Automatic Fence Insertion in Integer Programs via Predicate Abstraction*

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

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

Abstract. We propose an automatic fence insertion and verification framework for concurrent programs running under relaxed memory. Unlike previous approaches to this problem, which allow only variables of finite domain, we target programs with (unbounded) integer variables. The problem is difficult because it has two different sources of infiniteness: unbounded store buffers and unbounded integer variables. Our framework consists of three main components: (1) a finite abstraction technique for the store buffers, (2) a finite abstraction technique for the integer variables, and (3) a counterexample guided abstraction refinement loop of the model obtained from the combination of the two abstraction techniques. We have implemented a prototype based on the framework and run it successfully on all standard benchmarks together with several challenging examples that are beyond the applicability of existing methods.

1 Introduction

Modern concurrent process architectures allow *relaxed* memory, in which certain memory operations may overtake each other. The use of weak memory models makes reasoning about the behaviors of concurrent programs much more difficult and error-prone compared to the classical *sequential consistency* (SC) memory model. In fact, several algorithms that are designed for the synchronization of concurrent processes, such as mutual exclusion and producer-consumer protocols, are not correct when run on weak memories [3]. 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 forbids certain reordering between instructions issued by the same process. For example, a fence may forbid an operation issued after the fence instruction to overtake an operation issued before it. Recently, several research efforts [9, 8, 14, 6, 15, 13, 18, 5, 4, 10, 11, 2] have targeted developing automatic verification and fence insertion algorithms of concurrent programs under relaxed memory. However, all these approaches target finite state programs. For the problem of analyzing algorithms/programs with mathematical integer variables (i.e., variables of an infinite data domain), these approaches can only approximate them by, e.g., restricting the upper and lower

* This research was in part funded by the Uppsala Programming for Multicore Architectures Research Center (UPMARC).

bounds of variables. The main challenge of the problem is that it contains two different dimensions of infiniteness. First, under relaxed memory, memory operations may be temporarily stored in a buffer before they become effective and the size of the buffer is *unbounded*. Second, the variables are ranging over an *infinite* data domain.

In this paper, we propose a framework (Fig. 1) that can automatically verify a concurrent system $\mathcal{S}$ (will be defined in Sec. 2) with integer variables under relaxed memory and insert fences as necessary to make it correct. The framework consists of three main components. The first component (Sec. 4) is responsible for finding a finite abstraction of the unbounded store buffers. In the paper, we choose to instantiate it with a technique introduced in [15]. Each store buffer in the system keeps only the first k operations and makes a finite over-approximation of the rest. For convenience, we call this technique *k-abstraction* in this paper. The second component (Sec. 5) (1) finds a finite abstraction of the data and then (2) combines it with the first abstraction to form a finite *combined abstraction* for both the buffer and data. For the data abstraction, in this paper we choose to instantiate it with *predicate abstraction*; a finite set of predicates over integer variables in the system is applied to partition the infinite data domain into finitely many parts. The combined abstraction gives us a finite state abstraction of the concurrent system $\mathcal{S}$. A standard reachability algorithm (Sec. 6) is then performed on the finite abstraction. For the case that a counterexample is returned, the third component analyzes it (Sec. 7) and depending on the result of the analysis it may refine the concurrent system by adding fences, refine the abstract model by increasing k or adding more predicates, or report that *ce* is an unpreventable counterexample trace, i.e., a bad behavior exists even in the SC model and cannot be removed by adding fences.

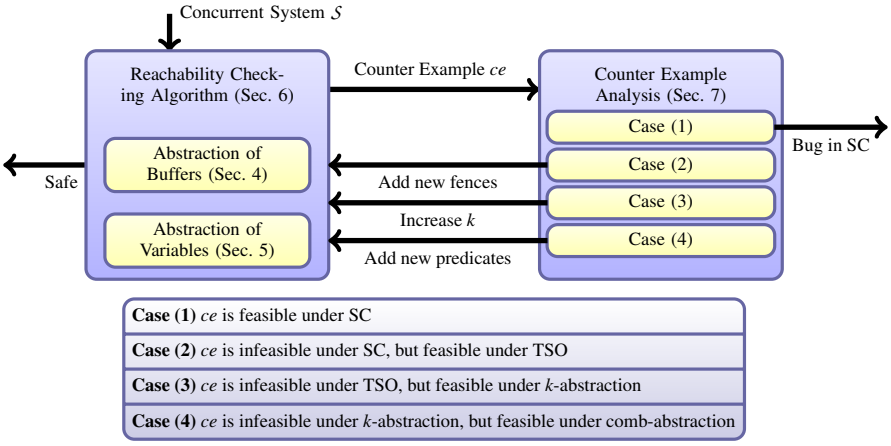

Fig. 1. Our fence insertion/verification framework

Because of the space limit and in order to simplify presentation, we demonstrate our technique under the total store order (TSO) memory model. However, our technique can

be generalized to other memory models such as the partial store order (PSO) memory model. In this paper, we use the usual formal model of TSO, developed in, e.g., [20, 23], and assume that it gives a faithful description of the actual hardware on which we run our programs. Conceptually, the TSO model adds a FIFO buffer between each process and the main memory (Fig. 2). The buffer is used to store the write operations performed by the process. Thus, a process executing a write operation inserts it into its store buffer and immediately continues executing subsequent operations. Memory updates are then performed by non-deterministically choosing a process and executing the oldest write operation in its buffer. A read operation by a process p on a variable x can overtake some write operations stored in its own buffer if all these operations concern variables that are different from x. Thus, if the buffer contains some write operations to x, then the read value must correspond to the value of the most recent write operation to x. Otherwise, the value is fetched from the memory. A fence means that the buffer of the process must be flushed before the program can continue beyond the fence. Notice that the store buffers of the processes are unbounded since there is *a priori* no limit on the number of write operations that can be issued by a process before a memory update occurs.

To our knowledge, our approach is the first automatic verification and fence insertion method for concurrent *integer* programs under relaxed memory. We implemented a prototype and run it successfully on all standard benchmarks together with challenging examples that are beyond the applicability of existing methods. For instance, we can verify Lamport's Bakery algorithm without assuming an upper bound on ticket numbers.

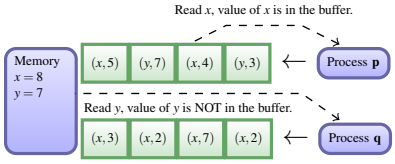

Fig. 2. TSO memory model.

2 Concurrent Systems

Our goal is to verify safety properties of
concurrent systems under relaxed memory. A *concurrent system* (P, A, X_S, X_L) consists of a set of processes P running in parallel with shared variables X_S and local variables X_L. These processes P are modeled by a set of finite automata $A = \{A_p \mid p \in P\}$. Each process p in P corresponds to an automaton A_p in A. Each local variable in X_L belongs to one process in P, i.e., we assume that two processes will not use the same local variable in X_L. The automaton A_p is a triple $(Q_p, q_p^{init}, \delta_p)$, where Q_p is a finite set of *program locations* (sometimes "locations" for short), q_p^{init} is the initial program location, and δ_p is a finite set of transitions. Each transition is a triple (l, op, l'), where l, l' are locations and op is an operation in one of the following forms: (1) read operation $read(x,v)$, (2) write operation $write(x,v)$, (3) fence operation $fence$, (4) atomic read write operation $arw(x,v,w)$, (5) assignment operation $v := e$, and (6) guard operation $e_1 \circ e_2$, for $\circ \in \{>, =, <\}$. In the above, x is a shared variable in X_S, v, w are local variables in X_L, and e, e_1, e_2 are quantifier-free Presburger formulae over X_L. We write

$l \xrightarrow{op}_p l'$ to denote that $(l, op, l') \in \delta_p$. We assume that $Q_p \cap Q_q = \emptyset$ for all $p, q \in P$ such that $p \neq q$ and use Q to denote the set of all locations in the concurrent program, i.e., $Q = \bigcup_{p \in P} Q_p$. In the next section, we formally define the semantics of concurrent systems under TSO and the verification problem we are interested in.

3 The TSO Transition System

We define the semantics of concurrent systems under TSO in this section. We begin with the definition of some terms and notations that will be used in this paper. In the rest of the paper, we fix a concurrent system $\mathcal{S} = (P, A, X_S, X_L)$.

3.1 Definitions and Notations

We write $\mathcal{N}$ for the set of natural numbers (the set of positive integers) and $\mathcal{Z}$ for the set of integers. Given a set S, we use $|S|$ to denote the cardinality of S. For each process $p \in P$ and an integer value $i \in \mathcal{N}$, we use the variable $b_{p,i}$ to denote the i-th operation of the store buffer of p. We assume that the smaller the value i is, the closer it is to the memory, i.e., the longer it stayed in the buffer. We use X_B to denote the set $\{b_{p,i} \mid p \in P \wedge i \in \mathcal{N}\}$ and call it the set of *buffer variables*. For a partial function f, we use the notation $f(x) = \perp$ to denote that f is undefined on x.

After these basic definitions, we will start to explain the semantics of concurrent systems under TSO. This is done by first defining system configurations and then the transition relation between these configurations w.r.t different operations.

3.2 Configurations

A *configuration* is a snapshot of a concurrent system, which captures values of shared and local variables, the current location of each process, and the content of the store buffers. Formally, we define a *configuration* as a tuple (M, L, pc, B_x, B_v), where $M : X_S \to \mathcal{Z}$ maps a shared variable to its value, $L : X_L \to \mathcal{Z}$ maps a local variable to its value, the function $pc : P \to Q$ maps a process p to its current location in Q_p, $B_x : X_B \to X_S$ maps a buffer variable to its corresponding shared variable, and $B_v : X_B \to \mathcal{Z}$ maps a buffer variable to its value. For example, if the i-th operation in the buffer of process p is $(x, 3)$ (update the value of x to 3), then $B_x(b_{p,i}) = x$ and $B_v(b_{p,i}) = 3$. Notice that here the functions B_x and B_v are partial. A configuration (M, L, pc, B_x, B_v) is said to be *initial* if $pc(p) = q_p^{init}$ for all $p \in P$, $M(x) = 0$, $L(v) = 0$, $B_x(b) = \perp$, $B_v(b) = \perp$ for all $x \in X_S$, $v \in X_L$ and $b \in X_B$[1].

3.3 Transition Relation

The transition relation between configurations is defined as follows. Assume that in the concurrent system $\mathcal{S}$, we have $l \xrightarrow{op}_p l'$. There exists a transition from the configuration

[1] Notice that for simplicity we assume the initial values of all shared and local variables are 0. This can be generalized by defining a new symbol $\top$ representing arbitrary integer values and assigning the initial values of all shared and local variables to $\top$.

(M, L, pc, B) to a next configuration (M', L', pc', B') if the following hold: (1) $pc(p) = l$, $pc'(p) = l'$, $\forall q \in P.q \neq p \rightarrow pc(q) = pc'(q)$ and (2) at least one of the transition rules in Fig.3 is satisfied. Below we explain the rules in Fig.3.

$$\frac{Contain(x)}{L'(v)=LastWrite(x)}\ \textbf{READ-B} \qquad \frac{\neg Contain(x)}{L'(v)=M(x)}\ \textbf{READ-M} \qquad \frac{|B_p| = i}{B'_x(b_{p,i+1}) = x \quad B'_v(b_{p,i+1}) = L(v)}\ \textbf{WRITE}$$

$$\frac{Empty}{}\ \textbf{FENCE} \qquad \frac{Empty \quad M(x) = L(v)}{M'(x) = L(w)}\ \textbf{ARW} \qquad \frac{}{L'(v) = e[L]}\ \textbf{ASSIGN} \qquad \frac{e_1[L] \circ e_2[L]}{}\ \textbf{GUARD}$$

$$\frac{|B_p| = i \quad B_x(b_{p,1}) = x_1 \ \ldots \ B_x(b_{p,i}) = x_i \quad B_v(b_{p,1}) = v_1 \ \ldots \ B_v(b_{p,i}) = v_i}{M'(x_1) = v_1 \quad B'_x(b_{p,1}) = x_2 \ \ldots \ B'_x(b_{p,i-1}) = x_i \quad B'_v(b_{p,1}) = v_2 \ \ldots \ B'_v(b_{p,i-1}) = v_i \quad B'_x(b_{p,i}) = B'_v(b_{p,i}) = \bot}\ \textbf{UPDATE}$$

Fig. 3. Transition Rules of a Transition System under TSO. The conditions above the horizontal line are the "pre-condition" that decide whether this transition can be triggered and those below the line are the "post-condition" that decide what the next configuration should be. For a more clear presentation, in the post-condition of the rules defined in this paper (including those in the other sections), we focus only on the component that has been changed. For the components that has not been changed, we assume implicitly that the primed version (the component in the next configuration) is equal to the non-primed version (the same component in the current configuration). For example, for all shared variables $x \in X_S$, if $M'(x)$ has not been assigned a value in the rule, we assume implicitly $M'(x) = M(x)$.

READ-B rule: When *op=read(x, v)*, if the buffer of p contains write operations to x, we read the value of the last write operation to x in p's buffer. We use $Contain(x)$ as a shorthand for $(\exists i \in \mathcal{N}.B_x(b_{p,i}) = x)$, or, informally, there exists some write operations to x in the buffer. We use $LastWrite(x)$ to denote the most recent value written to x in the buffer of p. Formally, $LastWrite(x) = X_v(b_{p,i})$, where $i = Max(\{j \in \mathcal{N} \mid B_x(b_{p,j}) = x\})$.
READ-M rule: When *op=read(x, v)*, if the buffer of p does not contain write operations to x, we read the value of x from the memory.
WRITE rule: When *op=write(x, v)*, we put the operation (x, v) to the end of the buffer. We use $|B_p|$ to denote the length of the buffer of p. Notice that this number equals the index of the most recent operation in p's buffer. Formally, $|B_p| = Max(\{j \in \mathcal{N} \mid B_x(b_{p,j}) \neq \bot\} \cup \{0\})$.
FENCE rule: When *op=fence*, the transition can be executed only when the buffer of p is empty. Here we use the predicate *Empty* as a shorthand for $B_x(b_{p,1}) = \bot$.
ARW rule: When *op=arw(x,v,w)*, the transition can be executed only when the buffer of p is empty and the value of x in the memory equals the value of v in p. When it is executed, the value of x in the memory is immediately changed to the value of w in p.
UPDATE rule: The write operations in the buffer can be at any time nondeterministically delivered to the memory. This is handled by implicitly adding self-loop transitions $l \xrightarrow{update} l$ from all the locations in Q. Notice that the transition $l \xrightarrow{update} l$ is internal, i.e., it never appears explicitly in the definition of the concurrent system. In this rule, the oldest operation in p's buffer (the one with index 1) will be used to update the memory while all the other operations in the buffer are shifted one step closer to the memory, i.e., their indices are reduced by 1.

ASSIGN rule: When $op = (v := e)$, where e is a Presburger expression over X_L, we update the value of v to the *evaluation* of e under the assignment L (denoted as $e[L]$).
GUARD rule: When $op = (e_1 \circ e_2)$, where e_1 and e_2 are Presburger expressions over X_L, the transition can be executed only when $(e_1[L] \circ e_2[L])$ holds, i.e., the evaluations of e_1 and e_2 under L is in the binary relation $\circ$. Here we let $\circ \in \{>, =, <\}$.

3.4 The Reachability Problem

The problem of verifying safety properties can be reduced to reachability problems. We use c^{init} to denote the initial configuration (defined in Section 3.2) and assume that a partial function $Bad : P \to Q$ is given. We use C^{Bad} to denote the set of bad configurations $\{(M, L, pc, B_x, B_v) \mid \forall p \in P.Bad(p) = \bot \vee pc(p) = Bad(p)\}$. Intuitively, taking a mutex problem of processes p_1, p_2, and p_3 as an example. If we want to describe the property that p_1 and p_2 cannot enter their critical sections at the same time, we define $Bad(p_1) = l_{cs1} \wedge Bad(p_2) = l_{cs2} \wedge Bad(p_3) = \bot$, where l_{cs1} and l_{cs2} are the locations of the critical sections. The reachability problem of a concurrent system under TSO asks if there exists some configuration in C^{Bad} reachable from the initial configuration c^{init} following the transition rules described in Fig. 3. We say that the concurrent system is "correct" iff all configurations in C^{Bad} are not reachable from c^{init}. Notice that we can extend this approach to allow finitely many bad functions $Bad_1 : P \to Q, \ldots, Bad_m : P \to Q$. In this case, the set of bad configurations becomes $\{(M, L, pc, B_x, B_v) \mid \bigvee_{1 \le i \le m} \forall p \in P.Bad_i(p) = \bot \vee pc(p) = Bad_i(p)\}$.

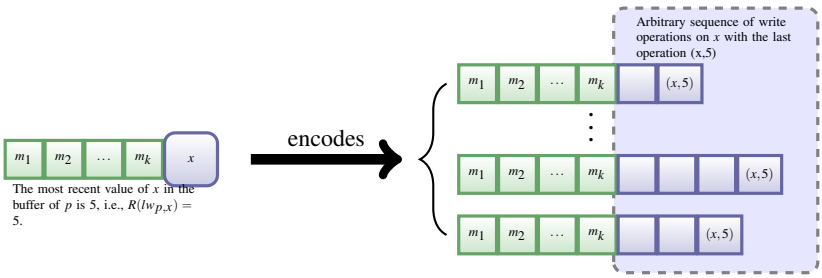

Fig. 4. A k-abstract buffer and the TSO buffer it encodes.

4 k-Abstraction

Notice that the store buffers under TSO may grow infinitely large. Therefore, a naive algorithm that exhaustively explores all reachable configurations would not work. One way to deal with the problem is to find a proper finite abstraction of the buffer. In this section, we introduce a finite abstraction technique of the buffer and the corresponding

abstract transition system [15]. We call this technique k-abstraction (for a given integer k). The basic idea is that, for a buffer with more than k write operations, we keep only the oldest k operations and assume that any operation can appear in the buffer after these k operations. To be more specific, for the operations with index larger than k, we only use (1) a set to record the variable part of those write operations together with (2) a function to record the most recent value of each shared variable in the buffer and abstract away other information. In Fig. 4, we illustrate the relation between a *k-abstract buffer* and the set of TSO buffers it encodes.

4.1 Definitions and Notations

In the sequel, we refer to the transition system induced from the concurrent system under TSO as "TSO system" and the system after k-abstraction as "k-abstract system". As a consequence, we call a configuration, a buffer, and a transition in the TSO system a "TSO configuration", a "TSO buffer", and a "TSO transition", respectively. We call a configuration, a buffer, and a transition in a k-abstract system a "k-abstract configuration", a "k-abstract buffer", and a "k-abstract transition", respectively. In a similar manner, to the case of buffer variables, for a process $p \in P$ and a shared variable $x \in X_S$, we use the variable $lw_{p,x}$ to refer to the value of the last write operation to x in the buffer of p. Let $X_{LW} = \{lw_{p,x} \mid p \in P \wedge x \in X_S\}$.

4.2 k-Abstract Configurations

Formally, a k-abstract configuration is a tuple $(M, L, pc, B_x, B_v, S, R)$, where M, L, pc, B_x,and B_v are defined in the same way as in a TSO configuration, $S : P \to 2^{X_S}$ records, for each process in P, the set of variables in the TSO buffer with index larger than k, and $R : X_{LW} \to \mathcal{Z}$ is a partial function that records the most recent value of each shared variable in the buffer.

In the rest of this section, we introduce the two functions γ_k (concretization) and α_k (abstraction) that relate k-abstract configurations and TSO configurations. Here we only give an informal description of these two functions and leave the formal definition to the appendix.

Given a k-abstract configuration c_k, the function $\gamma_k(c_k)$ maps the k-abstract configuration c_k to a set C_{TSO} of TSO configurations it encodes. A TSO configuration c_{TSO} in C_{TSO} has the same memory, valuation to local variables, and locations as c_s. The relation between the buffers of c_k and c_{TSO} can be best explained using Fig.4. If the buffer of c_k is the k-abstract buffer on the left of Fig.4, then the buffer of c_{TSO} is one of the TSO buffers on the right. Similarly, given a TSO configuration c_{TSO}, the function $\alpha_k(c_{TSO})$ maps it to a k-abstract configuration c_k with the same memory, valuation to local variables, and locations. The relation between their buffers can again be explained using Fig.4. The buffer of c_{TSO} corresponds to one of the buffers on the right of Fig. 4. After k-abstraction, we should obtain the k-abstract buffer on the left of Fig.4.

4.3 k-Abstract Transition Relation

Assume that we have $l \xrightarrow{op}_p l'$ in the concurrent system $\mathcal{S}$. There exists a k-abstract transition from a k-abstract configuration $(M, L, pc, B_x, B_v, S, R)$ to the other k-abstract

configuration $(M', L', pc', B'_x, B'_v, S', R')$ if the following holds (1) $pc(p) = l$, $pc'(p) = l'$, $\forall q \in P.q \neq p \rightarrow pc(q) = pc'(q)$ and (2) one of the k-abstract transition rules (Fig.5) holds.

$$\frac{|B_p| = k \vee S(p) \neq \emptyset}{R'(lw_{p,x}) = L(v) \quad S'(p) = S(p) \cup \{x\}}\ \textbf{WRITE-G} \qquad \frac{|B_p| = i < k \quad S(p) = \emptyset}{B'_x(b_{p,i+1}) = x \quad B'_v(b_{p,i+1}) = R'(lw_{p,x}) = L(v)}\ \textbf{WRITE-L}$$

$$\frac{\begin{array}{c}|B_p| = i \neq 0 \quad B_x(b_{p,1}) = x_1 \ldots B_x(b_{p,i}) = x_i \quad B_v(b_{p,1}) = v_1 \ldots B_v(b_{p,i}) = v_i \\ M'(x_1) = v_1 \quad B'_x(b_{p,1}) = x_2 \ldots B'_x(b_{p,i-1}) = x_i \quad B'_v(b_{p,1}) = v_2 \ldots B'_v(b_{p,i-1}) = v_i \quad B'_x(b_{p,i}) = B'_v(b_{p,i}) = \perp\end{array}}{}\ \textbf{UPDATE-NE}$$

$$\frac{x \in S(p) \quad |B_p| = 0}{M'(x) = M'(x)}\ \textbf{UPDATE-AM} \qquad \frac{x \in S(p) \quad |B_p| = 0}{M'(x) = R(lw_{p,x}) \quad S'(p) = S(p) \setminus \{x\}}\ \textbf{UPDATE-AS}$$

Fig. 5. k-Abstract Transition Rules. We list only rules that are different from the rules in Fig.3.

READ-B, READ-M, FENCE, ARW, ASSIGN, GUARD rules: For $op = read(x, v)$, the rule of k-abstract transitions is almost the same as the one of TSO transitions. The only exception is that the definition of the predicate $Contain(x)$ should be changed to $(x \in S) \vee (\exists i \in \mathcal{N}.B_x(b_{p,i}) = x)$ and $LastWrite(x) = R(lw_{p,x})$. The case of $op = fence$ or $op = arw(x, v, w)$ can be handled in a similar way. We only need to change the definition of $Empty$ to $(B_x(b_{p,1}) = \perp \wedge S(p) = \emptyset)$. The case of local operations $op = (v := e)$ and $op = (e_1 \circ e_2)$ can be handled by exactly the same rule as in a TSO transition.

WRITE rules: When $op = write(x, v)$, we need to consider the cases where the size of the abstract buffer is less than k (**WRITE-L**) and equal to or greater than k (**WRITE-G**). When the size is smaller than k, it behaves the same as in the TSO system. For the case that the size is equals to or greater than k, we (1) modify the record of the last write operation of x and (2) add x to the set $S(p)$.

UPDATE rules: When $op = update$, different cases have to be considered. When the k-bounded buffer is not empty, i.e., $B(b_{p,1}) \neq \perp$ (**UPDATE-NE**), the oldest operation in the buffer (the one with index 1) is sent to the memory and all the other operations in the buffer are shifted one step closer to the memory. For the case that the k-bounded buffer is empty, i.e., $B(b_{p,1}) = \perp$, but the k-abstract buffer is already an over-approximation, i.e, $S(p) \neq \emptyset$, there are two possible sub-cases. One is when the corresponding TSO buffer has more than one operation on x (**UPDATE-AM**) and one is when the TSO buffer has only one operation on x left (**UPDATE-AS**). For the former case, the update operation may change the memory value of any variables in $S(p)$ to any value in $\mathcal{Z}$. Hence we do not need any constraint on $M'(x)$ and put a tautology $M'(x) = M'(x)$ to show that the value of x in the memory has been changed. For the latter case, since only one write operation to x is left in the buffer, the most recent and the oldest write operation to x in the buffer coincide. Therefore, the update operation changes the memory value of the variable x in $S(p)$ to $R(lw_{p,x})$.

5 Combined Abstraction

The elements in a k-abstract state $(M, L, pc, B_x, B_v, S, R)$ can be categorized into two parts. The *data components* include M, L, B_v, R, which are assignments to variables ranging over $\mathcal{Z}$, and the rest belongs to the *control components*. Since the numbers of shared variables $|X_S|$, processes $|P|$, locations of each process $|Q|$, and the lengths of the k-bounded buffers are finite, there exists only a finite number of different control components. However, this is not the case for data components. Since the data domain $\mathcal{Z}$ is an infinite set, there exists an infinite number of different data components. It follows that the number of possible configurations can be infinite. In this case, the reachability problem becomes non-trivial. One possible solution is to also apply abstraction techniques on data in order to get a finite abstraction of reachable configurations. Then the reachability problem can be solved by simple depth first or breadth first search algorithms. In this section, we will demonstrate how to use *predicate abstraction* to form a finite abstraction of the data components and how k-abstraction and predicate abstraction are combined.

5.1 Definitions and Notations

We have $X = X_S \cup X_L \cup X_B \cup X_{LW}$, the set of all integer variables in the k-abstract system (recall that X_S,X_L,X_B,X_{LW}, is the set of shared, local, buffer, and last-write variables, respectively). Given a formula e, we define the substitution operation $e[x/x']$ as the formula obtained by replacing all free occurrences of x in e with x'. Given a set of variables $X = \{x_1, \ldots, x_n\}$, we use X' to denote the primed version of X, i.e., $X' = \{x' \mid x \in X\}$. If X and X' are two disjoint sets, we write $e[x/x']_{x \in X}$ as a shorthand for $e[x_1/x'_1][x_2/x'_2]\ldots[x_n/x'_n]$, i.e., replacing all free occurrences of elements $x \in X$ appearing in e with their new variants x'. In the paper, we refer to a transition system on the combined abstraction domain as a "comb-abstract system". As a consequence, we call a configuration, a buffer, and a transition in a comb-abstract system a "comb-abstract configuration", a "comb-abstract buffer", and a "comb-abstract transition", respectively.

5.2 The Idea

The idea of *predicate abstraction* is to use predicates over variables in X to partition the data components of k-abstract configurations into finitely many parts. Each partition may encode an infinite number of different data components. An example can be found in Fig.6. In the figure on the left, we abstract the data components by a predicate $f = (x > y \wedge b_{p,4} = t \wedge lw_{p,y} = t)$ while we store the control component exactly. We call the result a *comb-abstract configuration*. In the example, the comb-abstract configuration encodes k-abstract configurations with the same control components and with data components satisfying the constraint defined in the predicate f. Taking the k-abstract configuration $(M, L, pc, B_x, B_v, S, R)$ [4] on the top-right of Fig 6 as an example, it has the

[4] Recall that $S : P \to 2^{X_S}$ records, for each process in P, the set of variables in the TSO buffer with index larger than k, and $R : X_{LW} \to \mathcal{Z}$ is a partial function that records the most recent value of each shared variable in the buffer.

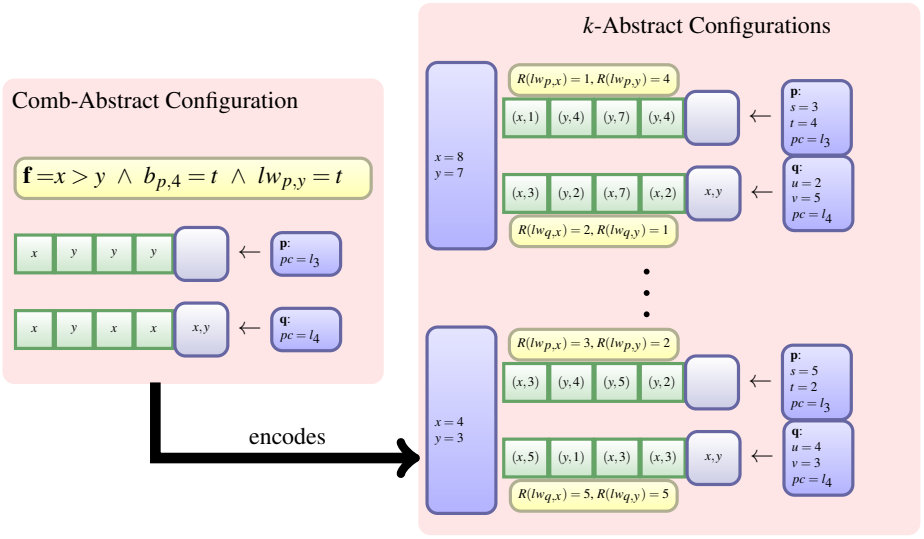

Fig. 6. A comb-abstract configuration and the k-abstract configurations it encodes. Here $X_S = \{x,y\}$ and $X_L = \{s,t,u,v\}$. All the configurations in the figure have the same control components pc, B_x, and S, where $pc(p) = l_3 \wedge pc(q) = l_4$, $B_x(b_{p,1}) = B_x(b_{q,1}) = B_x(b_{q,3}) = B_x(b_{q,4}) = x \wedge B_x(b_{p,2}) = B_x(b_{p,3}) = B_x(b_{p,4}) = B_x(b_{q,2}) = y$, and $S(p) = \emptyset \wedge S(q) = \{x,y\}$.

same control components as the comb-abstract configuration on the left. By substituting x and y in f with $M(x)$ and $M(y)$, t with $L(t)$, $b_{p,4}$ with $B_v(b_{p,4})$, $lw_{p,y}$ with $R(lw_{p,y})$, we obtain the formula $8 > 7 \wedge 4 = 4 \wedge 4 = 4$, which evaluates to $true$. Hence it is a k-abstract configuration encoded by the comb-abstract configuration.

5.3 Comb-Abstract Configurations

Formally, a comb-abstract configuration is a tuple (f, pc, B_x, S), where f is a formula over X that encodes data components, and the control components pc, B_x, S are defined in a similar manner as in a k-abstract configuration. Given a k-abstract configuration $c_k = (M, L, pc, B_x, B_v, S, R)$ and a formula f over X, we define the *evaluation* of f in c_k, denoted as $f[c_k]$, as the value obtained by substituting all free occurrences of $x \in X_S$ in f with $M(x)$, $v \in X_L$ in f with $L(v)$, $b \in X_B$ in f with $B_v(b)$, and $lw \in X_{LW}$ in f with $R(lw)$. Given a comb-abstract configuration $c_c = (f, pc, B_x, S)$, we define the concretization function $\gamma_c(c_c) = \{c_k = (M, L, pc, B_x, B_v, S, R) \mid f[c_k]\}$. Given a set of comb-abstract configurations C_b, we define $\gamma_c(C_b) = \bigcup_{c_b \in C_b} \gamma_c(c_b)$. Given a set of k-abstract configurations C_k, we use $\alpha_c(C_k)$ to denote the set of comb-abstract configurations that encodes exactly C_k, i.e., $C_k = \gamma_c(\alpha_c(C_k))$.

5.4 Predicate Abstraction

Let f be a formula over X and $\mathcal{P}$ a set of predicates over X. Each predicate in $\mathcal{P}$ partitions the valuation of variables in X into two parts. For each predicate $\pi \in \mathcal{P}$ such that $f \rightarrow \pi$ is valid, or equivalently, $f \wedge \neg\pi$ is unsatisfiable, π characterizes a superset of data components of those characterized by f. The predicate abstraction function $\alpha_{pa}(f, \mathcal{P})$ returns a conjunction of all predicates $\pi \in \mathcal{P}$ such that $f \rightarrow \pi$ is valid.

5.5 Comb-Abstract Transition Relation (w.r.t a Set of Predicates $\mathcal{P}$)

Assume that we have $l \xrightarrow{op}_p l'$ in the concurrent system $\mathcal{S}$. There exists a comb-abstract transition w.r.t. $\mathcal{P}$ from the comb-abstract configuration (f, pc, B_x, S) to a next configuration $(\alpha_{pa}(f', \mathcal{P}), pc', B'_x, S')$ if the following hold (notice that we always apply predicate abstraction to the formula f' of the next configuration): (1) $pc(p) = l$, $pc'(p) = l'$, $\forall q \in P.q \neq p \rightarrow pc(q) = pc'(q)$, (2) f' is satisfiable, and (3) at least one of the comb-abstract transition rules in Fig.7 is satisfied.

$$\frac{\neg Contain(x)}{f' = (\exists X.f \wedge v' = x \wedge Equ(X \setminus \{v\}))[x'/x]_{x' \in X'}}\ \textbf{READ-M} \qquad \frac{Contain(x)}{f' = (\exists X.f \wedge v' = lw_{p,x} \wedge Equ(X \setminus \{v\}))[x'/x]_{x' \in X'}}\ \textbf{READ-B}$$

$$\frac{|B_p| = k \vee S(p) \neq \emptyset}{f' = (\exists X.f \wedge lw'_{p,x} = v \wedge Equ(X \setminus \{lw'_{p,x}\}))[x'/x]_{x' \in X'} \quad S'(p) = S(p) \cup \{x\}}\ \textbf{WRITE-G}$$

$$\frac{|B_p| = i < k \quad S(p) = \emptyset}{f' = (\exists X.f \wedge b'_{p,i} = lw'_{p,x} = v \wedge Equ(X \setminus \{b_{p,i}, lw_{p,x}\}))[x'/x]_{x' \in X'} \quad B'_x(b_{p,i}) = x}\ \textbf{WRITE-L} \qquad \frac{Empty}{}\ \textbf{FENCE}$$

$$\frac{Empty}{f' = (\exists X.f \wedge x = v \wedge x' = w \wedge Equ(X \setminus \{x\}))[x'/x]_{x' \in X'}}\ \textbf{ARW}$$

$$\frac{\begin{array}{c}|B_p| = i \neq 0 \quad B_x(b_{p,1}) = x_1 \ldots B_x(b_{p,i}) = x_i \\ B'_x(b_{p,1}) = x_2 \ldots B'_x(b_{p,i-1}) = x_i \quad B'_x(b_{p,i}) = \bot \\ f' = (\exists X.f \wedge x'_1 = b_{p,1} \wedge \bigwedge_{1 \leq k \leq i-1} b'_{p,k} = b_{p,k+1} \wedge Equ(X \setminus \{x_1, b_{p,1}, \ldots, b_{p,i-1}\}))[x'/x]_{x' \in X'}\end{array}}{}\ \textbf{UPDATE-NE}$$

$$\frac{x \in S(p) \quad |B_p| = 0}{f' = (\exists X.f \wedge Equ(X \setminus \{x\}))[x'/x]_{x' \in X'}}\ \textbf{UPDATE-AM}$$

$$\frac{x \in S(p) \quad |B_p| = 0}{f' = (\exists X.f \wedge x' = lw_{p,x} \wedge Equ(X \setminus \{x\}))[x'/x]_{x' \in X'} \quad S'(p) = S(p) \setminus \{x\}}\ \textbf{UPDATE-AS}$$

$$\frac{}{f' = (\exists X.f \wedge v' = e \wedge Equ(X \setminus \{v\}))[x'/x]_{x' \in X'}}\ \textbf{ASSIGN} \qquad \frac{f \wedge (e_1 \circ e_2)\}}{}\ \textbf{GUARD}$$

Fig. 7. Comb-Abstract Transition Rules. We use the predicate $Equ(V)$ to denote $\bigwedge_{v \in V} v' = v$, i.e., no change made to variables in V in this transition. We assume all bounded variables are renamed to fresh variables that are not in $X \cup X'$ so the substitution will not assign the names of bounded variables to some free variable.

6 The Reachability Checking Algorithm

Alg.1 solves they reachability problem of a comb-abstract system derived from a given concurrent system. The inputs of the algorithm include a value k, a set of predicates $\mathcal{P}$, a concurrent system $\mathcal{S} = (P, A, X_S, X_L)$, and a partial function $Bad : P \rightarrow Q$. We

Algorithm 1: Reachability Algorithm

Input : $\mathcal{S} = (P, A, X_S, X_L)$, an integer k, a set of predicates $\mathcal{P}$, a partial function $Bad : P \to Q$
Output: Either the program is safe or a counterexample ce

1 $c^{init} = (true, pc, B_x, S)$, where $\forall p \in P.(pc(p) = q_p^{init} \wedge S(p) = \emptyset) \wedge \forall b \in X_B.B_x(b) = \bot$;
2 $Next$:=$\{(c^{init}, \varepsilon)\}$, $Visited$:=$\emptyset$;
3 **while** $Next \neq \emptyset$ **do**
4 Pick and remove $((pd, pc, B_x, S), ce)$ from $Next$;
5 **if** $\forall p \in P.Bad(p) \neq \bot \to pc(p) = Bad(p)$ **then return** ce is a counterexample;
6 **if** $\exists (f, pc, B_x, S) \in Visited$ **then** replace it with $(f \vee pd, pc, B_x, S)$ **else** add (pd, pc, B_x, S) to $Visited$;
7 **foreach** $l \xrightarrow{op}_p l'$ *such that* $pc(p) = l$ **do**
8 **foreach** *comb-abstract transition rule* r **do**
9 compute the next configuration (pd', pc', B_x', S') of (pd, pc, B_x, S) w.r.t $l \xrightarrow{op}_p l'$, r, and $\mathcal{P}$;
10 **if** $\neg(\exists (f, pc', B_x', S') \in Visited$ *s.t.* $pd' \to f)$ **then**
11 add $((pd', pc', B_x', S'), ce \cdot (l \xrightarrow{op}_p l', r))$ to $Next$;
12 **return** The program is safe;

first generate the initial comb-abstract configuration $c^{init} = (true, pc, B_x, S)$, where $\forall p \in P.(pc(p) = q_p^{init} \wedge S(p) = \emptyset) \wedge \forall b \in X_B.B_x(b) = \bot$.

For the reachability algorithm, we maintain two sets, *Next* and *Visited* (Line 2). *Next* contains pairs of a comb-abstract configuration c and a path that leads to c. *Visited* contains comb-abstract configurations that have been visited. Notice that *Visited* stores comb-abstract configurations in an efficient way; if both the comb-abstract configurations (f_1, pc, B_x, S) and (f_2, pc, B_x, S) should be put into *Visited*, we put $(f_1 \vee f_2, pc, B_x, S)$ instead. When *Next* is not empty (Line 3), a pair $((pd, pc, B_x, S), ce)$ is removed from *Next* and the algorithm tests if (pd, pc, B_x, S) encodes some bad TSO configurations (Line 5). For the case that it does, the algorithm stops and returns ce as a counterexample. Otherwise (pd, pc, B_x, S) is merged into *Visited* (Line 6). Then the reachability algorithm explores the next configurations of (pd, pc, B_x, S) w.r.t the transitions in $\mathcal{S}$ and the comb-abstract transition rules (Lines 7-11). Once *Next* becomes empty, the algorithm reports that the program is safe. Notice that in the counterexample ce, we record not only the sequence of transitions of $\mathcal{S}$ but also the sequence of transition rules that have been applied. We need this in order to remove non-determinism in the comb-abstract system and thus simplify the counterexample analysis. To be more specific, assume that $l \xrightarrow{op}_p l'$ and a comb-abstract configuration c is given, it is possible that there exists more than one transition rules that can be applied and thus the same transition $l \xrightarrow{op}_p l'$ may lead to two different comb-abstract configurations. For example, assume that $op = update$ and the length of the TSO buffer is larger than k. It could happen that both of the rules **UPDATE-AM** and **UPDATE-AS** can be applied. Then the current comb-abstract configuration c may have two different next comb-abstract configurations w.r.t the same transition $l \xrightarrow{op}_p l'$.

7 Counter Example Guided Abstraction Refinement

The counterexample detected by the reachability checking algorithm is a sequence of pairs in the form of (δ, r), where δ is a transition in $\mathcal{S}$ and r is a comb-abstract transition rule. Let $ce = (l_1 \xrightarrow{op_1}_{p_1} l_1', r_1)(l_2 \xrightarrow{op_2}_{p_2} l_2', r_2) \ldots (l_n \xrightarrow{op_n}_{p_n} l_n', r_n)$ be the counterexample

returned from the reachability module. We next analyze *ce* and decide how to respond to it. Four possible responses are described in Fig.1.

Case (1): We will not formally define the transition system induced from the concurrent system under sequential consistency (SC) model for lack of space. Informally, under the SC model, all operations will be immediately sent to the memory without buffering. We simulate *ce* under SC and if *ce* is feasible under SC, *ce* is not a bug caused by the relaxation of the memory model. In this case, it cannot be fixed by just adding fences. The algorithm reports that *ce* is a bug of the concurrent system under the SC model.

Case (2): We can check if the counterexample *ce* is feasible under TSO by simulating it on the TSO system following the rules defined in Fig. 3. For the case that *ce* is infeasible under SC, but feasible under TSO, we can find a set of fences that can help to remove the spurious counterexample *ce* by the following steps. First we add fences immediately after all write operations in *ce*. We then repeatedly remove these newly added fences while keeping it infeasible under the TSO system. We do this until we reach a point where removing any fences would make *ce* feasible under TSO. In such case, the subsequently remaining such fences are those that need to be added. A more efficient algorithm of extracting fences from error traces can be found in [2].

Case (3): When *ce* is infeasible under TSO, but feasible under k-abstraction, we keep increasing the value of k until we reach a value i such that *ce* is feasible under $(i\text{-}1)$-abstraction, but infeasible under i-abstraction. In such case, we know that we need to increase the value of k to i in order to remove this spurious counterexample. Such a value i always exists, because the length of the sequence *ce* is finite, which means that it contains a finite number of write operations, say n operations, and thus the size of the buffer will not exceed n. When we set k to n, then in fact the behavior of *ce* will be the same under TSO and under k-abstraction. It follows that it is infeasible under k-abstraction when k equals n.

Case (4): When *ce* is infeasible under k-abstraction, but is feasible in the comb-abstract system, it must be the case that predicate abstraction made a too coarse over-approximation of the data components and has to be refined. An example can be found in Fig. 8, where g_0 (respectively, f_0) characterizes the data components of the initial k-abstract configuration (respectively, comb-abstract configuration) and g_i (respectively, f_i) characterizes the data components of the k-abstract configuration (respectively, comb-abstract configuration) after i steps of *ce* are executed. The rule r_3 has a precondition on data components such that g_2 cannot meet this condition, but f_2 can (note that this can happen only when r_3 is a **GUARD** rule or an **ARW** rule). This situation arises because the predicate abstraction in the first 2 steps of *ce* made a too coarse over-approximation. That is, some data components encoded in $f_2 \wedge \neg g_2$ that satisfy the pre-condition of transition rule r_3 are produced from the predication abstraction. In order to fix the problem, we have to find some proper predicates to refine f_0, f_1, and f_2 so the *ce* cannot be executed further after 2 steps in the comb-abstract system. Hence we

have to generate some more predicates to refine the comb-abstract system. This can be done using the classical predicate extraction technique based on *Craig interpolation* [7].

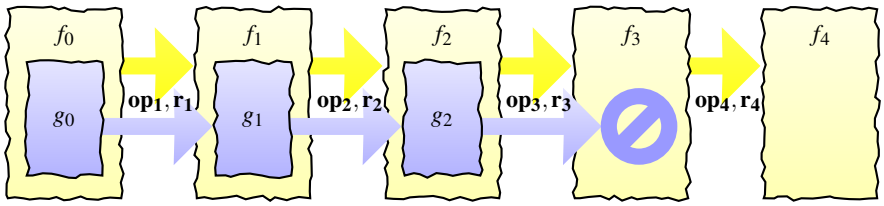

Fig. 8. Data components produced by *ce*.

8 Discussion

How to generalize the proposed technique? The proposed technique can be generalized to memory models such as the partial store order memory model or the power memory model. Such models use infinite buffers and one can define finite abstractions by applying the k-abstraction technique [15]. Predicate abstraction and counterexample analysis can be done in the same way as we described in this paper. The Presburger expressions used in this paper can also be extended to any theory for which satisfiability and interpolation are efficiently computable. Notice that although the formula f' in the comb-abstract transtion rules has existential quantifiers, we do not need to assume that quantifier elimination is efficiently computable for the given theory. This is because in predicate abstraction, for a given predicate π, instead of checking whether $f' \rightarrow \pi$ is valid, we check if $f' \wedge \neg\pi$ is unsatisfiable. For satisfiability checking, we can ignore the outermost existential quantifiers in f'.

Further optimizations. Assume that two local variables v, u of process p and a predicate $v < u$ describing their relation are given. When the size of the buffer of p is k and p executes the operation $write(x, v)$, the value of the buffer variable $b_{p,k+1}$ will be assigned to the value of v. Then the relation $v < u$ should propagate to the buffer variable and hence we should also have $b_{p,k+1} < u$. However, in order to generate this predicate, it requires another counterexample guided abstraction refinement iteration. It would require even more loop iterations for the relation $v < u$ to propagate to the variable x and generate the relation $x < u$. Notice that for such situations, the "shapes" of the predicates remain the same while propagating in the buffer. Based on this observation, we propose an idea called "predicate template". In this example, instead of only keeping $v < u$ in the set $\mathcal{P}$ of predicates, we keep a predicate template $\Box < \Box$. The formulae returned by the predicate abstraction function $\alpha_{pa}(f, \mathcal{P})$ are then allowed to contain predicates $x_0 < x_1$ for any $x_0, x_1 \in X$ s.t. $f \rightarrow x_0 < x_1$ is valid. We call predicates in this form *parameterized predicates*.

Modules in the Framework Our framework is in fact flexible. The k-abstraction can be replaced with any abstraction technique that abstracts the buffers to finite sequences. E.g., instead of keeping the oldest k operations in the buffer, one can also choose to keep the newest k operations and abstract away others. For the integer variable, instead of applying predicate abstraction techniques, we also have other choices. In fact, a k-abstract system essentially can be encoded as a sequential program with integer variables running under the SC model. Then one can choose to verify it using model checkers for sequential programs such as BLAST or CBMC.

9 Experimental Results

We have implemented the method described in this paper in C++ geared with parameterized predicates. Instead of keeping the oldest k operations in the buffer, we choose to keep the newest k operations and abstract away older operations. In the counter-example guided refinement loop, for Case 2 (fence placement) we use the more efficient algorithm described in [2].

	LOC	Time	Fences/proc	# Predicates
1. Burns [19]	9	0.02 s	1	1
2. Simple Dekker [24]	10	0.04 s	1	1
3. Full Dekker [12]	22	0.06 s	1	1
4. Dijkstra [19]	22	0.35 s	1	4
5. Lamport Bakery [16]	20	154 s	2	17
6. Lamport Fast [17]	32	2 s	2	4
7. Peterson [21]	12	2 s	1	6
8. Linux Ticket Lock[2]	16	2 s	0	2

Table 1. Experimental results

We applied it to several classical examples. Among these examples, the Lamport Bakery and Linux Ticket Lock involves integer variables whose values can grow unboundedly. To our knowledge, these examples cannot be handled by any existing algorithm. The experiments were run on a 2.27 GHz laptop with 4 GB of memory. The MathSat4 [1] solver is used as the procedure for deciding satisfiability and computing interpolants. All of the examples involve two processes. The results are given in Table 1. For each protocol we give the total number of instructions in the program, the total time to infer fence positions, the number of necessary fences per process, and the greatest number of parameterized predicates used in any refinement step.

References

1. *MATHSat4*. http://mathsat4.disi.unitn.it/.
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.
3. S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12), 1996.
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.

[2] The "Linux Ticket Lock" protocol was taken from the Linux kernel. Its correctness on x86 was the topic of a lively debate among the developers on the Linux Kernel Mailing List in 1999. (See the mail thread starting with `https://lkml.org/lkml/1999/11/20/76`.)

6. M. F. Atig, A. Bouajjani, and G. Parlato. Getting rid of store-buffers in TSO analysis. In *CAV*, 2011.
7. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast: Applications to software engineering. *STTT*, 2007.
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. 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. Sound and complete monitoring of sequential consistency in relaxed memory models. In *TACAS*, 2011.
12. E. W. Dijkstra. *Cooperating sequential processes*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.
13. T. Huynh and A. Roychoudhury. A memory model sensitive checker for C#. In *FM*, 2006.
14. M. Kuperstein, M. Vechev, and E. Yahav. Automatic inference of memory fences. In *FM-CAD*, 2011.
15. M. Kuperstein, M. Vechev, and E. Yahav. Partial-coherence abstractions for relaxed memory models. In *PLDI*, 2011.
16. L. Lamport. A new solution of dijkstra's concurrent programming problem. *CACM*, 17, August 1974.
17. L. Lamport. A fast mutual exclusion algorithm, 1986.
18. A. Linden and P. Wolper. A verification-based approach to memory fence insertion in relaxed memory systems. In *SPIN*, 2011.
19. N. Lynch and B. Patt-Shamir. DISTRIBUTED ALGORITHMS , Lecture Notes for 6.852 FALL 1992. Technical report, MIT, Cambridge, MA, USA, 1993.
20. S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-tso. In *TPHOL*, 2009.
21. G. L. Peterson. Myths About the Mutual Exclusion Problem. *IPL*, 12(3), 1981.
22. P. Ruemmer. *Princess*. http://www.philipp.ruemmer.org/princess.shtml.
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. *CACM*, 53, 2010.
24. D. Weaver and T. Germond, editors. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall, 1994.

A Abstraction and Concretization Functions for k-abstraction

Given a k-abstract configuration $c_k = (M, L, pc, B_x, B_v, S, R)$. Let $|B_p| = Max(\{i \in \mathcal{N} \mid B_x(b_{p,i}) \neq \bot\} \cup \{0\})$ denote the length of the buffer of process p, as encoded by B_x and B_v. We define $LastWrite_p(x, B'_x, B'_v) = B'_v(b_{p,i})$, where $i = Max(\{j \in \mathcal{N} \mid B'_x(b_{p,j}) = x\})$ and the last write constraint $LW(p, c_k, B'_x, B'_v) = \forall x \in S(p).LastWrite_p(x, B'_x, B'_v) = R(lw_{p,x})$. Let the buffer constraint $BC(p, m, c_k, B'_x, B'_v)$ equal the following

$$\forall 0 < i \leq |B_p|.(B'_x(b_{p,i}) = B_x(b_{p,i}) \wedge B'_v(b_{p,i}) = B_v(b_{p,i}))$$
$$\wedge$$
$$S(p) \neq \emptyset \rightarrow \left(\begin{array}{c} \forall x \in S(p).\exists |B_p| < i < m.B'_x(b_{p,i}) = x \\ \wedge \\ \forall |B_p| < i < m.(B'_x(b_{p,i}) \in S(p) \wedge B'_v(b_{p,i}) \neq \bot) \\ \wedge \\ \forall m \leq i.(B'_x(b_{p,i}) = B'_v(b_{p,i}) = \bot) \end{array} \right)$$
$$\wedge$$
$$S(p) = \emptyset \rightarrow (\forall |B_p| < i.(B'_x(b_{p,i}) = B'_v(b_{p,i}) = \bot))$$

We use $\gamma_k(c_k)$ to denote the set of TSO configurations encoded in c_k, which equals the set $\{(M, L, pc, B'_x, B'_v) \mid \forall p \in P.((\exists m \in \mathcal{N}.BC(p, m, c_k, B'_x, B'_v)) \wedge LW(p, c_k, B'_x, B'_v))\}$

On the other hand, given a TSO configuration $c_{TSO} = (M, L, pc, B_x, B_v)$, we define $\alpha_k(c_{TSO}) = (M, L, pc, B'_x, B'_v, S, R)$, where (1) $\forall 0 < i \leq k, p \in P.(B'_x(b_{p,i}) = B_x(b_{p,i}) \wedge B'_v(b_{p,i}) = B_v(b_{p,i}))$, (2) $\forall k < i, p \in P.((B_x(b_{p,i}) \neq \bot \rightarrow B_x(b_{p,i}) \in S(p)) \wedge (B'_x(b_{p,i}) = B'_v(b_{p,i}) = \bot))$, and (3) $\forall p \in P, x \in X_S.R(lw_{p,x}) = LastWrite_p(x, B_x, B_v)$.