A Backward Reachability Algorithm for Parameterized Systems on Weak Memory

Sylvain Conchon, David Declerck and Fatiha Zaïdi

Université Paris Sud — Comprendre le monde, construire l'avenir

université PARIS-SACLAY

Tool Download

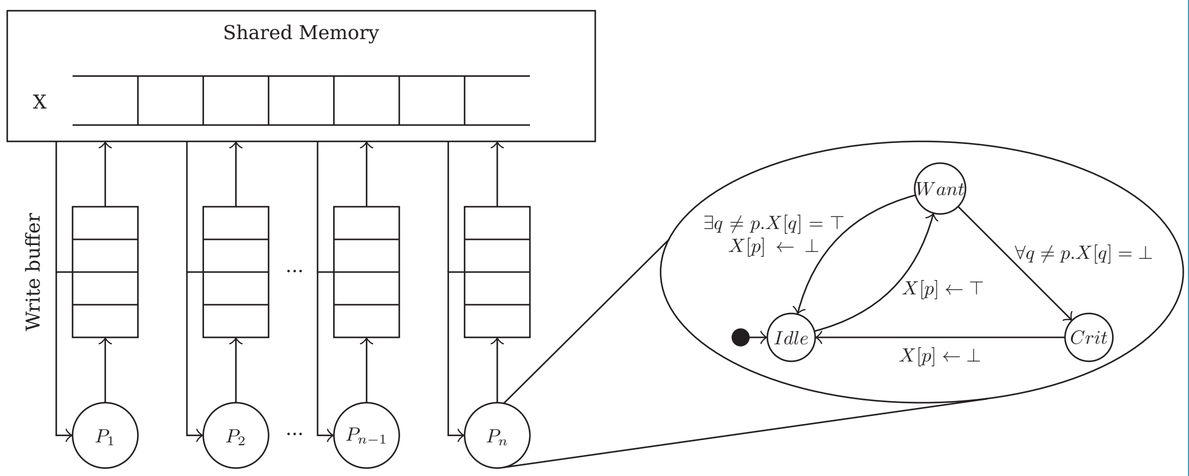
The algorithm described in this poster has been successfully implemented in a new version of the Cubicle model checker called Cubicle-$\mathcal{W}$. This tool, as well as various examples, can be downloaded here :

`https://www.lri.fr/~declerck/cubiclew/`

Weak Memory Parameterized Systems

★ Weak memory :

- order of memory accesses ≠ interleaving of memory instructions : algorithms are harder to check
- different "flavors" of reorderings : TSO, PSO, ARM...
- we adopt a TSO-like model, as shown on the left
- reorgerings can be prevented using *fences*



★ Parameterized systems :

- concurrent systems
- unbounded number of processes
- process-indexed arrays

Our example : a (naive and inefficient) mutual exclusion algorithm using a process-indexed array of booleans X, and each process p executes the automaton on the right.

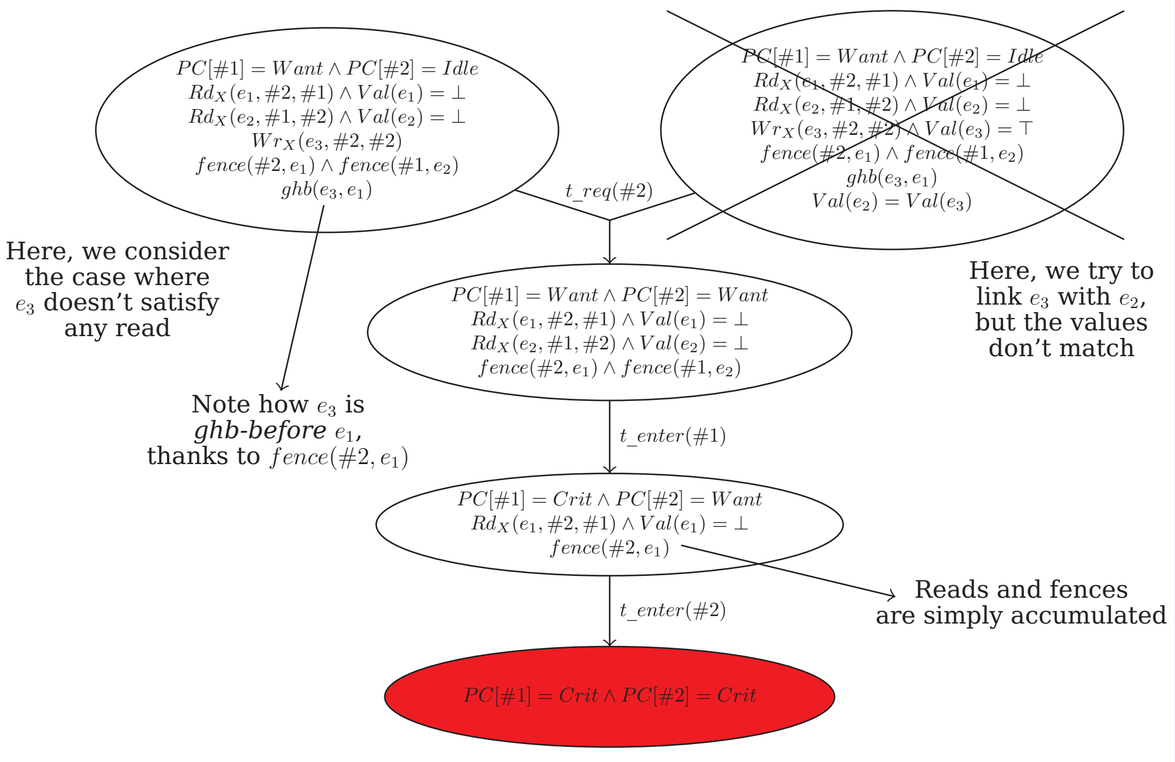
Our approach

★ The base framework :

- Model Checking Modulo Theories (MCMT)
- checks safety properties of parameterized systems
- assumes a sequentially consistent (SC) memory
- uses a backward reachability algorithm

★ Our extension :

- adds weak memory reasoning using an axiomatic model
- maps memory instructions to read/write events
- builds a *global-happens-before* relation over events



Cubicle-$\mathcal{W}$ code

The automaton on the left can be expressed as a parameterized transition system in the language of Cubicle-$\mathcal{W}$.

```
type loc = Idle | Want | Crit

array PC[proc] :  loc

weak array X[proc] :  bool

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

unsafe (p1 p2) { PC[p1] = Crit && PC[p2] = Crit }

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

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

transition t_cancel ([p] q)
requires { PC[p] = Want && fence(p) &&
           X[p] = True }
{ PC[p] := Idle; X[p] := False }

(* Critical section *)

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

Note the use of the *fence* predicate, that allows a transition to be taken only when the process' buffer is empty.

Benchmarks

The implementation was tested on some typical concurrent algorithms. Some algorithms are incorrect due to the effects of weak memory. In this case, we created fixed versions of these algorithms by adding *fences*. The test machine features an Intel Core i7 CPU @ 2.9 Ghz and 8GB of RAM.

Benchmark Name	Correct ?	Analysis Time
naive_mutex_N	No	0,05s
naive_mutex_f_N	Yes	0,06s
lamport_N	No	0,42s
lamport_f_N	Yes	0,62s
spinlock_N	Yes	0,07s
sense_rev_N	Yes	0,15s
msi_N	Yes	0,09s
moesi_N	Yes	0,21s

Acknowledgements

This work is supported by the French ANR project PARDI (DS0703)