

ENS 2016 - 2017

Model Checking Modulo Theories (MCMT)

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
- ▶ (Extra materials)

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 ϕ 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 }
```

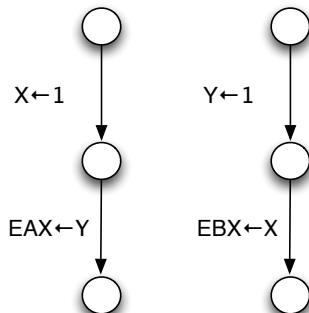
```
transition t4 ()
```

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

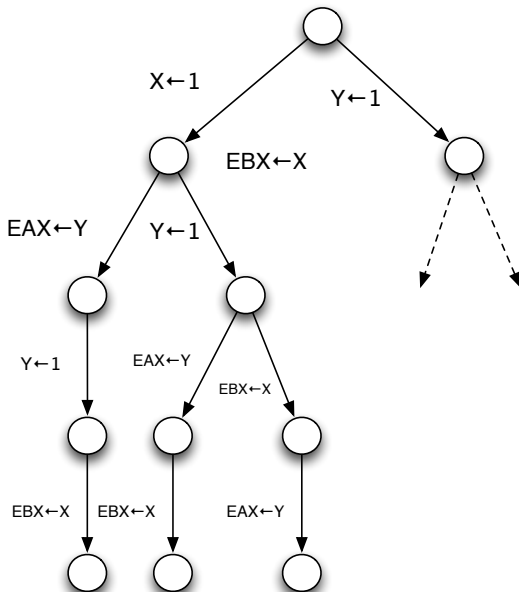
Example 3 : threads

$X, Y = 0$

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

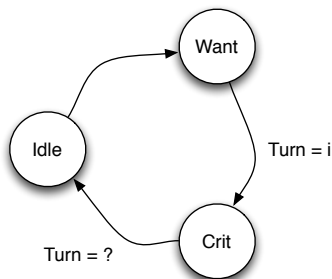
Array-based transition systems

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 5 : array-based transition systems

A Dekker-like mutual exclusion algorithm



```
var Turn : proc  
array Req[proc] : bool  
array CS[proc] : bool
```

```
init (i) { Req[i]=False && CS[i]=False }  
unsafe (i j) { CS[i]=True && CS[j]=True }
```

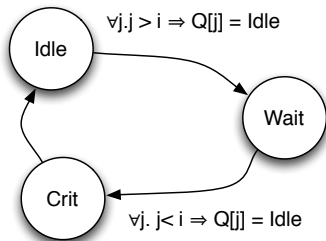
```
transition req (i)  
requires { Req[i] = False }  
{ Req[i] := True }
```

```
transition enter (i)  
requires { Req[i]=True && CS[i]=False && Turn=i }  
{ CS[i] := True }
```

```
transition exit (i)  
requires { CS[i]=True }  
{ Turn := ? ;  
  CS[i] := False;  
  Req[i] := False }
```

Example 5 : global conditions

A Bakery-like mutual exclusion algorithm



type $t = \text{Idle} \mid \text{Wait} \mid \text{Crit}$

array $Q[\text{proc}] : t$

init $(i) \{ Q[i] = \text{Idle} \}$

unsafe $(i \ j) \{ Q[i] = \text{Crit} \ \&\& \ Q[j] = \text{Crit} \}$

transition $\text{wait} \ (i)$

requires $\{ Q[i] = \text{Idle} \ \&\&$

$\text{forall_other } j. (j < i \mid \mid Q[j] = \text{Idle}) \}$

$\{ Q[i] := \text{Wait} \}$

transition $\text{enter} \ (i)$

requires $\{ Q[i] = \text{Wait} \ \&\&$

$\text{forall_other } j. (i < j \mid \mid Q[j] = \text{Idle}) \}$

$\{ Q[i] := \text{Crit} \}$

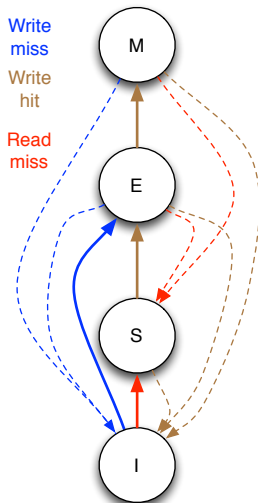
transition $\text{exit} \ (i)$

requires $\{ Q[i] = \text{Crit} \}$

$\{ Q[i] := \text{Idle} \}$

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 }

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

- 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** :

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

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

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

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

Example 7 : Another version of a 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 }
```


Example

```
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
```

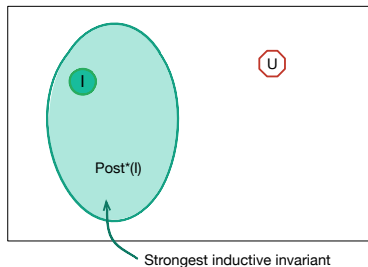
A possible interpretation for Reach could be $\{0, 2, 4, 6, \dots\}$

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

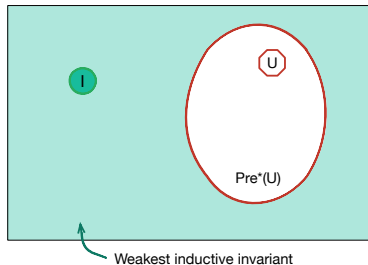
How to find interpretations for inductive invariants ?

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

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 φ -state is reachable in one t -step

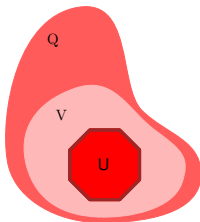
$$\text{pre}_{\mathcal{T}}(\varphi) = \bigvee_{t \in \mathcal{T}} \text{pre}_t(\varphi)$$

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

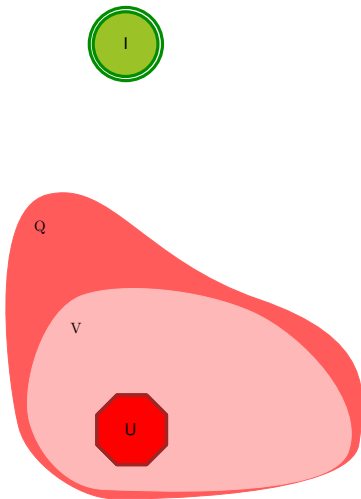
Running BR



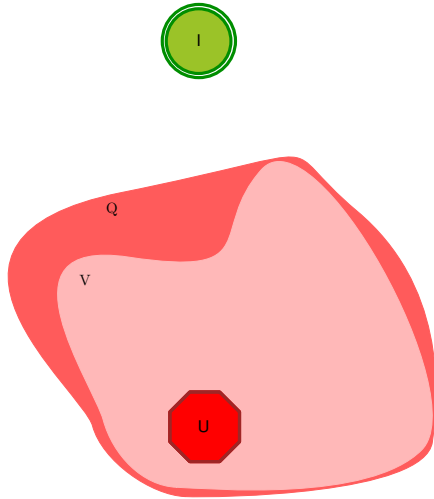
Running BR



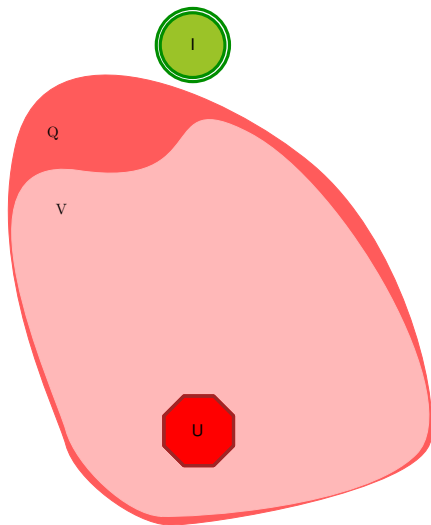
Running BR



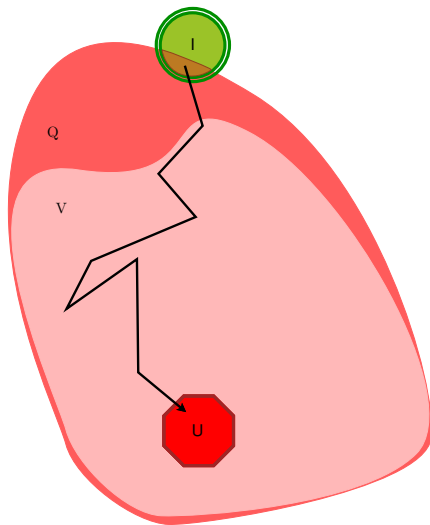
Running BR



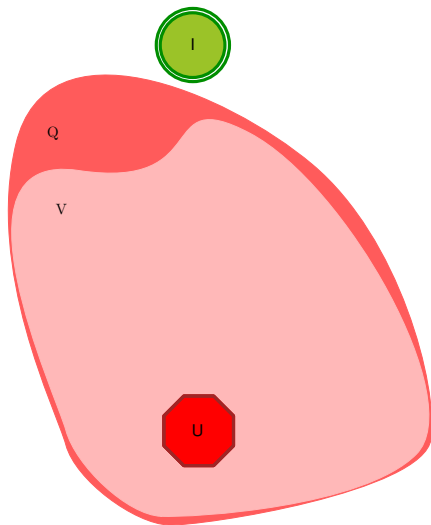
Running BR



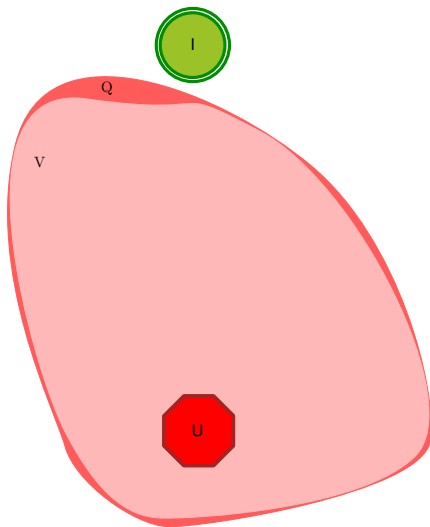
Running BR



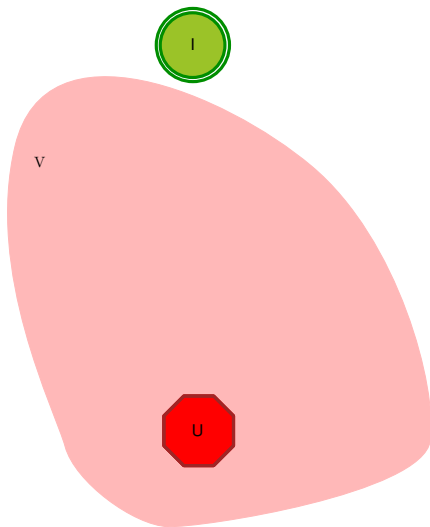
Running BR



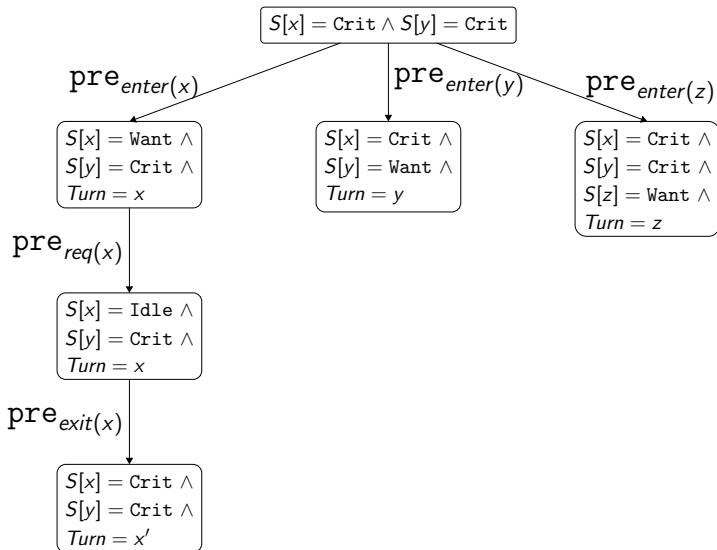
Running BR



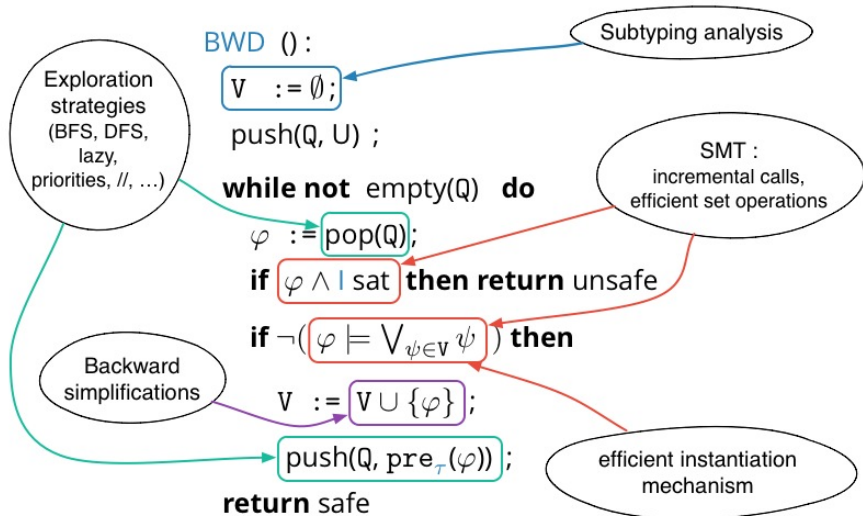
Running BR



BR : example



Implementation issues and optimizations



Fixpoint computation

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

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

Fixpoint computation: a challenge

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

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_{ψ} are conjunctions of literals involving **several theories**
(uninterpreted function symbols, linear arithmetic, enumerations)

Fixpoint computation: a challenge

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

Fixpoint computation: a challenge

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

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

$$\left| \bigwedge_{\psi \in \mathbf{V}} \bigwedge_{\sigma \in \Sigma} (\neg G_{\psi})\sigma \right| \sim 2.10^6 \text{ clauses}$$

Fixpoint computation: optimizations

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

- ▶ **Irrelevant permutations:**

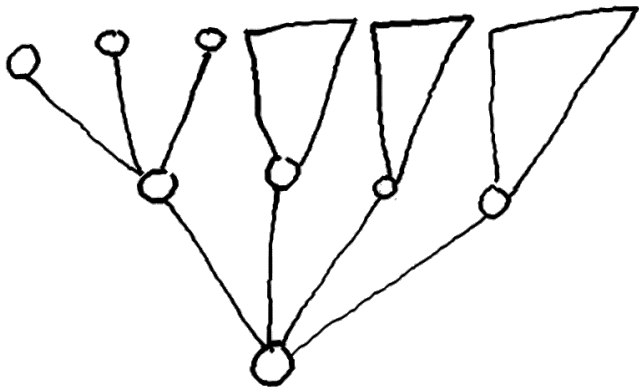
$L \in G_\psi\sigma$ and $L' \in F$ and $\neg(L \wedge L')$ is immediate

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

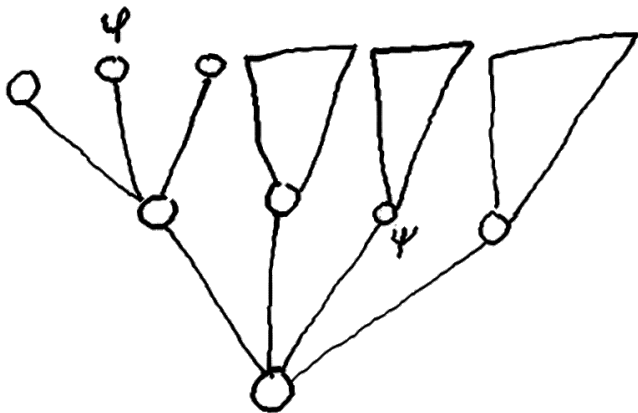
Node deletion

```
V :=  $\emptyset$ 
push(Q, U)
while not_empty(Q)
   $\varphi$  := 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, pre( $\varphi$ ))
return(safe)
```

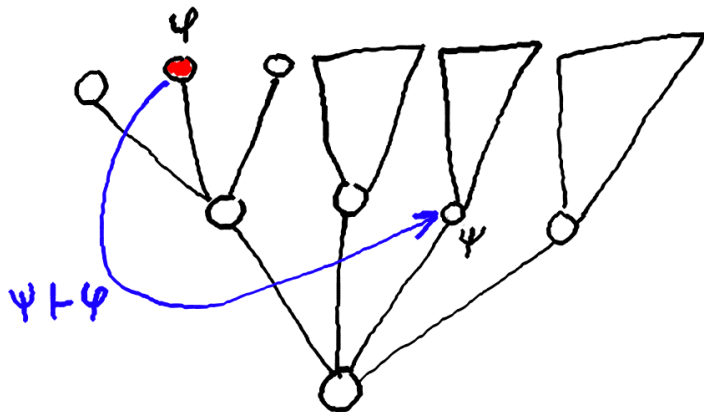
Node deletion : backward simplification



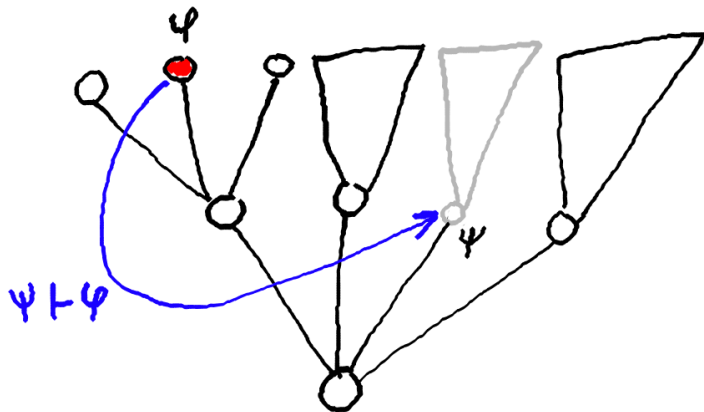
Node deletion : backward simplification



Node deletion : backward simplification



Node deletion : backward simplification

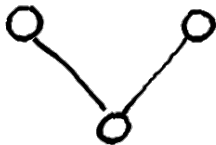


Towards a concurrent implementation

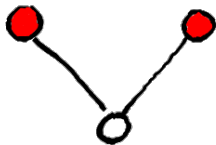
`cubicle -j`

- ▶ 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

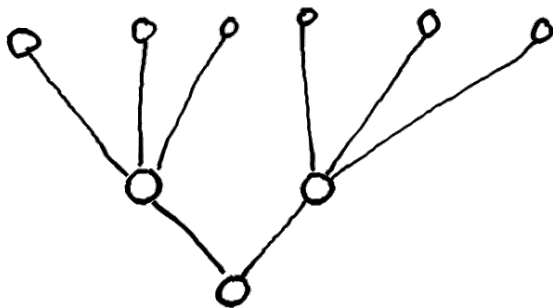
Node deletion in parallel BFS



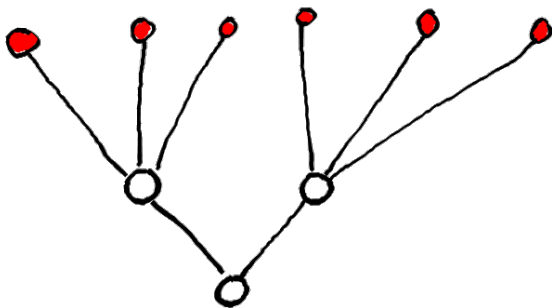
Node deletion in parallel BFS



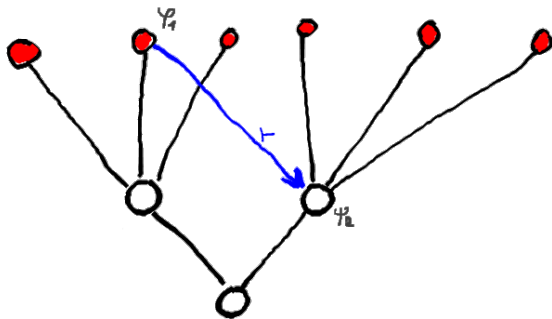
Node deletion in parallel BFS



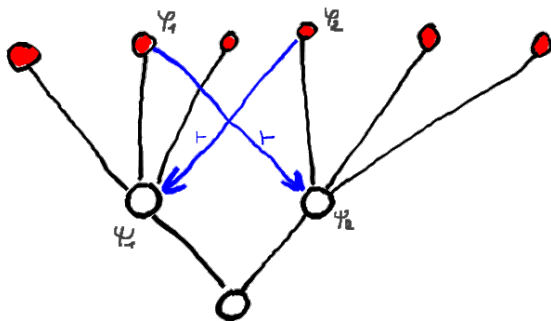
Node deletion in parallel BFS



Node deletion in parallel BFS



Node deletion in parallel BFS



Spurious Traces

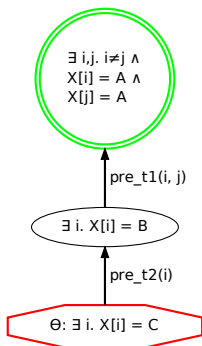
Crash Failure Model

$$\left\{ \begin{array}{l} \text{type } t = A \mid B \mid C \\ \forall i. X[i] = A \quad (\text{inital states}) \\ t_1 : \exists i, j. i \neq j \wedge X[i] = A \wedge X[j] = A \wedge X'[i] = B \\ t_2 : \exists i. X[i] = B \wedge \forall j. j \neq i \implies X[j] \neq A \wedge X'[i] = C \end{array} \right.$$

Spurious Traces

Crash Failure Model

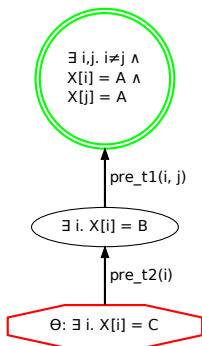
$$\left\{ \begin{array}{l} \text{type } t = A \mid B \mid C \\ \forall i. X[i] = A \quad (\text{initial states}) \\ t_1 : \exists i, j. i \neq j \wedge X[i] = A \wedge X[j] = A \wedge X'[i] = B \\ t_2 : \exists i. X[i] = B \wedge \forall j. j \neq i \implies X[j] \neq A \wedge X'[i] = C \end{array} \right.$$



Spurious Traces

Crash Failure Model

$$\left\{ \begin{array}{l} \text{type } t = A \mid B \mid C \\ \forall i. X[i] = A \quad (\text{initial states}) \\ t_1 : \exists i, j. i \neq j \wedge X[i] = A \wedge X[j] = A \wedge X'[i] = B \\ t_2 : \exists i. X[i] = B \wedge \forall j. j \neq i \implies X[j] \neq A \wedge X'[i] = C \end{array} \right.$$

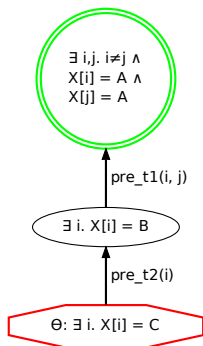


the program **could** contain only **one** thread

Spurious Traces

Crash Failure Model

$$\left\{ \begin{array}{l} \text{type } t = A \mid B \mid C \\ \forall i. X[i] = A \quad (\text{initial states}) \\ t_1 : \exists i, j. i \neq j \wedge X[i] = A \wedge X[j] = A \wedge X'[i] = B \\ t_2 : \exists i. X[i] = B \wedge \forall j. j \neq i \implies X[j] \neq A \wedge X'[i] = C \end{array} \right.$$



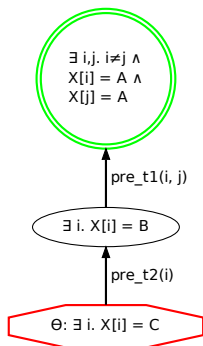
the program contains **at least two** threads

the program **could** contain only **one** thread

Spurious Traces

Crash Failure Model

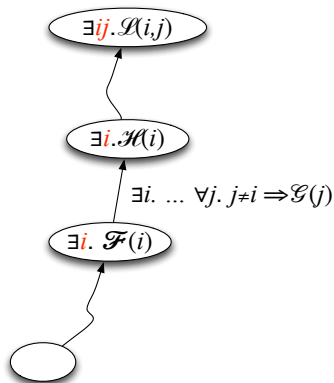
$$\left\{ \begin{array}{l} \text{type } t = A \mid B \mid C \\ \forall i. X[i] = A \quad (\text{initial states}) \\ t_1 : \exists i, j. i \neq j \wedge X[i] = A \wedge X[j] = A \wedge X'[i] = B \\ t_2 : \exists i. X[i] = B \wedge \forall j. j \neq i \implies X[j] \neq A \wedge X'[i] = C \end{array} \right.$$



the program contains **at least two** threads

the program **could** contain only **one** thread
thus, t_2 is not possible from that state

How to Refine the Crash Failure Model ?



How to Refine the Crash Failure Model ?

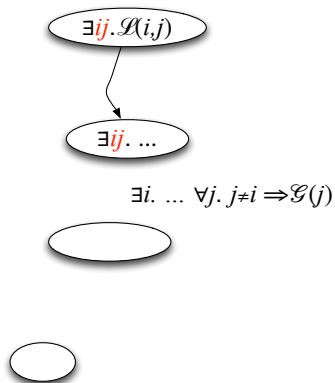
$$\exists ij. \mathcal{A}(i, j)$$



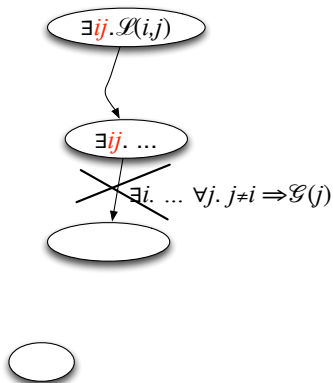
$$\exists i. \dots \forall j. j \neq i \Rightarrow \mathcal{G}(j)$$



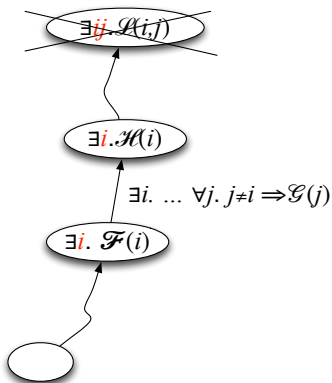
How to Refine the Crash Failure Model ?



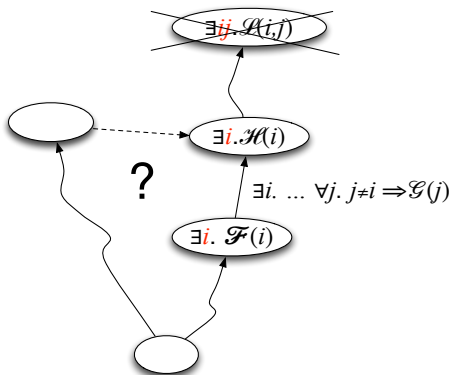
How to Refine the Crash Failure Model ?



How to Refine the Crash Failure Model ?



How to Refine the Crash Failure Model ?



How to scale?

Cubicle's benchmarks on academic problems are promising

	Cubicle	CMurphi		
Szymanski_at	0.30s	8.04s (8)	5m12s (10)	2h50m (12)
German_Baukus	7.03s	0.74s (4)	19m35s (8)	4h49m (10)
German.CTC	3m23s	1.83s (4)	43m46s (8)	12h35m (10)
German_pfs	3m58s	0.99s (4)	22m56s (8)	5h30m (10)
Chandra-Toueg	2h01m	5.68s (4)	2m58s (5)	1h36m (6)

How to scale?

But how to scale up on **industrial-like** problems?

	Cubicle	CMurphi		
Szymanski_at	0.30s	8.04s (8)	5m12s (10)	2h50m (12)
German_Baukus	7.03s	0.74s (4)	19m35s (8)	4h49m (10)
German.CTC	3m23s	1.83s (4)	43m46s (8)	12h35m (10)
German_pfs	3m58s	0.99s (4)	22m56s (8)	5h30m (10)
Chandra-Toueg	2h01m	5.68s (4)	2m58s (5)	1h36m (6)
Szymanski_na	T.O.	0.88s (4)	8m25s (6)	7h08m (8)
Flash_nodata	O.M.	4.86s (3)	3m33s (4)	2h46m (5)
Flash	O.M.	1m27s (3)	2h15m (4)	O.M. (5)

O.M. > 20 GB

T.O. > 20 h

What Industrial-Like Means (for us)

A well-known candidate : the **FLASH** protocol (stanford multiprocessor architecture — 1994)

- ▶ Cache-coherence shared memory
- ▶ High-performance message passing
- ▶ **67 million** states for 4 processes (~ 40 variables, ~ 75 transitions)

Our Solution to Scale Up

We designed a new core algorithm for Cubicle that

- ▶ infers **invariants** for parameterized case using **finite** instances
- ▶ inserts and checks them **on the fly** in a backward reachability loop
- ▶ **backtracks** if necessary

BRAB (Backward Reachability Algorithm with Approximations)
[FMCAD 2013]

Example: German-*ish* cache coherence protocol

Client i :

$\text{Cache}[i] \in \{E, S, I\}$

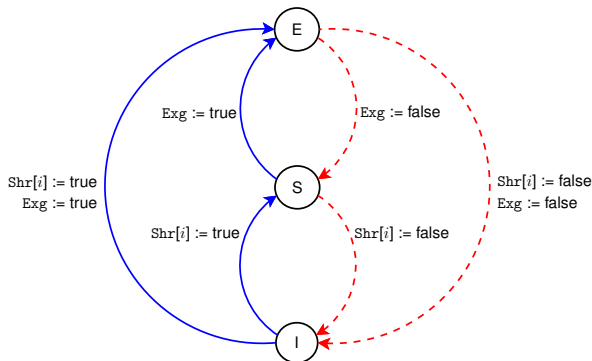
Directory:

$\text{Cmd} \in \{\text{rs}, \text{re}, \epsilon\}$

$\text{Ptr} \in \text{proc}$

$\text{Shr}[i] \in \{\text{true}, \text{false}\}$

$\text{Exg} \in \{\text{true}, \text{false}\}$



Initial states: $\forall i. \text{Cache}[i] = I \wedge \neg \text{Shr}[i] \wedge \neg \text{Exg} \wedge \text{Cmd} = \epsilon$

Unsafe states: $\exists i, j. i \neq j \wedge \text{Cache}[i] = E \wedge \text{Cache}[j] \neq I$?
(cubes)

Example: German-*ish* cache coherence protocol

Client i :

$\text{Cache}[i] \in \{E, S, I\}$

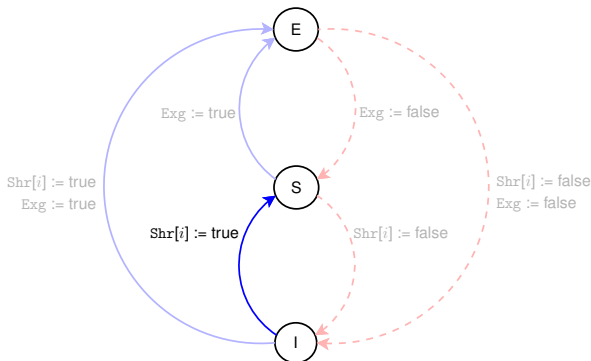
Directory:

$\text{Cmd} \in \{\text{rs}, \text{re}, \epsilon\}$

$\text{Ptr} \in \text{proc}$

$\text{Shr}[i] \in \{\text{true}, \text{false}\}$

$\text{Exg} \in \{\text{true}, \text{false}\}$

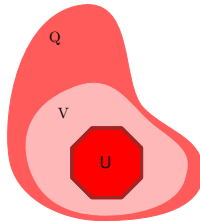


$t_5 : \exists i. \quad \text{Ptr} = i \wedge \text{Cmd} = \text{rs} \wedge \neg \text{Exg} \wedge$
 $\text{Cmd}' = \epsilon \wedge \text{Shr}'[i] \wedge \text{Cache}'[i] = S$

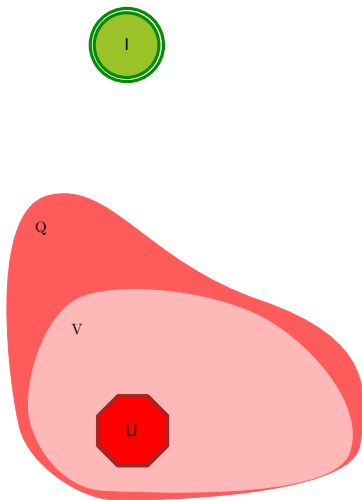
Backward reachability algorithm



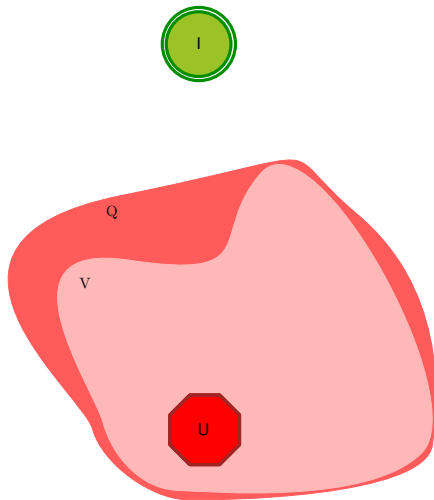
Backward reachability algorithm



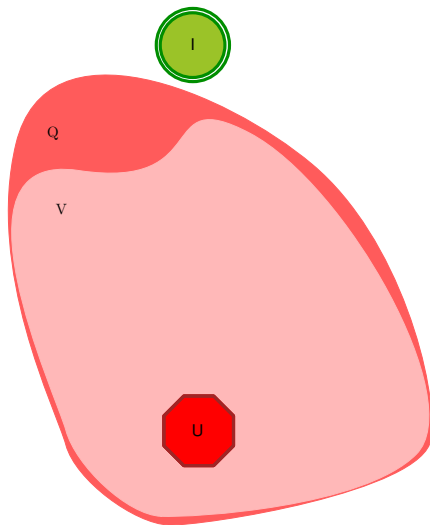
Backward reachability algorithm



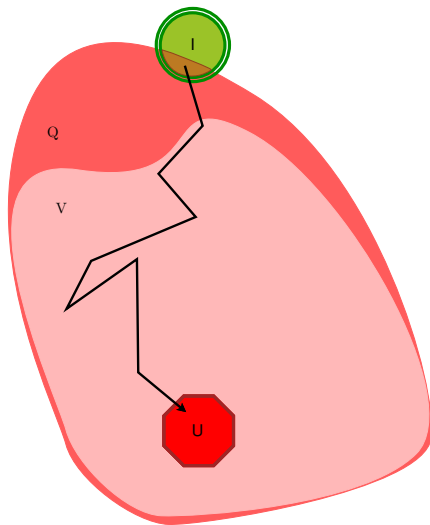
Backward reachability algorithm



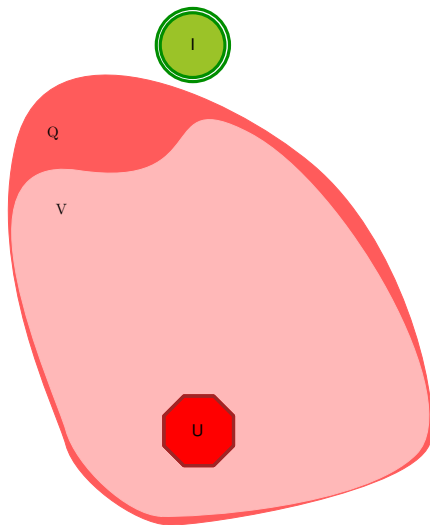
Backward reachability algorithm



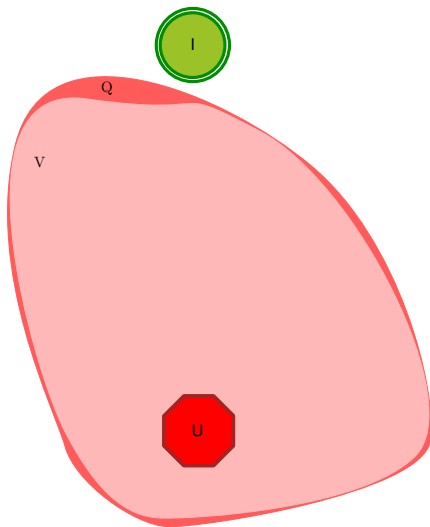
Backward reachability algorithm



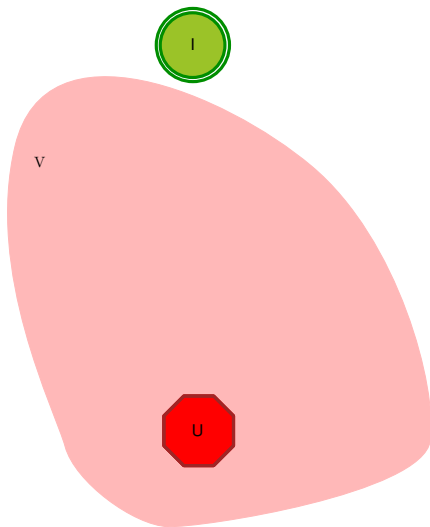
Backward reachability algorithm



Backward reachability algorithm



Backward reachability algorithm



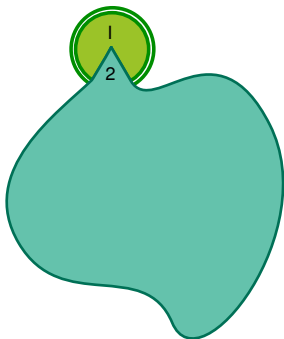
BRAB: intuition



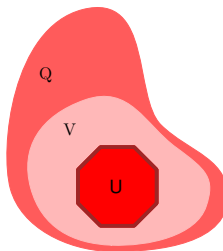
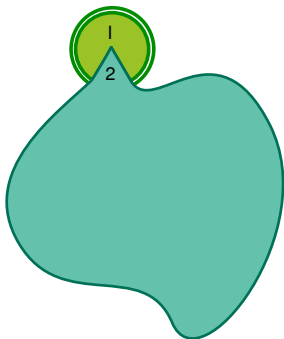
BRAB: intuition



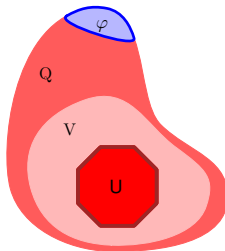
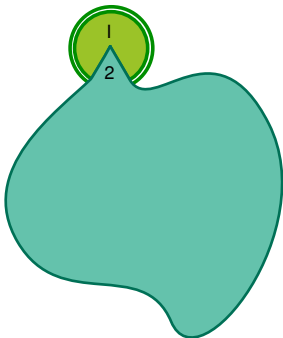
BRAB: intuition



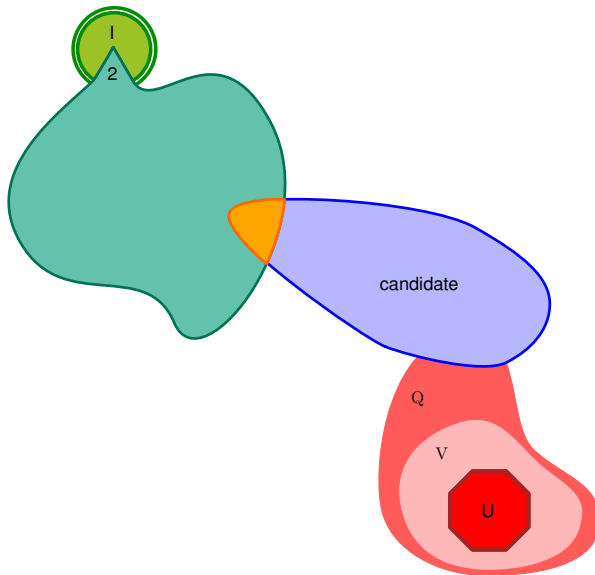
BRAB: intuition



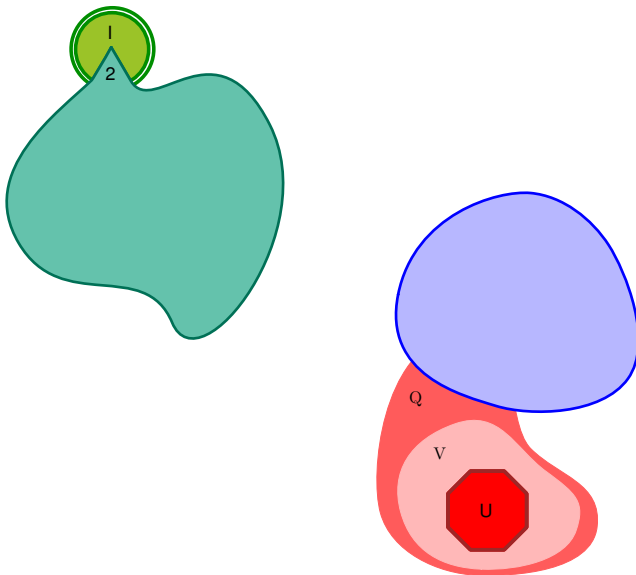
BRAB: intuition



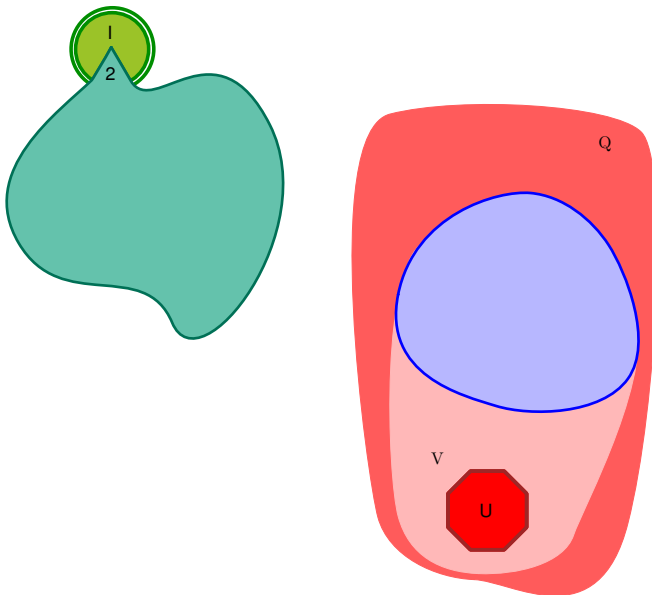
BRAB: intuition



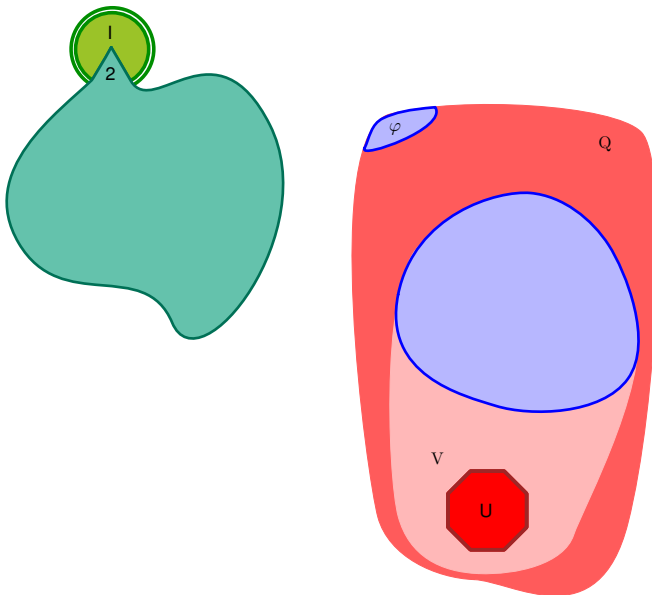
BRAB: intuition



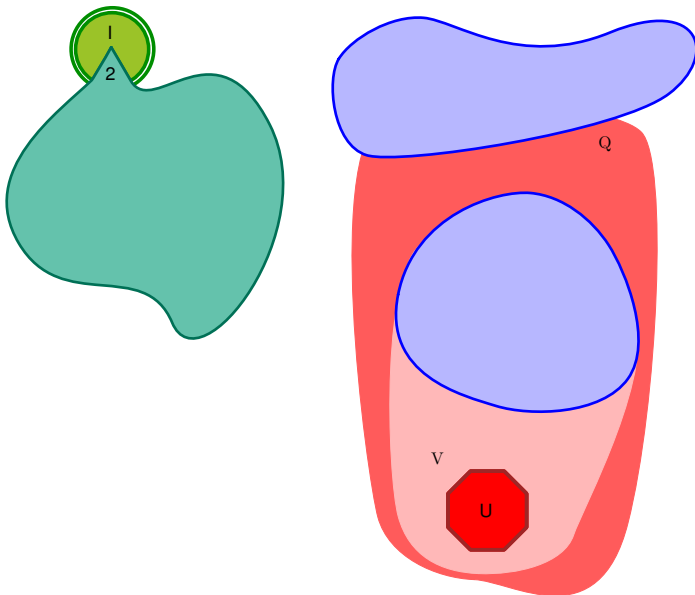
BRAB: intuition



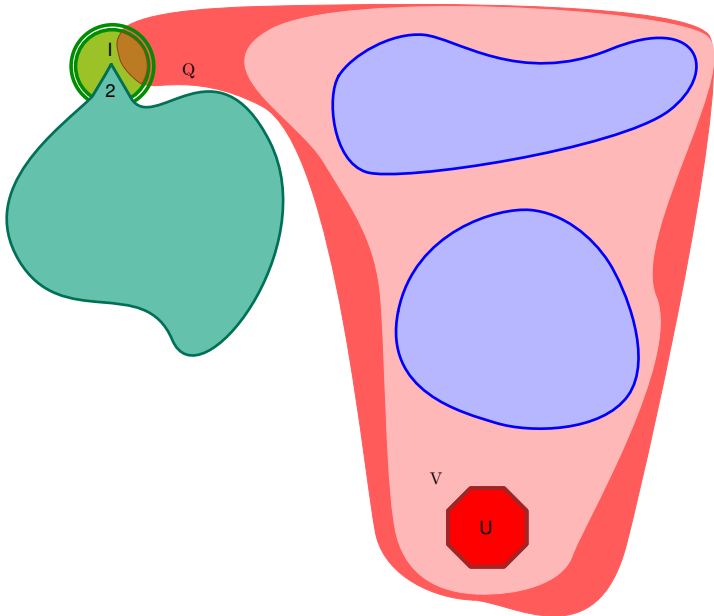
BRAB: intuition



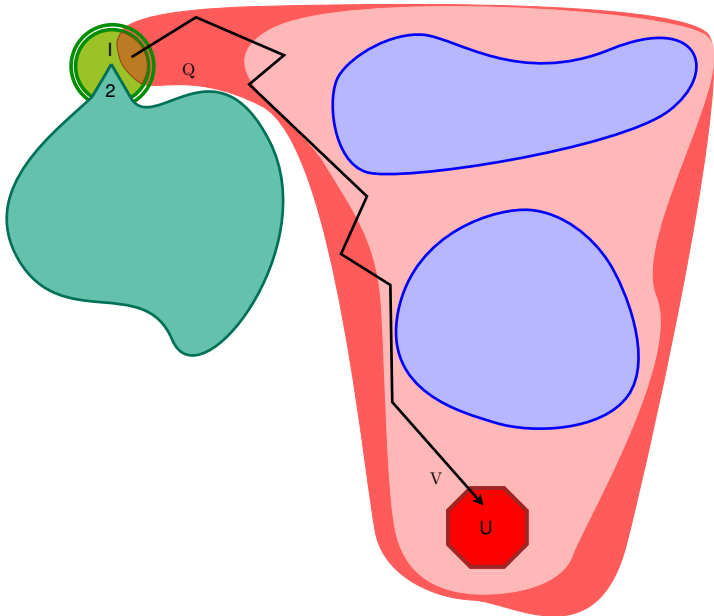
BRAB: intuition



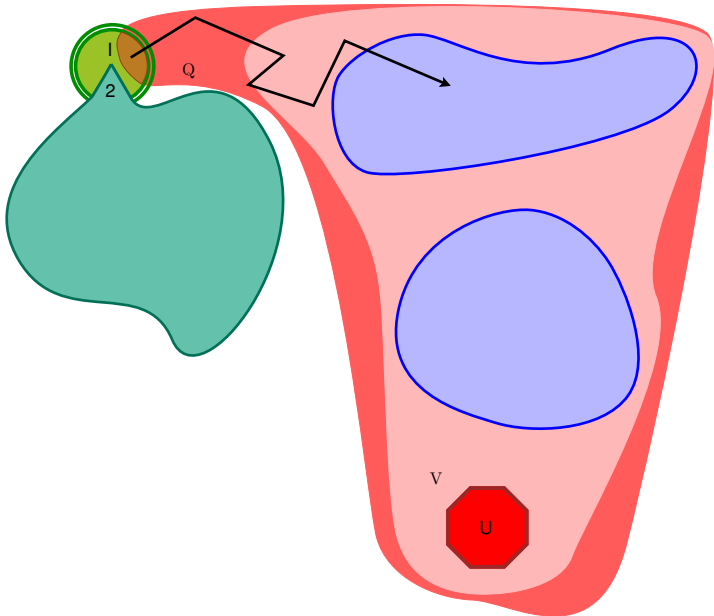
BRAB: intuition



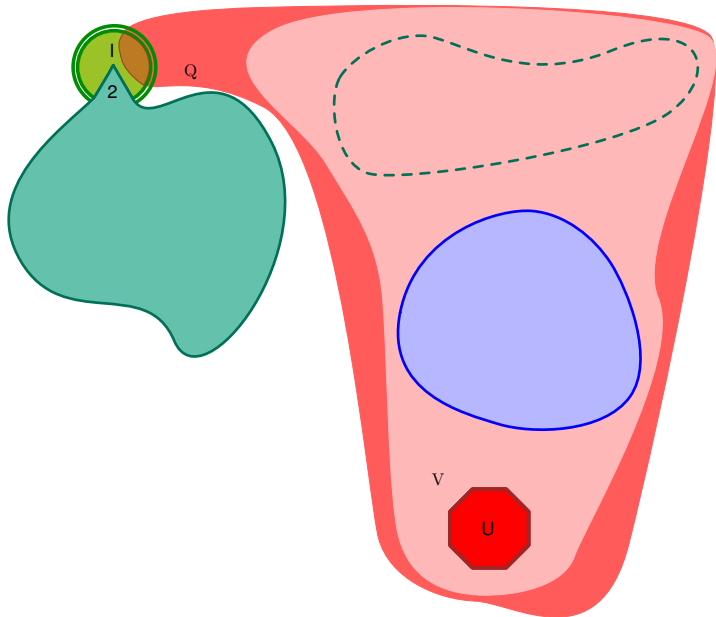
BRAB: intuition



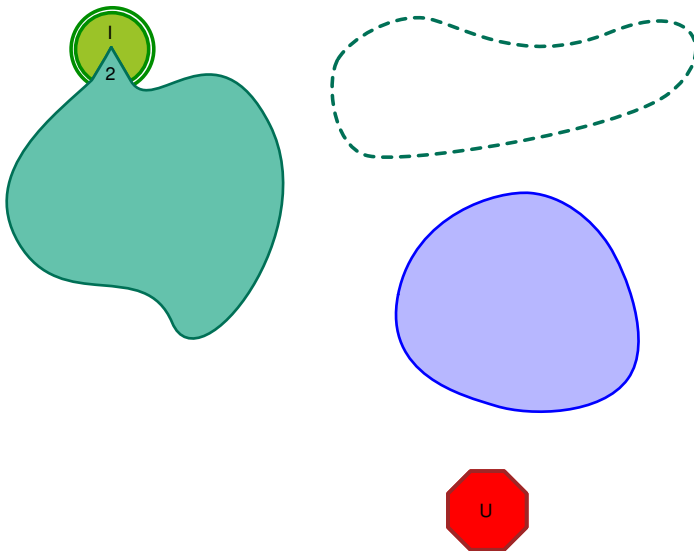
BRAB: intuition



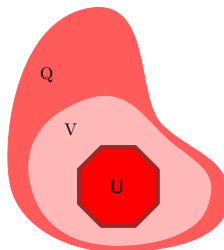
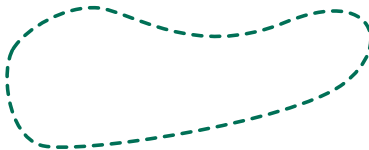
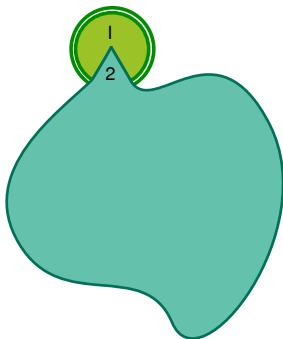
BRAB: intuition



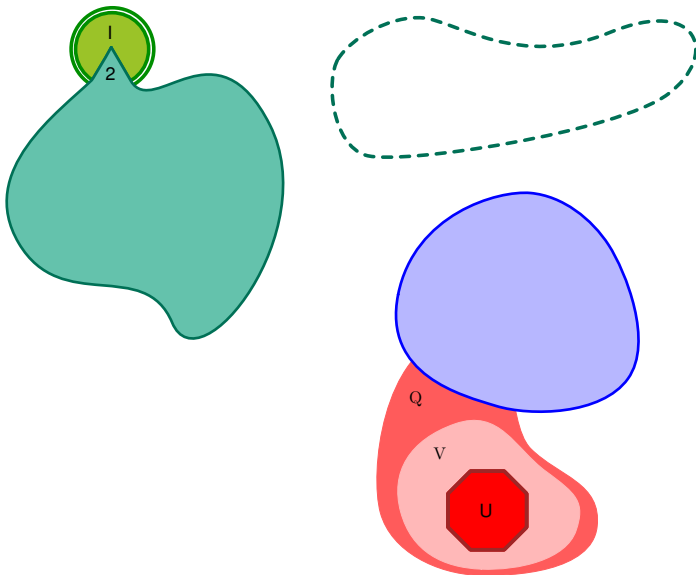
BRAB: intuition



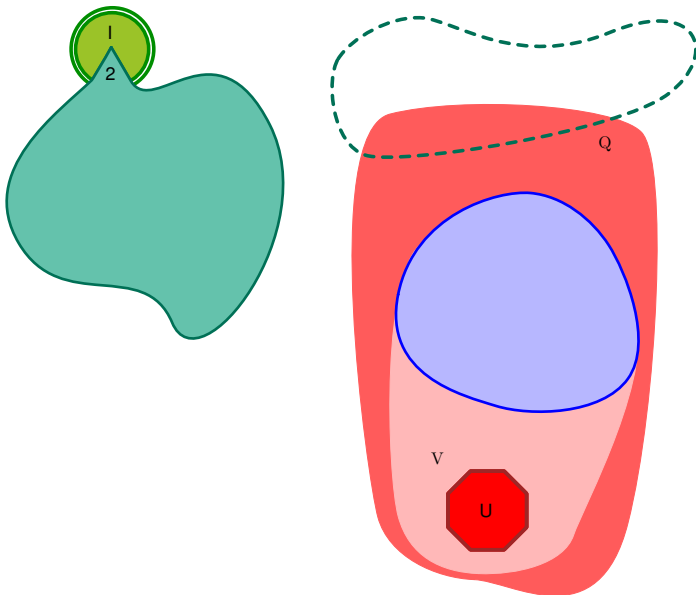
BRAB: intuition



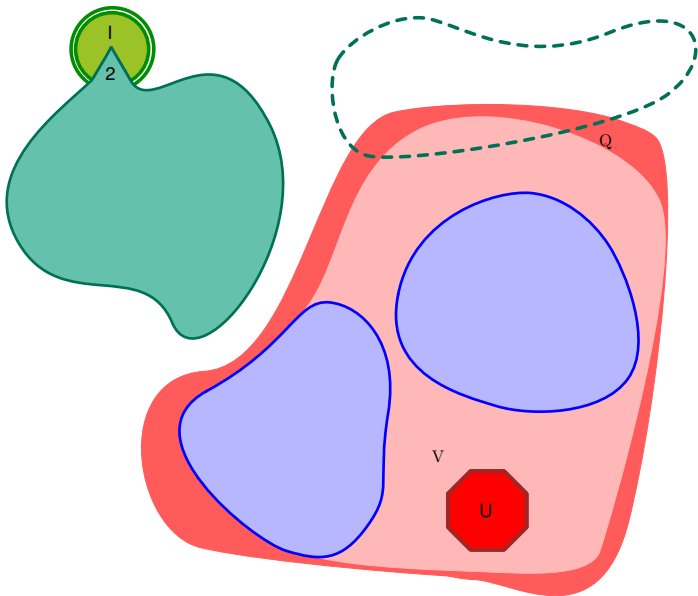
BRAB: intuition



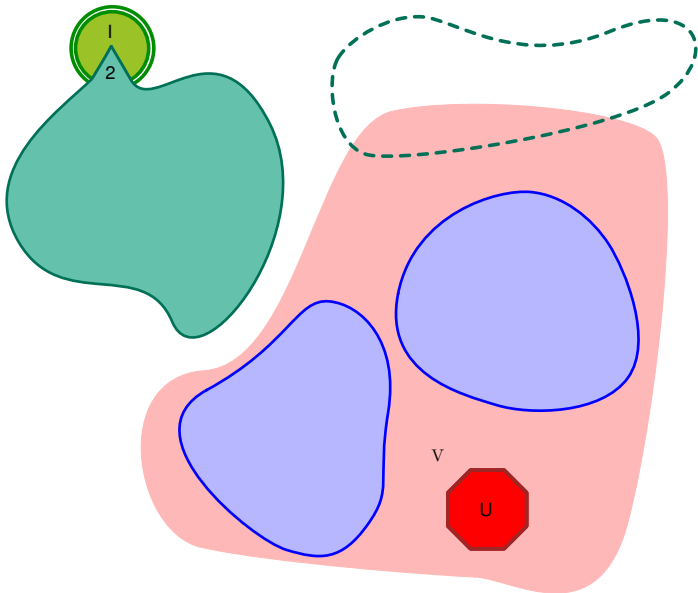
BRAB: intuition



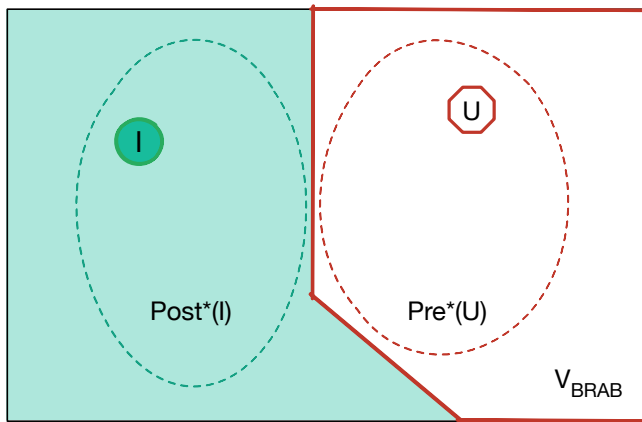
BRAB: intuition



BRAB: intuition



Inductive invariants and BRAB



$$\phi \equiv \neg V_{\text{BRAB}}$$

BRAB algorithm

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

BRAB ():

$B := \emptyset$; $\text{Kind}(U) := \text{Orig}$; $\text{From}(U) := U$;

$\mathcal{M} := \text{FWD}(d_{\max}, k)$;

while BWDA() = unsafe **do**

if $\text{Kind}(F) = \text{Orig}$ **then return** unsafe

$B := B \cup \{ \text{From}(F) \}$;

return safe

BRAB algorithm

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

BWD ():

$V := \emptyset$;

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

BRAB algorithm

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

BWDA ():

$V := \emptyset$;

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, $\text{Approx}_{\mathcal{T}}(\varphi)$);

return safe

BRAB algorithm

I : initial states U : unsafe states (**red**) \mathcal{T} : transitions

$\text{Approx}_{\mathcal{T}}(\varphi)$:

foreach ψ in $\text{candidates}(\varphi)$ **do**

if $\psi \notin B \wedge \mathcal{M} \not\models \psi$ **then**

$\text{Kind}(\psi) := \text{Appr}$;

...

return ψ

...

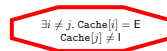
return $\text{pre}_{\mathcal{T}}(\varphi)$

Example: BRAB on German-*ish*

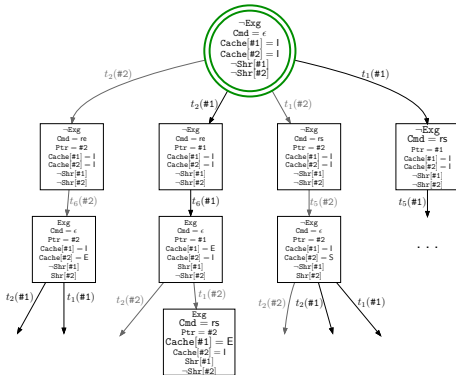
$$\begin{array}{l} \neg \text{Exg} \\ \text{Card} = \epsilon \\ \forall i. \text{Cache}[i] = \text{I} \\ \neg \text{Shr}[i] \end{array}$$

$$\begin{array}{l} \exists i \neq j. \text{Cache}[i] = \text{E} \\ \text{Cache}[j] \neq \text{I} \end{array}$$

Example: BRAB on German-*ish*

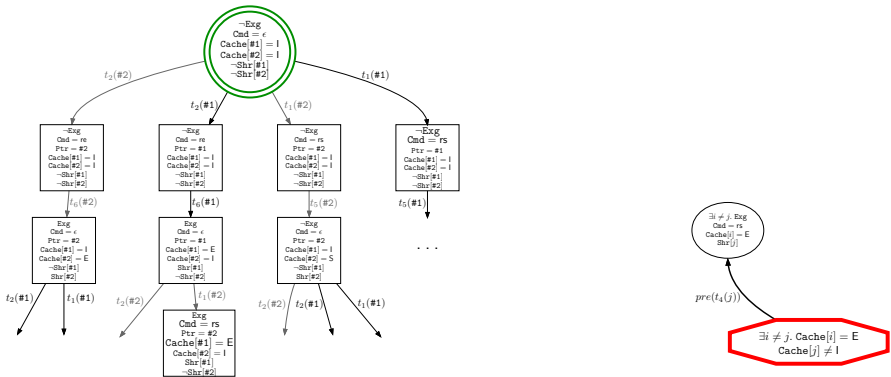


Example: BRAB on German-ish

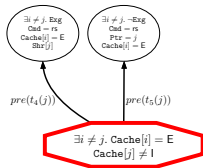
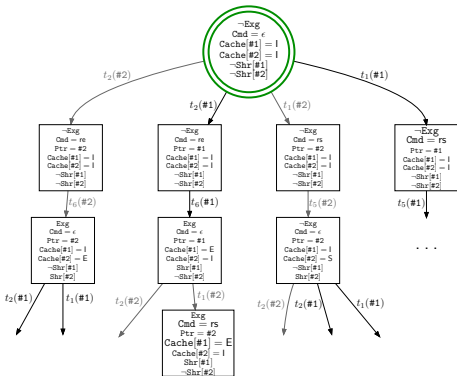


$\exists i \neq j. \text{Cache}[i] = E$
 $\text{Cache}[j] \neq I$

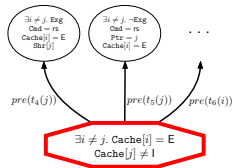
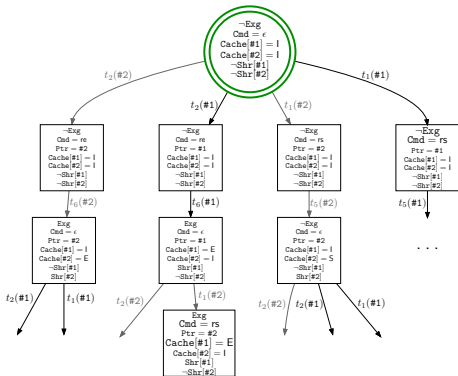
Example: BRAB on German-ish



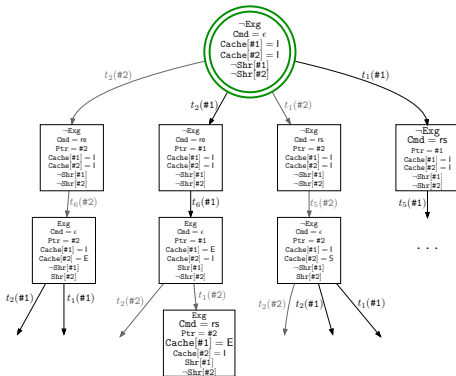
Example: BRAB on German-*ish*



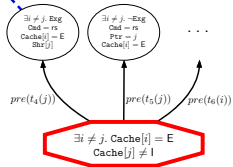
Example: BRAB on German-ish



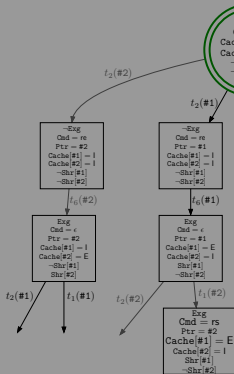
Example: BRAB on German-ish



$\exists i. \text{Cmd} = \text{rs}$
 $\text{Cache}[i] = E$



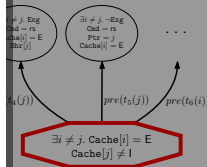
Example: BRAB on German-ish



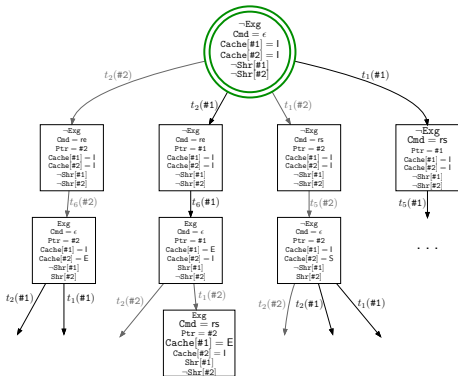
$\exists i. \text{Cmd} = \text{rs}$
 $\text{Cache}[i] = \text{E}$

$\exists i \neq j. \text{Exg}$
 $\text{Cmd} = \text{rs}$
 $\text{Cache}[i] = \text{E}$
 $\text{Shr}[j]$

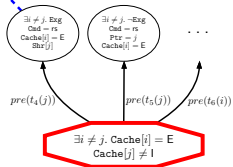
Extracting a candidate ($\text{Approx}_{\mathcal{T}}$)



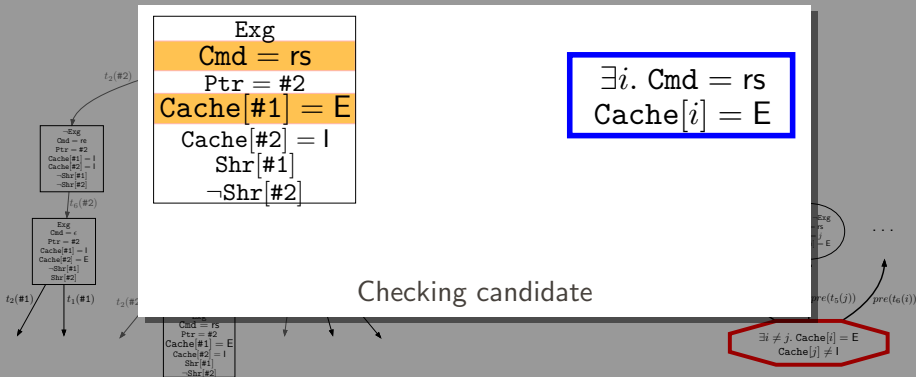
Example: BRAB on German-ish



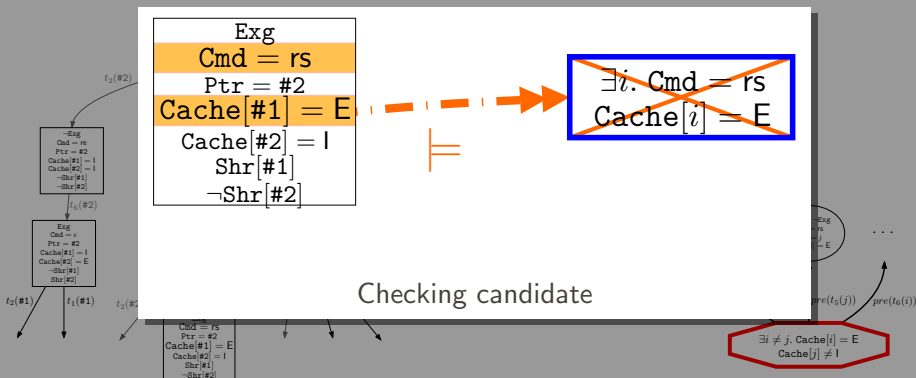
$\exists i. \text{Cmd} = rs$
 $\text{Cache}[i] = E$



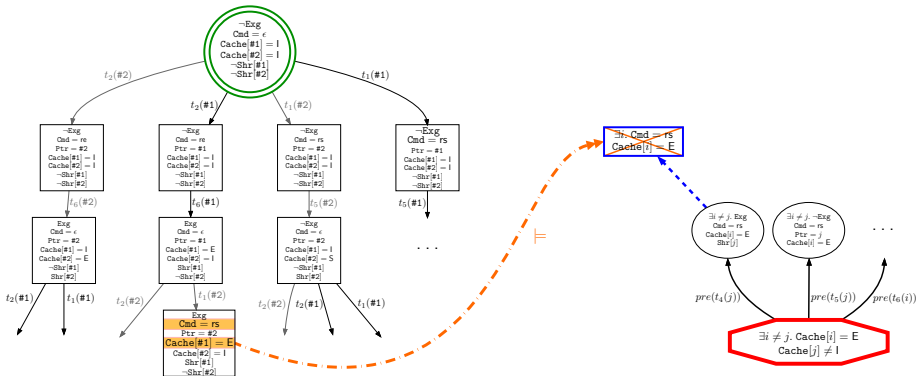
Example: BRAB on German-*ish*



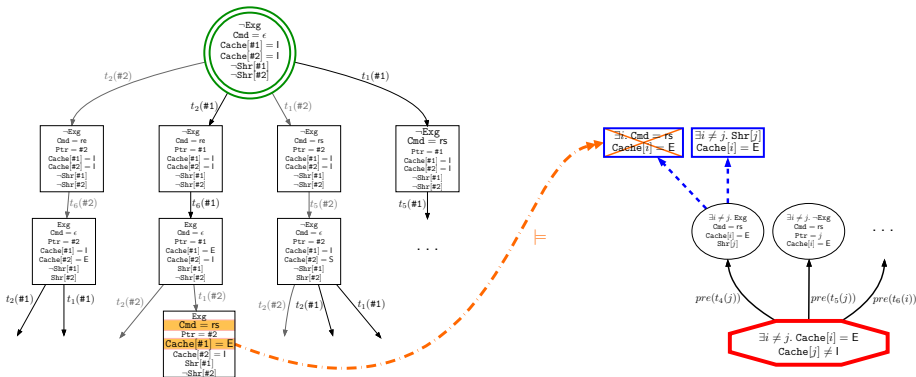
Example: BRAB on German-*ish*



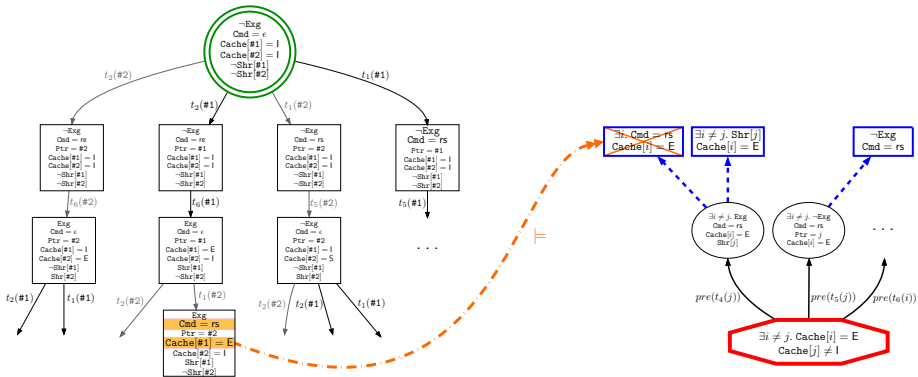
Example: BRAB on German-ish



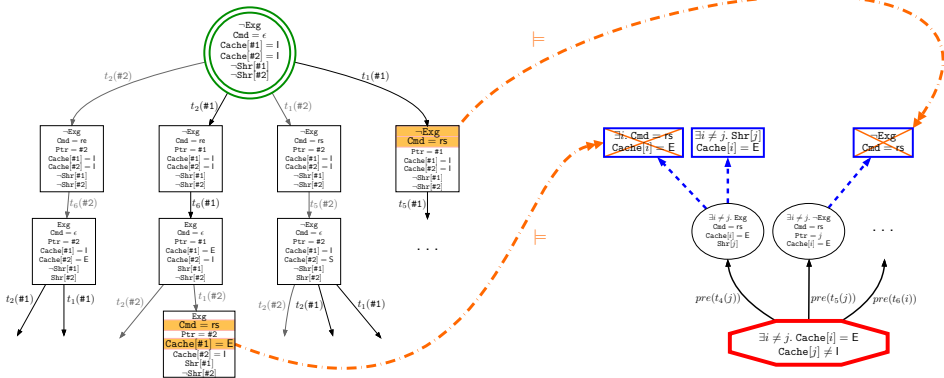
Example: BRAB on German-*ish*



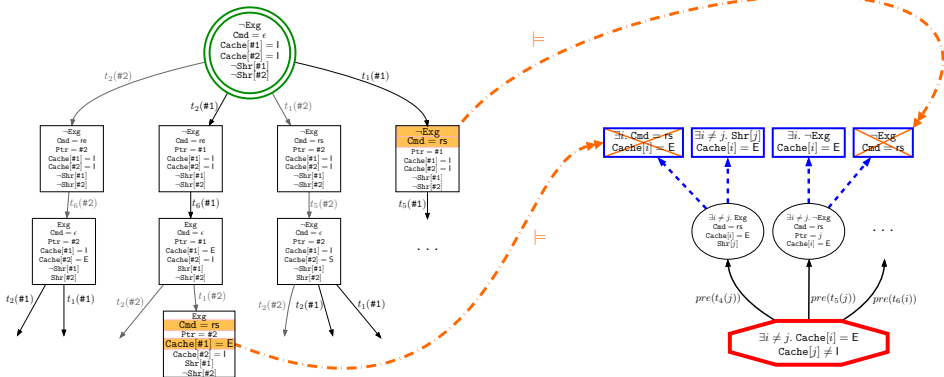
Example: BRAB on German-ish



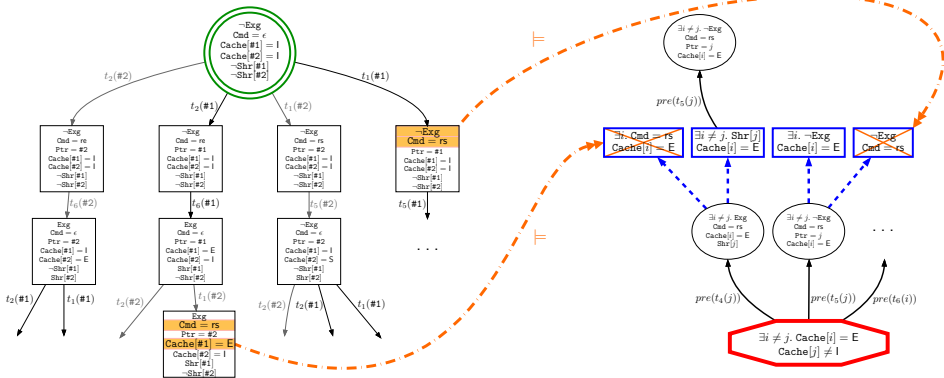
Example: BRAB on German-ish



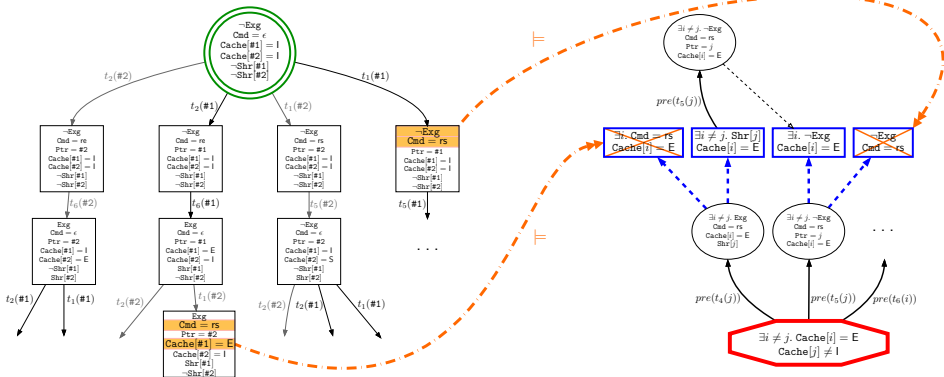
Example: BRAB on German-*ish*



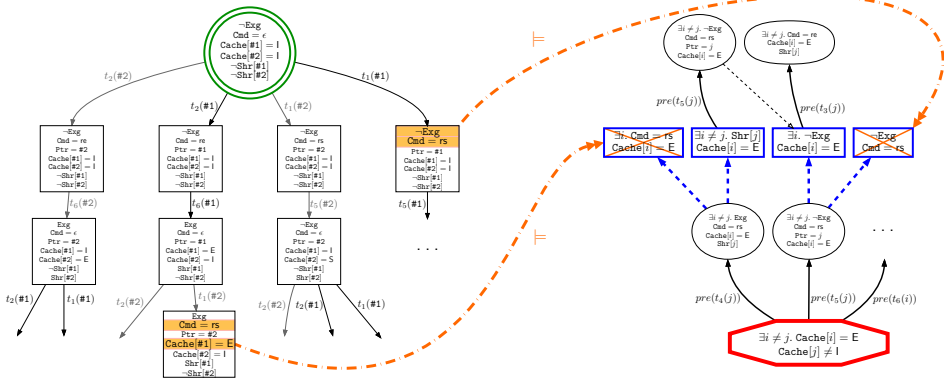
Example: BRAB on German-*ish*



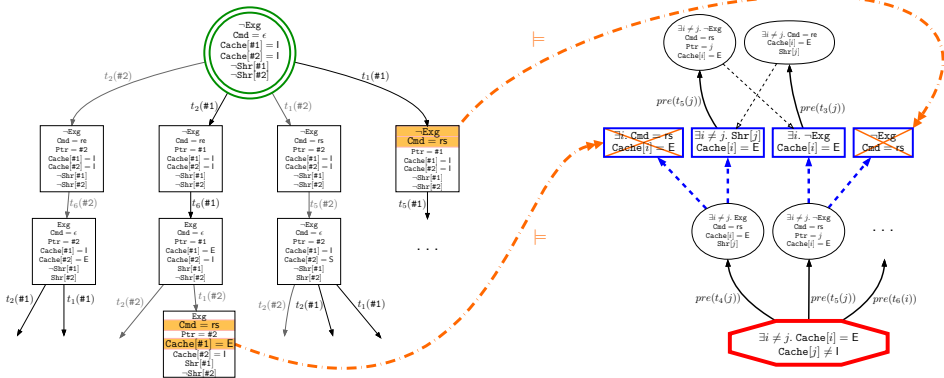
Example: BRAB on German-*ish*



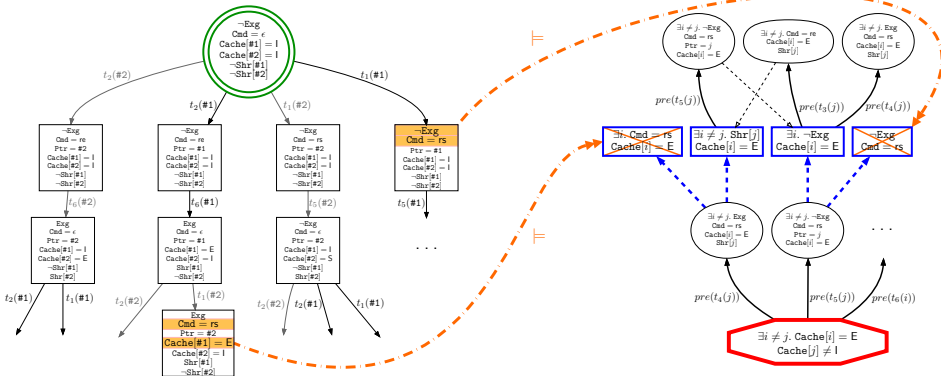
Example: BRAB on German-ish



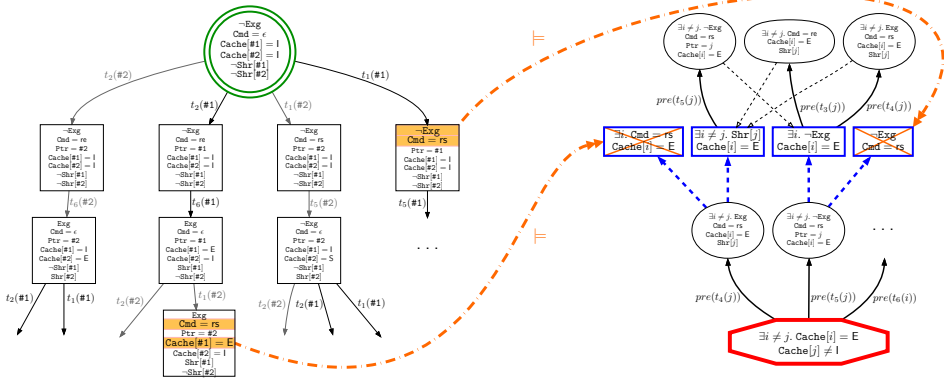
Example: BRAB on German-ish



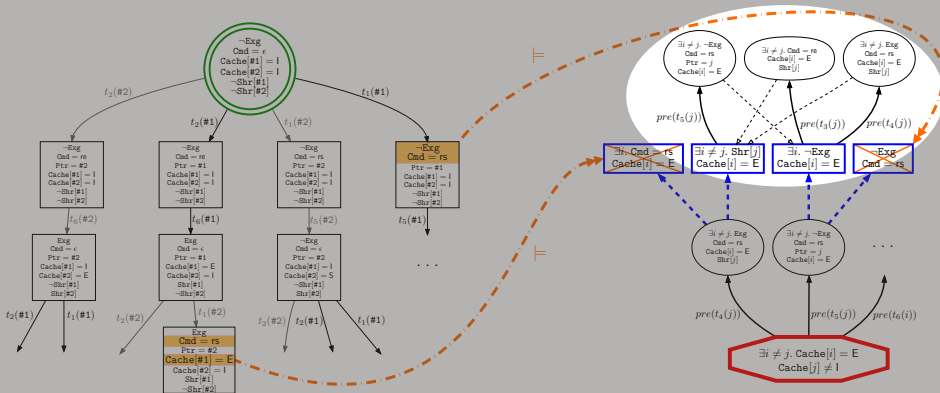
Example: BRAB on German-*ish*



Example: BRAB on German-ish



Example: BRAB on German-*ish*



Some benchmarks

	BRAB	Cubicle	CMurphi		
Szymanski_at	0.14s	0.30s	8.04s (8)	5m12s (10)	2h50m (12)
German_Baukus	0.25s	7.03s	0.74s (4)	19m35s (8)	4h49m (10)
German.CTC	0.29s	3m23s	1.83s (4)	43m46s (8)	12h35m (10)
German_pfs	0.34s	3m58s	0.99s (4)	22m56s (8)	5h30m (10)
Chandra-Toueg	2m17s	2h01m	5.68s (4)	2m58s (5)	1h36m (6)
Szymanski_na	0.19s	T.O.	0.88s (4)	8m25s (6)	7h08m (8)
Flash_nodata	0.36s	O.M.	4.86s (3)	3m33s (4)	2h46m (5)
Flash	5m40s	O.M.	1m27s (3)	2h15m (4)	O.M. (5)

O.M. > 20 GB

T.O. > 20 h

The **Model Checking Modulo Theories** paradigm has been originally proposed by Silvio Ghilardi et Silvio Ranise

► <http://users.mat.unimi.it/users/ghilardi/mcmt/>

In particular, I recommend

Backward Reachability of Array-based Systems by SMT Solving:
Termination and Invariant Synthesis [LMCS, vol.6, n.4, 2010]

More information about Cubicle

In English :

- ▶ Certificates for Parameterized Model Checking [FM 2015]
- ▶ Invariants for Finite Instances and Beyond [FMCAD 2013]
- ▶ Cubicle: A Parallel SMT-based Model Checker for Parameterized Systems [CAV 2012]

In french :

- ▶ Inférence d'Invariant pour le *model checking* de systèmes paramétrés [Alain Mebsout, PhD thesis, 2014]
- ▶ Vérification de programmes C concurrents avec Cubicle : Enfoncer les barrières [JFLA 2014]
- ▶ Vérification de systèmes paramétrés avec Cubicle [JFLA 2013]

Terminaison de l'analyse

La terminaison de l'analyse arrière n'est pas garantie **en général**, mais elle l'est sous certaines conditions

- ▶ étudions l'évolution de l'ensemble V et considérons la séquence

$$V_0 \subseteq V_1 \subseteq V_1 \subseteq \dots \subseteq V_n \subseteq \dots$$

où V_n représente l'ensemble V à la $n^{\text{ième}}$ itération de la boucle `while`

- ▶ pour que la boucle ne termine pas, il faut nécessairement que de nouveaux cubes soient ajoutés régulièrement dans V , *i.e.* qu'il existe une infinité d'ensembles de nœuds visités V_{k_i} tels que $V_{k_1} \subset V_{k_2} \subset \dots$

On va définir les conditions suffisantes pour que cette sous-séquence d'inclusions **strictes** soit finie

Terminaison: configurations

Soit \mathcal{S} un système paramétré à tableaux

Une **configuration** de \mathcal{S} est un état concret du système, *i.e.* un modèle pour

- ▶ les types
- ▶ les variables globales
- ▶ les tableaux du système

En particulier, une configuration doit fixer le nombre de processus du système, *i.e.* la **cardinalité** du type `proc`.

Un système à tableaux \mathcal{S} a un nombre **potentiellement infini** de configurations.

Terminaison: pré-ordres et idéaux

Définition. un **pré-ordre bien fondé** \preceq est une relation binaire réflexive et transitive sans suite infinie strictement décroissante

On note $\llbracket \varphi \rrbracket$ l'ensemble des configurations qui satisfont un cube φ

Si l'ensemble des configurations est muni d'un pré-ordre bien fondé \preceq , alors on peut montrer [Ghilardi&Ranise, LMCS 2010] que $\llbracket \varphi \rrbracket$ est un **idéal**, c'est-à-dire que

$$\text{si } s \in \llbracket \varphi \rrbracket \text{ et } s \preceq s' \text{ alors } s' \in \llbracket \varphi \rrbracket$$

Par extension, il est immédiat de montrer que chaque ensemble V_n de nœuds visités est également un idéal.

Définition. Un pré-ordre bien fondé \preceq est un **bel ordre** si pour toute séquence infinie de configurations s_1, s_2, \dots , il existe nécessairement $i < j$ tels que $s_i \preceq s_j$

Théorème. Si \preceq est un bel ordre, alors toute séquence d'inclusions strictes d'idéaux $\llbracket V_{k_1} \rrbracket \subset \llbracket V_{k_2} \rrbracket \subset \dots$ est finie.

De plus, $\llbracket . \rrbracket$ est **monotone**, i.e. si $V \subset V'$ alors $\llbracket V \rrbracket \subset \llbracket V' \rrbracket$ donc la finitude de la séquence d'inclusion $\llbracket V_{k_1} \rrbracket \subset \llbracket V_{k_2} \rrbracket \subset \dots$ implique bien la finitude de la séquence $V_0 \subseteq V_1 \subseteq \dots \subseteq V_n \subseteq \dots$

Terminaison: preuve du théorème

On raisonne par **contradiction**

- ▶ Si la séquence $\llbracket V_{k_1} \rrbracket \subset \llbracket V_{k_2} \rrbracket \subset \dots$ est infinie, alors il existe également une séquence infinie de configurations s_1, s_2, \dots telle que $s_i \in \llbracket V_{k_i} \rrbracket$ et $s_i \notin \llbracket V_{k_j} \rrbracket$, pour tout $j < i$.
- ▶ De plus, $s_j \not\preceq s_i$, sinon $s_i \in \llbracket V_{k_j} \rrbracket$ puisque chaque $\llbracket V_{k_j} \rrbracket$ est un idéal.
- ▶ L'existence de cette suite infinie contredit donc l'hypothèse que \preceq est un bel ordre

Terminaison: conditions suffisantes

Les conditions suffisantes pour que l'ensemble des configurations soit muni d'un **bel ordre** dépendent essentiellement du choix des opérations de comparaison sur le type proc et sur les types des éléments des tableaux

```
type t = A | B | C  
array V[proc] : t
```

On peut exhiber un bel ordre grâce au lemme de Dickson :

(\mathbb{N}^k, \leq_k) est muni d'un bel ordre pour tout k

Ainsi, la configuration $V[\#1]=A$, $V[\#2]=B$, $V[\#3]=A$, $V[\#4]=C$ est encodée par le triplet d'entiers $(2, 1, 1)$