Falsification and Verification by Searching

CafeOBJ Team of JAIST

Topics

- Search command of CafeOBJ
- Falsification with the search command
- Verification with the search command

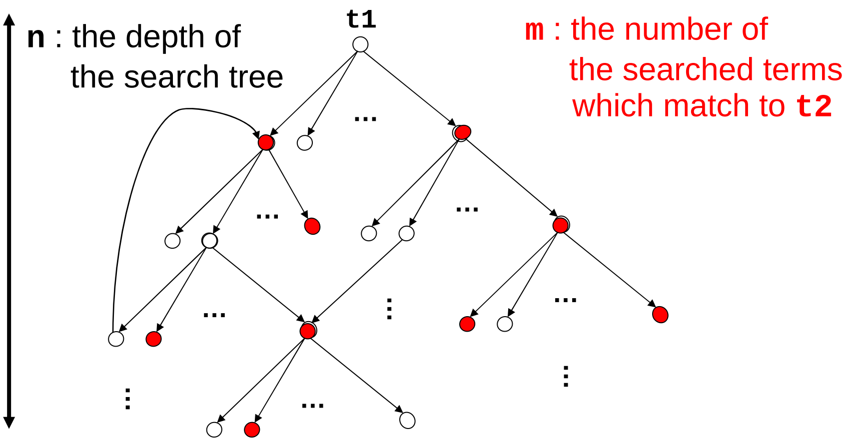
Search command of CafeOBJ

CafeOBJ System has the following built-in predicate:
- **ANY** is any sort (that is, the command is available for any sort)
- **NzNat*** is a built-in sort containing non-zero natural number and the special symbol **"*"** which stands for infinity

pred _=(_,_)=>*_ : Any NzNat* NzNat* Any

(t1 =(m,n)=>* t2) returns **true** if **t1** can be translated (or rewritten), via more than 0 times transitions, to some term which matches to **t2.** Otherwise, it returns **false** . Possible transitions/rewritings are searched in breadth first fashion. **n** is upper bound of the depth of the search, and **m** is upper bound of the number of terms which match to **t2**. If either of the depth of the search or the number of the matched terms reaches to the upper bound, the search stops.

t1 =(m,n)=>* t2



Two other variants of Search command

```
pred _=(_,_)=>+_ : Any NzNat* NzNat* Any
pred _=(_,_)=>!_ : Any NzNat* NzNat* Any
```

(t1 =(m,n)=>+ t2) indicates that the application of transition rules are more than 1 time.

(t1 =(m,n)=>! t2) indicates that the term matching to **t2** should be a term to which no transition rules are applicable.

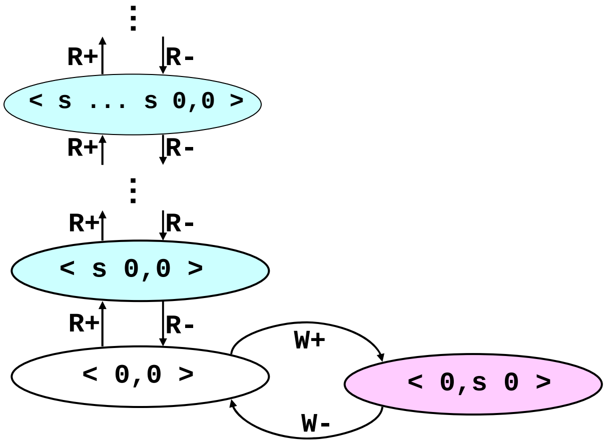
An example for search command: Readers/Writers Policy

searchCommand.mod

READERS-WRITERS

```
-- the following four transitions rules
-- are specifying a Readers/Writers policy
vars R W : Counter .

-- can start to write if no readers and no writers
trans [+w] : <   0,    0 > => <   0, s 0 > .

-- can start to read if no writers
trans [+r] : <   R,    0 > => < s R,    0 > .

-- can stop reading anytime
trans [-r] : < s R,    W > => <   R,    W > .

-- can stop writing anytime
trans [-w] : <   R, s W > => <   R,    W > .
```

State transition diagram for READERS-WRITERS



7

suchThat condition

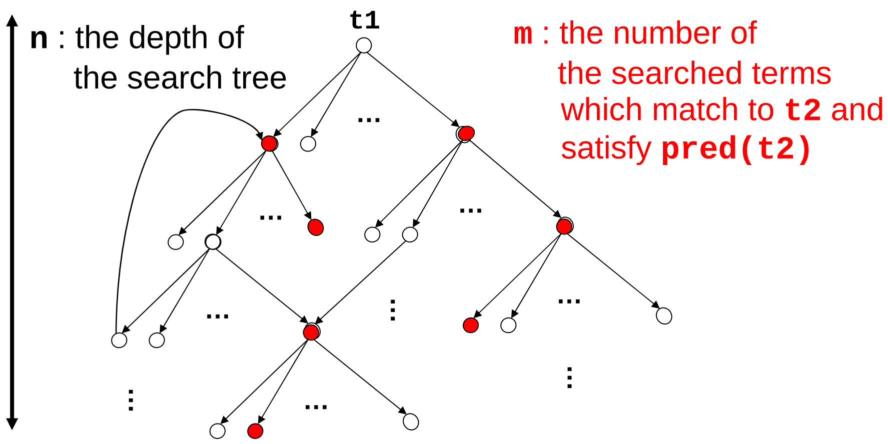
searchCommand.mod

```
t1 =(m,n)=>* t2 suchThat pred1(t2)
```

`pred1(t2)` is a predicate about **`t2`** and can refer to the variables which appear in **`t2`**. **`pred1(t2)`** enhances the condition used to determine the term which matches to **t2**.

8

t1 =(m, n)=>* t2 suchThat pred1(t2)



withStateEq predicate

searchCommand.mod

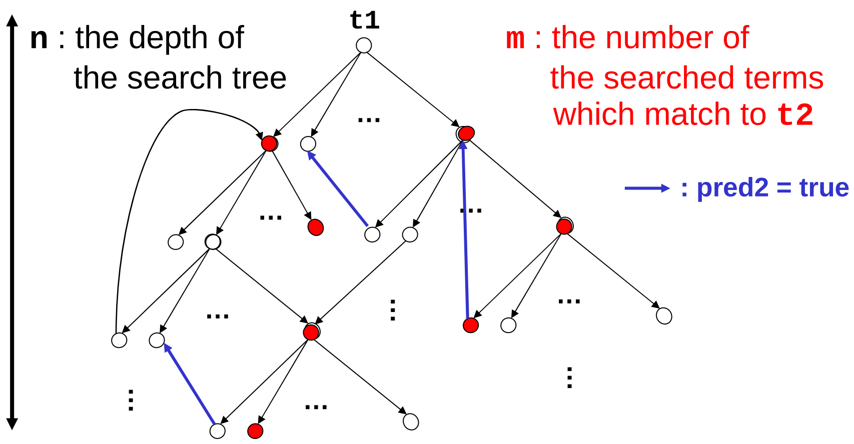
```
t1 =(m,n)=>* t2
   withStateEq pred2(S1:Sort,S2:Sort)
```

pred2(S1:Sort,S2:Sort) is a predicate of two arguments with the same (or greater) sort of **t2**.
pred2(S1:Sort,S2:Sort) is used to determine a newly searched term (a state configuration) is already searched one.
If this **withStateEq** predicate is not given, the term identity binary predicate is used for the purpose.

Using both of suchTant and withStateEq is also possible

```
t1 =(m,n)=>* t2 suchThat pred1(t2)
                withStateEq pred2(S1:Sort,S2:Sort)
```

t1 =(m,n)=>* t2
withStateEq pred2(S1:Sort, S2:Sort)



Abbreviated search commands

searchCommand.mod

```
(CXU ==>1 CYU) =def (CXU =(1,*)=>+ CYU)

(CXU ==>* CYU) =def (CXU =(*,*)=>* CYU)

(CXU ==>! CYU) =def (CXU =(*,*)=>! CYU)

(CXU ==>+ CYU) =def (CXU =(*,*)=>+ CYU)
```

Verifying properties of READERS-WRITERS readerWriter.mod

```
vars R W : Counter
-- mutual exclusion property
pred mutualEx_ : Config
eq mutualEx < 0 , W > = true .
eq mutualEx < R , 0 > = true .
eq mutualEx < s R , s W > = false .
-- only one writer property
pred oneWt_ : Config
eq oneWt < R, 0 > = true .
eq oneWt < R, s 0 > = true .
eq oneWt < R, s s W > = false .
```


Search command (red using (_=(_,_)=>*)) for verifying that mutualEx holds for all reachable states readerWriter.mod

```
red < 0 , 0 > ==>* C:Config
    suchThat (mutualEx(C) == false) .
```

If this returns "false" then the verification is done!

```
red < 0 , 0 > ==>*
        < s R:Counter , s W:Counter >  .
```

If this returns "false" then the verification is done!

Unfortunately both of these reductions do not stop!

Equational abstraction for READERS-WRITERS

readerWriter.mod

The following two equations hold.

```
mutualEx(< s s R:Counter,0 >) = mutualEx(< s 0,0 >)
oneWt(< s s R:Counter,0 >) = oneWt(< s 0,0 >)
```

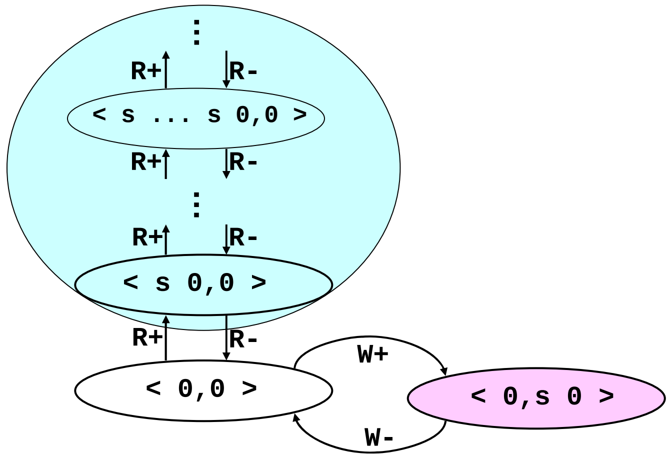
The first equation can give the following search command for the verification of `mutualEx`.

```
eq < s s R:Counter , 0 > = < s 0 , 0 > .
red < 0 , 0 > ==>* C:Config
       suchThat (mutualEx(C) == false) .
```

For `oneWt` property:

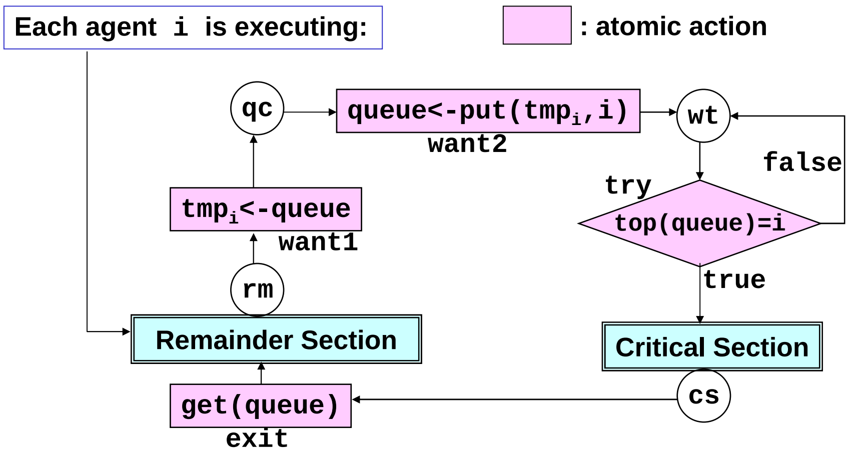
```
eq < s s R:Counter , 0 > = < s 0 , 0 > .
red < 0 , 0 > ==>* C:Config
          suchThat (oneWt(C) == false) .
```


State transition diagram for READERS-WRITERS after the equational abstraction



QLOCK with separable want action

qlockWithSepWant.mod



Signature for QLOCKw2w

```
-- state space of the system
*[Sys]*

-- visible sorts for observation
[Queue Pid Label]

-- observations
bop pc : Sys Pid -> Label
bop tmp : Sys Pid -> Queue
bop queue : Sys -> Queue

-- actions
bop want1 : Sys Pid -> Sys
bop want2 : Sys Pid -> Sys
bop try   : Sys Pid -> Sys
bop exit  : Sys Pid -> Sys
```

Hiden sort declaration

visible sort declaration

Observation declaration

action declaration

Separable want: want1 and want2

```
-- for want1
  op c-want1 : Sys Pid -> Bool {strat: (0 1 2)}
  eq c-want1(S,I) = (pc(S,I) = rm) .
  --
  ceq pc(want1(S,I),J)
      = (if I = J then qc else pc(S,J) fi)        if c-want1(S,I) .
  ceq tmp(want1(S,I),J)
      = (if I = J then queue(S) else tmp(S,J) fi) if c-want1(S,I) .
  ceq queue(want1(S,I)) = queue(S)                if c-want1(S,I) .
  ceq want1(S,I)        = S                       if not c-want1(S,I) .
-- for want2
  op c-want2 : Sys Pid -> Bool {strat: (0 1 2)}
  eq c-want2(S,I) = (pc(S,I) = qc) .
  --
  ceq pc(want2(S,I),J)
      = (if I = J then wt else pc(S,J) fi) if c-want2(S,I) .
  ceq tmp(want2(S,I),J) = tmp(S,J)         if c-want2(S,I) .
  ceq queue(want2(S,I)) = put(I,tmp(S,I))  if c-want2(S,I) .
  ceq want2(S,I)        = S                if not c-want2(S,I) .
```

19

$R_{QLOCKw2w}$: set of reachable states of QLOCKw2w

Signature determining $R_{QLOCKw2w}$

```
-- any initial state
  op init : -> Sys
-- actions
 bop want1 : Sys Pid -> Sys
 bop want2 : Sys Pid -> Sys
 bop try   : Sys Pid -> Sys
 bop exit : Sys Pid -> Sys
```

Recursive definition of $R_{QLOCKw2w}$

$$
\begin{aligned}
R_{QLOCKw2w} = {} & \{\mathtt{init}\} \cup \\
& \{\mathtt{want1}(s,i) \mid s \in R_{QLOCKw2w}, i \in \mathtt{Pid}\} \cup \\
& \{\mathtt{want2}(s,i) \mid s \in R_{QLOCKw2w}, i \in \mathtt{Pid}\} \cup \\
& \{\mathtt{try}(s,i) \mid s \in R_{QLOCKw2w}, i \in \mathtt{Pid}\} \cup \\
& \{\mathtt{exit}(s,i) \mid s \in R_{QLOCKw2w}, i \in \mathtt{Pid}\}
\end{aligned}
$$

20

Making actions into transitions for agents `i,j,x`

```
-- possible transitions in transition rules
 ctrans [want1-i] : < S > => < want1(S,i) > if c-want1(S,i) .
 ctrans [want1-j] : < S > => < want1(S,j) > if c-want1(S,j) .
 ctrans [want1-x] : < S > => < want1(S,x) > if c-want1(S,x) .

 ctrans [want2-i] : < S > => < want2(S,i) > if c-want2(S,i) .
 ctrans [want2-j] : < S > => < want2(S,j) > if c-want2(S,j) .
 ctrans [want2-x] : < S > => < want2(S,x) > if c-want2(S,x) .

 ctrans [try-i] : < S > => < try(S,i) > if c-try(S,i) .
 ctrans [try-j] : < S > => < try(S,j) > if c-try(S,j) .
 ctrans [try-x] : < S > => < try(S,x) > if c-try(S,x) .

 ctrans [exit-i] : < S > => < exit(S,i) > if c-exit(S,i) .
 ctrans [exit-j] : < S > => < exit(S,j) > if c-exit(S,j) .
 ctrans [exit-x] : < S > => < exit(S,x) > if c-exit(S,x) .
```


Falsification can be done by the search command

```
eq mutualEx(S:Sys,I:Pid,J:Pid) =
   ((pc(S,I) = cs and pc(S,J) = cs) implies I = J) .
eq mutualEx-ij(S:Sys) = mutualEx(S,i,j) .

red < init > =(1,6)=>* < S:Sys >
               suchThat (not mutualEx-ij(S)) .
```

If

```
  red < init > =(1,6)=>* < S:Sys >
                  suchThat (not mutualEx-ij(S)) .
```

returns `true` for some term, the term represent a state s in $R_{QLOCKw2w}$ for which `mutualEx(s,i,j)` does not hold.

A counter example found

qlockWithSepWantFalsify.mod

show path 382

```
[state 0] (< init >):Config
  ctrans [want1-i]
[state 1] (< want1(init, i) >):Config
  ctrans [want1-j]
[state 4] (< want1(want1(init, i), j) >):Config
  ctrans [want2-i]
[state 14] (< want2(want1(want1(init, i), j), i) >):Config
  ctrans [try-i]
[state 45] (< try(want2(want1(want1(init, i), j), i), i) >):Config
  ctrans [want2-j]
[state 136] (< want2(try(want2(want1(want1(init, i), j), i), i), j) >):Config
  ctrans [try-j]
[state 382] (< try(want2(try(want2(want1(want1(init, i), j), i), i), j), j) >):Config
```

try(want2(try(want2(want1(want1(init,i),j),i),i),j),j)

23