Parameterized Model Checking on the TSO Weak Memory Model

Sylvain Conchon · David Declerck * · Fatiha Zaïdi

Abstract We present an extended version of the Model Checking Modulo Theories (MCMT) framework for verifying parameterized systems under the TSO weak memory model. Our extension relies on three main ingredients: (1) an axiomatic theory of the TSO memory model based on relations over (read, write) events, (2) a TSO-specific backward reachability algorithm and (3) an SMT solver for reasoning about TSO formulas. One of the main originality of our work is a partial order reduction technique that exploits specificities of the TSO memory model. We have implemented this framework in a new version of the Cubicle model checker called Cubicle-$\mathscr{W}$. Our experiments show that Cubicle-$\mathscr{W}$ is expressive and efficient enough to automatically prove safety of concurrent algorithms, for an arbitrary number of processes, ranging from mutual exclusion to synchronization barriers translated from actual x86-TSO implementations.

Keywords Parameterized Model Checking, MCMT, SMT, Weak Memory, Partial Order Reduction

1 Introduction

Concurrent algorithms are usually designed under the sequential consistency (SC) memory model [24] which enforces a global-time linear ordering of (read or write) accesses to shared memories. However, modern multiprocessor architectures do not follow this

Work supported by the French ANR project PARDI (ANR-16-CE25-0006)

Sylvain Conchon
LRI (CNRS & Univ. Paris-Sud), Université Paris-Saclay, F-91405 Orsay
Inria, Université Paris-Saclay, F-91120 Palaiseau
E-mail: sylvain.conchon@lri.fr

David Declerck
OCamlPro, F-91190 Gif-sur-Yvette
E-mail: david.declerck@ocamlpro.com

Fatiha Zaïdi
LRI (CNRS & Univ. Paris-Sud), Université Paris-Saclay, F-91405 Orsay
E-mail: fatiha.zaidi@lri.fr

SC semantics. Instead, they implement several optimizations which lead to relaxed consistency models on shared memory where read and write operations may be reordered. For instance, in x86-TSO [25,26] writes can be delayed after reads due to a store buffering mechanism. Other relaxed models (PowerPC [7], ARM) allow even more types of reorderings.

The new behaviors induced by these models may make out-of-the-shelf algorithms incorrect for subtle reasons mixing interleaving and reordering of events. In this context, finding bugs or proving the correctness of concurrent algorithms is very challenging. The challenge is even more difficult if we consider that most algorithms are *parameterized*, that is designed to be run on architectures containing an arbitrary (large) number of processors and manipulating an arbitrary number of variables.

One of the most efficient technique for verifying concurrent systems is model checking. While this technique has been used to verify parameterized algorithms [14,5,4,19, 11,2] and systems under some relaxed memory assumptions [8,13,12,3,2], hardly any state-of-the-art model checker supports *both* parameterized verification and weak memory models (the only exception we could find is Dual-TSO [2], which supports a finite number of variables).

In this paper, we present a new model checking algorithm for verifying parameterized systems running under the TSO weak memory model. Our approach relies on the Model Checking Modulo Theories (MCMT) framework by Ghilardi and Ranise [21]. This is a symbolic SMT-based model checking technique where logical formulas (expressed in a fragment of first-order logic) are used to represent both transitions and sets of states, and safety properties are verified by backward reachability analysis.

Our extended version of MCMT for weak memory implements a new pre-image computation which takes into account the delays between write and read operations. In order to consider only coherent read/write pairs, our framework relies on a buffer-free memory model inspired by the logical framework of [10] which is implemented as a new theory in its SMT solver. To reduce the state space explosion problem made worse by weak memories, our reachability algorithm embeds a partial order reduction technique that exploits specificities of the TSO memory model.

We have implemented this framework in Cubicle-$\mathscr{W}$ [1,15], the new version of the Cubicle model checker[17,16,18]. Cubicle-$\mathscr{W}$ is a conservative extension which allows the user to manipulate both SC and weak variables. Its relaxed consistency model is similar to x86-TSO: each process has a FIFO buffer of pending store operations whose side effect is to delay the outcome of its memory writes to all processes. Our experiments show that Cubicle-$\mathscr{W}$ is expressive and efficient enough to automatically prove safety of concurrent algorithms, for an arbitrary number of processes, ranging from mutual exclusion to synchronization barriers translated from actual x86 implementations.

The paper is organized as follows. In the next section, we present the axiomatic framework we used for modeling weak memory models. In section 3, we present our extension of the backward reachability analysis of MCMT for weak memories. Section 4 contains the TSO-specific partial order reduction optimization. We give the new syntactic features of Cubicle-$\mathscr{W}$ and some experimental evaluation in Section 5. Then, we conclude in Section 6.

2 Axiomatic weak memory models

Models for reasoning about weak memory generally fall into two categories: operational models, which rely on explicit buffers to simulate the possible behaviors of a program on a given architecture, and axiomatic models, which use *events* and *relations* over these events to describe the possible executions of a program.

For instance, consider the TSO memory model. Under this model, the write operations do not take effect immediately: when a process performs a write, followed by a read, the write might still be pending when the read is performed. An operational model of TSO would typically use write buffers to simulate such behavior: the write would be immediately buffered when encountered, but committed to the shared memory later, possibly after the following read occurs. An axiomatic model of TSO would instead state that there is a write event and a read event, and these two events are not ordered by any relation, meaning they can occur in any order.

In our approach, we make use of on an axiomatic model of weak memory. We adopt the formalism of Alglave *et al.* introduced in [6] and [10]. This framework is generic enough to describe a broad variety of models, though we only introduce the subset we need to cover TSO. The rest of the framework easily fits into our approach.

2.1 Events

The axiomatic approach relies on the use of *execution traces* to describe the semantics of a program. The instructions of a program generate events, described by a unique event identifier, and characterized by their direction (W for a write, R for a read), the process performing the operation, the accessed variable[1] and the read or written value. When an instruction is repeated several times, for instance when it belongs to a loop, it generates as many events as the number of times it is executed, each having a distinct event identifier. Additionally, there exists an initial write event for each variable ; that event not being associated to any process. We call $\mathbb{E}$ the generated set of events.

As an example, we take the following program, involving two processes i and j, in which $R1$ and $R2$ are registers and α is a shared variable initially set to 0:

i	j
$\alpha \leftarrow 1$	$\alpha \leftarrow 2$
$\mathrm{R1} \leftarrow \alpha$	$\mathrm{R2} \leftarrow \alpha$

The events generated by this program are as follows:

$$e_1{:}\mathsf{W}_\alpha{=}0$$

$$e_2{:}\mathsf{W}^i_\alpha{=}1 \qquad e_4{:}\mathsf{W}^j_\alpha{=}2$$

$$e_3{:}\mathsf{R}^i_\alpha{=}? \qquad e_5{:}\mathsf{R}^j_\alpha{=}?$$

The instruction $\alpha \leftarrow 1$ from process i is a write (W) of the value 1 into the shared variable α. As such, it generates the event $e_2{:}\mathrm{W}^i_\alpha{=}1$, where e_2 represents its unique identifier. The instruction $\mathrm{R1} \leftarrow \alpha$ from process i is a read (R) from the shared variable

[1] by variable, we mean a distinct memory location

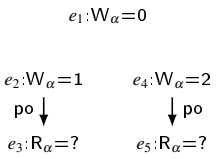
α, whose value we don't (yet) know. As such, it generates the event e_3:R^i_α=?. The same reasoning applies to the two instructions from process j. Finally, as the initial value of α is 0, we add the event e_1:W_α=0, which isn't associated to any process. From now on, to avoid cluttering the schemas, we'll omit the process prefix on events: since events from the same process are displayed in columns, there won't be any ambiguity as to which event belongs to which process.

2.2 Main relations

In order to give a semantics to these events, we then define a number of relations over events, representing different kind of interactions between them.

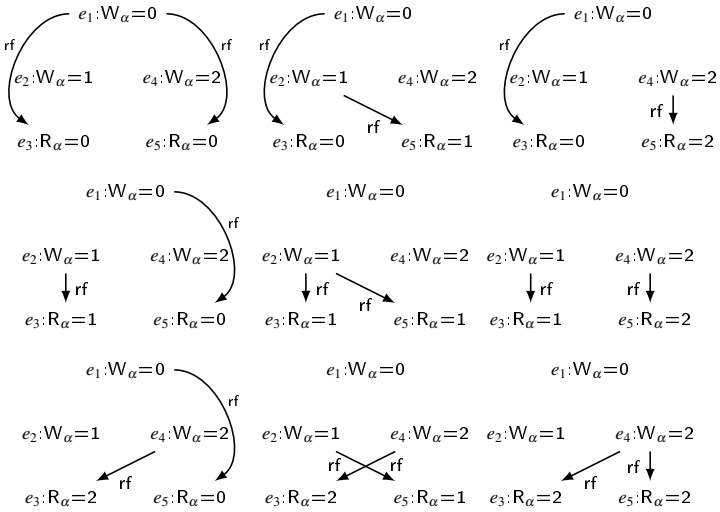
Relation po (program order) All events issued by the same process are ordered by a strict order relation *po*, according to the order in which the instructions that generated them appear in the program's source code. As such, it is a partial relation over the set of all events $\mathbb{E}$, but a total relation over the events of a single process. Naturally, events corresponding to initial writes are not included in this relation.

The events described in the previous schema yield the following *po* relation:



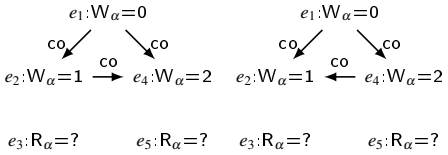
Relation rf (read-from) Every read event must take its value from a unique write event. The *rf* relation allows to link each read to the write that it takes its value from. As a consequence, there exist many different ways to build this relation. Also, a single write event can provide its value to several different read events.

Using the events described earlier, we may build the nine *rf* relations depicted below:



Relation co (coherence) Independently of the underlying memory model, there exists a total order in which all the writes to the same variable become globally visible to all processes, regardless of any local view a process could have of this variable. This can be seen as the order in which these writes are *actually* committed in the shared memory. This order is expressed using the *co* relation. Note that the initial writes are necessarily before all the other writes in this relation.

Still using the events described earlier, we obtain the two *co* relations presented below:



2.3 Candidate and valid executions

Using the relations described above, an execution will be defined by a tuple of the form $(\mathbb{E}, po, rf, co)$. The definition of the various relations being minimalist, they leave a high degree of freedom in the way executions can be built. However, executions created in this manner may not necessarily be correct, so at this stage, we call them *candidate executions*. Additional constraints will allow to filter these executions depending whether they are actually feasible or not.

In order to express the constraints that allow to check whether a candidate execution is actually a valid execution, we need to define new relations, derived from the previous ones.

Relation ppo (preserved program order) One of the main effects of a weak memory model is to relax the order in which the memory operations of a single process are actually performed. In other words, the order of memory operations does not necessarily follows *po*. Different memory models allow different kind of relaxations. Under the TSO model, only the W $\rightarrow$ R order is relaxed, so we define *ppo* as the restriction of *po* to events pairs other than WR (write followed by read).

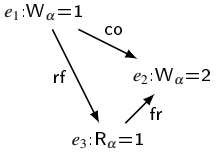
Relation fence Since a weak memory model does not preserve in *ppo* all the event pairs from *po*, the semantics of a program is different compared to the one it has under SC and this can lead to incorrect behaviors. In order to enforce an order between specific pairs of event that were not preserved in *ppo*, the programmer may insert memory fences in his program. Different kind of memory barriers allow to prevent different kind of relaxations. Under TSO, memory fences allow to enforce an ordering between all writes preceding it and all reads following it. The *fence* relation then represents the event pairs from the same process that are separated by a memory barrier.

Relation fr (from-read) We saw earlier that in every execution, there is a total order in which all writes to the same variable become globally visible to all processes (*co*). As a consequence, when a read takes its value from a write, then all the writes that are after

this specific write in the *co* relation also *have* to occur after the read. We express this constraint by the smallest relation *fr* that satisfies the following axiom:

$$\forall e_1, e_2, e_3 \cdot co(e_1, e_2) \wedge rf(e_1, e_3) \rightarrow fr(e_3, e_2) \quad \text{FR}$$

For instance, if we consider two write events e_1 et e_2 such that $co(e_1,e_2)$ and a read event e_3 that takes its value from e_1 (*i.e.* we have $rf(e_1,e_3)$), then we obtain the following *fr* relation:



Relations rfe and rfi (external and internal read-from) The *rf* relation may be split in two sub-relations, depending on whether it relates to events issued by the same process or events issued by distinct processes. We call *rfi* the restriction of *rf* to the pairs of events from the same process, and *rfe* the restriction of *rf* to the pairs of event from distinct processes[2].

Relations fre and fri (external and internal from-read) Similarly, the *fr* relation may be split in two sub-relations: *fri* for the pairs of events from *fr* relative to the same process, and *fre*, for the pairs of events from *fr* relative to distinct processes

Relation ghb (global happens-before) The previous relations allow us to define a strict (partial) order relation *ghb* that represents the order in which the memory operations are performed, from a global viewpoint. All relations do not necessarily contribute to this global order, which moreover depends on the underlying memory model. Under TSO, *ghb* is defined as the smallest relation that satisfies the following axioms:

$$\begin{aligned}
\forall e \cdot \quad & \neg ghb(e,e) && \text{GHB-IR}\\
\forall e_1, e_2, e_3 \cdot ghb(e_1,e_2) \wedge ghb(e_2,e_3) & \rightarrow ghb(e_1,e_3) && \text{GHB-T}\\
\forall e_1, e_2 \cdot ppo(e_1,e_2) & \rightarrow ghb(e_1,e_2) && \text{GHB-PPO}\\
\forall e_1, e_2 \cdot fence(e_1,e_2) & \rightarrow ghb(e_1,e_2) && \text{GHB-FENCE}\\
\forall e_1, e_2 \cdot rfe(e_1,e_2) & \rightarrow ghb(e_1,e_2) && \text{GHB-RFE}\\
\forall e_1, e_2 \cdot co(e_1,e_2) & \rightarrow ghb(e_1,e_2) && \text{GHB-CO}\\
\forall e_1, e_2 \cdot fr(e_1,e_2) & \rightarrow ghb(e_1,e_2) && \text{GHB-FR}
\end{aligned}$$

Note that we use the *ppo* relation and not *po*. Indeed, the event pairs from *po* that are not preserved by the memory model do not contribute to the global ordering of memory events. Also, we consider the *rfe* relation, and not *rf*: when a process performs an intra-process read, that read might occur from the process' write buffer, which does not have any consequences on the global ordering of events. As such, *rfi* doesn't contribute to the *ghb* relation either.

Additionally, in a single process, a read can't take its value from a write that appears after it. Similarly, a write that appears before a read can't occur after it, from the viewpoint of the process performing these two operations. In other words, these constraints

[2] e = external, i = internal

state that the relations *rfi* and *fri must* be compatible with *po*. They can be expressed as follows:

$$\forall e_1, e_2 \cdot rfi(e_1, e_2) \rightarrow po(e_1, e_2) \quad \text{UNIPROCRW}$$
$$\forall e_1, e_2 \cdot fri(e_1, e_2) \rightarrow po(e_1, e_2) \quad \text{UNIPROCWR}$$

Finally, we can define when a candidate execution is considered valid (or feasible). A candidate execution is valid when it allows to derive a *ghb* relation that is a valid partial order, *i.e* an acyclic relation, *and* when the properties UNIPROCRW and UNIPROCWR are verified.

2.4 Taking into account atomic operations

The model we've just introduced lacks features to express atomic accesses to a variable, which is necessary to model atomic *read-modify-write* operations. Moreover, in the context of transition systems, we may also need to simultaneously read or write several different variables at once. In other words, we need a way to express that several events occur "at the same time". To do so, we will rely on two mechanisms.

Firstly, all events of the same kind (read or write) issued by the same process in a single transition will be given the same identifier. As such, an identifier will no longer identify a single event, but a set of events of the same kind, only distinguishable by the variable they access. The relations introduced earlier will thus be defined on these event sets, and will be established when *at least* two events from distinct sets satisfy the conditions to establish the relation. For instance, if e_w identifies a set of writes and e_r a set of reads from a different process, we will have $rf(e_w, e_r)$ if *at least* one of the reads identified by e_r is satisfied by one of the writes identified by e_w. As a consequence, we will need no more than two event identifiers per transition: one for the reads, and one for the writes.

Secondly, when a transition contains both reads and writes, it represents a *read-modify-write* operation. In certain cases, we may want to make this operation atomic. In order to do so, we define an equivalence relation *atom*. The set of all read e_r and all writes e_w from the same transition will be made atomic by injecting the constraint $atom(e_r, e_w)$. That relation may also be used to define synchronization point between events from distinct processes: we will use this possibility under some specific circumstances.

To take into account this new relation, we simply add the following axioms when building *ghb*:

$$\forall e_1, e_2, e_3 \cdot ghb(e_1, e_2) \wedge atom(e_2, e_3) \rightarrow ghb(e_1, e_3) \quad \text{GHB-ATOM-R}$$
$$\forall e_1, e_2, e_3 \cdot atom(e_1, e_2) \wedge ghb(e_2, e_3) \rightarrow ghb(e_1, e_3) \quad \text{GHB-ATOM-L}$$

A complete axiomatization of this model can be found in Appendix A.

3 Model Checking Modulo Theories for Weak Memory Models

Our approach relies on the MCMT framework of Ghilardi and Ranise [21,20] which combines an SMT solver and a backward reachability algorithm. In this section, we first briefly recall how MCMT works, then we give details about how we extend it to support our weak memory model. In the rest of this paper, we assume the usual syntactic

and semantic notions of first-order logic. In particular, we use the symbol $\models$ for the logical entailment relation between sets of formulas. For convenience, disjunctions are represented by sets of formulas.

3.1 Model Checking Modulo Theories

MCMT is a declarative framework for parameterized systems in which (set of) states, transitions and properties are expressed in a particular fragment of first order logic. Systems expressible in this framework are called array-based transition systems because their states can be seen as a set of unbounded arrays (denoted by capital letters $X, Y, \ldots$) whose indexes range over elements of a parameterized domain, called *proc*, of process identifiers (denoted by i, j, p, k). Given an array variable X and a process variable i, we write $X[i]$ for an array access of X at index i. Systems may also contain variables but, from a theoretical point of view, a variable is seen as an array with the same value in all its cells. Arrays may contain integer or real numbers, booleans (or constructors from an enumerative user-defined datatype), or process identifiers.

A parameterized array-based system $\mathscr{S}$ is defined by a triplet $(\mathscr{X}, I, \tau)$ where $\mathscr{X}$ is a set of array symbols, I is a formula describing the initial states of the system and τ is a set of (possibly quantified) formulas, called *transitions*, relating states of $\mathscr{S}$. The formula I is a universal conjunction of literals of the form $\forall \vec{i}. \bigwedge_n \ell_n$ which characterizes the values for some array entries. Each literal ℓ_n is a comparison $(=, \neq, <, \leq)$ between two terms. A term can be a constant (integer, boolean, real, constructor), a process variable (i), an array access $X[i]$. A transition $t \in \tau$ is represented by a formula parameterized by the set of variables before and after the transition ($\mathscr{X}$ and $\mathscr{X}'$) and prefixed by the existentially quantified process variables involved in the transition:

$$t(\mathscr{X}, \mathscr{X}') = \exists \vec{i}. \Delta(\vec{i}) \wedge \gamma(\vec{i}, \mathscr{X}) \\ \wedge \bigwedge_{X' \in \mathscr{X}'} \forall k. \bigwedge_n \Big(C_n(\vec{i}, k, \mathscr{X}) \Rightarrow X'[k] = v_n(\vec{i}, k, \mathscr{X}) \Big)$$

where $\Delta(\vec{i})$ is the conjunction of all disequations between the variables in $\vec{i}$, the formula $\gamma(\vec{i}, \mathscr{X})$ is a conjunction of literals that represents the transition's guard, *i.e.* the conditions that must be met for the transition to be triggered and the conjunction $\bigwedge_n \Big(C_n(\vec{i}, k, \mathscr{X}) \Rightarrow X'[k] = v_n(\vec{i}, k, \mathscr{X}) \Big)$ represents the updated value of each array X defined by a case-split expression where each conjunction of literals $C_n(\vec{i}, k, \mathscr{X})$ and term $v_n(\vec{i}, k, \mathscr{X})$ may depend on $\vec{i}, k$ and $\mathscr{X}$.

Safety properties to be verified on array-based systems are expressed in their negated form as formulas that represent unsafe states. Each unsafe formula $\varphi(\mathscr{X})$ must be a *cube*, i.e., have the form $\exists \vec{k}. (\Delta(\vec{k}) \wedge \bigwedge_m \ell_m(\vec{k}, \mathscr{X}))$, where each literal $\ell_m(\vec{k}, \mathscr{X})$ may depend on $\vec{k}$ and array symbols in $\mathscr{X}$. For a state formula φ and a transition $t \in \tau$, let $pre_t(\varphi)$ be the formula describing the set of states from which a φ-state can be reached in one t-step. The pre-image of a formula $\varphi(\mathscr{X})$ by a transition t is given by:

$$pre_t(\varphi)(\mathscr{X}) = \exists \mathscr{X}'. t(\mathscr{X}, \mathscr{X}') \wedge \varphi(\mathscr{X}')$$

which is proven to be equivalent to a disjunction of cubes (Proposition 3.2 in [20]).

The pre-image *closure* of φ w.r.t a set of transitions τ, denoted by $\text{PRE}^*_\tau(\varphi)$, is defined as follows:

$$\begin{cases} \text{PRE}^0_\tau(\varphi) \triangleq \varphi \\ \text{PRE}^n_\tau(\varphi) \triangleq \bigcup\{pre_t(\psi) \mid \psi \in \text{PRE}^{n-1}_\tau(\varphi), t \in \tau\} \\ \text{PRE}^*_\tau(\varphi) \triangleq \bigcup_{k\in\mathbb{N}} \text{PRE}^k_\tau(\varphi) \end{cases}$$

and the pre-image of a set of formulas V is defined by $\text{PRE}^*_\tau(V) = \bigcup_{\varphi\in V} \text{PRE}^*_\tau(\varphi)$. We also write $\text{PRE}_\tau(\varphi)$ for $\text{PRE}^1_\tau(\varphi)$.

Definition 1 A set of formulas V is said to be *reachable* iff $\text{PRE}^*_\tau(V) \wedge I$ satisfiable.

The core of the analysis of MCMT is a symbolic backward reachability loop (Algorithm 1) that computes the pre-image closure of a (set of) unsafe states.

Given an array-based parameterized system $\mathscr{S} = (\mathscr{X}, I, \tau)$ and a set of unsafe states represented by a cube U, the algorithm maintains two collections of states: Q contains the (unsafe) states to visit (it is initialized with U) and V is filled with the visited states (initially empty). Each iteration of the loop performs the following operations:

1. (*pop*) retrieve and remove a formula φ from Q
2. (*safety test*) check the satisfiability of $\varphi \wedge I$, *i.e.* determine if the states described by φ intersect with the initial states I. If so, the system is declared as *unsafe*
3. (*fixpoint test*) check if $\varphi \models V$ is valid, *i.e.* determine if the states described by φ have already been visited. If so, discard φ and go back to 1
4. (*pre-image computation*) compute the pre-image $\text{PRE}_\tau(\varphi)$ of φ and add these new (set of) states to Q and add φ to V.

If Q is empty at step 1, then all the states space has been explored and the system is declared *safe*. Note that the (non-trivial) fixpoint and safety tests are discharged to an embedded SMT solver.

3.2 MCMT for Weak Memory Models

Our extension of MCMT to weak memory models uses the same procedure, but the logical language and some operations have been extended to reason modulo an axiomatic description of our weak memory model, as described in the previous section.

The first step to define a parameterized weak array-based transition system is to consider given a set $\mathscr{W}$ of weak array symbols (denoted by $\alpha, \beta, \ldots$).

Event identifiers and predicates. In order to reason about the semantics of weak arrays, as defined by the axiomatic model, we introduce the notion of events and new literals to represent read and write operations. For that, we assume given a (countable) set of events $\mathscr{E}$ whose elements are denoted by e letters ($e_1, e_2, \ldots$). A literal of the form $e{:}\texttt{Rd}^i_\alpha[j]$ denotes a read access to the cell $\alpha[j]$ by a process i labeled with an event identifier e. Similarly, literals of the form $e{:}\texttt{Wr}^i_\alpha[j]$ represent write accesses. The value returned by a read (resp. assigned by a write) on a weak array α is given by the term $val_\alpha(e, j)$, where e is the event identifier associated to the operation and j the cell being accessed. We also introduce literals of the form $e{:}\texttt{Fce}^i$ which indicate that a process i

```
1  function BWD(𝒮, U) : begin
2  |  V := ∅;
3  |  push(Q, U);
4  |  while not_empty(Q) do
5  |  |  φ := pop(Q);
6  |  |  if φ ∧ I satisfiable then
7  |  |  |  return unsafe
8  |  |  end
9  |  |  else if φ ⊭ V then
10 |  |  |  V := V ∪ {φ};
11 |  |  |  push(Q, PRE_τ(φ));
12 |  |  end
13 |  end
14 |  return safe
15 end
```

Algorithm 1: MCMT backward reachability algorithm

has a memory barrier on the event e, where e is an event identifier associated to a read by the same process. We define a family of predicates $pending_\alpha(e,j)$ to denote that the read event identified by e on $\alpha[j]$ is pending, that is *not linked* to some write event.

Transitions in weak array-based transition systems. Transitions are also very similar to the ones in MCMT. However, as the semantics of read and write operations depends on the process which performs the operation, we have to define the notion of *current process* in a transition t as the process which performs all reads and writes operations of t. By convention, the identifier of the current process is given as the first parameter of t. Furthermore, the combination of SC and weak arrays is also limited in weak memory models: as SC variables correspond to the (private) registers of a process, it makes no sense for a process to access to the registers of another process. Therefore, operations on SC arrays are restricted to accessing only the cell belonging to the *current process*. In order to enforce such discipline, we write $X[i \leftarrow u]$ for updating only the cell i of an SC array X and leaving the other cells unchanged. Transitions in MCMT for weak memory models have a form very similar to the ones for SC memory models:

$$\begin{aligned} t(\mathscr{X},\mathscr{X}',e_r,e_w) = \exists i,\vec{j}.\, \Delta(i,\vec{j}) \wedge \gamma(i,\vec{j},\mathscr{X},e_r,\mathscr{W}) \wedge \\ \bigwedge_{X'\in\mathscr{X}'} X' = X[i \leftarrow u(i,\vec{j},\mathscr{X},e_r,\mathscr{W})] \wedge \\ \bigwedge_{\alpha\in\mathscr{W}_i} \forall k.\, (\textstyle\bigwedge_m C_m(i,\vec{j},k,\mathscr{X},e_r,\mathscr{W}) \Rightarrow \\ e_w{:}\mathtt{Wr}^i_\alpha[k] \wedge val_\alpha(e_w,k) = v_m(i,\vec{j},k,\mathscr{X},e_r))] \end{aligned}$$

This definition allows us to instantiate a transition t with fresh read and write event identifiers e_r and e_w (it is assumed that t only refers to those events). The first existential variable i is the current process of t (it is also assumed that all read or write predicates in that formula are done by i). The transition's guard $\gamma(i,\vec{j},\mathscr{X},e_r,\mathscr{W})$ and the case-splits $C_m(i,\vec{j},k,\mathscr{X},e_r,\mathscr{W})$ are conjunctions of literals that may contain reads to weak

variables represented by literals of the form $e_r{:}\mathtt{Rd}^i_\alpha[.]$. It is worth noting that any read to a weak variable α must be associated to a literal $pending_\alpha(e,.)$ so as to express that reads introduced in a transition are not currently linked. Updates of weak arrays in a transition only concern a subset $\mathscr{W}_t$ of $\mathscr{W}$. Each update takes the form of case-split formulas where assignments are of the form $e_w{:}\mathtt{Wr}^i_\alpha[k] \wedge val_\alpha(e_w,k) = v_m(i,\vec{j},k,\mathscr{X},e_r)$, where each term v_m may depend on $i,\vec{j},k,\mathscr{X}$ and e_r.

Reachability loop . The reachability loop implemented in our extended framework is based on a new pre-image computation $pre^w_t(\varphi)(\mathscr{X})$ defined below. Unsafe states $\varphi(\mathscr{X})$ are now described by cubes of the following form:

$$\varphi(\mathscr{X}) = \exists\vec{e}.\exists\vec{k}.\Delta(\vec{e}) \wedge \Delta(\vec{k}) \wedge \bigwedge_m \ell_m(\vec{k},\vec{e},\mathscr{X},\mathscr{W})$$

where literals $\ell_m(\vec{k},\vec{e},\mathscr{X},\mathscr{W})$ are defined in a logic extended with various predicates over pairs of event identifiers (e_1,e_2) to represent the relations $po(e_1,e_2)$, $rf(e_1,e_2)$, $co(e_1,e_2)$, $ppo(e_1,e_2)$, $fr(e_1,e_2)$, $fence(e_1,e_2)$, $atom(e_1,e_2)$, $ghb(e_1,e_2)$, etc. The pre-image of an unsafe formula $\varphi(\mathscr{X})$ is of the form:

$$\begin{aligned} pre^w_t(\varphi)(\mathscr{X}) = \exists\mathscr{X}'.\exists e_r,e_w,\vec{e}.t(\mathscr{X},\mathscr{X}',e_r,e_w) \wedge extend\_rels(e_r,e_w,\vec{e}) \\ \wedge\exists\vec{k}.\Delta(\vec{k}) \wedge \bigwedge_m \ell_m(\vec{k},\vec{e},\mathscr{X},\mathscr{W}) \end{aligned}$$

where the *extend_rels* function adds the *po*, *rf* and *co* relations as prescribed by our axiomatic memory model. In particular:

- for each event e in $\vec{e}$, we add $po(e_r,e)$ and $po(e_w,e)$ predicates when the events belong to the same process
- for each write event e in $\vec{e}$, we add $co(e_w,e)$ or $co(e,e_w)$ predicates when the events relate to the same variable ; all possible *co* combinations must be generated
- for each read event e in $\vec{e}$, we add an $rf(e_w,e)$ predicate when we decide that the write event satisfies the read event ; all possible *rf* combinations must be generated

It is worth noting that the pre-image computation generates as many formulas as possible combinations of *co* an *rf* predicates. Also, whenever a formula contains both $rf(e_w,e)$ and $pending_\alpha(e,.)$ the former is replaced by $\neg pending_\alpha(e,.)$, indicating the read has been linked to some write.

Initial states and safety test. The initial states of a parameterized weak array-based system are defined by a universal conjunction I of literals of the form $\forall\vec{i}.\forall\vec{e}.\bigwedge_m \ell_m(\vec{i},\vec{e})$ which characterizes the values of some weak and SC arrays. Literals about SC arrays are as in MCMT. The initializations of weak arrays take the form of *pending* read events. For instance, the initialization of all cells of an array α to 0 takes the following form:

$$\forall i,j.\forall e.\ e{:}\mathtt{Rd}^i_\alpha[j] \wedge val_\alpha(e,j) = 0 \wedge pending_\alpha(e)$$

As it is defined, the formula I cannot be coherent with unsafe states that contain reads linked to some writes. Indeed, any instantiation of I will immediately contradict the literals characterizing read operations linked to some writes, represented by a sub-formula $R(i,j,e,\alpha)$ of the form $e : \mathtt{Rd}^i_\alpha[j] \wedge val_\alpha(e,j) = v \wedge \neg pending_\alpha(e)$.

As a consequence, to be correct, the safety test on line 6 of Algorithm 1 is modified to check satisfiability of $I \wedge filter(\varphi)$, where *filter* is a function that removes all sub-formulas R from φ.

SMT solving. Finally, we assume that the embedded SMT solver involving in the backward reachability analysis is given the axioms of our model, allowing to build the derived relations and check for the validity of executions, as shown in Appendix A.

3.3 Example

As an example, we consider an extension of the simple program from the previous section to an arbitrary number of processes. This variant has two weak variables (α and β) and three SC arrays: PC, that contains the program counter of each process, R_1 (resp. R_2) a register used to store the value of α (resp. β). Initially, α and β both contain 0 and the registers R_i are assumed to contain an arbitrary value, but not 0. We assume given three user-defined constructors $\mathtt{L}_1$, $\mathtt{L}_2$ and $\mathtt{L}_3$ used to represent the locations of each process (initially, each program counter contains $\mathtt{L}_1$).

$$\forall i.\forall e_1,e_2.\ PC[i]=\mathtt{L}_1 \wedge \quad \text{INIT}$$
$$e_1{:}\mathtt{Rd}^i_\alpha \wedge pending_\alpha(e_1) \wedge val_\alpha(e_1)=0 \wedge$$
$$e_2{:}\mathtt{Rd}^i_\beta \wedge pending_\beta(e_2) \wedge val_\beta(e_2)=0$$

$$\exists i_1,i_2.\ i_1\neq i_2 \wedge PC[i_1]=\mathtt{L}_3 \wedge PC[i_2]=\mathtt{L}_3 \wedge \quad \text{UNSAFE}$$
$$R_1[i_1]=0 \wedge R_2[i_2]=0$$

$$\exists i.\exists e.\ PC[i]=\mathtt{L}_1 \wedge \quad \text{TRANS11}$$
$$e{:}\mathtt{Wr}^i_\alpha \wedge val_\alpha(e)=1 \wedge PC'[i]=\mathtt{L}_2$$

$$\exists i.\exists e.\ PC[i]=\mathtt{L}_2 \wedge \quad \text{TRANS12}$$
$$e{:}\mathtt{Rd}^i_\beta \wedge R'_1[i]=val_\beta(e) \wedge pending_\beta(e) \wedge PC'[i]=\mathtt{L}_3$$

$$\exists i.\exists e.\ PC[i]=\mathtt{L}_1 \quad \text{TRANS21}$$
$$e{:}\mathtt{Wr}^i_\beta \wedge val_\beta(e)=1 \wedge PC'[i]=\mathtt{L}_2$$

$$\exists i.\exists e.\ PC[i]=\mathtt{L}_2 \quad \text{TRANS22}$$
$$e{:}\mathtt{Rd}^i_\alpha \wedge R'_2[i]=val_\alpha(e) \wedge pending_\alpha(e) \wedge PC'[i]=\mathtt{L}_3$$

Let's consider the *unsafe* formula. After pre-image by TRANS22, we obtain (changes are underlined):

$$\exists i_1,i_2,e_1.\ i_1\neq i_2 \wedge$$
$$PC[i_1]=\mathtt{L}_3 \wedge \underline{PC[i_2]=\mathtt{L}_2} \wedge$$
$$R_1[i_1]=0 \wedge \underline{val_\alpha(e_1)=0} \wedge$$
$$\underline{e_1{:}\mathtt{Rd}^{i_2}_\alpha \wedge pending_\alpha(e_1)}$$

We now compute the pre-image by TRANS12, which gives:

$$\exists i_1,i_2,e_1,e_2.\ i_1\neq i_2 \wedge e_1\neq e_2 \wedge$$
$$\underline{PC[i_1]=\mathtt{L}_2} \wedge PC[i_2]=\mathtt{L}_2 \wedge$$
$$\underline{val_\beta(e_2)=0} \wedge val_\alpha(e_1)=0 \wedge$$
$$e_1{:}\mathtt{Rd}^{i_2}_\alpha \wedge pending_\alpha(e_1) \wedge$$
$$\underline{e_2{:}\mathtt{Rd}^{i_1}_\beta \wedge pending_\beta(e_2)}$$

When computing the pre-image by TRANS21, we have to consider every possibility to satisfy or not the read e_2 with the new write e_3, hence we have two possible formulas. If the write satisfies the read, we have:

$$\begin{aligned}\exists i_1,i_2,e_1,e_2,e_3.\ & i_1\neq i_2\wedge e_1\neq e_2\wedge e_1\neq e_3\wedge e_2\neq e_3\wedge\\ & PC[i_1]=\mathtt{L}_2\wedge \underline{PC[i_2]=\mathtt{L}_1}\wedge\\ & val_\beta(e_2)=0\wedge val_\alpha(e_1)=0\wedge\\ & e_1{:}\mathtt{Rd}_\alpha^{i_2}\wedge pending_\alpha(e_1)\wedge\\ & e_2{:}\mathtt{Rd}_\beta^{i_1}\wedge\neg pending_\beta(e_2)\wedge\\ & \underline{e_3{:}\mathtt{Wr}_\beta^{i_2}\wedge val_\beta(e_3)=1}\\ & \underline{po(e_3,e_1)\wedge rf(e_3,e_2)\wedge val_\beta(e_3)=val_\beta(e_2)}\end{aligned}$$

This formula contains an obvious contradiction ($val_\beta(e_3)=val_\beta(e_2)$ is false because the values are different), so it is discarded. The other possibility is simply that the write e_3 does not satisfy the read e_2, and we have:

$$\begin{aligned}\exists i_1,i_2,e_1,e_2,e_3.\ & i_1\neq i_2\wedge e_1\neq e_2\wedge e_1\neq e_3\wedge e_2\neq e_3\wedge\\ & PC[i_1]=\mathtt{L}_2\wedge \underline{PC[i_2]=\mathtt{L}_1}\wedge\\ & val_\beta(e_2)=0\wedge val_\alpha(e_1)=0\wedge\\ & e_1{:}\mathtt{Rd}_\mathtt{X}^{i_2}\wedge pending_\alpha(e_1)\wedge\\ & e_2{:}\mathtt{Rd}_\beta^{i_1}\wedge pending_\beta(e_2)\wedge\\ & \underline{e_3{:}\mathtt{Wr}_\beta^{i_2}\wedge val_\beta(e_3)=1}\\ & \underline{po(e_3,e_1)}\end{aligned}$$

We can then compute the pre-image by TRANS11, which for the same reason only produces a single valid formula:

$$\begin{aligned}\exists i_1,i_2,e_1,e_2,e_3,e_4.\ & i_1\neq i_2\wedge\\ & e_1\neq e_2\wedge e_1\neq e_3\wedge e_1\neq e_4\wedge\\ & e_2\neq e_3\wedge e_2\neq e_4\wedge e_3\neq e_4\wedge\\ & \underline{PC[i_1]=\mathtt{L}_1}\wedge PC[i_2]=\mathtt{L}_1\wedge\\ & \underline{val_\beta(e_2)=0\wedge val_\alpha(e_1)=0}\wedge\\ & e_1{:}\mathtt{Rd}_\alpha^{i_2}\wedge pending_\alpha(e_1)\wedge\\ & e_2{:}\mathtt{Rd}_\beta^{i_1}\wedge pending_\beta(e_2)\wedge\\ & e_3{:}\mathtt{Wr}_\beta^{i_2}\wedge val_\beta(e_3)=1\\ & \underline{e_4{:}\mathtt{Wr}_\alpha^{i_1}\wedge val_\alpha(e_4)=1}\\ & po(e_3,e_1)\wedge\underline{po(e_4,e_2)}\end{aligned}$$

This formula intersects with the formula INIT which we described earlier, thus, there is a path from the initial states to the dangerous states and the system is declared unsafe.

4 A TSO-specific partial order reduction technique for an efficient analysis

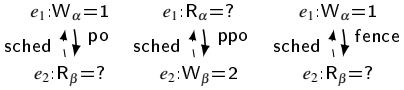
While generic, the analysis scheme presented in the previous section may lead to a high number of states being built, due to the many ways the *co* and *rf* relations can be built. In this section, we propose a TSO-specific partial order reduction technique that allows to drastically reduce the exploration state space.

This technique relies on the fact that our backward reachability algorithm can be seen as an implicit (backward) scheduling of a system's transitions. We choose to make this scheduling explicit, using a strict order relation *sched* that totally orders the events according to the order in which the instructions that generated them are executed. We build this relation during the backward analysis: any new event encountered will be considered scheduled before all the events already in *sched*.

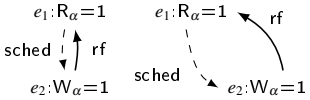
We also define the notion of compatibility with a scheduling:

Definition 2 A relation r is compatible with a scheduling *sched* if there does not exist any pair of events (e_1, e_2) such that $sched(e_1, e_2)$ and $r(e_2, e_1)$ are both true.

For instance, the *po* relation and any relation based upon it, *i.e. ppo* and *fence*, are necessarily compatible with the scheduling, since they match with the order in which the program's instructions are executed. As such, the following executions are invalid with respect to *sched*:



Moreover, since a read can only take its value from a write that was scheduled before, we also have that *rf* must be compatible with *sched*, which forbids the following executions:



However, we note that the *co* relation may be ordered independently from the scheduling. Indeed, because of the "theoretical" write buffers in TSO, two writes ordered in *sched* may be committed to the shared memory in the opposite order.

Yet, we will show that in practice, we can *actually* consider only *co* pairs that are *compatible* with the scheduling, without loosing any feasible execution. In the end, this yields a very powerful partial order reduction technique, since the *extend_rels* function only has to build the base relations of our model according to the scheduling. Note however that the derived relations, in particular *fr* and *ghb*, do not have to be compatible with *sched* (otherwise it would not make sense to use such an axiomatic model and we would just be in the SC case).

4.1 Correctness of our partial reduction technique

In order to prove the correctness of our partial reduction approach, we first redefine *ghb* so as to isolate the relations that depend on the scheduling, as well as the *co* relation. We thus define a new relation *hb* as the smallest relation that satisfies the following axioms:

$$\begin{aligned}
\forall e \cdot \quad & \neg hb(e,e) && \text{HB-IR}\\
\forall e_1,e_2,e_3 \cdot\ & hb(e_1,e_2) \wedge hb(e_2,e_3) \rightarrow hb(e_1,e_3) && \text{HB-T}\\
\forall e_1,e_2 \cdot\ & ppo(e_1,e_2) \rightarrow hb(e_1,e_2) && \text{HB-PPO}\\
\forall e_1,e_2 \cdot\ & fence(e_1,e_2) \rightarrow hb(e_1,e_2) && \text{HB-FENCE}\\
\forall e_1,e_2 \cdot\ & rfe(e_1,e_2) \rightarrow hb(e_1,e_2) && \text{HB-RFE}\\
\forall e_1,e_2 \cdot\ & co(e_1,e_2) \rightarrow hb(e_1,e_2) && \text{HB-CO}
\end{aligned}$$

Then, the *ghb* relation is redefined as the smallest relation that satisfies the following axioms:

$$\begin{aligned}
\forall e \cdot \quad & \neg ghb(e,e) && \text{GHB-IR}\\
\forall e_1,e_2,e_3 \cdot\ & ghb(e_1,e_2) \wedge ghb(e_2,e_3) \rightarrow ghb(e_1,e_3) && \text{GHB-T}\\
\forall e_1,e_2 \cdot\ & hb(e_1,e_2) \rightarrow ghb(e_1,e_2) && \text{GHB-HB}\\
\forall e_1,e_2 \cdot\ & fr(e_1,e_2) \rightarrow ghb(e_1,e_2) && \text{GHB-FR}
\end{aligned}$$

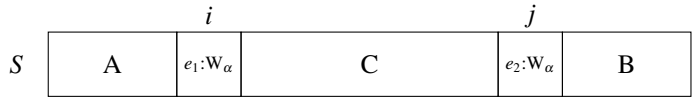
We will now show that, for any *co* relation that belongs to a *valid* execution, there exists a scheduling that is compatible with that *co* relation. We first state and demonstrate the intermediary lemmas we use, before proving this theorem.

Definition 3 Let $\mathbb{E}$ be an event set and *sched* a scheduling of all the events in $\mathbb{E}$. We call scheduling sequence a sequence S of length $l = |\mathbb{E}|$ containing once and only once each and every event of $\mathbb{E}$ and such that $\forall i, j \in \{1..l\} \cdot i < j \leftrightarrow sched(S[i], S[j])$.

Lemma 1 *For every execution $\mathscr{E} = (\mathbb{E}, po, rf, co)$ and every scheduling sched of a program $\mathscr{P}$, for every pair of write events (e_1, e_2) issued by two different processes and such that $sched(e_1, e_2)$ is true, if $hb(e_1, e_2)$ is false, then the scheduling sequence S corresponding to sched can be split into two sub-sequences S_1 and S_2 such that:*

- $\exists i \cdot S_1[i] = e_1$
- $\exists j \cdot S_2[j] = e_2$
- $\forall k_1, k_2 \cdot i \leq k_1 \wedge k_2 \leq j \rightarrow \neg hb(S_1[k_1], S_2[k_2])$
- $\forall k_1, k_2 \cdot i \leq k_1 \wedge k_2 \leq j \rightarrow \neg po(S_1[k_1], S_2[k_2])$

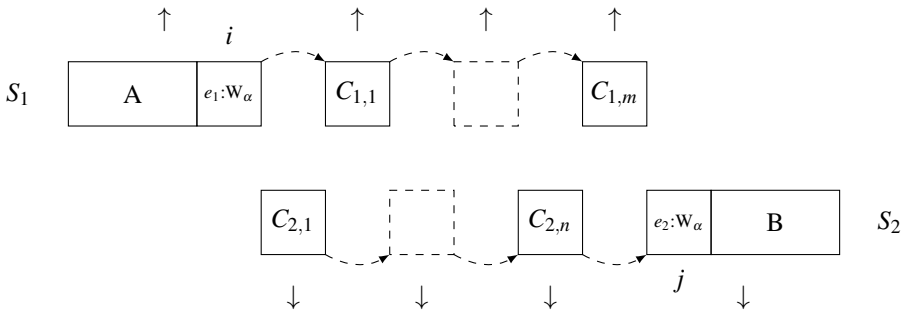
Proof. Let S be the scheduling sequence corresponding to *sched*. In this sequence, we focus on two write events to the same variable, $S[i] = e_1$ and $S[j] = e_2$, issued by two different processes and such that $i < j$ and $hb(S[i], S[j])$ is false. The following schema depicts this sequence and the relative position of the two write events $S[i]$ and $S[j]$:



Since $hb(S[i], S[j])$ is false, we know that there does not exist any index k such that $hb(S[i], S[k])$ and $hb(S[k], S[j])$, otherwise we would have $hb(S[i], S[j])$. We can then propose to split S into two *sub-sequences* S_1 and S_2 such that, for every index k:

- if $k \leq i$ then $S[k] \in S_1$
- if $k \geq j$ then $S[k] \in S_2$
- if $i < k < j$ and $(S[i], S[k]) \in (hb \cup po)^+$ then $S[k] \in S_1$
- if $i < k < j$ and $(S[k], S[j]) \in (hb \cup po)^+$ then $S[k] \in S_2$
- if none of the previous rules applies, $S[k]$ may belong either to S_1 or S_2, but every event in the same case must belong to the same sub-sequence, hence we arbitrarily decide that $S[k] \in S_1$; in other words: if $i < k < j$ and $(S[i], S[k]) \notin (hb \cup po)^+$ and $(S[k], S[j]) \notin (hb \cup po)^+$ then $S[k] \in S_1$

The following schema depicts the resulting splitting:



The rules we used to create this splitting imply that no event from C_1 is before an event from C_2 in the *hb* and *po* relations, otherwise those events would have to be in the same sub-sequence, which would also imply that e_1 and e_2 would be in the same sub-sequence (by transitivity), which contradicts the construction rules.

We can then pack these two sub-sequences as follows:

As a consequence, we now have two sub-sequences S_1 and S_2 such that:

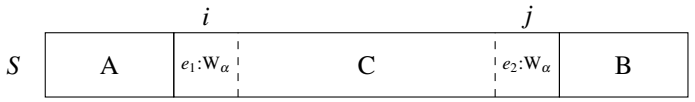
- $\exists i \cdot S_1[i] = e_1$
- $\exists j \cdot S_2[j] = e_2$
- $\forall k_1, k_2 \cdot i \leq k_1 \wedge k_2 \leq j \rightarrow \neg hb(S_1[k_1], S_2[k_2])$
- $\forall k_1, k_2 \cdot i \leq k_1 \wedge k_2 \leq j \rightarrow \neg po(S_1[k_1], S_2[k_2])$

□

Lemma 2 *For every execution $\mathscr{E} = (\mathbb{E}, po, rf, co)$ and every scheduling sched of a program $\mathscr{P}$, for every pair of write events (e_1, e_2) issued by two different processes and such that $sched(e_1, e_2)$ is true, if $co(e_2, e_1)$ is also true, then there* necessarily *exists another scheduling $sched'$ such that:*

- *$sched'(e_2, e_1)$ is true*
- *for every pair of events (e_3, e_4) such that $hb(e_3, e_4)$ and $sched(e_3, e_4)$ are both true, $sched'(e_3, e_4)$ is true*

Proof. Let S be the scheduling corresponding to *sched*. In this sequence, we focus on two write events to the same variable, $S[i] = e_1$ and $S[j] = e_2$, issued by two different processes and such that $i < j$ and $co(S[j], S[i])$ is true. The following schema depicts this sequence and the relative position of the two write events $S[i]$ et $S[j]$:

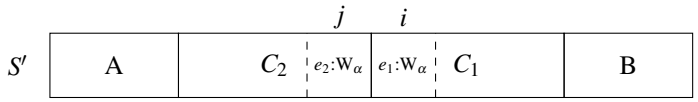


Since $co(S[j], S[i])$ is true, then necessarily $hb(S[j], S[i])$ is true, hence $hb(S[i], S[j])$ is false. We may then use Lemma 1, which allows us to split the sequence S into two sub-sequences S_1 and S_2 such that:

- $\exists i \cdot S_1[i] = e_1$
- $\exists j \cdot S_2[j] = e_2$
- $\forall k_1, k_2 \cdot i \leq k_1 \wedge k_2 \leq j \rightarrow \neg hb(S_1[k_1], S_2[k_2])$
- $\forall k_1, k_2 \cdot i \leq k_1 \wedge k_2 \leq j \rightarrow \neg po(S_1[k_1], S_2[k_2])$

The following schema illustrate such splitting:

For every pair of events (e_{C_1}, e_{C_2}) such that $e_{C_1} \in C_1$ and $e_{C_2} \in C_2$, we have that $hb(e_{C_1}, e_{C_2})$ and $po(e_{C_1}, e_{C_2})$ are both false (by definition of the splitting produced by Lemma 1). As such, (e_{C_1}, e_{C_2}) may freely be scheduled in one way or the other without contradicting *hb* nor *po*. As a consequence, any scheduling of the events of C_1 with the events of C_2 constitutes a valid scheduling, since the relative order of events from C_1 and C_2 is preserved in the resulting scheduling. In particular, we may reschedule all the events from C_2 before the events from C_1, as depicted in the following schema:



That sequence S' constitutes a new scheduling $sched'$ in which we do have both $sched'(e_2, e_1)$ and $co(e_2, e_1)$, and for every pair of events (e_3, e_4) such that $hb(e_3, e_4)$ and $sched(e_3, e_4)$ are both true, $sched'(e_3, e_4)$ is true.

□

Theorem 1 *For every execution $\mathscr{E} = (\mathbb{E}, po, rf, co)$ and every scheduling sched of a program $\mathscr{P}$, there exists a scheduling sched′ such that co is compatible with sched′.*

Proof. By induction on the number of event pairs in *co* that are incompatible with *sched*. Let *n* be the number of event pairs (e_1, e_2) such that $co(e_2, e_1)$ and $sched(e_1, e_2)$ are both true.

Base case If $n = 0$, then $sched' = sched$ and the theorem is true

Inductive case If $n > 0$, then there exist a pair of write events (e_1, e_2) such that $co(e_2, e_1)$ and $sched(e_1, e_2)$ are both true. Then, according to Lemma 2, there exists another scheduling $sched'$ such that $sched'(e_2, e_1)$ is true and for every event pair (e_3, e_4) such that $hb(e_3, e_4)$ and $sched(e_3, e_4)$ are both true, $sched'(e_3, e_4)$ is true. As a consequence, all event pairs that were already both in *co* and *sched* still are in $sched'$, and a pair of events that was in *co* but not in *sched* is now in $sched'$. We then substitute the scheduling *sched* by the scheduling $sched'$. We now have one less event pair such that $co(e_2, e_1)$ and $sched(e_1, e_2)$ are both true ; in other words, *n* has decreased.

□

4.2 A new TSO-specific backward strategy to build *ghb*

Using the additional knowledge brought by the scheduling *sched*, together with the *co*/*sched* compatibility property shown in the previous section, we can now build the different relations more efficiently. In fact, we can now even compute *ghb* directly, instead of computing *po*, *fence*, *co* and *rf* and letting the SMT solver derive *ghb*. As a consequence, we remove all the relation predicates except *ghb* and *atom*, and remove all the weak memory axioms from the solver. We also perform the *ghb* acyclicity test outside the solver to make it more efficient.

We give a new strategy to build *ghb*, that replaces the *extend_rels* function from the pre-image. In the following, we call a new event an event generated on a new iteration of the reachability algorithm, and an old event an event that was already known before the iteration (as if the time flowed backwards). The new events are thus before the old ones in *sched*. We then establish the *ghb* building rules as follows:

- a new read e_r will be before any old event e from the same process in *ghb*, according to GHB-PPO
- a new write e_{w1} will be before any old write e_{w2} from the same process in *ghb*, according to GHB-PPO
- a new write e_w will be before any old read e_r from the same process in *ghb* if they are separated by a memory barrier, according to GHB-FENCE
- a new write e_{w1} will be before any old write e_{w2} on the same variable in *ghb*, according to GHB-CO and using the compatibility property between *co* and *sched*
- a new write e_w will be before any old read e_r from a different process that it satisfies in *ghb*, according to GHB-RFE
- a new write e_w will be after any old read e_r from a different process that it does not satisfy in *ghb*, according to GHB-FR
- a new read e_r will be before any old write e_w on the same variable in *ghb*, according to GHB-FR

Building *ghb* in this manner also guarantees that the UNIPROCRW axiom is always verified: a new read e_r will never be able to take its value from an old write e_w, thus we will never have $po(e_r, e_w)$ and $rf(e_w, e_r)$. As such, we do not need to explicitly check UNIPROCRW.

We can also save us the burden of explicitly checking UNIPROCWR by wisely choosing the *rf* pairs that we build: when we discover a new write e_w, that write must satisfy all the reads e_r from the same process that are not yet satisfied. Indeed, if we have $po(e_w, e_r)$ but not $rf(e_w, e_r)$ that means the read e_r has to take its value from another write that has not been discovered yet. When this new write e_{w2} will be discovered, we will necessarily have $co(e_{w2}, e_w)$ (by the compatibility property of *co* and *sched*), and by FR we will also have $fr(e_r, e_w)$, which contradicts $po(e_w, e_r)$.

5 Experimental evaluation

The algorithm described earlier has been implemented in Cubicle-$\mathscr{W}$, an extension of the Cubicle model checker for weak memory models. Its concrete syntax extends Cubicle's with new constructs for manipulating weak memories, following the logic syntax given in Section 3. The reader can refer to [17] for the description of Cubicle's input language.

Variable and array declarations can be prefixed by the keyword `weak` for defining weak memories.

```
weak var X : int
weak array A[proc] : bool
```

Transitions must now have *at least* one parameter which represents the process that performs the operations. This parameter is identified using the `[.]` notation. For instance, in the following example, the parameter `[i]` of transition `t1` represents the process performing all read/write operations on `X`, `A[i]` and `A[j]` when `t1` is triggered.

```
transition t1 ([i] j)
requires { X = 42 && A[i] = False }
{ A[j] := False }
```

Even if there is no use of parameter `[i]` in transitions' guards and actions, this parameter is still mandatory, as in the transition `t2` below, to indicate which process performs the operations.

```
transition t2 ([i]) { X := 42 }
```

As we implement a weak memory model, we have to allow enforcing the global visibility of a write operation, using a *memory barrier*. In Cubicle-$\mathscr{W}$, barriers are provided as a new built-in predicate `fence()`. When used in the guard of a transition, `fence` is true only when the (theoretical) write buffer of the parameter `[i]` of the transition is empty. For instance, if a process executes `t2` then the following transition `t3`:

```
transition t3 ([i]) requires { fence() }{ ... }
```

the `fence` predicate in `t3`'s guard ensures that the effect of all previous assignments done by `i` are visible to all processes after `t3`. Note that `fence` is not an action: it does not force buffers to be flushed on memory, but just *waits* for a buffer to be empty. As a consequence, it can only be used in a guard.

Implicit memory barriers are also activated when a transition contains both a read and a write to weak variables (not necessarily the same). For instance, the execution of the following transition `t4` guarantees that the buffer of process `i` is empty *before* and *after* `t4`.

```
transition t4 ([i])
requires { A[i] = False }
{ X := 1 }
```

Because there is no unique view of the contents of weak variables, one can not talk about *the value* of `X`, but rather the value of `X` from the *point of view* of a process `i`, denoted `i@X` in Cubicle-$\mathscr{W}$. This notation is used when describing unsafe states. For instance, in the following formula, a state is defined as unsafe when there exist two (distinct) processes `i` and `j` reading respectively `42` and `0` in the weak variable `X`:

```
unsafe (i j) { i@X = 42 && j@X = 0 }
```

This notation is not used for describing initial states as Cubicle-$\mathscr{W}$ implicitly assumes that *all* processes have the same view of each weak variable in those states. For instance, the following formula defines initial states where, for all processes, `X` equals `0` and all cells of array `A` contain `False`.

```
init (i) { X = 0 && A[i] = False }
```

We have evaluated Cubicle-$\mathscr{W}$ on some classical parameterized concurrent algorithms (available on the tool's webpage [1]). Most of these algorithms are abstractions of real world protocols, expressed with up to eight transitions and up to four weak variables or two unbounded weak arrays. The spinlock example is a manual translation of an actual x86 implementation of a spinlock from the Linux 2.6 kernel. We compared Cubicle-$\mathscr{W}$'s performances with state-of-the-art model checkers supporting the TSO weak memory model, since our model is similar. The model checkers we used are CBMC [9], Trencher [13,12], MEMORAX [3] and Dual-TSO [2]. As most of these tools do not support parameterized systems, we used them on fixed-size instances of our benchmarks and increased the number of processes until we obtained a timeout (or until we reached a high number of processes, *i.e.* 11 in our case). Dual-TSO supports a restricted form of parameterized systems, but does not allow process-indexed arrays, which are often needed to express parameterized programs. When it was possible, we used it on both parameterized and non parameterized versions of our benchmarks.

		Cubicle $\mathscr{W}$	Memorax SB	Memorax PB	Trencher	CBMC Unwind 2	CBMC Unwind 3	Dual TSO
`naive mutex`	US	0.04s [N]	– TO [6] 7m54s [5]	– TO [10] 12m02s [9]	– TO [5] 10.1s [4]	– 23.6s [11] 14.7s [10]	– 5m37s [11] 3m39s [10]	NT [N] TO [6] 1m12s [5]
`naive mutex`	S	0.30s [N]	– TO [5] 23.3s [4]	– TO [11] 2m28s [10]	– TO [6] 54.8s [5]	– TO [5] 2m24s [4]	– TO [3] 19.4s [2]	NT [N] TO [5] 35.7s [4]
`lamport`	US	0.10s [N]	– TO [4] 17.4s [3]	– TO [4] 25.4s [3]	– KO [4] 1.73s [3]	– 7m42s [11] 4m29s [10]	– TO [7] 5m12s [6]	NT [N] TO [6] 13m12s [5]
`lamport`	S	0.60s [N]	– TO [3] 0.14s [2]	– TO [4] 3m02s [3]	– KO [5] 3.37s [4]	– TO [4] 8m39s [3]	– TO [3] 1m55s [2]	NT [N] TO [4] 9.42s [3]
`spinlock` [26]	S	0.07s [N]	– TO [5] 8m51s [4]	– TO [7] 9m52s [6]	– TO [7] 21.45s [6]	– TO [3] 19.58s [2]	– TO [3] 5m08s [2]	TO [N] TO [6] 1m16s [5]
`sense` [23] `reversing barrier`	S	0.06s [N]	– TO [3] 0.34s [2]	– TO [3] 0.09s [2]	– TO [5] 1m58s ☠ [4]	– TO [9] 12m25s [8]	– TO [4] 1m43s [3]	NT [N] TO [3] 0.09s [2]
`arbiter v1` [22]	S	0.18s [N]	– TO [3]	– TO [3]	– KO [6] 4.57s [5]	– TO [7] 12m02s [6]	– TO [4] 44.3s [3]	NT [N] TO [7] 2m45s ☠ [6]
`arbiter v2` [22]	S	13.5s [N]	– TO [3]	– TO [3]	– KO [5] 1.62s [4]	– TO [5] 2m56s [4]	– TO [3]	NT [N] TO [4] 24.2s [3]
`two phase commit`	S	54.1s [N]	– TO [2]	– TO [4] 39.7s [3]	– TO [4] 7.08s ☠ [3]	– TO [11] 12m39s [10]	– TO [11] 13m41s [10]	NT [N] TO [3] 12.3s [2]

The table above gives the running time for each benchmark, with the number of processes between square brackets, where N indicates the parametric case. The second column indicates whether the program is expected to be unsafe (US) or safe (S). Unsafe programs have a second version that was fixed by adding `fence` predicates. ☠ indicates that a tool gave a wrong answer. **KO** means that a tool crashed. **NT** indicates a benchmark that was not translatable to Dual-TSO.

The tests were run on a MacBook Pro with an Intel Core i7 CPU @ 2,9 Ghz and 8GB of RAM, under OSX 10.11.6. The timeout (TO) was set to 15 minutes.

These results show that in spite of the relatively small size of each benchmark, state-of-the-art model checkers suffer from scalability issues, which justifies the use of parameterized techniques. Cubicle-$\mathscr{W}$ is thus a very promising approach to the verification of concurrent programs that are both parameterized and operating under weak memory.

6 Conclusions and Perspectives

We have presented in this paper an extended version of MCMT to verify parameterized systems under the TSO weak memory model. We have defined a new backward reachability analysis by defining a new Pre-image computation. This later relies on an axiomatic model of TSO weak memory model and events that correspond to write and read operations. The axiomatic model enables to not model the buffer memory model and to consider only coherent read/write pairs. To circumscribe the state space explosion, we have embedded in our reachability algorithm a partial order reduction technique that relies on the specific feature of the TSO memory model. We have implemented this theoretical framework in Cubicle-$\mathscr{W}$, a new version of the model checker Cubicle, which is conservative and provides specific constructs that allow to handle both SC and weak variables.

We have exercised our implementation on concurrent algorithms to prove their safety for an unknown number of processes. The experiments range from mutual exclusion algorithms to synchronization barriers translated form their x86 implementations. Experimental results show that the approach is very promising.

Yet, there is still room for improvement in order to tackle larger programs and gain in efficiency. We can adapt the Cubicle's invariant generation mechanism to our weak memory model. As future work, we would also like to add support for more complex weak memory models, such as PSO, PowerPC and ARM.

References

1. Cubicle-$\mathscr{W}$. `http://cubicle.lri.fr/cubiclew/`
2. Abdulla, P.A., Atig, M.F., Bouajjani, A., Ngo, T.P.: The benefits of duality in verifying concurrent programs under TSO. In: 27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada, pp. 5:1–5:15 (2016)
3. Abdulla, P.A., Atig, M.F., Chen, Y., Leonardsson, C., Rezine, A.: Memorax, a precise and sound tool for automatic fence insertion under TSO. In: 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, pp. 530–536 (2013)
4. Abdulla, P.A., Delzanno, G., Henda, N.B., Rezine, A.: Regular model checking without transducers (on efficient verification of parameterized systems). In: Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings, pp. 721–736 (2007)
5. Abdulla, P.A., Delzanno, G., Rezine, A.: Parameterized verification of infinite-state processes with global conditions. In: Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings, pp. 145–157 (2007)
6. Alglave, J.: A shared memory poetics. Ph.D. thesis, University of Paris 7 - Denis Diderot, Paris, France (2010)
7. Alglave, J., Fox, A.C.J., Ishtiaq, S., Myreen, M.O., Sarkar, S., Sewell, P., Nardelli, F.Z.: The semantics of power and ARM multiprocessor machine code. In: Proceedings of the POPL 2009 Workshop on Declarative Aspects of Multicore Programming, DAMP 2009, Savannah, GA, USA, January 20, 2009, pp. 13–24 (2009)
8. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In: 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, pp. 512–532 (2013)
9. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings, pp. 141–157 (2013)

10. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. **36**(2), 7:1–7:74 (2014)
11. Apt, K.R., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett. **22**(6), 307–309 (1986)
12. Bouajjani, A., Calin, G., Derevenetc, E., Meyer, R.: Lazy TSO reachability. In: 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, pp. 267–282 (2015)
13. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: 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, pp. 533–553 (2013)
14. Clarke, E.M., Grumberg, O., Browne, M.C.: Reasoning about networks with many identical finite-state processes. In: Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing, Calgary, Alberta, Canada, August 11-13, 1986, pp. 240–248 (1986)
15. Conchon, S., Declerck, D., Zaïdi, F.: Cubicle-$\mathscr{W}$: Parameterized model checking on weak memory. In: System Descriptions - 9th International Joint Conference, IJCAR 2018, Oxford, United Kingdom, July 14 - 17, 2018, Proceedings (2018)
16. Conchon, S., Goel, A., Krstic, S., Mebsout, A., Zaïdi, F.: Invariants for finite instances and beyond. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013, pp. 61–68 (2013)
17. Conchon, S., Goel, A., Krstic, S., Mebsout, A., Zaïdi, F.: Cubicle: A parallel SMT-based model checker for parameterized systems - tool paper. In: Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings, pp. 718–724 (2012)
18. Conchon, S., Mebsout, A., Zaïdi, F.: Certificates for parameterized model checking. In: FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings, pp. 126–142 (2015)
19. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM **39**(3), 675–735 (1992)
20. Ghilardi, S., Ranise, S.: Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. LMCS **6**(4) (2010)
21. Ghilardi, S., Ranise, S.: MCMT: A model checker modulo theories. In: Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings, pp. 22–29 (2010)
22. Goeman, H.J.M.: The arbiter: an active system component for implementing synchronizing primitives. Fundam. Inform. (1981)
23. Herlihy, M., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann (2008)
24. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers **28**(9), 690–691 (1979)
25. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings, pp. 391–407 (2009)
26. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM **53**(7), 89–97 (2010)

A Axiomatic memory model (with ppo instantiated for TSO)

```
type eid
type proc

(* Model relations *)
logic po: eid,eid -> prop
logic ppo: eid,eid -> prop
logic co: eid,eid -> prop
logic rf: eid,eid -> prop
logic rfe: eid,eid -> prop
logic rfi: eid,eid -> prop
logic fr: eid,eid -> prop
logic fre: eid,eid -> prop
```

```
logic fri: eid,eid -> prop
logic ghb: eid,eid -> prop
logic fence: eid,eid -> prop
logic atom: eid,eid -> prop

(* Helper predicates *)
logic isRd: eid -> prop
logic isWr: eid -> prop
logic evtProc: eid,proc -> prop
logic sameProc: eid,eid -> prop

(* Event predicates/functions, defined for each variable X *)
logic wrX: proc,eid -> prop
logic rdX: proc,eid -> prop
logic valX: proc,eid -> int

(* Helper predicates axioms, defined for each variable X *)
axiom isRd: forall e:eid,p:proc. rdX(p,e) -> isRd(e)
axiom isWr: forall e:eid,p:proc. wrX(p,e) -> isWr(e)
axiom evtProc: forall e:eid,p:proc. rdX(p,e) or wrX(p,e) -> evtProc(e,p)

(* Other helper predicates axioms *)
axiom sameProc:
  forall e1,e2:eid,p:proc. evtProc(e1,p) and evtProc(e2,p) -> sameProc(e1,e2)

(* Preserved program order for TSO *)
axiom ppo:
  forall e1,e2:eid. po(e1,e2) and (isRd(e1) or isWr(e2)) -> ppo(e1,e2)

(* From-read relation *)
axiom fr:
  forall e1,e2,e3:eid (*[rf(e1,e2),co(e1,e3)|fr(e2,e3)]*).
    rf(e1,e2) and co(e1,e3) -> fr(e2,e3)

(* Internal-external relation definitions *)
axiom rfi: forall e1,e2:eid. rf(e1, e2) and sameProc(e1,e2) -> rfi(e1,e2)
axiom rfe: forall e1,e2:eid. rf(e1, e2) and (not sameProc(e1,e2)) -> rfe(e1,e2)
axiom fri: forall e1,e2:eid. fr(e1, e2) and sameProc(e1,e2) -> fri(e1,e2)
axiom fre: forall e1,e2:eid. fr(e1, e2) and (not sameProc(e1,e2)) -> fre(e1,e2)

(* Transitivity axioms *)
axiom poTrans: forall e1,e2,e3:eid. po(e1,e2) and po(e2,e3) -> po(e1,e3)
axiom coTrans: forall e1,e2,e3:eid. co(e1,e2) and co(e2,e3) -> co(e1,e3)

(* Global-happens-before relation definition *)
axiom ghbIr: forall e:eid. not ghb(e,e)
axiom ghbPpo: forall e1,e2:eid. ppo(e1,e2) -> ghb(e1, e2)
axiom ghbFence: forall e1,e2:eid. fence(e1,e2) -> ghb(e1, e2)
axiom ghbRfe: forall e1,e2:eid. rfe(e1,e2) -> ghb(e1, e2)
axiom ghbCo: forall e1,e2:eid. co(e1,e2) -> ghb(e1, e2)
axiom ghbFr: forall e1,e2:eid. fr(e1,e2) -> ghb(e1, e2)
axiom ghbTrans: forall e1,e2,e3:eid. ghb(e1, e2) and ghb(e2,e3) -> ghb(e1, e3)
axiom ghbAtomR: forall e1,e2,e3:eid. ghb(e1, e2) and atom(e2,e3) -> ghb(e1, e3)
axiom ghbAtomL: forall e1,e2,e3:eid. atom(e1, e2) and ghb(e2,e3) -> ghb(e1, e3)

(* Uniproc axioms *)
axiom uniprocRW: forall e1,e2:eid. not (po(e1,e2) and rfi(e2,e1))
axiom uniprocWR: forall e1,e2:eid. not (po(e1,e2) and fri(e2,e1))
```