Parameterized Model Checking with Partial Order Reduction Technique for the TSO Weak Memory Model

Sylvain Conchon[1], David Declerck[2], Fatiha Zaïdi[1]

[1]Université Paris-Saclay, [2]OCamlPro SAS

Cambium Seminar
January 27th, 2020

Context - part 1 : the x86-TSO Memory Model

Initial state : x = 0, y = 0	
Thread 1	Thread 2
mov [x], 1	mov [y], 1
mov eax, [y]	mov ebx, [x]

Context - part 1 : the x86-TSO Memory Model

Initial state : x = 0, y = 0	
Thread 1	Thread 2
mov [x], 1 mov eax, [y]	mov [y], 1 mov ebx, [x]

Possible outcomes in (t1:eax, t2:ebx) :

- Obviously: (0, 1), (1, 0), (1, 1)

Context - part 1 : the x86-TSO Memory Model

Initial state : x = 0, y = 0

Thread 1	Thread 2
mov [x], 1	mov [y], 1
mov eax, [y]	mov ebx, [x]

Possible outcomes in (t1:eax, t2:ebx) :

- Obviously: (0, 1), (1, 0), (1, 1)
- Surprisingly: (0, 0)

Context - part 1 : the x86-TSO Memory Model

Initial state : x = 0, y = 0

Thread 1	Thread 2
mov [x], 1	mov [y], 1
mov eax, [y]	mov ebx, [x]

Possible outcomes in (t1:eax, t2:ebx) :

- Obviously: (0, 1), (1, 0), (1, 1)
- Surprisingly: (0, 0)

TSO is a weak memory model :
orders of memory accesses $\neq$ interleaving of instructions

Context - part 1 : the x86-TSO memory model

Eliminating TSO behaviors

New behaviors are not necessarily incorrect

Memory fences may be used to prevent some of these behaviors

Initial state : x = 0, y = 0

Thread 1	Thread 2
mov [x], 1	mov [y], 1
mfence	**mfence**
mov eax, [y]	mov ebx, [x]

Context - part 2 : Parameterized Systems

Parameterized systems:

- concurrent systems with an arbitrary number of processes
- expressed as transition systems manipulating arrays indexed by process identifiers

Example :

- mutual exclusion algorithms
- synchronization barriers
- cache coherence protocols
- ...

Context - part 3 : the Cubicle Model Checker

`http://cubicle.lri.fr`

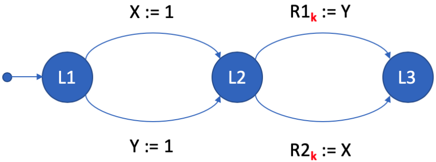
Université Paris-Sud
Intel Strategic Cad Lab

Cubicle is an open source SMT based model checker, written in OCaml and its implementation relies on a lightweight and enhanced version of the SMT solver Alt-Ergo

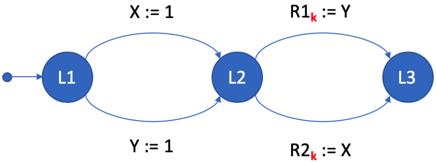
Cubicle implements the Model Checking Modulo Theories framework of S. Ghilardi and S. Ranise

MCMT = SMT + Backward Reachability Algorithm

Context - part 3 : Running Example

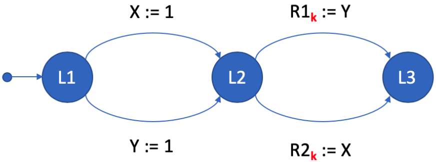


Context - part 3 : Running Example



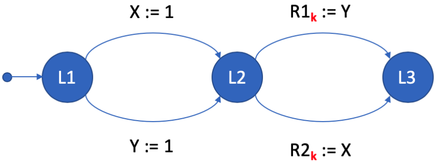

```
type state = L1 | L2 | L3
array PC[proc] :  state
var X : int
var Y : int
array R1[proc] :  int
array R2[proc] :  int
```

Context - part 3 : Running Example



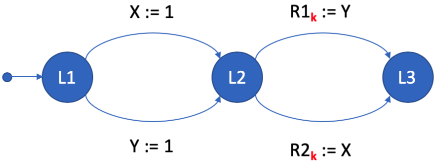

```
type state = L1 | L2 | L3
array PC[proc] :  state
var X : int
var Y : int
array R1[proc] :  int
array R2[proc] :  int

init (k) { PC[k] = L1
    && X = 0 && Y = 0
    && R1[k]<>0 && R2[k]<>0 }
```

Context - part 3 : Running Example




```
type state = L1 | L2 | L3
array PC[proc] :  state
var X : int
var Y : int
array R1[proc] :  int
array R2[proc] :  int

init (k) { PC[k] = L1
   && X = 0 && Y = 0
   && R1[k]<>0 && R2[k]<>0 }

unsafe (i j) {
PC[i] = L3 && PC[j] = L3 &&
R1[i] = 0 && R2[j] = 0 }
```

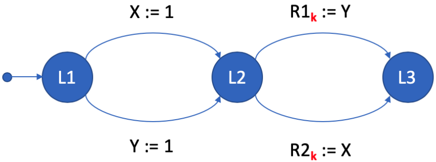
Context - part 3 : Running Example




```
type state = L1 | L2 | L3
array PC[proc] :  state
var X : int
var Y : int
array R1[proc] :  int
array R2[proc] :  int

init (k) { PC[k] = L1
    && X = 0 && Y = 0
    && R1[k]<>0 && R2[k]<>0 }

unsafe (i j) {
PC[i] = L3 && PC[j] = L3 &&
R1[i] = 0 && R2[j] = 0 }
```

```
transition t1_1 (k)
requires { PC[k] = L1 }
{ PC[k] := L2; X := 1 }
```

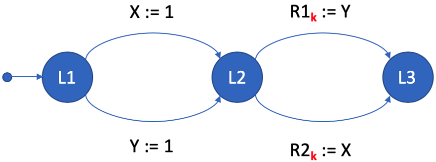
Context - part 3 : Running Example




```
type state = L1 | L2 | L3
array PC[proc] :  state
var X : int
var Y : int
array R1[proc] :  int
array R2[proc] :  int

init (k) { PC[k] = L1
    && X = 0 && Y = 0
    && R1[k]<>0 && R2[k]<>0 }

unsafe (i j) {
PC[i] = L3 && PC[j] = L3 &&
R1[i] = 0 && R2[j] = 0 }
```

```
transition t1_1 (k)
requires { PC[k] = L1 }
{ PC[k] := L2; X := 1 }

transition t1_2 (k)
requires { PC[k] = L1 }
{ PC[k] := L2; Y := 1 }
```

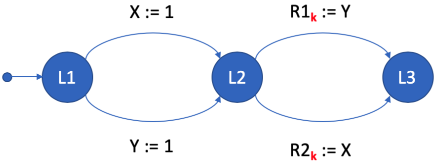
Context - part 3 : Running Example




```
type state = L1 | L2 | L3
array PC[proc] :  state
var X : int
var Y : int
array R1[proc] :  int
array R2[proc] :  int

init (k) { PC[k] = L1
    && X = 0 && Y = 0
    && R1[k]<>0 && R2[k]<>0 }

unsafe (i j) {
PC[i] = L3 && PC[j] = L3 &&
R1[i] = 0 && R2[j] = 0 }
```

```
transition t1_1 (k)
requires { PC[k] = L1 }
{ PC[k] := L2; X := 1 }

transition t1_2 (k)
requires { PC[k] = L1 }
{ PC[k] := L2; Y := 1 }

transition t2_1 (k)
requires { PC[k] = L2 }
{ PC[k] := L3; R1[k] := Y }
```

Context - part 3 : Running Example




```
type state = L1 | L2 | L3
array PC[proc] :  state
var X : int
var Y : int
array R1[proc] :  int
array R2[proc] :  int

init (k) { PC[k] = L1
    && X = 0 && Y = 0
    && R1[k]<>0 && R2[k]<>0 }

unsafe (i j) {
PC[i] = L3 && PC[j] = L3 &&
R1[i] = 0 && R2[j] = 0 }
```

```
transition t1_1 (k)
requires { PC[k] = L1 }
{ PC[k] := L2; X := 1 }

transition t1_2 (k)
requires { PC[k] = L1 }
{ PC[k] := L2; Y := 1 }

transition t2_1 (k)
requires { PC[k] = L2 }
{ PC[k] := L3; R1[k] := Y }

transition t2_2 (k)
requires { PC[k] = L2 }
{ PC[k] := L3; R2[k] := X }
```

Context - part 3 : Running Example

If X and Y are weak memories, this algorithm should be considered as unsafe

$$\texttt{init(\#1,\#2)} \xrightarrow{t1\_1(\#1)} \texttt{X = 1} \xrightarrow{t2\_1(\#1)} \texttt{R1[\#1] = 0} \xrightarrow{t1\_2(\#2)}$$

$$\texttt{Y = 1} \xrightarrow{t2\_2(\#2)} \texttt{R2[\#2] = 0}$$

Context - part 3 : Running Example

If X and Y are weak memories, this algorithm should be considered as unsafe

$$\texttt{init(\#1,\#2)} \xrightarrow{t1\_1(\#1)} \texttt{X = 1} \xrightarrow{t2\_1(\#1)} \texttt{R1[\#1] = 0} \xrightarrow{t1\_2(\#2)}$$

$$\texttt{Y = 1} \xrightarrow{t2\_2(\#2)} \texttt{R2[\#2] = 0}$$

Unfortunately, the memory model underlying Cubicle is sequential consistency (SC) : all reads and writes are in order

Context - part 3 : Running Example

If X and Y are weak memories, this algorithm should be considered as unsafe

$$\texttt{init(\#1,\#2)} \xrightarrow{t1\_1(\#1)} \texttt{X = 1} \xrightarrow{t2\_1(\#1)} \texttt{R1[\#1] = 0} \xrightarrow{t1\_2(\#2)}$$

$$\texttt{Y = 1} \xrightarrow{t2\_2(\#2)} \texttt{R2[\#2] = 0}$$


Unfortunately, the memory model underlying Cubicle is sequential consistency (SC) : all reads and writes are in order

Demo

Our goal in this work

Implementing a new version of Cubicle, called Cubicle-$\mathcal{W}$, for the TSO memory model

Input Language of Cubicle-$\mathcal{W}$

```
type state = L1 | L2 | L3
array PC[proc] :  state
weak var X : int
weak var Y : int
array R1[proc] :  int
array R2[proc] :  int

init (k) { PC[k] = L1
    && X = 0 && Y = 0
    && R1[k]<>0 && R2[k]<>0 }

unsafe (i j) {
PC[i] = L3 && PC[j] = L3 &&
R1[i] = 0 && R2[j] = 0 }
```

```
transition t1_1 ([k])
requires { PC[k] = L1 }
{ PC[k] := L2; X := 1 }

transition t1_2 ([k])
requires { PC[k] = L1 }
{ PC[k] := L2; Y := 1 }

transition t2_1 ([k])
requires { PC[k] = L2 }
{ PC[k] := L3; R1[k] := Y }

transition t2_2 ([k])
requires { PC[k] = L2 }
{ PC[k] := L3; R2[k] := X }
```

In the rest of the talk

Axiomatic weak memory models

Model checking modulo theories for weak memory models

A TSO-specific partial order reduction technique

Experimental evaluation with Cubicle-$\mathcal{W}$

Conclusion

Axiomatic Weak Memory Models

An axiomatic description of the TSO memory model

TSO reasoning is done through an axiomatic model that:

- maps memory instructions to read and write events
- builds various relations over these events, according to their dependencies
    - ***po*** : program order
    - *ppo* : preserved program order
    - ***rf*** : read-from
    - ***co*** : coherence
    - *fr* : from-read
    - ***fence*** : memory fence
- builds a *global happens-before* (*ghb*) relation out of the different relations

An axiomatic description of the TSO memory model

TSO reasoning is done through an axiomatic model that:

- maps memory instructions to read and write events
- builds various relations over these events, according to their dependencies
  - ***po*** : program order
  - *ppo* : preserved program order
  - ***rf*** : read-from
  - ***co*** : coherence
  - *fr* : from-read
  - ***fence*** : memory fence
- builds a *global happens-before* (*ghb*) relation out of the different relations

An execution is feasible if we can build an acyclic *ghb* relation

An axiomatic description of the TSO memory model

TSO reasoning is done through an axiomatic model that:

- maps memory instructions to read and write events
- builds various relations over these events, according to their dependencies
  - ***po*** : program order
  - *ppo* : preserved program order
  - ***rf*** : read-from
  - ***co*** : coherence
  - *fr* : from-read
  - ***fence*** : memory fence
- builds a *global happens-before* (*ghb*) relation out of the different relations

An execution is feasible if we can build an acyclic *ghb* relation

We use the formalism of J. Alglave and L. Maranget

Axiomatic TSO model

Events

Memory operations generate events

Axiomatic TSO model

Events

Memory operations generate events

Initial state : x = 0, y = 0

Thread 1	Thread 2
mov [x], 1	mov [y], 1
mfence	mfence
mov eax, [y]	mov ebx, [x]

Axiomatic TSO model

Events

Memory operations generate events

Initial state : x = 0, y = 0	
Thread 1	Thread 2
mov [x], 1	mov [y], 1
mfence	mfence
mov eax, [y]	mov ebx, [x]

e_1:Wx=0 e_2:Wy=0

e_3:Wx=1 e_5:Wy=1

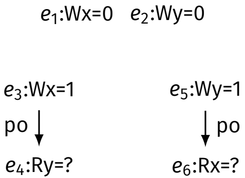
e_4:Ry=? e_6:Rx=?

Axiomatic TSO model

Program Order (po)

Events from the same process are in Program Order

Initial state : x = 0, y = 0	
Thread 1	Thread 2
mov [x], 1	mov [y], 1
mfence	mfence
mov eax, [y]	mov ebx, [x]



Axiomatic TSO model

Preserved Program Order (ppo)

Under TSO, WR pairs are not preserved in Program Order

Initial state : x = 0, y = 0	
Thread 1	Thread 2
mov [x], 1	mov [y], 1
mfence	mfence
mov eax, [y]	mov ebx, [x]

e_1:Wx=0 e_2:Wy=0

e_3:Wx=1 e_5:Wy=1

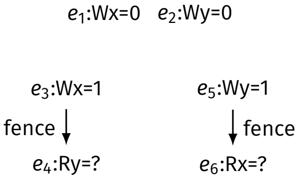
e_4:Ry=? e_6:Rx=?

Axiomatic TSO model

Fence

All WR pairs separated by a fence are in a Fence relation

Initial state : x = 0, y = 0	
Thread 1	Thread 2
mov [x], 1	mov [y], 1
mfence	mfence
mov eax, [y]	mov ebx, [x]

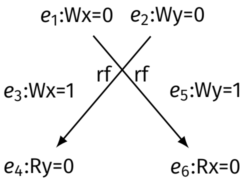


Axiomatic TSO model

Read-From (rf)

Each read takes its value from a single write

Initial state : x = 0, y = 0	
Thread 1	Thread 2
mov [x], 1	mov [y], 1
mfence	mfence
mov eax, [y]	mov ebx, [x]



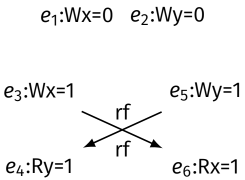
Axiomatic TSO model

Read-From (rf)

Each read takes its value from a single write

Initial state : x = 0, y = 0

Thread 1	Thread 2
mov [x], 1	mov [y], 1
mfence	mfence
mov eax, [y]	mov ebx, [x]

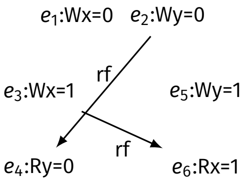


Axiomatic TSO model

Read-From (rf)

Each read takes its value from a single write

Initial state : x = 0, y = 0	
Thread 1	Thread 2
mov [x], 1	mov [y], 1
mfence	mfence
mov eax, [y]	mov ebx, [x]

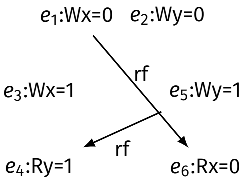


Axiomatic TSO model

Read-From (rf)

Each read takes its value from a single write

Initial state : x = 0, y = 0	
Thread 1	Thread 2
mov [x], 1	mov [y], 1
mfence	mfence
mov eax, [y]	mov ebx, [x]



Axiomatic TSO model

Read-From (rf)

Each read takes its value from a single write

Initial state : x = 0, y = 0	
Thread 1	Thread 2
mov [x], 1	mov [y], 1
mfence	mfence
mov eax, [y]	mov ebx, [x]

rf is split in two sub-relations, ***rfi*** (internal) and ***rfe*** (external), depending on whether it relates to events issued by the same process or events issued by distinct processes

Axiomatic TSO model

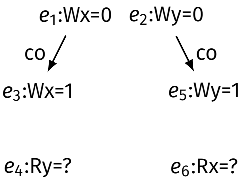
Coherence (co)

There is a total order on all writes to the same variable

Initial state : x = 0, y = 0

Thread 1	Thread 2
mov [x], 1	mov [y], 1
mfence	mfence
mov eax, [y]	mov ebx, [x]

→



Axiomatic TSO model

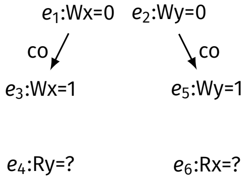
Coherence (co)

There is a total order on all writes to the same variable

Initial state : x = 0, y = 0

Thread 1	Thread 2
mov [x], 1	mov [y], 1
mfence	mfence
mov eax, [y]	mov ebx, [x]

$\rightarrow$



From *rf* and *co*, we derive a new relation *fr* :

$$\forall e_1, e_2, e_3 . rf(e_1, e_2) \wedge co(e_1, e_3) \rightarrow fr(e_2, e_3)$$

Axiomatic TSO model

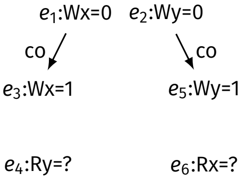
Coherence (co)

There is a total order on all writes to the same variable

Initial state : x = 0, y = 0

Thread 1	Thread 2
mov [x], 1	mov [y], 1
mfence	mfence
mov eax, [y]	mov ebx, [x]

→



From *rf* and *co*, we derive a new relation *fr* :

$$\forall e_1, e_2, e_3 . rf(e_1, e_2) \land co(e_1, e_3) \rightarrow fr(e_2, e_3)$$

" When a read takes its value from a write, then a write that is after this specific write in the co relation also has to occur after the read"

Axiomatic TSO model: Global Happens-Before

ghb is the smallest partial order relation such that:

$$\forall e_1, e_2 \cdot ppo(e_1, e_2) \rightarrow ghb(e_1, e_2) \quad \text{GHB-PPO}$$

$$\forall e_1, e_2 \cdot fence(e_1, e_2) \rightarrow ghb(e_1, e_2) \quad \text{GHB-FENCE}$$

$$\forall e_1, e_2 \cdot rfe(e_1, e_2) \rightarrow ghb(e_1, e_2) \quad \text{GHB-RFE}$$

$$\forall e_1, e_2 \cdot co(e_1, e_2) \rightarrow ghb(e_1, e_2) \quad \text{GHB-CO}$$

$$\forall e_1, e_2 \cdot fr(e_1, e_2) \rightarrow ghb(e_1, e_2) \quad \text{GHB-FR}$$

Axiomatic TSO model: Global Happens-Before

ghb is the smallest partial order relation such that:

$$\forall e_1, e_2 \cdot ppo(e_1, e_2) \rightarrow ghb(e_1, e_2) \quad \text{GHB-PPO}$$

$$\forall e_1, e_2 \cdot fence(e_1, e_2) \rightarrow ghb(e_1, e_2) \quad \text{GHB-FENCE}$$

$$\forall e_1, e_2 \cdot rfe(e_1, e_2) \rightarrow ghb(e_1, e_2) \quad \text{GHB-RFE}$$

$$\forall e_1, e_2 \cdot co(e_1, e_2) \rightarrow ghb(e_1, e_2) \quad \text{GHB-CO}$$

$$\forall e_1, e_2 \cdot fr(e_1, e_2) \rightarrow ghb(e_1, e_2) \quad \text{GHB-FR}$$

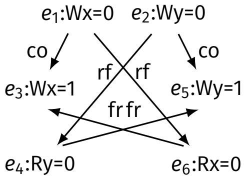
An execution defined by $(po, rf, co, fence)$ is feasible if the *ghb* relation that it generates is acyclic

Axiomatic TSO Model : valid execution

Without fences, an execution that ends with (eax=0,ebx=0) is feasible

Initial state : x = 0, y = 0	
Thread 1	Thread 2
mov [x], 1 mov eax, [y]	mov [y], 1 mov ebx, [x]

$\longrightarrow$

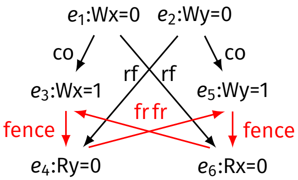


Axiomatic TSO Model : invalid execution

Using memory barriers, such execution is not feasible

Etat initial : x = 0, y = 0	
Thread 1	Thread 2
mov [x], 1	mov [y], 1
mfence	mfence
mov eax, [y]	mov ebx, [x]

$\rightarrow$



MCMT for Weak Memory Models

MCMT [Ghilardi, Ranise]

System states and transitions are first-order formulas

MCMT [Ghilardi, Ranise]

System states and transitions are first-order formulas

Initial states are defined by a universally quantified formula:

```
init(i) { A[i] = True && PC = L1 }
```

$\forall i : \texttt{proc}.A[i] \wedge PC = \texttt{L1}$

Bad states are defined by special existentially quantified formulas, called cubes:

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

$\exists i, j : \texttt{proc}.\ i \neq j \wedge \texttt{S[i]} = \texttt{Crit} \wedge \texttt{S[j]} = \texttt{Crit}$

Transitions correspond to existentially quantified formulas:

```
transition t(i)
  requires { S[i] = A && PC = L1 }
  { S[i] = B; X = X+1 }
```

$\exists i : \texttt{proc}.\ \texttt{S[i]} = \texttt{A} \wedge \texttt{PC} = \texttt{L1} \wedge \texttt{S' = S[i <- B]} \wedge \texttt{X'} = \texttt{X} + 1$

Inductive invariants

We are looking for a predicate Reach such that :

Reach is an inductive invariant :

$$\forall \vec{x}.\mathrm{Init}(\vec{x}) \Rightarrow \mathrm{Reach}(\vec{x})$$

$$\forall \vec{x}, \vec{x}'.\mathrm{Reach}(\vec{x}) \wedge \tau(\vec{x}, \vec{x}') \Rightarrow \mathrm{Reach}(\vec{x}')$$

The system is safe if there exists an interpretation of Reach such that :

$$\forall \vec{x}.\mathrm{Reach}(\vec{x}) \models \neg \mathrm{unsafe}(\vec{x})$$

Inductive invariants

We are looking for a predicate Reach such that :

Reach is an inductive invariant :

$$\forall \vec{x}. \text{Init}(\vec{x}) \Rightarrow \text{Reach}(\vec{x})$$

$$\forall \vec{x}, \vec{x}'. \text{Reach}(\vec{x}) \wedge \tau(\vec{x}, \vec{x}') \Rightarrow \text{Reach}(\vec{x}')$$

The system is safe if there exists an interpretation of Reach such that :

$$\forall \vec{x}. \text{Reach}(\vec{x}) \models \neg \text{unsafe}(\vec{x})$$

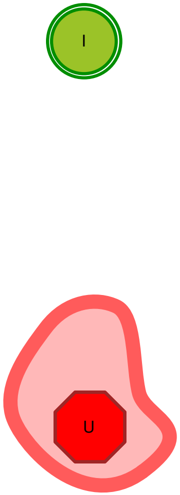
Cubicle computes Reach by backward reachability

Backward Reachability

```
BR (τ,I,U):
    V := ∅
    push(Q, U)
    while not_empty(Q) do
        φ := pop(Q)
        if  φ ∧ I sat  then
            return unsafe
        if  ¬(φ ⊨ ⋁_{ψ∈V} ψ)  then
            V := V ∪ { φ }
            push(Q,  pre_τ(φ))
    return safe
```

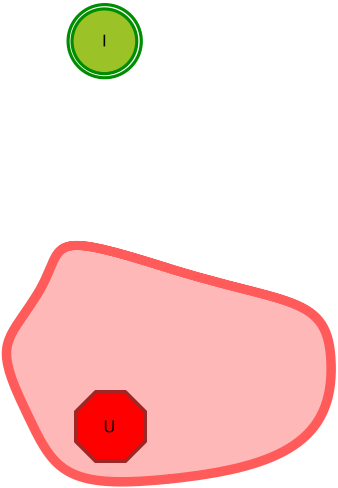
Backward Reachability

BR (τ,I,U):
    V := $\emptyset$
    push(Q, U)
    while not_empty(Q) **do**
        φ := pop(Q)
        if $\varphi \wedge$ I sat **then**
            return unsafe
        if $\neg(\varphi \models \bigvee_{\psi \in V} \psi)$ **then**
            V := V $\cup$ { φ }
            push(Q, $pre_\tau(\varphi)$)
    return safe



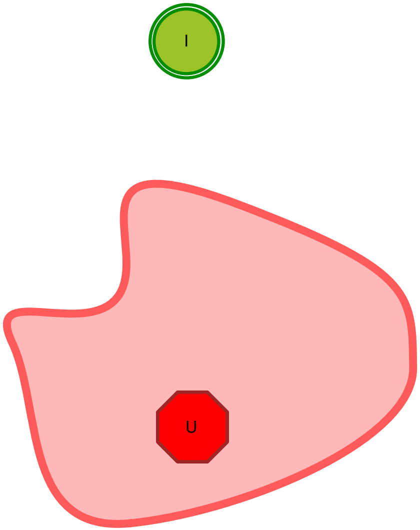
Backward Reachability

```
BR (τ,I,U):
  V := ∅
  push(Q, U)
  while not_empty(Q) do
    φ := pop(Q)
    if φ ∧ I sat then
      return unsafe
    if ¬(φ ⊨ ⋁_{ψ∈V} ψ) then
      V := V ∪ { φ }
      push(Q, pre_τ(φ))
  return safe
```



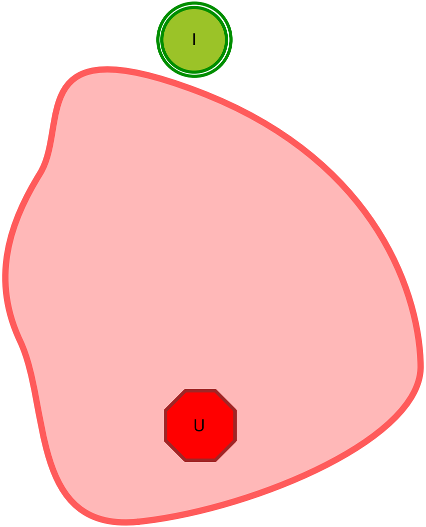
Backward Reachability

BR (τ,I,U):
    V := $\emptyset$
    push(Q, U)
    while not_empty(Q) **do**
        φ := pop(Q)
        if $\varphi \wedge$ I sat **then**
            return unsafe
        if $\neg(\varphi \models \bigvee_{\psi \in V} \psi)$ **then**
            V := V $\cup$ { φ }
            push(Q, $pre_\tau(\varphi)$)
    return safe



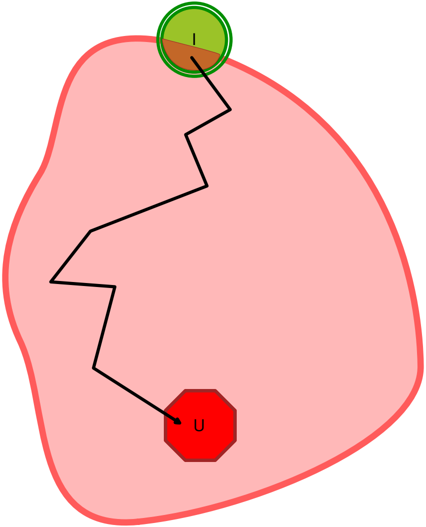
Backward Reachability

BR (τ,I,U):
    V := $\emptyset$
    push(Q, U)
    while not_empty(Q) **do**
        φ := pop(Q)
        if $\varphi \wedge$ I sat **then**
            return unsafe
        if $\neg(\varphi \models \bigvee_{\psi \in V} \psi)$ **then**
            V := V $\cup$ { φ }
            push(Q, $pre_\tau(\varphi)$)
    return safe



Backward Reachability

BR (τ,I,U):
    V := $\emptyset$
    push(Q, U)
    while not_empty(Q) **do**
        φ := pop(Q)
        if $\varphi \wedge$ I sat **then**
            return unsafe
        if $\neg(\varphi \models \bigvee_{\psi \in V} \psi)$ **then**
            V := V $\cup$ { φ }
            push(Q, $pre_\tau(\varphi)$)
    return safe



Backward Reachability

BR (τ,I,U):

V := $\emptyset$

push(Q, U)

while not_empty(Q) **do**

φ := pop(Q)

if $\varphi \wedge$ I sat **then**

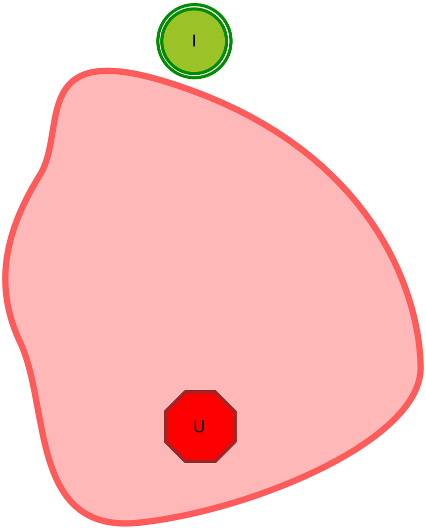
return unsafe

if $\neg(\varphi \models \bigvee_{\psi \in V} \psi)$ **then**

V := V $\cup$ { φ }

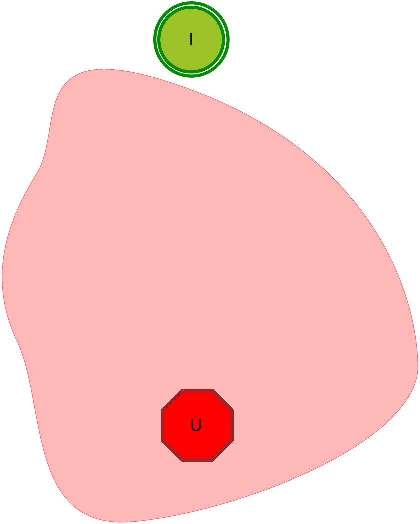
push(Q, $\text{pre}_\tau(\varphi)$)

return safe



Backward Reachability

BR (τ,I,U):
  V := $\emptyset$
  push(Q, U)
  while not_empty(Q) **do**
    φ := pop(Q)
    if $\varphi \wedge$ I sat **then**
      return unsafe
    if $\neg(\varphi \models \bigvee_{\psi \in V} \psi)$ **then**
      V := V $\cup$ { φ }
      push(Q, $pre_\tau(\varphi)$)
  return safe



Backward Reachability Modulo TSO

Cubicle-$\mathcal{W}$ implements an extended version of the backward reachability algorithms of Cubicle

For reasoning about TSO, one needs to :

- associate events to read and write operations in logical formulas
- build a global happens-before (ghb) relation during the backward analysis according to the dependencies between those events
- add an axiomatic model of TSO inside the SMT solver to check satisfiability of TSO fomulas

Backward Reachability Modulo TSO

BR (τ, I, U):

```
V := ∅
push(Q, U)
while not_empty(Q) do
    φ := pop(Q)
    if φ ∧ I sat then
        return unsafe
    if ¬(φ ⊨ ⋁_{ψ∈V} ψ) then
        V := V ∪ { φ }
        push(Q, pre_τ(φ))
return safe
```

Backward Reachability Modulo TSO

BR (τ, I, U):

  V := $\emptyset$

  push(Q, U)

  while not_empty(Q) **do**

    φ := pop(Q)

    if $\varphi \wedge$ I sat **then**

      return unsafe

    if $\neg(\varphi \models \bigvee_{\psi \in V} \psi)$ **then**

      V := V $\cup$ { φ }

      push(Q, $pre_\tau(\varphi)$)

  return safe

Safety test & Fixpoint check

- Performed by an SMT solver
- Logic and SMT extended to reason about events and TSO relations

Pre-image computation

- Instrumented to produce events and relations
- Decides which writes satisfy each read

Events

```
weak var X : int
weak array T[proc] : int
```

Events

```
weak var X : int
weak array T[proc] : int
```

When a variable is **read** :

```
transition t([i]) requires { X = 42 } { ... }
```

$$\exists e_1.\ \ e_1{:}R_X^i \land val(e_1) = 42 \land pending_X(e_1)$$

Events

```
weak var X : int
weak array T[proc] : int
```

When a variable is **read** :

```
transition t([i]) requires { X = 42 } { ... }
```

$$\exists e_1.\ \ e_1{:}R^i_X \wedge val(e_1) = 42 \wedge pending_X(e_1)$$

When a variable is **assigned** :

```
transition t([i]) requires { ... } { X := 42 }
```

$$\exists e_2 \cdot e_2{:}W^i_X \wedge val(e_2) = 42$$

Initial states

```
type state = L1 | L2 | L3
array PC[proc] :  state
weak var X : int
weak var Y : int
array R1[proc] :  int
array R2[proc] :  int

init (k) {PC[k] = L1 && X = 0 && Y = 0 && R1[k]<>0 && R2[k]<>0 }
```

Initial states

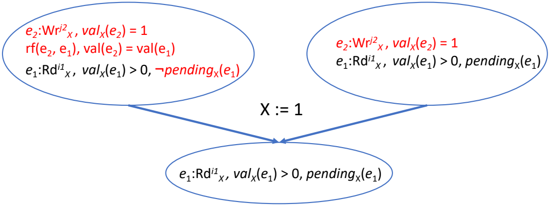
```
type state = L1 | L2 | L3
array PC[proc] :  state
weak var X : int
weak var Y : int
array R1[proc] :  int
array R2[proc] :  int

init (k) {PC[k] = L1 && X = 0 && Y = 0 && R1[k]<>0 && R2[k]<>0 }
```

$$\begin{aligned} \forall k.\forall e_1, e_2. \quad & PC[k] = L1 \wedge R_1[k] \neq 0 \wedge R_2[k] \neq 0 \wedge \\ & e_1 : Rd_X^k \wedge pending_X(e_1) \wedge val_X(e_1) = 0 \wedge \\ & e_2 : Rd_Y^k \wedge pending_Y(e_2) \wedge val_Y(e_2) = 0 \end{aligned}$$

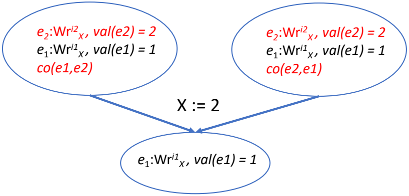
Pre-image Computation

Similare to Cubicle, but …

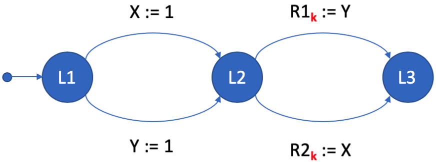
- connect reads and writes with the *rf* relation



- decides the memory coherence between writes with the *co* relation



Running Example



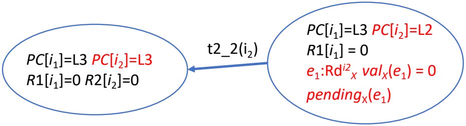

```
type state = L1 | L2 | L3
array PC[proc] :  state
weak var X : int
weak var Y : int
array R1[proc] :  int
array R2[proc] :  int

init (k) { PC[k] = L1
    && X = 0 && Y = 0
    && R1[k]<>0 && R2[k]<>0 }

unsafe (i j) {
PC[i] = L3 && PC[j] = L3 &&
R1[i] = 0 && R2[j] = 0 }
```

```
transition t1_1 ([k])
requires { PC[k] = L1 }
{ PC[k] := L2; X := 1 }

transition t1_2 ([k])
requires { PC[k] = L1 }
{ PC[k] := L2; Y := 1 }

transition t2_1 ([k])
requires { PC[k] = L2 }
{ PC[k] := L3; R1[k] := Y }

transition t2_2 ([k])
requires { PC[k] = L2 }
{ PC[k] := L3; R2[k] := X }
```

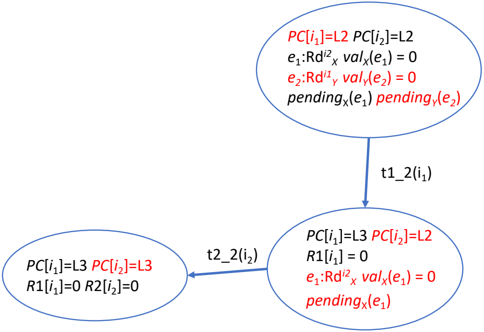
Backward Reachability Modulo TSO : Example

$PC[i_1]$=L3 $PC[i_2]$=L3
$R1[i_1]$=0 $R2[i_2]$=0

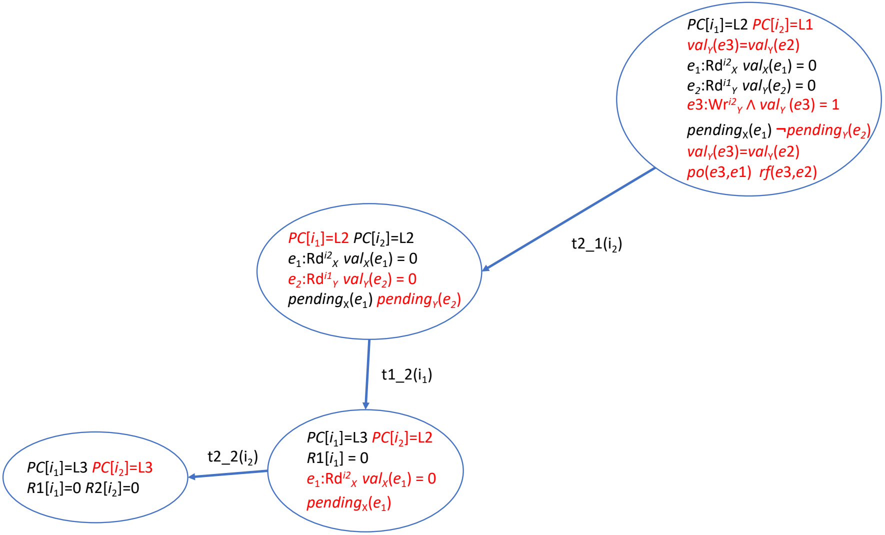
Backward Reachability Modulo TSO : Example



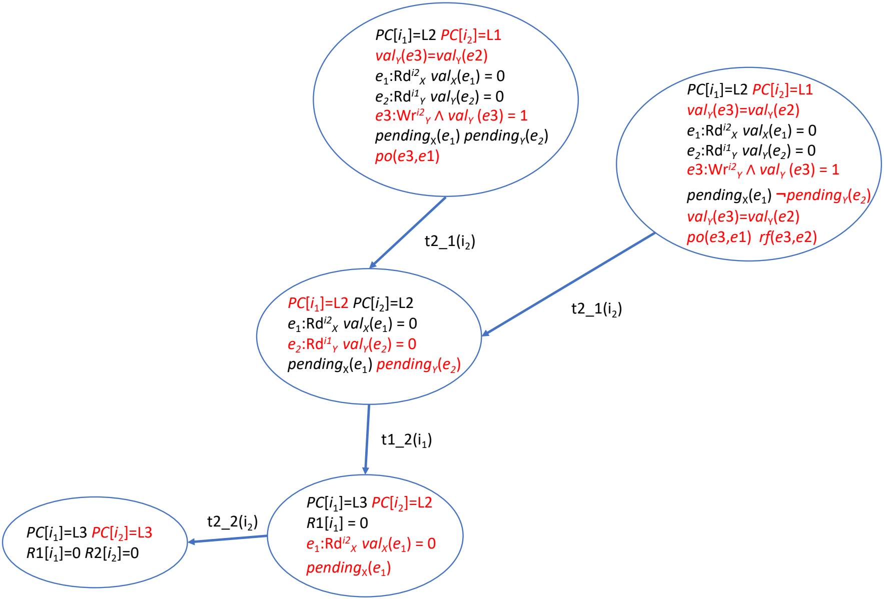
Backward Reachability Modulo TSO : Example



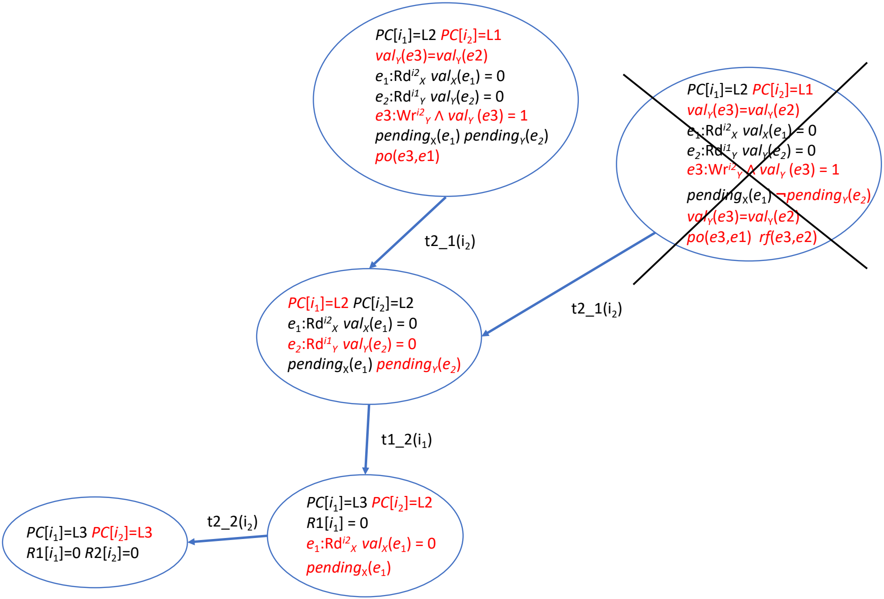
Backward Reachability Modulo TSO : Example



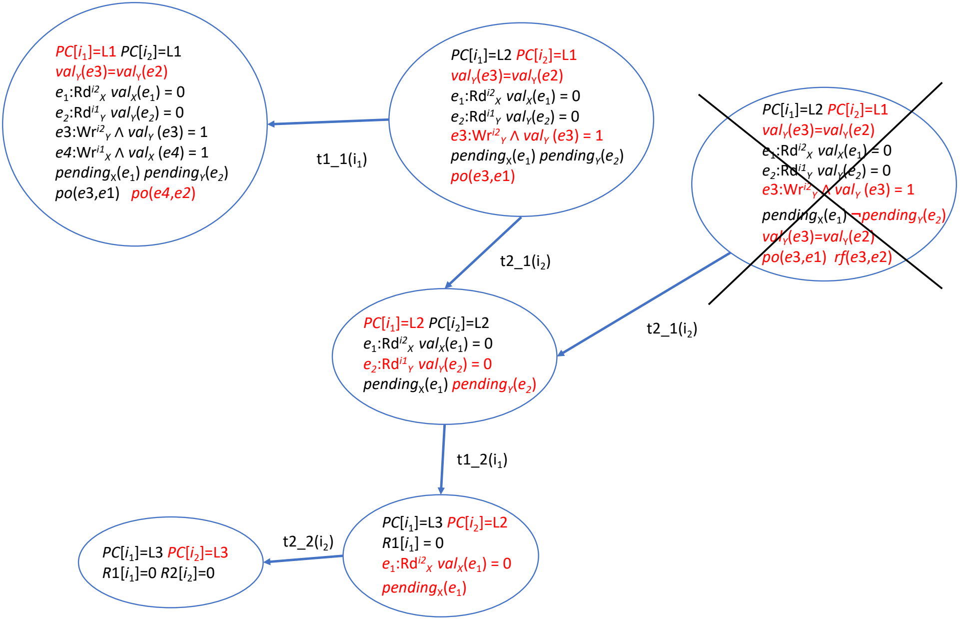
Backward Reachability Modulo TSO : Example



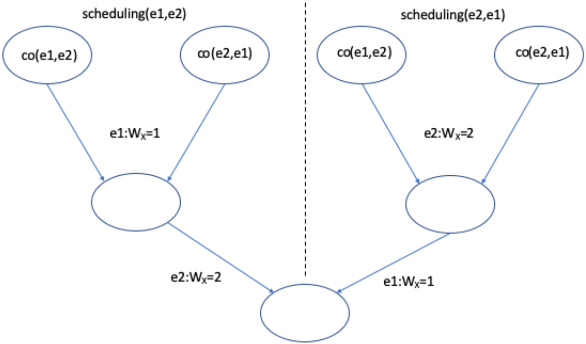
Backward Reachability Modulo TSO : Example



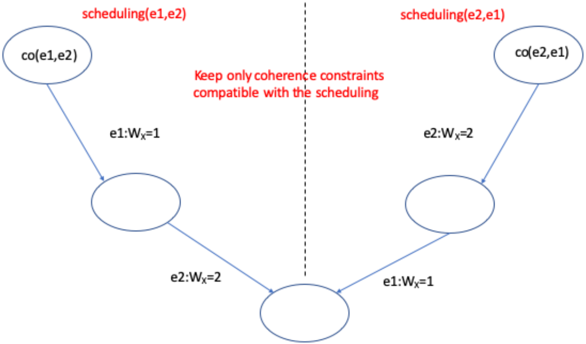
Backward Reachability Modulo TSO : Example



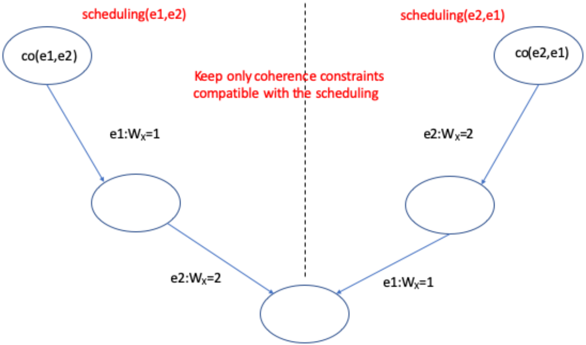
A TSO-Specific Partial Order Reduction Technique



A TSO-Specific Partial Order Reduction Technique



A TSO-Specific Partial Order Reduction Technique



Theorem. For every execution defined by (*po*,*rf*,*co*) and every scheduling *S* of a program, there exists a scheduling *S'* such that *co* is compatible with *S'*.

Efficient Backward Reachability Modulo TSO

We exploit the partial order reduction property to simplify backward search of TSO by directly building the *ghb* relation, on the fly..

- a new read will be before any old event event from the same process in ghb
- a new write will be before any old write from the same process in ghb
- a new write will be before any old read from the same process in ghb if they are separated by a fence
- a new write will be before any old write on the same variable in ghb (compatibility of co/sched)
- a new write will be before any old read from a different process that it satisfies in ghb
- a new write will be after any old read from a different process that it does not satisfy in ghb
- a new read will be before any old write on the same variable in ghb

Benchmarks

Cubicle-$\mathcal{W}$ has been evaluated for several kinds of algorithms:

- Mutual exclusion
  - High level : naive mutex, arbitrer, Dekker, Peterson, Burns
  - Assembly code : Spinlock Linux, Mutex/xchg, Mutex/cmpxchg
- Sense-Reversing Barrier
- Two-Phase Commit

Benchmarks : Other Verification Tools for Weak Memory

Parameterized systems :

Dual-TSO (Abdulla, Atig, Bouajjani, Ngo, Univ. Uppsala & Univ. Paris 7)

$\rightarrow$ safety properties

Fix number of processes :

MEMORAX (Abdulla, Atig *et al*, Univ. Uppsala)

$\rightarrow$ safety properties

Trencher (Bouajjani, Calin *et al*, Univ. Paris 7 & Univ. Kaiserslautern)

$\rightarrow$ robustness
$\rightarrow$ bug finding

CBMC (Alglave, Kroening *et al*, Univ. College London & Univ. Oxford)

$\rightarrow$ bug finding
$\rightarrow$ C code analysis

Benchmarks: Results

	Cubicle $\mathcal{W}$	Dual TSO	Memorax PB	Trencher	CBMC Unwind 2
`naive mutex`	0.30s [N]	– TO[5] 35.7s [4]	– TO [11] 2m27 [10]	– TO [6] 54.8s [5]	– TO [5] 2m24 [4]
`lamport`	0.60s [N]	– TO [4] 9.42s [3]	– TO [4] 3m02 [3]	– ✘ [5] 3.37s [4]	– TO [4] 8m39 [3]
`spinlock`	0.06s [N]	TO [N] TO [6] 1m16 [5]	– TO [7] 9m52 [6]	– TO [7] 21.45s [6]	– TO [3] 19.58s [2]
`sense_rev`	0.06s [N]	– TO [3] 0.09s [2]	– TO [3] 0.09s [2]	– TO [5] ☠ [4]	– TO [9] 12m25 [8]
`arbiter_v2`	13.5s [N]	– TO [1+3] 24.2s [1+2]	– TO [1+2]	– ✘ [1+4] 1.62s [1+3]	– TO [1+4] 2m56 [1+3]
`two_phase`	54.1s [N]	– TO [3] 12.3s [2]	– TO [4] 39.7s [3]	– TO [4] ☠ [3]	– TO [11] 12m39 [10]

✘ = crash of the tool ☠ = incorrect answer TO $>$ 20 minutes

Conclusion and Perpectives

Contributions :

- A Model Checker for TSO parameterized systems
- A partial order reduction technique for TSO
- An extension of Cubicle for TSO called Cubicle-$\mathcal{W}$

Perspectives :

- Generation of invariants
- Other memory models

Thank you

Some papers (by *S. Conchon, D. Declerck, F. Zaïdi*)

Parameterized model checking with partial order reduction technique for the TSO weak memory model [JAR 2020, to appear]

Cubicle-$\mathcal{W}$: Parameterized Model Checking on Weak Memory [IJCAR 2018]

Compiling Parameterized x86-TSO Concurrent Programs to Cubicle-$\mathcal{W}$ [ICFEM 2017]

Parameterized Model Checking Modulo Explicit Weak Memory Models [IMPEX 2017]

Cubicle-$\mathcal{W}$: `http://cubicle.lri.fr/cubiclew/`