Precise and Sound Automatic Fence Insertion Procedure under PSO [⋆]

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Magnus Lång, and Tuan Phong Ngo

Department of Information Technology, Uppsala University, Sweden
{parosh, mohamed_faouzi.atig, magnus.lang.7837, tuan-phong.ngo}@it.uu.se

Abstract. We give a sound and complete procedure for fence insertion for concurrent finite-state programs running under the PSO memory model. This model allows "write to read" and "write-to-write" relaxations corresponding to the addition of an unbounded store buffers between processors and the main memory. We introduce a novel machine model, called the Hierarchical Single-Buffer (HSB) semantics, and show that the reachability problem for a program under PSO can be reduced to the reachability problem under HSB. We present a simple and effective backward reachability analysis algorithm for the latter, and propose a counter-example guided fence insertion procedure. The procedure infers automatically a minimal set of fences that ensures correctness of the program. We have implemented a prototype and run it successfully on all standard benchmarks, together with several challenging examples.

1 Introduction

For performance reasons, most of the modern architectures implement *weak memory models* [16,5]. Such models allows the reordering of memory instructions issued by the set of processes. For instance, the most common reordering is "write to read" which allows that writes to shared memory may be delayed past subsequent reads from memory. The "write to read" reordering leads to the *Total Store Order* (TSO) memory model that is adopted by Sun's SPARC and x86 architectures [22,23]. Adding the "write to write" reordering to TSO leads to the *Partial Store Order* (PSO) memory model (described in the Sun's SPARC architecture [24]). The "write to write" reordering may swap the order between two writes of the same process if they concern different variables.

The gain in the performance through the use of weak memory models comes with a price since reasoning about the behavior of even very small programs running under weak memory models is more difficult and counter-intuitive than under the usual *Sequentially Consistent* (SC) memory model. In fact, the SC memory model is the one that is usually assumed by the programmers where

[⋆] This work was supported in part by the Swedish Research Council and carried out within the Linnaeus centre of excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center.

the program instructions of different processes should appear as if these instructions are interleaved in a consistent global order. This means that a program under weak memory models can deviate from its intended behaviour (under the SC model) and hence violates its specifications. For example, several mutual exclusion algorithms and produce-consumer protocols become incorrect when executed under weak memory models. To avoid such undesired behaviours, programmers can use special *memory fence* instructions that prevent some reordering of instructions issued before and after the fence. Then, an important problem is to find the set of fences that ensures the correctness of programs when run under a weak memory model without compromising the performance. In fact, inserting too many fences would result in a degradation of program performance.

In this paper, we present the first *precise* and *sound* method for automatic fence insertion for concurrent finite-state programs running under the PSO memory model. To this end, we make the following contribution:

• We propose a *new model*, called the *Hierarchical Single-Buffer* (HSB), that is equivalent to the PSO memory model and allows the application of efficient infinite state model-checking techniques.

• A *simple* and *effective algorithm* to solve the reachability problem under HSB, using a backward analysis algorithm.

• A *fence insertion procedure* that infers a minimal fence set in order to correct programs under PSO.

• A *prototype* that is integrated to Memorax [2,1,3]. We evaluate our prototype on a wide range of benchmarks. The download link can be seen in Section 6.

Related Work Weak memory models are an active research area today. Many techniques have been developed to help programmers, in the form of precise model-checking algorithms (e.g., [9,10,12]), monitoring and testing tools (e.g., [13,14,21]), explicit state-space exploration (e.g., [20,19]), bounded model checking (e.g., [17,25,8]) and program transformations (e.g., [11,7,12]). Most of these works have focused on different memory models than PSO and thus are not directly comparable. Almost all the existed works on the PSO memory model are either (i) based on under-approximation techniques and which leads to sound but potentially imprecise analysis (e.g., [20,14]), or (ii) based on over-approximations techniques and which leads to potentially unsound analysis (e.g., [19,6,15]). Finally, checking safety property for finite-state programs running under TSO and PSO memory models has been shown to be decidable with a non-primitive recursive complexity [9,10]. A tool implementing an exact procedure for checking safety properties for programs running under TSO was presented in [2,1,3]. Our reachability algorithms can be seen as an efficient instance of the work [10] to the PSO memory model. Moreover, [10] does not discuss fence insertion.

2 Preliminaries

In this section, we introduce some notations and definitions that we use later.

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 $|\epsilon| = 0$. 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 $w \neq \epsilon$ 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 $|w_1| = i$.

Set Ordering Given an ordering $\sqsubseteq$ on C, we say that $\sqsubseteq$ is a *well-quasi ordering* if for every (infinite) sequence $c_0, c_1, \ldots$ in C, there are $i < j$ with $c_i \sqsubseteq c_j$. The *upward closure* of a set C wrt. $\sqsubseteq$ is defined as $C\!\uparrow := \{c' \mid \exists c \in C, c \sqsubseteq c'\}$. A set C is *upward closed* if $C = C\!\uparrow$. We use $\texttt{Min}\,(C)$ to denote the *minor set* of a given set C wrt. $\sqsubseteq$, that satisfies the following conditions: (i) for all $c \in C$ there is a $c' \in \texttt{Min}\,(C)$ such that $c' \sqsubseteq c$, and (ii) for all $c, c' \in \texttt{Min}\,(C)$, $c \neq c'$ implies $c \not\sqsubseteq c'$.

Transition System A transition system $\mathcal{T}$ is a triple $(C, \texttt{Init}, \rightarrow)$ where C is a (infinite) set of *configurations*, $\texttt{Init} \subseteq C$ is a set of *initial configurations*, and $\rightarrow \subseteq C \times C$ is a reflexive *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 $c_0 \rightarrow \cdots \rightarrow c_n$, where $c_i \rightarrow c_{i+1}$ for all $i : 0 \leq i < n$. Then, we write $c_0 \xrightarrow{\pi} c_n$. We use *target* (π) to denote 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 $c_0 \in \texttt{Init}$. A configuration c is said to be reachable if there is a computation π such that $c = \textit{target}\,(\pi)$. 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 $c_2 \xrightarrow{\pi} c_2'$.

3 Concurrent Programs under PSO

A *concurrent program* $\mathcal{P}$ has a finite number of finite-state processes, each with its own program code. Communication between processes is performed by reading and writing through a shared-memory with finite number of shared variables and finite domains. First, we introduce the PSO semantics (similar to the one described in [20]) and its reachability problem. Then we propose a new model, the HSB model, that we use to analyse programs under the PSO model.

3.1 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 $A = \{A_p |\ p \in P\}$ 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 six forms: (i) the *"no operation"* nop, (ii) the *read operation* $\mathsf{r}(x, v)$, (iii) the *write operation* $\mathsf{w}(x, v)$, (iv) the *full fence operation* mfence, (v) the *write-write fence operation* sfence, and (vi) 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 $p \in P$.

3.2 PSO Semantics

Transition System We define the transition system induced by a program running under the PSO semantics. To do that, we define the set of configurations and transition relation. A *PSO-configuration* c is a triple $\left(\underline{q}, \underline{b}, mem\right)$ where $\underline{q}$ is a local state definition, $\underline{b} : P \mapsto \left[X \mapsto (V \cup \{\star\})^*\right]$, and $mem : X \mapsto V$. Intuitively, $\underline{q}(p)$ gives the local state of process p. The value of $\underline{b}(p)(x)$ is the content of the buffer belonging to variable x of p. This buffer associates a sequence of values from V to the variable x, where each value v represents a write operation that assigns v to the variable x. The buffer may also contain the *write-write fence* symbol $\star$ that restricts the ordering of writes. In our model, writes will be appended to *the tail of buffer* (the right most one), and fetched from *the head of buffer* (the left most one). The head of buffer $\underline{b}(p)(x)$ is at the index 1, and the tail of buffer is at the index $|\underline{b}(p)(x)|$. Finally, mem defines the state of the memory (defines the value of each variable in the memory). We use C_{PSO} to denote the set of PSO-configurations.

We define the transition relation $\rightarrow_{PSO}$ on C_{PSO}. The relation is induced by (i) members of Δ; (ii) a set $\Delta' := \left\{\mathsf{update}_{p,x} |\ p \in P, x \in X\right\}$ where $\mathsf{update}_{p,x}$ is an operation that updates the memory using the message at the head of the buffer for variable x of process p; and (iii) a set $\Delta'' := \left\{\mathsf{update}_{p,\star} |\ p \in P\right\}$ where $\mathsf{update}_{p,\star}$ removes the write-write fence symbol from the head of all the buffers of process p. For configurations $c = \left(\underline{q}, \underline{b}, mem\right)$, $c' = \left(\underline{q}', \underline{b}', mem'\right)$, a process $p \in P$, and $t \in \Delta_p \cup \left\{\mathsf{update}_{p,x}, \mathsf{update}_{p,\star}\right\}$, we write $c \xrightarrow{t}_{PSO} c'$ 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: $t = (q, \mathsf{w}(x, v), q')$, $\underline{q}(p) = q$, $\underline{q}' = \underline{q}\,[p \hookleftarrow q']$, $\underline{b}' = \underline{b}\,[p \hookleftarrow \underline{b}(p)\,[x \hookleftarrow \underline{b}(p)(x) \cdot v]]$, and $mem' = mem$. The write operation is appended to the tail of the buffer for variable x of process p.

• Memory update: $t = \mathsf{update}_{p,x}$, $\underline{q}' = \underline{q}$, $\underline{b} = \underline{b}' \left[p \hookleftarrow \underline{b}'(p) \left[x \hookleftarrow v \cdot \underline{b}'(p)(x)\right]\right]$, and $mem' = mem\,[x \hookleftarrow v]$. The write at the head of the buffer for x of p is removed and the memory is updated accordingly.

• Write-write fence update: $t = \mathsf{update}_{p,\star}$, $\underline{q}' = \underline{q}$, $\forall x \in X : \underline{b} = \underline{b}' \left[p \hookleftarrow \underline{b}'(p)\right] \left[x \hookleftarrow \star \cdot \underline{b}'(p)(x)\right]$, and $mem' = mem$. The write-write fence symbol $\star$ is removed from the head of all buffers of process p.

• 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. (i) Read own write: There is an $i : 1 \leq i \leq |\underline{b}(p)(x)|$ such that $\underline{b}(p)(x)(i) = v$, and $v' \notin (\underline{b}(p)(x) \odot i)$ for all $v' \in V$. If there is a write in the buffer for x of p then we consider the write at the tail of the buffer (the right most one of the buffer). This operation should assign v to x. (ii) Read memory: $v' \notin \underline{b}(p)(x)$ for all $v' \in V$ and $mem(x) = v$. If there is no write operation in the buffer for x of p then the value v of x is fetched from the memory.

• Full fence: $t = (q, \mathsf{mfence}, q')$, $\underline{q}(p) = q$, $\underline{q}' = \underline{q}\,[p \hookleftarrow q']$, $\forall x \in X : \underline{b}(p)(x) = \epsilon$, $\underline{b}' = \underline{b}$, and $mem' = mem$. A full fence operation may be performed by a process only if all its buffers are empty.

• Write-write fence: $t = (q, \mathsf{sfence}, q')$, $\underline{q}(p) = q$, $\underline{q}' = \underline{q}\,[p \hookleftarrow q']$, $\forall x \in X : \underline{b}' = \underline{b}\,[p \hookleftarrow \underline{b}(p)\,[x \hookleftarrow \underline{b}(p)(x) \cdot \star]]$, and $mem' = mem$. A write-write fence operation adds the symbol $\star$ to the tail of all buffers of process p.

• ARW: $t = (q, \mathsf{arw}(x,v,v'), q')$, $\underline{q}(p) = q$, $\underline{q}' = \underline{q}\,[p \hookleftarrow q']$, $\underline{b}(p)(x) = \epsilon$, $\underline{b}' = \underline{b}$, $mem(x) = v$, and $mem' = mem\,[x \hookleftarrow v']$. The operation $\mathsf{arw}(x,v,v')$ is performed atomically. It may be performed by a process only if its buffer for x 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 compare-and-swap (or test-and-set) under SPARC [24].

We use $c \rightarrow_{PSO} c'$ to denote the reflexive closure of $c \xrightarrow{t}_{PSO} c'$ for some $t \in \Delta \cup \Delta' \cup \Delta''$. The set $\mathtt{Init}_{PSO}$ of *initial* PSO-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 $\underline{b_{init}}(p)(x) = \epsilon$ for all $x \in X$. 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 PSO semantics is then given by $(C_{PSO}, \mathtt{Init}_{PSO}, \rightarrow_{PSO})$.

The PSO Reachability Problem Given a set `Target` of local state definitions, we use $Reachable(PSO)\,(\mathcal{P})\,(\mathtt{Target})$ to be a predicate that indicates the reachability of one of the following configurations $\left\{\left(\underline{q}, \underline{b}, mem\right) \middle|\ \underline{q} \in \mathtt{Target}\right\}$, i.e., whether a configuration c, where the local state definition of c belongs to `Target`, is reachable. The reachability problem for PSO is to check, for a given `Target`, whether $Reachable(PSO)\,(\mathcal{P})\,(\mathtt{Target})$ holds or not. We use `Target` to denote *"bad configurations"* that we do not want to occur during the execution of the system. Therefore, we often say that the *"program is correct (or safe)"* to indicate that `Target` is not reachable.

3.3 Hierarchical Single-Buffer Semantics

The PSO semantics make use of *unbounded perfect FIFO buffers* that induces an infinite transition system. However, the reachability problem under PSO is still decidable as shown in [9,10]. In fact, it can be solved using the framework of well-structured transition systems [4]. For the case of TSO, the paper [2] proposes an ordering partly based on the sub-word relations of the configuration's buffer contents. However, because PSO configurations can contain the $\star$ symbol (which can not be lost), a similar ordering is not monotonic wrt. the PSO semantics. Therefore, our goal is to derive a new semantical model, called the *Hierarchical Single-Buffer model (HSB)*, that is both equivalent to PSO wrt. reachability problems and monotonic wrt. some ordering. The buffer contents of HSB configurations will not contain $\star$ symbol.

Formal Semantics A *HSB-configuration* c is a quadruple $\left(\underline{q}, \underline{b}, m, \underline{z}\right)$ where $\underline{q}$ is (as in the case of the PSO semantics) a local state definition, $\underline{b} : P \mapsto [X \mapsto V^*]$, $m \in ([X \mapsto V] \times P \times X)^+$, and $\underline{z} : P \mapsto \mathbb{N}$. Intuitively, $\underline{b}(p)(x)$ is a per process and variable buffer, the *channel* m contains *messages* as 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 by the process p. Furthermore, $\underline{z}$ represents a set of *pointers* (one for each process) where, from the point of view of p, the word $m \odot \underline{z}(p)$ is the sequence of write operations that have not yet been used for memory updates and the first element of the triple $m(\underline{z}(p))$ represents the memory content. We use C_{HSB} to denote the set of HSB-configurations. As we shall see below, the channel 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 HSB semantics, the update operation moves a pointer to the right instead of removing a message in the channel). This implies (among other things) that the invariant $\underline{z}(p) > 0$ is always maintained. Messages are appended to *the tail of the channel* (the right most one) that has index $|m|$. The *bottom of channel*, index 1, is the *initial message*.

Let $c = \left(\underline{q}, \underline{b}, m, \underline{z}\right)$ be a HSB-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 channel 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 channel. Formally, $\texttt{LastWrite}\,(c, p, x)$ is the largest index $i : \underline{z}(p) \leq i \leq |m|$, such that $m(i) = (mem, p, x)$ for some mem, or $i = \underline{z}(p)$ if such $m(i)$ does not exist.

We define the transition relation $\rightarrow_{HSB}$ on the set of HSB-configurations as follows. For configurations $c = \left(\underline{q}, \underline{b}, m, \underline{z}\right)$, $c' = \left(\underline{q}', \underline{b}', m', \underline{z}'\right)$, and $t \in \Delta_p \cup \left\{\mathsf{update}_p, \mathsf{serialize}_{p,x}\right\}$ where update_p is an operation that updates memory from the view point of p by increasing $\underline{z}(p)$ by one, and $\mathsf{serialize}_{p,x}$ is an operation that serialises the write (the left most one) at the head of the buffer $\underline{b}(p)(x)$ into a new message at the tail of m, we write $c \xrightarrow{t}_{HSB} c'$ to denote that one of the following conditions is satisfied:

- Nop: $t = (q, \mathsf{nop}, q')$, $\underline{q}(p) = q$, $\underline{q}' = \underline{q}\,[p \hookleftarrow q']$, $\underline{b}' = \underline{b}$, $m' = m$, and $\underline{z}' = \underline{z}$. The operation changes only local states.

• Write to store: $t{=}(q, \mathsf{w}(x,v), q')$, $\underline{q}(p){=}q$, $\underline{q}'{=}\underline{q}\,[p \hookleftarrow q']$, $\underline{b}' = \underline{b}\,[p \hookleftarrow \underline{b}(p)\,[x \hookleftarrow \underline{b}(p)(x) \cdot v]]$, $m' = m$, and $\underline{z}' = \underline{z}$. The write operation is added to the tail of $\underline{b}(p)(x)$.

• Serialize: $t{=}\mathsf{serialize}_{p,x}$, $\underline{q}' = \underline{q}$, $\underline{b} = \underline{b}'\,\left[p \hookleftarrow \underline{b}'(p)\,\left[x \hookleftarrow v \cdot \underline{b}'(p)(x)\right]\right]$, $m(|m|)$ is of the form (mem_1, p_1, x_1), $m' = m \cdot (mem_1\,[x \hookleftarrow v]\,, p, x)$, and $\underline{z}' = \underline{z}$. A new message is serialised to the head of the channel. 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.

• Update: $t = \mathsf{update}_p$, $\underline{q}' = \underline{q}$, $\underline{b}' = \underline{b}$, $m' = m$, $\underline{z}(p) < |m|$ and $\underline{z}' = \underline{z}\,[p \hookleftarrow \underline{z}(p) + 1]$. 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.

• Read: $t = (q, \mathsf{r}(x,v), q')$, $\underline{q}(p) = q$, $\underline{q}' = \underline{q}\,[p \hookleftarrow q']$, $\underline{b}' = \underline{b}$, $m' = m$, $\underline{z}' = \underline{z}$, and one of the following conditions is satisfied: (i) Read own write: $\underline{b}(p)(x)(|\underline{b}(p)(x)|) = v$. If there is a write on x in the buffer for x of p then we consider the most recent of such write operations (the right most one in the buffer). (ii) Read memory: $m(\mathtt{LastWrite}\,(c,p,x)) = (mem_1, p_1, x_1)$ for some mem_1, p_1, x_1 with $mem_1(x) = v$, $\underline{b}(p)(x) = \epsilon$. If there is no write operation in the buffer for x of p then the value v of x is fetched from the memory. Note that $\underline{b}(p)(x)$ always does not contain the symbol $\star$.

• Full fence: $t = (q, \mathsf{mfence}, q')$, $\underline{q}(p) = q$, $\underline{q}' = \underline{q}\,[p \hookleftarrow q']$, $\underline{b}' = \underline{b}$, $\forall x \in X$: $\underline{b}(p)(x) = \epsilon$, $\underline{b}' = \underline{b}$, $m' = m$, $\underline{z}' = \underline{z}$, and $\underline{z}(p) = |m|$. A full fence operation may be performed by a process p only if all its buffers are empty, and process p is observing the most recent message.

• Write-write fence: $t = (q, \mathsf{sfence}, q')$, $\underline{q}(p) = q$, $\underline{q}' = \underline{q}\,[p \hookleftarrow q']$, $\underline{b}' = \underline{b}$, $\forall x \in X$: $\underline{b}(p)(x) = \epsilon$, $m' = m$, and $\underline{z}' = \underline{z}$. A write-write fence operation requires all previous writes of p to be serialised before continuing, hence a write of p cannot reorder past a sfence.

• ARW: $t = (q, \mathsf{arw}(x,v,v'), q')$, $\underline{q}(p) = q$, $\underline{q}' = \underline{q}\,[p \hookleftarrow q']$, $\underline{b}' = \underline{b}$, $\underline{b}(p)(x) = \epsilon$, $\underline{z}(p) = |m|$, $m(|m|)$ is of the form (mem_1, p_1, x_1), $mem_1(x) = v$, $m' = m \cdot (mem_1\,[x \hookleftarrow v']\,, p, x)$, and $\underline{z}' = \underline{z}\,[p \hookleftarrow \underline{z}(p) + 1]$. The fact that the buffer is empty from the point of view of p is encoded by the equality $\underline{z}(p) = |m|$. The content of the memory can then be fetched from the right most element $m(|m|)$ in the channel. To encode that the buffer is still empty after the operation (from the point of view of p) the pointer of p is moved one step to the right.

We define the sets $\mathsf{update} := \cup_{p \in P} \mathsf{update}_p$, $\mathsf{serialize}_x := \cup_{p \in P} \mathsf{serialize}_{p,x}$, and $\mathsf{serialize} := \cup_{x \in X} \mathsf{serialize}_x$. We use $c \rightarrow_{HSB} c'$ to denote that $c \xrightarrow{t}_{HSB} c'$ for some $t \in \Delta \cup \{\mathsf{update}, \mathsf{serialize}\}$. The set $\mathtt{Init}_{HSB}$ of *initial* HSB-configurations of the form $\left(\underline{q_{init}}, \underline{b_{init}}, m_{init}, \underline{z_{init}}\right)$ where $|m_{init}| = 1$, and for all $p \in P$, we have that $\underline{q_{init}}(p) = q_p^{init}$, $\underline{b_{init}}(p)(x) = \epsilon$, and $\underline{z_{init}}(p) = 1$. In other words, each process is in its initial local state. The channel 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 processes point to the first position in the channel. Moreover, all buffers are all empty. The transition system induced by a concurrent system under the HSB semantics is then given by $(C_{HSB}, \mathtt{Init}_{HSB}, \rightarrow_{HSB})$.

The HSB Reachability Problem In a similar manner to the case of PSO, we define the predicate $\mathit{Reachable}(\mathit{HSB})\,(\mathcal{P})\,(\mathtt{Target})$, and define the reachability problem for the HSB semantics. The following theorem states equivalence of the reachability problems under the PSO and HSB semantics.

Theorem 1. *For a finite-state program $\mathcal{P}$ and a local state definition* $\mathtt{Target}$*, the reachability problems are equivalent under the PSO and HSB semantics.*

4 The HSB Reachability Algorithm

We present an algorithm to check HSB reachability problem for a given set $\mathtt{Target}$. Then according to Theorem 1, we can solve the PSO reachability problem. First, we define an ordering $\sqsubseteq$ on the set of HSB-configurations. We then show that it satisfies two properties: (i) it is well-quasi ordering (wqo), and (ii) the HSB relation $\rightarrow_{HSB}$ is effectively monotonic wrt. $\sqsubseteq$. Recall that the term *well-quasi ordering* and *effectively monotonic* are defined in Section 2.

4.1 Ordering

We define $\mathtt{Active}\,(c) := \mathit{min}\,\{\underline{z}(p)|\, p \in P\}$ for a HSB-configuration $c = \big(\underline{q}, \underline{b}, m, \underline{z}\big)$. In other words, the part of m to the right of (and including) $\mathtt{Active}\,(c)$ is *"active"*, while the left part is *"dead"* in the sense that it is not needed for computations starting from c.

Given two HSB configurations $c = \big(\underline{q}, \underline{b}, m, \underline{z}\big)$ and $c' = \big(\underline{q}', \underline{b}', m', \underline{z}'\big)$. Define $j := \mathtt{Active}\,(c)$ and $j' := \mathtt{Active}\,(c')$. We write $c \sqsubseteq c'$ to denote that: • (i) $\underline{q} = \underline{q}'$; • (ii) for every $p \in P$ and $x \in X$, there is a mapping $g_{p,x} : \{1, 2, \ldots, |\underline{b}(p)(x)|\} \mapsto \big\{1, 2, \ldots, |\underline{b}'(p)(x)|\big\}$ such that the following conditions are satisfied: for every $i, i_1, i_2 \in \{1, 2, \ldots, |\underline{b}(p)(x)|\}$, (1) $i_1 < i_2$ implies $g_{p,x}(i_1) < g_{p,x}(i_2)$, and (2) $\underline{b}(p)(x)(i) = \underline{b}'(p)(x)(g_{p,x}(i))$; • (iii) there is a mapping $h : \{j, j+1, \ldots, |m|\} \mapsto \{j', j'+1, |m'|\}$ such that the following conditions are satisfied: for every $i, i_1, i_2 \in \{j, j+1, \ldots, |m|\}$, (1) $i_1 < i_2$ implies $h(i_1) < h(i_2)$, (2) $m(i) = m'(h(i))$, (3) $\mathtt{LastWrite}\,(c', p, x) = h(\mathtt{LastWrite}\,(c, p, x))$ for all $p \in P$ and $x \in X$, (4) $\underline{z}'(p) = h(\underline{z}(p))$ for all $p \in P$; • (iv) For every $p \in P$ and $x \in X$, one of the following condition holds: (1) if $\underline{b}(p)(x)(|\underline{b}(p)(x)|) = v$ then $\underline{b}'(p)(x)(|\underline{b}'(p)(x)|) = v$, or (2) if $\underline{b}(p)(x) = \epsilon$ then $\underline{b}'(p)(x) = \epsilon$.

The conditions (ii-1) and (iii-1) mean that g and h are strictly monotonic. The condition (ii) indicates that $\underline{b}(p)(x)$ is a *sub-word* of $\underline{b}'(p)(x)$. The conditions (iii-1,2) present the active part of m is a *sub-word* of the active part of m'. The conditions (iii-2,3) ensure the last write indices wrt. all processes and variables are consistent. The conditions (iii-2,4) ensure each process points to identical elements in m and m'. The last condition (iv) shows that the two buffer are empty or contains the same element at the tail of buffers (the right most ones).

We get the following lemma about the ordering on HSB-configurations.

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

Proof. The lemma is an immediate consequence of the fact that: (1) the subsequence relations (ii) and (iii) are well-quasi orderings on finite words [18], and (2) the number of states (i), pointers (iii-4), observed memory states (iii-2), and last writes (iii-3) and (ii) that should be equal, is finite.

The following lemma shows the effectively monotonicity of HSB-transition relation wrt. $\sqsubseteq$.

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

Proof. We show that give HSB-configurations c_1, c_1', and c_2 such that $c_1 \rightarrow_{HSB} c_1'$ and $c_1 \sqsubseteq c_2$, there exists an HSB-configuration c_2' and a run π satisfying: $c_2 \xrightarrow{\pi}_{HSB} c_2'$ and $c_1' \sqsubseteq c_2'$. Let h and $g_{p,x}$ be the mappings defined by $c_1 \sqsubseteq c_2$. We will consider each transition $t \in \Delta_p \cup \{\mathsf{update}_p, \mathsf{serialize}_p\}$ for some $p \in P$ such that $c_1 \xrightarrow{t}_{HSB} c_1'$, and show that $c_2 \xrightarrow{\pi}_{HSB} c_2'$ for some c_2' and π. Then because a run is a concatenation of some transitions, we have the proof.

• Nop: $t = (q, \mathsf{nop}, q')$, select c_2' such that $c_2 \xrightarrow{t}_{HSB} c_2'$. Because nop operation only change the local state of c_1 to c_1' and of c_2 to c_2', and $c_1 \sqsubseteq c_2$, we have $c_1' \sqsubseteq c_2'$.

• Write to store: $t = (q, \mathsf{w}(x, v), q')$, select c_2' such that $c_2 \xrightarrow{t}_{HSB} c_2'$. We add a value to $\underline{b}(p)(x)$ of c_1, and we add the same value to $\underline{b}(p)(x)$ of c_2. Hence the condition (ii) and (iv) hold. Because in this transition we only change the buffers and local states, and $c_1 \sqsubseteq c_2$, we have $c_1' \sqsubseteq c_2'$.

• Read: $t = (q, \mathsf{r}(x, v), q')$, select c_2' such that $c_2 \xrightarrow{t}_{HSB} c_2'$. We do not change the buffers and channel, and require process p to observe x with value v. Because conditions (iii-3) and (iv), c_2' exists. Because of $c_1 \sqsubseteq c_2$, we have $c_1' \sqsubseteq c_2'$.

• Serialize: t=$\mathsf{serialize}_{p,x}$. This transition takes an element from buffer for x of p and send a message to channel. The same element exists in c_2 and will make the same message when serialised. However, there might be more elements in buffer for x of p of c_2 that must be serialised before that element can be reached. Select a run π as a sequence of serialised transitions of p on x will do this work. Formally, select $\pi = c_2 \underbrace{\xrightarrow{\mathsf{serialize}_{p,x}} \cdots \xrightarrow{\mathsf{serialize}_{p,x}}}_{g_{p,x}(1) \text{ times}} c_2'$. In other words, π will serialise all p's writes to x up to and including the one that corresponds to the one that is being serialised in c_1. Because π only removes the element from buffer for x of p, the same message is created at the end of channels of both c_1 and c_2. Since neither t nor π change the local states or pointers, and the serialised operation changes the `LastWrite` (c_1, p, x) and `LastWrite` (c_2, p, x) in both two configurations to a new consistent message, we have $c_1' \sqsubseteq c_2'$.

• Update: $t = \mathsf{update}_p$. This transition advances the pointer of p to a more recent message. However, the corresponding message in c_2 might not be immediately following the message currently pointed by p. But by performing several updates, the pointer in c_2 can be advanced to the message corresponding to the more recent message in c_1. Formally, select $\pi = c_2 \underbrace{\xrightarrow{\mathsf{update}_p} \cdots \xrightarrow{\mathsf{update}_p}}_{h(\underline{z}(p)+1)-h(\underline{z}(p)) \text{ times}} c_2'$. Since the pointer of p in c_1 has been forwarded from $\underline{z}(p)$ to $\underline{z}(p)+1$, and the pointer of p in c_2 has moved from $h(\underline{z}(p))$ to $h(\underline{z}(p)+1)$, (iii-4) holds. Also (iii-3) holds between c_1' and c_2' because of (iii-3,4) and $c_1 \sqsubseteq c_2$. Then we have $c_1' \sqsubseteq c_2'$.

• ARW: $t = (q, \mathsf{arw}(x, v, v'), q')$, select c_2' such that $c_2 \xrightarrow{t}_{HSB} c_2'$. This transition performs all read, write, serialise, and update as a single operation. Above we show that any operation of read, write, serialise, and update operations is an effectively monotonic operation. The arw requires p's buffer of x to be empty in c_2. This requirement holds because the p's buffer of x is empty in c_1 and because of (ii). The arw also requires the p's pointer to be on the last message, but it must be the case in c_2 if it is in c_1. Suppose that this requirement does not hold. Then because the pointer of p points to the last message in channel in c_1 (the one at the tail of channel), (1) the last message in channel of c_2 (the one at the tail of channel) does not have a corresponding message in channel of c_1. But (2) the last message of c_2 must be the $\texttt{LastWrite}\,(c_2, p', y)$ for some $p' \in P, y \in X$ (because some process must add more messages after the position $\texttt{LastWrite}\,(c_2, p, x)$). (1) and (2) make (iii-3) not hold. This is a contradiction. Thus c_2' exists, and $c_1' \sqsubseteq c_2'$.

• Full fence and write-write fence cases are trivial, because we do not change anything for buffers and channels.

4.2 Reachability Algorithm

Recall that the terms *upward closure*, *upward closed set*, and *minor set* are defined in Section 2. We define the *pre-set* $\texttt{Pre}\,(C)$ of a set C as $\texttt{Pre}\,(C) := \{c' \mid \exists c \in C, t \in \Delta \cup \{\mathsf{update}, \mathsf{serialize}\}, c' \xrightarrow{t}_{HSB} c\}$. Bellow we present our algorithm to check the HSB reachability problem using the ordering $\sqsubseteq$ that is well-quasi and monotonic wrt. $\rightarrow_{HSB}$. The algorithm performs backward reachability analysis from the set of configurations that are defined by $\texttt{Target}$. It inputs a finite set $\texttt{Target}$, and checks the predicate *Reachable*$(HSB)\,(\mathcal{P})\,(\texttt{Target})$. If the predicate does not hold then Algorithm 1 returns *"u"* (unreachable), otherwise it returns *"r"* (reachable). It maintains a *working set* $\mathcal{W}$ that contains *detected* configurations that need to be checked. If one of configuration in $\mathcal{W}$ can be reached by a configuration c *smaller* than the initial configurations (in the sense that there

Algorithm 1: Reachability Algorithm.

```
input  : A concurrent program $\mathcal{P}$, and a finite set
         Target of local state definitions.
output: "u" if $\neg Reachable(HSB)\,(\mathcal{P})\,(\texttt{Target})$, "r"
         otherwise.
$\mathcal{W} \leftarrow \texttt{Min}\left(\left\{\left(\underline{q}, \underline{b}, mem\right) \mid \underline{q} \in \texttt{Target}\right\}\right)$;
$\mathcal{F} \leftarrow \emptyset$;
while $\mathcal{W} \neq \emptyset$ do
    Pick, remove a configuration $c'$ from $\mathcal{W}$;
    $\mathcal{O} \leftarrow \texttt{Min}\,(\texttt{Pre}\,(\{c'\}\uparrow) \cup \{c'\})$;
    foreach $c \in \mathcal{O}$ do
        if $\exists c_0 \in \texttt{Init}_{HSB} : c \sqsubseteq c_0$ then return "r";
        if $\exists f \in \mathcal{F} : f \sqsubseteq c$ then discard $c$;
        else
            $\mathcal{W} \leftarrow \mathcal{W} \setminus \{w \in \mathcal{W} \mid c \sqsubseteq w\} \cup \{c\}$;
            $\mathcal{F} \leftarrow \mathcal{F} \cup \{c\}$;
return "u";
```

exists a computation c_0 from $\texttt{Init}_{HSB}$ such that $c \sqsubseteq c_0$), the finite set $\texttt{Target}$ also can be reachable (line 7). The set $\mathcal{F}$ is a set of all analysed configurations.

Initially, $\mathcal{W}$ has all elements from a minor set of $\texttt{Target}$, and $\mathcal{F}$ is an empty set. At the beginning of each iteration, the algorithm picks and removes a configuration c' from the set $\mathcal{W}$. Then it computes the set $\mathcal{O}$ that is a minor set of c' and all configurations that can reach a configuration in $\{c'\}\uparrow$ in one transition t, $t \in \Delta \cup \{\mathsf{update}, \mathsf{serialize}\}$. For each minor element c, it checks whether the element is smaller than an initial configuration. If *yes*, it returns *"r"*. If not, it checks whether c is *presented* in $\mathcal{F}$ (in the sense that $\mathcal{F}$ already has a configuration f such that $f \sqsubseteq c$). If *yes* then c can be discarded. Otherwise the algorithm performs the following operations: (i) discards all elements w of $\mathcal{W}$ that $c \sqsubseteq w$, (ii) adds to $\mathcal{W}$ the configuration c, and (iii) adds c to $\mathcal{F}$. The algorithm terminates when $\mathcal{W}$ is empty and return *"u"*.

Theorem 2. *The reachability algorithm always terminates.*

Proof. An immediate consequence of the framework of well-structured transition systems from [4] and the fact that it is possible to compute the finite sets $\texttt{Min}\left(\left\{\left(\underline{q}, \underline{b}, mem\right) \mid \underline{q} \in \texttt{Target}\right\}\right)$ and $\texttt{Min}\left(\texttt{Pre}\left(\{c'\}\uparrow\right) \cup \{c'\}\right)$ for a configuration c' in the same manner as done in [2].

We can modify the Alg. 1 to return a *trace* (if exists) from a configuration in $\texttt{Init}_{HSB}$ to a configuration in $\texttt{Bad} = \left\{\left(\underline{q}, \underline{b}, mem\right) \mid \underline{q} \in \texttt{Target}\right\}$ in the form $t_0 \cdot t_1 \ldots t_{n-1}$ such that there is a computation: $\pi = c_0 \xrightarrow{t_0} c_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} c_n$ with $c_0 \in \texttt{Init}_{HSB}$ and $c_n \in \texttt{Bad}$. Indeed, in the algorithm for each configuration c we keep the trace from this configuration to one configuration in $\texttt{Bad}$. Initially, all configurations in $\mathcal{W}$ have empty traces (line 1). There are two more positions in the algorithm we need to modify. At line 5, when we calculate the list of configurations $\texttt{Pre}\left(\{c'\}\uparrow\right)$, we add the corresponding transition to the current trace of c'. We do the similar modification in line 10.

5 Fence Insertion

In this section we describe our fence insertion procedure that given a set of *bad configurations*, we can find a minimal set of fences to avoid these configurations under PSO. A *minimal fence set* is the one sufficient for *correctness*; and if we remove any fences from this set, we violate the correctness. There are cases when these fence sets do not exist because the program can reach to bad configurations *even* under SC semantics. In this case we return an empty set. Bellow we fix a configuration $c_i = \left(\underline{q}_i, \underline{b}_i, m_i, \underline{z}_i\right)$ with $0 \leq i \leq n$.

Fence Inference. We will identify the set of points along a trace returned by Algorithm 1, $\pi = c_0 \xrightarrow{t_0} c_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} c_n$ with $c_0 \in \texttt{Init}_{HSB}$ and $c_n \in \texttt{Bad}$ with $\texttt{Bad} = \left\{\left(\underline{q}, \underline{b}, mem\right) \mid \underline{q} \in \texttt{Target}\right\}$, in which (i) read operations overtake write operations, or (ii) write operations overtake write operations, and derive the set of fences such that any one of them forbids an overtaking, $\mathsf{NewFences}(\pi) := \mathsf{NewFences}_{\mathsf{mfence}}(\pi) \cup \mathsf{NewFences}_{\mathsf{sfence}}(\pi)$. The set $\mathsf{NewFences}_{\mathsf{mfence}}(\pi)$ (or $\mathsf{NewFences}_{\mathsf{sfence}}(\pi)$) can prevent write-read overtaking (or write-write overtaking) in π.

First, we show how to find the set of $\mathsf{NewFences}_{\mathsf{mfence}}(\pi)$ for π. Define $n_i := |m_i| + \Sigma_{p \in P, x \in X} \underline{b}_i(p)(x)$. We define a sequence of functions $\alpha_0, \alpha_1, \ldots, \alpha_n$ where $\alpha_i(j)$ (with $1 \leq j \leq n_i$) associates to each element in the channel m_i or buffers $\underline{b}_i$ the position in π of the corresponding write transition. Note that the lowest index element (index 1) is the initial message in the channel, and the highest index element (index n_i) is the newest element added to buffers. We define $\alpha_0, \alpha_1, \ldots, \alpha_n$ in a recursive way. (i) At the beginning, c_0 contains only initial values in the channel, and all buffers are empty, $\alpha_0(j)$ is undefined for all $1 \leq j \leq n_0$. (ii) The first element in buffers and channel is the initial message in channel, therefore $\alpha_i(1)$ is undefined also. (iii) If t_{i+1} is not a *write operation* then the number of elements in buffers and channel are not changed, define $\alpha_{i+1} := \alpha_i$. (iv) Otherwise, we define $\alpha_{i+1}(j) := \alpha_i(j)$ if $2 \leq j \leq n_i$, and define $\alpha_{i+1}(n_i + 1) := i + 1$. The definition (iv) means that a new write operation will add a new element to the tail of one buffer, and for this element we associate $i+1$. Next, we find the write transitions that have been overtaken by *read operations*. We define a function $\mathtt{OverRead}$ such that if t_i (with $1 \leq i \leq n$) is a read transition then $\mathtt{OverRead}(\pi)(i)$ gives the positions of write transitions in π that have been overtaken by t_i. Formally, if t_i is not a read then define $\mathtt{OverRead}(\pi)(i) := \emptyset$. Otherwise, $t_i = (q, \mathsf{r}(x, v), q') \in \Delta_p$ for some $p \in P$, define $\mathtt{OverRead}(\pi)(i) := \left\{\alpha_i(j) \middle| \, \mathtt{LastWrite}\,(c_i, p, x) < j \leq n_i \wedge t_{\alpha_i(j)} \in \Delta_p\right\}$. In other words, we consider the process p that performed t_i and the variable x that is read by p in t_i. We search for pending write operations are issued by p and associated with elements in buffers and channel that are not updated to the memory. Now define $\mathsf{NewFences}_{\mathsf{mfence}}(\pi) := \left\{\underline{q}_k(p) \middle| \, \exists i, j : 1 \leq i \leq n, j \in \mathtt{OverRead}(\pi)(i), j \leq k < i\right\}$. In other words, it is necessary to insert a mfence fence at least one position between a pair (j, i) for each $i : 1 \leq i \leq n$ and each $j \in \mathtt{OverRead}(\pi)(i)$ in order to eliminate at least one of write-read overtaking.

Second, we show how to find the set of $\mathsf{NewFences}_{\mathsf{sfence}}(\pi)$ for π in a similar way. Define $n'_i := |m_i|$. We define a sequence of function $\gamma_0, \gamma_1, \ldots, \gamma_n$ where $\gamma_i(j)$ (with $1 \leq j \leq n'_i$) associates to each element in the channel m_i the position in π of the write transition that is correspond to the element. We define $\gamma_0, \gamma_1, \ldots, \gamma_n$ in a recursive way. (i) $\gamma_0(j)$ is undefined for all $1 \leq j \leq n'_i$. (ii) $\gamma_i(1)$ is undefined also. (iii) If t_{i+1} is not a *serialised operation* then define $\gamma_{i+1} := \gamma_i$. (iv) Otherwise, we define $\gamma_{i+1}(j) := \gamma_i(j)$ if $2 \leq j \leq n'_i$, and define $\gamma_{i+1}(n'_i + 1) := i + 1$. Next, we find the write transitions that have been overtaken by *write operations*. We define a function $\mathtt{OverWrite}$ such that if t_i (with $1 \leq i \leq n$) is a write transition then $\mathtt{OverWrite}(\pi)(i)$ gives the positions of write transitions in π that have been overtaken by t_i. Formally, if t_i is not a write then define $\mathtt{OverWrite}(\pi)(i) := \emptyset$. Otherwise, $t_i = (q, \mathsf{w}(x, v), q') \in \Delta_p$ for some $p \in P$, define $\mathtt{OverWrite}(\pi)(i) := \{\alpha_i(j) \| \mathtt{LastWrite}\,(c_i, p, x) < j \leq n_i, t_{\alpha_i(j)} \in \Delta_p, \exists 1 \leq k_1 < k_2 \leq n'_n : \gamma_n(k_1) = t_i \wedge \gamma_n(k_2) = t_{\alpha_i(j)}\}$. Now define $\mathsf{NewFences}_{\mathsf{sfence}}(\pi) := \left\{\underline{q}_k(p) \middle| \, \exists 1 \leq i \leq n, j \in \mathtt{OverWrite}(\pi)(i), j \leq k < i\right\}$.

Algorithm. We present our fence insertion algorithm (Algorithm 2). The algorithm takes a concurrent finite-state program $\mathcal{P}$, a finite set `Target`, and returns a minimal set of fences that is sufficient to make the program safe wrt. `Target`. If this set is empty then we conclude that the program cannot be corrected by placing fences. It means that the program is *not safe* (i.e. can reach to `Target`) even under SC semantics. The algorithm uses a set, namely $\mathcal{W}$, for sets of fences that have been *partially* constructed (but not yet large enough to make the program correct). During each iteration, a set F is picked and removed from $\mathcal{W}$. We use the HSB reachability analysis algorithm (Algorithm 1) to check whether the set F is sufficient to make the program correct. If *yes*, we return F as a possible set of minimal fences. If *no*, we compute the set of fences N such that inserting a member of N will eliminate one overtaking in the trace generated by Algorithm 1. We use $\mathcal{P} \oplus F$ to denote the program we get by inserting a set of fences F to $\mathcal{P}$, and use π for the trace. For each $\mathfrak{f} \in \mathsf{N}$ we add $F' = F \cup \{\mathfrak{f}\}$ back to $\mathcal{W}$ unless there is already a subset of F' in $\mathcal{W}$.

Algorithm 2: Fence Insertion.

```
input : A concurrent program $\mathcal{P}$, and a
        finite set Target of local state
        definitions.
output: A minimal set of fences if it
        exists, or an empty set.
1  $\mathcal{W} \leftarrow \{\emptyset\}$;
2  while true do
3      Pick and remove a set $F$ from $\mathcal{W}$;
4      if $Reachable(HSB)\ (\mathcal{P} \oplus F)$ (Target)
       then
5          $\mathsf{N} \leftarrow \mathsf{NewFences}(\pi)$;
6          if $\mathsf{N} = \emptyset$ then return $\emptyset$;
7          foreach $\mathfrak{f} \in \mathsf{N}$ do
8              $F' \leftarrow F \cup \{\mathfrak{f}\}$;
9              if $\exists F'' \in \mathcal{W} : F'' \subseteq F'$
               then discard $F'$;
10             else $\mathcal{W} \leftarrow \mathcal{W} \cup \{F'\}$;
11     else
12         return $F$;
```

Theorem 3. *For a concurrent finite-state program $\mathcal{P}$ and a finite set* `Target`*, Algorithm 2 terminates and returns a minimal set of fences wrt. $\mathcal{P}$ if the set exists, or an empty one otherwise.*

6 Experimental Results

Tool We have implemented our techniques from Section 3-Section 5 for reachability analysis and fence insertion of programs under PSO semantics to Memorax[1]. The current version of Memorax only applies for TSO semantics [2]. We compare our method with state-of-the-art tools: Remmex [20] (a tool based on state-space verification with acceleration for program analysis wrt. safety properties under TSO and PSO), and Musketeer [6] (a static analysis tool for correctness analysis wrt. robustness property under weak memory model). We compare based on two criteria: *number of fences* and *the running time*. We run the experiments using an Intel x86-32 Core2 2.4 Ghz machine and 4GB of RAM on 16 programs used as benchmarks in [2,20,6,19]. The results are given in Table 1. For each experiment, we report the number of processes (#P), the number of detected fences (#F) (including mfence and sfence if possible), the running time in seconds (#T). Musketeer does not make difference between mfence and sfence, so we put the total number of fences for it.

[1] https://github.com/margnus1/memorax

For all experiments, we set up the time out to 1800 seconds. If a tool runs out of time (resp. memory), we put "TO" (resp. "OM") in the #T column, and • in #F column. We use "m" for mfence and "s" for sfence. Bellow we summarise the main observations: (i) Memorax successfully finds the minimal fence sets in 15/16 experiments, and only fails in one test because of running out memory (CLHQLock). The minimal fence sets of Memorax and Remmex are the same. (ii) The running time of Memorax and Remmex are compatable (Memorax is better in 9 examples, and Remmex is better in 7 ones). (iii) Musketter is the fastest tool, but also fails in CLHQLock test as Memorax does. However, in most cases (13/15), Musketter returns redundant fences that are not optimal. Especially, AltBit, TickSLock, RevBarrier, and SpinLock are declared to be safe under PSO according to Memorax and Remmex, but still need fences using Musketter.

Program	#P	Memorax		Remmex		Musketeer	
		#F	#T	#	#T	#F	#T
SimDek	2	2 m,0 s	1.0	2 m,0 s	2.2	6	1.0
Dekker	2	4 m,0 s	2.2	4 m,0 s	4.8	10	1.0
LamBak	2	4 m,2 s	1253.7	4 m,2 s	9.3	8	1.0
Dijkstra	2	2 m,0 s	5.0	2 m,0 s	5.5	8	1.0
LamFast2	2	4 m,2 s	241.6	4 m,2 s	12.9	12	1.0
Peterson	2	2 m,2 s	4.1	2 m,2 s	7.6	6	1.0
Burns	2	2 m,0 s	1.0	2 m,0 s	4.2	6	1.0
IncSeq	2	0 m,0 s	1.0	0 m,0 s	104.3	0	1.0
Szymanski	2	3 m,0 s	3.3	3 m,0 s	5.8	10	1.0
AltBit	2	0 m,0 s	49.4	0 m,0 s	2.2	4	1.0
CLHQLock	2	•	OM	0 m,0 s	3.1	•	TO
TaskSched	2	0 m,0 s	153.2	0 m,0 s	3.0	0	1.0
Pgsql	2	2 m,1 s	5.4	2 m,1 s	22.82	4	1.0
TickSLock	2	0 m,0 s	24.5	0 m,0 s	5.03	2	1.0
RevBarrier	2	0 m,0 s	2.4	0 m,0 s	1.5	4	1.0
SpinLock	2	0 m,0 s	1.0	0 m,0 s	1.4	1	1.0

Table 1: Analyzed concurrent program.

7 Conclusion

We have presented a precise and sound automatic fence insertion method for concurrent finite-state programs under PSO memory model. We have introduced a new HSB semantics that is equivalent to PSO semantics in the sense of reachability problems, we use a backward analysis to solve the HSB reachability problem. In the case of an unsafe program under PSO but safe under SC, we propose a counter-example algorithm to find a minimal fence set to correct it. We prove the efficiency of our approach by running several benchmarks including challenging ones in existed methods.

References

1. P. A. Abdulla, M. F. Atig, Y.-F. Chen, C. Leonardsson, and A. Rezine. Automatic fence insertion in integer programs via predicate abstraction. In *SAS*, volume 7460 of *LNCS*, pages 164–180. Springer, 2012.
2. P. A. Abdulla, M. F. Atig, Y.-F. Chen, C. Leonardsson, and A. Rezine. Counter-example guided fence insertion under TSO. In *TACAS 2012*, volume 7214 of *LNCS*, pages 204–219. Springer Heidelberg, 2012.
3. P. A. Abdulla, M. F. Atig, Y.-F. Chen, C. Leonardsson, and A. Rezine. Memorax, a precise and sound tool for automatic fence insertion under TSO. In *TACAS*, volume 7795 of *LNCS*, pages 530–536. Springer, 2013.
4. P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *LICS*, pages 313–321, 1996.

5. S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996.
6. J. Alglave, D. Kroening, V. Nimal, and D. Poetzl. Don't sit on the fence. In A. Biere and R. Bloem, editors, *CAV 2014*, volume 8559 of *LNCS*, pages 508–524. Springer, Heidelberg, 2014.
7. J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In *ESOP*, volume 7792 of *LNCS*, pages 512–532. Springer, 2013.
8. J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *CAV*, volume 8044 of *LNCS*, pages 141–157. Springer, 2013.
9. M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In *POPL*, pages 7–18. ACM, 2010.
10. M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. What's decidable about weak memory models? In *ESOP*, volume 7211 of *LNCS*, pages 26–46. Springer, 2012.
11. M. F. Atig, A. Bouajjani, and G. Parlato. Getting rid of store-buffers in TSO analysis. In *CAV*, volume 6806 of *LNCS*, pages 99–115. Springer, 2011.
12. A. Bouajjani, E. Derevenetc, and R. Meyer. Checking and enforcing robustness against TSO. In *ESOP*, volume 7792 of *LNCS*, pages 533–553. Springer, 2013.
13. S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *CAV*, volume 5123 of *LNCS*, pages 107–120. Springer, 2008.
14. J. Burnim, K. Sen, and C. Stergiou. Testing concurrent programs on relaxed memory models. In *ISSTA*, pages 122–132. ACM, 2011.
15. A. M. Dan, Y. Meshman, M. T. Vechev, and E. Yahav. Effective abstractions for verification under relaxed memory models. In *VMCAI*, pages 449–466. Springer, 2015.
16. K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *ASPLOS'91*, pages 245–257, 1991.
17. G. Gopalakrishnan, Y. Yang, and H. Sivaraj. QB or not QB: An efficient execution verification tool for memory orderings. In *CAV*, volume 3114 of *LNCS*, pages 401–413. Springer, 2004.
18. G. Higman. Ordering by divisibility in abstract algebras. *Proc. London Math. Soc. (3)*, 2(7):326–336, 1952.
19. M. Kuperstein, M. T. Vechev, and E. Yahav. Partial-coherence abstractions for relaxed memory models. In *PLDI*, pages 187–198. ACM, 2011.
20. A. Linden and P. Wolper. A verification-based approach to memory fence insertion in pso memory systems. In *TACAS*, volume 7795 of *LNCS*, pages 339–353. Springer, 2013.
21. F. Liu, N. Nedev, N. Prisadnikov, M. T. Vechev, and E. Yahav. Dynamic synthesis for relaxed memory models. In *PLDI*, pages 429–440. ACM, 2012.
22. S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 391–407. Springer, 2009.
23. 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. *Commun. ACM*, 53(7):89–97, 2010.
24. SPARC International, Inc. *The SPARC Architecture Manual Version 9*, 1994.
25. Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *IPDPS*. IEEE Computer Society, 2004.