

EJCP 2017

# Model Checking Modulo Theories with Cubicle

Sylvain Conchon

LRI (UMR 8623), Université Paris-Sud  
Équipe Toccata, INRIA Saclay – Île-de-France



# Cubicle

An SMT based model checker for  
parameterized systems

- ▶ A short tutorial on Cubicle
- ▶ Theoretical foundations
- ▶ Implementation details

# Cubicle : an open source model checker

Cubicle is an **open source** software written in OCaml.

- ▶ It is based on the theoretical framework of **Model Checking Modulo Theories** [Ghilardi, Ranise]
- ▶ Its implementation relies on a lightweight and enhanced version of the SMT solver **Alt-Ergo**

Cubicle is the result of a fruitful collaboration between University of Paris-Sud and Intel corporation

# Cubicle's input language

Inspired by the description language of the Mur $\varphi$  model checker (Dill's group at Stanford)

A program is only represented by a set of **global variables** and **guarded commands**.

# Guarded commands

A guarded command consists of a predicate on the state variables, called a **guard**, and a set of **assignments** that update those variables to change the state.

The control structure is only a single **infinite loop** which repeatedly execute two steps:

1. evaluate all the guards, given the current values of global variables
2. arbitrarily choose one of the commands whose guard is true and execute it, updating the variables

## Example 1 : sequential programs

X, Y = 0

L0:

X := 1;

L1:

Y := X + 1;

L2:

```
type loc = L0 | L1 | L2
```

```
var X : int
```

```
var Y : int
```

```
var PC : loc
```

```
init () { X = 0 && Y = 0 && PC = L0 }
```

```
transition t1 ()
```

```
requires { PC = L0 }
```

```
{ X := 1; PC := L1 }
```

```
transition t2 ()
```

```
requires { PC = L1 }
```

```
{ Y := X + 1; PC := L2 }
```

## Example 2 : granularity

L0:

X := 1;

L1:

Y := X + 1;

L2:



## Example 2 : granularity

L0:

X := 1;

L1:

EAX := X;

L2:

EAX := EAX + 1;

L3:

Y := EAX;

L4:

## Example 2 : granularity

```
L0:  
  X := 1;  
L1:  
  EAX := X;  
L2:  
  EAX := EAX + 1;  
L3:  
  Y := EAX;  
L4:
```

```
type loc = L0 | L1 | L2 | L3 | L4
```

```
var X : int
```

```
var Y : int
```

```
var EAX : int
```

```
var PC : loc
```

```
init () { X = 0 && Y = 0 && PC = L0 }
```

```
transition t1 ()
```

```
requires { PC = L0 }{ X := 1; PC := L1 }
```

```
transition t2 ()
```

```
requires { PC = L1 }{ EAX := X; PC := L2 }
```

```
transition t3 ()
```

```
requires { PC = L2 }  
{ EAX := EAX + 1; PC := L3 }
```

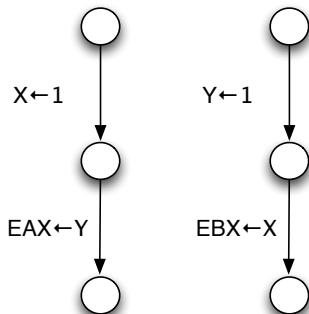
```
transition t4 ()
```

```
requires { PC = L3 }{ Y := EAX; PC := L4 }
```

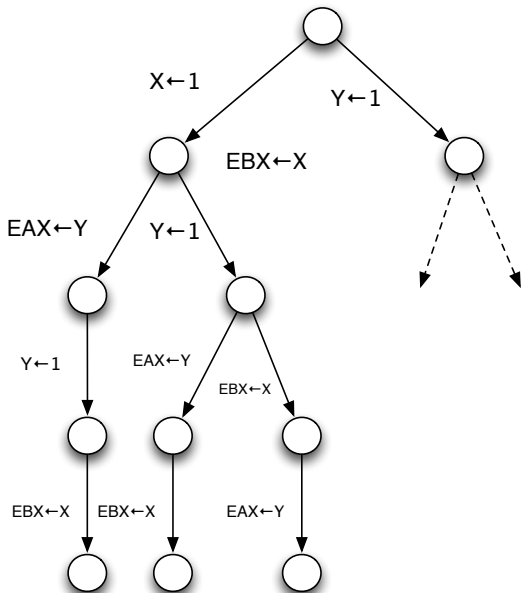
## Example 3 : threads

$X, Y = 0$

Thread 1	Thread 2
L0:	L0:
$X := 1;$	$Y := 1;$
L1:	L1:
$EAX := Y;$	$EBX := X;$
L2:	L2:



# Example 3 : interleaving semantics



## Example 3

```
type location = L0 | L1 | L2
var X : int
var Y : int
var EAX : int
var EBX : int
var PC1 : location
var PC2 : location

init () { PC1 = L0 && PC2 = L0 && X = 0 && Y = 0 }

transition writex_1 ()
requires { PC1 = L0 } { X := 1; PC1 := L1 }

transition ready_1 ()
requires { PC1 = L1 } { EAX := Y; PC1 := L2 }

transition writey_2 ()
requires { PC2 = L0 } { Y := 1; PC2 := L1 }

transition readx_2 ()
requires { PC2 = L1 } { EBX := X; PC2 := L2 }
```

## Example 3 : safety property

- ▶ Characterize **bad** states
- ▶ Use Cubicle to show that bad states cannot be **reach** from initial states

```
type loc = L0 | L1 | L2
var X : int
var Y : int
var EAX : int
var EBX : int
var PC1 : loc
var PC2 : loc
init () { PC1 = L0 && PC2 = L0 && X = 0 && Y = 0 }
unsafe () { PC1 = L2 && PC2 = L2 && EAX = 0 && EBX = 0 }
...

```

# Parameterized systems

Modeling and verifying programs involving an **arbitrary number** of processes

- ▶ Replication of components
- ▶ Unknown or very large number of components

Typical examples : cache coherence protocols, mutual exclusion algorithms, fault-tolerant protocols

# Array-based transition systems

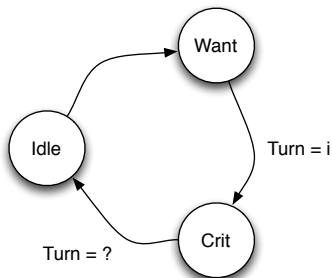
Cubicle handles parameterized systems through :

- ▶ a new built-in data type **proc** (with unspecified cardinality)
- ▶ state variables defined as **arrays** indexed by process identifiers
- ▶ initial states described with a **universally-quantified** formula over processes
- ▶ bad states described with **existentially-quantified** formulas over processes
- ▶ transitions **parameterized** by process identifiers



## Example 4 : array-based transition systems

A Dekker-like mutual exclusion algorithm



```
type st = Idle | Want | Crit  
var Turn : proc  
array S[proc] : st
```

```
init (z) { S[z] = Idle }
```

```
unsafe (x y)  
  { S[x] = Crit && S[y] = Crit }
```

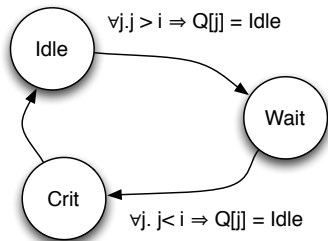
```
transition req (i)  
requires { S[i] = Idle }  
{ S[i] := Want }
```

```
transition enter (i)  
requires { S[i] = Want && Turn = i }  
{ S[i] := Crit }
```

```
transition exit (i)  
requires { S[i] = Crit }  
{ Turn := ? ; S[i] := Idle }
```

## Example 5 : global conditions

A Bakery-like mutual exclusion algorithm



```
type t = Idle | Wait | Crit
```

```
array Q[proc] : t
```

```
init (i) { Q[i]=Idle }
```

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

```
transition wait (i)
```

```
requires { Q[i]=Idle &&
```

```
    forall_other j. (j>i => Q[j]=Idle) }
```

```
{ Q[i] := Wait }
```

```
transition enter (i)
```

```
requires { Q[i]=Wait &&
```

```
    forall_other j. (j<i => Q[j]=Idle) }
```

```
{ Q[i] := Crit }
```

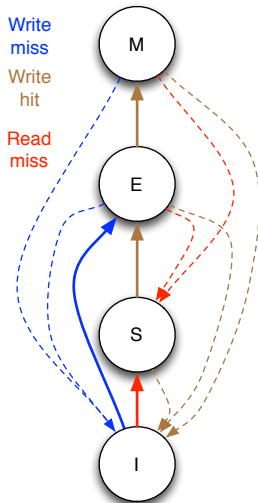
```
transition exit (i)
```

```
requires { Q[i] = Crit }
```

```
{ Q[i] := Idle }
```

# Example 6 : case analysis

## MESI protocol



```
type state = M | E | S | I
```

```
array St[proc]: state
```

```
init (i) { St[i] = I }
```

```
transition read_miss ( i )
```

```
requires { St[i] = I }
```

```
{ St[j] := case | j = i : S  
| St[j] = E : S  
| St[j] = M : S  
| _ : St[j] }
```

```
transition write_miss ( i )
```

```
requires { St[i] = I }
```

```
{ St[j] := case | j = i : E | _ : I }
```

```
transition write_hit_1 ( i )
```

```
requires { St[i] = E } { St[i] := M }
```

```
transition write_hit_2 ( i )
```

```
requires { St[i] = S }
```

```
{ St[j] := case | j = i : E | _ : I }
```

# Language limitations

Cubicle's input language is still limited; it could be further improved.

- ▶ **data types** (records, several parameterized data types)
- ▶ **programming constructs** (sequences, loops)
- ▶ **arithmetic expressions**

# Theoretical foundation : MCMT

Cubicle implements the **Model Checking Modulo Theories** (MCMT) framework [Ghilardi, Ranise] where system states and transitions are defined as first-order formulas

# Theoretical foundation : MCMT

Cubicle implements the **Model Checking Modulo Theories** (MCMT) framework [Ghilardi, Ranise] where system states and transitions are defined as first-order formulas

- ▶ Initial states are defined by a **universally** quantified formula

# Theoretical foundation : MCMT

Cubicle implements the **Model Checking Modulo Theories** (MCMT) framework [Ghilardi, Ranise] where system states and transitions are defined as first-order formulas

- ▶ Initial states are defined by a **universally** quantified formula  
**init** (i) { A[i] = True && PC = L1 }

# Theoretical foundation : MCMT

Cubicle implements the **Model Checking Modulo Theories** (MCMT) framework [Ghilardi, Ranise] where system states and transitions are defined as first-order formulas

- ▶ Initial states are defined by a **universally** quantified formula

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

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



# Theoretical foundation : MCMT

Cubicle implements the **Model Checking Modulo Theories** (MCMT) framework [Ghilardi, Ranise] where system states and transitions are defined as first-order formulas

- ▶ Initial states are defined by a **universally** quantified formula  
**init** (i) { A[i] = True && PC = L1 }

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

- ▶ Bad states are defined by special **existentially** quantified formulas, called **cubes**

# Theoretical foundation : MCMT

Cubicle implements the **Model Checking Modulo Theories** (MCMT) framework [Ghilardi, Ranise] where system states and transitions are defined as first-order formulas

- ▶ Initial states are defined by a **universally** quantified formula  
**init** (i) { A[i] = True && PC = L1 }

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

- ▶ Bad states are defined by special **existentially** quantified formulas, called **cubes**

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

# Theoretical foundation : MCMT

Cubicle implements the **Model Checking Modulo Theories** (MCMT) framework [Ghilardi, Ranise] where system states and transitions are defined as first-order formulas

- ▶ Initial states are defined by a **universally** quantified formula  
**init** (i) { A[i] = True && PC = L1 }

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

- ▶ Bad states are defined by special **existentially** quantified formulas, called **cubes**

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

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

# Theoretical foundation : MCMT

Cubicle implements the **Model Checking Modulo Theories** (MCMT) framework [Ghilardi, Ranise] where system states and transitions are defined as first-order formulas

- ▶ Initial states are defined by a **universally** quantified formula  
**init** (i) { A[i] = True && PC = L1 }

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

- ▶ Bad states are defined by special **existentially** quantified formulas, called **cubes**

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

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

- ▶ Transitions correspond to **existentially** quantified formulas

# Theoretical foundation : MCMT

Cubicle implements the **Model Checking Modulo Theories** (MCMT) framework [Ghilardi, Ranise] where system states and transitions are defined as first-order formulas

- ▶ Initial states are defined by a **universally** quantified formula  
**init** (i) { A[i] = True && PC = L1 }

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

- ▶ Bad states are defined by special **existentially** quantified formulas, called **cubes**

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

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

- ▶ Transitions correspond to **existentially** quantified formulas

**transition** t(i)

**requires** { S[i] = A && PC = L1 }

{ S[i] = B; X = X+1 }

# Theoretical foundation : MCMT

Cubicle implements the **Model Checking Modulo Theories** (MCMT) framework [Ghilardi, Ranise] where system states and transitions are defined as first-order formulas

- ▶ Initial states are defined by a **universally** quantified formula  
**init** (i) { A[i] = True && PC = L1 }

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

- ▶ Bad states are defined by special **existentially** quantified formulas, called **cubes**

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

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

- ▶ Transitions correspond to **existentially** quantified formulas

**transition** t(i)

**requires** { S[i] = A && PC = L1 }

{ S[i] = B; X = X+1 }

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

# Inductive invariants

We are looking for a predicate **Reach** such that :

Reach is an **inductive invariant** :

$$\begin{aligned}\forall \vec{x}. \text{Init}(\vec{x}) &\Rightarrow \text{Reach}(\vec{x}) \\ \forall \vec{x}, \vec{x}'. \text{Reach}(\vec{x}) \wedge \tau(\vec{x}, \vec{x}') &\Rightarrow \text{Reach}(\vec{x}')\end{aligned}$$

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

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

## Example 7 : Back to Dekker-like algorithm

```
type st = Idle | Want | Crit
var T : proc
array S[proc] : st

init (z) { S[z] = Idle }

unsafe (x y) {
  S[x] = Crit && S[y] = Crit
}

transition req (i)
requires { S[i] = Idle }
{ S[i] := Want }
```

```
transition enter (i)
requires { S[i] = Want &&
           T = i }
{ S[i] := Crit }
```

```
transition exit (i)
requires { S[i] = Crit }
{ T := ? ;
  S[i] := Idle }
```



# A pure SMT problem

```
type st = Idle | Want | Crit
type proc
logic Reach : proc, (proc,st) farray -> prop

axiom init :
  forall t:proc. forall s:(proc,st) farray. (forall z:proc. s[z]=Idle) -> Reach(t,s)

axiom req :
  forall t,t':proc. forall s,s':(proc,st) farray.
  Reach(t,s) and ( exists i:proc. s[i]=Idle and s'=s[i<-Want] and t'=t)
  -> Reach(t',s')

axiom enter :
  forall t,t':proc. forall s,s':(proc,st) farray.
  Reach(t,s) and (exists i:proc. s[i]=Want and t=i and s'= s[i<-Crit] and t'=t)
  -> Reach(t',s')

axiom exit :
  forall t,t':proc. forall s,s':(proc,st) farray.
  Reach(t,s) and ( exists i:proc. s[i]=Crit and s'=s[i<-Idle]) -> Reach(t',s')

goal unsafe :
  exists t:proc. exists s:(proc,st) farray. exists i,j:proc.
  i<>j and Reach(t,s) and s[i]=Crit and s[j]=Crit
```

# Inductive Invariants (1)

```
var X : int  
init () { X = 0 }
```

```
unsafe () { X = 9 }
```

```
transition t ()  
{ X := X + 2 }
```

```
axiom a1 : Reach(0)
```

```
axiom a2 :  
  forall x:int. Reach(x)->Reach(x+2)
```

```
goal unsafe :  
exists x. Reach(x) and x = 9
```

# Inductive Invariants (1)

```
var X : int  
init () { X = 0 }
```

```
unsafe () { X = 9 }
```

```
transition t ()  
{ X := X + 2 }
```

```
axiom a1 : Reach(0)
```

```
axiom a2 :  
forall x:int. Reach(x) -> Reach(x+2)
```

```
goal unsafe :  
exists x. Reach(x) and x = 9
```

A possible interpretation for Reach could be  $\{0, 2, 4, 6, \dots\}$

# Inductive Invariants (1)

```
var X : int  
init () { X = 0 }
```

```
unsafe () { X = 9 }
```

```
transition t ()  
{ X := X + 2 }
```

```
axiom a1 : Reach(0)
```

```
axiom a2 :  
  forall x:int. Reach(x)->Reach(x+2)
```

```
goal unsafe :  
  exists x. Reach(x) and x = 9
```

A possible interpretation for Reach could be  $\{0, 2, 4, 6, \dots\}$

or  $\{0, 2, 4, 6, 8, 10, 11, 12, 13, \dots\}$

# Inductive Invariants (1)

```
var X : int
init () { X = 0 }
```

```
unsafe () { X = 9 }
```

```
transition t ()
{ X := X + 2 }
```

```
axiom a1 : Reach(0)
```

```
axiom a2 :
  forall x:int. Reach(x)->Reach(x+2)
```

```
goal unsafe :
  exists x. Reach(x) and x = 9
```

A possible interpretation for Reach could be  $\{0, 2, 4, 6, \dots\}$

or  $\{0, 2, 4, 6, 8, 10, 11, 12, 13, \dots\}$  but not  $\mathbb{N}$  nor  $\{0, 2, 4, 5, 6, 7, \dots\}$

# Inductive Invariants (1)

```
var X : int
init () { X = 0 }

unsafe () { X = 9 }

transition t ()
{ X := X + 2 }
```

```
axiom a1 : Reach(0)
axiom a2 :
  forall x:int. Reach(x)->Reach(x+2)

goal unsafe :
  exists x. Reach(x) and x = 9
```

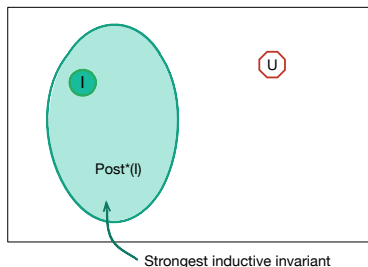
A possible interpretation for Reach could be  $\{0, 2, 4, 6, \dots\}$

or  $\{0, 2, 4, 6, 8, 10, 11, 12, 13, \dots\}$  but not  $\mathbb{N}$  nor  $\{0, 2, 4, 5, 6, 7, \dots\}$

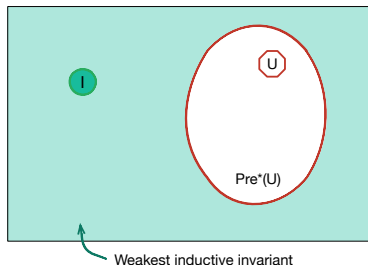
How to find interpretations for inductive invariants ?

## Inductive invariants (2)

Set of **reachable** states from **Init**  
= **Strongest** inductive invariant



Set of states that **cannot reach**  
**Unsafe**  
= **Weakest** inductive invariant



# Backward Reachability (BR) : How to compute $\text{Pre}^*(U)$

$I$  : initial states     $U$  : unsafe states (cube)     $\mathcal{T}$  : transitions

---

BR ():

$V := \emptyset$ ;

push(Q, U);

**while** not\_empty(Q) **do**

$\varphi := \text{pop}(Q)$

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

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

$V := V \cup \{\varphi\}$

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

**return** safe



# Backward Reachability (BR) : How to compute $\text{Pre}^*(U)$

$I$  : initial states     $U$  : unsafe states (cube)     $\mathcal{T}$  : transitions

---

BR ():

$V := \emptyset$ ;

push(Q, U);

**while** not\_empty(Q) **do**

$\varphi := \text{pop}(Q)$

**if**  $\varphi \wedge I$  sat **then return** unsafe (\* SMT check \*)

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

$V := V \cup \{\varphi\}$

    push(Q, pre $_{\mathcal{T}}$ ( $\varphi$ ))

**return** safe

# Pre-images

Given a transition  $t \in \mathcal{T}$ ,  $\text{pre}_t(\varphi)$  is a formula that describes the set of states from which a  $\varphi$ -state is reachable in one  $t$ -step

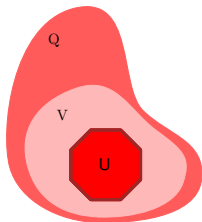
$$\text{pre}_{\mathcal{T}}(\varphi) = \bigvee_{t \in \mathcal{T}} \text{pre}_t(\varphi)$$

If  $\varphi$  is a cube, then  $\text{pre}_{\mathcal{T}}(\varphi)$  can also be represented as a union of cubes

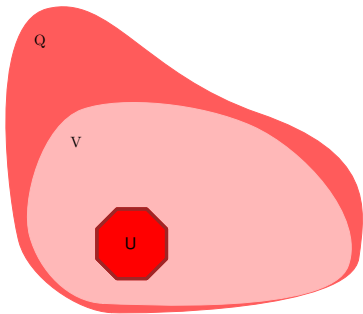
# Running BR



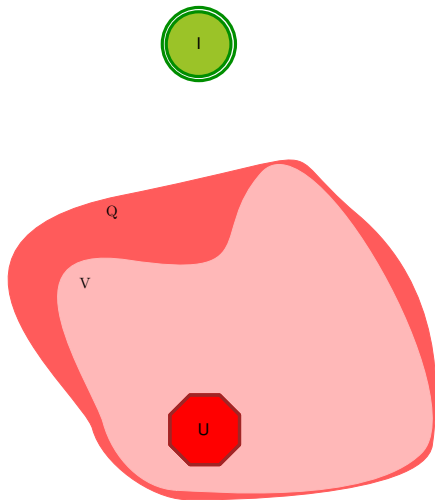
# Running BR



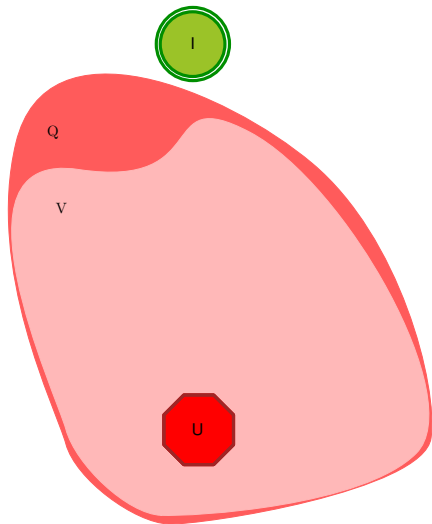
# Running BR



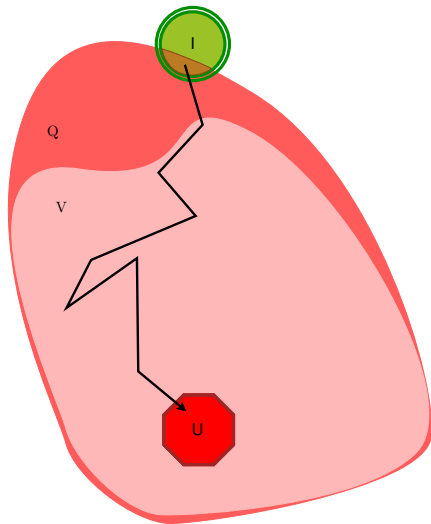
# Running BR



# Running BR

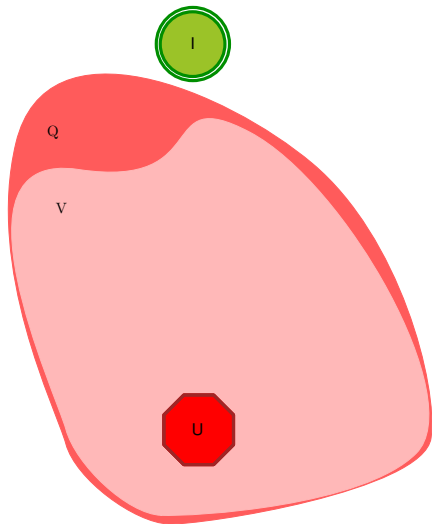


# Running BR

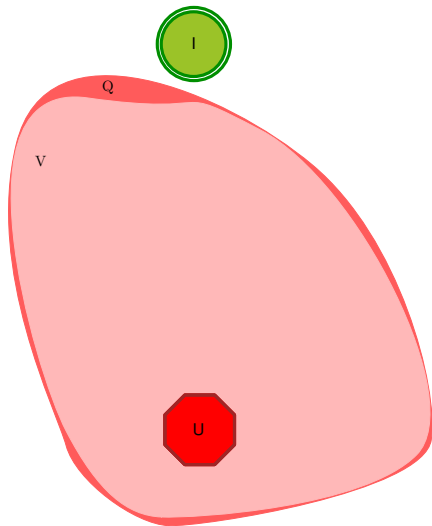




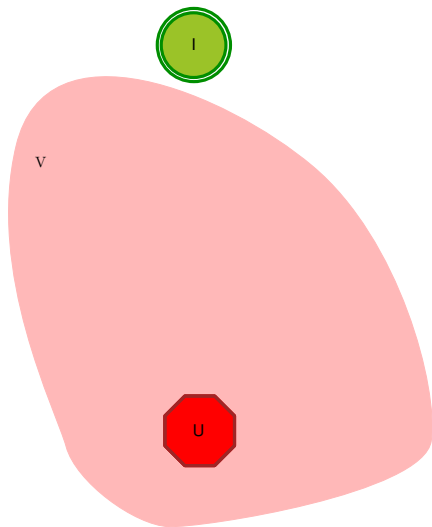
# Running BR



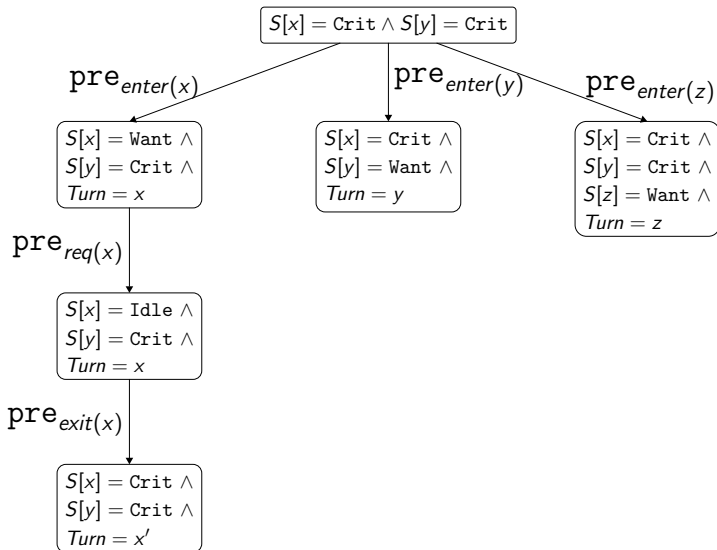
# Running BR



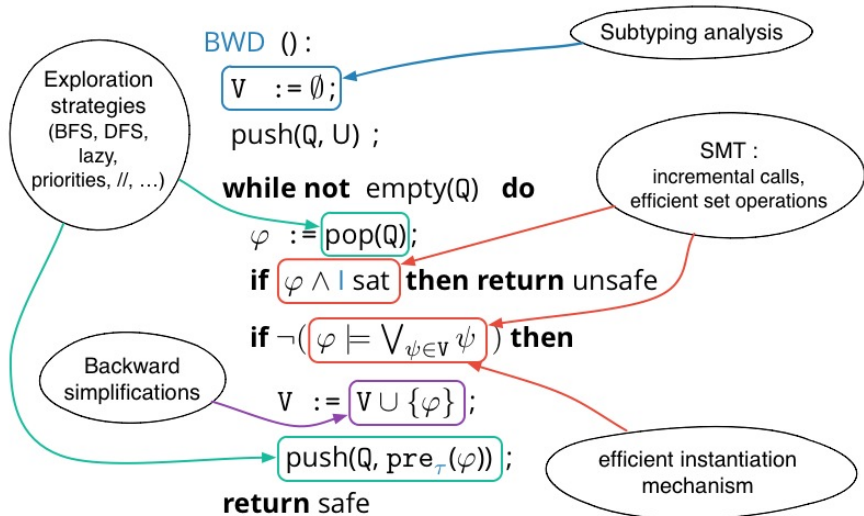
# Running BR



# BR : example



# Implementation issues and optimizations



# Fixpoint computation

```
V :=  $\emptyset$ 
push(Q, U)
while not_empty(Q) do
   $\varphi := \text{pop}(Q)$ 
  if  $\varphi \wedge I \text{ sat}$  then return(unsafe)
  if  $\neg(\varphi \models \bigvee_{\psi \in V} \psi)$  then
    V :=  $V \cup \{\varphi\}$ 
    push(Q,  $\text{pre}_{\mathcal{T}}(\varphi)$ )
return(safe)
```

# Fixpoint computation: a challenge

$$\varphi \models \bigvee_{\psi \in \mathbf{V}} \psi$$

# Fixpoint computation: a challenge

$$\exists \bar{x}. F \models \bigvee_{\psi \in V} \exists \bar{y}. G_{\psi}$$



# Fixpoint computation: a challenge

$$\exists \bar{x}. F \wedge \bigwedge_{\psi \in V} \forall \bar{y}. \neg G_{\psi} \quad \text{satisfiable?}$$

# Fixpoint computation: a challenge

$$\exists \bar{x}. F \wedge \bigwedge_{\psi \in \mathbf{V}} \forall \bar{y}. \neg G_{\psi} \quad \text{satisfiable ?}$$

$F$  and  $G_{\psi}$  are conjunctions of literals involving **several theories**  
(uninterpreted function symbols, linear arithmetic, enumerations)

# Fixpoint computation: a challenge

$$\exists \bar{x}. F \wedge \bigwedge_{\psi \in V} \bigwedge_{\sigma \in \Sigma} (\neg G_{\psi}) \sigma \quad \text{satisfiable ?}$$

# Fixpoint computation: a challenge

$$\exists \bar{x}. F \wedge \bigwedge_{\psi \in V} \bigwedge_{\sigma \in \Sigma} (\neg G_{\psi})\sigma \quad \text{satisfiable?}$$

Suppose  $|V| = 20000$  and  $|\Sigma| = 120$  (e.g.  $|\bar{x}| = |\bar{y}| = 5$ ) then

$$\left| \bigwedge_{\psi \in V} \bigwedge_{\sigma \in \Sigma} (\neg G_{\psi})\sigma \right| \sim 2 \cdot 10^6 \text{ clauses}$$

# Fixpoint computation: optimizations

▶ **Fast checks:**  $G_\psi\sigma \subseteq F$

▶ **Irrelevant permutations:**

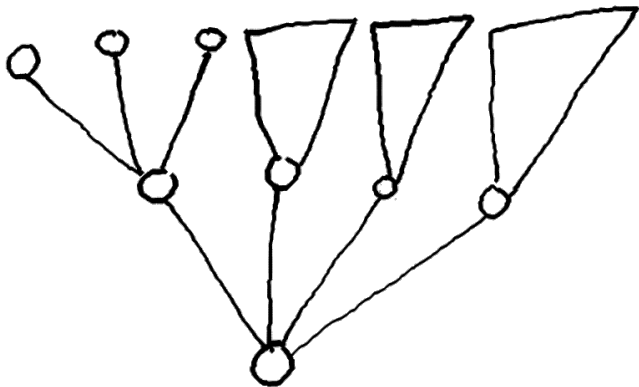
$L \in G_\psi\sigma$  and  $L' \in F$  and  $\neg(L \wedge L')$  is immediate

▶ A **single** SMT-context is used for each fixpoint check; it just gets **incremented** and repeatedly verified

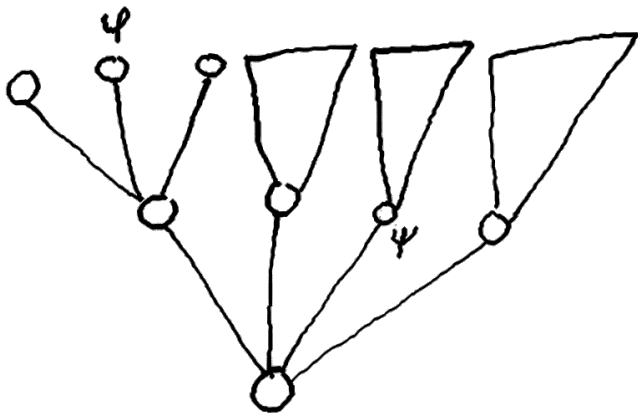
# Node deletion

```
V :=  $\emptyset$ 
push(Q, U)
while not_empty(Q)
   $\varphi := \text{pop}(Q)$ 
  if  $\varphi \wedge I \text{ sat}$  then return(unsafe)
  if  $\neg(\varphi \models \bigvee_{\psi \in V} \psi)$  then
    V := V  $\cup \{\varphi\}$ 
    push(Q, pre( $\varphi$ ))
return(safe)
```

## Node deletion : backward simplification

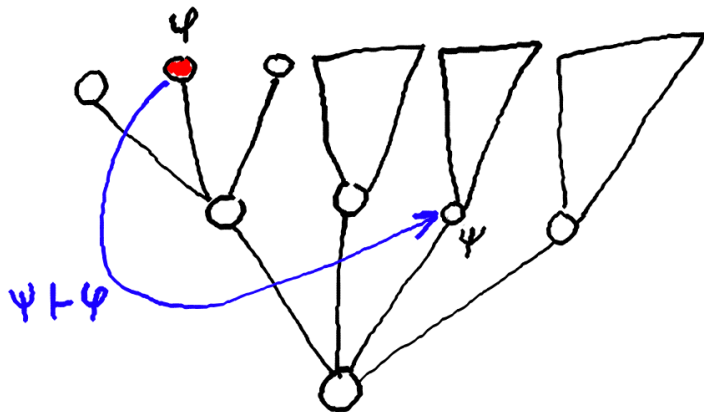


# Node deletion : backward simplification

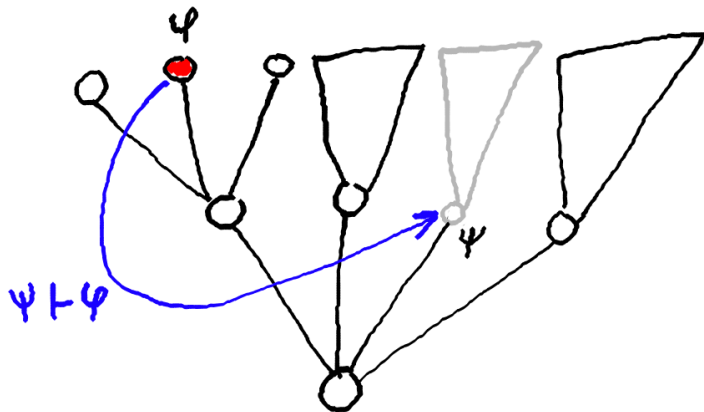




# Node deletion : backward simplification



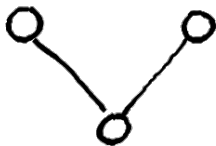
# Node deletion : backward simplification



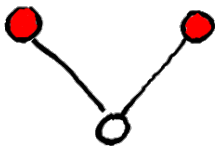
# Towards a concurrent implementation

- ▶ Based on the Functor Library [Filliâtre, Krishnamani] :  
master / worker architecture
- ▶ Search can be parallelized:
  - Expensive tasks = fixpoint checks
  - Synchronization to keep a precise guidance (BFS)
  - Deletion becomes dangerous

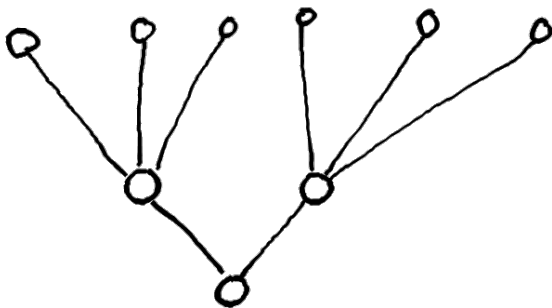
## Node deletion in parallel BFS



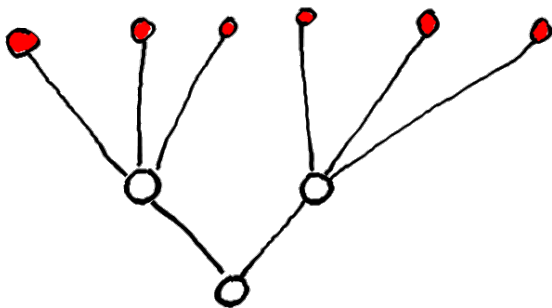
## Node deletion in parallel BFS



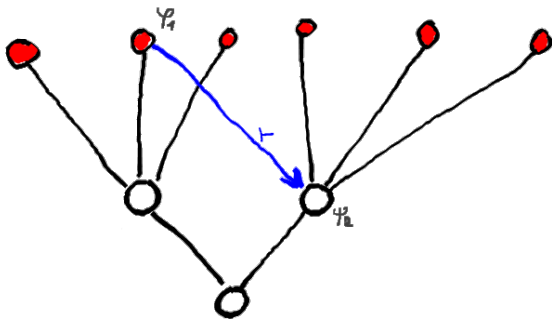
## Node deletion in parallel BFS



## Node deletion in parallel BFS

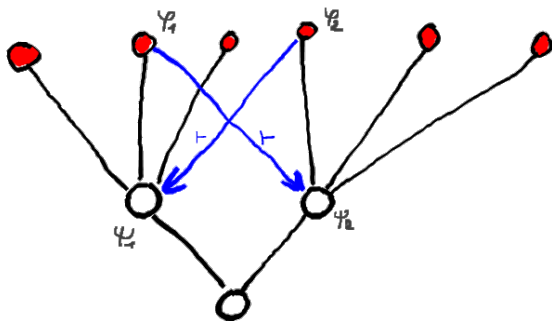


# Node deletion in parallel BFS





# Node deletion in parallel BFS



End of part II

Thank you !