Generate and Check Method for Invariant Verification in CafeOBJ

FUTATSUGI, Kokichi
Research Center for Software Verification (RCSV)
JAIST

Verification Methodology

1. Model and describe a system/problem in order-sorted (equational) algebraic specification.
2. Construct “proof score” and verify the specification by reductions/rewritings
MSV with Proof Scores in CafeOBJ

1. Understand problem and construct model
2. Write system spec SPsys and Write property spec SPprop
3. Construct Proof Score of SPprop w.r.t. SPsys

Transition System and Invariant

A majority of systems/problems in many fields can be modeled with transition systems.

**> System Spec
-- (1) states and transitions over them [State Trans]
-- (2) initial states predicate
op init : State -> Bool .
-- (3) transition function
op tr : State Trans -> State .

**> Property Spec
-- (4) invariant predicate
op inv : State -> Bool .

---

**> System Spec
-- (1) states and transitions over them [State Trans]
-- (2) initial states predicate
op init : State -> Bool .
-- (3) transition function
op tr : State Trans -> State .

**> Property Spec
-- (4) invariant predicate
op inv : State -> Bool .
State, init, tr, inv

Terminology for invariant

\[ \text{inv} = \text{p}_1 \text{ and p}_2 \text{ and ... and p}_n \]

- \( \text{inv} \): invariant
- \( \text{p}_1, \text{p}_2, \ldots, \text{p}_n \): invariant fragments

In Mann&Pnueli
- invariant \( \leftrightarrow \) inductive invariant
- invariant fragment \( \leftrightarrow \) invariant
Invariant Verification (Spec Verification)

Verify that the invariant predicate is true for any reachable state.
A sufficient condition is formalized as the following two conditions.

** (a) initial state condition

\[
\text{op \ init-c : State \to Bool}.
\]
\[
\text{eq \ init-c(S:State) = (init(S) \implies inv(S))}.
\]
\[
\text{-- (for-all \ S:State) (init-c(S))}
\]

** (b) invariant condition

\[
\text{op \ inv-c : State Trans \to Bool}.
\]
\[
\text{eq \ inv-c(S:State,T:Trans) =}
\]
\[
\text{(inv(S) \implies inv(tr(S,T)))}.
\]
\[
\text{-- (for-all \ S:State,T:Trans) (inv-c(S,T))}
\]

Generate and Check Method
-- goal and assumption

The **goal** is to prove that predicates

- \text{init-c(S:State)}
- \text{inv-c(S:State,T:Trans)}

are true for all ground terms of sorts \text{State} and \text{(State,Trans)} respectively.

We **assume** that

1. ground terms of sort \text{State} and \text{(State,Trans)} are partitioned into ill-formed and well-formed, and
2. \text{init-c} and \text{inv-c} is true for ill-formed ground terms by definition.
Generate and Check Method
-- for inv-c (1) (for init-c is similar)

Generate finit set of well-formed terms
(s_1,t_1),(s_2,t_2),...,(s_n,t_n)
((s_i,t_i) is non ground, i.e. contains variables)
of sort (State,Trans) systematically as follows.
(1) inv-c(s_i,t_i) (i = 1,2,...,m) are reduced to ‘true’
by rewriting (check).
(2) For any well-formed ground term (s_c,t_c) of sort
(State,Trans), there exits some j in {1,2,...,n}
such that
(s_c,t_c) = as(s_j,t_j)
is true for an appropriate assignment as.

(2) implies that
(inv-c(s_1,t_1) and inv-c(s_2,t_2)
and ... and inv-c(s_n,t_n))
implies
(for-all S:State,T:Trans)(inv-c(S,T))
is true, and because of (1)
inv-c(S:State,T:Trans)
(i.e. (for-all S:State,T:Trans)(inv-c(S,T)))
is proved.
An example: mutual exclusion protocol

Assume that many agents 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 (mechanism or algorithm) which can achieve the mutual exclusion is called “mutual exclusion protocol”.

QLOCK (locking with queue):
a mutual exclusion protocol

Each agent \(i\) is executing:

- **Put its name \(i\) into the bottom of the queue**
- **Remove/get the top of the queue**

**Atomic action**

- **Is \(i\) at the top of the queue?**
  - If true, enter the **Critical Section**
  - If false, put its name into the bottom of the queue and repeat.
Some Scenario of QLOCK

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 $rs$), 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)
System Specification
of QLOCK
as a Transition System

\[
\begin{align*}
\text{[State Trans]} & \\
\text{op init} & : \text{State} \rightarrow \text{Bool} \\
\text{op tr} & : \text{State Trans} \rightarrow \text{State} \\
\end{align*}
\]

LABEL and PID

-- labels for indicating location
-- of each agent; three point set
mod! LABEL {
-- literal labels and labels
[LabelLt < Label]
-- rs: remainder section
-- ws: waiting section
-- cs: critical section
ops rs ws cs : \rightarrow LabelLt \{constr\} .
\text{eq} \ (L1:\text{LabelLt} = L2:\text{LabelLt}) \ = \ (L1 == L2) .
}

-- agent identifiers
mod* AID {{Aid}}
QUEUE : first in first out storage (1)

-- queue (First In First Out storage)
-- for storing agent identifiers
mod! QUEUE (X :: TRIV) {
  -- elements and their queues
  [Elt < Qu]
  -- empty queue
  op empQ : -> Qu {constr}.
  -- associative queue constructor with id: empQ
  op (_&_) : Qu Qu -> Qu {constr assoc id: empQ}.
  -- equality _=_ over Qu
  eq (empQ = (E:Elt & Q:Qu)) = false.
  ceq ((E1:Elt & Q1:Qu) = (E2:Elt & Q2:Qu))
      = ((E1 = E2) and (Q1 = Q2))
      if not((Q1 = empQ) and (Q2 = empQ)).

QUEUE : first in first out storage (2)

-- head
op hd_ : Qu -> Elt.
eq hd(E:Elt & Q:Qu) = E.
-- hd(empQ) is not defined intentionally
-- a nice error handling method
-- tail
op tl_ : Qu -> Qu.
eq tl(E:Elt & Q:Qu) = Q.
-- tl(empQ) is not defined intentionally
-- a nice error handling method
}
**OBS: observers**

-- observers
mod! OBS {
pr(LABEL)
pr(QUEUE(AID{sort Elt -> Aid}))
-- there are two kinds of observers
[Obs]
op (qu:_ ) : Qu -> Obs {constr} .
op (lb[_]:_ ) : Aid Label -> Obs {constr} .
}

**STATE : sets of observers**

-- generic set
mod! SET(X :: TRIV) {
[El.X < Set]
op empty : -> Set {constr} .
op _ _ : Set Set -> Set
 {constr assoc comm id: empty} .
-- idempotency
eq E:Elt E = E .
}

-- a state is defined as a set of observers
mod! STATE {pr(SET(OBS{sort Elt -> Obs})
  {*{sort Set -> State})}}
Terms of the sort State

**Ill-Formed:**

(empty)
(qu: Q:Qu)
((qu: Q1:Qu)(qu: Q2:Qu))
(ld[A:Aid] L:Label)

**Well-Formed:**

  (ld[A:Aid] L2:Label) S:State)

STATEfuns (1)

-- elementary functions on states
mod! STATEfuns {
pr(NAT + STATE)
-- variable declarations
vars L1 L2 : Label .
vars A1 A2 : Aid .
var S : State .
var Q : Qu .
-- the number of queues in a state
op #q : State -> Nat .
eq #q(empty) = 0 .
eq #q((qu: Q) S) = 1 + #q(S) .
eq #q((ld[A1]: L1) S) = #q(S) .
STATEfuns (2)

-- the number of labels in a state
op #ls: State Label -> Nat .
eq #ls(empty,L1) = 0 .
eq #ls(((qu: Q) S,L1) = #ls(S,L1) .
eq #ls(((lb[A1]: L1) S),L2) =
    if (L1 = L2) then 1 + #ls(S,L2)
    else #ls(S,L2) fi .

-- the number of aids in a state
op #as: State Aid -> Nat .
eq #as(empty,A1) = 0 .
eq #as(((qu: Q) S,A1) = #as(S,A1) .
eq #as(((lb[A1]: L1) S),A2) =
    if (A1 = A2) then 1 + #as(S,A2)
    else #as(S,A2) fi .

STATEfuns (3)

-- the number of aids in a queue
op #aq: Qu Aid -> Nat .
eq #aq(empQ,A1) = 0 .
eq #aq(A1 & Q,A2) =
    if (A1 = A2) then 1 + #aq(Q,A2)
    else #aq(Q,A2) fi .
PNAMEcj
-- names of predicates on states
-- and conjunction of the predicates
mod! PNAMEcj {
pr(STATE)
-- names of predicates on States
-- and sequences of them
[Pname < PnameSeq]
op (_ _) : PnameSeq PnameSeq -> PnameSeq
{assoc} .
-- conjunction of predicates indicated in
PnameSeq
op cj : PnameSeq State -> Bool .
eq cj(PN:Pname PNS:PnameSeq,S:State)
  = cj(PN,S) and cj(PNS,S) .
}

STATEprop (1)
-- properties on states
-- for defining well formed states
-- and initial state condition
mod! STATEprop ( pr(STATEfuns) ex(PNAMEcj) )

-- one queue in a state
op 1q : -> Pname .
eq[1q]: cj(1q,S:State) = (#q(S) = 1) .

-- no duplication of Aid in a state
-- for an arbitrary Aid
op 1a : -> Pname .
eq[1a-1]: cj(1a,empty) = true .
eq[1a-2]: cj(1a,((lb[A:Aid]: L:Label) S:State))
  = (#as(S,A) = 0) and cj(1a,S) .
eq[1a-3]: cj(1a,((qu: Q:Qu) S:State)) = cj(1a,S) .
STATEprop (2)

-- qas pattern, only the state with this pattern
-- needs to be considered
pred qas : State .
  = true .
op qas : -> Pname .
eq[qas] : cj(qas,S:State) =
  if (qas(S) == true) then true else false fi .

-- well formed states
op wfs : -> Pname .
eq wfs = qas 1q 1a .
-- notice that only the state s
-- with (cj(wfs,s) = true)
-- represents a meaningful state

STATEprop (3)

-- the following two properties
-- are for defining initial condition
-- queue is empty
op qe : -> Pname .
eq[qe1]: cj(qe,empty) = false .
eq[qe2]: cj(qe,((lb[A:Aid]: L:Label) S:State))
  = (#q(S) = 1) and cj(qe,S) .
eq[qe3]: cj(qe,((qu: Q:Qu) S:State))
  = (Q = empQ) and (#q(S) = 0) .
-- any Aid is in rs, i.e. no ws, no cs
op allRs : -> Pname .
eq[allRs]: cj(allRs,S:State)
  = (#ls(S,ws) = 0) and (#ls(S/cs) = 0) .
}
**QLOCK (1)**

-- QLOCK
-- (mutual exclusion protocol with queue of aids)
-- defined as monoActionOTS
-- (observational transition system)

```
mod! QLOCK {
  pr(STATEprop)
  -- variables
  var Q : Qu .
  var L : Label .
  vars A1 A2 A3 : Aid .
  var S : State .

  -- initial state condition
  op init : -> PnameSeq .
  eq init = wfs ge allRs .
  pred init : State .
  eq init(S) = cj(init,S) .
```

**QLOCK (2)**

-- transitions and names of transitions
[Trans Tname]
-- want (wt), try (ty), and exit (ex)
ops wt ty ex : -> Tname .
-- transition is product of tname and aid
op (_ _) : Tname Aid -> Trans {constr} .

-- definition of transitions
```
op tr : State Trans -> State .
** notice that the following definitions assume
** (cj(wfs,((qu: Q)(lb[A1]: L) S)) = true) and
** (cj(wfs,((qu: (A1 & Q))(lb[A2]: L) S))
** = true)
```
QLOCK (3)

-- wt : wait
eq[wt]: tr(((\text{qu}: Q)(\text{lb}[A1]: L) S),(wt A2))
   = if ((L = rs) and (A1 = A2))
      then ((\text{qu}: (Q & A1))(\text{lb}[A1]: ws) S)
      else ((\text{qu}: Q)(\text{lb}[A1]: L) S)
   fi .

-- ty : try
eq[ty1]: tr(((\text{qu}: \text{empQ})(\text{lb}[A1]: L) S),(ty A2))
   = ((\text{qu}: \text{empQ})(\text{lb}[A1]: L) S) .
eq[ty2]:
   tr(((\text{qu}: (A1 & Q))(\text{lb}[A2]: L) S),(ty A3))
   = if ((A1 = A2) and (A2 = A3) and (L = ws))
      then ((\text{qu}: (A1 & Q))(\text{lb}[A2]: cs) S)
      else ((\text{qu}: (A1 & Q))(\text{lb}[A2]: L) S)
   fi .

QLOCK (4)

-- ex : exit
eq[ex1]: tr(((\text{qu}: \text{empQ})(\text{lb}[A1]: L) S),(ex A2))
   = ((\text{qu}: \text{empQ})(\text{lb}[A1]: L) S) .
eq[ex2]:
   tr(((\text{qu}: (A1 & Q))(\text{lb}[A2]: L) S),(ex A3))
   = if ((A1 = A2) and (A2 = A3) and (L = cs))
      then ((\text{qu}: Q)(\text{lb}[A2]: rs) S)
      else ((\text{qu}: (A1 & Q))(\text{lb}[A2]: L) S)
   fi .
Property Specification
(Invariant Specification)

QLOCKprop (1)

-- several properties on QLOCK and
-- an invariant property of QLOCK
mod! QLOCKprop {
  pr(QLOCK)
  -- variable declarations
  var L : Label . var A : Aid .
  var S : State . var Q : Qu .
  -- mutual exclusion property:
  -- at most one agent is with the label cs
  op mx : -> Pname .
  eq[mx]: cj(mx,S)
    = ((#ls(S,cs) = 0) or (#ls(S,cs) = 1)) .
QLOCKprop (2)

-- several fragment predicates for invariant
ops qep rs ws cs : -> Pname .
eq[qep]: \text{cj}(qep,((\text{qu}: Q)(\text{lb}[A]: L) S))
= ((Q = \text{emp}Q) \implies
  (\#ls(((\text{lb}[A]: L) S),cs) = 0)) .
eq[rs]: \text{cj}(rs,((\text{qu}: Q)(\text{lb}[A]: L) S))
= ((L = rs) \implies (\#aq(Q,A) = 0)) .
eq[ws]: \text{cj}(ws,((\text{qu}: Q)(\text{lb}[A]: L) S))
= ((L = ws) \implies
  ((\#aq(Q,A) = 1) \text{ and }
  ((A = \text{hd}(Q)) \implies
  (\#ls(S,cs) = 0)))) .

QLOCKprop (3)

\text{eq}[cs]: \text{cj}(cs,((\text{qu}: Q)(\text{lb}[A]: L) S))
= ((L = cs) \implies
  ((A = \text{hd}(Q))
  \text{ and (}\#aq(tl(Q),A) = 0)
  \text{ and (}\#ls(S,cs) = 0)))) .

-- invariant predicate
op inv : -> PnameSeq .
eq inv = qas mx qep rs ws cs .
pred inv : State .
eq inv(S) = \text{cj}(inv,S) .
}
-- the following two modules describe
-- the algorithm for generating a finite set
-- of patterns that cover all possible cases

-- predicate v that is to be checked
mod* PREDtbC {
  -- values and their sequences
  [Val < ValSq]
  op _,-, : ValSq ValSq -> ValSq {assoc} .
  -- predicate to be checked
  pred v : ValSq .
  }

PREDtbC
GENcases \( (X :: \text{PREDtbC}) \) (1)

-- generating a finite set of patterns
-- that cover all possible combinations
-- of values in a value sequence

mod! GENcases \( (X :: \text{PREDtbC}) \) {
-- sequences of values for expansions
[Val < V1Sq]
op \_\_ : V1Sq V1Sq -> V1Sq \{assoc\} .
-- sequence of ValSeq or VlSeq
[ValSq V1Sq < SqSq]
op \_\_ : SqSq SqSq -> SqSq \{assoc\} .
-- SqSq enclosures and their trees
[SqSqEn < SqSqTr]
op \[\_\] : SqSq -> SqSqEn .
op \[\_] : SqSqTr SqSqTr -> SqSqTr .

GENcases \( (X :: \text{PREDtbC}) \) (2)

-- expanding \(_;\_\) into \(_||\_\)
var V : Val .
var VS : VlSq .
vars SS1 SS2 : SqSq .
eq [((V;VS),SS2)] = [(V,SS2)] || [(VS,SS2)] .
eq [(SS1,(V;VS)),SS2)]
    = [(SS1,V,SS2)] || [(SS1,VS,SS2)] .
eq [(SS1,(V;VS))] = [(SS1,V)] || [(SS1,VS)] .
GENcases \((X :: \text{PREDtbC})\) (3)

-- indicators and their trees
\[\text{Ind} < \text{IndTr}\]
\[
\begin{align*}
\text{op} \ & : \rightarrow \text{Ind}.
\text{op} \ & |\_ : \text{IndTr} \text{IndTr} \rightarrow \text{IndTr}.
\end{align*}
\]
-- indicator constructor
\[
\begin{align*}
\text{op} \ & i : \text{Bool} \text{ValSq} \rightarrow \text{Ind} \{\text{constr}\}.
\end{align*}
\]
-- make indicator out of \((v : \text{ValSq} \rightarrow \text{Bool})\)
-- that comes from \((X :: \text{PREDtbC})\)
\[
\begin{align*}
\text{op} \ & m\_ : \text{ValSq} \rightarrow \text{Ind}.
\text{eq} \ & m(VSQ:\text{ValSq}) = i(v(VSQ),VSQ).
\end{align*}
\]

GENcases \((X :: \text{PREDtbC})\) (4)

-- make make indicators:
-- translating a tree of SqSq \((\text{SqSqTr})\)
-- into a tree of indicators
\[
\begin{align*}
\text{op} \ & mmi_ : \text{SqSqTr} \rightarrow \text{IndTr}.
\text{eq} \ & mmi(SST1:\text{SqSqTr} || SST2:\text{SqSqTr})
\ & = (mmi\ SST1) \mid (mmi\ SST2).
\end{align*}
\]
-- if all \_;\_ in SqSq disappear
-- then translate mmi to mi
\[
\begin{align*}
\text{eq} \ & mmi[VSQ:\text{ValSq}] = m(VSQ).
\end{align*}
\]
-- making all indicators with "true" disappear
\[
\begin{align*}
\text{eq} \ & i(\text{true},VSQ:\text{ValSq}) \mid \text{IT}\text{IndTr} = \text{IT}.
\text{eq} \ & \text{IT}\text{IndTr} \mid i(\text{true},VSQ:\text{ValSq}) = \text{IT}.
\end{align*}
\]
FACTtbu

-- facts to be used,
-- this part changes according to modification
-- of spec and proof score
mod FACTtbu {pr(QLOCKprop)}

-- necessary facts about _=_ on Nat
var N : Nat .
eq (1 = 0) = false .
eq ((1 + N) = 0) = false .
eq ((2 + N) = 0) = false .
eq ((2 + N) = 1) = false .
eq (1 + N = 1) = (N = 0) .
eq ((N = 0) and (N = 1)) = false .
-- necessary fact about #aq
eq #aq(Q:Qu & A1:Aid,A2:Aid) =
   if (A1 = A2) then 1 + #aq(Q,A2) else #aq(Q,A2) fi .
}

Verification of the Initial State Condition (1)

--> [0] cj(wfs,s) = false .
open QLOCKprop .
op s : => State .
eq cj(wfs,s) = false .
red init(s) implies inv(s) .
close
Verification of the Initial State Condition (2)

---> [1] cj(wfs, s) = true.
-- define v for initial condition
mod QLOCKinit {
  pr(QLOCKprop)
  [Qu Aid Label State < Val < ValSq]
  op _/_ : ValSq ValSq -> ValSq {assoc}.
  -- prePreds : predicates for premise
  -- conPreds : predicates for conclusion
  ops prePreds conPreds : -> PnameSeq.
  -- predicate to be checked
  op v : ValSq -> Bool.
  eq v(Q:Qu, A:Aid, L:Label, S:State)
     = cj(prePreds, ((qu: Q)(lb[A]: L) S))
     implies cj(conPreds, ((qu: Q)(lb[A]: L) S)) .
}

Verification of the Initial State Condition (3)

-- generate and check all possible cases
-- for initial condition
mod CKallCasesInit {
  ex(GENcases(QLOCKinit))
  -- Aid constant literals
  [AidConLt < Aid]
  eq (B1:AidConLt = B2:AidConLt) = (B1 == B2).
  -- arbitrary (ordinary or literal) constants
  ops b1 b2 : -> AidConLt. op q : -> Qu.
  op s : -> State. op ck : -> IndTr.
  -- a term of sort IndTr
  -- for checking all possible cases
  eq ck = ($ | mmi{ (empQ; (b1 & q)),
     (b1; b2),
     (rs; ws; cs),
     (s)) } .
}
Verification of the Initial State Condition (4)

-- reduction for verification
-- of initial state condition
open CKallCasesInit.
pr(FACTtbu)
eq prePreds = init .
eq conPreds = inv .
red ck .
close

Generation of a finite set patterns for init

\[ \{(\text{empQ};(b_1 \& q)), (b_1;b_2), (rs;ws;cs), (s)\} \]

generates a finite set of patterns that cover all
ground terms that are instances of the following
sequence of variables

\[ (Q:\text{Qu}, A:\text{Aid}, L:\text{Label}, S:\text{State}) \]
**Verification of the Invariant Condition (1)**

\[
\rightarrow [0] \ cj(wfs, s) = false .
\]

open QLOCKprop .
op s : -> State .
op t : -> Trans .
eq cj(wfs, s) = false .
red inv(s) implies inv(tr(s, t)) .
close

**Verification of the Invariant Condition (2)**

\[
\rightarrow [1] \ cj(wfs, s) = true .
\]

-- define \( v \) for invariant condition
mod QLOCKinv { 
pr(QLOCKprop) 
-- val and ValSeq 
[Qu Aid Label State Tname < Val < ValSq] 
op _/_ : ValSq ValSq -> ValSq \{assoc\} . 
-- prePreds : predicates for pre-condition 
-- postPreds : predicates for post-condition 
ops prePreds postPreds : -> PnameSeq . 
-- predicate to be checked 
op v : ValSq -> Bool . 
eq v(\texttt{Q:Qu,Al:Aid,L:Label,S:State,T:Tname,A2:Aid}) 
= cj(prePreds,((\texttt{qu:Q})(\texttt{lb[A1]:L}) S)) 
implies cj(postPreds, 
tr(((\texttt{qu:Q})(\texttt{lb[A1]:L}) S),(T A2))) . 
}
Verification of the Invariant Condition (3)

-- generate and check all possible cases for
-- invariant condition (162 cases)
mod CKallCasesInv {
  ex(GENcases(QLOCKinv))
-- Aid constant literals
  [AidConLt < Aid]
  eq (B1:AidConLt = B2:AidConLt) = (B1 == B2) .
-- arbitrary (ordinary or literal) constants
  ops b1 b2 b3 : -> AidConLt .
  op q : -> Qu . op s : -> State . op ck : -> IndTr .
  eq ck = ($ | mmi[(empQ;(b1 & q)),
                  (b1;b2),
                  (rs;ws;cs),
                  (s),
                  (wt;ty;ex),
                  (b1;b2;b3)]) .
}

Verification of the Invariant Condition (4)

-- reduction for verification
-- of invariant condition
open CKallCasesInv .
pr(FACTtbu)
eq prePreds = inv .
eq postPreds = inv .
red ck .
close
Generation of a finite set patterns for \( \text{inv} \)

\[
[(\text{emp}Q; (b_1 \& q)),
(b_1; b_2),
(r_s; w_s; c_s),
(s),
(w_t; t_y; e_x),
(b_1; b_2; b_3)] .
\]

generates a finite set of patterns that cover all ground terms that are instances of the following sequence of variables

\[
(Q: \text{Qu}, A_1: \text{Aid}, L: \text{Label}, S: \text{State}, T: \text{Tname}, A_2: \text{Aid})
\]

What has been proved

For any state \( s_{\text{init}} \) that satisfies the predicate \( \text{init} \), any reachable state from \( s_{\text{init}} \) satisfies \( \text{inv} \).
Remarks on the generate and check methodology based on states as sets of observers

- Easy to generating a set of finite patterns that covers all the possible cases.
- Easy to select an appropriate abstraction level for generating patterns depending on the kinds of state transitions to be investigated.
- Failure of reduction into “true” can be seen as an failure case, and gives a chance to find lemma and to debug/improve specification.

APPENDIX: ABP (1)

The following shows the transition pattern for ABP

$$\begin{align*}
((s\text{Bit}: B)(s\text{Num}: N)(s\text{rCh}: <\text{SR1}>)
\quad (r\text{Bit}: B)(r\text{Nums}: NS)(r\text{rCh}: <\text{RS1}>))
\quad \Rightarrow
\quad ((s\text{Bit}: B)(s\text{Num}: N)(s\text{rCh}: <\text{SR1}>)
\quad (r\text{Bit}: \text{not}(B))(r\text{Nums}: (N NS))(r\text{rCh}: <\text{RS1}>))
\quad \Rightarrow
\quad ((s\text{Bit}: B)(s\text{Num}: N)(s\text{rCh}: <\text{SR1}>)
\quad (r\text{Bit}: \text{not}(B))(r\text{Nums}: (N NS))(r\text{rCh}: (<\text{RS2} <\text{RS1}>) )
\quad \Rightarrow
\quad ((s\text{Bit}: B)(s\text{Num}: N)(s\text{rCh}: <\text{SR1}>)
\quad (r\text{Bit}: \text{not}(B))(r\text{Nums}: (N NS))(r\text{rCh}: <\text{RS2}>))
\quad \Rightarrow
\quad ((s\text{Bit}: B)(s\text{Num}: N)(s\text{rCh}: <\text{SR1}>)
\quad (r\text{Bit}: \text{not}(B))(r\text{Nums}: (N NS))(r\text{rCh}: <\text{RS2}>))
\quad \Rightarrow
\quad ((s\text{Bit}: B)(s\text{Num}: N)(s\text{rCh}: <\text{SR1}>)
\quad (r\text{Bit}: \text{not}(B))(r\text{Nums}: (N NS))(r\text{rCh}: <\text{RS2}>))
\end{align*}$$
APPENDIX: ABP (2)

Notice that <SR1>, <RS1>, <SR2>, and <RS2> represent some sr or rs sequences which satisfy the followings:

- \( \text{zeroD}(dn(B,N) <SR1>) = \text{true} \),
- \( \text{zeroC}(B <RS1>) = \text{true} \),
- \( \text{zeroD}(dn(\neg(B), (s N)) <SR2>) = \text{true} \),
- and \( \text{zeroC}(\neg(B) <RS2>) = \text{true} \).

APPENDIX: ABP (3)

This implies that the state pattern

\[
((s\text{Bit}: B)(s\text{Num}: N)(s\text{rCh}: <SR1>)
(r\text{Bit}: B)(r\text{Nums}: NS)(r\text{rCh}: <RS1>))
\]

or

\[
((s\text{Bit}: B)(s\text{Num}: N)(s\text{rCh}: <SR1>)
(r\text{Bit}: \neg(B))(r\text{Nums}: (N NS))(r\text{rCh}: (<SR2> <RS1>)))
\]

or

\[
((s\text{Bit}: \neg(B))(s\text{Num}: (s N))(s\text{rCh}: (<SR2> <SR1>))
(r\text{Bit}: \neg(B))(r\text{Nums}: (N NS))(r\text{rCh}: (<SR2>)))
\]

represents the class of state configurations all of which are transferable by possibly multiple transitions composed of drSr, duSr, drRs, and/or duRS.
APPENDIX: ABP (4)

--> s-sr (rs;sr)
open CKallCssl .
-- ft s-sr sro sr-r rso r-rs-s
eq prePreds = ft s-sr sro sr-r rso r-rs-s .
eq postPreds = s-sr .
-- (ss;drSr;duSr;rr;rs;drRs;duRs;sr)

APPENDIX: ABP (5)

-- BS,N,SRC,BR,NS,RSC,TN
eq sqEn =
[ (bt0),
  (nt1),
  (empBN;
    pg([(bt0;bt1),(nt1;nt2)],p);
    pg([(bt0;bt1),(nt1;nt2),src1,(bt0;bt1),(nt1;nt2;nt3)],
      psp)),
  (bt0;bt1),
  (ns),
  (empB;bt0;bt1;pg([(bt0;bt1),rsc1,(bt0;bt1)],bsb)),
  (rs;sr)
].
-- (ss;drSr;duSr;rr;rs;drRs;duRs;sr)
-- this generates 812 cases
red ck.
**> s-sr (sr)
close
"{}
(0.000 sec for parse, 30556 rewrites(4.860 sec),
279596 matches, 38053 memo hits)130510-21:34
(0.000 sec for parse, 342614 rewrites(22.150 sec),
5112296 matches)130521-10:06

18 similar ‘open..red..close’ units are needed for the total
verification, most of them take 1 to 5 seconds