Parameterized Model Checking Modulo Explicit Weak Memory Models*

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

David Declerck
LRI (CNRS & Univ. Paris-Sud),
Université Paris-Saclay, F-91405 Orsay
Inria, Université Paris-Saclay, F-91120 Palaiseau
david.declerck@u-psud.fr

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

We present a modular framework for model checking parameterized array-based transition systems with explicit access operations on weak memory. Our approach extends the MCMT (Model Checking Modulo Theories) framework of Ghilardi and Ranise [10] with explicit weak memory models. We have implemented this new framework in Cubicle-$\mathscr{W}$, an extension of the Cubicle model checker. The modular architecture of our tool allows us to change the underlying memory model seamlessly (TSO, PSO...). Our first experiments with a TSO-like memory model look promising.

1 Introduction

With the emerging of multi-core architectures, concurrent (or multi-threading) programming is becoming a standard practice for boosting the efficiency of an application. To be as efficient as possible, concurrent programs are designed to be run for an arbitrary number of cores. Unfortunately, in practice, the conception and programming of such parameterized programs is error-prone and hard to debug.

The situation is even worse if we consider that modern computer architectures feature weak memory models in which the different processes of a program may not perceive memory operations to happen in the same order. For instance, under the TSO memory model, write operations made by a process might not be immediately visible to all other processes, while they are instantly visible to the process that performs them. The new behaviors induced by these memory models make it hard to design concurrent programs as one has now to take into account both interleavings and reordering of events.

To help debugging such applications, one can use model checking [7, 4, 3, 9, 6, 2], an efficient formal technique used for verifying safety of parameterized concurrent programs [7]. For instance, one can use Cubicle [8], a model checker for array-based transition systems [10], a restricted class of parameterized systems where states are represented as logical formulas manipulating (unbounded) arrays indexed by process identifiers.

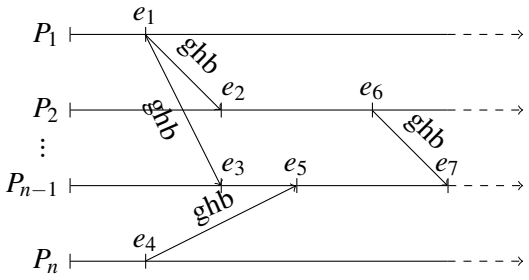
However, the MCMT [11] (Model Checking Modulo Theories) framework underlying Cubicle *implicitly* assumes a sequentially consistent (SC) memory model: the semantics of read and write operations is simply given by the order in which the operations are executed, and the process actually performing the operation is irrelevant (all processes share the same view of the memory).

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

R. Laleau, D. Méry, S. Nakajima, E. Troubitsyna (Eds): Joint Workshop on Handling IMPlicit and EXplicit knowledge in formal system development and Formal & Model-Driven Techniques for Developing Trustworthy Systems.
EPTCS 271, 2018, pp. 48–63, doi:10.4204/EPTCS.271.4

© S. Conchon, D. Declerck & F. Zaïdi
This work is licensed under the
Creative Commons Attribution License.

In this paper, we propose an extension of MCMT [11] with *explicit* read and write operations for weak memories. Our weak memory reasoning is based on the axiomatic framework of Alglave *et al.* [5], which describes the semantics of different weak memory models using *events* and *relations* over these events. More specifically, read and write operations on weak variables give rise to *events*, and according to the dependencies between these events, we build a *global-happens-before* relation (*ghb*) over these events, which orders these events in a global timeline, as depicted in the schema below.



In order to build this happens-before relation (*ghb*), we instrument the backward reachability analysis by (1) generating events for read and write operations on weak memory and (2) building on the fly a *ghb* relation on this events. The coherence of this relation is checked by an SMT solver, modulo a weak memory theory.

In the rest of the paper, we assume a TSO-like memory model[12], though this framework is modular: by changing how the *ghb* relation is built, other weak memory models can be expressed.

2 Preliminaries : axiomatic memory models

Our approach relies on an axiomatic model of weak memory. In this section, we give a brief presentation of this kind of models, based on the formalism of Alglave *et al.* [5]. Our presentation will be oriented towards a TSO-like model, but other models can be expressed using the same formalism.

We consider a concurrent program P composed of n processes $i_1...i_n$, each executing a sequence of instructions $s_1...s_n$. For simplicity, we consider instructions to be either read or write operations on weak variables.

These instructions are mapped to events, which are given a unique event identifier. Note that when an instruction is executed several times (for instance in loops), it will be given a new event identifier each time it is considered. A read event is described by a literal of the form $\mathsf{Rd}_\alpha(e,i)$, where α is the weak variable being read, e is the event identifier, and i is the identifier of the process performing the operation. Similarly, a write event is described by a literal of the form $\mathsf{Wr}_\alpha(e,i)$, where α, e and i have the same meaning as before. The value associated to an event e on a variable α is given by a function $\mathsf{Val}_\alpha(e)$.

Then, depending on the properties of these events, different relations are built, and a *global-happens-before* relation (*ghb*) is derived from them. All these relations order the events using their unique identifiers. The set of events together with the different relations constitutes a candidate execution. If the *ghb* relation is a valid partial order (*i.e.* is acyclic), then the execution is considered valid.

The first relation, *program order (po)*, is implied by the program's source code. It is a total order on all events of a process and it orders the events in the same order as the source code. Under our TSO-like memory model, this relation allows to derive two new relations:

- *preserved program order (ppo)* is a partial order on the events of a process which represents the events that remain ordered under the weak memory semantics ; it is defined as the subset of event pairs in *po* minus the write-read pairs
- *fence* indicates which write-read pairs in *po* are separated by a fence instruction ; it allows to maintain the order between events that would otherwise not be ordered in *ppo*

The next two relations depend on the actual execution of the program:

- *coherence (co)* is a total order on all writes to the same variable ; it represents the order in which the writes are made globally visible
- *read-from (rf)* orders each write with the reads it provides its value to

These two relations allow to derive two new relations:

- *from-read (fr)* indicates reads that occur before some write becomes globally visible ; it is defined as follows: $\forall e_1, e_2, e_2 \cdot rf(e_1, e_2) \wedge co(e_1, e_3) \rightarrow fr(e_2, e_3)$
- *external read-from (rfe)* is defined as the subset of event pairs in *rf* that belong to different processes

Finally, the *ghb* relation is defined as the transitive closure of the union of some of these relations:

$$ghb = (ppo \cup fence \cup co \cup rfe \cup fr)^+$$

This relation represents the order in which events appear to be ordered, from a global viewpoint. The key to the process of finding a feasible execution is thus to determine a *co* and *rf* relation that make the derived *ghb* relation acyclic.

Note that the axiomatic model of Alglave *et al.* specifies an auxiliary check (sequential consistency per variable), that we do not mention here. Indeed, for TSO, the exploration strategy we use (described in the next section) makes this check unnecessary.

Extensions for atomicity

In order to use this framework with array-based transition systems, that may manipulate several different variables in a single transition, we must make some adjustments.

First, we allow several events of the same kind and by the same process to share the same event identifier. This is useful for instance, if we want to have two reads by a process to occur simultaneously, without any other event from another process interfering. Similarly, a process may write to two variables at the same time. This means that the writes will be made globally visible to the other processes simultaneously. This does not require any particular change to the model: the form of literal we use already allows this, and building the relations considers the events independently. For instance, let's consider three processes i, j and k, two event identifiers e_1 and e_2, and the four events $\mathsf{Wr}_\alpha(e_1, i)$, $\mathsf{Wr}_\beta(e_1, i)$, $\mathsf{Rd}_\alpha(e_2, j)$ and $\mathsf{Rd}_\beta(e_3, k)$. The two write events use the same identifier e_1 and belong to the same process i, but write to different variables. Then, we may have both $rf(e_1, e_2)$ and $rf(e_1, e_3)$, even if e_2 and e_3 do not read from the same variable.

Another extension we need is that we must be able to (optionally) specify that a read followed by a write from the same process occur atomically, without any other event from another process interfering. This means that, from a global point of view, the events *appear* to happen simultaneously. For this purpose, we add a symmetric, reflexive and transitive *ghb-equal* relation, and redefine the *ghb* relation as follows:

$$ghb = (ppo \cup fence \cup co \cup rfe \cup fr \cup \textit{ghb-equal})^+ \setminus \textit{ghb-equal}$$

This means that *ghb-equal* is only used to expand the transitive closure of *ghb*, however events that are *ghb-equal* are removed from the actual *ghb* relation (otherwise we wouldn't be able to tell whether *ghb* is acyclic).

3 Weak Memory Array-Based Transition Systems

In our approach, programs are described by parameterized transition systems, *i.e.* systems manipulating variables and process-indexed arrays using guard-action transitions. From the programmer's point of view, the notion of event is irrelevant: accesses to variables and arrays are literally understood as direct accesses. However, during our analysis, we try to build a *ghb* relation on-the-fly, hence, we also need to be able to represent events and the relations over these events. To comply with these different points of view, we define two different logic languages: a description language $\mathscr{L}_{\mathscr{D}}$, in which the events are implicit, and a language $\mathscr{L}_{\mathscr{E}}$ that makes these events explicit. We use translation functions to translate a system from the description language $\mathscr{L}_{\mathscr{D}}$ to the explicit language $\mathscr{L}_{\mathscr{E}}$. To factor out the common parts between these two languages, we define a base language $\mathscr{L}_{\mathscr{B}}$, and we have $\mathscr{L}_{\mathscr{B}} \subset \mathscr{L}_{\mathscr{D}}$ and $\mathscr{L}_{\mathscr{B}} \subset \mathscr{L}_{\mathscr{E}}$.

Base language

We define the base language $\mathscr{L}_{\mathscr{B}}$ as follows:

$$
\begin{aligned}
\textit{const, } c &::= \text{constants} \\
\textit{proc, } i, j, k &::= \text{process variables} \\
x, y, z &::= \text{regular (= non-weak) arrays} \\
\alpha, \beta, \gamma &::= \text{weak variables and arrays} \\
\textit{op} &::= = \mid \neq \mid < \mid \leq \\
\textit{term, } t &::= c \mid i \mid \mathrm{x}[j] \\
\textit{atom, } a &::= t \ \textit{op} \ t \mid \mathsf{true} \mid \mathsf{false} \\
\textit{literal, } l &::= a \mid \neg a \\
\textit{qf\_form, } qff &::= l \mid qff \wedge qff
\end{aligned}
$$

This language defines quantifier free formulas, which are conjunctions of literals (or their negation). A literal is either true, false, or a comparison between two terms. A term is either a constant, a process variable, or the access to a regular array cell.

Description language

We define the description language $\mathscr{L}_{\mathscr{D}}$ as a superset of $\mathscr{L}_{\mathscr{B}}$:

$$
\begin{aligned}
\textit{term, } t &::= \ldots \mid \alpha \mid \alpha[j] \mid i \ @ \ \alpha \mid i \ @ \ \alpha[j] \\
\textit{atom, } a &::= \ldots \mid \mathsf{fence}() \\
\textit{uformula, } uf &::= \forall \vec{j} : \textit{proc}.\ qff \\
\textit{eformula, } ef &::= \exists \vec{j} : \textit{proc}.\ qff
\end{aligned}
$$

The description language includes the base language, and defines new terms for accessing the weak variables. i @ α and i @ $\alpha[j]$ represent accesses to weak variables by a specific process i, while α and $\alpha[j]$ do not explicitly specify the accessing process. The context imposes which of these two forms of access must be used. The language also defines a fence() predicate, which is true for some process when its writes become globally visible to all other processes. Finally, we define formulas prefixed by an existentially or universally quantified process variable.

Explicit language

We define the explicit language $\mathscr{L}_{\mathscr{E}}$ as a superset of $\mathscr{L}_{\mathscr{B}}$:

$$\begin{aligned}
\textit{eid, } e &::= \text{event identifiers} \\
\textit{term, } t &::= ... \mid \mathsf{Val}_\alpha(e) \mid \mathsf{Val}_\alpha(e, j) \\
\textit{atom, } a &::= ... \mid \mathsf{Rd}_\alpha(e, i) \mid \mathsf{Rd}_\alpha(e, i, j) \\
&\quad \mid \mathsf{Wr}_\alpha(e, i) \mid \mathsf{Wr}_\alpha(e, i, j) \\
&\quad \mid \mathsf{fence}(e, i) \mid \mathsf{ghb}(e, e) \mid \mathsf{ghb\text{-}equal}(e, e) \\
\textit{qf_form, qff} &::= ... \mid \textit{qff} \vee \textit{qff} \\
\textit{formula, } f &::= \textit{qff} \mid \forall \vec{x} : type.\, f \mid \exists \vec{x} : type.\, f
\end{aligned}$$

The new Rd_α terms allow to represent the read events, while the Wr_α terms represent the write events. The Val_α terms specify the value associated to events. $\mathsf{ghb}(e, e)$ and $\mathsf{ghb\text{-}equal}(e, e)$ allow to encode the *ghb* and *ghb-equal* relations. The fence predicate indicates that there is a fence before some event e (it is not to be confused with the fence literal in $\mathscr{L}_{\mathscr{D}}$, although the two are related). We also allow for more general formulas by adding disjunctions and quantification over types other than *proc*.

Note that Val_α, Rd_α and Wr_α are defined for every weak variable or array α. Also, Val_α may be considered as regular Cubicle arrays.

Convenience notations

For simplicity, we use the following notations to represent the different sets of variables that we often use:

- X: the set of all regular (*i.e.* non-weak) arrays $(x, y...)$
- $\hat{X}$: the set of all regular arrays $(x, y...)$ and event values $(\mathsf{Val}_\alpha, \mathsf{Val}_\beta ...)$
- A^0: the set of all weak variables
- A^1: the set of all weak arrays
- A: the set of all weak variables and arrays $(A^0 \cup A^1)$
- A^0_t, A^1_t and A_t: similar to the A^0, A^1 and A sets, but restricted to the variables and arrays manipulated by a transition t

From the programmer's point of view, only X and the different A's are relevant. However, the translated formulas in $\mathscr{L}_{\mathscr{E}}$ mainly use $\hat{X}$, since they explicitly manipulate events and their values.

We also often use the two following notations as shortcuts to represent common expressions:

$$\Delta(\vec{e}) \quad = \quad \bigwedge_{(e_1,e_2)\in\vec{e}} e_1 \neq e_2 \qquad \textit{i.e.} \text{ all elements of } \vec{e} \text{ are different}$$

$$\Diamond(\vec{e}) \quad = \quad \bigwedge_{(e_1,e_2)\in\vec{e}} \mathsf{ghb\text{-}equal}(e_1, e_2) \qquad \textit{i.e.} \text{ all elements of } \vec{e} \text{ are } \textit{ghb-equal}$$

Interpretation

The explicit language $\mathscr{L}_{\mathscr{E}}$ is to be interpreted as follows:

$$
\begin{array}{lcll}
\mathscr{M}[c] & = & \mathscr{M}(c) & \\
\mathscr{M}[i] & = & \mathscr{M}(i) & \\
\mathscr{M}[e] & = & \mathscr{M}(e) & \\
\mathscr{M}[x[j]] & = & x^{\mathscr{M}}(\mathscr{M}[j]) & \\
\mathscr{M}[\mathsf{Val}_\alpha(e)] & = & \mathsf{Val}_\alpha{}^{\mathscr{M}}(\mathscr{M}[e]) & \\
\mathscr{M}[\mathsf{Val}_\alpha(e,j)] & = & \mathsf{Val}_\alpha{}^{\mathscr{M}}(\mathscr{M}[e],\mathscr{M}[j]) & \\
\mathscr{M} \models t_1 \; op \; t_2 & = & \mathscr{M}[t_1] \; op \; \mathscr{M}[t_2] & \\
\mathscr{M} \models \mathsf{ghb}(e_1,e_2) & = & (\mathscr{M}[e_1],\mathscr{M}[e_2]) \in \mathsf{ghb}^{\mathscr{M}} & \\
\mathscr{M} \models \mathsf{ghb\text{-}equal}(e_1,e_2) & = & (\mathscr{M}[e_1],\mathscr{M}[e_2]) \in \mathsf{ghb\text{-}equal}^{\mathscr{M}} & \\
\mathscr{M} \models \mathsf{Rd}_\alpha(e,i) & = & (\mathscr{M}[e],\mathscr{M}[i]) \in \mathsf{Rd}_\alpha{}^{\mathscr{M}} & \\
\mathscr{M} \models \mathsf{Rd}_\alpha(e,i,j) & = & (\mathscr{M}[e],\mathscr{M}[i],\mathscr{M}[j]) \in \mathsf{Rd}_\alpha{}^{\mathscr{M}} & \\
\mathscr{M} \models \mathsf{Wr}_\alpha(e,i) & = & (\mathscr{M}[e],\mathscr{M}[i]) \in \mathsf{Wr}_\alpha{}^{\mathscr{M}} & \\
\mathscr{M} \models \mathsf{Wr}_\alpha(e,i,j) & = & (\mathscr{M}[e],\mathscr{M}[i],\mathscr{M}[j]) \in \mathsf{Wr}_\alpha{}^{\mathscr{M}} & \\
\mathscr{M} \models \mathsf{fence}(e,i) & = & (\mathscr{M}[e],\mathscr{M}[i]) \in \mathsf{fence}^{\mathscr{M}} & \\
\mathscr{M} \models \neg a & = & \mathscr{M} \not\models a & \\
\mathscr{M} \models qff_1 \wedge qff_2 & = & \mathscr{M} \models qff_1 \text{ and } \mathscr{M} \models qff_2 & \\
\mathscr{M} \models qff_1 \vee qff_2 & = & \mathscr{M} \models qff_1 \text{ or } \mathscr{M} \models qff_2 & \\
\mathscr{M} \models \forall x : type.\, f & = & \mathscr{M}\{x \mapsto v\} \models f & \text{for all } v \in \mathscr{D}^{type} \\
\mathscr{M} \models \exists x : type.\, f & = & \mathscr{M}\{x \mapsto v\} \models f & \text{for some } v \in \mathscr{D}^{type}
\end{array}
$$

The domain of the model is partitioned according to the types *proc*, representing the process identifiers, *eid*, representing the event identifiers, and *val*, for the different values of variables and arrays. We have $\mathscr{D}_{\mathscr{M}} = \mathscr{D}^{proc} \uplus \mathscr{D}^{eid} \uplus \mathscr{D}^{val}$.

We also define $\mathscr{D}^{ur} \subseteq \mathscr{D}^{eid}$ the subset of event identifiers that contains only unsatisfied read events, *i.e* read events that are not connected to any write, and we have:

$$\forall e_r \in \mathscr{D}^{ur}.\, unsat\_read(e_r)$$

$$
\begin{array}{lcl}
unsat\_read(e_r) & = & \forall i,j,k \in \mathscr{D}^{proc}, \forall e_w \in \mathscr{D}^{eid}.\, e_r = e_w \vee \\
& & \Big(\big(\bigvee_{\alpha \in A^0} (e_r,i) \in \mathsf{Rd}_\alpha \wedge (e_w,j) \in \mathsf{Wr}_\alpha \big) \vee \\
& & \quad \big(\bigvee_{\alpha \in A^1} (e_r,i,k) \in \mathsf{Rd}_\alpha \wedge (e_w,j,k) \in \mathsf{Wr}_\alpha \big) \rightarrow (e_r,e_w) \in \mathsf{ghb} \Big)
\end{array}
$$

Initial state

The initial state describes the constraints on regular arrays and weak variables and arrays. It is described by a formula in $\mathscr{L}_{\mathscr{D}}$, parameterized by a universally quantified process variable. Accesses to weak variables must use the α or $\alpha[j]$ form (i @ α and i @ $\alpha[j]$ are not allowed). Also, the fence() literal may not be used.

We consider an initial state formula I, parameterized by a set of regular variables X and written as follows in the description language $\mathscr{L}_{\mathscr{D}}$:

$$I(X) = \forall j : proc.\, \mathscr{I}(j,X)$$

To obtain the equivalent formula $\tilde{I}$ in $\mathscr{L}_{\mathscr{E}}$, we apply the transformation function $[\![]\!]_I$:

$$\tilde{I}(\hat{X}) = [\![I(X)]\!]_I = \forall i, j : proc, \forall \vec{e_\alpha} : ur. \bigwedge_{\alpha \in A^0} \mathsf{Rd}_\alpha(e_\alpha, i) \wedge \bigwedge_{\alpha \in A^1} \mathsf{Rd}_\alpha(e_\alpha, i, j) \wedge [\![\mathscr{I}(j, X)]\!]_I$$

This function generates a read event for every weak variable or array of the system, not only those actually used in I (however, only those will also have a value associated to). We have one event identifier e_α per weak variable or array α. The event identifiers are chosen in the domain $\mathscr{D}^{ur}$, which restricts the events to the reads that must take their value from the initial state. The process performing the operation is represented by the universally quantified variable i. This means that, in the initial state, any process reading a weak variable or array will obtain the same value. Note that since the resulting formula makes use of event value terms (Val_α), the formula $\tilde{I}$ is parameterized by the set $\hat{X}$ (while I was parameterized by X).

The transformation function $[\![]\!]_I : \mathscr{L}_{\mathscr{D}} \to \mathscr{L}_{\mathscr{E}}$ is defined as follows:

$$\begin{array}{rcll}
[\![qff_1 \wedge qff_2]\!]_I & = & [\![qff_1]\!]_I \wedge [\![qff_2]\!]_I & \\
[\![\neg a]\!]_I & = & \neg [\![a]\!]_I & \\
[\![\mathsf{true}]\!]_I & = & \mathsf{true} & \\
[\![\mathsf{false}]\!]_I & = & \mathsf{false} & \\
[\![t_1 \; op \; t_2]\!]_I & = & [\![t_1]\!]_I \; op \; [\![t_2]\!]_I & \\
[\![\alpha]\!]_I & = & \mathsf{Val}_\alpha(e_\alpha) & \\
[\![\alpha[j]]\!]_I & = & \mathsf{Val}_\alpha(e_\alpha, j) & \\
[\![t]\!]_I & = & t & \text{when } t \neq \alpha \text{ and } t \neq \alpha[j]
\end{array}$$

States

States represent the contraints on regular arrays, events, events values and relations. They are not meant to be manipulated directly by the programmer, so they are expressed in $\mathscr{L}_{\mathscr{E}}$, and are parameterized by a set $\hat{X}$.

A state is represented by a formula of the following form:

$$\varphi(\hat{X}) = \exists \vec{e} : eid. \Delta(\vec{e}) \wedge \Phi(\vec{e}, \hat{X})$$

This means that a state uses a set of event identifiers that are all different. The second part of the formula, Φ, is of the following form:

$$\Phi(\vec{e}, \hat{X}) = \exists \vec{j} : proc. \Delta(\vec{j}) \wedge \phi(\vec{e}, \vec{j}, \hat{X})$$

Similarly to events, the state uses a set of process identifiers that are all different. $\phi(\vec{e}, \vec{j}, \hat{X})$ is a conjunction of literals that actually describes the constraints.

Bad states

Bad states allow to describe the dangerous states of the system in terms of constraints on regular arrays and weak variables and arrays. They are described by a formula in $\mathscr{L}_{\mathscr{D}}$, parameterized by a set of regular variables X. They make use of a set of existentially quantified process variables. Contrary to the initial state, different processes may have different views of some weak variable, so accesses to weak variables must use the i @ α and i @ $\alpha[j]$ form (α and $\alpha[j]$ are not allowed). Also, $\mathsf{fence}()$ may not be used.

We consider a bad state formula Θ, written as follows in the description language $\mathscr{L}_{\mathscr{D}}$:

$$\Theta(X) = \exists \vec{j} : proc.\, \Delta(\vec{j}) \wedge \vartheta(\vec{j}, X)$$

To obtain the equivalent formula $\tilde{\Theta}$ in $\mathscr{L}_{\mathscr{E}}$, we apply the transformation function $[\![]\!]_u$:

$$\tilde{\Theta}(\hat{X}) = [\![\Theta(X)]\!]_u = \exists \vec{e} : eid.\, \Delta(\vec{e}) \wedge \Diamond(\vec{e}) \wedge \theta(\vec{e}, X)$$

Where:

$$\theta(\vec{e}, X) = \exists \vec{j} : proc.\, \Delta(\vec{j}) \wedge [\![\vartheta(\vec{j}, X)]\!]_u^{\vec{e}}$$

The translation function transforms every weak variable or array access to a read event, and generates one event identifier e_i per process.

The translation function $[\![]\!]_u : \mathscr{L}_{\mathscr{D}} \times \vec{eid} \to \mathscr{L}_{\mathscr{E}}$ is defined as follows:

$$\begin{array}{rcll}
[\![qff_1 \wedge qff_2]\!]_u^{\vec{e}} & = & [\![qff_1]\!]_u^{\vec{e}} \wedge [\![qff_2]\!]_u^{\vec{e}} & \\
[\![a]\!]_u^{\vec{e}} & = & [\![a]\!]_{u_E}^{\vec{e}} \wedge [\![a]\!]_{u_V}^{\vec{e}} & \\
[\![\neg a]\!]_u^{\vec{e}} & = & [\![a]\!]_{u_E}^{\vec{e}} \wedge \neg [\![a]\!]_{u_V}^{\vec{e}} & \\
[\![t_1\ op\ t_2]\!]_{u_E}^{\vec{e}} & = & [\![t_1]\!]_{u_E}^{\vec{e}} \wedge [\![t_2]\!]_{u_E}^{\vec{e}} & \\
[\![t_1\ op\ t_2]\!]_{u_V}^{\vec{e}} & = & [\![t_1]\!]_{u_V}^{\vec{e}}\ op\ [\![t_2]\!]_{u_V}^{\vec{e}} & \\
[\![i\ @\ \alpha]\!]_{u_E}^{\vec{e}} & = & \mathsf{Rd}_\alpha(e_i, i) & e_i \in \vec{e} \\
[\![i\ @\ \alpha]\!]_{u_V}^{\vec{e}} & = & \mathsf{Val}_\alpha(e_i) & e_i \in \vec{e} \\
[\![i\ @\ \alpha[j]]\!]_{u_E}^{\vec{e}} & = & \mathsf{Rd}_\alpha(e_i, i, j) & e_i \in \vec{e} \\
[\![i\ @\ \alpha[j]]\!]_{u_V}^{\vec{e}} & = & \mathsf{Val}_\alpha(e_i, j) & e_i \in \vec{e} \\
[\![t]\!]_{u_E}^{\vec{e}} & = & \mathsf{true} & \text{when } t \neq i\ @\ \alpha \text{ and } t \neq i\ @\ \alpha[j] \\
[\![t]\!]_{u_V}^{\vec{e}} & = & t & \text{when } t \neq i\ @\ \alpha \text{ and } t \neq i\ @\ \alpha[j]
\end{array}$$

Note that the function $[\![]\!]_u$ makes use of two sub-functions $[\![]\!]_{u_E}$ and $[\![]\!]_{u_V}$. The first one is used to build the literals describing the events, while the second one is used to build the event value terms.

Transitions

Transitions describe the changes made to regular arrays and weak variables and arrays. They are composed of a guard, representing the conditions that must be satisfied for the transition to be executed, and actions, which may be updates of regular arrays or and updates of weak variables and arrays. They are expressed in the description language $\mathscr{L}_{\mathscr{D}}$, with the restriction that accesses two weak variables must use the α or $\alpha[j]$ form ($i\ @\ \alpha$ and $i\ @\ \alpha[j]$ are not allowed). Also, the $\mathsf{fence}()$ predicate may only be used in the guard.

We consider a transition t, written as follows in the description language $\mathscr{L}_{\mathscr{D}}$:

$$\begin{array}{rcl}
t(X, X') & = & \exists i, \vec{j} : proc.\, \Delta(\vec{j}) \wedge \gamma(i, \vec{j}, X) \wedge \\
& & \bigwedge_{x \in X} x'[i] = \delta_x(i, \vec{j}, X) \wedge \\
& & \bigwedge_{\alpha \in A_t^0} \alpha' = \delta_\alpha(i, \vec{j}, X) \wedge \\
& & \bigwedge_{\alpha \in A_t^1} \bigwedge_{k \in \vec{j}} \alpha'[k] = \delta_\alpha(i, \vec{j}, X)
\end{array}$$

The existentially quantified process i is the process performing the actions. This also means non-weak array terms are restricted to $x[i]$. Note that process i may be equal to some process in $\vec{j}$.

To obtain the equivalent transition $\tilde{t}$ in $\mathscr{L}_{\mathscr{E}}$, we apply the transformation function $[\![]\!]_t$, which must be given two fresh event identifiers e_r and e_w:

$$\tilde{t}(\hat{X},\hat{X}',e_r,e_w) \quad = \quad [\![t(X,X')]\!]_t^{e_r,e_w}$$

Where:

$$\begin{array}{rcl}
[\![t(X,X')]\!]_t^{e_r,e_w} & = & \exists i,\vec{j}: proc.\,\Delta(\vec{j}) \wedge e_r \neq e_w \wedge \mathsf{ghb\text{-}equal}(e_r,e_w) \wedge [\![\gamma(i,\vec{j},X)]\!]_\gamma^{i,e_r} \wedge \\
& & \bigwedge\limits_{x\in X} \Big([\![x'[i] = \delta_x(i,\vec{j},X)]\!]_\gamma^{i,e_r} \wedge (\forall k.k = i \vee x'[k] = x[k]) \Big) \wedge \\
& & \bigwedge\limits_{\alpha\in A_t^0} \mathsf{Wr}_\alpha(e_w,i) \wedge [\![\mathsf{Val}'_\alpha(e_w) = \delta_\alpha(i,\vec{j},X)]\!]_\gamma^{i,e_r} \wedge \\
& & \bigwedge\limits_{\alpha\in A_t^1} \bigwedge\limits_{k\in\vec{j}} \mathsf{Wr}_\alpha(e_w,i,k) \wedge [\![\mathsf{Val}'_\alpha(e_w,k) = \delta_\alpha(i,\vec{j},X)]\!]_\gamma^{i,e_r}
\end{array}$$

This translation ensures the two event identifiers e_r and e_w are different, and links them in the *ghb-equal* relation. The regular array updates are extended so that all array cells different from i receive a value equals to the previous one. The weak variable and array updates generate write events. The translation of the guard and updates is further delegated to the function $[\![]\!]_\gamma : \mathscr{L}_{\mathscr{D}} \times proc \times eid \to \mathscr{L}_{\mathscr{E}}$.

$$\begin{array}{rcll}
[\![qff_1 \wedge qff_2]\!]_\gamma^{i,e_r} & = & [\![qff_1]\!]_\gamma^{i,e_r} \wedge [\![qff_2]\!]_\gamma^{i,e_r} & \\
[\![a]\!]_\gamma^{i,e_r} & = & [\![a]\!]_{\gamma_E}^{i,e_r} \wedge [\![a]\!]_{\gamma_V}^{i,e_r} & \\
[\![\neg a]\!]_\gamma^{i,e_r} & = & [\![a]\!]_{\gamma_E}^{i,e_r} \wedge \neg [\![a]\!]_{\gamma_V}^{i,e_r} & \\
[\![\mathsf{fence}()]\!]_{\gamma_E}^{i,e_r} & = & \mathsf{fence}(e_r,i) & \\
[\![\mathsf{fence}()]\!]_{\gamma_V}^{i,e_r} & = & \mathsf{true} & \\
[\![t_1\ op\ t_2]\!]_{\gamma_E}^{i,e_r} & = & [\![t_1]\!]_{\gamma_E}^{i,e_r} \wedge [\![t_2]\!]_{\gamma_E}^{i,e_r} & \\
[\![t_1\ op\ t_2]\!]_{\gamma_V}^{i,e_r} & = & [\![t_1]\!]_{\gamma_V}^{i,e_r}\ op\ [\![t_2]\!]_{\gamma_V}^{i,e_r} & \\
[\![\alpha]\!]_{\gamma_E}^{i,e_r} & = & \mathsf{Rd}_\alpha(e_r,i) & \\
[\![\alpha]\!]_{\gamma_V}^{i,e_r} & = & \mathsf{Val}_\alpha(e_r,i) & \\
[\![\alpha[j]]\!]_{\gamma_E}^{i,e_r} & = & \mathsf{Rd}_\alpha(e_r,i,j) & \\
[\![\alpha[j]]\!]_{\gamma_V}^{i,e_r} & = & \mathsf{Val}_\alpha(e_r,i) & \\
[\![t]\!]_{\gamma_E}^{i,e_r} & = & \mathsf{true} & \text{when } t \neq \alpha \text{ and } t \neq \alpha[j] \\
[\![t]\!]_{\gamma_V}^{i,e_r} & = & t & \text{when } t \neq \alpha \text{ and } t \neq \alpha[j]
\end{array}$$

4 Backward reachability

Our approach relies on a rather classical backward reachability algorithm (function `BWD` below), whose objective is to check whether there is a possible path from the initial state to the dangerous states. However, the pre-image computation has been extended to produce events and relations, according to our weak memory semantics. The algorithm takes as input a transition system $S = (Q, I, \tau)$ and a cube Θ, where I is a formula describing the initial states of the system, τ is the set of all transitions, and Θ a cube describing the dangerous states. It maintains a set of visited states $\mathscr{V}$ and a working queue of cubes $\mathscr{Q}$.

```
1  function BWD(𝒮, Θ) : begin
2      𝒱 := ∅;
3      push(𝒬, [[Θ]]_u);
4      while not_empty(𝒬) do
5          φ := pop(𝒬);
6          if φ ∧ [[I]]_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
```

We assume the fomulas describing states to be of the form:

$$\varphi(\hat{X}) = \exists \vec{e} : eid.\, \Delta(\vec{e}) \wedge \Phi(\vec{e}, \hat{X})$$

The pre-image of a formula φ with respect to the set of transitions τ (line 11) is given by:

$$\textsc{Pre}_\tau(\varphi)(\hat{X}) = \bigvee_{t \in \tau} \textsc{Pre}_t(\varphi)(\hat{X})$$

The pre-image of a formula φ with respect to a single transition t is given by:

$$\begin{array}{rcl} \textsc{Pre}_t(\varphi)(\hat{X}) & = & \exists \hat{X}', \exists e_r, e_w, \vec{e} : eid.\, \Delta(e_r \cdot e_w \cdot \vec{e}) \wedge \\ & & [\![t(X, X')]\!]_t^{e_r, e_w} \wedge \Phi(\vec{e}, \hat{X}') \wedge \\ & & extend\_ghb(e_r, e_w, \vec{e}) \wedge rffr(X, X', e_w, \vec{e}) \end{array}$$

This pre-image generates two new event identifiers e_r and e_w. We ensure these identifiers to be different from those in φ using the expression $\Delta(e_r \cdot e_w \cdot \vec{e})$. These identifiers are given as parameters to the translation function $[\![]\!]_t$, which ensures the transition only manipulates new events.

The $rffr$ function determines whether the new writes e_w from the transition t satisfy the compatible unsatisfied reads in $\Phi(\vec{e}, \hat{X}')$, and if so orders them appropriately in the *ghb* relation. A read and a write are compatible if they refer to the same variable or the same array with the same parameters (the actual value associated to the events is irrelevant to this notion of compatibility). For each unsatisfied read in $\vec{e}$, either:

- there is a compatible write e_w from the *same* process, so the read MUST take its value from this write
- there is a compatible write e_w from a *different* process, in this case the read MAY take its value from this write ; if it does, e_w is before e_r in *ghb*, otherwise e_r is before e_w in *ghb*
- there is no compatible write ; the read remains unsatisfied

This implies the following logical definition of $rffr$:

$$\begin{aligned} rffr(X,X',e_w,\vec{e}) \quad = \quad & \bigwedge_{\alpha\in A}\bigwedge_{e_r\in\vec{e}}\Big(\exists i,k : proc.\, unsat\_read_\alpha(e_r,i,k,\vec{e}) \rightarrow \\ & \qquad internal\_rffr_\alpha(X,X',e_r,e_w,i,k) \veebar \\ & \qquad external\_rffr_\alpha(X,X',e_r,e_w,i,k) \veebar \\ & \qquad no\_rffr_\alpha(X,X',e_r,e_w,k)\Big) \end{aligned}$$

$$internal\_rffr_\alpha(X,X',e_r,e_w,i,k) \quad = \quad \mathsf{Wr}_\alpha(e_w,i,k) \wedge \mathsf{Val}_\alpha\text{'}(e_r,k) = \mathsf{Val}_\alpha\text{'}(e_w,k)$$

$$\begin{aligned} external\_rffr_\alpha(X,X',e_r,e_w,i,k) \quad = \quad & \exists j : proc.\ j \neq i \wedge \mathsf{Wr}_\alpha(e_w,j,k) \wedge \\ & (\mathsf{Val}_\alpha\text{'}(e_r,k) = \mathsf{Val}_\alpha\text{'}(e_w,k) \wedge \mathsf{ghb}(e_w,e_r) \vee \\ & \ \mathsf{Val}_\alpha\text{'}(e_r,k) = \mathsf{Val}_\alpha(e_r,k) \wedge \mathsf{ghb}(e_r,e_w)) \end{aligned}$$

$$no\_rffr_\alpha(X,X',e_r,e_w,i,k) \quad = \quad (\nexists j : proc.\, \mathsf{Wr}_\alpha(e_w,j,k)) \wedge \mathsf{Val}_\alpha\text{'}(e_r,k) = \mathsf{Val}_\alpha(e_r,k)$$

$rffr$ also relies on the $unsat\_read_\alpha$ function, that allows to determine if some read event e_r is not satisfied by any write event in $\vec{e}$ for some weak variable α:

$$unsat\_read_\alpha(e_r,i,k,\vec{e}) \quad = \quad \mathsf{Rd}_\alpha(e_r,i,k) \wedge \bigwedge_{e_w\in\vec{e}}\Big(e_r = e_w \vee \Big(\forall j : proc.\, \mathsf{Wr}_\alpha(e_w,j,k) \rightarrow \mathsf{ghb}(e_r,e_w)\Big)\Big)$$

The $extend\_ghb$ function extends the ghb relation by adding literals of the form ghb(e_1, e_2) and ghb-equal(e_1, e_2) in the formula, according to the dependencies between the new events e_r and e_w and old events in $\vec{e}$.

$$extend\_ghb(e_r,e_w,\vec{e}) \quad = \quad ppo(e_r \cdot e_w,\vec{e}) \wedge fence(e_w,\vec{e}) \wedge co(e_w,\vec{e}) \wedge fr(e_r,\vec{e})$$

fr adds ghb pairs between the new read(s) e_r and the old writes.

$$fr(e_r,\vec{e}) \quad = \quad \bigwedge_{\alpha\in A}\bigwedge_{e_w\in\vec{e}}\Big(\exists k : proc.\, read\_on_\alpha(e_r,k) \wedge write\_on_\alpha(e_w,k) \rightarrow \mathsf{ghb}(e_r,e_w)\Big)$$

co adds ghb pairs between the new write(s) e_w and the old writes.

$$co(e_{w_1},\vec{e}) \quad = \quad \bigwedge_{\alpha\in A}\bigwedge_{e_{w_2}\in\vec{e}}\Big(\exists k : proc.\, write\_on_\alpha(e_{w_1},k) \wedge write\_on_\alpha(e_{w_2},k) \rightarrow \mathsf{ghb}(e_{w_1},e_{w_2}\Big)$$

$fence$ adds ghb pairs between the new writes(s) e_w and the subsequent reads by the same process separated by a fence predicate.

$$fence(e_w,\vec{e}) \quad = \quad \bigwedge_{e_r\in\vec{e}}\Big(\exists i : proc.\, \mathsf{fence}(e_r,i) \wedge write\_by(e_w,i) \wedge read\_by(e_r,i) \rightarrow \mathsf{ghb}(e_w,e_r)\Big)$$

ppo adds ghb pairs between the new events e_a and the subsequent events e_b by the same process if they are read-read, read-write or write-write pairs.

$$ppo(\vec{e_a},\vec{e_b}) \quad = \quad \bigwedge_{e_1\in\vec{e_a}}\bigwedge_{e_2\in\vec{e_b}}\Big(ppo\_RR(e_1,e_2) \wedge ppo\_RW(e_1,e_2) \wedge ppo\_WW(e_1,e_2)\Big)$$

$$ppo\_RR(e_1,e_2) \quad = \quad \exists i : proc.\, read\_by(e_1,i) \ \wedge \ read\_by(e_2,i) \rightarrow \mathsf{ghb}(e_1,e_2)$$

$$ppo\_RW(e_1,e_2) \;=\; \exists i : proc.\, read\_by(e_1,i) \;\wedge\; write\_by(e_2,i) \rightarrow \mathsf{ghb}(e_1,e_2)$$
$$ppo\_WW(e_1,e_2) \;=\; \exists i : proc.\, write\_by(e_1,i) \;\wedge\; write\_by(e_2,i) \rightarrow \mathsf{ghb}(e_1,e_2)$$

The following $read\_on_\alpha$ and $write\_on_\alpha$ predicates allow to easily check whether some event identifier e corresponds to a read or a write event on a specific weak variable α or array cell $\alpha[k]$.

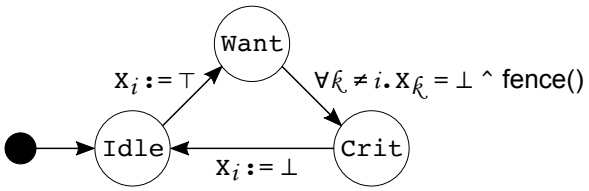
$$read\_on_\alpha(e,k) \;=\; \exists i : proc.\, \mathsf{Rd}_\alpha(e,i) \vee \mathsf{Rd}_\alpha(e,i,k)$$
$$write\_on_\alpha(e,k) \;=\; \exists i : proc.\, \mathsf{Wr}_\alpha(e,i) \vee \mathsf{Wr}_\alpha(e,i,k)$$

Similarly, the *read_by* and *write_by* predicates allow to check whether some event identifier e corresponds to a read or a write event performed by a specific process i.

$$read\_by(e,i) \;=\; \exists k : proc.\, \bigvee_{\alpha\in A^0} \mathsf{Rd}_\alpha(e,i) \vee \bigvee_{\alpha\in A^1} \mathsf{Rd}_\alpha(e,i,j)$$
$$write\_by(e,i) \;=\; \exists k : proc.\, \bigvee_{\alpha\in A^0} \mathsf{Wr}_\alpha(e,i) \vee \bigvee_{\alpha\in A^1} \mathsf{Wr}_\alpha(e,i,j)$$

5 Example

We illustrate our backward reachability algorithm through a simple example. We consider a simple parameterized mutual exclusion algorithm, where each process executes the automaton below.



This automaton is trivially encoded into the following transition system, where the current state is represented by a regular array `PC`, and the shared variables $\mathtt{X}_i$ are expressed as a weak array `X`.

$$t\_req = \exists i : proc.\, PC[i] = Idle \wedge PC'[i] = Want \wedge X'[i] = True$$
$$t\_enter = \exists i : proc.\, PC[i] = Want \wedge \mathsf{fence}() \wedge (\forall k.k = i \vee X[i] = False) \wedge PC'[i] = Crit$$
$$t\_exit = \exists i : proc.\, PC[i] = Crit \wedge PC'[i] = Idle \wedge X'[i] = False$$

The initial state is simply defined as:

$$I = \forall j : proc.\, PC[j] = Idle \wedge X[j] = False$$

The property we would like to check is that no pair of processes i and j can be in state `Crit` at the same time, *i.e.* that the following formula never holds:

$$\Theta = \exists i,j : proc.\, i \neq j \wedge PC[i] = Crit \wedge PC[j] = Crit$$

By applying the translation functions on the system above, we obtain the following event-explicit transition system:

$$\tilde{I} = \forall i,j : proc, \forall e_X : ur.\, PC[j] = Idle \wedge Rd_X(e_X,i,j) \wedge Val_X(e_X,j) = False$$

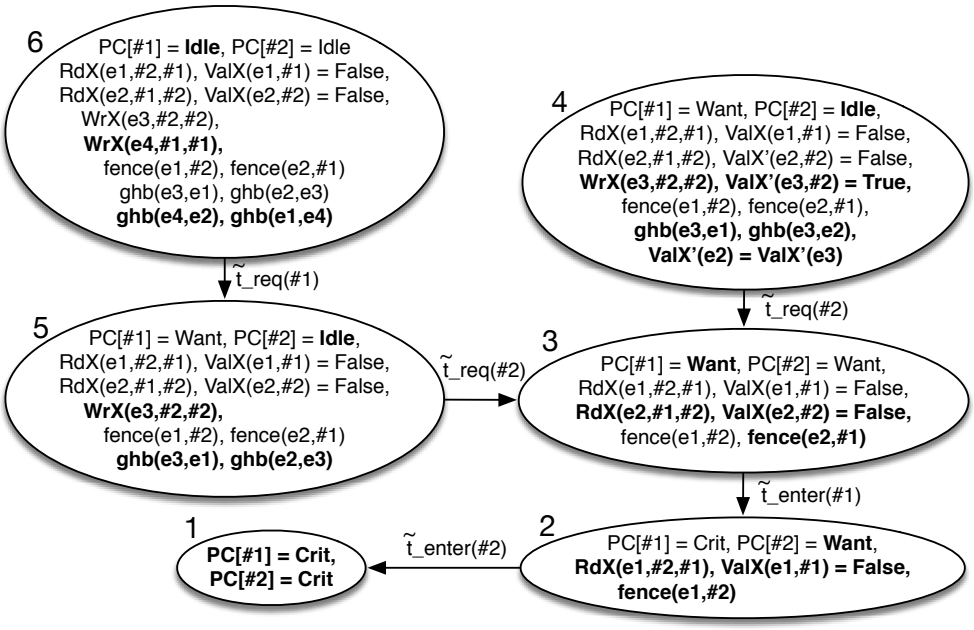
$$\tilde{\Theta} = \exists i,j : proc.\, i \neq j \wedge PC[i] = Crit \wedge PC[j] = Crit$$

$$\begin{aligned}
\tilde{t}_req(e_r,e_w) &= \exists i : proc.\, e_r \neq e_w \wedge \mathsf{ghb\text{-}equal}(e_r,e_w) \wedge \\
&\quad PC[i] = Idle \wedge \\
&\quad PC'[i] = Want \wedge Wr_X(e_w,i,i) \wedge Val'_X(e_w,i) = True
\end{aligned}$$

$$\begin{aligned}
\tilde{t}_enter(e_r,e_w) &= \exists i : proc.\, e_r \neq e_w \wedge \mathsf{ghb\text{-}equal}(e_r,e_w) \wedge \\
&\quad PC[i] = Want \wedge \mathsf{fence}(e_r,i) \wedge \\
&\quad (\forall k.k = i \vee Rd_X(e_r,i,k) \wedge Val_X(e_r,i) = False) \wedge \\
&\quad PC'[i] = Crit
\end{aligned}$$

$$\begin{aligned}
\tilde{t}_exit(e_r,e_w) &= \exists i : proc.\, e_r \neq e_w \wedge \mathsf{ghb\text{-}equal}(e_r,e_w) \wedge \\
&\quad PC[i] = Crit \wedge \\
&\quad PC'[i] = Idle \wedge Wr_X(e_w,i,i) \wedge Val'_X(e_w,i) = False
\end{aligned}$$

The graph below gives a possible exploration of the system's state space by our algorithm. We start by node 1, which represents the formula describing the dangerous states $\tilde{\Theta}$. Then, each node represents the result of a pre-image computation by an instance of a transition. The edges are labeled with the transition name and the process identifier we used to instantiate the existentially quantified process variable in the transition. Remark that formulas in the graph's nodes are implicitly existentially quantified and that a process identifier i is written $\#_i$. Also, to avoid cluttering the graph, we omit event identifiers in the edge labels, we remove the unused event identifiers from the nodes (they do not contribute to *ghb*), and we assume all event identifiers to be different.



We focus on node 3 which results from the pre-image of node 1 by `t_enter(#2)` then `t_enter(#1)`. In this state, both processes have read `False` in `X` (events e_1 and e_2). Also, since there is a memory barrier in `t_enter`, both reads are associated to a `fence` literal. The pre-image of node 3 by `t_req(#2)` introduces a new write event `WrX(e3,#2,#2)` with an associated value `ValX'(e3,#2) = True`. Since there is a memory barrier `fence(e1,#2)` on `e1` by the same process #2, the *extend_ghb* predicate causes

ghb(e3,e1) to be added to the formula. Now, the $rfrf$ predicate dictates that this new write event may or may not satisfy the read e2, so we must consider both cases (node 4 and 5).

In node 4, we consider the case where the write event e3 satisfies the read event e2. As prescribed by the *external_rfrf* predicate, the equality ValX'(e2,#2) = ValX'(e3,#2) is added to the formula, which obviously makes it inconsistent. In node 5, the write event e3 does not satisfy the read event e2, so the value ValX'(e3,#2) is discarded and ghb(e2,e3) is added to the formula, as indicated by the *no_rfrf* predicate. Similarly, the pre-image of node 5 by t_req(#1) yields the formula in state 6 where the new write event e4 does not satisfy the read event e1. Now, the *ghb* relation is not a valid partial order as the sequence $\texttt{ghb}(\texttt{e2},\texttt{e3}), \texttt{ghb}(\texttt{e3},\texttt{e1}), \texttt{ghb}(\texttt{e1},\texttt{e4}), \texttt{ghb}(\texttt{e4},\texttt{e2})$ forms a cyclic relation. Therefore, this state is discarded and the program is declared *safe*.

Remark that if we removed the fence predicate in t_enter, then we would only have $\texttt{ghb}(\texttt{e3},\texttt{e1})$, $\texttt{ghb}(\texttt{e4},\texttt{e2})$ in state 6, which is a valid partial order relation, so the formula would intersect with the initial state and the program would be *unsafe*.

6 Implementation

We have implemented this framework in Cubicle-$\mathscr{W}$ [1] and used it to check the correctness of several parameterized concurrent algorithms on a TSO-like model whose source codes are available on the tool's webpage.

Table 1 gives for each benchmark the number of non-weak arrays, the number of weak variables, the number of weak arrays, the number of transitions and the running time. The S/US next to the benchmark name indicates whether the algorithm is correct or not. Incorrect algorithm have a second version that was fixed by using fence predicates.

Case study		Arrays	Weak Var.	Weak Arr.	Transitions	Time
naive mutex	US	1	0	1	4	0.04s
naive mutex	S	1	0	1	4	0.30s
lamport	US	1	4	0	8	0.10s
lamport	S	1	4	0	8	0.60s
spinlock	S	1	1	0	6	0.07s
sense reversing barrier	S	1	0	1	4	0.06s
arbiter v1	S	2	1	1	7	0.18s
arbiter v2	S	2	0	2	8	13.5s
two phase commit	S	1	1	1	5	54.1s

Table 1: Performance of Cubicle-$\mathscr{W}$

The results shown in this table look promising. Although these benchmarks are of modest size, they are already consider as very challenging for state-of-the-art model checkers for weak memories as they combined parametricity, concurrency and non trivial used of weak memories.

7 Conclusion & Future Work

We have presented in this paper an extension of Model Checking Modulo Theories (MCMT) for model checking parameterized transitions with *explicit* read and write operations on weak memories.

The core of our procedure is a backward reachability algorithm combined with an SMT axiomatic model for reasoning about weak memory. The explicit relaxed consistency model underlying our framework is similar to x86-TSO where the effect of a store operation by a process is delayed (due to a store buffering) to all processes.

We have implemented this framework in Cubicle-$\mathscr{W}$, a conservative extension of the Cubicle model checker. Our first experiments show that our implementation is expressive and efficient enough to prove safety of concurrent algorithms, for an arbitrary number of processes, ranging from mutual exclusion to synchronization barriers.

Immediate future work includes the support for other models, such as PSO. We are also working on the extension of Cubicle's invariant generation mechanism for weak memory. With this mechanism, we should gain in efficiency and be able to tackle even more complex programs. Finally, we also plan to investigate how our framework for weak memories could be extended to reason about programs communicating via channels.

References

[1] `http://cubicle.lri.fr/cubiclew/`.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani & Tuan Phong Ngo (2016): *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, doi:10.4230/LIPIcs.CONCUR.2016.5.

[3] Parosh Aziz Abdulla, Giorgio Delzanno, Noomene Ben Henda & Ahmed Rezine (2007): *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, doi:10.1007/978-3-540-71209-1_56.

[4] Parosh Aziz Abdulla, Giorgio Delzanno & Ahmed Rezine (2007): *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, doi:10.1007/978-3-540-73368-3_17.

[5] Jade Alglave, Luc Maranget & Michael Tautschnig (2014): *Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory*. *ACM Trans. Program. Lang. Syst.* 36(2), pp. 7:1–7:74, doi:10.1145/2627752.

[6] Krzysztof R. Apt & Dexter Kozen (1986): *Limits for Automatic Verification of Finite-State Concurrent Systems*. *Inf. Process. Lett.* 22(6), pp. 307–309, doi:10.1016/0020-0190(86)90071-2.

[7] Edmund M. Clarke, Orna Grumberg & Michael C. Browne (1986): *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, doi:10.1145/10590.10611.

[8] Sylvain Conchon, Amit Goel, Sava Krstic, Alain Mebsout & Fatiha Zaïdi (2012): *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, doi:10.1007/978-3-642-31424-7_55.

[9] Steven M. German & A. Prasad Sistla (1992): *Reasoning about Systems with Many Processes*. *J. ACM* 39(3), pp. 675–735, doi:10.1145/146637.146681.

[10] Silvio Ghilardi, Enrica Nicolini, Silvio Ranise & Daniele Zucchelli (2008): *Towards SMT Model Checking of Array-Based Systems*. In: *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, pp. 67–82, doi:10.1007/978-3-540-71070-7_6.

[11] Silvio Ghilardi & Silvio Ranise (2010): *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, doi:10.1007/978-3-642-14203-1_3.

[12] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli & Magnus O. Myreen (2010): *x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors*. *Commun. ACM* 53(7), pp. 89–97, doi:10.1145/1785414.1785443.