Combining Inference and Search in CafeOBJ Verifications with Proof Scores

FUTATSUGI,Kokichi
二木 厚吉

JAIST
(Japan Advanced Institute of Science and Technology)

Introduction

- Describe an attempt of combining inference and search in the proof score method with CafeOBJ by using QLOCK example.
- This can be seen as an example of combining behavioral spec and rewriting spec in verification.
- The methodology described seems to have a potential of becoming a powerful verification technique.

Proof Score Approach

- Domain/requirement/design engineers are expected to construct proof scores together with formal specifications
- Proof scores are instructions such that when executed (or "played") and everything evaluates as expected, then the desired property is convinced to hold
  - Proof by construction/development
  - Proof by reduction/computation/rewriting

Development of proof scores in CafeOBJ

- Many simple proof scores are written in OBJ language from 1980’s; some of them are not trivial
- From around 1997 CafeOBJ group at JAIST use proof scores seriously for verifying specifications for various examples
  - From static to dynamic/reactive system
  - From ad hoc to more systematic proof scores
  - Introduction of OTS (Observational Transition System) was a most important step

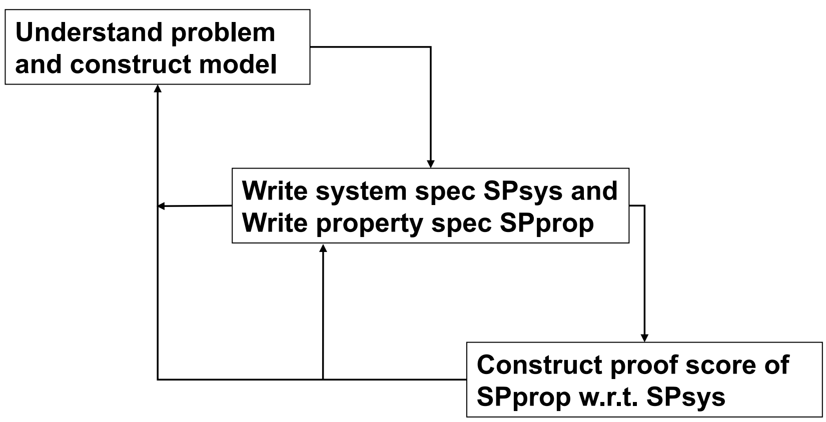
Modeling, Specifying, and Verifying (MSV) in CafeOBJ with Proof Scores

1. By understanding a problem to be modeled/ specified/verified, determine several sorts of objects (entities, data, agents, or states) and operations (functions, actions, or events) over them for describing the problem

2. Define the meanings/functions of the operations by declaring equations over expressions/terms composed of the operations

3. Write proof scores for properties to be verified

MSV with proof scores in CafeOBJ

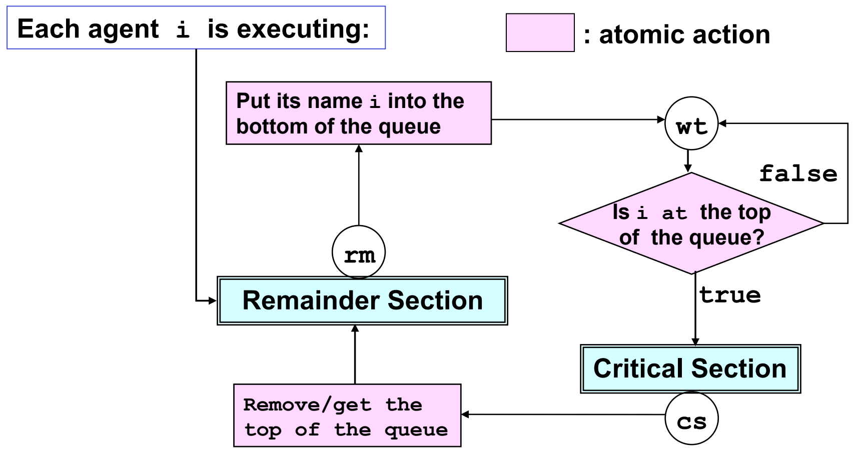


An example: mutual exclusion protocol

Assume that unboundedly many agents (or processes) are competing for a common equipment, but at any moment of time only one agent can use the equipment. That is, the agents are mutually excluded in using the equipment. A protocol (concurrent mechanism or algorithm) which can achieve the mutual exclusion is called "mutual exclusion protocol".

Modeling and Specification of QLOCK

QLOCK (locking with queue): a mutual exclusion protocol

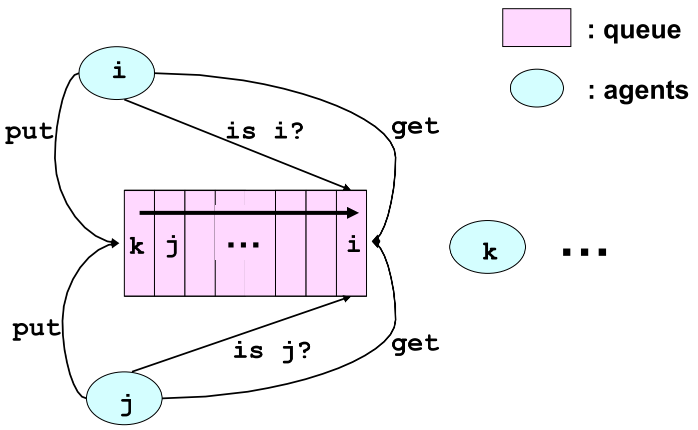


QLOCK: basic assumptions/characteristics

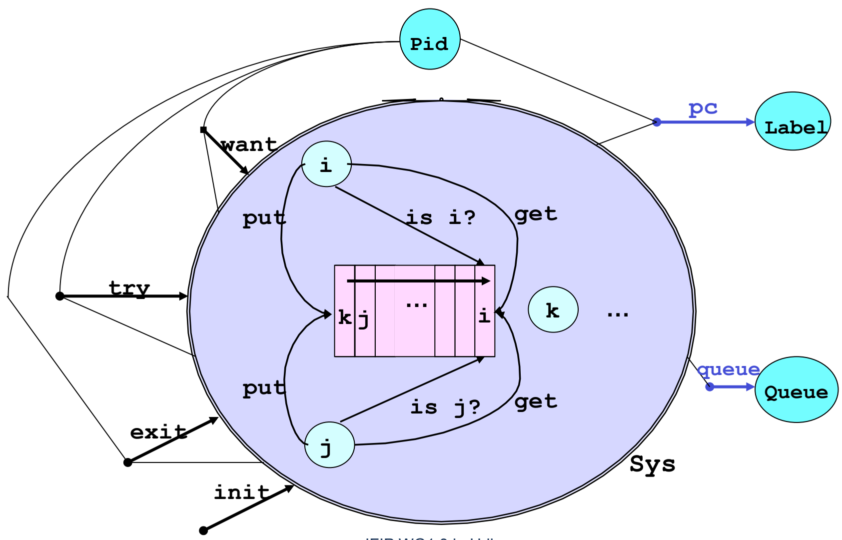
- There is only one queue and all agents share the queue.
- Any basic action on the queue is inseparable (or atomic). That is, when any action is executed on the queue, no other action can be executed until the current action is finished.
- There may be unbounded number of agents.
- In the initial state, every agents are in the remainder section (or at the label `rm`), and the queue is empty.

The property to be shown is that at most one agent is in the critical section (or at the label `cs`) at any moment.

Global (or macro) view of QLOCK



Modeling QLOCK (via Signature Diagram) with OTS (Observational Transition System)



Signature for QLOCKwithOTS

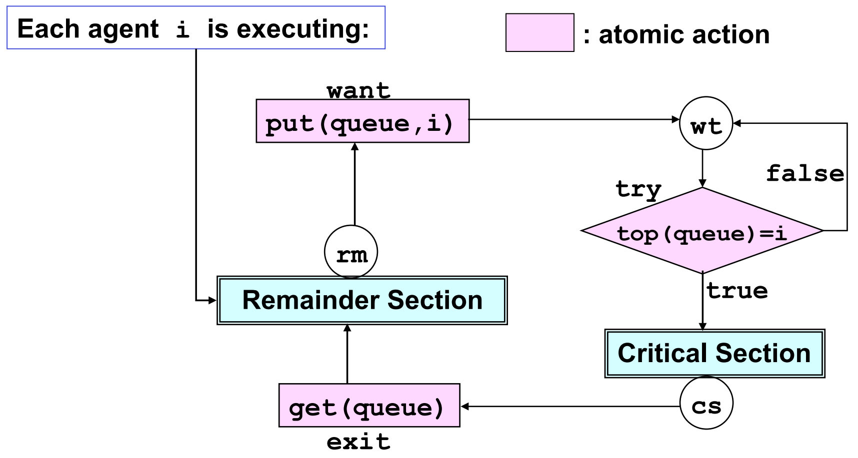
- `Sys` is the sort for representing the state space of the system.
- `Pid` is the sort for the set of agent/process names.
- `Label` is the sort for the set of labels; i.e. `{rm, wt, cs}`.
- `Queue` is the sort for the queues of `Pid`
- `pc` (program counter) is an observer returning a label where each agent resides.
- `queue` is an observer returning the current value of the waiting queue of `Pid`.
- `want` is an action for agent `i` of putting its name/id into the queue.
- `try` is an action for agent `i` of checking whether its name/id is at the top of the queue.
- `exit` is an action for agent `i` of removing/getting (its name/id from) the top of the queue.

CafeOBJ signature for QLOCKwithOTS

```
-- state space of the system
*[Sys]*                                System sort declaration

-- visible sorts for observation
[Queue Pid Label]                      visible sort declaration

-- observations
bop pc : Sys Pid -> Label              Observation declaration
bop queue : Sys -> Queue

-- any initial state
init : -> Sys (constr)
-- actions
bop want : Sys Pid -> Sys (constr)
bop try  : Sys Pid -> Sys (constr)     action declaration
bop exit : Sys Pid -> Sys (constr)
```

QLOCK using operators in the CafeOBJ module QUEUE

qlock.mod



(_ =*= _) is congruent for OTS -- an important property of OTS

The binary relation `(S1:Sys =*= S2:Sys)` is defined to be true iff `S1` and `S2` have the same observation values.

OTS style of defining the possible changes of the values of obervations is characterized by the equations of the form:
$o(a(s,d),d')$
$= \ldots o_1(s,d_1) \ldots o_2(s,d_2) \ldots o_n(s,d_n) \ldots$
for appropriate data values of $d, d', d_1, d_2, \ldots, d_n$.

It can be shown that OTS style guarantees that (_ =*= _) is congruent with respect to all actions.

Verification by Inference

R_{QLOCK} (set of reachable states) of OTS_{QLOCK} (OTS defined by the module QLOCK)

Signature determining R_{QLOCK}

```
-- any initial state
  op init : -> Sys {constr}
-- actions
  bop want : Sys Pid -> Sys {constr}
  bop try  : Sys Pid -> Sys {constr}
  bop exit : Sys Pid -> Sys {constr}
```

Recursive definition of R_{QLOCK}

$$
\begin{aligned}
R_{QLOCK} = \; & \{\mathtt{init}\} \;\cup \\
& \{\mathtt{want(s,I)} \mid \mathtt{s} \in R_{QLOCK}, \mathtt{I:Pid}\} \;\cup \\
& \{\mathtt{try(s,I)} \mid \mathtt{s} \in R_{QLOCK}, \mathtt{I:Pid}\} \;\cup \\
& \{\mathtt{exit(s,I)} \mid \mathtt{s} \in R_{QLOCK}, \mathtt{I:Pid}\}
\end{aligned}
$$

Mutual exclusion property as an invariant

```
mod*  INV1 {
   pr(QLOCK)
-- declare a predicate to verify to be an invariant
    pred inv1 : Sys Pid Pid
-- CafeOBJ variables
   var S :  Sys .
   vars I J : Pid .
-- define inv1 to be the mutual exclusion property
   eq inv1(S,I,J)
       = (((pc(S,I) = cs) and (pc(S,J) = cs)) implies I = J) .
}
```

Formulation of the proof goal for mutual exclusion property

INV1 |- $(\forall i,j$:Pid$)$inv1(s,i,j)
for all s:R_{QLOCK}

Splitting Proof Goal by Inductive Structure of R_{QLOCK}

INV1 |- ∀I,J:Pid.inv1(s,I,J)
for all s in R_{QLOCK}

⇩

INV1 |- ∀I,J:Pid.inv1(init,I,J)

INV1 ∪ {∀I,J:Pid.inv1(s,I,J)}|-
∀I,J:Pid.inv1(want(s,k),I,J)

INV1 ∪ {∀I,J:Pid.inv1(s,I,J)}|-
∀I,J:Pid.inv1(try(s,k),I,J)

INV1 ∪ {∀I,J:Pid.inv1(s,I,J)}|-
∀I,J:Pid.inv1(try(s,k),I,J)

Correspondence between Assertion and Proof Passage

Proof Passage

```
INV1 ∪
{∀I,J:Pid.inv1(s,I,J)}
|-
∀I,J:Pid.
   inv1(want(s,k),I,J)
```

~

```
open INV1
-- arbitrary objects
   op s : -> Sys .
   ops i j k : -> Pid .
-- assumptions
   eq inv1(s,I:Pid,J:Pid) = true .
-- |-
-- check if the predicate is true.
   red inv1(try(s,k),i,j) .
close
```

Logical Statement
of stating that Specification satisfies property

Logical Statement and CafeOBJ Code
If reduction part of the CafeOBJ code returns `true` then the assertion holds

Induction Scheme in Proof Passages

proof-01.mod

Induction Scheme

```
{[1-init],[1-want]*,[1-try]*,[1-exit]*}
           implies  [mx]*
```

Assertion Splitting via Case Splitting

proof-02.mod

Because

```
INV1 |= c-want(s,k) or ~c-want(s,k)
```

Holds, the following assertion splitting is justified.

Assertion Splitting via Case Splitting

```
{[1-want,c-w]*, [1-want,~c-w]}
      implies [1-want]*
```

{ (E |- (p_1 or p_2)), (E U {p_1=true} |- p) , (E U {p_2=true} |- p) }
implies E |- p

Some properties of E |- p

Meta **Level** Equation and Object Level Equation

E U { t_1 = t_2 } |- p iff E U { (t1 = t2) = true } |- p

E |- ((t_1 = t_2) implies p) iff
E U { t_1 = t_2 } |- p

E |= (t_1 = t_2) implies
(E U { t_1 = t_2 } |- p iff E |= p)

Proof Calculus (Entailment System) (1)

Entailment System of HCL

AES	(*Reflexivity*) $\emptyset \vdash t = t$ (*Symmetry*) $t = t' \vdash t' = t$ (*Transitivity*) $\{t = t', t' = t''\} \vdash t = t''$ (*Congruence*) $\{t_i = t'_i \mid 1 \leq i \leq n\} \vdash \sigma(t_1, ..., t_n) = \sigma(t'_1, ..., t'_n)$ (*PCongruence*) $\{t_i = t'_i \mid 1 \leq i \leq n\} \cup \{\pi(t_1, ..., t_n)\} \vdash \pi(t'_1, ..., t'_n)$
IES	(*Implications*) $\Gamma \vdash \bigwedge H \Rightarrow C$ iff $\Gamma \cup H \vdash C$
GUES	(*Substitutivity*) $(\forall x)\rho \vdash (\forall Y)\rho(x \leftarrow t)$ (*Generalization*) $\Gamma \vdash_\Sigma (\forall Z)\rho$ iff $\Gamma \vdash_{\Sigma(Z)} \rho$

Proposition (instance of an institution-independent result)

The entailment system of **HCL** *is sound, complete and compact.*

Proof Calculus (Entailment System) (2)

Reachable Universal Entailment Systems (RUES)

The entailment system of **CHCL**:

1. the rules for **HCL**
2. (*Case splitting*)

$$\frac{\{\Gamma \vdash (\forall Y)\rho(x \leftarrow t) \mid t \text{ built from constructors and loose variables}\}}{\Gamma \vdash (\forall x)\rho}$$

Definition

- A (S, F, P)-model M is S^c-reachable ($S^c \subseteq S$) if it is a quotient of $T_F(Y)$ for some Y of sorts $S - S^c$.
- $((S, F, F^c, P), \Gamma)$ is **sufficient-complete** iff every S^c-reachable (S, F, P)-model M satisfying Γ is a (S, F, F^c, P)-model (in **CHCL**).

CafeOBJ codes (system spec, property spec, and proof score) for verification of the mutual exclusion property

qlock.mod
inv.mod
proofScore.mod
proofByPS.mod

- These codes make a general verification of mutual exclusion property independent of the number of agents/processes.
- The proof score examine all possible cases and do symbolic test for each of them. Constructing proof scores sometimes become tedious and time consuming.
- Some room for improvement!

Verification by Inference and Search

Transition system for QLOCK

```
-- pre-transiton system with an agent/process p
mod* QLOCKpTrans { pr(QLOCKconfig)
  op p : -> PidConst .  var S : Sys .
  -- possible transitions
  ctrans < S > => < want(S,p) > if c-want(S,p) .
  ctrans < S > => < try(S,p) >     if c-try(S,p) .
  ctrans < S > => < exit(S,p) >   if c-exit(S,p) .
}
-- transition system which simulates QLOCK of 2 agents i j
mod* QLOCKijTrans {
  -- 2 QLOCKpTrans-es  corresponding to two different
  -- PidConst-s i j are declared
  -- by using renaming of modules
 using((QLOCKpTrans * {op p -> i}) +
          (QLOCKpTrans * {op p -> j}))  }
```

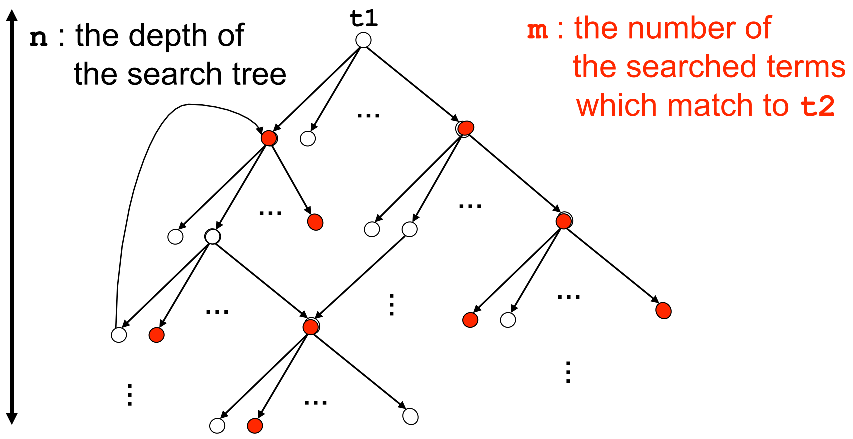

Search command of CafeOBJ a la Maude's search command

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`



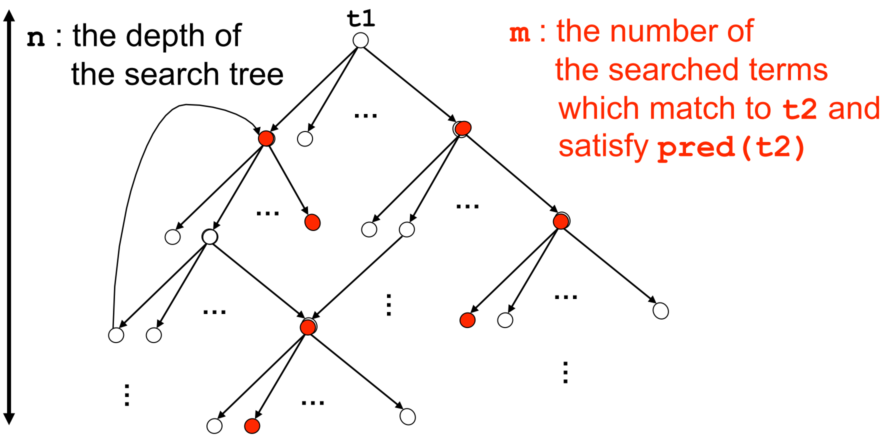
`suchThat` condition

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`**.

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



withStateEq predicate

searchCommand.mod

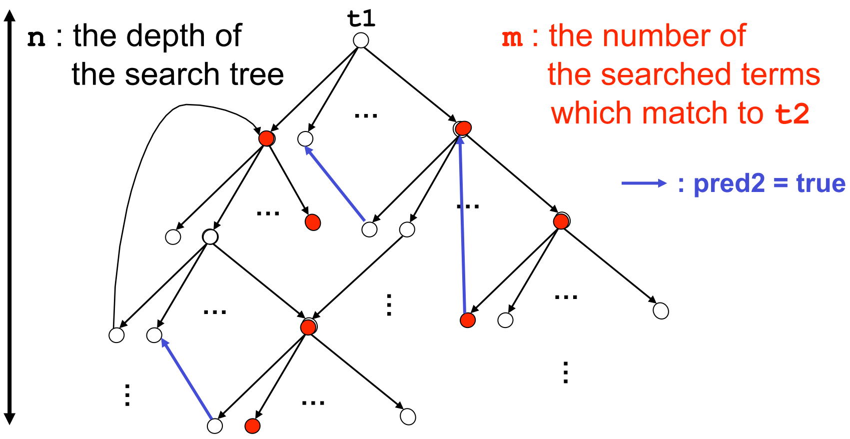
```
t1 =(m,n)=>* t2
   withStateEq pred2(S1:Sort,S2:Sort)
```

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)



CafeOBJ Codes for verification by searching with Observational Equivalence

qlockTrans.mod
mexStarve.mod
qlockObEq.mod
proofBySearchWithObEq.mod

This verification is effective only for small finite number (2, 3, or 4) of agents!

Simulation of any number of agents by two agents

If all the behaviors of the system with any number of agents with respect to any two agents can be simulated by the system with two agents, all the properties checked by searching all reachable states of the two-agents system are verified to hold for the system of any number of agents.

CafeOBJ proof scores for verifying the simulation

simOfQLOCKbyQLOCKijPS.mod
csQtopPS.mod

These proof scores are almost same amount to the original proof score for verifying mutual exclusion. However, once the simulation is verified, many properties other than mutual exclusion can be verified by searching over the two-agents systems.

Remarks

- OTS style of equations support fast executions/ reductions of proof scores. They are much faster than search.
- Developing proof scores requires deep understanding of problems, and sometimes require serious efforts.
- OTS style definition of transition directly corresponds to rewriting transition.
- Search is sometimes quite effective and easy to use not only in falsification but also in verification. Especially for small values of parameters.
- Proper combination of search and inference (with proof score) can constitute transparent and effective verification.