derive a set of fences such that any one of them will forbid 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. For example, if the 2nd message in b_5 is generated by the transition t_3, then $\alpha_5(2) = 3$. Below we explain how to generates those α functions. The first message $b_i(1)$ in each buffer represents the initial state of the 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 the transitions associated to all the messages have already 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 will 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 $\texttt{Overtaken}$ such that, for each $i : 1 \leq i \leq n$, if t_i is a read transition then the value $\texttt{Overtaken}(\pi)(i)$ 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 then define $\texttt{Overtaken}(\pi)(i) := \emptyset$. Otherwise, assume that $t_i = (q, \mathsf{r}(x,v), q') \in \Delta_p$ for some $p \in P$. We have $\texttt{Overtaken}(\pi)(i) := \left\{ \alpha_i(j) \mid \texttt{LastWrite}(c_i, p, x) < j \leq n_i \wedge t_{\alpha_i(j)} \in \Delta_p \right\}$. 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 $\texttt{LastWrite}(c_i, p, x)$). Below is an example.



Finally, we notice that, for each $i : 1 \leq i \leq n$ and each $j \in \texttt{Overtaken}(\pi)(i)$, 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 $\texttt{Barrier}(G)(\pi) := \left\{ \underline{q}_k(p) \mid \exists i : 1 \leq i \leq n.\ \exists j \in \texttt{Overtaken}(\pi)(i).\ j \leq k < i \right\} \cap G$.

Algorithm We present our fence insertion algorithm (Algorithm 2). It inputs a concurrent program $\mathcal{P}$, a placement constraint G, and a finite set $\texttt{Target}$ of local state definitions, and returns all minimal sets of fences ($F_{min}^{G}(\mathcal{P})(\texttt{Target})$). 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 is sufficient for making it correct).

Algorithm 2: Fence Inference

```
input : A concurrent program $\mathcal{P}$, a placement constraint $G$,
        and a finite set Target of local state definitions.
output: $F_{min}^{G}(\mathcal{P})(\texttt{Target})$.
1  $\mathcal{W} \leftarrow \{\emptyset\}$;
2  $\mathcal{C} \leftarrow \emptyset$;
3  while $\mathcal{W} \neq \emptyset$ do
4      Pick and remove a set $F$ from $\mathcal{W}$;
5      if $Reachable(SB)(\mathcal{P} \oplus F)(\texttt{Target}) = \delta$ then
6          $F_B \leftarrow \texttt{Barrier}(G)(\texttt{CounterEx}(\delta))$;
7          if $F_B = \emptyset$ then
8              return $\emptyset$
9          else foreach $f \in F_B$ do
10             $F' \leftarrow F \cup \{f\}$;
11             if $\exists F'' \in \mathcal{C} \cup \mathcal{W}.\ F'' \subseteq F'$ then discard $F'$;
12             else $\mathcal{W} \leftarrow \mathcal{W} \cup \{F'\}$
13     else
14         $\mathcal{C} \leftarrow \mathcal{C} \cup \{F\}$
15 return $\mathcal{C}$;
```

The algorithm uses two sets of sets of fences, namely $\mathcal{W}$ for those that have been *partially* constructed (but yet not large enough to make the program correct), and $\mathcal{C}$ for those that are *completed* (shown to be sufficient for making the program correct). The union of the sets $\mathcal{C}$ and $\mathcal{W}$ maintains the invariant that its elements are mutually incomparable wrt. set inclusion. During each iteration, a set F is picked and removed from $\mathcal{W}$. We use Algorithm 1 to check whether the set F is sufficient to make the program correct. If *yes*, we add F to $\mathcal{C}$. If *no*, we know that Algorithm 1 will return a trace δ. We derive a counter-example from δ from which we compute the barrier set (as described above). In such a way we obtain a (possibly empty) set of fences such that at least one member f of the set is necessary (but perhaps not enough) to add to F in order to make the program correct. Therefore, for each such f we add $F' = F \cup \{f\}$ back to $\mathcal{W}$ unless there is already a subset of F' in the set $\mathcal{C} \cup \mathcal{W}$.

THEOREM 6.1. *For a concurrent program $\mathcal{P}$, a placement constraint G, and a finite set* $\texttt{Target}$*, Algorithm 2 terminates and returns* $F_{min}^{G}(\mathcal{P})(\texttt{Target})$.

	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 [40]	2/8/2/10	0.02	0.02	1 per process	1
2. Full Dekker [13]	2/14/3/18	0.28	0.28	1 per process	1
3. Peterson [36]	2/10/3/14	0.24	0.6	1 per process	1
4. Lamport Bakery [25]	2/22/4/32	52	5538	2 per process	4
5. Lamport Fast [27]	2/26/4/38	6.5	6.5	2 per process	1
6. CLH Queue Lock[32]	2/48/4/60	26	26	0	1
7. Sense Reversing Barrier [33]	2/16/2/24	1.1	1.1	0	1
8. Burns [31]	2/9/2/11	0.07	0.07	1 per process	1
9. Dijkstra [31]	2/14/3/24	9.5	10	1 per process	1
10. Tournament Barriers [18]	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

Remark One can show that 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 evaluate our approach on several benchmark examples and some difficult problem sets that cannot be handled by *any previous algorithms*. 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 is set so that fences may only be placed 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 Lamport Bakery there are four minimal fence sets (two ways of inserting two fences per process). For all other experiments, there is one or zero minimal fence set(s). 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 of the above examples, which is beyond the capabilities of previous approaches. In particular, none of our examples is data-race free (in fact some even contain triangular data-races). 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 [24] and in [29] are potentially able to handle such general classes of problems. However, the approach of [29] does not guarantee termination. The work in [24] 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). According to [24], they cannot infer optimal fences for Lamport Bakery algorithm [25], Lamport Fast Mutex [27], and CLH Queue Lock [32]. Our tool handles all these examples and finds optimal fence sets. We refer the reader to the appendix for further details of the implementation and the examples.

8. Related Work

To our knowledge, our approach is the first sound and complete automatic fence insertion method for finite-state concurrent programs running under TSO; it generates the *minimal* sets of fences that make the program correct or proves the program incorrect even under the sequential consistent memory model. Existing approaches for automatic fence insertion under relaxed memory models can be categorized into the following types.

Over-Approximations The work [24] uses abstractions to over-approximate reachable state spaces. To be more specific, the method records exactly the first k buffer messages and stores messages with indices larger than k in sets (i.e., the order between messages is discarded). By increasing the size of k, more accurate abstractions are obtained at the cost of larger and heavier abstract domains. Comparing to our approach, [24] might produce false counterexamples and thus may introduce unnecessary fences.

Exact State Space Exploration with Acceleration Techniques In [29, 30], automata are used as abstract representations of the store buffers. When loops are encountered in a program, acceleration techniques are used to speed up the calculation of reachable states. In contrast to [24], the abstraction does not over-approximate reachable state spaces and thus will not generate false counterexamples. However, the method is in general not guaranteed to terminate, while termination is guaranteed in our case. Furthermore, a counter-example guided approach is added to infer fences. However, the method does not have the concept of "placement constraints" and might for instance put fences at locations that might be frequently executed.

Decidability Results It has been shown in [5] that state reachability for TSO is decidable, but highly complex (it has a non primitive recursive complexity). Decidability and complexity have been established by showing reductions to/from lossy channel systems. However, the reduction of [5] involves nondeterministically guessing the buffer content, which introduces a serious state space explosion problem. We have implemented this approach and found that in practice it even cannot verify the simplest examples. Moreover, [5] did not discuss fence insertion. An important contribution of our work is the introduction of a single buffer semantics whose reachability problem is equivalent to the TSO one. Restricting the semantics to a single buffer is crucial for avoiding the immediate state space explosion. This allows us to automatically infer fences for several challenging programs.

Data-Race Freedom Assumption It has been shown that data-race free programs do not have non-sequential consistent behaviors under relaxed memory models [37] and thus there is no need to insert fences for this type of programs. Those programs can be verified using approaches assuming a sequential consistent memory model. For the TSO memory model, a weaker property [34] (called *triangular data-race freedom*) has been proposed which also ensures that program behaviors are identical under TSO and SC. Nevertheless, considering only data-race free programs might be unrealistic since many important classes of programs such as mutual exclusion and look-free algorithms come in nature with data races. Our approach is able to handle programs without using assumptions about data-race freedom.

Removal of Sequential Inconsistent Behaviors One possible way to make programs correct under relaxed memory models is to remove all non-sequential consistent behaviors, e.g., by inserting fences. Then the rest of the program behavior is verified using algorithms for a sequential consistent memory model. The major drawback of these approaches is that they overlook the existence of "harmless" non-sequential consistent behaviors. Indeed, these approaches will remove sequential inconsistent behaviors even though they will not violate the desired correctness properties. Previous works that fall in this category include those using monitors [4, 10, 12], compiler techniques [15, 28], and explicit state model checking [20]. None of these approaches can guarantee to generate minimal sets of fences to make programs correct because they also remove benign sequential inconsistent behaviors.

Under-approximations Under-approximations are achieved by, e.g., bounding the size of the buffers [23] or the number of contextswitches [6], testing [11], and bounded model checking [8, 9]. These approaches are good for detecting bugs. However, in general they cannot verify the correctness of a program under a relaxed memory model. In order words, they cannot tell whether the inserted set of fences is sufficient for the program to be correct.

9. Conclusions, Discussion, and Future Work

We have presented a sound and complete method for automatic fence insertion in finite-state programs running under the TSO memory model. Our approach is based on a simple and effective backward reachability analysis for a novel semantics, namely the SB model (that we show to be equivalent to TSO for reachability problems). We have proposed a counter-example guided fence insertion procedure to infer all minimal sets of fences that ensure program correctness. The procedure is augmented by a placement constraint that defines the program states in which fences may be inserted. We have implemented our approach in a prototype and applied it to several challenging examples. In particular, we have verified several examples that cannot be handled by existing approaches. We have identified several direction for future work:

Other Weak Memory Models To make our presentation accessible, we described our framework based on the TSO semantics. However, the framework can be easily extended to several other memory models, e.g. the *PSO* model considered in [5, 24]. This memory model is weaker than TSO in the sense that reads still overtake writes on different variables as in the case of TSO; and in addition, also writes may overtake writes on other variables. The operational semantics for PSO is defined by associating a set of unbounded buffers to each process, namely one for each variable [5, 24]. In the case of this model, the transition relation is monotonic with respect to an ordering on buffers similar to the one we have defined in this paper (the sub-word relation taking into consideration last-write operations). This means that our framework can be re-applied to the case of PSO. In fact, since the transition relation is monotonic wrt. the defined ordering, the instantiation of our framework will be easier, since there is no need to go through the SB semantics. It is worth noticing however that the model of [5, 24] does not include the *sfence* instruction (defined in the SPARC architecture [40]) that prevents the re-ordering of write operations before and after the fence (such a fence is not relevant for TSO, since TSO does not allow re-ordering of write operations in the first place).

Infinite Data and Control We have applied our method to finite-state programs (a finite number of finite-state processes operating on finite data). We intend to integrate counter-example guided abstraction refinement (CEGAR) with our framework to deal with variables ranging over unbounded domains. The CEGAR loop will then contain two loops, one for refining data predicates and one for refining sets of fences. We also plan to consider programs with unbounded control structures. In particular, we will consider *parameterized verification* in the context of weak memory models. This would enable us for instance to analyze mutual exclusion protocols (e.g., Burn's protocol in Section 2) for an arbitrary number of processes (rather than a fixed number as we do in this paper).

Acknowledgments

We are grateful to Peter Sewell and to Mark D.Hill for pointing us to the formal definitions of the operational semantics of the Total Store Ordering memory model.

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] S. Adve and M. D. Hill. Weak ordering - a new definition. In *ISCA*, 1990.

[4] J. Alglave and L. Maranget. Stability in weak memory models. In *CAV*, 2011.

[5] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In *POPL*, 2010.

[6] M. F. Atig, A. Bouajjani, and G. Parlato. Getting rid of store-buffers in TSO analysis. In *CAV*, 2011.

[7] D. Brand and P. Zafiropulo. On communicating finite-state machines. *JACM*, 30(2), 1983.

[8] S. Burckhardt, R. Alur, and M. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *PLDI*, 2007.

[9] S. Burckhardt, R. Alur, and M. M.K. Martin. Bounded model checking of concurrent data types on relaxed memory models: A case study. In *CAV*, 2006.

[10] S. Burckhardt, R. Alur, and M. Musuvathi. Effective program verification for relaxed memory models. In *CAV*, 2008.

[11] J. Burnim, K. Sen, and C. Stergiou. Testing concurrent programs on relaxed memory models. Technical Report UCB/EECS-2010-32, UCB, 2010.

[12] J. Burnim, K. Sen, and C. Stergiou. Sound and complete monitoring of sequential consistency in relaxed memory models. In *TACAS*, 2011.

[13] E. W. Dijkstra. *Cooperating sequential processes*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[14] M. Dubois, C. Scheurich, and F. A. Briggs. Memory access buffering in multiprocessors. In *ISCA*, 1986.

[15] X. Fang, J. Lee, and S. P. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *ICS*. ACM, 2003.

[16] A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *TCS*, 256(1-2), 2001.

[17] K. Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, 2004.

[18] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *IJPP*, 17, February 1988.

[19] G. Higman. Ordering by divisibility in abstract algebras. *Proc. London Math. Soc. (3)*, 2(7), 1952.

[20] T.Q. Huynh and A. Roychoudhury. A memory model sensitive checker for C#. In *FM*, 2006.

[21] IBM, editor. *Power ISA v.2.05*. 2007.

[22] Intel Inc. Intel^{TM}64 and IA-32 Architectures Software Developer's Manuals.

[23] M. Kuperstein, M. Vechev, and E. Yahav. Automatic inference of memory fences. In *FMCAD*, 2011.

[24] M. Kuperstein, M. Vechev, and E. Yahav. Partial-coherence abstractions for relaxed memory models. In *PLDI*, 2011.

[25] L. Lamport. A new solution of dijkstra's concurrent programming problem. *CACM*, 17, August 1974.

[26] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comp.*, C-28(9), 1979.

[27] L. Lamport. A fast mutual exclusion algorithm, 1986.

[28] J. Lee and D. A. Padua. Hiding relaxed memory consistency with a compiler. *Computers*, 50(8), 2001.

[29] A. Linden and P. Wolper. An automata-based symbolic approach for verifying programs on relaxed memory models. In *SPIN*, 2010.

[30] A. Linden and P. Wolper. A verification-based approach to memory fence insertion in relaxed memory systems. In *SPIN*, 2011.

[31] N. Lynch and B. Patt-Shamir. DISTRIBUTED ALGORITHMS , Lecture Notes for 6.852 FALL 1992. Technical report, MIT, Cambridge, MA, USA, 1993.

[32] P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *IPPS*. IEEE Computer Society, 1994.

[33] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9, February 1991.

[34] S. Owens. Reasoning about the implementation of concurrency abstractions on x86-tso. In *ECOOP*. 2010.

[35] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-tso. In *TPHOL*, 2009.

[36] G. L. Peterson. Myths About the Mutual Exclusion Problem. *IPL*, 12(3), 1981.

[37] V. A. Saraswat, R. Jagadeesan, M. M. Michael, and C. Praun. A theory of memory models. In *PPOPP*, 2007.

[38] 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.

[39] J. E. Smith and A. R. Pleszkun. Implementation of precise interupts in pipelined processors. In *25 Years ISCA: Retrospectives and Reprints*, 1998.

[40] D. Weaver and T. Germond, editors. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall, 1994.

A. Examples

We present in the following the examples we used for testing our approach. Unless otherwise specified, all examples are correct under SC semantics and the initial values of variables are set to 0. All of the examples are not data-race free. Typically, we have applied our approach on systems with two processes. For a variable v, we write $v[i]$ to mean a (possibly shared) variable associated with process i. Local variables are prefixed with an underscore. In our experiments, we used targets of write operations as a placement constraint. We make explicit in the code the fences automatically introduced by our tool. Here, we consider correctness properties such as mutual exclusion for locks and mutex algorithms, and synchronization for barrier algorithms.

Dekker algorithm: We experimented with two versions of the classical Dekker algorithm for mutual exclusion. First (see Figure 7) the simplified version (mentioned in the manual of the SPARC architecture [40]) uses one boolean $flag$ per process i. Each process uses $flag$ to notify the other process of its interest in accessing the critical section. A process only accesses the critical section if the other one is not interested. Our analysis finds derives a fence after the write at line 2 and proves the resulting program respects mutual exclusion under TSO.

```
  // Process[i : {0,1}]
1 while true
2   store flag[i]=1;
3   fence;
4   load _flag=flag[1-i];
5   if _flag=1
6     store flag[i]=0;
7     goto 2;
8   //CS;
9   store flag[i]=0;
```

Figure 7. A simple version of the Dekker's algorithm for ensuring mutual exclusion among two processes.

The second version (see Figure 8) is starvation free. This is achieved with the addition of a variable $turn$ (initially set to 0 or 1) that gives alternatively the priority to each of the two processes. Again, our analysis correctly inserts one fence per process after the write at line 2, and proves the resulting program correct under TSO.

The algorithm can be equivalently implemented by replacing the **goto** at line 11 by a `store flag[i]=1` and a fence. This alternative implementation requires two fences per process, but there will be no difference in the number of times a fence is actually executed in order to access the critical section.

```
   // Process[i : {0,1}]:
1  while true
2    store flag[i]=1;
3    fence;
4    load _flag=flag[1-i];
5    while _flag==1
6      load _turn=turn;
7      if _turn != i
8        store flag[i]=0;
9        while _turn != i
10         load _turn=turn;
11       goto 2;
12     load _flag=flag[1-i];
13   //CS
14   store turn=1-i;
15   flag[i]=0;
```

Figure 8. Dekker's algorithm without deadlock for mutual exclusion with two processes.

Peterson algorithm: This algorithm makes use of a boolean $flag$ per process in addition to a $turn$ variable (initially set to 0 or 1) to ensure mutual exclusion (see Figure 9). The $flag$ is used to notify the other process of the interest in accessing the critical section. Each process systematically gives priority to the other process by setting the $turn$ variable. In case of a tie, the last process to set the variable $turn$ loses. Our analysis correctly inserts a fence after the write at line 3 and proves the resulting program respects mutual exclusion under TSO.

```
   // Process[i : {0,1}]:
1  while true
2    store flag[i] = 1;
3    store turn = 1-i;
4    fence;
5    do
6        load _flag = flag[1-i];
7        load _turn = turn;
8    while _flag=1 & _turn=1-i;
9    //CS;
10   store flag[i]=0;
```

Figure 9. Peterson's algorithm for mutual exclusion with two processes

Lamport's Bakery algorithm: In this mutual exclusion algorithm, each process that wants to access the critical section needs to choose a number that is strictly larger than the numbers of all other processes (see Figure 10). The process with the smallest number is allowed to enter the critical section. In case of a tie, the algorithm uses the identifiers to define priorities. In addition, a boolean variable c per process is used to notify other processes of whether the concerned process is choosing its number. Without making the test on c at lines 13-15, a process p might access its critical section based on the old value of the other process q, when in fact q is about to choose the same value as p for its number. This would violate mutual exclusion if the slower process q has higher priority when it reaches line 17. We bounded possible values for the number variables by 2, and checked whether mutual exclusion is respected under TSO. Our approach automatically infers four minimal sets of fences, and proves the resulting programs respect mutual exclusion under TSO. These sets correspond to placing two fences per process, one at line 3 and the other at either line 10 or line 12.

```
   // Process[i : {0,1}]:
1  while true
2    store c[i]=1;
3    fence;
4    load _n[1-i] = n[1-i];
5    if _n[1-i]==2
6      store c[i]=0
7      goto 2;
8    _n[i]=1+_n[1-i];
9    store n[i]=_n[i];
10   fence; // Alternative 1
11   store c[i]=0;
12   fence; // Alternative 2
13   do
14     load _c = c[1-i];
15   while _c != 0;
16   do
17     load _n[1-i]=n[1-i];
18   while     _n[1-i]!=0
19     & (_n[1-i] <_n[i]
20        | (_n[1-i] ==_n[i]
21          & i < 1-i ));
22   //CS;
23   store n[i]=0;
```

Figure 10. Lamport's Bakery algorithm for mutual exclusion. We analyzed an instance with two processes.

Fast Lamport: The algorithm focuses on having a constant number of reads and writes when there is no contention, as opposed to other parameterized algorithms that require a linear amount of memory accesses. The processes share an array of N variable and two variables x and y. Our analysis derives two fences (per process) after the writes at lines 3 and 11, and proves the resulting program safe with respect to mutual exclusion under TSO.

Burns algorithm: This mutual exclusion algorithm assumes the (arbitrarily many, here two) processes to be ordered according to their identifiers (see Figure 12). It uses one boolean $flag$ per process. Each process performs three global reads to check that the flags associated to the other processes are 0 in order to access its critical section. More precisely, a process performs two global reads to the right, and one to the left. If the process sees that another process to the right has a $flag$ equal to one, it gives up by restarting and setting its flag to 0. It however does not give up if it sees a process to the right with $flag$ equals one and waits for it instead. As a result, the code is different for process 0 and process 1 as the first process does not have any process to the left, while the second does not have any process to the right (see Figure 12). Our analysis infers two fences (one per process) after the writes at line 2 for

```
   // Process[i : {1,2}]:
1  while true
2    store b[i] = 1;
3    store x = i;
4    fence;
5    load _y = y;
6    if _y != 0
7      store b[i] = 0;
8      while _y!=0
9        load _y = y;
10     goto 1;
11   store y=i;
12   fence;
13   load _x = x;
14   if _x!=i
15     store b[i]=0
16     do
17       load _b = b[3-i];
18     while _b != 0
19     load _y = y;
20     if _y!=i
21       while _y!=0
22         load _y=y;
23       goto 1;
24   //CS;
25   store y=0;
26   sotre b[i]=0;
```

Figure 11. Lamport's Fast algorithm. We analyzed an instance with two processes.

process one and line 6 for process 2. The analysis proves mutual exclusion for the resulting program under TSO.

```
   // process[0]:
1  while true
2    store flag[0]=1;
3    fence;
4    load _flag=flag[1];
5    if _flag==1
6      goto 4;
7    //CS
8    store flag[0]=0;

   // process[1]:
1  while true
2    store flag[1]=0;
3    load _flag=flag[0];
4    if _flag==1
5      goto 2;
6    store flag[1]=1;
7    fence;
8    load _flag=flag[0];
9    if _flag==1
10     goto 2;
11   // CS
12   store flag[1] = 0;
```

Figure 12. Burns mutual exclusion algorithm instantiated for two processes

Dijkstra algorithm: The algorithm uses one $flag$ with values in $\{0,1,2\}$ per process, and one global variable $turn$ with values in the set of identifiers of the arbitrary many processes union the singleton $\{0\}$ (here there are two processes with id 1 and 2, so $turn$ has the domain $\{0,1,2\}$, see Figure 13). A process uses the $turn$ variable to gain access to the critical section by setting the variable to its id each time the process with $turn$ as id has released the critical section (by resetting $turn$ to 0). Among those that succeeded in setting turn to their id, a simple Dekker is used to ensure that exactly one of them accesses the critical section. Our analysis infers one fence per process after the write at line 9, and proves mutual exclusion is respected by the resulting program on TSO.

```
   // process[i : {1,2}]:
1  while true
2    store flag[i] = 1;,
3    load _turn = turn;
4    while _turn != i
5      load _flag=flag[_turn];
6      if _flag = 0
7        store turn = i;
8      load _turn=turn
9    store flag[i]=2;
10   fence;
11   load _flag=flag[3-i];
12   if _flag = 2
13     goto 2;
14   //CS;
15   store flag[i]=0;
```

Figure 13. Dijkstra mutual exclusion algorithm instantiated for two processes.

CLH queue lock: Our model of the CLH queue-based locking algorithm is presented in Figure 14. We consider in our model two processes competing for the lock and using a memory modeled as a shared array mem with three boolean cells. In addition, the processes share the variable $lock$ used as an index of mem. Each process owns two local variables $\_i$ and $\_p$ that take their values in $\{0, ..., |mem| - 1\}$. Initially both local variables $\_i$ and $\_p$ of each

```
   // a process:
1  while true
2    store mem[_i] = 1;
3    load _lock=lock;
4    arw(lock, _lock, _p);
5    _p=_lock;
6  do
7      load _mp=mem[_p]
8    while _mp != 0;
9    //CS;
10   store mem[_i]=0;
11   _i=_p;
```

Figure 14. CLH Queue Locking algorithm. We analyzed an instance with two processes sharing a memory array of three cells and an index lock.

process are equal but different from the local variables of the other process.

A process that wants to grab the lock starts by setting *mem*[*_i*] to 1 (line 2). Then, the process puts itself in the queue and points to the boolean flag held by the previous process in the queue using an atomic operation involving flushing its write buffer and swapping the values of *lock* and *_p*. We simulate this operation using lines 3-5. Observe that the resulting code may block as *lock* may change from line 3 to line 4; this however only adds blocking behaviors which preserves exactness of the reachability analysis. The process waits then for its turn at line 8, and releases the lock by resetting its flag to 0. Our analysis showed there was no need to introduce fences in order for the algorithm to ensure mutual exclusion under TSO.

Sense reversing tournament barrier: This algorithm ensures barrier synchronization for N processes in phases of $log_2(N)$ rounds. The algorithm proceeds in two phases: arrival and wakeup, each represented by a loop involving $log_2(N)$ rounds where processes, at each round, are statically divided into winners and losers. In the arrival phase, losers notify winners they are waiting for them, and winners qualify to the next round where half of them become losers. The unique winner at the last round is declared champion. In the wakeup phase, the champion wakes up its loser, and at each round, each winner wakes up the corresponding loser. For two processes, the algorithm boils down to the one presented in Figure 15.

```
   // process[0]: champion           // process[1]: loser
1  while true                     1  while true
2    // epoch i                    2    // epoch i
3    do                           3    store flag[0] = sense;
4      load _flag = flag[0];      4    do
5    while _flag != _sense;       5      load _flag = flag[1];
6    store flag[1]=_sense;        6    while _flag != _sense
7    _sense = !_sense;            7    _sense = !_sense;
8    // epoch i+1                 8    // epoch i+1
```

Figure 15. Sense reversing tournament barrier instantiated for two processes.

We applied our analysis and showed correctness (here that no process crosses the barrier if the other is still in the previous epoch) without the need to insert fences on TSO.

Centralized sense-reversing barrier This barrier algorithm boils down to Figure 16 for two processes. In the general case, it synchronizes N processes using a global counter *cnt* and a shared boolean *sense*. The counter *cnt* is initially set to 2, the number of processes in the system. Each process that encounters the barrier atomically fetches and decreases the value of *cnt*. We simulate this operation with the possibly blocking lines 4-5 (see swap operation in the CLH algorithm above). Thereafter, the process waits for the value of *sense* to change. The last process that decreased *cnt* sets back the value of *cnt* to 2 and releases all other processes by changing the value of *sense*. We applied our analysis to the system and verified its correctness (that no process crosses the barrier when the other is still in the previous epoch) without the need to insert fences under TSO.

```
   // a process :
1  while true
2    // epoch i
3    load _sens = sens;
4    load _cnt = cnt;
5    arw(cnt,_cnt,_cnt-1);
6    if _cnt == 1
7      store cnt = 2;
8      store sens= not _sens;
9    else
10     do
11       load _rel = sens;
12     while _rel = _sens;
13   // epoch i+1
```

Figure 16. A centralized sens reversing barrier instantiated for two processes.

A Task Scheduling Algorithm. Figure 17 is a task scheduling algorithm. The client repeatedly executes two tasks in order and the arbiter checks if there exists sufficient resources to execute the tasks. The arbiter grants permission by copying the allowed task number from variable *req* to variable *turn*. We add an assertion in line 11 and verify it to ensure that in the next iteration, the client will not execute the task before the arbiter checks the resources. Our analysis proves the assertion is respected under TSO without the need for fence insertion.

```
   // arbiter:
1  while true
2    load _req=req;
3    //check resource
4    store turn=_req;

   // client[i : {0,1}]
1  while true
2    do
3      store req=i*2;
4      load _turn=turn;
5    while _turn≠ i*2;
6    do
7      store req=i*2+1;
8      load _turn=turn;
9    while _turn≠ i*2+1;
10   load _turn=turn;
11   assert _turn≠i*2;
```

Figure 17. A Task Scheduling algorithm. We analyzed an instance with two clients.

Overwriting Producer Consumer Figure 18 shows a producer-consumer algorithm. The processes share an N element array, *arena*. The producer process continually stores resources to the shared array (by setting the values of array elements to 1), which are then consumed by the consumer process (by setting the values back to 0) in a cyclic manner. The producer is allowed to overtake the consumer, thereby replacing some existing resources with new ones. The consumer may not overtake the producer and consume non-existing resources. Therefore the consumer needs to keep track of the position of the producer through a shared variable *head*.

Our prototype analyzed the producer-consumer algorithm and found that it is erroneous even under sequential consistency and hence cannot be corrected only by insertion of fences. This was detected during the analysis of the first counter-example generated by the reachability analysis. The counter-example did not contain any instruction reorderings and thus respected sequential consistency.

```
   // producer:
1  _hd = 0;
2  while true
3    store arena[_hd]=1;
4    _hd = (_hd+1)%N;
5    store head = _hd;

   // consumer:
1  _tl = 0;
2  while true
3    load _hd = head;
4    if _hd != _tl
5      load _a = arena[_tl];
6      assert(_a == 1);
7      store arena[_tl] = 0;
8      _tl = (_tl+1)%N;
```

Figure 18. An Overwriting Producer-Consumer algorithm with one producer and one consumer. (Version 1)

Figure 19 shows a similar producer-consumer algorithm, where the producer creates two resources at a time, which are consumed one at a time by the consumer. In this case, the *assert* on line 6 cannot be violated, which is verified by our prototype. No fences are necessary for the correctness of this program under TSO.

```
// producer:
_hd = 0;
while true
  store arena[_hd]=2;
  _hd = (_hd+1)%N;
  store head = _hd;
```

```
// consumer:
_tl = 0;
while true
  load _hd = head;
  if _hd != _tl
    load _a = arena[_tl];
    assert(_a != 0);
    arw(arena[_tl],_a,_a-1);
    _tl = (_tl+1)%N;
```

Figure 19. An Overwriting Producer-Consumer algorithm with one producer and one consumer. (Version 2)

Alternating algorithm The algorithm in Figure 20 can be used to transmit values from one process to the other using shared variables (here *msg* and *ack*) in case these can be overwritten by other processes using other values than those used by the algorithm. Essentially, the algorithm simulates a version of the alternating bit protocol where overwritten shared variables play the role of unreliable channels. We want to check that a process can not start writing the next message or acknowledgment before the other process managed to read it; for example, the sender can not move from lines 2-5 to lines 6-9 without the receiver having moved from lines 2-5 to lines 6-9. For this, our analysis deduces that there is no need to introduce fences under TSO.

```
  // sender:
1 while true
2   do
3     store msg=0;
4     load _a=ack;
5   while _a != 0;
6   do
7     store msg=1;
8     load _a=ack;
9   while _a!=1
```

```
  // receiver:
1 while true
2   do
3     store ack=1;
4     load _m=msg;
5   while _m != 0;
6   do
7     store ack=0;
8     load _m=msg;
9   while _m!=1
```

Figure 20. Alternating bit protocol with two variables used for synchronization.

Increasing Sequence The algorithm in Figure 21 is a small but challenging example to many existing approaches. The server process may generate an arbitrary number of messages to the buffer. The client reads the value of *msg* twice and check if the value of the second read is not smaller than the first read. The *order* of messages sent to the buffer is important for proving the assertion at line 7; the value read in line 6 cannot be smaller than the one read in line 5. If an approach (e.g., [24]) abstracts away the order between buffer messages, then it cannot verify this example. Moreover, this example is not triangular data-race free. It allows the following *sequential consistent* execution; lines 5,6,1,2, and then line 3, which contains a triangular data-race. So it cannot be handled by approaches with data-race free assumptions. Our analysis concludes that the system is correct under TSO without any fence.

```
  // Server Process
1 for(_i = 1 to 20)
2   while(*)
3     store msg= _i;
```

```
  // Client Process
4 store msg = 0;
5 load _val1=msg;
6 load _val2=msg;
7 assert(_val1≤_val2);
```

Figure 21. Increasing Sequence.

B. Implementation Details

The algorithms described in this paper were implemented as a prototype using OCaml. A number of optimizations were implemented to enhance the performance of the algorithm. Below, we describe the most important ones (in no particular order):

- *Placement constraint:* Algorithm 2 describes our algorithm for fence insertion, where the allowed positions for fences are restricted by a placement constraint. The constraint we used in our experiments is that fences may only be placed immediately after writes. This corresponds to a fence method of the x86 architecture where writes are replaced by LOCK'd writes [22, 38] which forces a write buffer flush after LOCK'd writes are executed.
- *Augmented alphabet:* We let SB-automata operate over an *augmented alphabet*. The augmented alphabet is the same as that of the SB-buffer, except that individual values in the memory snapshot may be left undefined, with the interpretation that any value is possible for that variable. This allows us to avoid having to enumerate all possible values for a variable and proceed instead in a lazy manner when performing the backward reachability analysis.
- *Symmetry reduction:* Many of the concurrent algorithms and protocols analyzed in our experiments consist of symmetrical processes. The reachability analysis was adapted to exploit this by only searching one of the multiple symmetrical paths through the transition system.
- *Partial order reduction:* To reduce the number of equivalent paths in the transition system that are searched, we consider the timing of updates from store buffers to main memory. In the TSO semantics, an update may occur at any time. However, it is enough for the reachability analysis to only schedule an update (i) immediately after the write that generated the buffer message, or (ii) immediately after some later read by the same process. The reason is that scheduling the update at other locations does not change the behavior of the process that issued the buffer message (as it does not read variables in those locations); nor does it change the behavior of the other processes (as the updates will anyway take place after the write or after some read of the issuing process).
- *Combination with a rough forward analysis:* To reduce the amount of searching performed in the backward reachability analysis when adding messages to the single buffer, our tool first performs a rough forward reachability analysis. The invariants inferred by the forward analysis will then be used in the backward analysis to rule out some of the superfluous transitions. In the forward reachability analysis we use an over-approximative abstraction of TSO where the write buffers are represented as unordered sets of messages rather than as FIFO channels.

C. Reachability Equivalence: SB–TSO

In this section, we show equivalence of the reachability problems under the TSO and SB semantics. Assume a concurrent program $\mathcal{P} = (P, A)$.

We show the following theorem (which immediately implies the result.)

Theorem 4.1. *Reachable*$(SB)\,(\mathcal{P})\,(\mathtt{Target})$ *iff* *Reachable*$(TSO)\,(\mathcal{P})\,(\mathtt{Target})$ *for all local state definitions* $\mathtt{Target}$.

For a TSO-configuration $c = \left(\underline{q}, \underline{b}, mem\right)$, we use $\mathtt{StatesOf}\,(c)$, $\mathtt{BuffersOf}\,(c)$, and $\mathtt{MemoryOf}\,(c)$ to denote $\underline{q}$, $\underline{b}$, and *mem* respectively. For an SB-configuration $c = \left(\underline{q}, b, \underline{z}\right)$, we use $\mathtt{StatesOf}\,(c)$, $\mathtt{BufferOf}\,(c)$, and $\mathtt{PointersOf}\,(c)$ to denote $\underline{q}$, b, and $\underline{z}$ respectively.

From SB to TSO We show *only-if* direction of Theorem 4.1. Consider an SB-computation $\pi_{SB} = c_0 \xrightarrow{t_1}_{SB} c_1 \xrightarrow{t_2}_{SB} \cdots \xrightarrow{t_{n-1}}_{SB} c_{n-1} \xrightarrow{t_n}_{SB} c_n$. We will derive a

TSO-computation π_{TSO} such that $\texttt{StatesOf}(target(\pi_{TSO})) = \texttt{StatesOf}(c_n)$.

First, we show that we can assume that π_{SB} is of a particular form defined as follows. An SB-configuration c is said to be *balanced* if $\texttt{PointersOf}(c)(p) = \texttt{PointersOf}(c)(p')$ for all $p, p' \in P$. In other words, the pointers of the processes are all at the same position inside the buffer (i.e., the processes have all a consistent view of the memory). An SB-computation π is said to be *balanced* if its last configuration $target(\pi)$ is balanced. We can assume without loss of generality, that π_{SB} is balanced. The reason is that, in case π_{SB} is not balanced, we can continue from c_n and perform a sequence of update operations, until all the pointers point to the same position in the buffer. Notice that the configuration c'_n reached in this manner has the same local state definition as c_n.

Define $r := |\texttt{BufferOf}(c_n)|$. For a process $p \in P$ and an SB-message $a = (mem, p', x)$, we define $\texttt{SBtoTSO}(p)(a)$ to be the pair $(x, mem(x))$ if $p' = p$ and ε otherwise. For a word $w = a_1 a_2 \cdots a_n$ over SB-messages, we define $\texttt{SBtoTSO}(p)(w) := \texttt{SBtoTSO}(p)(a_1) \cdot \texttt{SBtoTSO}(p)(a_2) \cdot \cdots \cdot \texttt{SBtoTSO}(p)(a_n)$, i.e., we concatenate the results of applying the operation individually on each a_i.

Let $\prec$ be an arbitrary total order on the set p of processes. We use p_{min} and p_{max} to be the smallest resp. largest elements of $\prec$. For $p \neq p_{max}$, we define $succ(p)$ to be the successor of p wrt. $\prec$, i.e., $p \prec succ(p)$ and there is no p' with $p \prec p' \prec succ(p)$. We define $prev(p)$ for $p \neq p_{min}$ analogously.

The computation π_{TSO} consists of r phases (henceforth referred to as the phase $1, 2, \ldots, r$). At phase k, the computation π_{TSO} simulates the movements of the processes when their pointers are at position k in the buffer. The order in which the processes are simulated during phase k is defined by the ordering $\prec$. First, process p_{min} will perform a sequence of transitions. This sequence is identical to the sequence of transitions it performs in π_{SB} when its pointer is equal to k. Then, the next process performs its transitions. This continues until p_{max} has made all its transitions. When all processes have performed their transitions in phase k, phase $k+1$ starts by p_{min} executing its transitions, and so on. Formally, we define a "scheduling function" α that defines the order in which the processes run and the order in which they execute their transitions during the different phases. For $k : 1 \leq k \leq r$, $p \in P$, and $\ell \in \mathbb{N}$, the value of $\alpha(k, p, \ell)$ is a natural number j such that $0 \leq j \leq n$. We will use j to identify the point at which process p makes its ℓ^{th} move during phase k. These moves will be defined in terms of configurations and transitions inside π_{SB} whose indices are derived in a certain manner from j defined below.

- $\alpha(k, p, 0) := \min\{j | \texttt{PointersOf}(c_j)(p) = k\}$. Phase k starts for process p at the point where the value of its pointer becomes equal to k. Notice that $\alpha(1, p, 0) = 0$ for all $p \in P$ (all processes are initially in phase 1); and that for $k > 1$, the transition $t_{\alpha(k,p,0)}$ is of the form $(q, \texttt{update}_p, q')$, i.e., a transition in which p performs an update (its $(k-1)^{th}$ update operation.)
- $\alpha(k, p, \ell+1)$ is defined to be the smallest j such that $\alpha(k, p, \ell) < j$ and $t_j \in \Delta_p$ and $\texttt{PointersOf}(c_j)(p) = k$. The $(\ell+1)^{st}$ move of process p is defined by the next transition that belongs to Δ_p. Notice that $t_{\alpha(k,p,\ell+1)}$ is not an update transition since we require that p remains in phase k after performing the transition. Also, observe that $\alpha(k, p, \ell)$ is defined only for finitely many ℓ. We define $\#(k, p) := \max\{\ell | \alpha(k, p, \ell) \text{ is defined}\}$, i.e., the value of $\#(k, p)$ is the number of transitions process p executes during phase k.

In order to define π_{TSO}, we first define the set of configurations that appear in π_{TSO}. For each $k : 1 \leq k \leq r$, $p \in P$, and $\ell : 0 \leq \ell \leq \#(k, p)$ we have a TSO-configuration $d_{k,p,\ell}$ that is defined in terms of the SB-configurations that appear in π_{SB}. We define $d_{k,p,\ell}$ by defining its local state definition, buffer contents, and memory state. First, we define the local states of the processes.

- $\texttt{StatesOf}(d_{k,p,\ell})(p) := \texttt{StatesOf}\left(c_{\alpha(k,p,\ell)}\right)(p)$. When process p has performed its ℓ^{th} transition during phase k, its state is identical to its state in the corresponding SB-configuration $c_{\alpha(k,p,\ell)}$.
- If $p' \prec p$ then $\texttt{StatesOf}(d_{k,p,\ell})(p') := \texttt{StatesOf}\left(c_{\alpha(k,p',\#(k,p'))}\right)(p')$. If $p' \prec p$ then the state of p' will not change while p is making its moves. This state is given by the state of p' after it made its last move during phase k.
- If $p \prec p'$ then $\texttt{StatesOf}(d_{k,p,\ell})(p') := \texttt{StatesOf}\left(c_{\alpha(k,p',0)}\right)(p')$. If $p \prec p'$ then the state of p' will not change while p is making its moves. This state is given by the state of p' when it entered phase k (before it has made any moves during phase k.)

The buffer contents of $d_{k,p,\ell}$ are defined as follows.

- $\texttt{BuffersOf}(d_{k,p,\ell})(p) := \texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell)}\right) \odot k\right)$. When process p has performed its ℓ^{th} transition during phase k, the content of its buffer is defined by (i) considering the buffer of the corresponding SB-configuration $c_{\alpha(k,p,\ell)}$; (ii) removing the prefix of that buffer up to the position of the pointer of p; (iii) considering only messages corresponding to p; and (iv) converting these messages to the corresponding TSO-messages.
- If $p' \prec p$ then $\texttt{BuffersOf}(d_{k,p,\ell})(p') := \texttt{BufferOf}\left(c_{\alpha(k,p',\#(k,p'))}\right)(p')$. In a similar manner to the case of states, if $p' \prec p$ then the buffer of p' will not change while p is making its moves.
- If $p \prec p'$ then $\texttt{BuffersOf}(d_{k,p,\ell})(p') := \texttt{BuffersOf}\left(c_{\alpha(k,p',0)}\right)(p')$. Again, this case can be explained in a similar manner to the case of states.

Finally, the memory state is defined by

- $\texttt{MemoryOf}(d_{k,p,\ell}) := mem$ where $\texttt{BufferOf}(c_n)(k) = (mem, p', x)$ for some process $p' \in P$ and variable $x \in X$. This definition is consistent with the fact that all processes have identical views of the memory when they are in the same phase k. This view is defined by the memory component mem of the message at position k in the buffer.

Notice that the values of $\texttt{StatesOf}(d_{k,p,\ell})(p')$ and $\texttt{BuffersOf}(d_{k,p,\ell})(p')$ above are well-defined since the computation π_{SB} is balanced.

The following lemma implies the result. More precisely, it shows the existence of a TSO-computation that starts from an initial TSO-configuration and whose target has the same local state definitions as the target c_n of the SB-computation π_{SB}.

LEMMA C.1. *$d_{1,p_{min},0} \xrightarrow{\pi_{TSO}}_{TSO} d_{r,p_{max},\#(r,p_{max})}$ for some TSO-computation π_{TSO}. Furthermore $d_{1,p_{min},0}$ is an initial TSO-configuration, and $\texttt{StatesOf}\left(d_{r,p_{max},\#(r,p_{max})}\right) = \texttt{StatesOf}(c_n)$.*

The Lemma follows from Lemmata C.2–C.5. Lemma C.2–C.4 show the existence of the computation π_{TSO}, while Lemma C.6 and Lemma C.5 show the conditions on the initial and target configurations. For a word $w \neq \varepsilon$ and $i : 0 \leq i \leq |w|$, we define $w \lhd i$

to be the suffix of w of length i, i.e., it is the (unique) word w_2 such that $|w_2| = i$ and $w = w_1 \cdot w_2$ for some word w_1. Notice that $w \lhd i = w \odot (|w| - i)$.

LEMMA C.2. *If $\ell < \#(k,p)$ then $d_{k,p,\ell} \xrightarrow{t_{\alpha(k,p,\ell+1)}}_{TSO} d_{k,p,\ell+1}$.*

Proof We recall that $t_{\alpha(k,p,\ell+1)}$ is not an update operation. Let $t_{\alpha(k,p,\ell+1)}$ be of the form (q, op, q'). First, we show that $\texttt{StatesOf}\left(d_{\alpha(k,p,\ell)}\right)(p) = q$ and that $\texttt{StatesOf}\left(d_{k,p,\ell+1}\right) = \texttt{StatesOf}\left(d_{\alpha(k,p,\ell)}\right)[p \hookleftarrow q']$.

By definition of α, we know that $t_j \notin \Delta_p$ for all $j : \alpha(k,p,\ell) < j < \alpha(k,p,\ell+1)$. Therefore, $\texttt{StatesOf}(c_j)(p) = \texttt{StatesOf}\left(c_{\alpha(k,p,\ell)}\right)(p)$ for all $j : \alpha(k,p,\ell) < j < \alpha(k,p,\ell+1)$. In particular, $\texttt{StatesOf}\left(c_{\alpha(k,p,\ell+1)-1}\right)(p) = \texttt{StatesOf}\left(c_{\alpha(k,p,\ell)}\right)(p)$. From the definition of π_{SB} it follows that $c_{\alpha(k,p,\ell+1)-1} \xrightarrow{t_\alpha(k,p,\ell+1)}_{SB} c_{\alpha(k,p,\ell+1)}$, and hence $\texttt{StatesOf}\left(c_{\alpha(k,p,\ell)}\right)(p) = \texttt{StatesOf}\left(c_{\alpha(k,p,\ell+1)-1}\right)(p) = q$. It also follows that $\texttt{StatesOf}\left(c_{\alpha(k,p,\ell+1)}\right)(p) = q'$.

Next, we show that $\texttt{StatesOf}\left(c_{\alpha(k,p,\ell+1)}\right)(p') = \texttt{StatesOf}\left(c_{\alpha(k,p,\ell)}\right)(p')$ if $p' \neq p$. By definition of d it follows that if $p' \prec p$ then $\texttt{StatesOf}\left(d_{k,p,\ell+1}\right)(p') = \texttt{StatesOf}\left(c_{\alpha(k,p',\#(k,p'))}\right)(p') = \texttt{StatesOf}\left(d_{k,p,\ell}\right)(p')$; and if $p \prec p'$ then $\texttt{StatesOf}\left(d_{k,p,\ell+1}\right)(p') = \texttt{StatesOf}\left(c_{\alpha(k,p',0)}\right)(p') = \texttt{StatesOf}\left(d_{k,p,\ell}\right)(p')$.

Now, we proceed to prove the lemma. We consider the cases where *op* is a write or a read operation. The other cases can be treated in a similar way.

- If $op = \mathsf{w}(x,v)$. By definition of α, we know that $t_j \notin \Delta_p$ for all $j : \alpha(k,p,\ell) < j < \alpha(k,p,\ell+1)$. Therefore, $\texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,j)}\right) \odot k\right) = \texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell)}\right) \odot k\right)$ for all $j : \alpha(k,p,\ell) < j < \alpha(k,p,\ell+1)$. In particular, $\texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)-1}\right) \odot k\right) = \texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell)}\right) \odot k\right)$. From the definition of π_{SB} it follows that $c_{\alpha(k,p,\ell+1)-1} \xrightarrow{t_\alpha(k,p,\ell+1)}_{SB} c_{\alpha(k,p,\ell+1)}$, and hence $\texttt{BuffersOf}\left(d_{k,p,\ell+1}\right)(p) = \texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)}\right) \odot k\right) = \texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)-1}\right) \odot k\right) \cdot (x,v) = \texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell)}\right) \odot k\right) \cdot (x,v) = \texttt{BuffersOf}\left(d_{k,p,\ell}\right)(p) \cdot (x,v)$.
  Also in similar manner to the case of states we can show that $\texttt{BuffersOf}\left(d_{k,p,\ell+1}\right)(p') = \texttt{BuffersOf}\left(d_{k,p,\ell}\right)(p')$ in case $p' \neq p$. In other words, $\texttt{BufferOf}\left(d_{k,p,\ell+1}\right) = \texttt{BufferOf}\left(d_{k,p,\ell}\right)\left[p \hookleftarrow (x,v) \cdot \texttt{BufferOf}\left(d_{k,p,\ell}\right)\right]$.
  Suppose that $\texttt{BufferOf}(c_n)(k) = (mem, p, x)$. Then, $\texttt{MemoryOf}\left(d_{k,p,\ell+1}\right) = \texttt{MemoryOf}\left(d_{k,p,\ell}\right) = mem$.
- If $op = \mathsf{r}(x,v)$. In a similar manner to the above reasoning about write operations, we can show that $\texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)-1}\right) \odot k\right) = \texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell)}\right) \odot k\right)$. From the definition of π_{SB} it follows that $c_{\alpha(k,p,\ell+1)-1} \xrightarrow{t_\alpha(k,p,\ell+1)}_{SB} c_{\alpha(k,p,\ell+1)}$. There are two cases
  - $\texttt{LastWrite}\left(c_{\alpha(k,p,\ell+1)-1}, p, x\right) > k$. By the definition of $\texttt{IndexOf}$ it follows that there is an $i : k < i \leq \left|\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)-1}\right)\right|$ such that $\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)-1}\right)(i) = (mem, p, x)$, $mem(x) = v$, and there no $j : i < j \leq \left|\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)-1}\right)\right|$ and mem' such that $\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)-1}\right)(j) = (mem', p, x)$. Since $k < i$ it follows that $\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)-1}\right) \odot k\right)(i-k) = (mem, p, x)$, and there are no $j : i < j \leq \left|\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)-1}\right)\right|$ and mem' such that $\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)-1}\right) \odot k\right)(j-k) = (mem', p, x)$. By the definition of $\texttt{SBtoTSO}$ it follows that there is an $i : 1 \leq i \leq \left|\texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)-1}\right) \odot k\right)\right|$ such that $\texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)-1}\right) \odot k\right)(i) = (x,v)$, and $(x,v') \notin \texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)-1}\right) \odot k\right) \lhd (i-1)$ for all $v' \in V$. Hence, there is an $i : 1 \leq i \leq \left|\texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)-1}\right) \odot k\right)\right|$ such that $\texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell)}\right) \odot k\right)(i) = (x,v)$, and $(x,v') \notin \texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell)}\right) \odot k\right) \lhd (i-1)$ for all $v' \in V$. By definition of d it follows there is an $i : 1 \leq i \leq \left|\texttt{BuffersOf}\left(d_{k,p,\ell}\right)(p)\right|$ such that $\texttt{BuffersOf}\left(d_{k,p,\ell}\right)(p)(i) = (x,v)$ and $(x,v') \notin \texttt{BuffersOf}\left(d_{k,p,\ell}\right)(p) \lhd (i-1)$ for all $v' \in V$.
  - $\texttt{LastWrite}\left(c_{\alpha(k,p,\ell+1)-1}, p, x\right) = k$. By the definition of $\texttt{IndexOf}$ it follows that there is no $i : k < i \leq \left|\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)-1}\right)\right|$ and mem such that $\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)-1}\right)(i) = (mem, p, x)$. Since $k < i$ it follows that there is no mem such that $(mem, p, x) \in \left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)-1}\right) \odot k\right)$. By the definition of $\texttt{SBtoTSO}$ it follows that there is no $v' \in V$ such that $(x,v') \in \texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)-1}\right) \odot k\right)$. Hence, there is no $v' \in V$ such that $(x,v') \in \texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell)}\right) \odot k\right)$. By definition of d it follows there is no $v' \in V$ such that $(x,v') \in \texttt{BuffersOf}\left(d_{k,p,\ell}\right)(p)$. Also, by the definition of $\texttt{IndexOf}$ it follows that $\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)-1}\right)(k) = (mem, p', x')$ for some process $p' \in P$ and variable $x' \in X$ and mem with $mem(x) = v$. This implies that $\texttt{BufferOf}(c_n)(k) = (mem, p, x)$. By definition of d it follows that $\texttt{MemoryOf}\left(d_{k,p,\ell}\right) = mem$.

  It remains to show that $\texttt{BufferOf}\left(d_{k,p,\ell+1}\right) = \texttt{BufferOf}\left(d_{k,p,\ell}\right)$ and $\texttt{MemoryOf}\left(d_{k,p,\ell+1}\right) = \texttt{MemoryOf}\left(d_{k,p,\ell}\right)$.
  We know that $\texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)-1}\right) \odot k\right) = \texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell)}\right) \odot k\right)$.
  From the definition of π_{SB}, we know that $\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)-1}\right) = \texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)}\right)$. By

definition of d, we know that $\texttt{BufferOf}\left(d_{k,p,\ell+1}\right) = \texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)}\right)\odot k\right)$ and that $\texttt{BufferOf}\left(d_{k,p,\ell}\right) = \texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell)}\right)\odot k\right)$. Therefore $\texttt{BufferOf}\left(d_{k,p,\ell+1}\right) = \texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)}\right)\odot k\right) = \texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell+1)-1}\right)\odot k\right) = \texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\ell)}\right)\odot k\right) = \texttt{BufferOf}\left(d_{k,p,\ell}\right)$.
By definition of α, we know that $\texttt{MemoryOf}\left(d_{k,p,\ell+1}\right) = mem$ where $\texttt{BufferOf}(c_n)(k) = (mem,p,x)$ for some process $p \in P$ and variable $x \in X$. By the definition of α it also follows that $\texttt{MemoryOf}\left(d_{k,p,\ell}\right) = mem$, and hence $\texttt{MemoryOf}\left(d_{k,p,\ell+1}\right) = \texttt{MemoryOf}\left(d_{k,p,\ell}\right)$. ▮

LEMMA C.3. *If $p \prec p_{max}$ then $d_{k,p,\#(k,p)} = d_{k,succ(p),0}$.*

Proof First, we show that $\texttt{StatesOf}\left(d_{k,p,\#(k,p)}\right)(p') = \texttt{StatesOf}\left(d_{k,succ(p),0}\right)(p')$ for all $p' \in P$. There are four cases:

- If $p' = p$. Since $p \prec succ(p)$ it follows that $\texttt{StatesOf}\left(d_{k,succ(p),\ell}\right)(p) = \texttt{StatesOf}\left(d_{k,p,\#(k,p)}\right)(p)$ for all $\ell : 0 \le \ell \le \#(k,succ(p))$; and in particular $\texttt{StatesOf}\left(d_{k,succ(p),0}\right)(p) = \texttt{StatesOf}\left(d_{k,p,\#(k,p)}\right)(p)$.
- If $p' = succ(p)$. We have that $\texttt{StatesOf}\left(d_{k,p,\ell}\right)(succ(p)) = \texttt{StatesOf}\left(d_{k,succ(p),0}\right)(succ(p))$ for all $\ell : 0 \le \ell \le \#(k,p)$; and in particular $\texttt{StatesOf}\left(d_{k,p,\#(k,p)}\right)(succ(p)) = \texttt{StatesOf}\left(d_{k,succ(p),0}\right)(succ(p))$.
- If $p' \prec p \prec succ(p)$, then $\texttt{StatesOf}\left(d_{k,succ(p),\ell}\right)(p') = \texttt{StatesOf}\left(d_{k,p',\#(k,p')}\right)(p')$ for all $\ell : 0 \le \ell \le \#(k,succ(p))$ and in particular $\texttt{StatesOf}\left(d_{k,succ(p),0}\right)(p') = \texttt{StatesOf}\left(d_{k,p',\#(k,p')}\right)(p')$. Also, $\texttt{StatesOf}\left(d_{k,p,\ell}\right)(p') = \texttt{StatesOf}\left(d_{k,p',\#(k,p')}\right)(p')$ for all $\ell : 0 \le \ell \le \#(k,p)$ and in particular $\texttt{StatesOf}\left(d_{k,p,\#(k,p)}\right)(p') = \texttt{StatesOf}\left(d_{k,p',\#(k,p')}\right)(p')$. Hence, $\texttt{StatesOf}\left(d_{k,succ(p),0}\right)(p') = \texttt{StatesOf}\left(d_{k,p,\#(k,p)}\right)(p')$.
- If $p \prec succ(\prec)p'$, then $\texttt{StatesOf}\left(d_{k,succ(p),\ell}\right)(p') = \texttt{StatesOf}\left(d_{k,p',0}\right)(p')$ for all $\ell : 0 \le \ell \le \#(k,p)$ and in particular $\texttt{StatesOf}\left(d_{k,succ(p),0}\right)(p') = \texttt{StatesOf}\left(d_{k,p',0}\right)(p')$. Also, $\texttt{StatesOf}\left(d_{k,p,\ell}\right)(p') = \texttt{StatesOf}\left(d_{k,p',0}\right)(p')$ for all $\ell : 0 \le \ell \le \#(k,p)$ and in particular $\texttt{StatesOf}\left(d_{k,p,\#(k,p)}\right)(p') = \texttt{StatesOf}\left(d_{k,p',0}\right)(p')$. Hence, $\texttt{StatesOf}\left(d_{k,succ(p),0}\right)(p') = \texttt{StatesOf}\left(d_{k,p,\#(k,p)}\right)(p')$.

In a similar manner to the case of states, we can show that $\texttt{BuffersOf}\left(d_{k,p,\#(k,p)}\right)(p') = \texttt{BuffersOf}\left(d_{k,succ(p),0}\right)(p')$ for all $p' \in P$.

Finally, $\texttt{MemoryOf}\left(d_{k,p,\#(k,p)}\right) = mem$ where $\texttt{BufferOf}(c_n)(k) = (mem,p,x)$ for some process $p \in P$ and variable $x \in X$. Also, $\texttt{MemoryOf}\left(d_{k,succ(p),0}\right) = mem$, and hence $\texttt{MemoryOf}\left(d_{k,p,\#(k,p)}\right) = \texttt{MemoryOf}\left(d_{k,succ(p),0}\right)$.

LEMMA C.4. *if $k < r$ then $d_{k,p_{max},\#(k,p_{max})} \xrightarrow{\texttt{update}_{p^u}}_{TSO} d_{k+1,p_{min},0}$ for some $p^u \in P$.*

Proof We take p^u to be the process such that $\texttt{BufferOf}(c_n)(k+1)$ is of the form (mem,p^u,x) for some mem and x. From the definition of $\texttt{SBtoTSO}$ we know that $\texttt{SBtoTSO}(p^u)\left(\texttt{BufferOf}\left(d_{k,p_{max},\#(k,p_{max})}\right)\right)(k+1) = (x,mem(x))$. From the definition of α we know that $operation\left(t_{\alpha(k+1,p,0)}\right) = \texttt{update}_p$ for each $p \in P$.

First, we show that $\texttt{StatesOf}\left(d_{k,p_{max},\#(k,p_{max})}\right) = \texttt{StatesOf}\left(d_{k+1,p_{min},0}\right)$. Take any $p \in P$. From the definition of d we know that, for each $p \in P$, we have that $\texttt{StatesOf}\left(d_{k,p_{max},\#(k,p_{max})}\right)(p) = \texttt{StatesOf}\left(d_{k,p,\#(k,p)}\right)(p) = \texttt{StatesOf}\left(c_{\alpha(k,p,\#(k,p))}\right)(p)$ and that $\texttt{StatesOf}\left(d_{k+1,p_{min},0}\right)(p) = \texttt{StatesOf}\left(d_{k+1,p,0}\right)(p) = \texttt{StatesOf}\left(c_{\alpha(k+1,p,0)}\right)(p)$. Since $operation\left(t_{\alpha(k+1,p,0)}\right) = \texttt{update}_p$ we have $\texttt{StatesOf}\left(c_{\alpha(k+1,p,0)-1}\right) = \texttt{StatesOf}\left(c_{\alpha(k+1,p,0)}\right)$. Also from the definition of α it follows that $t_j \notin \Delta_p$ for all $j : \alpha(k,p,\#(k,p)) < j < \alpha(k+1,p,0)$. Therefore, $\texttt{StatesOf}(c_j)(p) = \texttt{StatesOf}\left(c_{\alpha(k,p,\#(k,p))}\right)(p)$ for all $j : \alpha(k,p,\#(k,p)) < j < \alpha(k+1,p,0)$. In particular, $\texttt{StatesOf}\left(c_{\alpha(k+1,p,0)-1}\right)(p) = \texttt{StatesOf}\left(c_{\alpha(k,p,\#(k,p))}\right)(p)$. Therefore, $\texttt{StatesOf}\left(c_{\alpha(k+1,p,0)}\right)(p) = \texttt{StatesOf}\left(c_{\alpha(k,p,\#(k,p))}\right)(p)$, and hence $\texttt{StatesOf}\left(d_{k+1,p_{min},0}\right)(p) = \texttt{StatesOf}\left(d_{k,p_{max},\#(k,p_{max})}\right)(p)$.

Next, we show that $\texttt{BuffersOf}\left(d_{k,p_{max},\#(k,p_{max})}\right) = \texttt{BuffersOf}\left(d_{k+1,p_{min},0}\right)\left[p^u \hookleftarrow (x,mem(x))\cdot\texttt{BuffersOf}\left(d_{k+1,p_{min},0}\right)\right]$. Take any $p \in P$. From the definition of d we know that, for each $p \in P$, we have that $\texttt{BuffersOf}\left(d_{k,p_{max},\#(k,p_{max})}\right)(p) = \texttt{BuffersOf}\left(d_{k,p,\#(k,p)}\right)(p) = \texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\#(k,p))}\right)\odot k\right)$ and that $\texttt{BuffersOf}\left(d_{k+1,p_{min},0}\right)(p) = \texttt{BuffersOf}\left(d_{k+1,p,0}\right)(p) = \texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k+1,p_{min},0)}\right)\odot k+1\right)$. From $\texttt{BufferOf}(c_n)(k+1) = (mem,p^u,x)$ it follows that $\texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\#(k,p))}\right)\odot k\right) = (x,mem(x))\cdot\texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k+1,p_{min},0)}\right)\odot k+1\right)$ if $p = p^u$, and that $\texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k,p,\#(k,p))}\right)\odot k\right) = \texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(k+1,p_{min},0)}\right)\odot k+1\right)$ if $p \neq p^u$. The result follows immediately.

Finally, we show that $\texttt{MemoryOf}\left(d_{k+1,p_{min},0}\right) = \texttt{MemoryOf}\left(d_{k,p_{max},\#(k,p_{max})}\right)\left[x \hookleftarrow \texttt{MemoryOf}\left(d_{k,p_{max},\#(k,p_{max})}\right)\right]$. By definition of d we know that $\texttt{MemoryOf}\left(d_{k+1,p_{min},0}\right) = mem$ $\texttt{MemoryOf}\left(d_{k,p_{max},\#(k,p_{max})}\right) = mem'$ where $\texttt{BufferOf}(c_n)(k) =$

(mem', p', x') for some $p' \in P$ and $x \in X$. From the definition of the SB-semantics it follows that $mem = mem'\,[x \hookleftarrow mem(x)]$. ▌

The following lemma shows that π_{TSO} starts from an initial TSO-configuration.

LEMMA C.5. *$d_{1,p_{min},0}$ is an initial TSO-configuration.*

Proof By the definitions of d and α we know that $\texttt{StatesOf}\left(d_{1,p_{min},0}\right)(p) = \texttt{StatesOf}\left(d_{1,p,0}\right)(p) = \texttt{StatesOf}\left(c_{\alpha(1,p,0)}\right)(p) = \texttt{StatesOf}\left(c_0\right)(p) = \underline{q_{init}}$.

Also, $\texttt{BufferOf}\left(d_{1,p_{min},0}\right)(p) = \texttt{BufferOf}\left(d_{1,p,0}\right)(p) = \texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_{\alpha(1,p,0)}\right) \odot 1\right) = \texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(c_1\right) \odot 1\right) = \varepsilon$.

The result follows immediately for the definition of initial TSO-configurations. ▌

The following lemma shows that the target of π_{TSO} has the same local process states as the target c_n of the SB-computation π_{SB}.

LEMMA C.6. $\texttt{StatesOf}\left(d_{r,p_{max},\#(r,p_{max})}\right) = \texttt{StatesOf}\left(c_n\right)$.

Proof Take any $p \in P$. By the definitions of d and α it follows that $\texttt{StatesOf}\left(d_{r,p_{max},\#(r,p_{max})}\right)(p) = \texttt{StatesOf}\left(d_{r,p,\#(r,p)}\right)(p) = \texttt{StatesOf}\left(c_{\alpha(r,p,\#(r,p))}\right)(p)$. By definition of α, we know that $t_j \notin \Delta_p$ for all $j : \alpha(r,p,\#(r,p)) < j < n$. Therefore, $\texttt{StatesOf}\left(c_j\right)(p) = \texttt{StatesOf}\left(c_n\right)(p)$ for all $j : \alpha(r,p,\#(r,p)) \le j < n$. In particular, $\texttt{StatesOf}\left(c_{\alpha(r,p,\#(r,p))}\right)(p) = \texttt{StatesOf}\left(c_n\right)(p)$. Hence $\texttt{StatesOf}\left(d_{r,p_{max},\#(r,p_{max})}\right)(p) = \texttt{StatesOf}\left(c_n\right)(p)$. ▌

From TSO to SB In the following, we show the *if* direction of Theorem 4.1. Consider a TSO-computation $\pi_{TSO} = c_0 \xrightarrow{t_1}_{TSO} c_1 \cdots \xrightarrow{t_{n-1}}_{TSO} c_{n-1} \xrightarrow{t_n}_{TSO} c_n$. To simplify the presentation, we assume without loss of generality that the last operation performed by each process is an atomic read-write. This implies that the process buffers in the TSO-configuration c_n are empty (i.e., $\texttt{BuffersOf}\left(c_n\right)(p)$ for all process $p \in P$). Moreover, we assume that the given concurrent program $\mathcal{P}$ does not contain fence operations since they can be simulated with atomic read-write operations. In the following, we will derive a SB-computation π_{SB} such that $\texttt{StatesOf}\left(target\left(\pi_{SB}\right)\right) = \texttt{StatesOf}\left(c_n\right)$.

The idea is to construct the run π_{SB} such that the buffer content (say $b = b(1)b(2)\cdots b(m+1)$) in the configuration $target\left(\pi_{SB}\right)$ is exactly the sequence of successive shared memory contents induced by the computation π_{TSO}. Then, the computation of π_{SB} consists of $m = (|b|-1)$ phases (henceforth referred as the phases $1, 2, \ldots, m$). Let $d_0, d_1, \ldots, d_m$ be a sequence of SB-configurations such that the phase $j : 1 \le j \le m$ starts at configuration d_{j-1} and ends at the configuration d_j. The effect of a phase j is to append the message $b(j+1)$ to the tail of the buffer of d_j. This is done by simulating a run of the process producing the new message $b(j+1)$.

For every $p \in P$, let $\Delta_p^{\mathsf{w}} \subseteq \Delta_p$ (resp. $\Delta_p^{\mathsf{u}} \subseteq \Delta_p \cup \{\mathsf{update}_p\}$) be the set of write (resp. update) and atomic read-write operations that can be performed by the process p.

Let $I = i_1, \ldots, i_m$ be the maximal sequence of indices such that $1 \le i_1 < i_2 < \cdots < i_m \le n$ and for every $j : 1 \le j \le m$, we have t_{i_j} is an update operation or an atomic read-write operation (i.e., $t_{i_j} \in \bigcup_{p \in P} \Delta_p^{\mathsf{u}}$).

For every $j : 1 \le j \le m$, let $\texttt{ProcessOf}(j)$ be the process $p \in P$ such that $t_{i_j} \in \Delta_p^{\mathsf{u}}$ is an update operation or an atomic read-write operation of p. We define $\texttt{Match}(j)$ (inductively on j) to be the index $i = \min\left\{k \mid \left(t_k \in \Delta_p^{\mathsf{w}}\right) \wedge \left(\nexists \ell < j, \texttt{Match}(\ell) = k\right)\right\}$ of the first write (resp. atomic read-write) operation of p which is not yet matched with any update operations (resp. atomic read-write). This means that the update (resp. atomic read-write) operation t_{i_j} corresponds to the execution of the write (resp. atomic read-write) operation t_i. Moreover, we can show that if t_{i_j} is an atomic read-write operation then $\texttt{Match}(j) = i_j$.

Let $b = b(1)b(2)\cdots b(m+1)$ be the SB-buffer content induced by the sequence $t_{i_1}, \ldots, t_{i_m}$ of memory updates and atomic read-write operations from the initial buffer content. I.e., the SB-buffer b satisfies the following two conditions:

- $b(1) = \left(\texttt{MemoryOf}\left(c_0\right), p, x\right)$. Initially, the buffer contains a single message where $\texttt{MemoryOf}\left(c_0\right)$ is the initial value of the memory, and p and x are respectively some process and some shared variable.
- For every $j : 1 \le j \le m$, $b(j+1) = \left(mem_{j+1}, p_{j+1}, x_{j+1}\right)$ such that $mem_{j+1} = mem_j\left[x_{j+1} \hookleftarrow v_{j+1}\right]$, $p_{j+1} = \texttt{ProcessOf}(j)$, $i = \texttt{Match}(j)$, and t_i is of the following form: $t_i = \left(q_j, \mathsf{w}(x_{j+1}, v_{j+1}), q_j'\right)$ or $t_i = \left(q_j, \mathsf{arw}(x_{j+1}, v_j, v_{j+1}), q_j'\right)$. The new element $b(j+1)$ is appended to the tail of the buffer. The value of the variables in the new element are identical to those in the previous last element $b(j)$ expect that the value of the variable x_{j+1} has been updated to v_{j+1}.

Let $d_0 = \left(\underline{q_{init}}, b_{init}, \underline{z_{init}}\right)$, where b_{init} contains the unique message $b(1)$, be the initial SB-configuration. We define (inductively) the sequence $d_1, \ldots, d_m$ of SB-configurations as follows: For every $j : 1 \le j \le m$:

First, we define the local states of the processes:

- $\texttt{StatesOf}\left(d_j\right)(p') = \texttt{StatesOf}\left(d_{j-1}\right)(p')$ for all $p' \in P$ such that $p' \ne \texttt{ProcessOf}(j)$. The state of the process p' in the SB-configuration d_j is identical to its state in the SB-configuration d_{j-1}. In fact, the configuration d_{j-1} is reachable from the configuration d_j by a run where $\texttt{ProcessOf}(j)$ is the only active process.
- $\texttt{StatesOf}\left(d_j\right)(p) = \texttt{StatesOf}\left(c_i\right)(p)$ with $i = \texttt{Match}(j)$ and $p = \texttt{ProcessOf}(j)$. The state of the process p in the SB-configuration d_j corresponds to the state of the process p in the TSO-configuration c_i (reachable after performing the write operation t_i associated with the update operation t_{i_j}).

The buffer content of d_j is defined as follows:

- $\texttt{BufferOf}\left(d_j\right) = b(1)b(2)\cdots b(j+1)$. The buffer content of $\texttt{BufferOf}\left(d_j\right)$ corresponds to the sequence of messages induced by the sequence $t_{i_1}, \ldots, t_{i_j}$ of memory updates and atomic read-write operations from the initial buffer content.

Finally, the pointer of each process is defined by:

- $\texttt{PointersOf}\left(d_j\right)(p') = \texttt{PointersOf}\left(d_{j-1}\right)(p')$ for all $p' \in P$ such that $p' \ne \texttt{ProcessOf}(j)$. The pointer of the process p' in the SB-configuration d_j is identical to its pointer in the SB-configuration d_{j-1}.
- $\texttt{PointersOf}\left(d_j\right)(p) = \max(\{k \mid i_k \le \texttt{Match}(j)\} \cup \{0\}) + 1$ with $p = \texttt{ProcessOf}(j)$. The process p points to the element corresponding to the current content of the shared memory in the TSO-configuration c_i (i.e., $\texttt{BufferOf}\left(d_j\right)(k) = \left(\texttt{MemoryOf}\left(c_i\right), p', x'\right)$ for some $p' \in P$ and $x' \in X$). Furthermore, if $r = \texttt{PointersOf}\left(d_j\right)(p)$ then t_{i_r} corresponds to the last update or atomic read-write operation performed by the concurrent program $\mathcal{P}$ under TSO before reaching c_i. Observe that if t_i is an atomic read-write operation then $\texttt{Match}(j) = i_j$ and $\texttt{PointersOf}\left(d_j\right)(p) = j+1$.

The relation between the SB-configuration d_j and the TSO-configuration c_i is given by the following lemma which ensures that d_j and c_i have the same state and the sequence of pending elements associated with the process $\mathtt{ProcessOf}\,(j)$.

LEMMA C.7. *For every $j : 1 \leq j \leq m$, let $p = \mathtt{ProcessOf}\,(j)$, $i = \mathtt{Match}\,(j)$ and $k = \mathtt{PointersOf}\,\big(d_j\big)\,(p)$. Then, we have:*

1. $\mathtt{StatesOf}\,\big(d_j\big)\,(p) = \mathtt{StatesOf}\,(c_i)\,(p)$,
2. $\mathtt{SBtoTSO}\,(p)\,\big(\mathtt{BufferOf}\,\big(d_j\big) \odot k\big) = \mathtt{BuffersOf}\,(c_i)\,(p)$, *and*
3. $\mathtt{BufferOf}\,\big(d_j\big)\,(k) = (\mathtt{MemoryOf}\,(c_i)\,, p', x')$ *for some $p' \in P$ and $x' \in X$.*

Proof By construction, we have that the properties 1 and 3 hold. Moreover, if t_i is an atomic read-write then $\mathtt{SBtoTSO}\,(p)\,\big(\mathtt{BufferOf}\,\big(d_j\big) \odot k\big) = \mathtt{BuffersOf}\,(c_i)\,(p) = \varepsilon$ since $\mathtt{PointersOf}\,\big(d_j\big)\,(p) = j+1$ and $\mathtt{BufferOf}\,\big(d_j\big) = b(1)b(2)\cdots b(j+1)$.

Now, we will show that for any pending element (x,v) of $\mathtt{BuffersOf}\,(c_i)$, there is a unique index $k \leq j' \leq j$ such that $\mathtt{SBtoTSO}\,(p)\,(b(j'+1)) = (x,v)$ (i.e., $\mathtt{BuffersOf}\,(c_i)\,(p)$ is a sub-word of $\mathtt{SBtoTSO}\,(p)\,\big(\mathtt{BufferOf}\,\big(d_j\big) \odot k\big)$). If there is a pending element (x,v) in the buffer $\mathtt{StatesOf}\,(c_i)\,(p)$ associated with the process p then this element is issued by a write operation $t_{i'}$ of the process p. Notice that the write operation $t_{i'}$ should be performed before t_i since i is the index of the last write operation performed by the process p before reaching the configuration c_i. This implies that $i' \leq i$. Let $i_{j'}$ be the (unique) index of the matching update operation associated with the write operation $t_{i'}$ (i.e., $i' = \mathtt{Match}\,(j')$). Such index j' exists since in the target TSO-configuration c_n all the buffers are empty. This implies that $j' \leq j$ since $i' \leq i$ and the SB-buffer $\mathtt{BufferOf}\,\big(d_j\big)$ is FIFO. In addition, we have $k = \mathtt{PointersOf}\,\big(d_j\big)\,(p) \leq j'$ since the index k corresponds to the last update operation $t_{i_k}$of p performed before reaching the configuration c_i and the update operation $t_{i_{j'}}$ has not been yet performed (i.e., its associated matching write operation $t_{i'}$ is still pending in the buffer $\mathtt{BuffersOf}\,(c_i)\,(p)$ associated with the process p). Hence, the element $b(j'+1)$ associated with the update operation $t_{i_{j'}}$ (and so with the write operation $t_{i'}$) is in $\mathtt{BufferOf}\,\big(d_j\big) \odot k$ and we have $\mathtt{SBtoTSO}\,(p)\,(b(j'+1)) = (x,v)$ (see the definition of the SB-buffer b).

Similarly, we can show that for any index $j' : k \leq j' \leq j$ such that $\mathtt{SBtoTSO}\,(p)\,(b(j'+1)) \neq \varepsilon$ there is a unique index $r : 1 \leq r \leq |\mathtt{BuffersOf}\,(c_i)|$ such that $\mathtt{SBtoTSO}\,(p)\,(b(j')) = \mathtt{BuffersOf}\,(c_i)\,(r)$. This means that $\mathtt{SBtoTSO}\,(p)\,\big(\mathtt{BufferOf}\,\big(d_j\big) \odot k\big)$ is a sub-word of $\mathtt{BuffersOf}\,(c_i)\,(p)$. Hence, we have $\mathtt{SBtoTSO}\,(p)\,\big(\mathtt{BufferOf}\,\big(d_j\big) \odot k\big) = \mathtt{BuffersOf}\,(c_i)\,(p)$. ∎

Lemmata C.8 and C.9 imply the *if* direction of Theorem 4.1. More precisely, they show the existence of a SB-computation that starts from an initial SB-configuration d_0 and whose target d_m has the same local state definition as the target c_n of π_{TSO}.

LEMMA C.8. $\mathtt{StatesOf}\,(d_m) = \mathtt{StatesOf}\,(c_n)$.

Proof For every process $p \in P$, let $i : 1 \leq i \leq n$ be the index of the last operation t_i performed by the process p. Recall that we assume that the last operation performed by any process along the computation π_{TSO} is an atomic read-write operation. Let $j : 1 \leq j \leq m$ be the index such that $i = \mathtt{Match}\,(j)$. (In fact, we have $i = i_j$ since t_{i_j} is an atomic read-write operation.) Since t_i is the last operation performed by p, we have $\mathtt{StatesOf}\,(c_i)\,(p) = \mathtt{StatesOf}\,(c_n)\,(p)$. On the other hand, we have $\mathtt{StatesOf}\,\big(d_j\big)\,(p) = \mathtt{StatesOf}\,(d_m)\,(p)$ since $t_{i_{j+1}}, \ldots, t_{i_m}$ are not operations of the process p (see the definition of the sequence of SB-configurations $d_0, d_1, \ldots, d_m$). Now, we can use Lemma C.7 to show that $\mathtt{StatesOf}\,\big(d_j\big)\,(p) = \mathtt{StatesOf}\,(c_i)\,(p)$. This implies that $\mathtt{StatesOf}\,(d_m) = \mathtt{StatesOf}\,(c_n)$. ∎

LEMMA C.9. *For every $j : 1 \leq j \leq m$, $d_{j-1} \xrightarrow{\pi_j}_{SB} d_j$.*

Proof Let $p = \mathtt{ProcessOf}\,(j)$ and $i = \mathtt{Match}\,(j)$. Let i' be the *minimal* index such that for every $\ell : i' \leq \ell < i$, $t_\ell \notin \Delta_p^{\mathrm{w}}$ is not a write or an atomic read-write operation of the process p. In the case that such index does not exist, we simply take $i' = i$. Then, we define inductively the sequence of SB-configurations $d_j^{i'-1}, \ldots, d_j^{i-1}$ such that for every $\ell : i'-1 \leq \ell \leq i-1$, the SB-configuration d_j^ℓ is defined as follows:

First, we define the local states of the processes:

- $\mathtt{StatesOf}\,\Big(d_j^\ell\Big)\,(p') = \mathtt{StatesOf}\,\big(d_{j-1}\big)\,(p')$ for all $p' \in P$ such that $p' \neq p$. The state of the process p' in the SB-configuration d_j^ℓ is the same as the state of the process p' in the SB-configuration d_{j-1}.
- $\mathtt{StatesOf}\,\Big(d_j^\ell\Big)\,(p) = \mathtt{StatesOf}\,(c_\ell)\,(p)$. The state of the process p in the SB-configuration d_j^ℓ corresponds to its state in the TSO-configuration c_ℓ.

The buffer content of d_j^ℓ is defined as follows:

- $\mathtt{BufferOf}\,\Big(d_j^\ell\Big) = b(1)b(2)\cdots b(j)$. The buffer content of $\mathtt{BufferOf}\,\Big(d_j^\ell\Big)$ corresponds to the sequence of messages induced by the sequence $t_{i_1}, \ldots, t_{i_{j-1}}$ of memory updates and atomic read-write operations from the initial buffer content.

Finally, the pointer of each process is defined by:

- $\mathtt{PointersOf}\,\Big(d_j^\ell\Big)\,(p') = \mathtt{PointersOf}\,\big(d_{j-1}\big)\,(p')$ for all $p' \in P$ such that $p' \neq p$. The pointer of the process p' in the SB-configuration d_j is identical to its pointer in the SB-configuration d_{j-1}.
- $\mathtt{PointersOf}\,\Big(d_j^\ell\Big)\,(p) = (\max(\{k|\, i_k \leq \ell\} \cup \{0\})) + 1$. The process p points to the element corresponding to the current content of the shared memory in the TSO-configuration c_ℓ. If $r = \mathtt{PointersOf}\,\Big(d_j^\ell\Big)\,(p)$ then t_{i_r} corresponds to the last update or atomic read-write operation performed by the concurrent program $\mathcal{P}$under TSO before reaching c_ℓ.

The relation between the SB-configuration d_j^ℓ and the TSO-configuration c_ℓ is given by the following lemma (whose proof is similar to Lemma C.7):

LEMMA C.10. *Let $\ell : (i'-1) \leq \ell < i$ and $k = \mathtt{PointersOf}\,\Big(d_j^\ell\Big)\,(p)$. Then, we have*

1. $\mathtt{StatesOf}\,\Big(d_j^\ell\Big)\,(p) = \mathtt{StatesOf}\,(c_\ell)\,(p)$,
2. $\mathtt{SBtoTSO}\,(p)\,\Big(\mathtt{BufferOf}\,\Big(d_j^\ell\Big) \odot k\Big) = \mathtt{BuffersOf}\,(c_\ell)\,(p)$, *and*
3. $\mathtt{BufferOf}\,\Big(d_j^\ell\Big)\,(k) = (\mathtt{MemoryOf}\,(c_\ell)\,, p_\ell, x_\ell)$ *for some $p_\ell \in P$ and $x_\ell \in X$.*

Lemmata C.11, C.12, and C.13 show the existence of the computation π_j of Lemma C.9.

LEMMA C.11. $d_{j-1} = d_j^{i'-1}$.

Proof To prove Lemma C.11, it is sufficient to show that $\texttt{StatesOf}\left(d_{j-1}\right)(p) = \texttt{StatesOf}\left(d_j^{i'-1}\right)(p)$ and $\texttt{PointersOf}\left(d_{j-1}\right)(p) = \texttt{PointersOf}\left(d_j^{i'-1}\right)(p)$. From the definition of the index i', one of the following two cases holds

- If $(i'-1) = 0$ then for every $r : 1 \leq r < j$, we have $p \neq \texttt{ProcessOf}(r)$ (i.e., the transition $t_{i_r} \notin t_p^{\mathsf{u}}$ is not an update or an atomic read-write operation of the process p). This means that the state and the pointer of p along the sequence of SB-configurations $d_0 = d_1 = \cdots = d_{j-1}$ are kept the same. This implies that $\texttt{StatesOf}(d_0)(p) = \texttt{StatesOf}\left(d_{j-1}\right)(p)$ and $\texttt{PointersOf}(d_0)(p) = \texttt{PointersOf}\left(d_{j-1}\right)(p) = 1$. On the other hand, we have $\texttt{StatesOf}(c_{i'-1})(p) = \texttt{StatesOf}(d_0)(p) = \texttt{StatesOf}\left(d_j^{i'-1}\right)(p)$ and $\texttt{PointersOf}\left(d_j^{i'-1}\right)(p) = 1$. Hence, we have $\texttt{StatesOf}\left(d_{j-1}\right)(p) = \texttt{StatesOf}\left(d_j^{i'-1}\right)(p)$ and $\texttt{PointersOf}\left(d_{j-1}\right)(p) = \texttt{PointersOf}\left(d_j^{i'-1}\right)(p)$.
- If $i' > 1$ then $t_{i'-1} \in \Delta_p^{\mathsf{w}}$ is a write or an atomic read-write operation of the process p. Let j' be the unique index such that $i'-1 = \texttt{Match}(j')$ (i.e., $t_{i_{j'}}$ is the update or atomic read-write operation associated with $t_{i'-1}$). Since $i'-1 < i$, we have $j' < j$. We can use Lemma C.7 and the definition of the pointer of the process p in $d_{j'}$ and $d_j^{i'-1}$ to show that $\texttt{StatesOf}\left(d_{j'}\right)(p) = \texttt{StatesOf}(c_{i'-1})(p)$ and $\texttt{PointersOf}\left(d_{j'}\right)(p) = \texttt{PointersOf}\left(d_j^{i'-1}\right)(p)$. Moreover, for every $r : j' < r < j$, we have $p \neq \texttt{ProcessOf}(r)$ (i.e., the transition $t_{i_r} \notin t_p^{\mathsf{u}}$ is not an update or an atomic read-write operation of the process p). This means that the state and the pointer of p along the sequence of SB-configurations $d_{j'} = d_{j'+1} = \cdots = d_{j-1}$ are kept the same. This implies that $\texttt{StatesOf}\left(d_{j'}\right)(p) = \texttt{StatesOf}\left(d_{j-1}\right)(p)$ and $\texttt{PointersOf}\left(d_{j'}\right)(p) = \texttt{PointersOf}\left(d_{j-1}\right)(p)$. Hence, we have $\texttt{StatesOf}\left(d_{j-1}\right)(p) = \texttt{StatesOf}\left(d_{j'}\right)(p) = \texttt{StatesOf}\left(d_j^{i'-1}\right)(p)$ and $\texttt{PointersOf}\left(d_{j-1}\right)(p) = \texttt{PointersOf}\left(d_{j'}\right)(p) = \texttt{PointersOf}\left(d_j^{i'-1}\right)(p)$. ∎

LEMMA C.12. *For every $\ell : (i'-1) \leq \ell < (i-1)$, we have $d_j^\ell \rightarrow_{SB} d_j^{\ell+1}$ or $d_j^\ell = d_j^{\ell+1}$.*

Proof We recall that $c_\ell \xrightarrow{t_{\ell+1}}_{TSO} c_{\ell+1}$, $\texttt{StatesOf}(c_\ell)(p) = \texttt{StatesOf}\left(d_j^\ell\right)(p)$, and $\texttt{StatesOf}(c_{\ell+1})(p) = \texttt{StatesOf}\left(d_j^{\ell+1}\right)(p)$ (see Lemma C.10). We consider three cases depending on the form of the transition $t_{\ell+1}$:

- If $t_{\ell+1} \in \Delta_{p'}$ is a transition of a process $p' \neq p$ and $t_{\ell+1}$ is not an atomic read-write operation, then we have $d_j^\ell = d_j^{\ell+1}$ since $\texttt{StatesOf}(c_\ell)(p) = \texttt{StatesOf}(c_{\ell+1})(p)$ (i.e., the state of the process p has not been modified), and $\texttt{PointersOf}\left(d_j^\ell\right)(p) = \texttt{PointersOf}\left(d_j^{\ell+1}\right)(p)$ (i.e., the indices of the last update or atomic read-write operation before reaching c_ℓ and $c_{\ell+1}$ are the same).
- If $t_{\ell+1} \in \Delta'$ is an update operation or an atomic read-write operation of a process $p' \neq p$, then we can show that $d_j^\ell \xrightarrow{t}_{SB} d_j^{\ell+1}$ with t is an update operation of the process p. This is possible since we can show that $\texttt{PointersOf}\left(d_j^{\ell+1}\right) = \texttt{PointersOf}\left(d_j^\ell\right) + 1$ and $\texttt{StatesOf}(c_\ell)(p) = \texttt{StatesOf}(c_{\ell+1})(p)$. In fact, $\texttt{PointersOf}\left(d_j^\ell\right)$ is pointing to the index of the last update (or atomic read-write) operation performed before reaching the configuration c_ℓ and $\texttt{PointersOf}\left(d_j^{\ell+1}\right)$ is exactly pointing to the next last update (or atomic read-write) operation before reaching the configuration $c_{\ell+1}$ which is the index $r : 2 \leq r \leq (m+1)$ such that $i_{r-1} = \ell+1$.
- If $t_{\ell+1} \in \Delta_p$ then $t_{\ell+1}$ is necessarily a nop or a read operation of the process p. This implies that the indices of the last update or atomic read-write operation before reaching c_ℓ and $c_{\ell+1}$ are the same (i.e., $\texttt{PointersOf}\left(d_j^\ell\right)(p) = \texttt{PointersOf}\left(d_j^{\ell+1}\right)(p) = k$). Moreover, we can show that $d_j^\ell \xrightarrow{t_{\ell+1}}_{SB} d_j^{\ell+1}$ since we have (from Lemma C.10):
  - The states of the process p in c_ℓ (resp. $c_{\ell+1}$) and d_j^ℓ (resp. $d_j^{\ell+1}$) are the same (i.e., $\texttt{StatesOf}(c_\ell)(p) = \texttt{StatesOf}\left(d_j^\ell\right)(p)$, and $\texttt{StatesOf}(c_{\ell+1})(p) = \texttt{StatesOf}\left(d_j^{\ell+1}\right)(p)$).
  - The process p has the the same sequence of pending write operations in the configurations c_ℓ and d_j^ℓ (i.e., $\texttt{SBtoTSO}(p)\left(\texttt{BufferOf}\left(d_j^\ell\right) \odot k\right) = \texttt{BuffersOf}(c_\ell)(p)$).
  - The memory content in c_ℓ and the pointed memory content by the process p in d_j^ℓ are the same (i.e., $\texttt{BufferOf}\left(d_j^\ell\right)(k) = (\texttt{MemoryOf}(c_\ell), p_\ell, x_\ell)$ for some $p' \in P$ and $x' \in X$). ∎

LEMMA C.13. $d_j^{i-1} \xrightarrow{t_i}_{SB} d_j$.

Proof Recall that $c_{i-1} \xrightarrow{t_i}_{SB} c_i$. The fact that $d_j^{i-1} \xrightarrow{t_i}_{SB} d_j$ is an immediate consequence of the definition of the configurations d_j^{i-1} and d_j and the following facts: (1) the states of the process p in c_{i-1} (resp. c_i) and d_j^{i-1} (resp. d_j) are the same, (2) $\texttt{BufferOf}\left(d_j^{i-1}\right) = b(1)b(2)\cdots b(j)$ and $\texttt{BufferOf}\left(d_j\right) = b(1)b(2)\cdots b(j+1)$, (3) the states and the pointers of any process $p' \neq p$ in d_j^{i-1} and d_j are the same, and (4) the process p in d_j^{i-1} (resp. d_j) is pointing to the index of the last update (or atomic read-write) operation performed before reaching the configuration c_{i-1} (resp. c_i). ∎

D. Proof of Lemmas in Section 5

Lemma 5.1. *The relation $\sqsubseteq$ is a well-quasi ordering on SB-configurations.*

Proof This is an immediate consequence of the fact that (i) the subword relation is a well-quasi ordering on finite words [19], and that (ii) the number of states and messages (associated with last write operations and pointers) that should be equal, is finite. ∎

Lemma 5.2. *$\rightarrow_{SB}$ is effectively monotonic wrt. $\sqsubseteq$.*

Proof We need to show the following: For every SB-configurations c_1, c_1', and c_2 such that $c_1 \rightarrow_{SB} c_1'$ and $c_1 \sqsubseteq c_2$, there exists an SB-configuration c_2' such that $c_2 \xrightarrow{*}_{SB} c_2'$ and $c_1' \sqsubseteq c_2'$. Also, we need to show that, given c_1, c_1', c_2, we can compute c_2' and also compute a run π such that $c_2 \xrightarrow{\pi}_{SB} c_2'$.

The interesting case is when an update operation is performed (i.e., $c_1 \xrightarrow{t}_{SB} c_1'$ with $t = \mathsf{update}_p$). The effect of such operation is moving the pointer of p one step to the right. Since $c_1 \sqsubseteq c_2$, we know that there is index i such that the element at position i in $\mathtt{BufferOf}(c_2)$ is matched with the element pointed by the process p in c_1 with respect to the ordering relation $\sqsubseteq$. Moreover, the element at position i in $\mathtt{BufferOf}(c_2)$ corresponds to the position of the pointer of p in c_2. There is also an index $i < j$ such that the element at position j in $\mathtt{BufferOf}(c_2)$ is matched with the element one step to the right of the pointer of p in c_1. From the configuration c_2 the concurrent system can now perform several update operations to move the pointer of p from the position i to the position j and reach a configuration c_2'. (In fact, the number of such update operations is bounded by the size of the buffer of c_2). Moreover, we can show that $c_1' \sqsubseteq c_2'$ since there is no element associated with a last write operation at any position k with $i < k < j$.

Let us assume that $c_1 \xrightarrow{t}_{SB} c_1'$ with $t \in \Delta$. Then, we can show that there is a configuration c_2' such that $c_1' \xrightarrow{t}_{SB} c_2'$ and $c_1' \sqsubseteq c_2'$ since c_1 and c_2 have the same states and the same sequence of elements associated with the last write operations and pointers in their respective buffers. Observe that for ARW and Fence operations, we have for every $p \in P$, $\mathtt{PointersOf}(c_1)(p) = |\mathtt{BufferOf}(c_1)|$ if and only if $\mathtt{PointersOf}(c_2)(p) = |\mathtt{BufferOf}(c_2)|$. ∎

Computing the set of minimal elements. Let C be a set of SB-configurations. We use $\mathtt{Min}(C)$ to denote the smallest subset of C such that for all $c' \in C$ there exists an SB-configuration $c \in \mathtt{Min}(C)$ such that $c \sqsubseteq c'$. The set $\mathtt{Min}(C)$ is called the minor set of C. Notice that $\mathtt{Min}(C)$ is finite since $\sqsubseteq$ is a well-quasi ordering.

In the following, we show that it is possible to compute the set of minimal elements of a regular set of SB-configurations

LEMMA D.1. *Let A be an SB-automaton. Then it is possible to compute* $\mathtt{Min}(L(A))$.

Proof Let $A = (S, \Delta, S^{final}, h)$. In the following, we show that if a configuration $c = (\underline{q}, b, \underline{z})$ is in the minor set of $L(A)$ then $|b| \le (|P| + |P||X|)(|S|+1)$. The proof of this fact is done by contradiction:

Let us assume that $c = (\underline{q}, b, \underline{z})$ is in the minor set of $L(A)$ and $|b| > (|P| + |P||X|)(|S|+1)$. Then, there is a word $w \in \Sigma^*$ such that $w \in L(A, h(\underline{q}))$ and $(b, \underline{z}) = decoding(w)$. This implies that $|w| = |b|$ and for every $i \in \{1, \ldots, |w|\}$, $w(i) = (b(i), \sigma_i)$ with $\sigma_i = \{p \in P \,|\, \underline{z}(p) = i\}$. Moreover, let $i_1, \ldots, i_m \in \{1, \ldots, |w|\}$, with $i_1 < i_2 < \cdots < i_m$ and $m \le (|P| + |P||X|)$, be the sequence of indices of w such that for $j : 1 \le j \le m$, $w(i_j)$ is either a symbol associated with a last write operation of some process or a process pointer. Therefore, w can be decomposed as follows: $w = u_1 w(i_1) u_2 w(i_2) u_3 \cdots u_m w(i_m)$. Since $|w| > (|P| + |P||X|)(|S|+1)$, then there is an index $k : 1 \le k \le m$ such that $|u_{i_k}| > |S|$. Now, we can use the pumping lemma for regular languages to show that there are x, y, z such that $u_{i_k} = tyz$, $|tz| \le |S|$, and the word $w' = u_1' w(i_1) u_2' w(i_2) u_3' \cdots u_m' w(i_m)$ is in $L(A, h(\underline{q}))$ with for every $j : 1 \le j \le m$, $u'_{i_j} = u_{i_j}$ if $j \ne k$ and $u'_{i_k} = tz$. Let $c' = (\underline{q}, b', \underline{z}')$ be the SB-configuration such that $(b', \underline{z}') = decoding(w')$. This implies that $|w'| = |b'|$ for every $i \in \{1, \ldots, |w'|\}$, $w'(i) = (b'(i), \sigma_i)$ with $\sigma_i = \{p \in P \,|\, \underline{z}'(p) = i\}$. Observe that then, the word u'_{i_k} does not contain any symbol associated with a last write operation of some process or a process pointer in the SB-configuration c'. Then, we can show that $c' \sqsubseteq c$ with $c' \ne c$ (which contradicts the fact that c is in the minor set of $L(A)$.

To construct the minor set $L(A)$, we can use an enumerative algorithm which compares any two configurations $c = (\underline{q}, b, \underline{z}) \in L(A)$ and $c' = (\underline{q}, b', \underline{z}') \in L(A)$ with $|b|, |b'| \le (|P| + |P||X|)(|S|+1)$, and discards the non-minimal one. ∎

Lemma 5.3. *We can compute an SB-automaton A^{init} such that $L(A^{init}) = \mathtt{Init}_{SB}$. For a set* $\mathtt{Target}$ *of local state definitions, we can compute an SB-automaton $A(\mathtt{Target})$ such that $L(A(\mathtt{Target})) := \{(\underline{q}, b, \underline{z}) \,|\, \underline{q} \in \mathtt{Target}\}$.*

Proof Recall that 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 $\underline{z_{init}}(p) = 1$. Moreover, the buffer contains a single message of the form (mem, p, x), where $p \in P$, $x \in X$, and mem represents some value of the memory. We can construct the SB-automaton $A^{init} = (S_0, \Delta_0, S_0^{final}, h_0)$ such that $L(A^{init}) = \mathtt{Init}_{SB}$ as follows:

- The set of states S_0 contains only three states s_{init}, s_{error}, and s_{final}.
- The set of transitions Δ_0 is the smallest set such that for every SB-message (mem, p, x), where $p \in P$, $x \in X$, and mem represents some value of the memory, we have $(s_0, ((mem, p, x), P), s_{final})$ is in Δ_0.
- The set of final states S_0^{final} contains only the state s_{final}.
- The function h_0 is defined as follows $h_0(\underline{q}) = s_{init}$ if $\underline{q} = \underline{q_{init}}$ and $h_0(\underline{q}) = s_{error}$ otherwise.

Then, it is easy to see that $L(A^{init}) = \mathtt{Init}_{SB}$.

Let $\mathtt{Target}$ be a set of local state definitions. We recall that an SB-configuration c is said to be *balanced* if $\mathtt{PointersOf}(c)(p) = \mathtt{PointersOf}(c)(p')$ for all $p, p' \in P$. Here, we can restrict ourselves to balanced configurations as if $\mathtt{Target}$ are reachable in some configurations, one can fire some updates to obtain balanced configurations. Then, we can compute the SB-automaton $A(\mathtt{Target}) = (S, \Delta, S^{final}, h)$ as follows:

- The set of states S contains only three states s_i, s_e, and s_f.
- The set of transitions Δ is the smallest set such that the following condition is satisfied: For every SB-message (mem, p, x), where $p \in P$, $x \in X$, and mem represents some value of the memory, we have $(s_i, ((mem, p, x), \emptyset), s_i)$, $(s_i, ((mem, p, x), P), s_f)$, and $(s_f, ((mem, p, x), \emptyset), s_f)$ are in Δ.
- The set of final states S_0^{final} contains only the state s_f.
- The function h_0 is defined as follows $h_0(\underline{q}) = s_i$ if $\underline{q} \in \mathtt{Target}$ and $h_0(\underline{q}) = s_e$ otherwise.

Then, we can easily show that $L(A(\mathtt{Target})) := \{(\underline{q}, b, \underline{z}) \,|\, \underline{q} \in \mathtt{Target}\}$. ∎

Lemma 5.4 *For an SB-automaton A we can compute an SB-automaton $A\!\uparrow$ such that $L(A\!\uparrow) = L(A)\!\uparrow$.*

Proof Let us assume that A is given by the tuple $(S, \Delta, S^{final}, h)$. One way to construct the SB-automaton $A\!\uparrow$ is based on the use of Lemma D.1 which allows us to compute the finite set $\mathtt{Min}(L(A))$ of SB-configurations. Let us assume that $\mathtt{Min}(L(A)) = \{c_1, \ldots, c_n\}$. For every $i : 1 \le i \le n$, we can construct an SB-automaton A_i such that $L(A_i) = \{c_i\}\!\uparrow$. Let $A\!\uparrow$ be the SB-automaton defined as $\bigcup_{i=1}^n A_i$. Then, we can show that $L(A\!\uparrow) = L(A)\!\uparrow$ since $L(A)\!\uparrow = \mathtt{Min}(L(A))\!\uparrow$, $\mathtt{Min}(L(A))\!\uparrow = \bigcup_{i=1}^n L(A)_i\!\uparrow$, and $L(A\!\uparrow) = \bigcup_{i=1}^n L(A)_i\!\uparrow$. ∎

Lemma 5.5. *For a transition t and a SB-automaton A, we can compute a SB-automaton $\mathtt{Pre}_t(A)$ such that $L(\mathtt{Pre}_t(A)) = \mathtt{Pre}_t(L(A))$.*

Proof Let us assume that $A = \left(S, \Delta, S^{final}, h\right)$. We consider six cases depending on the form of the transition t:

- Nop: If $t = (q, \mathsf{nop}, q') \in \Delta_p$ with $p \in P$ then we can construct $A' = \left(S', \Delta', S'_{final}, h'\right)$ as follows: (1) the set of states of A' contains the set of states of A and a new state s_{error} such that $s_{error} \notin S$ (i.e., $S' = S \cup \{s_{error}\}$), (2) the set of transitions Δ' is equal to the set Δ, (3) the set of final states S'_{final} is defined by the set S^{final}, and (4) the function h' is defined as follows $h'(\underline{q}) = h(\underline{q}')$ for all $\underline{q}$ and $\underline{q}'$ such that $\underline{q}(p) = q$ and $\underline{q}' = \underline{q}\,[p \hookleftarrow q']$, and $h'(\underline{q}) = s_{error}$ otherwise.

- Write to store: If $t = (q, \mathsf{w}(x,v), q') \in \Delta_p$ with $p \in P$ then we can construct $A' = \left(S', \Delta', S'_{final}, h'\right)$ as follows: A' contains all states of A and two new states s_{error} and s_f (i.e., $S' = S \cup \{s_{error}, s_f\}$). The final state S'_{final} consists of the final state s_f (i.e., $S^{final'} = \{s_f\}$). The transition relation of A' is defined as the smallest relation such that: (1) A' contains all the transitions of A (i.e., $\Delta \subseteq \Delta'$), and (2) if $\left(s, \left(\left(mem', p', x'\right), \sigma\right), s'\right)$ and $\left(s', \left(\left(mem'\,[x \hookleftarrow v], p, x\right), \emptyset\right), s''\right)$ are two transitions of A' such that $s'' \in S^{final}$, then $\left(s, \left(\left(mem', p', x'\right), \sigma\right), s_f\right)$ is also a transition of A'. The function h' is defined as follows $h'(\underline{q}) = h(\underline{q}')$ for all $\underline{q}$ and $\underline{q}'$ such that $\underline{q}(p) = q$ and $\underline{q}' = \underline{q}\,[p \hookleftarrow q']$, and $h'(\underline{q}) = s_{error}$ otherwise.

- Update: If $t = \mathtt{update}_p$ with $p \in P$ then we can construct $A' = \left(S', \Delta', S'_{final}, h'\right)$ as follows: A' has as a set of states $S' = S \cup \Delta \cup (S \times \{F\})$. The set of final states is defined by the set $S'_{final} = S^{final} \times \{F\}$. The transition relation of A is defined as the smallest relation such that: (1) if (s, a, s') is a transition of A then (s, a, s') and ((s,F),a,(s',F)) are transitions of A', and (2) for every transitions $t = \left(s, \left(\left(mem', p', x'\right), \sigma'\right), s'\right) \in \Delta$ and $t' = \left(s', \left(\left(mem'', p'', x''\right), \sigma'' \cup \{p\}\right), s''\right) \in \Delta$, A' contains the following transitions $\left(s, \left(\left(mem', p', x'\right), \sigma' \cup \{p\}\right), t'\right)$ and $\left(t', \left(\left(mem'', p'', x''\right), \sigma''\right), (s'', F)\right)$. Finally, we have $h' = h$.

- Read: If $t = (q, \mathsf{r}(x,v), q') \in \Delta_p$ with $p \in P$ then we can construct $A' = \left(S', \Delta', S'_{final}, h'\right)$ as follows: A' has as a set of states $S' = S \cup (S \times \{1\}) \cup \{s_{error}\}$. The set of final states is defined by the set $S'_{final} = S^{final} \times \{1\}$. The transition relation of A' is defined as the smallest relation such that: if (s, a, s') is a transition of A then there are three cases depending on a: (i) if a is not of the form $\left(\left(mem, p', x'\right), \sigma \cup \{p\}\right)$ or $\left(\left(mem, p, x\right), \sigma\right)$, then (s, a, s') and $((s,1), a, (s',1))$ are transitions of A', else (ii) if a is of the form $\left(\left(mem, p', x'\right), \sigma \cup \{p\}\right)$ or $\left(\left(mem, p, x\right), \sigma\right)$ with $mem(x) = v$, then $(s, a, (s',1))$ and $((s,1), a, (s',1))$ are transitions of A', otherwise (iii) if a is of the form $\left(\left(mem, p', x'\right), \sigma \cup \{p\}\right)$ or $\left(\left(mem, p, x\right), \sigma\right)$ with $mem(x) \neq v$, then $((s,1), a, (s',1))$ is a transition of A'. The function h' is defined as follows $h'(\underline{q}) = h(\underline{q}')$ for all $\underline{q}$ and $\underline{q}'$ such that $\underline{q}(p) = q$ and $\underline{q}' = \underline{q}\,[p \hookleftarrow q']$, and $h'(\underline{q}) = s_{error}$ otherwise.

- ARW: If $t = (q, \mathsf{arw}(x, v, v'), q') \in \Delta_p$ with $p \in P$ then we can construct $A' = (S', \Delta', S'_{final}, h)$ as follows: A' has as a set of states $S' = S \cup \{s_f, s_{error}\}$ where s_f and s_{error} are new states. The set of final states is defined by the set $S'_{final} = \{s_f\}$. The transition relation of A' is defined as the smallest relation such that: (1) $\Delta \subseteq \Delta'$, and (2) if $\left(s, \left(\left(mem', p', x'\right), \sigma\right), s'\right)$ and $\left(s', \left(\left(mem'\,[x \hookleftarrow v'], p, x\right), \{p\}\right), s''\right)$ are transitions of A with $s'' \in S^{final}$ and $mem'(x) = v$, then $\left(s, \left(\left(mem', p', x'\right), \sigma \cup \{p\}\right), s_f\right)$ is a transition of A'. The function h' is defined as follows $h'(\underline{q}) = h(\underline{q}')$ for all $\underline{q}$ and $\underline{q}'$ such that $\underline{q}(p) = q$ and $\underline{q}' = \underline{q}\,[p \hookleftarrow q']$, and $h'(\underline{q}) = s_{error}$ otherwise.

- Fence: $t = (q, \mathsf{fence}, q') \in \Delta_p$ with $p \in P$ then we can construct $A' = (S', \Delta', S'_{final}, h)$ as follows: A' has as set of states $S' = S \cup \{s_f, s_{error}\}$ where s_f and s_{error} are new states. The set of final states is defined by the set $S'_{final} = \{s_f\}$. The transition relation of A' is defined as the smallest relation such that: (1) $\Delta \subseteq \Delta'$, and (2) if $\left(s, \left(\left(mem', p', x'\right), \sigma \cup \{p\}\right), s'\right)$ is a transition of A with $s' \in S^{final}$, then $\left(s, \left(\left(mem', p', x'\right), \sigma \cup \{p\}\right), s_f\right)$ is a transition of A'. The function h' is defined as follows $h'(\underline{q}) = h(\underline{q}')$ for all $\underline{q}$ and $\underline{q}'$ such that $\underline{q}(p) = q$ and $\underline{q}' = \underline{q}\,[p \hookleftarrow q']$, and $h'(\underline{q}) = s_{error}$ otherwise. ∎

Theorem 5.6. *The reachability algorithm always terminates returning the correct answer.*

Proof It follows from Lemmata 5.3, 5.1, 5.4, 5.2, and 5.5 and from the properties of well structured transition systems [1]. ∎

E. Correctness of Algorithm 2 for fence insertion

Theorem 6.1. *For a concurrent system $\mathcal{P}$, a placement constraint G, and a finite set* `Target`*, Algorithm 2 terminates and returns* $F^G_{min}(\mathcal{P})(\mathtt{Target})$.

Proof We show termination and partial correctness of Algorithm 2. We start with termination and define the function $rank(\mathcal{W}, \mathcal{C}) = (n_0, \ldots n_K)$ where K is the total number of local states in $\mathcal{P}$, i.e., $K = |Q|$, and n_i is a pair $(n_i^{\mathcal{W}}, n_i^{\mathcal{C}})$ where $n_i^{\mathcal{W}}$ is the size of $\left\{F \mid F \in \mathcal{W} \text{ with } |F| = i\right\}$ ($n_i^{\mathcal{C}}$ is defined analogously). We write $n <_p n'$, for $n = (a,b)$ and $n' = (a',b')$, to mean $(a < a' \vee (a = a' \wedge b < b'))$. We define the lexicographic ordering $<_{lex}$ as follows: $(n_0, \ldots n_K) <_{lex} \left(n'_0, \ldots n'_K\right)$ iff $\exists i : 0 \leq i \leq K.\left(n_i <_p n'_i \wedge \forall j : 0 \leq j < i.\left(n_j \leq_p n'_j\right)\right)$. Observe that there will never be sets with a cardinality larger or equal to $K+1$ in $\mathcal{W}$ or $\mathcal{C}$. We show $rank(\mathcal{W}, \mathcal{C})$ strictly decreases at each iteration of the loop. Assume $\mathcal{W}', \mathcal{C}'$ are obtained from $\mathcal{W}, \mathcal{C}$ after one iteration of the loop; then $\mathcal{W}', \mathcal{C}'$ are derived by removing a set F from $\mathcal{W}$ and possibly: i) adding a number of sets with strictly larger cardinality to $\mathcal{W}$, or ii) moving F to $\mathcal{C}$. Each one of these operations strictly decreases $rank(\mathcal{W}, \mathcal{C})$, which proves termination by well foundedness of $<_{lex}$ on the image of *rank*.

In case `Target` is reachable even if all fences in G are inserted (in particular if reachable under SC-semantics), then the algorithm will never get to line 13, and will either exit at line 8 or keep on adding sets included in G to $\mathcal{W}$. Termination ensures it will eventually exit at line 8 with $\mathcal{C} = \emptyset$. Concretely, this will be due to a trace that does not involve overtakings of write operations (hence possible under SC-semantics) or that involves only overtakings of write operations along paths that do not intersect G (hence impossible to avoid by placing fences from G).

In the following, we assume `Target` unreachable if all fences from G are inserted, and say that a set of fences is complete if it suffices to ensure `Target` unreachable under TSO. We say that a set is incomplete otherwise. We show the following invariant to hold for the while loop. At each iteration, i) all sets in $\mathcal{W} \cup \mathcal{C}$ are subsets of G, ii) each set in $F^G_{min}(\mathcal{P})(\mathtt{Target})$ has a subset in $\mathcal{W} \cup \mathcal{C}$, and iii) the set $\mathcal{W} \cup \mathcal{C}$ is minimal with respect to the subset relation (i.e., $F_i \not\subseteq F_j$ for all F_i, F_j in $\mathcal{W} \cup \mathcal{C}$). Observe that this suffices, taking into account that $\mathcal{C}$ only contains complete sets of fence, for partial correctness as $\mathcal{W} = \emptyset$ implies $\mathcal{C} = F^G_{min}(\mathcal{P})(\mathtt{Target})$.

The statement holds at the loop entrance as $\mathcal{C}$ is empty and $\mathcal{W}$ contains the empty set. Suppose the statement holds at the beginning of some iteration, we show it is preserved by the iteration. New sets are obtained by adding a fence from F_B to a set from $\mathcal{W}$. Both are subsets of G, and hence all manipulated sets in the algorithm are subsets of G. If a set of fences is added to $\mathcal{C}$ at line 14, then it is complete (as it reached line 13) and the addition preserves minimality of $\mathcal{W} \cup \mathcal{C}$ as we just moved F from $\mathcal{W}$ to $\mathcal{C}$. We show in the following that each minimal set of fences has a subset in $\mathcal{W} \cup \mathcal{C}$. Suppose the set F picked at line 4 is not subset to a minimal set. The set F will not eliminate sets from $\mathcal{W}$ or $\mathcal{C}$, and the statement holds after the loop iteration. Otherwise, F is subset of a number of minimal sets or a minimal set itself. If it is a minimal set and it does not already belong to $\mathcal{C}$, it will pass the test at line 13 as $\mathcal{C}$ only contains complete sets. If it is a strict subset to minimal sets $F_1, \ldots, F_n$, then there should be a (possibly identical) representative in F_B from each set $F_i \setminus F$, as otherwise the trace δ is possible even in $\mathcal{P} \oplus F_i$; implying F_i is not complete, which contradicts F_i being a minimal fence set. This means that each of the minimal sets $F_1, \ldots, F_n$ will have a (possibly shared) subset among those F' generated at line 10. If one of the F' is discarded at line 11, then the corresponding F_i has another subset already in $\mathcal{W} \cup \mathcal{C}$ (namely the one that discarded F'). Hence, in all cases the statement is preserved which shows partial correctness. ∎