Cubicle-$\mathcal{W}$: Parameterized Model Checking on Weak Memory

Sylvain Conchon[1,2], David Declerck[1,2(✉)], and Fatiha Zaïdi[1]

[1] LRI (CNRS & University Paris-Sud), Université Paris-Saclay, F-91405 Orsay, France
david.declerck@u-psud.fr
[2] Inria, Université Paris-Saclay, F-91120 Palaiseau, France

Abstract. We present Cubicle-$\mathcal{W}$, a new version of the Cubicle model checker to verify parameterized systems under weak memory models. Its main originality is to implement a backward reachability algorithm modulo weak memory reasoning using SMT. Our experiments show that Cubicle-$\mathcal{W}$ 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.

Keywords: Parameterized model checking · MCMT · SMT
Weak memory

1 Introduction

Concurrent algorithms are usually designed under the sequential consistency (SC) memory model [20] which enforces a global-time linear ordering of (read or write) accesses to shared memories. However, modern multiprocessor architectures do not follow this 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 [21,22] writes can be delayed after reads due to a store buffering mechanism. Other relaxed models (PowerPC [6], 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.

One of the most efficient technique for verifying concurrent systems is model checking. While this technique has been used to verify parameterized algorithms [2,4,5,9,12,16] and systems under some relaxed memory assumptions [2,3,7,10,11], hardly any state-of-the-art model checker support *both* parameterized verification and weak memory models [2].

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

D. Galmiche et al. (Eds.): IJCAR 2018, LNAI 10900, pp. 152–160, 2018.
https://doi.org/10.1007/978-3-319-94205-6_11

In this paper, we present Cubicle-$\mathcal{W}$ [1], the new version of the Cubicle [13–15] model checker for verifying safety properties of parameterized array-based transition systems on weak memory. Cubicle-$\mathcal{W}$ 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.

Like Cubicle, Cubicle-$\mathcal{W}$ is based on the MCMT framework of Ghilardi and Ranise [17]. Its core extends the SMT-based backward reachability procedure with 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, Cubicle-$\mathcal{W}$ relies on a buffer-free memory model inspired by the logical framework of [8] which is implemented as a new theory in its SMT solver. Cubicle-$\mathcal{W}$ is an open-source software freely available at http://cubicle.lri.fr/cubiclew.

2 Tool Presentation

The syntax of Cubicle-$\mathcal{W}$ extends Cubicle's with new constructs for manipulating weak memories. The reader can refer to [13] for the description of Cubicle's input language.

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

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

Transitions in Cubicle-$\mathcal{W}$ have the same syntactic guard/action form as in Cubicle and they are also supposed to be executed atomically. The new feature is that they 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 }
```

Note that, as Cubicle-$\mathcal{W}$'s transitions are atomic, having several processes performing reads or writes operations in the same transition would require an unrealistic powerful synchronization mechanism between processes.

The main aspect of our relaxed memory semantics is that, from a global viewpoint, the effect of a write operation on a `weak` memory is not immediately visible to all processes. It is only locally visible to the process that performs it. For instance, if some process `i` executes the transition `t2` above, then `X = 42` is true for `i` after the transition (as the effect of the assignment is immediately locally visible), while all other processes can still read a different value for `X`.

To enforce the global visibility of a write operation, one has to use a *memory barrier*. In Cubicle-$\mathcal{W}$, barriers are provided as a new built-in predicate `fence()`. When used in the guard of a transition, `fence` is true only when the FIFO 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-$\mathcal{W}$. 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-$\mathcal{W}$ 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 }
```

Finally, it is important to note that non weak arrays are restricted to be used only locally by processes: given a non weak array `T`, only `i` can read or write to `T[i]`.

3 Backward Reachability Modulo Weak Memory

The core of Cubicle-$\mathcal{W}$ is an extension of Cubicle's symbolic backward reachability algorithm [13,14]. We first briefly recall how the original Cubicle works, then we give details about our new algorithm.

States in Cubicle are represented by cubes, i.e., formulas of the form $\exists \bar{i}.(\Delta \wedge F)$, where $\bar{i}$ is a set of process variables, Δ is the conjunction of all disequations between the variables in $\bar{i}$ and F is a conjunction of literals. Each literal in F is a comparison ($=, \neq, <, \leq$) between two terms. A term can be a constant (integer, boolean, real, constructor), a process variable (i), a variable (X) or an array access ($A[i]$, where i is a process variable). All process variables in a state are implicitly existentially quantified. Initial states are represented by a universally quantified formula I of the form $\forall \bar{i}.(\Delta \wedge F)$, where Δ and F are as described above.

The core of Cubicle is a symbolic backward reachability loop that maintains two collections of states: $\mathcal{Q}$ contains the states to visit (it is initialized with the states declared as `unsafe`), and $\mathcal{V}$ 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 $\mathcal{Q}$
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 $\varphi \models \mathcal{V}$ 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 $pre(\varphi, t)$ of φ for all instances of transitions t, i.e. determine the set of states that can reach φ in one step by applying t with the processes identifiers `#1`, ..., `#n` as parameters, add these states to $\mathcal{Q}$ and add φ to $\mathcal{V}$.

If $\mathcal{Q}$ 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.

Cubicle-$\mathcal{W}$ uses the same procedure but some operations have been extended to reason modulo an axiomatic description of our weak memory model. This axiomatization uses the notion of *events* to describe weak memory accesses and a global-happens-before (*ghb*) relation defined as a partial order relation over these events. This relation is used to determine if an execution is valid.

Our logic is extended with new literals to represent read and write operations on weak memories. We assume given a (countable) set of events $\mathcal{E}$. A literal of the form $e{:}\mathtt{Rd}_{\mathtt{X}}(i)$ denotes a read access on variable `X` by a process i labeled with an event identifier $e \in \mathcal{E}$. Similarly, literals of the form $e{:}\mathtt{Wr}_{\mathtt{X}}(i)$ represent write accesses. The value returned by a read (resp. assigned by a write) is given by the term $val(e)$, where e is the event identifier associated to the operation. Operations on weak arrays are represented by literals of the form $e{:}\mathtt{Rd}_{\mathtt{A}}(i, j)$ and $e{:}\mathtt{Wr}_{\mathtt{A}}(i, j)$, which represent an access by a process i to the cell j of an array A. Last, there is also literals of the form $e{:}\mathtt{fence}(i)$ which indicate that a process

i has a memory barrier on the event e, where e is an event identifier associated to a read by the same process.

The reachability loop of Cubicle-$\mathcal{W}$ implements a new pre-image computation. At step 4, $pre(\varphi, t)$ is modified so that read and write operations from t give rise to `Rd` and `Wr` literals labeled with fresh event identifiers. These new events are ordered w.r.t the older ones in the *ghb* relation expressed by predicates of the form $ghb(e_1, e_2)$, indicating that event e_1 is *ghb-before* (i.e., occurs before) event e_2. The ghb-ordering of events is built w.r.t. the following rules:

- New read events are *ghb-before* old read and write events from the same process.
- New write events are *ghb-before* old write events from the same process, however they are *ghb-before* old reads events from the same process only if there is a fence on these reads.
- New write events are *ghb-before* all the old write events to the same variable.
- New read events are *ghb-before* all the old write events to the same variable.

Finally, when a memory fence is encountered, a literal e:`fence`(i) is added on all old reads events e which belong to the process i executing the transition.

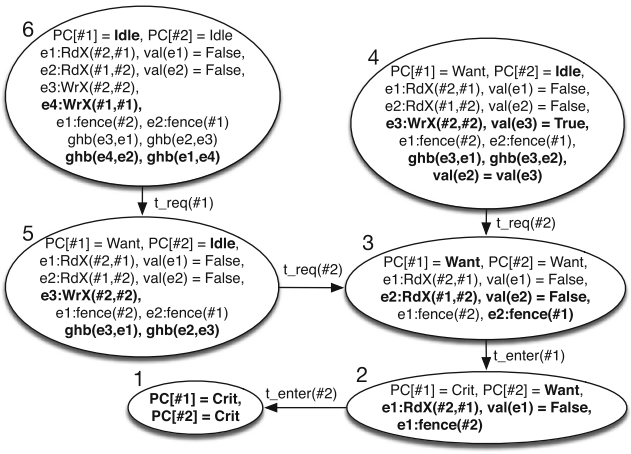
The treatment of write events is also specific when we have to consider the delays introduced by store buffers: when a new write event e is produced, all possible combinations of e with older compatible reads are considered (unlike in SC), as a write operation *may or may not* satisfy subsequent reads. By *compatible read*, we mean a read on the same variable or array cell as the write, though we may also consider the constant values associated to these events in order to obtain a more accurate set of compatible reads. The connection between a write and an older read obeys the following rules:

- When the write event satisfies an old read event from a different process, the write is *ghb-before* the read.
- When the write event does not satisfy an old read event from a different process, the read is *ghb-before* the write.
- When the write and the read events belong to the same process, none of them is considered *ghb-before* the other (unless there is a fence on the read event).

In order to show how our reachability procedure works, we consider the simple parameterized mutual exclusion algorithm and the exploration graph given below. Cubicle-$\mathcal{W}$ starts with the `unsafe` formula in node 1. Then, each node represents the result of a pre-image computation by an instance of a transition (denoted by the label of the edge). Remark that formulas in the graph's nodes are implicitly existentially quantified and that a process identifier i is written $\#_i$.

```
type loc = Idle | Want | Crit

weak array X[proc] : bool
array PC[proc] : loc

init (i) {PC[i] = Idle && X[i] = False}

unsafe (i j) {PC[i] = Crit && PC[j] = Crit}
```

```
transition t_req ([i])
requires { PC[i] = Idle }
{ X[i] := True ; PC[i] := Want }

transition t_enter ([i])
requires { PC[i] = Want && fence() &&
           forall_other k. X[k] = False }
{ PC[i] := Crit }

transition t_exit ([i])
requires { PC[i] = Crit }
{ X[i] := False ; PC[i] := Idle }
```



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 `e3:WrX(#2,#2)` with an associated value `val(e3) = True`. Since there is a memory barrier `e1:fence(#2)` on `e1` by the same process `#2`, we add `ghb(e3,e1)` in the formula. Now, 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, event `e3` satisfies `e2`. The equality `val(e2) = val(e3)` is then added to the formula which obviously makes it inconsistent. In node 5, the write `e3` does not satisfy the read `e2`, then the value `val(e3)` is discarded and `ghb(e2,e3)` is added to the formula. Similarly, the pre-image of node 5 by `t_req(#1)` yields the formula in state 6 where the new write `e4` does not satisfy the read `e1`. Now, the *ghb* relation is not a valid partial order as the sequence $\texttt{ghb(e2,e3)}, \texttt{ghb(e3,e1)}, \texttt{ghb(e1,e4)}, \texttt{ghb(e4,e2)}$ 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 $\texttt{ghb(e3,e1)}, \texttt{ghb(e4,e2)}$ 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*.

4 Benchmarks and Conclusion

We have evaluated Cubicle-$\mathcal{W}$ on some classical parameterized concurrent algorithms (available on the tool's webpage [1]). Most of these algorithms are abstraction 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-$\mathcal{W}$'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 [7], Trencher [10,11], 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 $\mathcal{W}$	Memorax SB	Memorax PB	Trencher	CBMC Unwind 2	CBMC Unwind 3	Dual TSO
naive mutex	US	0.04s [N]	–	–	–	–	–	NT [N]
			TO [6]	TO [10]	TO [5]	23.6s [11]	5m37s [11]	TO [6]
			7m54s [5]	12m02s [9]	10.1s [4]	14.7s [10]	3m39s [10]	1m12s [5]
naive mutex	S	0.30s [N]	–	–	–	–	–	NT [N]
			TO [5]	TO [11]	TO [6]	TO [5]	TO [3]	TO [5]
			23.3s [4]	2m28s [10]	54.8s [5]	2m24s [4]	19.4s [2]	35.7s [4]
lamport	US	0.10s [N]	–	–	–	–	–	NT [N]
			TO [4]	TO [4]	KO [4]	7m42s [11]	TO [7]	TO [6]
			17.4s [3]	25.4s [3]	1.73s [3]	4m29s [10]	5m12s [6]	13m12s [5]
lamport	S	0.60s [N]	–	–	–	–	–	NT [N]
			TO [3]	TO [4]	KO [5]	TO [4]	TO [3]	TO [4]
			0.14s [2]	3m02s [3]	3.37s [4]	8m39s [3]	1m55s [2]	9.42s [3]
spinlock [22]	S	0.07s [N]	–	–	–	–	–	TO [N]
			TO [5]	TO [7]	TO [7]	TO [3]	TO [3]	TO [6]
			8m51s [4]	9m52s [6]	21.45s [6]	19.58s [2]	5m08s [2]	1m16s [5]
sense [19] reversing barrier	S	0.06s [N]	–	–	–	–	–	NT [N]
			TO [3]	TO [3]	TO [5]	TO [9]	TO [4]	TO [3]
			0.34s [2]	0.09s [2]	1m58s ☠ [4]	12m25s [8]	1m43s [3]	0.09s [2]
arbiter v1 [18]	S	0.18s [N]	–	–	–	–	–	NT [N]
			TO [1+2]	TO [1+2]	KO [1+5]	TO [1+6]	TO [1+3]	TO [1+6]
					4.57s [1+4]	12m02s [1+5]	44.3s [1+2]	2m45s ☠ [1+5]
arbiter v2 [18]	S	13.5s [N]	–	–	–	–	–	NT [N]
			TO [1+2]	TO [1+2]	KO [1+4]	TO [1+4]	TO [1+2]	TO [1+3]
					1.62s [1+3]	2m56s [1+3]		24.2s [1+2]
two phase commit	S	54.1s [N]	–	–	–	–	–	NT [N]
			TO [2]	TO [4]	TO [4]	TO [11]	TO [11]	TO [3]
				39.7s [3]	7.08s ☠ [3]	12m39s [10]	13m41s [10]	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 min.

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-$\mathcal{W}$ is thus a very promising approach to the verification of concurrent programs that are both parameterized and operating under weak memory. We have yet to tackle larger programs, which can be achieved by adapting Cubicle's invariant generation mechanism to our weak memory model.

References

1. Cubicle-$\mathcal{W}$. 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: CONCUR (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: TACAS (2013)
4. Abdulla, P.A., Delzanno, G., Henda, N.B., Rezine, A.: Regular model checking without transducers. In: TACAS (2007)
5. Abdulla, P.A., Delzanno, G., Rezine, A.: Parameterized verification of infinite-state processes with global conditions. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 145–157. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_17
6. Alglave, J., Fox, A., Ishtiaq, S., Myreen, M.O., Sarkar, S., Sewell, P., Nardelli, F.Z.: The semantics of power and arm multiprocessor machine code. In: DAMP (2008)
7. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software Verification for weak memory via program transformation. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 512–532. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_28
8. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: modelling, simulation, testing, and data mining for weak memory. In: ACM TPLS (2014)
9. Apt, K.R., Kozen, D.C.: Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett. **22**(6), 307–309 (1986)
10. Bouajjani, A., Calin, G., Derevenetc, E., Meyer, R.: Lazy tso reachability. In: FASE (2015)
11. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 533–553. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_29
12. Clarke, E.M., Grumberg, O., Browne, M.C.: Reasoning about networks with many identical finite-state processes. In: PODC (1986)
13. Conchon, S., Goel, A., Krstić, S., Mebsout, A., Zaïdi, F.: Cubicle: a parallel SMT-based model checker for parameterized systems. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 718–724. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_55
14. Conchon, S., Goel, A., Krstic, S., Mebsout, A., Zaidi, F.: Invariants for finite instances and beyond. In: FMCAD (2013)
15. Conchon, S., Mebsout, A., Zaïdi, F.: Certificates for parameterized model checking. In: Bjørner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 126–142. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19249-9_9
16. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM **39**(3), 675–735 (1992)
17. Ghilardi, S., Ranise, S.: MCMT: a model checker modulo theories. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 22–29. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_3
18. Goeman, H.J.M.: The arbiter: an active system component for implementing synchronizing primitives. Fundam. Inform. **4**(3), 517–530 (1981)
19. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming (2008)
20. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **9**, 690–691 (1979)

21. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 391–407. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_27
22. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. In: CACM (2010)