

Using Shared Memory Abstractions to Design Eager Sequentializations for Weak Memory Models

Ermenegildo Tomasco¹, Truc L. Nguyen¹, Bernd Fischer², Salvatore La Torre³, and Gennaro Parlato¹

¹ Electronics and Computer Science, University of Southampton, UK

² Division of Computer Science, Stellenbosch University, South Africa

³ Dipartimento di Informatica, Università di Salerno, Italy

Abstract. Sequentialization is one of the most promising approaches for the symbolic analysis of concurrent programs. However, existing sequentializations assume sequential consistency, which modern hardware architectures no longer guarantee. In this paper we describe an approach to embed weak memory models within eager sequentializations. Our approach is based on the separation of intra-thread computations from inter-thread communications by means of a shared memory abstraction (SMA). We give details of SMA implementations for the SC, TSO, and PSO memory models that are based on the idea of individual memory unwindings. We use our approach to implement a new, efficient BMC-based bug finding tool for multi-threaded C programs under SC, TSO, or PSO based on these SMAs, and show experimentally that it is competitive to existing tools.

1 Introduction

Developing correct concurrent programs is a complex and difficult task, due to the large number of possible concurrent executions that must be considered. Modern multi-core hardware architectures with *weak memory models* (WMMs) have made this task even harder, because they introduce additional executions that can lead to seemingly counter-intuitive results that confound the developers' reasoning.

Testing remains the most widely used approach to finding bugs; however, it is ineffective for bugs that manifest themselves only rarely and are difficult to reproduce [20]. Such “Heisenbugs” are unfortunately more prevalent with WMMs. Static verification approaches that handle individual executions *explicitly* face the same state space explosion as testing, even with optimizations that eliminate redundant executions. We thus need approaches that can handle multiple concurrent executions *symbolically*.

However, building efficient symbolic verification tools for realistic programming languages like C is hard and extending them for concurrency is harder yet. Tools thus often fold the concurrency handling deep into their general verification approaches (see [1,4,7,9,24,25]), focussing on a specific memory model, typically sequential consistency (SC). This introduces a strong coupling between the two aspects, which makes it hard to reuse existing tools and to generalize solutions to other memory models.

Our goal is to improve on this without losing the efficiency of existing approaches. For this, we separate the computation (i.e., individual threads) and the communication (i.e., shared memory) concerns of concurrent programs as follows. First, we replace all

standard concurrency operations in multi-threaded programs (such as shared memory reads, writes, and allocations, thread creation and termination, and synchronization operations such as thread join and mutex locking and unlocking) by abstract operations over an API called *shared memory abstraction* (SMA). We then provide efficient SMA implementations tailored for the targeted WMM and class of verification algorithms so that we can reuse existing efficient originally designed for SC now for WMMs.

The notion of SMA was originally introduced in [22] where the focus was on *lazy* sequentialization techniques, i.e., based on state-space search algorithms exploring only reachable states. A main achievement of [22] is an efficient SMA implementation based on temporal circular doubly-linked lists. The correctness of such SMAs is only guaranteed when SMA operations are invoked in the program execution order.

Here, we extend the SMA-based design from [22] to *eager* model-checking algorithms (in the style of Lal/Reps sequentialization [17]). In eager approaches, each thread is analysed in isolation thus avoiding the state-space explosion (cross product of the thread local states) in which lazy approaches may incur. However, eager explorations guess variable valuations when a read operation matches a value written by another thread that has not been explored yet, and maintain auxiliary information to discard unfeasible executions resulting from spurious variable valuations (unreachable states can be visited during the analysis). Eager exploration algorithms, implemented through sequentialization, have led to the implementation of mature symbolic bug-finding model-checking tools for SC concurrent C programs (e.g., Smack [10], MU-CSeq [21]).

As first contribution, we extend the API of the SMA from [22] to achieve a deeper decoupling of the program computation and communication aspects thus making it more suitable for general implementations. By the SMA, we see a program as the composition (synchronized over the SMA API) of a thread control-flow system and an SMA system. We then identify the semantic notions of *thread-wise equivalence* and *thread-asynchronous closure* of transition systems as general properties that allow us to state correctness of a class of methods. Namely, we get that reachability is preserved if we replace in the composed system its thread control-flow part for a thread-wise equivalent one (assuming the SMA part is thread-asynchronous) or the SMA part for its thread-asynchronous closure. This has two important consequences. First, we can extend existing concurrent verification algorithms that do not reorder statements within each thread (such as the eager ones) to different memory models *simply by implementing* the corresponding SMAs. Second, we get a degree of freedom in designing concurrent verification algorithms, since in executions exploration we can rearrange the order of the statements from different threads. This is implicitly exploited by some algorithms from the literature (e.g., [17,15,21]) which can be recast in the setting we propose here and thus be extended to WMMs.

As second contribution, we instantiate our general approach to achieve an efficient BMC-based bug-finding tool. We give efficient SMA-implementations for SC, total store ordering (TSO), and partial store ordering (PSO) that are based on the idea of *individual memory-location unwindings*, where for each variable we keep the (temporally ordered) sequence of all its writes occurring in a computation. We then show through experiments that our prototype tool compares well with existing tools.

Due to the lack of space, some details are reported in the Appendix.

2 Weak Memory Models

A *shared memory* is a sequence of memory locations of fixed size. The content of each location can be read or written using an explicit memory operation. The semantics of read and write operations depend upon the adopted memory model. Besides SC, we also consider TSO and PSO, which are implemented in modern computer architectures.

Sequential consistency (SC). SC is the “standard model”, where a write into the shared memory is performed directly on the memory location. This has the effect that the newly written value is instantaneously visible to all the other threads.

Total store ordering (TSO). The behaviour of the TSO memory model can be described using a simplified architecture with explicit store buffers [18]. Each thread t is equipped with a local *store buffer* that is used to cache the write operations performed by t according to a FIFO policy. Updates to the shared memory occur nondeterministically along the computation, by selecting a thread, removing the oldest write operation from its store buffer, and then updating the shared memory valuation accordingly. Before updating, the effect of a cached write is visible only to the thread that has performed it. A read by t of a variable y retrieves the value from the shared memory unless there is a cached write to y pending in its store buffer; in that case, the value of the *most recent* write in t 's store buffer is returned. A thread can also execute a *fence*-operation to block its execution until its store buffer has been emptied.

Partial store ordering (PSO). The semantics of PSO is the same as for TSO except that each thread is endowed with a store buffer for each shared memory location.

3 Multi-Threaded Programms over Shared Memory Abstractions

We consider multi-threaded programs with a standard C-like syntax including pointer arithmetics and dynamic memory allocation. We further consider POSIX-like threads with dynamic thread creation, thread join, and mutex locking and unlocking operations for thread synchronization, but no thread communication primitives: threads communicate only via the shared memory. We also assume a `fence`-statement that commits all pending write operations of a thread into the shared memory; for TSO and PSO this means it flushes all store buffers of a thread.

Shared memory abstractions. The semantics of multi-threaded programs ultimately depends on the underlying memory model. In order to combine existing concurrent verification techniques with different memory models we define a “concurrency interface” or *shared memory abstraction (SMA)* that abstracts away the shared memory operations in the syntax of multi-threaded programs. The intended meaning of the SMA's functions is standard; note that most functions carry the calling thread τ as an extra argument to allow the SMA to update its internal state. In detail, the SMA API is:

- `init()` initializes the SMA and the shared variables; this must be the first statement in the program;
- `terminate(τ)` ends the execution of τ ; each thread must explicitly call it;
- `error(τ)` flags that an assertion has failed in τ ; the entire computation ends in an *error state*;

- $\text{address}(v, t)$ returns the memory address of the shared variable v ;
- $\text{malloc}(n, t)$ allocates a continuous block of n memory locations and returns the base address of the block;
- $\text{read}(v, t)$ (resp. $\text{ind_read}(a, t)$) returns the valuation of the shared variable v (resp. memory location with address a) as seen by t ;
- $\text{write}(v, \text{val}, t)$ (resp. $\text{ind_write}(a, \text{val}, t)$) sets the valuation of the shared variable v (resp. memory location with address a) to the value val ;
- $\text{fence}(t)$ commits all pending write operations of t into the shared memory;
- $\text{lock}(m, t)$ and $\text{unlock}(m, t)$ are the standard thread synchronization primitives that acquire and release a mutex m for t ; if m is currently acquired, the lock operation is blocking for t , i.e., t is suspended until m is released and then acquired;
- $\text{create}(f, t)$ spawns a new thread that starts from function f , and returns a fresh thread identifier for this thread;
- $\text{join}(t', t)$ pauses the execution of t until t' has terminated its execution.

Multi-threaded programs as composition of transition systems. The formal semantics of multi-threaded programs is given by a transition system that captures the program computations by interleaving the computations of each thread. We exploit the separation between the control flow and the shared memory aspects introduced with the notion of SMA, and give the semantics of a multi-threaded program as the composition $\mathcal{C}|\mathcal{M}$ of the *control-flow transition system* \mathcal{C} that captures the control flow of all threads and the *SMA transition system* \mathcal{M} that implements the behaviours of the SMA. This allows us to keep the semantics of the sequential part and re-interpret it in different ways with different WMMs; it also aligns nicely with different SMA implementations.

The two transition systems are synchronized over the alphabet Σ_{SMA} which contains the calls to the SMA API functions that do not return values, and the calls augmented with a parameter denoting the returned value for the others. For example, $\text{read}(3, v, t)$ is the letter corresponding to a call $\text{read}(v, t)$ that returns value 3.

Control-flow transition system. The states of the control-flow transition system \mathcal{C} are the tuples of thread configurations. A thread configuration consists of a program counter, an evaluation of the thread-local variables and a call stack. \mathcal{C} has a unique initial state that corresponds to the empty configuration (i.e., no threads are active in the beginning).

The transitions correspond to the execution of any of the statements. Those corresponding to invocations of API functions of SMA are labeled with the corresponding letter from Σ_{SMA} . In particular, transitions from the initial state are labeled with $\text{init}()$ and enter a state with the starting configuration of the main thread. No other transitions are labeled with $\text{init}()$. Transitions corresponding to SMA functions that return a value are handled as thread local assignments with the returned values. On a thread creation the tuple of thread configurations is augmented with the starting configuration of the newly created thread. Similarly, the effect of a transition on $\text{terminate}(t)$ is to delete the configuration of the terminated thread and that of a transition on $\text{error}(t)$ is to enter an *error state*. Both these kinds of transition disallow further transitions corresponding to statements from thread t . The remaining transitions labeled with Σ_{SMA} letters just update the program counter. Transitions corresponding to all other (i.e., sequential) statements are labeled with the empty word ε and update the configuration of the issuing thread as usual.

Shared memory abstraction transition system. With \mathcal{M}_{sc} , \mathcal{M}_{tso} and \mathcal{M}_{pso} we denote the *canonical* SMA transitions systems capturing respectively the semantics of SC, TSO and PSO memory models as described in Section 2. We observe that each of such systems has an initial state and a state for each possible configuration of the corresponding memory model. In addition, the states of \mathcal{M}_{tso} and \mathcal{M}_{pso} account also for the content of the thread store buffers.

Transitions update the memory configurations to capture the memory model’s intended meaning. In particular, from the initial state there are only outgoing transitions labeled with `init()` that take to any state with: just one thread (which must be active), any number of shared locations (which must all have the value of zero), and any number of mutexes (which must all be unlocked). No other transition have this label. \mathcal{M}_{sc} has no `fence`-transitions. \mathcal{M}_{tso} and \mathcal{M}_{pso} have instead such transitions on calls to `fence` by t and also ε -transitions for store buffer updates. Further, in a transition on `terminate(t)`, the three transition systems enter a state where the status of t is terminated. Similarly, on `error(t)`, they enter an *error state*. From any of these two kinds of states no other transitions corresponding to invocations of API functions from t are allowed. The final states are error states and all states where all threads are terminated.

4 Verification with thread-asynchronous SMAs

Splitting the design of a verification tool into an SMA implementation and a search algorithm for program execution exploration gives a convenient way to extend it to other memory models: one can just replace the SMA implementation. However, obtaining scalable tools would still be an issue. In fact, for correctness, a direct implementation of memory models would require to invoke memory operations as they occur in a run. This may result into a bottleneck for summary based analysis (e.g., BDD-based model checking) due to the state-space explosion caused by the cross product of thread-local states, as well as for bounded model checking where the code of all threads must be included at each possible context-switch point, thus leading to large SAT/SMT formulas.

We thus propose a general framework where we assume the SMA implementation to be *thread-asynchronous*, i.e., insensitive to how the threads are interleaved. This allows us to freely transform the threads as long as we stay within the class of *thread-wise equivalent* programs, i.e., programs where the *intra-thread ordering* of the statements remains the same. Transformations into thread-wise equivalent programs has been already exploited in successful approaches from the literature where program executions are rearranged such that each thread is simulated in turn to completion [15,17,21].

For a thread t , we denote with Σ_{SMA}^t the maximal subset of Σ_{SMA} containing only letters that are issued by t . Clearly, for threads t and t' with $t \neq t'$, Σ_{SMA}^t and $\Sigma_{SMA}^{t'}$ are disjoint. For a thread t and a word α over Σ_{SMA} , let $\alpha|_t$ be the projection of α onto Σ_{SMA}^t , i.e., the word obtained from α by deleting all the letters that do not belong to Σ_{SMA}^t . If t_1, \dots, t_h are all the threads that issue at least a letter in α , we define $\pi(\alpha)$ as the map $\pi(\alpha)(t_i) = \alpha|_{t_i}$ for $i \in [1, h]$.

A language L over Σ_{SMA} is *thread-asynchronous* if for each $\alpha \in L$ and for each α' starting with `init()` s.t. $\pi(\alpha) = \pi(\alpha')$, also $\alpha' \in L$. The *thread-asynchronous closure* of L , denoted by $L^\#$, is the smallest thread-asynchronous language such that $L \subseteq L^\#$.

Let \mathcal{A}_1 and \mathcal{A}_2 be two transition systems over the alphabet Σ_{SMA} . We say that \mathcal{A}_1 and \mathcal{A}_2 are *thread-wise equivalent* if for each word α accepted by one of them there is a word α' that is accepted by the other one such that $\pi(\alpha) = \pi(\alpha')$.

A standard analysis for a multi-threaded program is to search for the reachability of an *error*, often denoted by an error label or a `false`-assertion in the program. In our setting, an error is captured by a transition over label `error(t)`. Since program executions are captured by accepting runs of corresponding transition systems, a program error is reachable if and only if a word containing label `error(t)` is accepted. We say that an error is reachable in two transition systems \mathcal{A}_1 and \mathcal{A}_2 , if there are words $\alpha_i \in L(\mathcal{A}_i)$, with $i = 1, 2$, that contain a same label `error(t)` and such that $\pi(\alpha_1) = \pi(\alpha_2)$.

We conclude this section with two theorems stating sufficient conditions under which the reachability of error states is preserved. The first theorem states that if the SMA system is thread-asynchronous, by transforming a program P_1 into a program P_2 such that the corresponding control-flow transition systems are thread-wise equivalent, an error is reachable in P_1 if and only if it is reachable in P_2 . Intuitively, this theorem holds since the fact that the SMA transition system is thread-asynchronous ensures that the interaction of each thread with the SMA is independent of how threads are interleaved; in particular, by fixing a run ρ , the values returned by the read operations performed by a thread are ensured to be the same in all the possible interleavings of the projections of ρ onto each thread. Since we assume that the sequences of SMA operations issued along the runs of P_1 and P_2 may differ only as caused by different interleavings of the threads, we get that reachability is preserved.

Theorem 1. *Let \mathcal{C}_i be a control-flow transition system for $i = 1, 2$ and \mathcal{M} be an SMA transition system. If \mathcal{C}_1 and \mathcal{C}_2 are thread-wise equivalent, and \mathcal{M} is thread-asynchronous, then an error is reachable in $\mathcal{C}_1|\mathcal{M}$ iff it is reachable in $\mathcal{C}_2|\mathcal{M}$.*

Theorem 1 states a crucial property for our approach: we can implement a thread-asynchronous SMA, and combine it with any transformation of the program that rearranges the interleaving among threads and still get a correct verification approach.

The second theorem shows that we can replace an SMA \mathcal{M}_1 with another SMA \mathcal{M}_2 that captures its thread-asynchronous closure, and still preserve reachability of errors. The interesting case of the proof is when a sequence α is accepted by \mathcal{M}_2 but not by \mathcal{M}_1 . In this case, since the returned values are visible in Σ_{SMA} letters and there must be a sequence α' that is accepted by \mathcal{M}_1 such that $\pi(\alpha) = \pi(\alpha')$, we get that the sequence of local states that are visited by any thread of any program P are the same for both sequences α and α' . Therefore, the following theorem holds.

Theorem 2. *Let \mathcal{C} be a control-flow transition system and \mathcal{M}_i be an SMA transition system for $i = 1, 2$. If $L(\mathcal{M}_2) = (L(\mathcal{M}_1))^\#$, then an error is reachable in $\mathcal{C}|\mathcal{M}_1$ iff it is reachable in $\mathcal{C}|\mathcal{M}_2$.*

By the above theorems, we can show the correctness of WMM extensions of correct verification methods that transform programs by keeping the ordering of the operations within each thread, such as the methods from [17,15,16,21]. In fact, we just need to provide an SMA that captures the thread-asynchronous closure of the memory model.

5 Individual Memory-Location Unwindings

We now discuss an implementation of thread-asynchronous SMAs for SC, TSO and PSO. The key notion is the *individual memory-location unwinding* (IMU), a set containing exactly one sequence of writes for each shared memory location (*location unwinding*, LU for short) and such that the unique timestamps associated to each write determine a total order among all the writes of all the LUs (where each timestamp denotes the time of occurrence of a write according to a discrete-time global clock).

Precisely, an LU for a memory location v , denoted by v -LU, is a sequence of triples (t, val, d) where t and val denote the thread identifier and the value of the write and $d > 0$ is the associated timestamp. If Var is the set of location names and μ_v a v -LU for each $v \in Var$, an IMU is a set $\{\mu_v \mid v \in Var\}$ such that: a) the tuples in each LU are ordered by increasing timestamps, and b) for each pair of different location names $v_1, v_2 \in Var$ and for each (t_i, val_i, d_i) in μ_{v_i} with $i = 1, 2$, then also $d_1 \neq d_2$ (thus timestamps define a total order among all the writes in the IMU).

IMU-based SMA for SC. A transition system \mathcal{M}_{sc}^{imu} for an IMU-based implementation of SMA first guesses an IMU on the $init()$ -transition and then consistently executes the operations. Namely, it keeps for each thread the current timestamp (i.e., the timestamp of the last executed SMA operation) and for any input sequence α , it ensures that:

- on $write(v, val, t)$ (resp. $ind_write(a, val, t)$), the next write in the v -LU (resp. the LU identified by the address a) for thread t matches the value val ; the current timestamp of t is updated to the timestamp of the matched write in the next state;
- on $read(val, v, t)$ (resp. $ind_read(val, a, t)$), there must be in the v -LU (resp. the LU identified by the address a) a write with timestamp d that assigns value val to v such that either d is the timestamp of the most recent (before t 's current timestamp) write to v or d is between t 's current timestamp and the timestamp of t 's next write; in the latter case t 's current timestamp is updated to d in the next state;
- for each thread, the writes are matched by increasing timestamps.

In order to accept α , $create(t, f, t')$ must occur in α for each thread t with writes guessed in the IMU and the writes in the IMU should be mapped 1-to-1 to the writes in α .

The transition system \mathcal{M}_{sc}^{imu} is thread-wise equivalent to \mathcal{M}_{sc} , and additionally, it can execute all computations of \mathcal{M}_{sc} by advancing each involved thread in any order. Moreover, due to the fact that all writes are guessed in advance, the ordering in which we interleave the threads is irrelevant. We thus get the following lemma.

Lemma 1. $L(\mathcal{M}_{sc}^{imu}) = (L(\mathcal{M}_{sc}))^\#$.

IMU-based SMA for TSO and PSO. We augment the IMU by adding a second timestamp for each write. In particular, we now make a distinction between the time a write occurs (*occurrence timestamp*) and the time the shared memory is updated with an occurred write (*update timestamp*). For correctness, we also impose on the IMU that for each write the occurrence timestamp should not be greater than the update timestamp.

For TSO, in order to ensure the FIFO policy of the store buffers, we additionally require that for each thread the occurrence and the update timestamps must both order all the writes according to the program order. For PSO, instead it is sufficient to require this only for the writes of a same location.

We will denote with \mathcal{M}_{tso}^{imu} and \mathcal{M}_{pso}^{imu} the IMU-based SMA transition systems corresponding to the TSO and PSO memory models, respectively. \mathcal{M}_{tso}^{imu} can be obtained from \mathcal{M}_{sc}^{imu} with a few changes: on the `init()`-transition we now guess the IMU with occurrence and update timestamps as observed above; in a read of location v by a thread t the position of the matching write is the last occurred write still in the store buffer of t (i.e., current timestamp of t is between the occurrence timestamp and the update timestamp of the last write of v by t), if any, and the last updated write of v , otherwise (this case works as the read in \mathcal{M}_{sc}^{imu}); the current timestamp of a thread t is also updated to the occurrence timestamp of a write when this is executed; a `fence(t)`-transition updates the current timestamp to the largest update timestamp of the already occurred writes performed by t . Obtaining \mathcal{M}_{pso}^{imu} from \mathcal{M}_{tso}^{imu} is very simple and the only difference is hidden in the properties that are required on the guessed IMU as observed above.

By the above observations we can derive that \mathcal{M}_{pso}^{imu} and \mathcal{M}_{tso}^{imu} capture the semantics of the corresponding memory models. Moreover, since all the writes are guessed in advance, the ordering in which we interleave the threads is irrelevant. Thus, we get:

Lemma 2. For $m \in \{tso, pso\}$, $L(\mathcal{M}_m^{imu}) = (L(\mathcal{M}_m))^{\#}$.

Verification by IMU. By composing the control-flow system obtained by the program transformation from [21] along with \mathcal{M}_{sc}^{imu} , \mathcal{M}_{tso}^{imu} and \mathcal{M}_{pso}^{imu} , we get new verification methods for concurrent programs under SC, TSO and PSO semantics. The correctness of such methods is a consequence of the above lemmas, and Theorems 1 and 2.

6 IMU-based SMA implementations

In this section, we discuss concrete C-implementations of the SMA API from Section 3 according to the semantics captured by \mathcal{M}_{sc}^{imu} , \mathcal{M}_{tso}^{imu} and \mathcal{M}_{pso}^{imu} . We will give some details of the implemented code. Note that our code is optimized for an efficient analysis using BMC tools but implementations for other backends are possible.

IMU implementation for SC. The implementation is parameterized over several constants. N and U denote the number of *locations with names* (i.e., shared scalar variables) and *locations without names* (i.e., heap locations accessed only through memory addresses), respectively. \bar{W} denotes the maximum number of write operations for each of these $V=N+U$ tracked memory locations, M and T denote the maximum number of dynamic memory allocations and thread creations, respectively, that may happen during any execution of the input program.

Data structures and invariants. We use several scalar variables and arrays to maintain the LUs and support the implementation of the SMA operations. We sketch below the main ones that are relevant to the read and write operations; others are used to model thread creation, join, and termination, and the dynamic memory allocation. All are declared global such that they are visible and can be modified in all the functions. For simplicity, we assume that all data is represented by unsigned integers.

The triples (t, val, d) of the LUs are maintained by three different arrays `thread`, `value` and `tstamp`. For every location $v \in [0, V-1]$ and $i \in [0, \bar{W}-1]$, the triple at position i in the v -LU is stored in `thread[v][i]`, `value[v][i]` and `tstamp[v][i]`. We

```

int read(uint v, uint t){
    if(is_terminated(t)) return 0;
    uint jump = Jump(t,v);
    return (value[v][jump]);
}

void write(uint v, int val, uint t){
    if(is_terminated(t)) return;
    uint i, jump;
    i = th_pos[v][t];
    jump=th_next_write[v][i][t];
    assume( (jump<=last_write[v])
        && (value[v][jump] == val)
        && (tstamp[v][jump] > cur_tstamp[t])
    );
    th_pos[v][t]=jump;
    cur_tstamp[t]=tstamp[v][jump];
}

uint Jump(uint t, uint v){
    uint jump=*;
    uint j=th_pos[v][t];

    ts_jump = tstamp[v][jump];
    assume( (jump <= last_write[v])
        && (jump < th_next_write[v][t][j])
        && (tstamp[v][jump+1]>cur_tstamp[t])
    );

    cur_tstamp[t] =
        (ts_jump > cur_tstamp[t]) ?
            ts_jump : cur_tstamp[t];
    return jump;
}

```

Fig. 1. Read, write, and jump functions.

link the writes of a same thread in each LU by an additional array `th_next_write`. All these arrays are nondeterministically assigned in the function `init` and never changed in the program execution. `init` also ensures that:

- timestamps are assigned in increasing order for each LU;
- no two writes in the IMU are assigned with the same timestamp;
- for every location $v \in [0, V-1]$, position $i \in [0, W-1]$ and thread identifier $t \in [0, T-1]$, `th_next_write[v][i][t]` is the first position in the v -LU after i that corresponds to a write by t , if any; otherwise, it is set to W , denoting that no further writes of v by t are expected.

To keep track of the execution of each thread in the IMU, we use the arrays `th_pos`, `last_write` and `cur_tstamp`, and maintain the following invariants for every location $v \in [0, V-1]$ and thread identifier $t \in [0, T-1]$:

- `th_pos[v][t]` stores the current position of thread t in the v -LU;
- `last_write[v]` stores the position $i \in [0, W-1]$ of the last executed write operation of location v in the v -LU;
- `cur_tstamp[t]` stores the current timestamp of thread t during its simulation.

Verification stubs. We only discuss here the implementation of the API functions `read` and `write`, which is given in Fig. 1. Both functions first check whether the execution of the simulated thread has been stopped, and return immediately if this is the case.

For a read operation of thread t from location v , we first jump forward into v -LU by invoking the auxiliary function `Jump` and then return the value of v at this new position of v -LU. `Jump` (cf. Fig. 1) works as follows. If the timestamp of the selected write is past the current thread timestamp, the latter is updated to this value, acknowledging the fact that the corresponding write into the shared memory has occurred. The value of `jump` is selected nondeterministically within a range of proper values. Namely, `jump` should not pass the last legal write position for v and must be strictly less than the position of the next write of v by the same thread t (that has not occurred yet). Further, we require that the timestamp at position `jump+1` is greater than the current timestamp of t , as we must point to a write of v that is not superseded by already occurred writes.

With the stated invariants we get that `Jump` identifies a position i in the v -LU that is correct w.r.t. the v -LU (in the sense that it is not jumping over the next write of v by

```

int read(uint v,uint t){
    if(is_terminated(t)) return 0;

    uint ts_jump, i;
    i = th_pos[v][t];
    uint nxt_write=th_next_write[v][i][t];
    uint fst_write=th_next_write[v][0][t];
    assume (
        (ts_jump >= cur_tstamp[t]) &&
        (ts_jump < btstamp[v][nxt_write])
    );
    cur_tstamp[t]=ts_jump;
    if( fst_write <= i &&
        tstamp[v][i] > cur_tstamp[t]
    ) return value[v][i];
    return Read_SC(v,t);
}

void fence(uint t){
    if(ts_lastW[t]>cur_tstamp[t])
        cur_tstamp[t] = ts_lastW[t];
}

void write(uint v,int val,uint t){
    if(is_terminated(t)) return;
    i = th_pos[v][t];
    jump=th_next_write[v][i][t];
    th_pos[v][t]=jump;
    assume (
        btstamp[v][jump] > cur_tstamp[t]
        && value[v][jump] == val
        && tstamp[v][jump] > ts_lastW[t]
    );
    ts_lastW[t] = tstamp[v][jump];
    cur_tstamp[t] = btstamp[v][jump];
}

```

Fig. 2. Functions read, fence and write for TSO.

t). However, note that the corresponding timestamp could be still larger than the next write by t (for a different location) but we will catch this while executing the next write of t , when the current timestamp of t will be larger than the one of that write.

In a write operation, we first move forward to the position of the next write by t in the v -LU and block the execution if the value to be written differs from that stored in the v -LU. We also check that the timestamp associated with the new v -LU position for t is greater than the current timestamp of t ; if this is not the case, we are then in the error case generated by a wrong update of the thread timestamp in a read, and thus the execution is aborted. If all checks are passed, we update the current position of thread t in the v -LU and the current timestamp accordingly, thus maintaining the invariants.

IMU implementation for TSO. We give this implementation incrementally on that given for SC; the code of the functions read, fence and write is illustrated in Fig. 2. We use: $tstamp[v][i]$ to store the update timestamp and $btstamp[v][i]$ to store the occurrence timestamp of the write at position i in the v -LU; $ts_lastW[t]$ to store the update timestamp of the write by thread t that occurred last.

For *init*, we guess the initial values for $btstamp[v][i]$ and then impose that $btstamp[v][i] \leq tstamp[v][i]$ must hold (i.e., the update of the shared memory according to an occurred write may be delayed w.r.t. its occurrence time). Note that here we slightly diverge from the transition system \mathcal{M}_{tso}^{imu} described in Section 5. In fact, since we do not require any other condition on the guessed update timestamps, we can carry over an IMU with timestamps that may violate the FIFO policy on the store buffers. We fix this by checking the proper ordering on matching the writes (see below).

The *fence*-operation flushes the store buffer of the executing thread. We thus need to synchronize the current thread timestamp with its last update timestamp, i.e., if $ts_lastW[t]$ is larger than the timestamp of the last occurred write by t , we set $ts_lastW[t]$ to $cur_tstamp[t]$. Note that if this is not the case then the local store buffer of t is certainly empty, since $btstamp[v][i] \leq tstamp[v][i]$.

The *read*-function first increases nondeterministically the current timestamp of thread t such that it remains smaller than the occurrence timestamp of the next write of v by t . Now, if at least a write of location v by t has occurred and the last write of v by t is still in the thread buffer, then we return the value of this write. Otherwise, a read

from the shared memory is performed by invoking the auxiliary function `Read_SC` that is exactly the function `read` from Fig. 1. Note that the update of the current thread timestamp by `read` can cause this value to be larger than the update timestamp of the last write, which is correct. To avoid that we wrongly move the time back, in `fence` we make the assignment only when this is not the case.

The `write`-function first updates the current position in the v-LU of thread `t` to the next write provided that: the time of occurrence of this write is larger than the current thread timestamp, the value of the write matches the guessed value for it and the update timestamp of the next write is larger than that of the last occurred write (the last one ensures that the thread store buffers are emptied according to a FIFO policy). Note that, in the case of a wrong guess of the update timestamps in `init`, this condition would not hold and thus the execution would abort. Before returning, the update timestamp of the last write and the current timestamp of thread `t` are modified consistently.

IMU implementation for PSO. We just need to slightly modify the implementation for TSO as follows. We use a new array `max_tsW` instead of `ts_lastW` to keep for each thread `t` the maximum update timestamp among all the occurred writes of `t`. Thus, we replace in `write` the update of `ts_lastW` with the assignment of `max_tsW[t]` with `(tstamp[v][jump] > max_tsW[t]) ? tstamp[v][jump] : max_tsW[t]`.

We further modify function `write` by removing from the `assume`-statement the conjunct `tstamp[v][jump] > ts_lastW[t]` (see Fig. 2). We recall that this conjunct was required in the TSO implementation to ensure the store-buffer FIFO policy for each thread; in PSO, we only need to require this within each LU.

7 Experimental Evaluation

We implemented the approach of Section 6 in the IMU-CSeq tool¹ that analyzes C programs over the pthreads API. It uses modules from MU-CSeq [13,21] to transform the original multi-threaded program into a sequential one (sequentialization), then links this against an IMU-based SMA implementation, and finally verifies the resulting program with a BMC tool for sequential programs, in particular CBMC (v5.3). By varying the SMA implementation we thus obtain a tool for verifying multi-threaded programs under SC, TSO, and PSO, respectively. A hybrid tool combining IMU-CSeq and MU-CSeq [23] has won the gold medal in the Concurrency-category of the TACAS Software Verification Competition (SV-COMP16) [8]. We recall that MU-CSeq is based on the notion of *memory unwinding* where all the program writes are kept in a single sequence.

The experiments below were run on a dedicated machine with a Xeon E5-2650 v2 with 2.60 GHz and 132GB RAM, running Linux 4.2.0-22-generic. We set a 15GB memory limit and a 900s time limit. For each tool and benchmark, we set the parameters to the minimum value to expose the error. Verification time is reported in seconds.

SC benchmarks. We first evaluate IMU-CSeq on the Concurrency-benchmarks SV-COMP16 under SC semantics. These cover the core features of the C programming language and the basic concurrency mechanisms. Since we use a BMC tool as a back-end, we can only evaluate IMU-CSeq only on files that have a reachable error location. We used the files from the sub-categories shown in Table 1; each row shows the corresponding number of files and lines of code.

¹ <http://users.ecs.soton.ac.uk/gp4/cseq/files/IMU-2017.zip>

Table 1. Performance comparison among different tools for SC semantics on unsafe instances from the SV-COMP16 *Concurrency* category.

sub-category	files	l.o.c.	CBMC svc16			CIVL svc16			Lazy-CSeq svc16			MU-CSeq svc15			IMU-CSeq		
			pass	fail	time	pass	fail	time	pass	fail	time	pass	fail	time	pass	fail	time
pthread	15	2301	14	1	84.23	15	0	33.31	15	0	48.58	15	0	5.42	15	0	4.88
pthread-atomic	2	156	2	0	0.59	2	0	17.5	2	0	1.39	2	0	1.4	2	0	3.15
pthread-ext	8	616	7	1	154	8	0	13.12	8	0	11.23	8	0	5.45	8	0	4.88
pthread-lit	2	73	2	0	0.3	2	0	10.33	2	0	0.56	2	0	2.55	2	0	0.88
ldv-races	8	616	3	5	66.96	3	0	14.5	8	0	1.73	-	-	-	8	0	1.61

Table 1 shows the results for the SV-COMP16 versions of CBMC [5], CIVL [26], Lazy-CSeq [13,14], the SV-COMP15 version of MU-CSeq [21],² and of IMU-CSeq on these benchmarks. We indicate with *pass* the number of correctly found bugs, with *fail* the number of unsuccessful analyses including tool crashes, memory limit hits, and timeouts, and with *time* the average time in seconds to find the bug. The results clearly show that our approach is competitive with existing tools; in particular, the IMU-based SMA-implementation improves over MU-CSeq.

WMM benchmarks. We then compared IMU-CSeq against three tools with built-in support for WMM, LazySMA [22], CBMC [12], and Nidhugg [1] a bug-finding tool that combines stateless model checking with dynamic partial order reduction.

Simple benchmarks. Table 2 shows the results over a set of (relatively simple) benchmarks collected from the CBMC, Poet, and Nidhugg tools, and the SV-COMP benchmark suite. The unwind parameter was used by all the three tools considered in the comparison, while \bar{w} , \bar{u} , and \bar{m} are used only by IMU-CSeq, as detailed in Section 6. The parameter bitwidth gives the size of integers (in bits) used in the sequential analysis.

The first block contains results for some classical mutual exclusions algorithms. The implementations are correct under SC but not under TSO and PSO (as indicated by an entry in the column ‘bug?’). All tools find the errors, but because of the problems’ small size, Nidhugg outperforms IMU-CSeq, LazySMA and CBMC on these programs.

The second block contains safe and unsafe versions of one of the fibonacci-benchmarks, where two worker threads concurrently increase two shared counters, and a main thread checks whether any of the counters can reach a defined value. A full exploration of the thread interleavings is required to identify the error (or show its absence) in this program and techniques such as partial-order reduction do not apply. Here, IMU-CSeq has substantially a slight edge over both CBMC and LazySMA, while Nidhugg is substantially slower than the other three.

The next block contains benchmarks derived from industrial code. `pgsql` is a well-known SQL bug [4]; it is correct under SC and TSO but not under PSO. `parker` models a semaphore-like synchronization class that breaks under TSO [1], and `stack_unsafe` which was taken from SV-COMP [8]. All tools report the expected results; the performance differences between Nidhugg and CBMC are small, while IMU-CSeq’s and LazySMA’s performance could be improved (each implementation currently parses and unparses each file nearly 20 times).

The fourth block shows the average results for 5803 WMM litmus tests with 297K lines of code. For TSO, both our tool, LazySMA and CBMC successfully identified the

² Note that the SV-COMP16 version of MU-CSeq is a hybrid tool that already uses IMU for the shown sub-categories. We thus use the SV-COMP15 version here.

Table 2. Analysis runtime under TSO/PSO

	l.o.c.	parameters					TSO runtime (s)				PSO runtime (s)							
		unwind	W	U	M	bitwidth	bug?	files	IMU-CSeq	LazySMA	CBMC	NIDHUGG	bug?	files	IMU-CSeq	LazySMA	CBMC	NIDHUGG
dekker	52	1	2	0	0	5	•	1	0.76	0.77	0.29	0.04	•	1	0.76	0.75	0.25	0.05
lampport	78	1	2	0	0	5	•	1	0.97	0.88	0.31	0.05	•	1	0.97	0.88	0.29	0.05
peterson	40	1	3	0	0	5	•	1	0.67	0.66	0.26	0.04	•	1	0.68	0.65	0.25	0.04
szymanski	57	1	3	0	0	5	•	1	0.84	0.81	0.34	0.07	•	1	0.84	0.80	0.32	0.04
fib_longer_unsafe	30	6	7	0	0	10	•	1	2.10	6.47	8.19	94.84	•	1	2.50	6.51	1.69	135.45
fib_longer_safe	30	6	7	0	0	10	•	1	4.75	9.78	22.5	t.o.	•	1	3.90	8.82	31.8	t.o.
pgsql	47	1	2	0	0	5	•	1	1.92	2.03	0.03	0.07	•	1	0.69	0.65	0.22	0.04
parker	110	1	2	0	0	5	•	1	1.22	1.68	0.31	0.05	•	1	1.21	2.19	0.28	0.05
stack_unsafe	110	2	2	1	2	5	•	1	1.46	1.50	0.41	0.05	•	1	1.44	1.49	0.35	0.05
litmus_safe	-	1	6	1	0	10	•	5526	1.20	1.26	0.17	2.35	•	4835	1.06	1.22	0.15	6.65
litmus_unsafe	-	1	6	1	0	10	•	277	1.67	1.27	0.16	3.86	•	968	1.28	1.26	0.12	1.58
safestack	83	3	10	7	2	5	•	1	207.4	1474.6	t.o.	t.o.	•	1	1013.3	1207.3	t.o.	t.o.

277 test cases containing a reachable error, while Nidhugg failed to find one of them. For PSO, CBMC claims that there are 971 unsafe instances while Nidhugg, LazySMA and IMU-CSeq find only 968 unsafe ones (we suspect an error in CBMC). Here, symbolic methods are faster, and Nidhugg has two timeouts.

Complex benchmark. Safestack [11] is a lock-free stack implementation designed for WMM. It is written in C++ but we manually translated it into C, providing simulation functions for the C++11 atomic functions, and analyzed this version. It contains a rare bug that is hard to find with automatic bug-finding techniques already under SC (including random testing, Nidhugg, CIVL [26], and other approaches based on BMC) [20]. The only tool we are aware of that can automatically find a genuine counter-example is Lazy-CSeq [13], which requires a minimum of 3 loop unwindings and 4 rounds of computation and more than 7 hours to expose a bug. As shown in the last block of Table 2, both Nidhugg and CBMC failed to find the bug, while IMU-CSeq required approx. 3.5 minutes and 1.5GB of memory to find it under TSO, and approx. 17 minutes and 1.8GB of memory under PSO, and is faster than LazySMA in finding these bugs.

8 Related work, Conclusions and Future Work

Related Work. The BMC approach from [5] allows to handle different memory models by adding a conjunct to the formula. The verification algorithm in [3] works on a generic relaxed memory model that can be refined into actual memory models by adding constraints. Our work differs from these both in the scope and the techniques. In particular, we work at the level of source code with code-to-code transformations and give a general approach that allows to combine different verification algorithms with different implementations of memory models, not just a specific algorithm. The development of the two parts can be done independently as long as Theorems 1 and 2 hold.

Another important aspect of our approach is to identify a class of implementations of memory models that allows for a full rearrangement of the thread interleavings in the analysis. As already observed, this is a feature that has been already exploited in verifying concurrent programs [17,21] also with WMM semantics [7].

The axiomatic framework from [6] is introduced to capture the semantics of memory models. Our framework instead aims at a scalable verification approach that encap-

ulates all differences between the models within the SMA implementation such that the designs of the verification algorithm and of the memory model simulation can be developed independently.

The notion of IMU exactly captures the *coherence* relation that is often used in the description of memory models (see [6,2]). In our setting, we achieve the reordering of the statements that are observed in the relaxed memory models by guessing the timestamps and then checking their consistency with the expected behaviours. This is done while executing the statements with appropriate `assume`-statements.

The reachability analysis used in our algorithm [21] is bounded on the number of writes which is orthogonal to bounding the number of context-switches [19].

Conclusions. We have described and evaluated a new verification approach for concurrent programs over different memory models. Our main design goal was to break the coupling between computation (i.e., individual threads) and communication (i.e., shared memory) concerns of multi-threaded programs, without losing the efficiency of existing approaches. We have introduced shared memory abstractions, which capture the standard concurrency operations in multi-threaded programs. We have then shown that reachability is preserved if we exchange a program by a thread-wise equivalent one (assuming the SMA is thread-asynchronous) or an SMA for its thread-asynchronous closure. This allows us to generalize existing concurrent verification approaches to different memory models simply by implementing the corresponding different SMAs. We have described efficient SMA implementations for SC, TSO, and PSO based on the idea of individual memory-location unwindings, which have allowed us to instantiate our approach into an efficient eager-sequentialization-based BMC bug-finding tool. Our experiments show that the resulting prototype tool compares well with existing ones.

Future Work. We plan to extend our approach to other memory models such as POWER. POWER relaxes PSO (and thus TSO) in two key aspects (see [2]): (i) the propagation of a write in the shared memory by a thread can be asynchronous, i.e., each thread can see the write at a different time; (ii) the order of execution of the statements of a thread can be rearranged liberally (w.r.t. the program order) provided that the dependency relations such as data-flow, address, control and `isync` are respected. The asynchronous write propagation can be easily captured in the IMU by allowing for each write a different timestamp *per* thread. To capture the dependency relations, a more substantial addition may be required instead. However, on the basis of preliminary empirical experiments, we have evidences that our approach have a potential to scale well to more relaxed memory models. We leave this for future investigations.

References

1. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.F.: Stateless model checking for TSO and PSO. In: TACAS. pp. 353–367 (2015)
2. Abdulla, P.A., Atig, M.F., Bouajjani, A., Ngo, T.P.: Context-bounded analysis for POWER. In: TACAS. pp. 56–74 (2017)
3. Abe, T., Maeda, T.: A general model checking framework for various memory consistency models. In: IEEE PDP. pp. 332–341 (2014)

4. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In: ESOP. pp. 512–532 (2013)
5. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: CAV. pp. 141–157 (2013)
6. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.* 36(2), 7:1–7:74 (2014)
7. Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in TSO analysis. In: CAV. pp. 99–115 (2011)
8. Beyer, D.: Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016). In: TACAS. pp. 887–904 (2016)
9. Bouajjani, A., Calin, G., Derevenetc, E., Meyer, R.: Lazy TSO reachability. In: FASE. pp. 267–282 (2015)
10. Carter, M., He, S., Whitaker, J., Rakamaric, Z., Emmi, M.: SMACK software verification toolchain. In: ICSE. pp. 589–592 (2016)
11. Chen, G., Jin, H., Zou, D., Zhou, B.B., Liang, Z., Zheng, W., Shi, X.: Safestack: Automatically patching stack-based buffer overflow vulnerabilities. *IEEE Trans. Dependable Sec. Comput.* 10(6), 368–379 (2013)
12. Horn, A., Kroening, D.: On partial order semantics for sat/smt-based symbolic encodings of weak memory concurrency. In: FORTE. pp. 19–34 (2015)
13. Inverso, O., Nguyen, T.L., Fischer, B., La Torre, S., Parlato, G.: Lazy-cseq: A context-bounded model checking tool for multi-threaded c-programs. In: ASE. pp. 807–812 (2015)
14. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Bounded model checking of multi-threaded C programs via lazy sequentialization. In: CAV. pp. 585–602 (2014)
15. La Torre, S., Madhusudan, P., Parlato, G.: Model-Checking Parameterized Concurrent Programs Using Linear Interfaces. In: CAV. pp. 629–644 (2010)
16. La Torre, S., Madhusudan, P., Parlato, G.: Sequentializing Parameterized Programs. In: FIT. pp. 34–47 (2012)
17. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design* 35(1), 73–97 (2009)
18. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-tso. In: TPHOLs. pp. 391–407 (2009)
19. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: TACAS. pp. 93–107 (2005)
20. Thomson, P., Donaldson, A.F., Betts, A.: Concurrency testing using schedule bounding: an empirical study. In: PPOPP. pp. 15–28 (2014)
21. Tomasco, E., Inverso, O., Fischer, B., La Torre, S., Parlato, G.: Verifying concurrent programs by memory unwinding. In: TACAS. pp. 551–565 (2015)
22. Tomasco, E., Nguyen, T.L., Inverso, O., Fischer, B., La Torre, S., Parlato, G.: Lazy sequentialization for TSO and PSO via shared memory abstractions. In: FMCAD. pp. 193–200 (2016)
23. Tomasco, E., Nguyen, T.L., Inverso, O., Fischer, B., La Torre, S., Parlato, G.: Mu-cseq 0.4: Individual memory location unwindings - (competition contribution). In: TACAS. pp. 938–941 (2016)
24. Wehrheim, H., Travkin, O.: TSO to SC via symbolic execution. In: HVC. pp. 104–119 (2015)
25. Zhang, N., Kusano, M., Wang, C.: Dynamic partial order reduction for relaxed memory models. In: PLDI. pp. 250–259 (2015)
26. Zheng, M., Rogers, M.S., Luo, Z., Dwyer, M.B., Siegel, S.F.: CIVL: formal verification of parallel programs. In: ASE. pp. 830–835 (2015)

```

P ::= init(); (type f (<dec,>*) {(dec;)* stm}*)*
dec ::= type z | type * p
type ::= bool | int | void
stm ::= seq | conc | {<stm;>*}
seq ::= assume(b) | assert(b) | x = e | f(<e,>*) | return e
      | if(b) then stm else stm | while(b) do stm
conc ::= p = address(y, t) | p = malloc(e, t)
      | x = read(y, t) | x = ind.read(p, t) | write(y, x, t) | ind.write(p, x, t)
      | t = create(f, t) | join(t, t) | terminate(t)
      | fence(t) | lock(m, t) | unlock(m, t)

```

Fig. 3. Syntax of multi-threaded programs.

A Syntax of multi-threaded programs

The syntax of multi-threaded programs is defined by the grammar shown in Fig. 3. Terminal symbols are set in typewriter font. $\langle n \ \mathfrak{t} \rangle^*$ represents a possibly empty list of non-terminals n that are separated by terminals \mathfrak{t} ; x denotes a local variable, y an identifier of a shared variable, p an identifier of a pointer variable, m a mutex identifier, t a thread identifier and f a function name. We assume expressions e to be local variables, pointer value (returned by a read of a pointer variable), and integer constants that can be combined using mathematical operators. Boolean expressions b comprise the constants `true`, `false`, and Boolean variables, and can be combined using standard Boolean operations.

A *multi-threaded* program consists of an `init()` invocation followed by a list of functions. `init()` instantiates a shared memory abstraction that captures a number of *shared* locations. Each function has a list of zero or more typed parameters, and its body has a declaration of *local* variables followed by a statement.

A statement is either a sequential or a concurrent statement, or a sequence of statements enclosed in braces (*compound statement*).

A *sequential statement* can be an `assume`- or `assert`-statement, an assignment, a call to a function that takes multiple parameters (with an implicit call-by-reference parameter passing semantics), a `return`-statement, a conditional statement, or a loop. All variables involved in a sequential statement are local.

A *concurrent statement* involves an interaction with the shared memory abstraction and thus we have a different concurrent statement for each of the functions of the SMA API (other than `init` that is invoked only in the beginning).

We assume that a valid program P satisfies the usual well-formedness and type-correctness conditions. We also assume that P contains a function `main`, which is the starting function of the only thread that exists in the beginning. We call this the *main thread*. We further assume that there are no calls to `main` in P and that no other thread can be created that uses `main` as starting function.

B Transition systems

An alphabet is a set of symbols. For an alphabet Σ , a word over Σ is a sequence of zero or more symbols from Σ . The empty word, denoted by ε , is the word formed of zero symbols. Recall that $w\varepsilon = \varepsilon w = w$ for any word w .

A *transition system* \mathcal{A} is a tuple $(Q, \Sigma, \Delta, Q_0, F)$ where Q is a set of states, Σ is an alphabet, $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is a transition relation, $Q_0 \subseteq Q$ is a set of *initial* states, and $F \subseteq Q$ is a set of final states.

A *run* π of \mathcal{A} is a sequence $q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} q_2 \dots \xrightarrow{\sigma_d} q_d$ where $q_0 \in Q_0$ and $(q_{i-1}, \sigma_i, q_i) \in \Delta$ for each $i \in [1, d]$. Moreover, π is accepting if $q_d \in F$ and $\sigma_1 \dots \sigma_d$ is the corresponding *word*. We denote by $L(\mathcal{A})$ the set of all words that correspond to accepting runs of \mathcal{A} .

Let $\mathcal{A}_i = (Q_i, \Sigma, \Delta_i, Q_{0,i}, F_i)$ be a transition system for $i \in \{1, 2\}$. The composition of \mathcal{A}_1 and \mathcal{A}_2 , denoted $\mathcal{A}_1 | \mathcal{A}_2$, is the standard cross product, i.e., $\mathcal{A}_1 | \mathcal{A}_2$ is the transition system $(Q_1 \times Q_2, \Sigma, \Delta, Q_{0,1} \times Q_{0,2}, F_1 \times F_2)$ where Δ is the minimal set containing all tuples $((q_1, q_2), \sigma, (q'_1, q'_2))$ such that either one of the following cases hold: 1. $\sigma = \varepsilon$, $(q_1, \varepsilon, q'_1) \in \Delta_1$, $q_2 = q'_2$; or, 2. $\sigma = \varepsilon$, $q_1 = q'_1$, $(q_2, \varepsilon, q'_2) \in \Delta_2$; or, 3. $\sigma \neq \varepsilon$, and $(q_i, \sigma, q'_i) \in \Delta_i$ for $i \in \{1, 2\}$.

C Thread-asynchronous SMAs for thread interfaces and memory unwinding

We briefly recall the notions of thread interface [15] and memory unwinding [21], and discuss how to recast some approaches from the literature in our setting by means of the SMAs derived from these notions.

Thread interface. A *thread interface* for a thread t summarizes computations of t across a bounded number of context-switches. Formally, it is a sequence of pairs $(r_1, s_1), \dots, (r_k, s_k)$ where r_i, s_i for $i \in [1, k]$ are valuations of the shared locations. The intended meaning is that there is a computation of t such that t starts with r_1 as valuation of the shared locations and reaches s_1 , is suspended and then reactivated with shared valuation r_2 , and reaches s_2 , and so on.

In a bounded context switch analysis we can assume that computations of programs are arranged in k rounds where threads are always scheduled according to the same fixed round-robin schedule t_1, \dots, t_n . Thus, exploring the computations of a multi-threaded program up to k rounds corresponds to computing thread interfaces and composing them [15]. We start with thread t_1 and guess the in-valuations at rounds 2, \dots , k (i.e., the valuations r_2, \dots, r_k ; note that r_1 is the initial valuation of the program and thus known); we then compute the out-valuations (i.e., s_1, \dots, s_k) for thread t_1 and take them as the in-valuations of the next thread t_2 , and so on. In the end, in order to establish that the computed thread interfaces form a computation of the program we just need to check that the out-valuation of thread t_n at round $i \in [1, k-1]$ equals the (guessed) in-valuation of thread t_1 at round $i+1$.

This is the essence of the well-known sequentialization algorithm by Lal and Reps [17] and the fixed-point algorithm given in [15]. We can recast these two algorithms in our setting by means of an SMA that extends the standard SMA for SC by thread

interfaces. The resulting transition system $\mathcal{M}_{sc}^{t_i}$ is as follows. On the `init()`-transition, $\mathcal{M}_{sc}^{t_i}$ guesses a round schedule t_1, \dots, t_n , a bound k , and for each thread t_i an interface $I^i = (r_1^i, s_1^i) \dots (s_k^i, r_k^i)$ such that $r_j^i = s_j^i$ for $j \in [1, k]$. $\mathcal{M}_{sc}^{t_i}$ keeps for each thread the current round in the corresponding thread interface. If the current round of a thread is less than the round bound k , it can be increased by one by an ε -transition (i.e., it is nondeterministically either increased or left unmodified). Further, for any input sequence α , $\mathcal{M}_{sc}^{t_i}$ ensures that:

- on `write(v, val, t)` (resp. `ind_write(a, val, t)`), the out-valuation of the current round of thread t is updated according to the write;
- on `read(val, v, t)` (resp. `ind_read(val, a, t)`), the out-valuation of the current round of thread t must evaluate v (resp. a) as val .

In order to accept α , `create(t, f, t')` must occur in α for each thread t with a guessed interface, and the computed interfaces form a computation in the sense described above.

The transition system $\mathcal{M}_{sc}^{t_i}$ is thread-wise equivalent to \mathcal{M}_{sc} , and, moreover, it can execute all the computations of \mathcal{M}_{sc} by advancing each involved thread in any order. The proof of the following lemma is a consequence of the results from [15].

Lemma 3. $L(\mathcal{M}_{sc}^{t_i}) = (L(\mathcal{M}_{sc}))^\#$.

We can then recast the verification technique from [17] in our setting by taking the above SMA along with the transformation of the control-flow from [17]. Lemma 3, and Theorems 1 and 2 show the correctness of the resulting verification method. Similarly, we can combine $\mathcal{M}_{sc}^{t_i}$ with a control-flow part that at each transition nondeterministically selects the next thread to execute. The resulting system captures the verification technique from [15], and correctness is again ensured by Lemma 3, and Theorems 1 and 2. We remark that actual implementations of both these techniques require parameterization over the number of threads and rounds, as in the original implementations.

Memory unwinding. A *memory unwinding* (MU) [21] is a sequence of writes; each write w is a triple (t, v, val) where t is the identifier of the thread that has performed the write operation, v is the identifier of the memory location that is modified in the write and val is the value of v after the write. A corresponding transition system guesses an MU on the `init()`-transition and then executes the operations consistently with this guess. For SC, the corresponding transition system \mathcal{M}_{sc}^{mu} will keep for each thread the current position in the MU and for any input sequence α , it ensures that:

- on `write(v, val, t)` (resp. `ind_write(a, val, t)`), the next write in the MU for thread t matches the value `val` and variable identifier `v` (resp. address `a`);
- on `read(val, v, t)` (resp. `ind_read(val, a, t)`), there must be in the MU a write at a position i from the current position of t through the next write of t , that assigns value `val` to the location identified by `v` (resp. `a`); the current position of t is updated to i in the next state;
- for each thread, the writes are matched exactly in the same order as in the MU.

In order to accept α , `create(t, f, t')` must occur in α for each thread t with writes guessed in the MU and the writes in the MU should be mapped 1-to-1 to the writes in α .

The transition system \mathcal{M}_{sc}^{mu} is thread-wise equivalent to \mathcal{M}_{sc} , and additionally, it can execute all the computations of \mathcal{M}_{sc} by advancing each involved thread in any order. Moreover, due to the fact that all writes are guessed in advance, the ordering in which we interleave the threads is irrelevant. Thus, the following lemma holds.

Lemma 4. $L(\mathcal{M}_{sc}^{mu}) = (L(\mathcal{M}_{sc}))^\#$.

Proof. We start showing that $L(\mathcal{M}_{sc}^{mu}) \supseteq (L(\mathcal{M}_{sc}))^\#$. For $\alpha \in L(\mathcal{M}_{sc})$, denote with μ the MU that corresponds to the sequence of writes in α and with ρ an accepting run of \mathcal{M}_{sc} . We recall that \mathcal{M}_{sc}^{mu} on the $\text{init}()$ -transition can guess any MU and is built on the top of \mathcal{M}_{sc} . Thus, \mathcal{M}_{sc}^{mu} on the initial transition can enter a state storing the initial configuration γ as in ρ and μ . Now, since μ and the initial configuration γ fully capture the configurations of the shared memory along ρ (memory locations that are not assigned can be neglected), \mathcal{M}_{sc}^{mu} can simulate the execution ρ by arbitrarily advancing the execution of each involved thread in any order. Thus, \mathcal{M}_{sc}^{mu} accepts all words in $\{\alpha\}^\#$ and therefore, $L(\mathcal{M}_{sc}^{mu}) \supseteq (L(\mathcal{M}_{sc}))^\#$.

For the other direction, i.e., $L(\mathcal{M}_{sc}^{mu}) \subseteq (L(\mathcal{M}_{sc}))^\#$, let $\alpha \in L(\mathcal{M}_{sc}^{mu})$ and denote with μ the MU that is guessed on an accepting run over α . Note that for each word in $\{\alpha\}^\#$ there is an accepting run of \mathcal{M}_{sc}^{mu} such that μ is the guessed MU. Now, let $\alpha' \in \{\alpha\}^\#$ be a word where the write operations are ordered as in μ and the read operations are ordered such that for each pair of matching read and write: 1) the read follows the write, and 2) there are no other writes involving the same location between them. Clearly, $\alpha' \in L(\mathcal{M}_{sc})$ and therefore $\alpha \in (L(\mathcal{M}_{sc}))^\#$. \square

We can recast the verification approach from [21] in our setting by taking the above SMA along with the transformation of the control-flow from [21]. Lemma 4, and Theorems 1 and 2 show the correctness of the resulting verification method. Again, actual implementations would require parameterization on the number of writes and threads.

Extension to weak memory models. The discussed verification algorithms can be extended to handle programs under WMM semantics by giving the corresponding shared memory abstractions. This can be done for TSO and PSO by explicitly adding the store buffers to \mathcal{M}_{sc}^{ti} and \mathcal{M}_{sc}^{mu} , or for TSO by augmenting \mathcal{M}_{sc}^{ti} with guesses on the round when a write will be visible to all threads, as done in [7]. In the next section, we introduce a new implementation that refines the notion of MU and that works especially well for bounded model checking (BMC), and thus gives efficient BMC-implementations for verification under TSO and PSO program semantics.

D IMU-based SMA encodings

Here we give full details of the SMA implementations for SC, TSO, and PSO.

D.1 IMU implementation for SC

Data structures. We use several data structures to maintain the LUs and serve the implementation of the SMA operations. They are parameterized over the constants given

in Section 6. For simplicity, we assume that all the data is maintained as an unsigned integer (`uint`).

The triples (t, val, d) of the LUs are maintained by three different arrays `thread`, `value` and `tstamp`. Namely, for every location $v \in [0, V-1]$ and $i \in [0, W-1]$, the $(i+1)^{th}$ triple in the v -LU is stored in `thread[v][i]`, `value[v][i]` and `tstamp[v][i]`.

To keep track of the execution on the LUs we use several auxiliary variables and arrays. Namely, for every location $v \in [0, V-1]$, position $i \in [0, W-1]$ and thread identifier $t \in [0, T-1]$:

- `th_pos[v][t]` is the current position of thread t in the v -LU;
- `last_write[v]` stores the position $i \in [0, W-1]$ of the last executed write operation of location v in the v -LU. A different value for each v -LU is guessed for each simulated execution;
- `th_next_write[v][i][t]` is the first position after i in the v -LU that corresponds to a write by t , if any; otherwise, it is set to W (denoting that no further writes of v by t are expected).

Concerning to the management of threads, we keep some additional information. Variables `max_th` and `th_count` contain respectively the total number of threads that we assume should be created in the current program execution (a different value is guessed for each simulated execution) and the counting of the threads that have been actually created (this should match the guessed total number of threads in the end of computation). Also, for each thread $t \in [0, T-1]$:

- `cur_tstamp[t]` keeps track of the current timestamp of thread t during its simulation;
- `last_tstamp[t]` is the timestamp corresponding to the last write in the entire IMU by thread t ; (this value is guessed nondeterministically in the initialization and is never changed; it should match `cur_tstamp[t]` in the end of a computation);
- `ret[t]` is set to 1 to mean that t has been interrupted before reaching the end of its execution;
- `terminated[t]` is set to 1 to mean that we expect that thread t will be stopped before the execution of its last statement (this value is guessed nondeterministically in the initialization and is never changed).

To handle dynamic memory allocation and pointer arithmetics, for each location $v \in [0, V-1]$ and for each $i \in [0, M-1]$ we use:

- `address[v]` to store the physical memory address of v ;
- `mallocP[i]` to store the base address for each memory block that can be allocated dynamically;
- `mallocPallocated[i]` to track the dynamically allocated memory blocks.

IMU initialization. All the variables and arrays introduced above are declared global. On initializing the IMU we impose several constraints on them (see function `init()` in Fig. 4).

```

void init() {
    bool ts_used[V*W] = [0];
    int v=0,w=0,t=0;

    th_count = 0;

    max_th = *;
    assume( max_th <= T );

    init_address(V);
    init_malloc(M);

    while (v<V) {
        last_write[v] = *;
        assume( last_write[v] < W );
        w=0;
        while (w<W) {
            tstamp[v][w] = *;
            assume( (tstamp[v][w] < V*W) &&
                (!ts_used[tstamp[v][w]]) );
            ts_used[tstamp[v][w]]=1;
            if (w>0)
                assume(tstamp[v][w]>tstamp[v][w-1]);
            thread[v][w] = *;
            assume(thread[v][w] < max_th);
            w=w+1;
        }
        v=v+1;
    }

    while (t<T) {
        terminated[t] = *;
        last_tstamp[t] = *;
        assume (last_tstamp[t] < V*W);
        t=t+1;
    }
    v=0;
    while (v<V) {
        t=0;
        while (t<T) {
            th_next_write[v][W-1][t] = W;
            t=t+1;
        }
        v=v+1;
    }
    v=0;
    while (v<V) {
        w=W-2;
        while (w>=0) {
            t=1;
            while (t<T) {
                if (thread[v][w+1] == t)
                    th_next_write[v][w][t]=w+1;
                else
                    th_next_write[v][w][t]=
                        th_next_write[v][w+1][t];
                t=t+1;
            }
            w=w-1;
        }
        v=v+1;
    }
}

```

Fig. 4. IMU initialization.

Function `init_address` ensures that array address is nondeterministically initialized with increasing values (i.e., $address[i] < address[i+1]$ for $i \in [0, V-2]$). Function `init_malloc` ensures the same for array `mallocP` and additionally imposes that the address guessed for the last named location is less than the one assigned to the base location of the first memory allocation (i.e., $address[N-1] < mallocP[0]$). Functions `init_malloc()` and `init_address()` are illustrated in Fig. 5.

In the first while-block of Fig. 4, arrays `last_write`, `tstamp` and `thread` are nondeterministically assigned to legal values. Additionally, for each LU, timestamps are nondeterministically assigned in increasing order. The local array `ts_used` is used to ensure that different timestamps are assigned to each write in the IMU.

Legal values of `terminated` and `last_tstamp` are nondeterministically guessed in the second while-block. The rest of `init` initializes `th_next_write` such that for each thread t and each location v , all the writes from t in the v -LU are linked in the proper order (value W is used as a sentinel to denote the end of each LU).

Auxiliary functions. We make use of two auxiliary functions illustrated in Fig. 6.

Function `is_terminated` returns 1, if `ret[t]` is already set to 1, and nondeterministically chooses either to set `ret[t]` to 1 and then return 1, or to return 0. The purpose of function `Jump` is to determine the position `jump` in the v -LU of the write that determines the current value contained in v . If the timestamp of the selected write

```

void init_address(){
  int i=0;
  while (i<V){
    address[i] = *;
    if(i>0)
      assume( address[i] > address[i-1]);
    i=i+1;
  }
}

void init_malloc(){
  int i=0;
  while (i<M){
    mallocPallocated[i]=0;
    mallocP[i] = *;
    if(i>0)
      assume( mallocP[i] > mallocP[i-1]);
    i=i+1;
  }
  assume( mallocP[0] > address[N-1]);
}

```

Fig. 5. `init_address` and `init_malloc` functions for IMU implementation.

```

bool is_terminated(uint t){
  if(ret[tid]||nondet()) {ret[tid]=1; return 1;}
  return 0;
}

uint Jump(uint t, uint v){
  uint jump=*;

  ts_jump = tstamp[v][jump];
  assume( (jump <= last_write[v])
    && (jump < th_next_write[v][t][th_pos[v]])
    && (tstamp[v][jump+1] > cur_tstamp[t]));
  cur_tstamp[t] = (ts_jump > cur_tstamp[t]) ? ts_jump : cur_tstamp[t];
  return jump;
}

```

Fig. 6. Auxiliary functions for IMU implementation.

is past the current thread timestamp, the last is updated to this value by acknowledging the fact that the corresponding write into the shared memory has occurred. The value of `jump` is selected nondeterministically within a range of proper values. Namely, `jump` should not pass the last legal write position for `v` and must be strictly less than the position of the next write of `v` by the same thread `t` (that has not occurred yet). Further, we require that the timestamp at position `jump+1` is greater than the current timestamp of `t` (we wish to point to a write of `v` that is not superseded by an already occurred write).

Thread creation, termination, and join. The implementations of functions `create`, `terminate` and `join` are shown in Fig. 7.

In function `create`, if the maximal number of allowed threads is reached, the procedure immediately returns `-1` meaning that this thread will never be scheduled. Otherwise, the count of the created threads is incremented and the current timestamp and LU positions of the new created thread are initialized such that: they coincide with those of the parent thread.

The `assume` statement ensures that no write operations are entitled to the new created thread before its creation. Since we update the positions of each thread in the LUs forward only, this will ensure also that each thread will not use any LU position corresponding to a write operation that is supposed to occur before its creation.

Function `terminate` checks that all write operations guessed for thread `t` have been done (while-loop). Furthermore, the concluding `assume` checks that the values guessed by function `init` for `terminated[t]` and `last_tstamp[t]` are consis-

```

int create(void *f, uint pt){
  if(th_count >= max_th) then return -1;fi
  th_count++;
  uint v=0;
  if(pt == 0){
    while (v < V){
      th_pos[v][th_count]=0;
      v=v+1;
    }
    cur_tstamp[th_count]=0;
  }else{
    cur_tstamp[th_count]=cur_tstamp[pt];
    while (v < V){
      th_pos[v][th_count]=th_pos[v][pt];
      assume(th_next_write[v][0][th_count]
             >=th_pos[v][th_count]);
      v=v+1;
    }
  }
  return th_count;
}

void terminate(uint t){
  uint i, v=0;
  while (v < V){
    i=th_pos[v][t];
    assume( th_next_write[v][i][t]
           > last_write[v] );
    v=v+1;
  }
  assume( ret[t]==terminated[t] &&
         last_tstamp[t]==cur_tstamp[t] );
}

void join(uint t1, uint t2){
  if(is_terminated(t1)) then return;
  uint v;
  assume( v < V );
  Jump(t1,v);
  assume( (terminated[t2] == 0) &&
         (cur_tstamp[t1]>=last_tstamp[t2]));
}

```

Fig. 7. Functions `create`, `terminate` and `join`.

tent with the explored computation. We recall that `ret[t]` is initialized to 0 and can be nondeterministically set to 1 by the auxiliary function `is_terminated` (this has the effect of stopping the execution of the current thread).

Function `join` returns immediately if the execution of thread `t1` is stopped. Otherwise, the timestamp of `t1` is updated invoking `Jump` on a nondeterministically guessed variable (this ensures a choice of the new timestamp among all the LUs of `t1`). The computation is aborted whenever the other thread (`t2`) either will not terminate (i.e., `terminated[t2]==1`) or has not terminated yet at the current timestamp of `t1` (but it is supposed to terminate).

Read and write operations. The implementation of functions `read` and `write` is illustrated in Fig. 8. For a read operation, the thread under simulation t first jumps forward into the v -LU corresponding to the variable given as parameter by invoking the auxiliary procedure `Jump` described above and then returns the valuation of the variable at the new position from matrix `value`.

In a write operation, the thread first jumps to its next write operation for that variable and blocks the simulation if the value disagrees with that in the memory sequence at the new position. Furthermore, we also check that the timestamp associated to the new position is greater than the actual timestamp of t ; this to prevent to simulate already simulated write operations. Then we update the current position of thread t in the v -LU and the current timestamp.

Address and malloc operations. Method `address` is used to recover the address of a given location $v \in [V - 1]$. The implementation for this method is given in Fig. 9. If v corresponds to a scalar variable the method returns the value from `address[v]`; otherwise it simulates the read operation at that location.

During its execution a thread can require a block of n consecutive unallocated locations by invoking `malloc(n)`. When `malloc` is invoked, say with argument n , a block is chosen non deterministically, and it is allocated if its size is at least n by returning its

```

int read(uint v,uint t){
    if(is_terminated(t)) return 0;
    uint jump = Jump(t,v);
    return (value[v][jump]);
}

void write(uint v,int val,uint t){
    if(is_terminated(t)) return;
    uint i, jump;
    i = th_pos[v][t];
    jump=th_next_write[v][i][t];
    assume( (jump<=last_write[v])
        && (value[v][jump] == val)
        && (tstamp[v][jump] > cur_tstamp[t])
    );
    th_pos[v][t]=jump;
    cur_tstamp[t]=tstamp[v][jump];
}

int ind_read(uint addr,uint t){
    if(is_th_terminated(t)) return 0;
    uint pos;
    assume(pos<V);
    assume(address(pos,t)==addr);
    return read(pos,t);
}

void ind_write(uint addr,int val,uint t){
    if(is_th_terminated(t)) return;
    uint pos;
    assume(pos<V);
    assume(address(pos,t)==addr);
    write(pos, val,t);
}

```

Fig. 8. Read and write functions.

```

int address (uint v, uint t){
    if(is_th_terminated(t)) return 0;
    if(v<N) return address[v];
    return read(v, t);
}

int malloc(uint n, uint t){
    uint pos;

    if(is_th_terminated(t)) return 0;
    assume(pos<M);
    assume(!mallocPallocated[pos]);
    assume(mallocP[pos]+n < mallocP[pos+1]);
    mallocPallocated[pos]=1;
    return mallocP[pos];
}

```

Fig. 9. Functions address and malloc.

base address. The malloc procedure is implemented as shown in Fig. 9. We first find a position pos that corresponds to a not sill allocated block, by checking the value of `mallocPallocated` at that position. We recall that addresses stored in `mallocP` are ordered in ascending order; then in order to know if there is enough space we simply check that `mallocP[pos]+n < mallocP[pos+1]`. Then we set `mallocPallocated[pos]` to true to indicate that the address at position pos has been allocated. Finally, we return the base address corresponding to the position pos .

Ind_read and ind_write operations. When a read or write operation is performed using a memory address, i.e. $*p = 3$ for a pointer variable p , we invoke `ind_read` and `ind_write` methods. The implementation of the these procedures are straightforward (see Fig. 8). We first search for the location corresponding to that whose address corresponds to the given parameter and then simulate the read/write operation at that location.

Lock and unlock mutex variables. A thread can take or release a lock on a shared mutex variable by calling the procedure `loc` and `unlock`, respectively; their implementations are provided by Fig. 10. For a mutex variable, we assign value 0 when the lock is not acquired by any thread, and we assign value t if the mutex is held by thread t .

```

void lock(uint mut, uint t)
write(mut, t, t);
assume(ret[t] ||
value[mut][th_pos[mut][t]-1]==0);
}

void unlock(uint mut, uint t)
write(mut, 0, t);
assume(ret[t] ||
value[mut][th_pos[mut][t]-1]==t);
}

```

Fig. 10. Mutex lock and unlock operations.

For efficient implementation, we modify the value of variable `mut` using a write operation. For a `lock` operation we first write the value of `t` in `mut`; however, it may be the case that the mutex was already held by some thread. Thus, we check that the previous value of `mut` was 0. The implementation of the method `unlock` procedure is similar, the only difference is that we write 0 in to the `mut` variable. Note that, two consecutive write operations of `mut` are performed by the same thread (`lock` and `unlock`). Furthermore, the value written at the even positions of the `mut`-LU are always 0. These constrains can be added in the `init` function to reduce the number of runs to consider.

D.2 IMU implementation for TSO

We give this implementation incrementally on that given for SC.

To be consistent with the notation used in the implementation for SC, we use `tstamp[v][i]` to store the update timestamp concerning the $(i+1)^{th}$ write of location `v`, and `cur_tstamp[t]` to keep track of the current timestamp in the execution of thread `t` (i.e., the occurrence timestamp of the last occurred read or write). Additionally, we use two new arrays `btstamp` (buffer timestamps) and `ts_lastW` such that:

- `btstamp[v][i]` is the occurrence timestamp of the $(i+1)^{th}$ write of `v` (that is also the time at which it is stored in the local buffer of the thread that performs the write operation);
- `ts_lastW[t]` is the update timestamp of the last occurred write by thread `t`.

To implement the SMA API, we only need to give an implementation of `fence` and modify those given for SC of `init`, `read`, `write`, `lock` and `unlock`. The rest of the implementation is the same as for SC.

For `init`, we add to the implementation given for SC the following. We nondeterministically guess initial values for `btstamp[v][i]` and then impose that `btstamp[v][i] ≤ tstamp[v][i]` must hold (i.e., the update of the shared memory according to an occurred write may be delayed w.r.t. its occurrence time).

Note that here we slightly diverge from the transition system \mathcal{M}_{tso}^{imu} described in Section 5. In fact, since we do not require any other condition on the guessed update timestamps, we can carry over an IMU with timestamps that may violate the FIFO policy on the store buffers. This is fixed by checking the proper ordering on matching the writes (we return on this when discussing the write implementation).

Function `lock` from Fig. 10 is modified such that the write is done by a routine `Write_SC` that is exactly the write given for SC instead of the `write` for TSO. This ensures that lock acquisition is immediately visible to all the other threads. For function

```

int read(uint v, uint t){
    if(is_terminated(t)) return 0;

    uint ts_jump, i;
    i = th_pos[v][t];
    uint nxt_write=th_next_write[v][i][t];
    uint fst_write=th_next_write[v][0][t];
    assume (
        (ts_jump >= cur_tstamp[t]) &&
        (ts_jump < btstamp[v][nxt_write])
    );
    cur_tstamp[t]=ts_jump;
    if( (fst_write <= i) &&
        (tstamp[v][i] > cur_tstamp[t])
    )
        return value[v][i];
    return Read_SC(v,t);
}

void fence(uint t){
    if(ts_lastW[t]>cur_tstamp[t])
        cur_tstamp[t] = ts_lastW[t];
}

void write(uint v, int val, uint t){
    if(is_terminated(t)) return;
    i = th_pos[v][t];
    jump=th_next_write[v][i][t];
    th_pos[v][t]=jump;
    assume(
        (btstamp[v][jump] > cur_tstamp[t])
        && (value[v][jump] == val) &&
        (tstamp[v][jump] > ts_lastW[t])
    );
    ts_lastW[t] = tstamp[v][jump];
    cur_tstamp[t] = btstamp[v][jump];
}

```

Fig. 11. Functions `read`, `fence` and `write` for TSO.

unlock, we do the same and further before returning we call `fence`. This way, we make immediately visible to all the other threads all the writes that occurred in the critical section.

The code of functions `fence`, `read` and `write` are illustrated in Fig. 11.

A memory *fence* flushes the store buffer of the thread executing it and thus we need to synchronize the current thread timestamp with its last update timestamp. Namely, if `ts_lastW[t]` is larger than the timestamp of the last occurred write by `t`, we assign `ts_lastW[t]` to `cur_tstamp[t]`. Note that if this is not the case then the local store buffer of `t` is certainly empty (recall $btstamp[v][i] \leq tstamp[v][i]$).

Function `read` first updates nondeterministically the current timestamp of thread `t` such that it is not smaller than the current timestamp of `t` and is smaller than the update timestamp of the next write of `t`. Now, if at least a write of location `v` by `t` has occurred and the last write of `v` by `t` is still in the thread buffer, then we return the value of this write. Otherwise, a read from the shared memory is performed by invoking the auxiliary function `Read_SC` that is exactly the function `read` from Fig. 8.

Observe that the update of the current thread timestamp by `read` can cause this value to be larger than the update timestamp of the last write and this may be correct. To avoid that we wrongly move the time back, in `fence` we make the assignment only when this is not the case.

Function `write` first updates the current position in the `v`-LU of thread `t` to the next write provided that the time of occurrence of this write is larger than the current thread timestamp, the value of the write matches the guessed value for it and the update timestamp of the next write is larger than that of the last occurred write (the last one ensures that the thread store buffers are emptied according to a FIFO policy). Note that, in the case of a wrong guess of the update timestamps in `init`, this condition would not hold and thus the execution would abort. Before returning, the update timestamp of the last write and the current timestamp of thread `t` are modified consistently.

D.3 IMU implementation for PSO

We can give an implementation of SMA for PSO by slightly modifying the implementation given for TSO as follows.

We use a new array `max_tsW` in substitution of `ts_lastW` and change a few lines in the implementation of function `write`. Array `max_tsW` maintains for each thread `t` the maximum update timestamp among all the occurred writes of `t`.

In function `write` (Fig. 12), we do not require any more that the update timestamp of the current write is larger than the update timestamp of the previous write by `t`. Recall that this was required in the TSO implementation in order to ensure that for each thread `t`, the guessed occurrence and update timestamps for the sequence of writes by `t` (that may be contained in different LU's) are indeed consistent with the FIFO policy of a store buffer; in PSO we only need to ensure that the FIFO policy holds for each of the maximal subsequences containing all the writes of a same location which is ensured by the remaining constraints and function `init`. Moreover, the update of `ts_lastW[t]` is replaced with the update of `max_tsW[t]` as follows:

```
max_tsW[t] =  
    (tstamp[v][jump] > max_tsW[t]) ? tstamp[v][jump] : max_tsW[t];
```

```
void write(uint v, int val, uint t){  
    if(is_terminated(t)) return;  
    i = th_pos[v][t];  
    jump=th_next_write[v][i][t];  
    th_pos[v][t]=jump;  
    assume(  
        (btstamp[v][jump] > cur_tstamp[t])  
        && (value[v][jump] == val)  
    );  
    max_tsW[t] =  
        (tstamp[v][jump] > max_tsW[t]) ?  
            tstamp[v][jump] : max_tsW[t];  
    cur_tstamp[t] = btstamp[v][jump];  
}
```

Fig. 12. Function `write` for PSO.