Getting Rid of Store-Buffers in TSO Analysis*

Mohamed Faouzi Atig[1], Ahmed Bouajjani[2], and Gennaro Parlato[2]

[1] Uppsala University, Sweden
mohamed_faouzi.atig@it.uu.se
[2] LIAFA, CNRS and University Paris Diderot, France
{abou,gennaro}@liafa.jussieu.fr

Abstract. We propose an approach for reducing the TSO reachability analysis of concurrent programs to their SC reachability analysis, under some conditions on the explored behaviors. First, we propose a *linear* code-to-code translation that takes as input a concurrent program P and produces a concurrent program P' such that, running P' under SC yields the same set of reachable (shared) states as running P under TSO with at most k context-switches for each thread, for a fixed k. Basically, we show that it is possible to use only $O(k)$ additional copies of the shared variables of P as local variables to simulate the store buffers, even if they are unbounded. Furthermore, we show that our translation can be extended so that an *unbounded* number of context-switches is possible, under the condition that each write operation sent to the store buffer stays there for at most k context-switches of the thread. Experimental results show that bugs due to TSO can be detected with small bounds, using off-the-shelf SC analysis tools.

1 Introduction

The classical memory model for concurrent programs with shared memory is the sequential consistency (SC) model, where the behaviors of the different threads are interleaved while the order between actions of each single thread is maintained. For performance reasons, modern multi-processors as well as compilers may reorder some memory access operations. This leads to the adoption of weak (or relaxed) memory models such as TSO (Total Store Ordering). In this model, store operations are not immediately visible to all threads (as in SC). Each thread is supposed to have a store buffer where store operations are kept in order to be executed later. While the order of the store operations issued by a same thread are executed (i.e., written in the main memory) in the same order (i.e., the store buffers are FIFO), load operations by this thread can overtake pending stores in the buffer if they concern different variables, and read values from the main memory. Loads from a variable for which there is a store operation in the buffer gets the value of the last of such operation. The TSO model is in some sense the kernel of many common weak memory models [15,19].

Verifying programs by taking into account the effect of the weak memory models such as TSO is a nontrivial problem, both from the theoretical and the practical point of views. Although store buffers are necessarily finite in actual machines and implementations, we should not assume any fixed bound on their size in order to reason about the

* Partially supported by the french ANR-09-SEGI-016 project VERIDYC.

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 99–115, 2011.

correctness of general algorithms. For safety properties, the general question to address is whether there is a size of the buffers for which the program can reach some bad configuration, which is equivalent to check the reachability of bad configurations by considering unbounded buffers. This leads to the adoption of formal models (based on state machines with queues) for which the decidability of problems such as checking state reachability is not straightforward. It has been shown in [1] that the state reachability problem for TSO is actually decidable (for finite-data programs), but highly complex (nonprimitive recursive). This leaves open the problem of defining efficient verification techniques for TSO. Necessarily, such verification techniques should be based on upper/under-approximate analysis.

Roughly speaking, the source of complexity in TSO verification is that store buffers can encode lossy channels, and vice-versa. Then, the issue we address in this paper is how to define a verification approach for TSO that allows an efficient encoding of the store buffers, i.e., in a way that *does not depend on their size*. More precisely, we investigate an approach for reducing, with a limited overhead (i.e., a polynomial increase in the size of the program) the reachability problem under TSO (with unbounded store buffers) to the same problem under SC.

Our first idea is to consider the concept of context-bounded analysis in the case of TSO. Context-bounding has been shown (experimentally) to be a suitable notion of behavior coverage for effective bug detection in concurrent programs running under the SC model [14]. Moreover, this approach provides a decidable analysis (under SC) in the case of programs with recursive procedure calls [17]. In this paper, we extend this concept to TSO as follows. We consider that a *context* in this case is a computation segment where only one thread is active, and where all updates of the main memory use store operations taken from the store buffer of that thread. Then, we prove that for every fixed bound k, and for every concurrent program P, it is possible to construct, using a code-to-code translation, another concurrent program P' such that running P' under SC yields the same set of reachable (shared) states as running P under TSO with at most k context-switches for each thread. Our translation preserves the class of the original program in the sense that P and P' have the same features (e.g., recursive procedure calls, dynamic creation of threads, data manipulation). Basically, we show that encoding store buffers can be done using $O(k)$ additional copies of the shared variables as local variables. The obtained program has the same type of data structures and variables and the same control features (recursion, dynamic thread creation) as the original one. As a consequence, we obtain for instance that for finite-data programs, even when recursion is allowed, the context-bounded analysis of TSO programs is decidable (whereas the unrestricted reachability problem in this case is undecidable as in SC).

The translation we provide from TSO to SC, regardless of the decidability issue, does not depend fundamentally from the fact that we have a finite number of context switches for each thread. The key property we use is the fact that each store operation produced by some thread cannot stay in its store buffer for more than a bounded number of context switches of that thread. (This of course does not exclude that each thread may have an unbounded number of context switches.) Therefore, we define a notion for restricting the set of analyzed behaviors of TSO programs which consists in bounding the *age* of each store operation in the buffers. The age of a store operation, produced

by a thread T, is the number of context switches made by T since its production. We show that as before, for any bound on the age of all stores, it is possible to translate the reachability problem from TSO to SC. For the case of programs with recursion this translation does not provide a decision procedure. (The targeted class of programs is concurrent programs with an unbounded number of context switches.) However, in the case of finite-data programs without recursive procedures, this translation provides a decision procedure for the TSO reachability problem under store-age bounding (since obviously SC reachability for finite-state concurrent programs is decidable).

Our code-to-code translations allow to smoothly transfer present and future decidability and complexity results from the SC to the TSO case for the same class of programs. More importantly, our translations allow to use existing analysis and verification tools designed for SC in order to perform the same kind of analysis and verification for TSO. To show its practicability, we have applied our approach in checking that standard mutual exclusion protocols for SC are incorrect under TSO, using the tools POIROT [10] and ESBMC [4]. In our experiments, bugs appear for small bounds (≤ 2).

Related work. Context-bounded analysis has been developed in a series of paper in the recent year [17,3,14,11,8,9]. So far, it has been considered only for the SC memory model. As far as we know, the only works addressing the verification problem for TSO programs with unbounded store buffers are [1,13]. In [1], the decidability and the complexity of the state reachability problem under TSO (and other memory models) is considered for a finite number of finite-state threads. The decision procedure for TSO given in that paper is based on a reduction to the reachability problem in lossy channel systems, through a nontrivial and complex encoding. In [13], an approach based on Regular Model Checking is adopted. The paper proposes techniques for computing the set of reachable configurations in TSO programs. If the algorithm terminates, it provides the precise set of reachable configurations, however termination is not guaranteed.

2 Concurrent Programs

We define in this section the class of programs we consider. Basically, we consider concurrent programs with procedure calls and dynamic thread creation. We give the syntax of these programs and describe their semantics according to both SC (Sequential Consistency) and TSO (Total Store Order) memory models.

2.1 Syntax

The syntax of concurrent programs is given by the grammar in Fig. 1. A program has a finite set of processes defining the code executed by parallel threads that can be created dynamically by the spawn statement. The program has a distinguished process `main` that is initially executed to start running the program. We assume that there is a finite number of variables `Svar` that are shared by all the threads. They are used for the communication between threads at context switch points. We also assume that there is a finite number of variables `Gvar` that are global to all procedures. During its execution, each thread has its own copy of these global variables (that are not shared with the other threads) which can be used for value passing at procedure calls and returns. We consider

$$
\begin{array}{rcl}
\langle pgm \rangle & ::= & \texttt{Svar}\ \bar{s}\ \texttt{Gvar}\ \bar{g}\ \langle main \rangle \langle process \rangle^* \langle procedure \rangle^* \\
\langle main \rangle & ::= & \texttt{main p begin}\ \langle lstmt \rangle^+\ \texttt{end} \\
\langle process \rangle & ::= & \texttt{process p begin}\ \langle lstmt \rangle^+\ \texttt{end} \\
\langle procedure \rangle & ::= & \texttt{procedure f begin}\ \langle lstmt \rangle^+\ \texttt{end} \\
\langle lstmt \rangle & ::= & \texttt{loc} : \langle stmt \rangle\ \texttt{;} \\
\langle stmt \rangle & ::= & \langle simp\_stmt \rangle \mid \langle comp\_stmt \rangle \mid \langle sync\_stmt \rangle \\
\langle simp\_stmt \rangle & ::= & \texttt{skip} \mid \texttt{assume}(\langle pred \rangle) \mid \texttt{assert}(\langle pred \rangle) \mid \bar{\texttt{g}} := \langle expr \rangle \mid \texttt{call f} \mid \texttt{return} \\
\langle comp\_stmt \rangle & ::= & \texttt{if}(\langle pred \rangle)\ \texttt{then}\ \langle lstmt \rangle^+\ \texttt{else}\ \langle lstmt \rangle^+\ \texttt{fi} \mid \texttt{while}\ (\langle pred \rangle)\ \texttt{do}\ \langle lstmt \rangle^+\ \texttt{od} \\
\langle sync\_stmt \rangle & ::= & \texttt{atomic\_begin} \mid \texttt{atomic\_end} \mid \texttt{spawn p} \mid \texttt{fence} \mid \texttt{g} := \texttt{s} \mid \texttt{s} := \texttt{g}
\end{array}
$$

Fig. 1. The grammar for concurrent programs

that variables range over some (potentially infinite) data domain $\mathbb{D}$. We assume that we dispose of a language of expressions $\langle expr \rangle$ interpreted over $\mathbb{D}$, and of a language of predicates $\langle pred \rangle$ on global variables ranging over $\mathbb{D}$. The program has a finite number of control locations *Loc*. Its code is a nonempty sequence of labelled statements $\texttt{loc} : \langle stmt \rangle$ where `loc` is a control location, and $\langle stmt \rangle$ belongs to a simple language of C-like statements.

2.2 SC Semantics

We describe the semantics informally and progressively. Let us first consider the case of sequential programs where statements are restricted to simple statements $\langle simp\_stmt \rangle$ and composed statements $\langle comp\_stmt \rangle$. Then, the program has a single thread that can make procedure calls and manipulate only global variables. In that case, shared variables are omitted, and a configuration can be represented by a triple $\langle globals, \texttt{loc}, stack \rangle$ where *globals* is a valuation of the global variables, `loc` is a control location, and *stack* is a content of the call stack. The elements of this stack are the control locations of the pending procedure calls. A transition relation between these configurations can be defined as usual. At a procedure call, the current location of the caller is pushed in the stack, and the control moves to the initial location of the callee. At a procedure return, the first control location in the stack is popped, and the control moves to that location.

Now, for the general case, a concurrent program has several parallel threads $T_1, \ldots, T_n$ that have been created using the spawn statement. As mentioned above, each thread has its own copy of the global variables `Gvar` that is used throughout its procedure calls and returns, and all the threads share the variables in `Svar`. Then, a SC-configuration is a tuple of the form $\langle shared, thread_1, \ldots, thread_n \rangle$ for some $n \geq 1$, where *shared* is a valuation of the shared variables, and for each $i \in \{1, \ldots, n\}$, $thread_i$ is the local configuration of thread T_i. Such local configuration is defined as for a sequential program by a triple $\langle globals_i, \texttt{loc}_i, stack_i \rangle$, plus an additional flag $critical_i$ that indicates if the current thread is executing a critical section of the code that has to be executed atomically (without interference of other threads). When the thread executes an `atomic_begin` statement, this flag is set to 1, and it is set to 0 at the next `atomic_end`. The `spawn` statement creates a new thread, making the configuration of the program grow by the addition of the local configuration of the new thread (i.e., the number n of threads can get arbitrarily large, in general). Actions of different threads are interleaved in a

nondeterministic way, under the restriction that if a thread T has opened a critical section, no other thread can execute an action until T closes its section. In the SC model, write operations to shared variables are immediately visible to all threads. Then, a transition relation between global configurations is defined, where at each step one single thread is active. We denote this relation $\stackrel{i}{\Longrightarrow}_{SC}$ where $i \in \{1,\ldots,n\}$ is the index of the thread T_i that has performed the corresponding step.

2.3 TSO Semantics

In the TSO memory model, the SC semantics is relaxed by allowing that read operations can overtake write operations by the same thread on different shared variables. This corresponds to the use of FIFO buffers where write operations to shared variables can be stored and executed in a delayed way, allowing read operations from the main memory (but on different variables) to overtake them. We define hereafter an operational semantics corresponding to this memory model, in the spirit of the formal model defined in [1]. A store buffer is associated with each thread. Then, a global TSO-configuration of the program is defined as in the SC case, except that a local configuration of a thread includes also the content of its store buffer, i.e., it is a tuple of the form $\langle globals_i, \texttt{loc}_i, stack_i, critical_i, buffer_i\rangle$. Then, the semantics is defined as for SC, except for assignment operations involving shared variables, and for the synchronization actions `atomic_begin` and `atomic_end`. Let us consider each of these cases, and assume that the active thread is T_i: For an assignment of the form $\texttt{s} := \texttt{g}$ that writes some value `d` (the one stored in `g`) to the shared variable `s`, a pair $(\texttt{s},\texttt{d})$ is sent to the store buffer, that is, the buffer of T_i is updated to $buffer_i' = (\texttt{s},\texttt{d})buffer_i$. For an assignment of the form $\texttt{g} := \texttt{s}$ that loads a value from the shared variable `s` to the variable `g`, two cases can occur. First, if a pair $(\texttt{s},\texttt{d})$ is still pending in $buffer_i$, then the load returns the value `d` corresponding to the last of such pair in the buffer. Otherwise, the returned value is the one stored for `s` in the main memory. As for `atomic_begin` and `atomic_end`, they have the same semantics as in the SC cases, except that it is required that their execution can only occur when $buffer_i$ is empty. Notice that these statements allow in particular to encode fences, i.e., actions that cannot be reordered w.r.t. any other actions. Indeed, a fence can be encoded as `atomic_begin;atomic_end`.

In addition to transitions due to the different threads, memory updates can occur at any time. A memory update consists in getting some $(\texttt{s},\texttt{d})$ from some store buffer (of any thread) and updating the value of `s` in the main memory to `d`, i.e., if for some $j \in \{1,\ldots,n\}$, $buffer_j = buffer'_j(\texttt{s},\texttt{d})$, then `d` is stored in the main memory as a new value for `s`, and the buffer of T_j is updated to $buffer'_j$. Then, we can define a transition relation between global configurations $\stackrel{\alpha}{\Longrightarrow}_{TSO}$, where α is equal to the index $j \in \{1,\ldots,n\}$ if the transition corresponds to a memory update using $buffer_j$, or otherwise, to the index j of the thread T_j that has performed the transition step.

2.4 Reachability Problems

Let $\sharp \in \{SC, TSO\}$. We define $\Longrightarrow_\sharp$ to be the union of the relations $\stackrel{i}{\Longrightarrow}_\sharp$ for all $i \in \{1,\ldots,n\}$, and we denote by $\stackrel{*}{\Longrightarrow}_\sharp$ the reflexive-transitive closure of $\Longrightarrow_\sharp$.

Then, the $\sharp$-*reachability problem* is, given a $\sharp$-configuration γ and a valuation of the shared variables *shared*, to determine if there is a $\sharp$-configuration γ' such that: (1) the valuation of the shared variables in γ' is precisely *shared*, and (2) $\gamma \stackrel{*}{\Longrightarrow}_\sharp \gamma'$. In such a case, we say that γ' and *shared* are $\sharp$-*reachable* from γ.

Let us consider a computation $\rho = \gamma_0 \stackrel{\alpha_0}{\Longrightarrow}_\sharp \gamma_1 \stackrel{\alpha_1}{\Longrightarrow}_\sharp \gamma_2 \cdots \stackrel{\alpha_{m-1}}{\Longrightarrow}_\sharp \gamma_m$. A *context-switch* point in ρ is a configuration γ_j, for some $j \geq 1$, such that $\alpha_{j-1} \neq \alpha_j$. A *computation round* of a thread T_i in ρ is a computation segment (1) occurring between two consecutive points in ρ that are either context-switch or extremal points, and (2) where all transitions are labeled by the same index i, i.e., all transitions are either made by T_i, or are memory updates using the store buffer of T_i (in the case of TSO). Clearly, every computation can be seen as a sequence of computation rounds of different threads. In general, the number of rounds that a thread can have along a computation is unbounded.

Given a bound $k \in \mathbb{N}$, the *k-round* $\sharp$-*reachability problem* is, given a $\sharp$-configurations γ and a valuation of the shared variables *shared*, to determine if there is a $\sharp$-configuration γ' such that: (1) the valuation of the shared variables in γ' is precisely *shared*, and (2) γ' is reachable from γ by a computation where every thread T_i has at most k computation rounds. In that case, we say that γ' and *shared* are *k-round* $\sharp$-*reachable* from γ.

3 Bounded-Round Reachability: From TSO to SC

In this section we provide a code-to-code translation that, given a concurrent program P and a fixed bound $k \in \mathbb{N}$, builds another concurrent program P' which simulates P with the property that for any shared state *shared*, *shared* is k-round TSO-reachable in P iff *shared* is k-round SC-reachable in P'. The interesting feature of our translation is that the size of the constructed program P' is *linear* in the size of P. Furthermore, P' is in the same class of programs as P in the sense that it uses the same kind of control primitives (procedure calls and thread creation) and the same kind of data-structures and variables; the encoding of the unbounded store buffers requires only adding, as global variables, $(k+1)$ copies of the shared variables and k Boolean variables per process.

In the following, we assume that the number of rounds that a thread of P can have along any computation is bounded by $(k+1)$, and these rounds are indexed from 0 to k.

3.1 Simulating Store Buffers: Case $k = 1$

Before giving the details of the translation, let us present the main ideas behind it and justify its correctness. Assume (for the moment) that $k = 1$. Then, let us focus on the behavior of one particular thread, say T, and consider its computation rounds and its interactions with its environment (i.e., the set of all the other threads) at context switch points. For that, let us project computations on what is visible to T, i.e., the configurations are projected on the shared variables and the local configuration of T, and we only consider the two computation rounds of T which are of the form:

$$\langle \mathbf{shared_0}, (globals_0, \mathtt{loc}_0, stack_0, critical_0, buffer_0)\rangle \stackrel{*}{\Longrightarrow}_{TSO} \langle \mathbf{shared'_0}, (globals_1, \mathtt{loc}_1, stack_1, critical_1, \mathbf{buffer_1})\rangle \quad (1)$$

$$\langle \mathbf{shared_1}, (globals_1, \mathtt{loc}_1, stack_1, critical_1, \mathbf{buffer_1})\rangle \stackrel{*}{\Longrightarrow}_{TSO} \langle \mathbf{shared'_1}, (globals_2, \mathtt{loc}_2, stack_2, critical_2, buffer_2)\rangle \quad (2)$$

Notice that the local configurations of T at the end of round 0 and at the beginning of the round 1 are the same.

Encoding the store buffers. In the following, we show that we can use a finite number of global variables to encode the (unbounded) store buffers. This can be done based on two main observations. First, in order to execute correctly a load operation of T on some shared variable x, we need to know whether a store operation on x is still pending in its store buffer, and in this case, we need the last value of such operation, or otherwise we need the value of x in the main memory. Since in each round, only T is active and only operations in its store buffer can be used to modify the main memory, the number of information needed to execute correctly loads is finite and corresponds to the last values written by T to each of the variables composed with the initial content of the main memory (at the beginning of the round). For this purpose, we introduce a vector of data named *View* which is indexed with the shared variables of P. More precisely, $View[x]$ contains the valuation for the load of the variable x.

On the other hand, the order in which store operations of T (sent to the store buffer) on different variables have been consumed (written into the main memory) is not important. In fact, only the last consumed store operation to each variable is relevant. Again, this is true because only T is active during a round, and only its own store buffer can be used to update the main memory. Therefore, given a round j we also define (1) a Boolean vector $Mask_j$ such that $Mask_j[x]$ holds if there is a store operation on x in the buffer of T that is used to update the main memory at round j, and (2) a vector of data $Queue_j$ such that, if $Mask_j[x]$ holds then $Queue_j[x]$ contains the last value that will be written in the shared memory corresponding to x at round j (otherwise it is undefined).

Let us consider the concurrent program P' running under SC built from the concurrent program P by adding to each process of P the following global variables: (1) a vector of data named *View*, (2) two Boolean vectors $Mask_0$ and $Mask_1$, and (3) two vectors of data $Queue_0$ and $Queue_1$. Then, for any local TSO configuration $\langle shared, (globals, \texttt{loc}, stack, critical, buffer)\rangle$ of P, we can associate the following local SC configuration $\langle shared, ((globals, View, Mask_0, Mask_1, Queue_0, Queue_1), \texttt{loc}, stack, critical)\rangle$ of P' such that the following conditions are satisfied: (i) the value of $View[x]$ corresponds to the value of the last store operation to x still pending in the store buffer *buffer* if such operation exists, otherwise the value $View[x]$ is the value of the variable x in the main memory, and (ii) for every $j \in \{0,1\}$, $Mask_j[\texttt{x}]$ holds true iff at least one store operation on x pending in the store buffer *buffer* will update the shared memory in round j, and $Queue_j[\texttt{x}]$ contains the last pending value written into x and consumed at round j.

Simulation of P by P'. In the following, we construct for any two computation rounds of a thread of P, a two computation rounds of a thread of P' such that the invariants between the configurations of P and P' are preserved along the simulation.

For the issue of updating the main memory and of passing the store buffer from a round to the next one. We assume w.l.o.g. that the store buffer of any thread of P is empty at the end of the considered computation.

Let us consider first the special case where all store operations produced (sent to the store buffer) in round j are also consumed (written to the main memory) in the same round. It is actually possible to consider that all stores are immediately written to the main memory without store buffering, i.e., as in the SC model.

Consider now the case where not all stores produced in round 0 are consumed in round 0. So for instance, at the end of the execution of round 0 given by (1), we must ensure that the main memory contains $\mathbf{shared'_0}$, and we must pass $\mathbf{buffer_1}$ to the second round. The computation in round 0 can be seen as the concatenation of two subcomputations, ρ_0^0 where all produced stores are consumed in round 0, followed by ρ_1^0 where all stores are consumed in round 1. (Notice that, since the store buffer is a FIFO queue, store operations that are consumed in round 0 are necessarily performed (i.e., sent to the buffer) by T before those that will remain for round 1.) Then, it is clear that $\mathbf{shared'_0}$ is the result of executing ρ_0^0, and $\mathbf{buffer_1}$ contains all stores produced in ρ_1^0.

During the simulation of round 0 by P', the point p separating ρ_0^0 and ρ_1^0 is nondeterministically guessed. The stores produced along the segment ρ_0^0 are written immediately to the main memory as soon as they are produced. So, when the point p is reached, the content of the main memory is precisely $\mathbf{shared'_0}$. During the simulation of ρ_1^0, two operations are performed by P': (1) maintaining the view of T in round 0 by updating *View*, and (2) keeping in $Mask_1$ and $Queue_1$ the information about the last values sent to each variable in ρ_1^0. So, at the end of round 0, the pair $Mask_1$ and $Queue_1$ represent the summary of $\mathbf{buffer_1}$.

The simulation of round 1 by P' starts from the new state of the shared memory $\mathbf{shared_1}$ (which may be different from $\mathbf{shared'_0}$ as other threads could have changed it). Then, the main memory is immediately updated by P' using the content of $Mask_1$ and $Queue_1$. Intuitively, since all stores in $\mathbf{buffer_1}$ are supposed to be consumed in round 1, and again since T is the only active thread, we execute all these store operations at the beginning of round 1. The vector *View* is now updated as follows. Starting from the view obtained at the end of the previous round, we only change the valuation of all those variables x for which no store operations are pending in the store buffer for it. For all such variables x we update its valuation with the one of x contained in the shared memory ($View[x] := x$). Now, the simulation of round 1 can proceed. Since all stores produced in this last round are supposed to be consumed by the end of this round, they are immediately written into the shared memory.

3.2 Simulating Store Buffers: General Case

The generalization to bounds k greater than 1 requires some care. The additional difficulty comes from the fact that stores produced at some round will not necessarily be consumed in the next one (as in the previous case), but may stay in the buffer for several rounds. We start by defining the set of shared and global variables of P', denoted S' and G' respectively, and describe the role they play in P':

Shared variables: the set of shared variables of P and P' are the same, that is, $S' = S$, with $dom_P(\mathtt{x}) = dom_{P'}(\mathtt{x})$ for every $\mathtt{x} \in S$.

Global variables: The set of global variables P' is defined as

$G' = G \cup \left(\bigcup_{j=0}^{k} (Queue_j \cup Mask_j) \right) \cup View \cup \{\mathtt{r\_TSO}, \mathtt{r\_SC}, \mathtt{sim}\}$, where

- for each $j \in \{0,\ldots,k\}$, $Queue_j = \{\mathtt{queue\_j\_x} \mid \mathtt{x} \in S\}$, and $dom_{P'}(\mathtt{queue\_j\_x}) = dom_P(\mathtt{x})$ for every $\mathtt{x} \in S$;
- for each $j \in \{0,\ldots,k\}$, $Mask_j = \{\mathtt{mask\_j\_x} \mid \mathtt{x} \in S\}$, and $dom_{P'}(\mathtt{queue\_j\_x}) = \{\mathtt{true}, \mathtt{false}\}$ for every $\mathtt{x} \in S$;

- *View* $= \{\texttt{view\_x} \mid \texttt{x} \in S\}$, with $dom_{P'}(\texttt{view\_x}) = dom_P(\texttt{x})$ for every $\texttt{x} \in S$;
- `r_TSO` and `r_SC` are two fresh variables whose domain is the set of round indices, that is, $\{0,\ldots,k\}$;
- `sim` is a new variable whose domain is $\{\texttt{true},\texttt{false}\}$.

For sake of simplicity, we denote a variable named `queue_j_x` also as $Queue_j[\texttt{x}]$, for every j and $\texttt{x} \in S$. Similarly, for the set $Mask_j$ and *View*.

Next, we associate a "meaning" to each variable of P' which represents also the invariant we maintain during the simulation of P by P'.

Invariants for variables. The shared variables of S' keep the same valuation of the ones of P at context-switch points along the simulation. The variables in *View* is defined as in Sec. 3.1. The variables $Queue_j$ and $Mask_j$, with $j \in \{0,\ldots,k\}$, maintain the invariant that at the beginning of the simulation of round j, $Mask_j[\texttt{x}]$ holds true iff at least one write operation on `x` produced in the previous rounds will update the shared memory in round j, and $Queue_j[\texttt{x}]$ contains the last value written into `x`. Variable `r_SC` keeps track of the round under simulation, and `r_TSO` maintains the round number in which next write operation will be applied to the shared-memory. The global variable `sim` holds true iff the thread is simulating a round (which is mainly used to detect when a new round starts), and the global variables in G are used in the same way they were used in P. The program P' we are going to define maintains the invariants defined above along all its executions.

Simulation of P by P'. Here we first describe how P' simulates P for $k = 2$, and then generalize it for arbitrary values of k. For round 0, there is of course the case where all stores are consumed in the same round, or in round 0 and in round 1. Those cases are similar to what we have seen for $k = 1$. The interesting case is when there are stores that are consumed in round 2. Let us consider that the computation in round 0 is the concatenation of three sub-computations ρ_0^0, ρ_1^0, and ρ_2^0 such that ρ_i^0 represent the segment where all stores are consumed in round i.

The simulation of ρ_0^0 and ρ_1^0 is as before. (Stores produced in ρ_0^0 are written immediately to the main memory, and stores produced in ρ_1^0 are summarized using $Mask_1$ and $Queue_1$.) Then, during the simulation of ρ_2^0, the sequence of stores is summarized using a new pair of vectors $Mask_2$ and $Queue_2$. (Notice that stores produced in ρ_1^0 and ρ_2^0 are also used in updating *View* in order to maintain a consistent view of the store buffer during round 0.)

Then, at the beginning of round 1 (i.e., after the modification of the main memory due to the context switch), the needed information about the store buffer can be obtained by composing the contents of $Mask_2$ and $Queue_2$ with $Mask_1$ and $Queue_1$ which allow us to compute the new valuation of *View*. (Indeed, the store buffer at this point contains all stores produced in round 0 that will be consumed in rounds 1 and 2.) Moreover, for the same reason we have already explained before, it is actually possible at this point to write immediately to the memory all stores that are supposed to be executed in round 1. After this update of the main memory, the simulation of round 1 can start, and since there are stores in the buffer that will be consumed in round 2, this means that all forthcoming stores are also going to be consumed in round 2. Therefore, during this simulation, the vectors $Mask_2$ and $Queue_2$ must be updated. At the end of round 1,

these vectors contain the summary of all the stores that have been produced in rounds 0 and 1 and that will consumed in round 2.

After the change of the main memory due to the context switch, the memory content can be updated using $Mask_2$ and $Queue_2$ (all stores in the buffer can be flushed), and the simulation of round 2 can be done (by writing immediately stores to the memory).

The extension to any k should now be clear. In general, we maintain the invariant that at the beginning of every round j, for every $\ell \in \{j, \ldots, k\}$, the vectors $Mask_\ell$ and $Queue_\ell$ represent the summary of all stores produced in rounds $i < j$ that will be consumed at round ℓ. Moreover, we also know what is the round $r \geq j$ in which the next produced store will be consumed. The simulation starts of round j by updating the main memory using the content of $Mask_j$ and $Queue_j$, and then, when $r = j$, the simulation is done by writing stores to the memory, and when r is incremented (nondeterministically), the stores are used to update $Mask_r$ and $Queue_r$.

From what we have seen above, it is possible to simulate store buffers using additional copies of the shared variables, and therefore, it is possible to simulate the TSO behaviors of a concurrent program P under a bounded number of rounds by SC behaviors of a concurrent program P'. Notice that the latter is supposed to be executed under the SC semantics without any restriction on its behaviors. In order to capture the fact that P' will perform only execution corresponding to rounds in P, we must enforce in the code of P' that the simulation of each round of P must be done in an atomic way.

3.3 Code-to-Code Translation

In this section we provide our code-to-code translation from P to P'. The translation from P to P' that we provide is quite straightforward except for particular points in the simulation: (1) at the beginning of the simulation of each thread, (2) at the beginning of the simulation of each round j, with $j > 0$, (3) at the end of the simulation of each round, (4) during the execution of a statement $\mathtt{x} := g$, where $\mathtt{x}$ is a shared variables, and (5) the execution of a `fence` statement. Let us assume that P has $S = \{\mathtt{x\_1}, \mathtt{x\_2}, \ldots, \mathtt{x\_n}\}$ as a set of shared variables and G as a set of global variables. Next, we describe the procedures for these cases, which are used as building blocks for the general translation.

Init of each thread. Before starting the simulation of a thread, we set both `r_TSO` and `r_SC` to 0. Then, we initialize the *view-variables* to the evaluation of the shared variables, as the store-buffer is initially empty and the valuation of the view coincides with that of the shared variables. Finally, we set to `false` the variables of all *masks*. Procedure `init_thread()` is shown in Fig. 2.

```
procedure init_thread()
begin
 atomic_begin;   sim := true;   r_TSO := r_SC := 0;
     // set the view to the shared valuation
 for(i=1,i<=n,i++) do view_x_i := x_i; od
    // initialize the masks
 for(j=0,j<=k,j++) do mask_j_x_1:=mask_j_x_2:=...:=mask_j_x_n:= false; od
end
```

Fig. 2. Procedure `init_thread()`

```
procedure init_round()
begin
[*] if(r_SC == k) then atomic_end; assume(false);//last round simulated
[*] else
[*]  if(r_TSO == r_SC) then r_TSO:=r_TSO+1; fi //update r_TSO if needed
[*]  r_SC := r_SC+1; // increment of the round number
[*] fi
    for(j=0,j<=k,j++) do
      if (r_SC == j) then
        for(i=1,i<=n,i++) do
           // updating the shared memory
           if (mask_j_x_i) then x_i :=queue_j_x_i; mask_j_x_i:=false; fi
           // rebuild the view
[+]        if (!mask_j_x_i & ... & !mask_k_x_i) then view_x_i := x_i; fi
         od  fi  od
end
```

Fig. 3. Procedure `init_round`()

Starting a new round. When a new round is "detected" we accomplish the following operations. If the last round has been simulated we close the atomic section and block the execution of the thread. Otherwise, we increment `r_SC`, as well as `r_TSO` in case it becomes smaller than `r_SC` (next write operation can only modify the shared memory starting from the current round). Then we dump the part of the store-buffer that was supposed to change the shared memory during the execution of round `r_SC`. Let `r_SC` $= j$. The way we simulate such an operation is by using $Mask_j$ and $Queue_j$: for every shared variable `x`, we assign to `x` the value $Queue_j[\mathtt{x}]$ provided $Mask_j[\mathtt{x}]$ holds true. The last step is that of updating the view for the current round. A variable $View[\mathtt{x}]$ changes its valuation if no write operation is pending for `x` in the store-buffer, and its new value is that variable `x` in the shared memory ($View[\mathtt{x}] := \mathtt{x}$). Procedure `init_round` of Fig. 3 encodes the phases described above.

Finally, procedure `is_init_round`() of Fig. 4 detects that a new round has started checking that `sim` holds $false$. In such a case, we open an atomic section and set `sim` to `true`, and then call procedure `init_round` to initialize the round.

Terminating a round. We terminate non-deterministically a round by setting the variable `sim` to `false` and then closing the atomic section. (Next time the current thread will be scheduled it detects that a new round is started by checking the valuation of `sim`.) Procedure `is_end_round`() of Fig. 4 encoperates such operations.

Write into a shared variable. Consider a statement $\mathtt{x} := \mathtt{g}$ where `x` and `g` are a shared and global variable of P, respectively. In the simulation of such assignment, we first update the view for `x` to `g`. The next step consists in incrementing non-deterministically the value of the auxiliary variable `r_TSO` which represent the round where the current write operation will occur in the memory. Now, let `r_TSO` $= i$. In case `r_SC` is equal to i, we update `x` in the shared-memory. Instead, if `r_SC` $<$ `r_TSO`, we update $Mask_i[\mathtt{x}]$ to `true` and $Queue_i[\mathtt{x}]$ to `g` which captures that the write operation $\mathtt{x} := \mathtt{g}$ will modify the shared memory exactly at round i and it is the last operation for `x`. Notice that if

```
procedure is_init_round()
begin
  if ( !sim ) then
    atomic_begin;
    sim := true;
    init_round();
  fi
end
```

```
procedure is_end_round()
begin
  if (*) then
     sim := false;
     atomic_end;
  fi
end
```

```
procedure fence()
begin
 if(r_TSO!=r_SC)
 then
    atomic_end;
    assume(F);
  fi
end
```

Fig. 4. Procedures `is_init_round`(), `is_end_round`(), and `fence`()

```
procedure memory_update_x(g)
begin
     view_x:=g; // updating the view
     // non-deterministically increase r_TSO
[*]  while (*) do  if (r_TSO < k) then r_TSO:=r_TSO + 1; fi od
     if (r_SC==r_TSO) then x:=g; // shared memory update
     else  // updating the mask and the queue
       for(i=0,i<=k,i++) do
          if (r_TSO==i) then mask_i_x:=true; queue_i_x:=g; fi od
     fi
end
```

Fig. 5. Procedure `memory_update_x`, for each shared variable `x`

another write operation will be performed for `x`, when `r_TSO` contains the value i, then the value of $Queue_i[\mathrm{x}]$ will contain only the latest operation, and the previous value will be overwritten thus reestablishing the invariant. The procedure `memory_update_x`(`g`) in Fig. 5 subsumes the operations described in Sec. 3.2.

Fences. The statement `fence` is simply translated into a procedure, called `fence`(), that checks whether `r_TSO` is equal to `r_SC` and in case they are different blocks the execution. However, before blocking it, it first executes the statement `atomic_end` so that other threads can continue their evolution. Fig. 4 illustrates procedure `fence`().

General translation. We are now ready to give the general translation by defining a map $[\![\cdot]\!]_{tr}$ in which $P' = [\![P]\!]_{tr}$. The definition of the translation is given in Fig. 6. The new program P' first declares the variables as described in Sec. 3.2.

- $\overline{\texttt{mask}}$ is the list of the variables `mask_i_x`, for all $i \in \{0,\ldots,k\}$ and $\mathrm{x} \in S$;
- $\overline{\texttt{queue}}$ is the list of the variables `queue_i_x` for all $i \in \{0,\ldots,k\}$ and $\mathrm{x} \in S$;
- $\overline{\texttt{view}}$ is the list of the variables `view_i_x` for all $i \in \{0,\ldots,k\}$ and $\mathrm{x} \in S$.

Each `process` procedure starts with a call to procedure `init_thread`() that initializes the auxiliary variables used for the simulation. Then, the translation consists of an in-place replacement of each statement. Each statement `stmt` of P is translated by the sequence of statements `is_init_round`(); $[\![\texttt{stmt;}]\!]_{\texttt{tr}}$ `is_end_round`();. The call to `is_init_round`() checks whether a new round has just started and hence appropriately initialize the variables for the simulation of the new round; the call `is_end_round`();

$$
\begin{aligned}
[\![\textbf{Svar}\ \bar{s}\ \textbf{Gvar}\ \bar{g}\ \langle main\rangle\langle process\rangle^*\langle procedure\rangle^*\,]\!]_{tr} &\stackrel{\text{def}}{=} \textbf{Svar}\ \bar{s}\ \textbf{Gvar}\ \bar{g},\overline{\texttt{mask}},\overline{\texttt{queue}},\overline{\texttt{view}},\texttt{r\_TSO},\texttt{r\_SC},\texttt{sim}\\
&\quad [\![\langle main\rangle]\!]_{tr}\ [\![\langle process\rangle]\!]^*_{tr}[\![\langle procedure\rangle]\!]^*_{tr}\\
[\![\textbf{main}\ \mathrm{p}\ \textbf{begin}\ \langle lstmt\rangle^+\ \textbf{end}]\!]_{tr} &\stackrel{\text{def}}{=} \textbf{main}\ \mathrm{p}\ \textbf{begin}\ \texttt{init\_thread}();\ [\![\langle lstmt\rangle]\!]^+_{tr}\ \textbf{end}\\
[\![\textbf{process}\ \mathrm{p}\ \textbf{begin}\ \langle lstmt\rangle^+\ \textbf{end}]\!]_{tr} &\stackrel{\text{def}}{=} \textbf{process}\ \mathrm{p}\ \textbf{begin}\ \texttt{init\_thread}();\ [\![\langle lstmt\rangle]\!]^+_{tr}\ \textbf{end}\\
[\![\textbf{procedure}\ \mathrm{p}\ \textbf{begin}\ \langle lstmt\rangle^+\ \textbf{end}]\!]_{tr} &\stackrel{\text{def}}{=} \textbf{procedure}\ \mathrm{p}\ \textbf{begin}\ [\![\langle lstmt\rangle]\!]^+_{tr}\ \textbf{end}\\
[\![\texttt{loc}:\langle stmt\rangle\,\textbf{;}\,]\!]_{tr} &\stackrel{\text{def}}{=} \texttt{is\_init\_round}();\texttt{loc}:[\![\langle stmt\rangle]\!]_{tr}\,\textbf{;}\ \texttt{is\_end\_round}()\\
[\![\textbf{skip}]\!]_{tr} &\stackrel{\text{def}}{=} \textbf{skip}\\
[\![\textbf{if}\ \ (\langle pred\rangle)\ \textbf{then}\ \langle lstmt\rangle^+\ \textbf{else}\ \langle lstmt\rangle^+\ \textbf{fi}]\!]_{tr} &\stackrel{\text{def}}{=} \textbf{if}\ \ (\langle pred\rangle)\ \textbf{then}\ [\![\langle lstmt\rangle]\!]^+_{tr}\ \textbf{else}\ [\![\langle lstmt\rangle]\!]^+_{tr}\ \textbf{fi}\\
[\![\textbf{while}\ (\langle pred\rangle)\ \textbf{do}\ \langle lstmt\rangle^+\ \textbf{od}]\!]_{tr} &\stackrel{\text{def}}{=} \textbf{while}\ (\langle pred\rangle)\ \textbf{do}\ [\![\langle lstmt\rangle]\!]^+_{tr}\ \textbf{od}\\
[\![\textbf{assume}\ (\langle pred\rangle)]\!]_{tr} &\stackrel{\text{def}}{=} \textbf{assume}\ (\langle pred\rangle)\\
[\![\textbf{assert}\ (\langle pred\rangle)]\!]_{tr} &\stackrel{\text{def}}{=} \textbf{assert}\ (\langle pred\rangle)\\
[\![\bar{g} := \langle expr\rangle]\!]_{tr} &\stackrel{\text{def}}{=} \bar{g} := \langle expr\rangle\\
[\![\textbf{call}\ \mathrm{f}\ ]\!]_{tr} &\stackrel{\text{def}}{=} \textbf{call}\ \mathrm{f}\\
[\![\textbf{return}]\!]_{tr} &\stackrel{\text{def}}{=} \textbf{return}\\
[\![\textbf{atomic\_begin}]\!]_{tr} &\stackrel{\text{def}}{=} \textbf{atomic\_begin;}\ \texttt{fence}()\\
[\![\textbf{atomic\_end}]\!]_{tr} &\stackrel{\text{def}}{=} \texttt{fence}();\ \textbf{atomic\_end}\\
[\![\textbf{spawn}\ \mathrm{p}]\!]_{tr} &\stackrel{\text{def}}{=} \textbf{spawn}\ \mathrm{p}\\
[\![\textbf{fence}]\!]_{tr} &\stackrel{\text{def}}{=} \texttt{fence}()\\
[\![\mathrm{g} := \mathrm{s}]\!]_{tr} &\stackrel{\text{def}}{=} \mathrm{g} := \texttt{view\_s}\\
[\![\mathrm{s} := \mathrm{g}]\!]_{tr} &\stackrel{\text{def}}{=} \texttt{memory\_update\_s}(\mathrm{g})
\end{aligned}
$$

Fig. 6. Translation map $[\![\cdot]\!]_{tr}$

allows to non-deterministically terminate a round at any point in the simulation. The remaining part of the translation concerns the translation of each single statement `stmt`:

- $\mathrm{g} := \mathrm{x}$ is translated into $\mathrm{g} := \mathit{View}_x$;
- $\mathrm{x} := \mathrm{g}$ is replaced with the procedure call $\texttt{memory\_update\_s}(\mathrm{g})$;
- `fence` is translated as the call to the procedure `fence`();
- `atomic_begin` (resp. `atomic_end;`) is translated into the sequence `atomic_begin; fence`() (resp. `fence`(); `atomic_end`);
- All remaining kind of statements remain unchanged in the translation.

From the construction given above, and the reasoning followed in Sec. 3.2 we can prove the following theorem:

Theorem 1. *Let k be a fixed positive integer. A shared state shared is k-round TSO-reachable in P if and only if shared is SC-reachable in P'. Furthermore, if shared is SC-reachable in P' then shared is k'-round TSO-reachable in P for some $k' \leq k$. Moreover, the* size *of P' is linear in the size of P.*

4 Bounded Store-Age Reachability

In this section, we introduce a new notion for restricting the set of behaviors of concurrent programs to be analyzed under TSO. We impose that each store operation produced

by a thread T can not stay in the store-buffer more than k consecutive rounds. (Notice that this notion does not restrict the number of rounds that the thread T may have.) We show that, under this restriction, it is still possible to define a code-to-code translation (similar to that of Sec. 3) that associates with each concurrent program P another concurrent program P' such that running P' under SC captures precisely the set of behaviors of P under TSO. More precisely, we associate to each store operation an *age*. The age is initialized at 0 when this store operation is produced by T and sent in the store-buffer. Now, this age is incremented at each context-switch of thread T.

Let $k \in \mathbb{N}$ be a fixed bound. The *k-store-age* TSO-*reachability problem* is, given a TSO-configurations γ and a valuation of the shared variables *shared*, to determine if there is a TSO-configuration γ' such that: (1) the valuation of the shared variables in γ' is precisely *shared*, and (2) γ' is reachable from γ by a computation where at each step all the pending store operations have an age equal or less than k.

Let us consider a concurrent program P defined as in Sec. 3. In the following, we construct another concurrent program P' such that the k-store-age TSO-reachability problem for P can be reduced to the SC-reachability problem for P'. The provided code-to-code translation is very similar to the one given in Sec. 3. In fact, if we use the previous translation to simulate a thread T of P, we need to use an unbounded number of vectors of type *Mask* and *Queue*. The key observations (to overcome this difficulty) are that : (1) in order to simulate a round j of a thread T, we only use vectors $Mask_i$ and $Queue_i$ with $i \geq j$, and (2) at each moment of the simulation of a round j, we need only the vectors $Mask_l$ and $Queue_l$ with $l \leq j+k$ (since the age of any store operation is bounded by k). Therefore, we can define our translation using only k vectors of type *Mask* and *Queue* in a circular manner (modulo k). For instance, if the current simulated round of the thread T is 1, the variables $Mask_0$ and $Queue_0$ can be used in the simulation of the round $k+1$. Technically, we introduce only two modifications in the translation given in Sec. 3:
In Fig. 3, the piece of code marked with [*] is replaced with the following one:

```
[*]  // update r_TSO if needed
[*]  if (r_TSO == r_SC) then  r_TSO := (r_TSO+1 mod k+1); fi
[*]  // resetting the boolean vector mask_i
[*]  if (r_SC == i) then  mask_i_x_1 :=...:= mask_i_x_n := false;  fi
[*]  r_SC := (r_SC+1 mod k+1);  // increment of the round number
```

In Fig. 3 the line of code marked with [+] is replaced with the following one:

```
[+] if (!mask_1_x_i & ... & !mask_k_x_i) then view_x_i := x_i; fi
```

In Fig. 5 we replace the line of code marked with [*] with the following:

```
[*] if ( (r_SC - r_TSO) mod k+1 != 1) then r_TSO:=(r_TSO+1 mod k+1); fi
```

Finally, the relation between the given concurrent program P and the constructed program $P' = [\![P]\!]_{tr}$ is given by the following theorem:

Theorem 2. *A shared state shared is k-store-age TSO-reachable in P if and only if SC-reachable in P'. Moreover, the size of P' is linear in the size of P.*

Table 1. Experimental results for 4 mutual exclusion protocols by using POIROT and ESBMC

Mutual exclusion Protocols	**Poirot** ($L = 2$) Time (s)			**ESBMC** Time (s)	
	Version with no fences (Buggy for TSO) ($D = 1$)	Version with fences (Correct for TSO) ($D = 1$)	($D = 2$)	Version with no fences (Buggy for TSO) ($L = 2, C = 3$)	Version with fences (Correct for TSO) ($L = 3, C = 4$)
DEKKER	7	6	72	-	6
LAMPORT	26	110	1608	1	7
PETERSON	5	6	47	1	1
SZYMANSKI	8	6	978	1	6

5 Experiments

To show the practicability of our approach, we have experimented its application in detecting bugs due to the TSO semantics. For that, we have considered four well-known mutual exclusion protocols designed for the SC semantics: Dekker's [5], Lamport's [12], Peterson's [16], and Szymanski's [18]. All of these protocols are incorrect under TSO. Through our translations, we have analyzed the behaviors of these protocols under TSO using two SMT-based bounded model-checkers for SC concurrent programs. Our experimental results show that errors due to TSO appear within few rounds, and that off-the-shelf analysis tools designed for the SC semantics can be used for their detection.

In more details, we consider the four protocols mentioned above instantiated for two threads. We consider for each protocol two versions, one without fences (the original version of the protocol) that is buggy, and one with fences (neutralizing TSO) which is known to be correct. We have encoded each of these protocols (with and without fences) as C programs and manually translated by using the k-store-age translation with $k = 2$. We have instrumented the obtained C programs for both POIROT [10] and ESBMC [4] – two SMT-based bounded model-checkers for SC concurrent programs.

Table 1 illustrates the results of the analysis for the four mutual exclusion protocols we carried with both POIROT and ESBMC. The parameters L in the table indicate the number of loop unrolling. POIROT considers all runs by bounding L and the number of *delays* (we refer to [6] for the definition of delay). In our experiments with POIROT, we consider $L = 2$, and a bound D = 1 or D = 2 (= to the number of delays + 1). Turning to ESBMC, it analyzes all executions by bounding the number of loop unrolling and the number of context-switches. In the experiments with ESBMC, we consider a bound $L = 2$ or $L = 3$ on the number of loop unrolling, and a bound $C = 3$ or $C = 4$ on the number of context-switches. Both of tools are able to answer correctly, i.e., by finding the bugs for the buggy versions, except that ESBMC does not answer correctly for the buggy version of Dekker.)

6 Conclusion

We have presented a code-to-code translation from concurrent to concurrent programs such that the reachable shared states of the obtained program running under SC is exactly the same set of reachable shared states of the original program running under the

TSO semantics. The main characteristic of our translations is that it does not introduce any other auxiliary storage to model store buffers but only requires few copies of the shared variables that are local to threads in the resulted translated program (this is important for compositional analyses which track at each moment only one copy of the locals). Furthermore, our translations produce programs of linear size with respect the original ones, provided a constant value of k. Such characteristics allows, and this is the main interest of our approach, the use for relaxed memory models of mature tools designed for the SC semantics (such as BDD-based model-checkers [7], SMT/SAT-based model-checkers [10,4]) as well as tools for sequential analysis based on compositional sequentialization techniques for SC concurrent programs [11,8,6].

Moreover, our translations allow to transfer decidability and complexity results from the SC to the TSO case. In the following we discuss on the decidability/undecidability of the k bounded-round and k-store-age TSO-reachability for concurrent programs with variables ranging over finite domains. We consider first the case in which all processes are non-recursive. When a finite number of threads are involved in the computation, the problem is decidable by using the classical reachability algorithm for finite state concurrent programs. The same problem remains decidable if we add dynamic thread creation, by a reduction to the coverability problem for Petri nets [2]. On the other hand, if we have at least two recursive threads involved in the computation, the k-store-age TSO-reachability becomes undecidable for any k: For every concurrent program P we can construct a concurrent program P' (obtained from P by inserting a fence statement at each control location of P) such that the SC-reachability problem for P (which is an undecidable problem in general) can be reduced to the k-store-age TSO-reachability problem for P'. However, by retaining recursion and using context-bounded analysis for concurrent programs and our translation we can claim the decidability of a variety of restrictions of the k-store-age (and k bounded-round) TSO-reachability. For instance, TSO bounded context-switch reachability is decidable for finite number of threads [17], as well as for bounded round-robin reachability for the parametrized case [9]. Moreover, decidability results concerning the analysis of programs with dynamic thread creation for k context-switches per thread [2] can also be transferred to the TSO case.

Acknowledgments. We would like to thank Akash Lal and Lucas Cordeiro for their help with POIROT and ESBMC.

References

1. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: POPL, pp. 7–18. ACM, New York (2010)
2. Atig, M.F., Bouajjani, A., Qadeer, S.: Context-bounded analysis for concurrent programs with dynamic creation of threads. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 107–123. Springer, Heidelberg (2009)
3. Bouajjani, A., Esparza, J., Schwoon, S., Strejček, J.: Reachability analysis of multithreaded software with asynchronous communication. In: Sarukkai, S., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 348–359. Springer, Heidelberg (2005)
4. Cordeiro, L., Fischer, B.: Verifying multi-threaded software using SMT-based context-bounded model checking. In: ICSE. ACM/IEEE (2011)

5. Dijkstra, E.W.: Cooperating sequential processes. Technical report, Technological University, TR EWD-123 (1965)
6. Emmi, M., Qadeer, S., Rakamaric, Z.: Delay-bounded scheduling. In: POPL, pp. 411–422. ACM, New York (2011)
7. La Torre, S., Madhusudan, P., Parlato, G.: Analyzing recursive programs using a fixed-point calculus. In: PLDI, pp. 211–222. ACM, New York (2009)
8. La Torre, S., Madhusudan, P., Parlato, G.: Reducing context-bounded concurrent reachability to sequential reachability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 477–492. Springer, Heidelberg (2009)
9. La Torre, S., Madhusudan, P., Parlato, G.: Model-checking parameterized concurrent programs using linear interfaces. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 629–644. Springer, Heidelberg (2010)
10. Lahiri, S., Lal, A., Qadeer, S.: Poirot. Microsoft Research, `http://research.microsoft.com/en-us/projects/poirot`
11. Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 37–51. Springer, Heidelberg (2008)
12. Lamport, L.: A fast mutual exclusion algorithm. ACM Trans. Comput. Syst. 5(1), 1–11 (1987)
13. Linden, A., Wolper, P.: An automata-based symbolic approach for verifying programs on relaxed memory models. In: van de Pol, J., Weber, M. (eds.) Model Checking Software. LNCS, vol. 6349, pp. 212–226. Springer, Heidelberg (2010)
14. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI, pp. 446–455. ACM, New York (2007)
15. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO (extended version). Technical Report UCAM-CL-TR-745, Univ. of Cambridge (2009)
16. Peterson, G.L.: Myths about the mutual exclusion problem. IPL 12(3), 115–116 (1981)
17. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
18. Szymanski, B.K.: A simple solution to lamport's concurrent programming problem with linear wait. In: ICS, pp. 621–626 (1988)
19. Weaver, D., Germond, T. (eds.): The SPARC Architecture Manual Version 9. PTR Prentice Hall, Englewood Cliffs (1994)