Separating Computation from Communication: A Design Approach for Concurrent Program Verification

Ermenegildo Tomasco[1], Truc L. Nguyen[1], Bernd Fischer[2], Salvatore La Torre[3], and Gennaro Parlato[1]

[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
{et1m11,tnl2g10,gennaro}@ecs.soton.ac.uk,bfischer@cs.sun.ac.za,
slatorre@unisa.it

Abstract. We describe an approach to design static analysis and verification tools for concurrent programs that separates intra-thread computation from inter-thread communication by means of a shared memory abstraction (SMA). We formally characterize the concept of thread-asynchronous transition systems that underpins our approach and that allows us to design tools as two independent components, the intra-thread analysis, which can be optimized separately, and the implementation of the SMA itself, which can be exchanged easily (e.g., from the SC to the TSO memory model). We describe the SMA's API and show that several concurrent verification techniques from the literature can easily be recast in our setting and thus be extended to weak memory models. We give SMA implementations for the SC, TSO, and PSO memory models that are based on the idea of individual memory unwindings. We instantiate our approach by developing 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. The advent of modern multi-core hardware architectures that implement *weak memory models* (WMMs) has 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 ensuring correctness, or at least to finding bugs. It can be effective if the fraction of buggy executions is high, but it remains highly ineffective for bugs that manifest themselves only rarely and are difficult to reproduce [30]; however, such "Heisenbugs" are unfortunately more prevalent with WMMs. Since other verification approaches that explore executions *explicitly* face the same problems as testing, even with optimizations such as partial order reduction that eliminate redundant executions, we need approaches that can handle multiple concurrent executions *symbolically*.

However, it is difficult to build efficient symbolic verification tools for realistic programming languages like C, and harder yet to extend them to handle concurrency. Tools thus often compromise generality to achieve efficiency, by focusing on a specific memory model, typically sequential consistency (SC), and by folding the concurrency handling deep into their general verification approaches (see [1,3,5,8,33,34]). This in turn introduces a strong coupling between the two, which makes it hard to reuse existing tools and to generalize solutions to other memory models. Our goal here is to break this coupling and to separate the computation (i.e., individual threads) and the communication (i.e., shared memory) concerns of concurrent programs, without loosing the efficiency of existing approaches.

More specifically, we develop an approach that allows us to combine different concurrent verification techniques with different memory models in the style of a plug-and-play architecture. For this, we define and describe in Section 3 an interface that we call *shared memory abstraction* (SMA). The SMA captures the 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. We then assume that all operations involving concurrency are performed by invoking the corresponding SMA operations (which can easily be achieved by rewriting non-conforming programs). In this way, we achieve the desired separation of concens—in fact, we can even view a multi-threaded program as the composition of two independent *sub-systems*, one comprising all threads and one capturing the concurrency (including the memory model), which synchronize using the API provided by the SMA.

We formalize this view in Section 4.1 and introduce, as a first contribution, the concepts of *thread-wise equivalence* and *thread-asynchronous closure* of transition systems. We show that reachability is preserved if we exchange the transition system of a program for a thread-wise equivalent one (assuming the SMA is thread-asynchronous) or an SMA for its thread-asynchronous closure. This has two important consequences. First, it allows us to generalize existing concurrent verification approaches to different memory models simply by implementing the corresponding different SMAs. Second, it gives us a degree of freedom in designing concurrent verification algorithms, since it allows us to rearrange the order in which the verifier explores the execution of the statements among different threads. This is implicitly exploited by some algorithms from the literature, such as the sequenzialization by Lal and Reps [23] and the fixed-point algorithm from [21] where bounded-round computations are explored by executing each individual thread to completion, or by the sequentialization from [31] where each thread is executed in isolation with respect to a sequence of writes that is guessed in the beginning. We further show in Section 4.2 how these algorithms can be recast in our setting, yielding correctness proofs for free.

However, the way the computation and communication concerns are combined affects the scalability of the resulting verification tool. In Sections 5 and 6, we therefore instantiate, as second contribution, our general approach to achieve an efficient BMC-based bug-finding tool. We describe 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, and we show through experiments in Section 7 that our tool compares well with existing tools.

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. These models use buffering to speed up execution time of multi-threaded programs.

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 [24].

Total store ordering (TSO). The behaviour of the TSO memory model can be described using the simplified architecture shown in Fig. 1 (cf. [27]). 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.

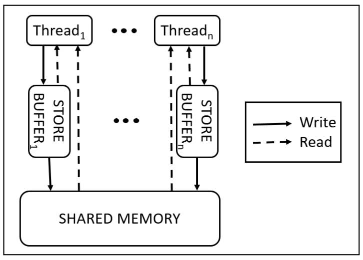

Fig. 1. TSO architecture.

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.

We represent memory models as state transition systems whose configurations keep track of the valuation of the shared locations, the state of the store buffers (if any), the status of each created thread (i.e., active, ready, suspended or terminated) and the status of each mutex (i.e., locked or unlocked).

3 Multi-Threaded Programms over Shared Memory Abstractions

In this paper, we consider multi-threaded programs with a 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 flushes all store buffers of a thread. The exact program syntax is defined in the Appendix by the grammar shown in Fig. 4.

3.1 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 `t` as an extra argument to allow the SMA to update its internal state. In detail, the SMA API is formed of the following functions:

- `init()` initializes the SMA and the shared variables; this must be the first statement in the program;
- `address(v,t)` returns the memory address of the shared variable `v`;
- `malloc(n,t)` allocates a continuous block of `n` memory locations and returns the base address of the block;
- `read(v,t)` (resp. `ind_read(a,t)`) returns the valuation of the shared variable `v` (resp. memory location with address `a`) as seen by `t`;
- `write(v,val,t)` (resp. `ind_write(a,val,t)`) sets the valuation of the shared variable `v` (resp. memory location with address `a`) to the value `val`;
- `fence(t)` flushes all store buffers of `t` and updates the shared memory;
- `lock(m,t)` and `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;
- `create(f,t)` spawns a new thread that starts from function `f`, and returns a fresh thread identifier for this thread;
- `terminate(t)` terminates the execution of `t`; each thread must explicitly call it at the end;
- `join(t',t)` pauses the execution of `t` until `t'` has terminated its execution.

3.2 Multi-threaded programs as composition of transition systems

The formal semantics of multi-threaded programs is often given by a transition system (see Appendix for the formal definitions) that captures the program computations by interleaving the computations of each thread. For our class of programs we exploit the separation between the control flow and the shared memory aspects introduced with the notion of SMA. Thus, the semantics of a multi-threaded program is given as the composition $\mathcal{C}|\mathcal{M}$ of the *control-flow transition system* $\mathcal{C}$ that captures the control flow of the program and the *shared memory abstraction 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.

These two transition systems are synchronized over the SMA API that defines the alphabet that labels the transitions of $\mathcal{C}$ and $\mathcal{M}$. More precisely, the alphabet consists of 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, read(3,v,t) is the letter corresponding to a call `read(v,t)` that returns value 3. We denote this alphabet with Σ_{SMA}.

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

The transitions correspond to the execution of any of the statements. Transitions corresponding to invocations of API functions of SMA are labeled with the corresponding letter from Σ_{SMA}. In particular, transition from the initial state are labeled with init() and enter a state with the starting configuration of the main thread. No other transitions are labeled with init(). Transitions corresponding to SMA functions that return a value are handled as assignments of the corresponding variables with the returned values. Additionally, 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 terminate(t) is to delete the configuration of the terminated thread. The remaining transitions over 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. In general, an SMA transition system $\mathcal{M}$ has an initial state and a state for each possible configuration of the corresponding memory model. The transitions update memory configurations to capture the memory model's intended meaning. Note that from the initial state there are only outgoing transitions, which are all labeled with init(), and no other transition have this label.

For SC, the system $\mathcal{M}_{sc}$ can enter from the initial state any state that has only one thread (which must be active), has any number of shared locations (which must all have the value of zero), and has any number of mutexes (which must all be unlocked). All other transitions update the state of $\mathcal{M}_{sc}$ according to the meaning given in Section 2. Since there are no store buffers in SC, there are no fence-transitions. Further, in a transition on terminate(t), $\mathcal{M}_{sc}$ enters a state where the status of *t* is terminated. From any such state only states where the *t* status remains terminated can be reached, and no other transitions corresponding to invocations of API functions from *t* are allowed. The final states are all the states where all the threads are terminated.

For the WMMs we denote the corresponding SMA transition system with $\mathcal{M}_{tso}$ and $\mathcal{M}_{pso}$, respectively. The states of both systems also account for the content of the thread store buffers, the transitions on reads and writes reflect the corresponding semantics as described in Section 2, and there are fence(t)-transitions on calls to `fence` by t and ε-transitions for store buffer updates.

4 Verification with thread-asynchronous SMAs

In this section, we discuss our design approach for the verification of multi-threaded programs. Its basis is the separation between the intra-thread control-flows and the SMA already discussed in Section 3. In this view, a verification tool is composed of an SMA implementation and a search algorithm that explores the program executions. This by itself allows for a convenient way to extend verification methods to other memory models by simply replacing the SMA implementation. However, this might not result in scalable verification tools, for the following reasons.

First, to preserve the correct semantics of the memory operations, these must be invoked in the same order as they appear along the run, which may be a bottleneck when we explore the state space of the program, both in case of the analysis based on summaries (e.g., BDD-based model checking) or bounded model checking. In the former, we must keep a cross-product of the states of all threads in the configurations; this is a well-known problem that leads to state-space explosion. In the latter, since context-switches can happen at any point, we must encode into the SAT/SMT formula the code of all threads for each of the context-switch points in the underlying bounded multi-threaded program, which leads to large formulas.

Some approaches from the literature instead explore the program executions by rearranging the order in which the memory operations of the different threads are executed, e.g., by simulating each thread to completion [21,23]. Another example is the sequentialization presented in [31] where each thread is executed in isolation with respect to a memory unwinding (i.e., a sequence of writes that is guessed at the beginning). More generally, the approach of verifying each thread in isolation is also the essence of the compositional approaches based on assume-guarantee reasoning [25,18].

We generalize the ad-hoc approaches above, and present a general framework in which to design concurrent program verification approaches. This requires that the used SMA implementation is *thread-asynchronous*, that is that its behaviours are 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, that is programs where the *intra-thread ordering* of the statements remains the same. This, along with the correctness of the derived design approach, will be formalized below.

We conclude the section by discussing how previous successful approaches from the literature fit into our setting. In following sections, we will then give an efficient implementation of a thread-asynchronous SMA and show that this can be combined with existing search algorithms to achieve a competitive verification tool.

4.1 Thread-asynchronous SMAs

For a thread t, we denote with Σ^t_{SMA} 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^t_{SMA} and $\Sigma^{t'}_{SMA}$ are disjoint. For a thread t and a word α over Σ_{SMA}, let $\alpha_{|t}$ be the projection of α onto Σ^t_{SMA}, i.e., the word obtained from α by deleting all the letters that do not belong to Σ^t_{SMA}. If $t_1, \ldots, t_h$ 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 of words over Σ_{SMA} is *thread-asynchronous* if for each $\alpha \in L$ and for each α' starting with $\mathsf{init}()$ s.t. $\pi(\alpha) = \pi(\alpha')$, also $\alpha' \in L$. The *thread-asynchronous closure* of a language 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' accepted by the other one such that $\pi(\alpha) = \pi(\alpha')$.

A standard analysis for multi-threaded programs is to search for the reachability of an error program counter of a given thread (*local error state*), often denoted by an error label or a `false`-assertion. In the following, we give two theorems stating sufficient conditions under which the reachability of local error states is preserved.

The first theorem states that if the SMA is thread-asynchronous we can transform a program P_1 into a thread-wise equivalent program P_2 such that a local error state is reachable in the resulting program P_2 if and only if it is reachable in P_1. 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 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 a local error state 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. In the next subsection, we discuss how to implement thread-asynchronous SMAs that are suitable to recast known verification approaches for SC and extend them to WMMs.

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 local error states. 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 a local error state is reachable in $\mathcal{C}|\mathcal{M}_1$ iff it is reachable in $\mathcal{C}|\mathcal{M}_2$.*

By combining both theorems, we can easily show the correctness of WMM extensions of correct verification methods that transform programs by keeping the ordering of the sequence of the operations within each thread. In fact, we just need to provide an SMA that captures the thread-asynchronous closure of the memory model.

4.2 Thread-asynchronous SMAs for thread interfaces and memory unwinding

We briefly recall the notions of thread interface [21] and memory unwinding [31], 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 $(r_1, s_1), \ldots, (r_k, s_k)$ 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 $t_1, \ldots, t_n$. Thus, exploring the computations of a multi-threaded program up to k rounds corresponds to computing thread interfaces and composing them [21]. We start with thread t_1 and guess the in-valuations at rounds $2, \ldots, k$ (i.e., the valuations $r_2, \ldots, r_k$; note that r_1 is the initial valuation of the program and thus known); we then compute the out-valuations (i.e., $s_1, \ldots, s_k$) 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 [23] and the fixed-point algorithm given in [21]. 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 $\mathcal{M}_{sc}^{ti}$ is as follows. On the init()-transition, $\mathcal{M}_{sc}^{ti}$ guesses a round schedule $t_1, \ldots, t_n$, a bound k, and for each thread t_i an interface $I^i = (r_1^i, s_1^i) \ldots (s_k^i, r_k^i)$ such that $r_j^i = s_j^i$ for $j \in [1, k]$. $\mathcal{M}_{sc}^{ti}$ 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 nonderministically either increased or left unmodified). Further, for any input sequence α, $\mathcal{M}_{sc}^{ti}$ 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 $\mathcal{M}_{sc}^{ti}$ 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 [21].

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

We can then recast the verification technique from [23] in our setting by taking the above SMA along with the transformation of the control-flow from [23]. Lemma 1, and Theorems 1 and 2 show the correctness of the resulting verification method. Similarly, we can combine $\mathcal{M}_{sc}^{ti}$ with a control-flow part that at each transition nondeterministically selects the next thread to execute. The resulting system captures the verification technique from [21], and correctness is again ensured by Lemma 1, 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) [31] 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 2. $L(\mathcal{M}_{sc}^{mu}) = (L(\mathcal{M}_{sc}))^{\#}$.

We can recast the verification approach from [31] in our setting by taking the above SMA along with the transformation of the control-flow from [31]. Lemma 2, 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 weak memory model 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 [5]. 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.

5 Individual Memory-Location Unwindings

In this section, we discuss an efficient implementation of thread-asynchronous shared memory abstractions for SC, TSO and PSO memory models. It builds on the idea of memory unwinding recalled in the previous section. The two main innovations are:

- the splitting of the memory unwinding into different sequences, one for each individual shared memory location (*location unwinding*, LU for short), and
- the introduction for each write of a *timestamp*, i.e., a natural number that denotes the time of occurrence of a write according to a discrete-time global clock.

An *individual memory-location unwinding* (IMU) is then a set containing exactly one LU for each memory location and such that the timestamps determine a total order among all the writes of all the LUs.

Splitting the memory unwinding into smaller sequences works well when used in combination with BMC verification tools: the read and write operations result in much smaller verification conditions; for each memory access, only the corresponding individual sequence needs to be duplicated and not the whole sequence of writes. Further, the shared memory abstraction capturing SC based on IMU can be easily extended to accommodate TSO and PSO. In fact, this can be done at the cost of adding for each write a second timestamp denoting the time at which the write is moved to the shared memory (and thus becomes visible to all threads). Moreover, the split of the writes among the shared memory locations makes the transition from TSO to PSO trivial.

IMU-based SMA for SC. 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 is a positive integer denoting the time at which val is written into v according to a discrete global clock (*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 $d_1 \neq d_2$. Note that timestamps define a total order among all the writes in the IMU.

The IMU-based shared memory abstraction for SC can be constructed similarly to $\mathcal{M}_{sc}^{mu}$. We only remark here the main differences: on the $\mathsf{init}()$-transition an IMU is guessed instead of an MU; the *current* timestamp (i.e., the timestamp of the last executed SMA operation) is maintained for each thread; in a read by a thread t the position of the matching write is guessed such that the corresponding timestamp d is between the current timestamp of t and the timestamp of the next write by t (the current timestamp of t is updated to d after the transition); the current timestamp of a thread t is also updated to the timestamp of a write when this is executed.

The total ordering of timestamps across all the IMU ensures the equivalence with a corresponding MU where the writes are written by increasing timestamps, and vice-versa (in an MU the timestamps are given implicitly by the order of the writes in the sequence). We thus get the following lemma for the resulting transition system $\mathcal{M}_{sc}^{imu}$.

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

IMU-based SMA for TSO and PSO. To capture the TSO and PSO semantics, we introduce into the IMU 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 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 for the store buffers along any program execution, we also require that for each thread the writes must be following the same order, if ordered by non-decreasing timestamps according to either one of the sequences of timestamps (i.e., either the occurrence or the update timestamps). For PSO, instead this requirement is replaced with a weaker one that ensures a FIFO policy only for the writes of a same location performed by the same thread.

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 $\mathsf{init}()$-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 $\mathsf{fence}(\mathsf{t})$-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 the described transition systems capture the semantics of the corresponding memory models. Moreover, as for the MU case, since all the writes are guessed in advance, the ordering in which we interleave the threads is again irrelevant. Thus, we get the following lemma:

Lemma 4. *For* $m \in \{tso, pso\}$, $L(\mathcal{M}_m^{imu}) = (L(\mathcal{M}_m))^{\#}$.

Verification by IMU. By composing the transformation of the control-flow from [31] along with the SMA implementations $\mathcal{M}_{sc}^{imu}$, $\mathcal{M}_{tso}^{imu}$ and $\mathcal{M}_{pso}^{imu}$ we get new methods for the verification of multi-threaded programs under SC, TSO and PSO semantics, respectively. The correctness of such methods is a consequence of the lemmas given above in this section, and Theorems 1 and 2. We will give concrete implementations of these methods and evaluate them in the next two sections.

6 IMU-based SMA implementations

In this section, we discuss concrete C-implementations of SMAs whose semantics is captured by $\mathcal{M}_{sc}^{imu}$, $\mathcal{M}_{tso}^{imu}$ and $\mathcal{M}_{pso}^{imu}$, respectively. Each of them implements the SMA API defined in Section 3. In the remainder of this section we will give some details of the implemented code; a full version is in the Appendix. Our code is optimized for an efficient analysis using BMC tools but implementations for other backends are possible.

6.1 IMU implementation for SC

The implementation is parameterized over several constants; note that `V`=`N`+`U` holds.

- `T` denotes the maximal number of threads that can be spawned during any execution of the input program,
- `V` denotes the number of tracked memory locations,
- `N` denotes the number of shared scalar variables (*locations with names*),
- `U` denotes the maximum number of locations that can be accessed only through their memory addresses (*locations without names*),
- `M` denotes the maximum number of dynamic memory allocations, and
- `W` denotes the maximum number of write operations for each location.

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 (see Appendix). 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` $\in [0, \texttt{V}-1]$ and `i` $\in [0, \texttt{W}-1]$, the triple at position `i` in the `v`-LU is stored in `thread[v][i]`, `value[v][i]` and `tstamp[v][i]`. We 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 the same timestamp;
– for every location `v` $\in [0, \texttt{V}-1]$, position `i` $\in [0, \texttt{W}-1]$ and thread identifier `t` $\in [0, \texttt{T}-1]$, `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` $\in [0, \texttt{V}-1]$ and thread identifier `t` $\in [0, \texttt{T}-1]$:

– `th_pos[v][t]` stores the current position of thread `t` in the `v`-LU;
– `last_write[v]` stores the position `i` $\in [0, \texttt{W}-1]$ 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 functions `read` and `write`, which is given in Fig. 2. 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. 2) 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 `t`). However, note that the corresponding timestamp could be still larger then 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.

```
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];

   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. 2. Read, write, and jump functions.

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 at the position. 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` as described above, 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 stated invariants.

6.2 IMU implementation for TSO

We give this implementation incrementally on that given for SC; the code of the functions `fence`, `read` and `write` is illustrated in Fig. 3. We use `tstamp`[`v`][`i`] to store the update timestamp concerning the write at position `i` in the `v`-LU, and `cur_tstamp` [`t`] to keep track of the current timestamp in the execution of thread `t` (i.e., the occurrence timestamp of the read or write that occurred last). Additionally, we use two new arrays `btstamp` (buffer timestamps) and `ts_lastW` such that:

- `btstamp`[`v`][`i`] is the occurrence timestamp of the write at position `i` in the `v`-LU (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 write by thread `t` that occurred last.

For `init`, we nondeterministically 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. This is fixed by checking the proper ordering on matching the writes (see below).

```
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. 3. Functions `read`, `fence` and `write` for TSO.

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. 2.

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.

6.3 IMU implementation for PSO

We can get a PSO-SMA by slightly modifying the TSO-version 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`. We achieve this by re-

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

			CBMC svc16			CIVL svc16			Lazy-CSeq svc16			MU-CSeq svc15			IMU-CSeq		
sub-category	files	l.o.c.	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**

placing 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. 3). We recall that this conjunct was required in the TSO implementation 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 store-buffer FIFO policy; in PSO, we only need to require this within each LU, which is thus ensured by the remaining constraints of `write` and `init`.

7 Experimental Evaluation

We have implemented our approach in a prototype tool IMU-CSeq, where we first use MU-CSeq [31] to transform the original multi-threaded program into a sequential one (*sequentialization*), then link this against an IMU-based SMA implementation, and finally verify the resulting program with a BMC tool for sequential programs, in particular CBMC (v5.3). Depending on the chosen SMA implementations we thus obtain a tool for verifying multi-threaded programs under SC, TSO, and PSO, respectively.

SC benchmarks. We have evaluated this prototype on benchmarks from the Concurrency-category of the TACAS Software Verification Competition (SV-COMP16) [7]. These are widespread benchmarks, and many state-of-the-art analysis tools have been trained on them; in addition, they offer a good coverage of the core features of the C programming language as well as of the basic concurrency mechanisms.

The whole benchmark set consists of 1015 files, of which 791 have a reachable error location. Since we use a BMC tool as a backend, we cannot prove correctness, but can only show that an error is not reachable within the given bounds. We therefore only evaluate our prototype on such unsafe files. In particular, we used the files from the sub-categories shown in Table 1.

The experiments were run on a dedicated machine with a Xeon E5-2650 v2 with 2.60 GHz and 132 of RAM, running a Linux 4.2.0-22-generic operating system. The verifiers were given a a 15GB memory limit and a 900s timeout. The files are analyzed under SC semantics. The experiments are summarized in Table 1. Each row corresponds to a sub-category of the SV-COMP16 benchmarks, where we report the number of files and the total number of lines of code. Note that the different *pthread-wmm-** sub-categories are missing. Our current prototype cannot currently handle these benchmarks, which have a large number of shared variables and write operations. However, the original MU-approach had similar problems which we could overcome by exposing only a subset of the write operations (*coarse-grained unwinding*, see [32]), and we are currently exploring similar ideas for the IMU-approach. The table shows the results for CBMC [4], CIVL [35], Lazy-CSeq [16,17], MU-CSeq [31],[1] and IMU-CSeq on these benchmarks. Furthermore, 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 the MU-based MU-CSeq.

WMM benchmarks. We also compared our prototype against two tools with built-in support for analysis under weak memory models, CBMC [15], and Nidhugg [1], a bug-finding tool that combines stateless model checking with dynamic partial order reduction on relaxed memory executions. These experiments were run on a dedicated machine with a Xeon W3520 2.6GHz processor and 12GB of physical memory running 64-bit linux 3.0.6. We set a 10GB memory limit and a 600s timeout for the analysis of each of the simple benchmarks and timeout of 14,400s for safestack. For each tool and benchmark, we set the parameters to the minimum value needed to expose the error.

Simple benchmarks. We first used a set of (relatively simple) benchmarks collected from the CBMC, Poet, and Nidhugg tools, and the SV-COMP benchmark suite. The results are summarized in Table 2. The unwind parameter was used by all the three tools considered in the comparison. The parameters `W`, `U`, and `M` are used by IMU-CSeq, with the meaning as given in Section 6.1. 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 (`dekker`, `lamport`, `peterson`, `szymanski`). The implementations are correct under SC but not under TSO and PSO. All tools find the errors, but because of their small size, Nidhugg outperforms both our prototype and CBMC on these programs.

The second block contains the safe and unsafe versions of one of the fibonacci-benchmarks, in which two worker threads concurrently increase two shared counters, and a main thread checks whether any of the two 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. Techniques such as partial-order reduction do not apply, and several tools struggle to analyze it. Here, our tool has the slight edge over CBMC while Nidhugg is slower than both CBMC and our tool.

The next two blocks contain a benchmark that is correct under SC and TSO but not under PSO, and two benchmarks that originate from industrial code: `parker` models a semaphore-like synchronization class that breaks under TSO [1], and `stack` which was taken from SV-COMP [7]. Here, all tools report the expected results; the performance differences between Nidhugg and CBMC are small, while the performance of our tool could be improved with a better implementation, as it currently transforms each file nearly 20 times, each time requiring parsing and unparsing.

[1] Note that this refers to the SV-COMP15 version and results, which did not include the *ldv-races* sub-category. For SV-COMP16 we submitted (under the label MU-CSeq) a hybrid tool that uses IMU for the shown sub-categories, and the unchanged MU-CSeq for the *pthread-wmm-** sub-categories.

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	CBMC	NIDHUGG	bug?	files	IMU-CSeq	CBMC	NIDHUGG
dekker	52	1	2	0	0	5	•	1	0.76	0.26	**0.04**	•	1	0.76	0.24	**0.04**
lamport	78	1	2	0	0	5	•	1	0.97	0.33	**0.05**	•	1	0.97	0.26	**0.04**
peterson	40	1	3	0	0	5	•	1	0.67	0.28	**0.06**	•	1	0.68	0.23	**0.04**
szymanski	57	1	3	0	0	5	•	1	0.84	0.37	**0.11**	•	1	0.84	0.28	**0.08**
fib_longer_unsafe	30	6	7	0	0	10	•	1	2.10	**1.89**	8.89	•	1	**2.50**	9.79	11.93
fib_longer_safe	30	6	7	0	0	10		1	**4.75**	13.10	41.85		1	**3.90**	20.96	60.94
pgsql	47	1	2	0	0	5		1	1.92	**0.03**	0.07	•	1	0.69	0.22	**0.04**
parker	110	1	2	0	0	5	•	1	1.22	0.35	**0.06**	•	1	1.21	0.26	**0.05**
stack_unsafe	110	2	2	1	2	5	•	1	1.46	0.45	**0.05**	•	1	1.44	0.38	**0.05**
litmus_safe (avg)	297K	1	6	1	0	10		5526	1.20	**0.17**	2.35		4835	1.06	**0.15**	6.65
litmus_unsafe (avg)		1	6	1	0	10	•	277	1.67	**0.16**	3.86	•	968	1.28	**0.12**	1.58

The last two lines show the average values over 5803 litmus tests for WMMs. For TSO, both our tool and CBMC successfully identified the 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 and IMU-CSeq both find only 968 unsafe ones. Since both tools agree, we suspect an error in CBMC. Here, symbolic methods are faster, and Nidhugg has two timeouts.

Safestack. We have conducted further experiments on a real world benchmark, Safestack [12], which is a lock-free stack implementation designed for weak-memory models. Safestack is written in C++ but we manually translated it into C, providing simulation functions for the C11 atomic functions used in the test, and have conducted the experiments with 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 [35], and other approaches based on BMC) [30]. The only tool we are aware of that can automatically find a genuine counter-example is Lazy-CSeq [16]. It requires a minimum of 3 loop unwindings and 4 rounds of computation to expose a bug. This actually shows that the error is quite deep which explains why other approaches based on explicit handling of interleaving fail. Both Nidhugg and CBMC failed to find the error with the given timeout but IMU-CSeq was able to find it also for a TSO- and PSO-semantics, respectively. IMU-CSeq required approx. 3.5 minutes and 1.5GB of memory to find the error under TSO, and approx. 17 minutes and 1.8GB of memory to find the error under PSO.

8 Related Work

The need for a general and reusable framework to accommodate different weak memory models in the analysis of programs has been identified in earlier papers. In [2], the verification algorithm works on a generic relaxed memory model that can be refined into actual memory models by adding constraints. The BMC approach from [4] allows to handle different memory models by adding a conjunct to the formula. Our work differs from these both in the scope and the techniques. In particular, we 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. All the interaction between the two parts is through the API of the shared memory abstraction.

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 [23,31] also with weak memory model semantics [5].

In addition to the already cited work, other marginally related recent papers that have dealt with the verification of concurrent programs under weak memory model semantics are [8,9,33].

We have implemented our IMU-based approach to target BMC backends and used the modules from MU-CSeq [16] to sequentialize the input programs for our experiments. MU-CSeq implements an efficient eager sequentialization of concurrent programs that works well with BMC backends. A hybrid prototype tool combining IMU-CSeq and MU-CSeq has won the gold medal at SV-COMP16 in the Concurrency-category [7].

The reachability analysis used in our algorithm is bounded on the number of writes per shared memory location. This is an orthogonal bounding parameter with respect to the well-known bounded context-switching [28]. The idea of sequentialization was originally proposed by Qadeer and Wu [29] but became popular with the first scheme for an arbitrary but bounded number of context switches given by Lal and Reps [23]. Several implementations and algorithms have been developed since then (see [14,22,10,20,19]).

9 Conclusions

In this paper 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 the communication (i.e., shared memory) concerns of multi-threaded programs, without loosing the efficiency of existing approaches. To achieve this goal, 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, and we have described efficient implementations SMAs for SC, TSO, and PSO that are based on the idea of individual memory-location unwindings. These implementations have allowed us to instantiate our approach into an efficient BMC-based bug-finding tool, and we have show experimentally that it compares well with existing tools.

We plan to enhance IMU-CSeq with the coarse-grained unwinding scheme [32] to handle programs with larger number of writes. We are also investigating other concurrency models like message passing, which should fit into our framework as well; here, the message buffers play the same role as the storage buffers in TSO.

References

1. P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. F. Sagonas. Stateless model checking for TSO and PSO. In Baier and Tinelli [6], pages 353–367.
2. T. Abe and T. Maeda. A general model checking framework for various memory consistency models. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops, Phoenix, AZ, USA, May 19-23, 2014*, pages 332–341. IEEE Computer Society, 2014.
3. J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In M. Felleisen and P. Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 512–532. Springer, 2013.
4. J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In N. Sharygina and H. Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2013.
5. M. F. Atig, A. Bouajjani, and G. Parlato. Getting rid of store-buffers in TSO analysis. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 99–115. Springer, 2011.
6. C. Baier and C. Tinelli, editors. *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*. Springer, 2015.
7. D. Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016). In Chechik and Raskin [11], pages 887–904.
8. A. Bouajjani, G. Calin, E. Derevenetc, and R. Meyer. Lazy TSO reachability. In A. Egyed and I. Schaefer, editors, *Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9033 of *Lecture Notes in Computer Science*, pages 267–282. Springer, 2015.
9. S. Burckhardt, R. Alur, and M. M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In J. Ferrante and K. S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 12–21. ACM, 2007.
10. S. Chaki, A. Gurfinkel, and O. Strichman. Time-bounded Analysis of Real-time Systems. In P. Bjesse and A. Slobodová, editors, *FMCAD*, pages 72–80. FMCAD Inc., 2011.
11. M. Chechik and J. Raskin, editors. *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*. Springer, 2016.
12. G. Chen, H. Jin, D. Zou, B. B. Zhou, Z. Liang, W. Zheng, and X. Shi. Safestack: Automatically patching stack-based buffer overflow vulnerabilities. *IEEE Trans. Dependable Sec. Comput.*, 10(6):368–379, 2013.
13. M. B. Cohen, L. Grunske, and M. Whalen, editors. *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. IEEE, 2015.

14. B. Fischer, O. Inverso, and G. Parlato. Cseq: A sequentialization tool for C - (competition contribution). In N. Piterman and S. A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 616–618. Springer, 2013.
15. A. Horn and D. Kroening. On partial order semantics for sat/smt-based symbolic encodings of weak memory concurrency. In S. Graf and M. Viswanathan, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 35th IFIP WG 6.1 International Conference, FORTE 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings*, volume 9039 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2015.
16. O. Inverso, T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato. Lazy-cseq: A context-bounded model checking tool for multi-threaded c-programs. In Cohen et al. [13], pages 807–812.
17. O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato. Bounded model checking of multi-threaded C programs via lazy sequentialization. In A. Biere and R. Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 585–602. Springer, 2014.
18. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.
19. S. La Torre, P. Madhusudan, and G. Parlato. Analyzing Recursive Programs Using a Fixed-point Calculus. In M. Hind and A. Diwan, editors, *PLDI*, pages 211–222. ACM, 2009.
20. S. La Torre, P. Madhusudan, and G. Parlato. Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *LNCS*, pages 477–492. Springer, 2009.
21. S. La Torre, P. Madhusudan, and G. Parlato. Model-checking parameterized concurrent programs using linear interfaces. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 629–644. Springer, 2010.
22. A. Lal, S. Qadeer, and S. K. Lahiri. A Solver for Reachability Modulo Theories. In P. Madhusudan and S. A. Seshia, editors, *CAV*, volume 7358 of *LNCS*, pages 427–443. Springer, 2012.
23. A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.
24. L. Lamport. On the proof of correctness of a calendar program. *Commun. ACM*, 22(10):554–556, 1979.
25. J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981.
26. J. Moreira and J. R. Larus, editors. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*. ACM, 2014.
27. A. Morrison and Y. Afek. Temporally bounding TSO for fence-free asymmetric synchronization. In Ö. Özturk, K. Ebcioglu, and S. Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 45–58. ACM, 2015.
28. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In N. Halbwachs and L. D. Zuck, editors, *Tools and Algorithms for the Construction and Anal-*

ysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2005.

29. S. Qadeer and D. Wu. KISS: keep it simple and sequential. In W. Pugh and C. Chambers, editors, *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, pages 14–24. ACM, 2004.
30. P. Thomson, A. F. Donaldson, and A. Betts. Concurrency testing using schedule bounding: an empirical study. In Moreira and Larus [26], pages 15–28.
31. E. Tomasco, O. Inverso, B. Fischer, S. La Torre, and G. Parlato. Verifying concurrent programs by memory unwinding. In Baier and Tinelli [6], pages 551–565.
32. E. Tomasco, T. L. Nguyen, O. Inverso, B. Fischer, S. La Torre, and G. Parlato. Mu-cseq 0.4: Individual memory location unwindings - (competition contribution). In Chechik and Raskin [11], pages 938–941.
33. H. Wehrheim and O. Travkin. TSO to SC via symbolic execution. In N. Piterman, editor, *Hardware and Software: Verification and Testing - 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, Proceedings*, volume 9434 of *Lecture Notes in Computer Science*, pages 104–119. Springer, 2015.
34. N. Zhang, M. Kusano, and C. Wang. Dynamic partial order reduction for relaxed memory models. In D. Grove and S. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 250–259. ACM, 2015.
35. M. Zheng, M. S. Rogers, Z. Luo, M. B. Dwyer, and S. F. Siegel. CIVL: formal verification of parallel programs. In Cohen et al. [13], pages 830–835.

$$
\begin{array}{lcl}
P & ::= & \texttt{init}(\,);\,(\mathit{type}\ f\,(\langle \mathit{dec}, \rangle^*)\ \{\,(\mathit{dec};)^*\,\mathit{stm}\,\}\,)^* \\
\mathit{dec} & ::= & \mathit{type}\ z \mid \mathit{type}{*}\ p \\
\mathit{type} & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{void} \\
\mathit{stm} & ::= & \mathit{seq} \mid \mathit{conc} \mid \{\,\langle \mathit{stm}; \rangle^*\,\} \\
\mathit{seq} & ::= & \texttt{assume}(b) \mid \texttt{assert}(b) \mid x = e \mid f(\langle e, \rangle^*) \mid \texttt{return}\ e \\
 & & \mid \texttt{if}(b)\ \texttt{then}\ \mathit{stm}\ \texttt{else}\ \mathit{stm} \mid \texttt{while}(b)\ \texttt{do}\ \mathit{stm} \\
\mathit{conc} & ::= & p = \texttt{address}(y,t) \mid p = \texttt{malloc}(e,t) \\
 & & \mid x = \texttt{read}(y,t) \mid x = \texttt{ind\_read}(p,t) \mid \texttt{write}(y,x,t) \mid \texttt{ind\_write}(p,x,t) \\
 & & \mid t = \texttt{create}(f,t) \mid \texttt{join}(t,t) \mid \texttt{terminate}(t) \\
 & & \mid \texttt{fence}(t) \mid \texttt{lock}(m,t) \mid \texttt{unlock}(m,t)
\end{array}
$$

Fig. 4. 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. 4. Terminal symbols are set in typewriter font. $\langle n\ \texttt{t}\rangle^*$ represents a possibly empty list of non-terminals n that are separated by terminals `t`; x denotes a local variable, y an identifier of a shared variable, p an identifies 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 $q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} q_2 \ldots \xrightarrow{\sigma_d} q_d$ 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 $\sigma_1 \ldots \sigma_d$ 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 A_1 and A_2, denoted $A_1|A_2$, is the standard cross product, i.e., $A_1|A_2$ is the transition system $(Q_1 \times Q_2, \Sigma, \Delta, Q_{0,1} \times Q_{0,1}, F_1 \times F_2)$ 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 Proof of Lemma 2

Lemma 5. $L(\mathcal{M}_{sc}^{mu}) = (L(\mathcal{M}_{sc}))^{\#}$.

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 `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 $\alpha \in (L(\mathcal{M}_{sc}))^{\#}$. $\square$

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 6.1. 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` $\in [0, \mathtt{V}{-}1]$ 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` $\in [0, \mathtt{V}{-}1]$, position `i` $\in [0, \mathtt{W}{-}1]$ and thread identifier `t` $\in [0, \mathtt{T}{-}1]$:

- `th_pos`[`v`][`t`] is the current position of thread `t` in the `v`-LU;
- `last_write`[`v`] stores the position `i` $\in [0, \mathtt{W}{-}1]$ 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.

```
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. 5. IMU initialization.

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. 5).

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. 6.

In the first while-block of Fig. 5, 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. 7.

```
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. 6. 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. 7. Auxiliary functions for `IMU` implementation.

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 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. 8.

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

```
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. 8. Functions `create`, `terminate` and `join`.

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 consistent 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. 9. 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 $v \in [\mathtt{V}-1]$. The implementation for this method is given in Fig. 10.

```
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. 9. Read and write functions.

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

```
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. 10. Functions address and malloc.

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 base address. The malloc procedure is implemented as shown in Fig. 10. 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 mallocP[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. 9). 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. 11. 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. 11. 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`] $\leq$ `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).

```
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. 12. Functions `read`, `fence` and `write` for TSO.

Note that here we slightly diverge from the transition system $\mathcal{M}^{imu}_{tso}$ 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. 11 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 `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. 12.

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. 9.

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`.

```
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. 13. Function `write` for PSO.

In function `write` (Fig. 13), 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];
```