

I636: Specification and Verification of Distributed Systems

5. Invariant Verification with search

Kazuhiro Ogata & Kokichi Futatsugi

Outline

- Observational transition systems (OTSs) are state machines, which can be used as mathematical models of distributed systems.
- This lecture describes OTSs, how to specify OTSs in Maude, and how to model check that OTSs (does not) satisfy invariant properties.

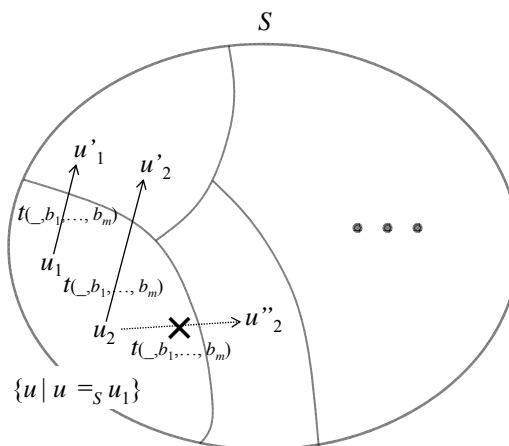
Observational Transition Systems (1)

- Assume that there exists a state space U and data types used in OTSs such as Bool.
- An OTS $S \langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ consists of
 - \mathcal{O} : A set of *observers*. Each $o : U \times V_{o1} \dots V_{om} \rightarrow V_o \in \mathcal{O}$ is a function that takes a state and m (≥ 0) data values, and returns a data value, which is the value of state component.
 $u_1 =_S u_2$ iff $o(u_1, x_1, \dots, x_m) = o(u_2, x_1, \dots, x_m)$ for each $o \in \mathcal{O}$ and each $x_i \in D_{oi}$ for $i = 1, \dots, m$.
 - \mathcal{I} : The set of *initial states*. \mathcal{I} is a subset of U .
 - \mathcal{T} : A set of *transitions*. Each $t : U \times V_{t1} \dots V_{tn} \rightarrow U \in \mathcal{T}$ is a function that takes a state and n (≥ 0) data values, and returns a state.
 Each $t \in \mathcal{T}$ preserves the equivalence relation $=_S$.
 That is, if $u_1 =_S u_2$, then $t(u_1, y_1, \dots, y_n) =_S t(u_2, y_1, \dots, y_n)$ for each $t \in \mathcal{T}$ and each $y_j \in D_{tj}$ for $j = 1, \dots, n$.
 Each $t \in \mathcal{T}$ is equipped with a condition $c-t : U \times V_{t1} \dots V_{tn} \rightarrow \text{Bool}$ called the *effective condition*.
 If $\neg c-t(u, y_1, \dots, y_n)$, then $t(u, y_1, \dots, y_n) =_S u$.

3

Observational Transition Systems (2)

- Preservation of the equivalence relation $=_S$ by transitions.



4

Observational Transition Systems (3)

- Each $t \in \mathbf{T}$ is defined in equations as follows:
 - For each $o \in \mathbf{O}$,

$$o(t(u, y_1, \dots, y_n), x_1, \dots, x_m) = e \text{ if } c\text{-}t(u, y_1, \dots, y_n)$$
 where e is an expression that consists of observers and variables appearing in the LHS, but does not contain any transitions.
 - There is one more equation:

$$t(u, y_1, \dots, y_n) =_S u \text{ if } \neg c\text{-}t(u, y_1, \dots, y_n)$$

5

Modeling Qlock (1)

- OTS S_{Qlock} modeling Qlock.
 - $O_{\text{Qlock}} = \{ \text{queue} : U \rightarrow \text{Queue}, \text{pc} : U \text{ Pid} \rightarrow \text{Label} \}$
 - $I_{\text{Qlock}} = \{ u \mid \text{queue}(u) = \text{empty}, \forall i:\text{Pid}. \text{pc}(u, i) = \text{rs} \}$
 - $T_{\text{Qlock}} = \{ \text{want} : U \text{ Pid} \rightarrow U, \text{try} : U \text{ Pid} \rightarrow U, \text{exit} : U \text{ Pid} \rightarrow U \}$

```

Loop {
  Remainder Section
  rs: atomic_enqueue(queue, i);
  ws: repeat until top(queue) = i;
  Critical Section
  cs: atomic_dequeue(queue)
}

```

Qlock

6

Modeling Qlock (2)

- want is defined as follows:
 $queue(want(u,i)) = enq(queue(u),i)$ if $c-want(u,i)$
 $pc(want(u,i),j) = (if\ i = j\ then\ ws\ else\ pc(u,j)\ fi)$ if $c-want(u,i)$
 $want(u,i) =_{S_{Qlock}} u$ if $\neg c-want(u,i)$
 where $c-want(u,i)$ is $pc(u,i) = rs$.
- try is defined as follows:
 $queue(try(u,i)) = queue(u)$ if $c-try(u,i)$
 $pc(try(u,i),j) = (if\ i = j\ then\ cs\ else\ pc(u,j)\ fi)$ if $c-try(u,i)$
 $try(u,i) =_{S_{Qlock}} u$ if $\neg c-try(u,i)$
 where $c-try(u,i)$ is $pc(u,i) = ws \wedge top(queue(u)) = i$.
- exit is defined as follows:
 $queue(exit(u,i)) = deq(queue(u))$ if $c-exit(u,i)$
 $pc(exit(u,i),j) = (if\ i = j\ then\ rs\ else\ pc(u,j)\ fi)$ if $c-exit(u,i)$
 $exit(u,i) =_{S_{Qlock}} u$ if $\neg c-exit(u,i)$
 where $c-exit(u,i)$ is $pc(u,i) = cs$.

7

Reachable States

- Reachable states are states that are reachable from an initial states by a sequence of transitions.
- *Reachable states* w.r.t. an OTS S are inductively defined as follows:
 - Each initial state $u_0 \in I$ is reachable w.r.t. S .
 - If u is reachable w.r.t. S , then so is $t(u, y_1, \dots, y_n)$ for each $t \in T$ and each $y_j \in D_{t_j}$ for $j = 1, \dots, n$.
 Let R_S be the set of all reachable states w.r.t. S .
- E.g.
 - Among reachable states w.r.t. S_{Qlock} :
 $u_0 \in I_{Qlock}, want(u_0,i), try(want(u_0,i),i)$
 - Among unreachable states w.r.t. S_{Qlock} :
 A state u such that $pc(u,i) = cs, pc(u,j) = cs$ and $i \neq j$
 A state u such that $pc(u,i) = cs$ and $top(queue(u)) \neq i$
 Note that they need to be verified.

8

Invariant Properties

- Invariant properties are properties that hold in every reachable states.
- A state predicate $p : U \rightarrow \text{Bool}$ is called *invariant* (or an *invariant property*) w.r.t. S if $p(u)$ holds for each $u \in \mathbf{R}_S$, i.e. $\forall u \in \mathbf{R}_S. p(u)$.
- E.g. Among invariant properties w.r.t. S_{Qlock} :
 - $\forall u \in \mathbf{R}_{S_{\text{Qlock}}}. \forall i, j: \text{Pid}. (\text{pc}(u, i) = \text{cs} \wedge \text{pc}(u, j) = \text{cs} \Rightarrow i = j)$
 - $\forall u \in \mathbf{R}_{S_{\text{Qlock}}}. \forall i: \text{Pid}. (\text{pc}(u, i) = \text{cs} \Rightarrow \text{top}(\text{queue}(u)) = i)$They need to be verified.

9

How to Specify OTSs in Maude (1)

- O and T are represented by sorts, say `Obs` and `Trans`.
 - Observers (Transitions) are represented by operators whose sorts are `Obs` (`Trans`).
- U is represented by a sort, say `Sys`, which is declared as a super-sort of `Obs` and `Trans`.
- States are represented as collections of terms whose tops are such operators, and such collections are constructed by

```
op void : -> Sys .
op _ _ : Sys Sys -> Sys
      [assoc comm id: void]
```

10

How to Specify OTSs in Maude (2)

- Each $o \in \mathcal{O}$ is represented by an operator
`op o[_ , ... , _] : _ : Vo1 ... Vom Vo -> Obs .`
- The term “ $o[a_1, \dots, a_m] : a$ ” appearing in a collection that corresponds to a state u represents (the instance $o(\cdot, a_1, \dots, a_m)$ of) the observer o such that $o(u, a_1, \dots, a_m) = a$.
- Each $t \in \mathcal{T}$ is represented by an operator
`op t : Vt1 ... Vtn -> Trans .`
- The term “ $t(b_1, \dots, b_n)$ ” appearing in a collection that corresponds to a state u represents (the instance $t(\cdot, b_1, \dots, b_n)$ of) the transition t .

11

How to Specify OTSs in Maude (3)

- E.g. The observers and transitions of S_{Qlock} are represented by the operators declared as follows:

```
*** Observers
op pc[_]:_ : Pid Label -> Obs .
op queue:_ : Queue -> Obs .
*** Transitions
op want : Pid -> Trans .
op try : Pid -> Trans .
op exit : Pid -> Trans .
```
- E.g. We suppose that there are two processes $p1$ and $p2$, and then the following term represents the initial state of Q_{lock} :

```
want(p1) try(p1) exit(p1)
want(p2) try(p2) exit(p2)
(queue : empty) (pc[p1] : rs) (pc[p2] : rs)
```

12

How to Specify OTSs in Maude (4)

- Each $t \in T$ is defined in rules.
- The basic form of a rule defining t is as follows:

```

crl [rule-t] :
  t(Y1, ..., Yn) (o[X1, ..., Xm] : X) ...
=> t(Y1, ..., Yn) (o[X1, ..., Xm] : a) ...
if c-t(Y1, ..., Yn) .

```
- Each $X_{(i)}$ ($Y_{(j)}$) is a Maude variable of sort $V_{o(i)}$ ($V_{t(j)}$), and a is a term of sort V_o .
- $t(Y_1, \dots, Y_n)$ denotes an arbitrary instance of $t \in T$.
- $(o[X_1, \dots, X_m] : X)$ denotes an arbitrary instance of $o \in O$.

13

How to Specify OTSs in Maude (5)

- E.g. `try` is defined in rules as follows:

```

crl [try] :
  try(I) (queue : Q) (pc[I] : L) (pc[I'] : L') ...
=> try(I) (queue : Q)
  (pc[I] : (if I = I then cs then L fi))
  (pc[I'] : (if I' = I then cs then L' fi)) ...
if L = ws /\ top(Q) = I .

```
- We suppose that each collection (state) contains at most one $(pc[P] : \dots)$ for each $P:Pid$, and therefore I' must be different from I .
 Then, the rule is simplified as follows:

```

crl [try] :
  try(I) (queue : Q) (pc[I] : L) (pc[I'] : L') ...
=> try(I) (queue : Q)(pc[I] : cs) (pc[I'] : L) ...
if L = ws /\ top(Q) = I .

```

14

How to Specify OTSs in Maude (6)

- If an instance of an observer is never involved in the transition, then it can be deleted from the rule.
- E.g. Since $(pc[I'] : L')$... is never involved in the transition, the rule `try` is modified as follows:

```

crl [try] :
  try(I) (queue : Q) (pc[I] : L)
  => try(I) (queue : Q)(pc[I] : cs)
  if L = ws /\ top(Q) = I .
  
```

15

How to Specify OTSs in Maude (7)

- If $c-t(Y_1, \dots, Y_n)$ contains $Z = d$ (or $d = Z$) as its conjunct where Z is a Maude variable and d is a ground term, then Z can be replaced with d in the LHS and RHS of the rule, and $Z = d$ (or $d = Z$) can be deleted from $c-t(Y_1, \dots, Y_n)$.
- If Z is Y_k , then Y_k and the corresponding argument can be deleted from $t(Y_1, \dots, Y_n)$ and the declaration of t such as

```

op t : Vt1 ... Vt(k-1) Vt(k+1) ... Vtn -> Trans .
  
```

- E.g. Since $L = ws$ appears in the condition, the rule `try` is modified as follows:

```

crl [try] :
  try(I) (queue : Q) (pc[I] : ws)
  => try(I) (queue : Q)(pc[I] : cs)
  if top(Q) = I .
  
```

16

How to Specify OTSs in Maude (8)

- If any Y_k appearing in $t(Y_1, \dots, Y_n)$ also appears anywhere else in the LHS, Y_k and the corresponding argument can be deleted from $t(Y_1, \dots, Y_n)$ and the declaration of t such as

```
op t : Vt1 ... Vt(k-1) Vt(k+1) ... Vtn -> Trans .
```

- E.g. Since I is such a variable, the rule `try` is modified as follows:

```
cr1 [try] :  
  try (queue : Q) (pc[I] : ws)  
  => try (queue : Q)(pc[I] : cs)  
  if top(Q) = I .
```

17

How to Specify OTSs in Maude (9)

- If t is a constant, t is deleted from the rule and the declaration is deleted from the specification.
- E.g. Now that `try` is a constant, the rule `try` is modified as follows:

```
cr1 [try] :  
  (queue : Q) (pc[I] : ws)  
  => (queue : Q)(pc[I] : cs)  
  if top(Q) = I .
```

18

How to Specify Initial States

- Initial states are expressed as ground terms whose sorts are *Sys*.
- To express a(n) (initial) state as a finite term, we may need to fix the number of entities involved in a distributed system.
- E.g. The number of processes should be fixed (say 2) for Qlock.

```
op init : -> Sys .  
eq init = (pc[p1]: rs) (pc[p2]: rs)  
          (queue: empty) .
```

19

Revisit to Command search

- `search` explores the reachable state space of a system described by a set of rules & an initial state using a breadth-first strategy.
`search [n,m] in M : t =>x p such that c .`
 - *n* is a bound on the number of solutions (option).
 - *m* is the maximum depth of the search (option).
 - *t* is the starting term (initial state).
 - *p* is the pattern that has to be reached.
 - `=>x` indicates the form of the transitions from *t* to *p* :
 - `=>1` means the transitions consisting of exactly one step.
 - `=>+` means the transitions consisting of one or more steps.
 - `=>*` means the transitions consisting of zero or more steps.
 - `=>!` Indicates that only canonical final states are allowed.
 - *c* is the condition that has to be satisfied by the reached state (option); the syntactic form of the condition is the same as the one of conditions for conditional equations and memberships.

20

How to Express Invariant Properties for Command search (1)

search $[n, m]$ in $M : t \Rightarrow_x p$ such that c .

- For invariant verification (or falsification) that an OTS S specified in M satisfies (or does not satisfy) an invariant property, i.e. $\forall u \in \mathbf{R}_S.q(u)$,
 - n is often set to 1.
 - m is omitted when the reachable state space is reasonably small or falsification is conducted.
 - \Rightarrow_x is \Rightarrow^* .
 - t is a ground term denoting an initial state.
 - $\neg q$ is expressed in the pattern p and the condition c .

21

How to Express Invariant Properties for Command search (2)

- One possible way to express $\neg q$ in the pattern p and the condition c :
 - p is set to a term denoting an arbitrary state.
 - c is set to a term denoting $\neg q$.
- E.g. To verify that S_{Qlock} satisfies the mutual exclusion property,
 - p is set to $S : \text{Sys}$.
 - c is set to $\text{not}(\text{pc}(S : \text{Sys}, I : \text{Pid}) == \text{cs and } \text{pc}(S : \text{Sys}, J : \text{Pid}) == \text{cs implies } I == J)$, where pc is declared and defined as follows:


```
op pc : Sys Pid -> Label .
eq pc((pc[I]: L) S, I) = L .
```

22

How to Express Invariant Properties for Command search (3)

- All variables appearing in c must appear in p .
- If c contains variables that do not appear in p , p should be modified or part of c should be expressed as part of p so that all variables in c appear in p .
- E.g. An arbitrary state of S_{Qlock} is also expressed as
(pc[I:Pid]: L1:Label)
(pc[J:Pid]: L2:Label) (S:Sys),
and then p is set to the term.

```
search [1] in QLOCK-INIT : init
  =>* (pc[I:Pid]: L1:Label)
      (pc[J:Pid]: L2:Label) (S:Sys)
such that not(L1:Label == cs and
              L2:Label == cs implies I:Pid == J:Pid) .
```

23

How to Express Invariant Properties for Command search (4)

- One possible way to express part of c as part of p :
 - If c is in the form $\text{not}(p_1 \text{ and } p_2 \text{ implies } p_3)$ and all variables in p_2 and p_3 also appear in p_1 , then p_1 is expressed as part of c , making all variables in p appear in c .
- E.g. $L1:\text{Label} == \text{cs}$ and $L2:\text{Label} == \text{cs}$ is expressed as part of the pattern as follows:
(pc[I:Pid]: cs) (pc[J:Pid]: cs) (S:Sys).
search [1] in QLOCK-INIT :
 init =>* (pc[I:Pid]: cs)
 (pc[J:Pid]: cs) (S:Sys)
 such that not(I:Pid == J:Pid) .

24

How to Express Invariant Properties for Command `search` (5)

- Some variables in p and/or c can be instantiated.
- E.g. We suppose that the two processes are involved in Qlock, and then I and J are instantiated by the two processes.

```
search [1] in QLOCK-INIT :  
  init =>* (pc[p1]: cs) (pc[p2]: cs) (S:Sys)  
  such that not(p1 == p2) .
```

Since $p1$ is different from $p2$, the condition always holds.

```
search [1] in QLOCK-INIT :  
  init =>* (pc[p1]: cs) (pc[p2]: cs) (S:Sys) .
```

25

Ticket0: A (Flawed) Mutex Protocol

The program for each process i :

```
Loop {  
  Remainder Section  
  rs: ticket[i] := vm;  
  is: vm := vm + 1;  
  ws: repeat until ticket[i] = turn;  
  Critical Section  
  cs: turn := turn + 1;  
}
```

- vm and $turn$ are natural number variables shared by all process.
- $ticket[i]$ is a natural number variable that is local to process i .
- rs , is , ws , and cs are labels.
- Initially, every variable is set to 0 and each process is at label rs .

26

OTS S_{Ticket0} Modeling Ticket0 (1)

- $O_{\text{Ticket0}} = \{ \text{vm} : U \rightarrow \text{Nat}, \text{turn} : U \rightarrow \text{Nat}, \text{ticket} : U \text{ Pid} \rightarrow \text{Nat}, \text{pc} : U \text{ Pid} \rightarrow \text{Label} \}$
- $I_{\text{Ticket0}} = \{ u \mid \text{vm}(u) = 0, \text{tuen}(u) = 0, \forall i:\text{Pid}.\text{ticket}(u,i) = 0, \forall i:\text{Pid}.\text{pc}(u,i) = \text{rs} \}$
- $T_{\text{Ticket0}} = \{ \text{get} : U \text{ Pid} \rightarrow U, \text{inc} : U \text{ Pid} \rightarrow U, \text{try} : U \text{ Pid} \rightarrow U, \text{exit} : U \text{ Pid} \rightarrow U \}$

27

OTS S_{Ticket0} Modeling Ticket0 (2)

- get is defined as follows:
 $\text{vm}(\text{get}(u,i)) = \text{vm}(u)$ if c-get(u,i)
 $\text{turn}(\text{get}(u,i)) = \text{turn}(u)$ if c-get(u,i)
 $\text{ticket}(\text{get}(u,i),j) = (\text{if } i=j \text{ then } \text{vm}(u) \text{ else } \text{ticket}(u,j) \text{ fi})$ if c-get(u,i)
 $\text{pc}(\text{get}(u,i),j) = (\text{if } i=j \text{ then } \text{is} \text{ else } \text{pc}(u,j) \text{ fi})$ if c-get(u,i)
 $\text{get}(u,i) =_{S_{\text{Ticket0}}} u$ if $\neg \text{c-get}(u,i)$
 where c-get(u,i) is $\text{pc}(u,i) = \text{rs}$.
- inc is defined as follows:
 $\text{vm}(\text{inc}(u,i)) = \text{vm}(u) + 1$ if c-inc(u,i)
 $\text{turn}(\text{inc}(u,i)) = \text{turn}(u)$ if c-inc(u,i)
 $\text{ticket}(\text{inc}(u,i),j) = \text{ticket}(u,j)$ if c-inc(u,i)
 $\text{pc}(\text{inc}(u,i),j) = (\text{if } i=j \text{ then } \text{ws} \text{ else } \text{pc}(u,j) \text{ fi})$ if c-inc(u,i)
 $\text{inc}(u,i) =_{S_{\text{Ticket0}}} u$ if $\neg \text{c-inc}(u,i)$
 where c-inc(u,i) is $\text{pc}(u,i) = \text{is}$.

28

OTS S_{Ticket0} Modeling Ticket0 (3)

- try is defined as follows:
 - $\text{vm}(\text{try}(u,i)) = \text{vm}(u)$ if $\text{c-try}(u,i)$
 - $\text{turn}(\text{try}(u,i)) = \text{turn}(u)$ if $\text{c-try}(u,i)$
 - $\text{ticket}(\text{try}(u,i),j) = \text{ticket}(u,j)$ if $\text{c-try}(u,i)$
 - $\text{pc}(\text{try}(u,i),j) = (\text{if } i = j \text{ then } \text{cs} \text{ else } \text{pc}(u,j) \text{ fi})$ if $\text{c-try}(u,i)$
 - $\text{try}(u,i) =_{S_{\text{Ticket0}}} u$ if $\neg \text{c-try}(u,i)$
 - where $\text{c-try}(u,i)$ is $\text{pc}(u,i) = \text{rs} \wedge \text{ticket}(u,i) = \text{turn}(u)$.
- exit is defined as follows:
 - $\text{vm}(\text{exit}(u,i)) = \text{vm}(u)$ if $\text{c-exit}(u,i)$
 - $\text{turn}(\text{exit}(u,i)) = \text{turn}(u) + 1$ if $\text{c-exit}(u,i)$
 - $\text{ticket}(\text{exit}(u,i),j) = \text{ticket}(u,j)$ if $\text{c-exit}(u,i)$
 - $\text{pc}(\text{exit}(u,i),j) = (\text{if } i = j \text{ then } \text{rs} \text{ else } \text{pc}(u,j) \text{ fi})$ if $\text{c-exit}(u,i)$
 - $\text{exit}(u,i) =_{S_{\text{Ticket0}}} u$ if $\neg \text{c-exit}(u,i)$
 - where $\text{c-exit}(u,i)$ is $\text{pc}(u,i) = \text{cs}$.

29

Specification of S_{Ticket0} (1)

- The observers and transitions of S_{Ticket0} are represented by the operators declared as follows:


```
*** Observers
op vm:_ : Nat -> Obs .
op turn:_ : Nat -> Obs .
op ticket[_]:_ : Pid Nat -> Obs .
op pc[_]:_ : Pid Label -> Obs .
*** Transitions
op get : Pid -> Trans .
op inc : Pid -> Trans .
op try : Pid -> Trans .
op exit : Pid -> Trans .
```

30

Specification of S_{Ticket0} (2)

- The transition `get` is specified in rules as follows:

```
cr1 [get] : get(I) (tv: N) (turn: M)
(ticket[I]: X) (pc[I]: L) (ticket[I']: X')
(pc[I']: L')...
=> get(I) (tv: N) (turn: M) (ticket[I]: N)
(pc[I]: is) (ticket[I']: X') (pc[I']: L')...
if L = rs .
```
- `(turn: M)` and `(ticket[I']: X') (pc[I']: L')...` are not involved in the transition, and then deleted.

```
cr1 [get] : get(I) (tv: N) (ticket[I]: X)
(pc[I]: L)
=> get(I) (tv: N) (ticket[I]: N) (pc[I]: is)
if L = rs .
```

31

Specification of S_{Ticket0} (3)

- `L` is replaced with `rs` and `L = rs` is deleted.

```
r1 [get] : get(I) (tv: N) (ticket[I]: X)
(pc[I]: rs)
=> get(I) (tv: N) (ticket[I]: N) (pc[I]:
is).
```
- `I` in `get(I)` also appears in `(pc[I]: rs)` and `(ticket[I]: X)`, and then is deleted from `get(I)`. Moreover, `get` becomes a constant, and then is deleted.

```
r1 [get] : (tv: N) (ticket[I]: X)
(pc[I]: rs)
=>(tv: N) (ticket[I]: N) (pc[I]: is).
```

32

Specification of S_{Ticket0} (4)

- The remaining three transitions are specified as follows:

```
rl [inc] : (pc[I]: is) (vm: N)
  => (pc[I]: ws) (vm: (N + 1)) .
```

```
rl [try] : (pc[I]: ws) (ticket[I]: N)
  (turn: N)
  => (pc[I]: cs) (ticket[I]: N) (turn: N) .
```

```
rl [exit] : (pc[I]: cs) (turn: N)
  => (pc[I]: rs) (turn: (N + 1)) .
```

33

Specification of S_{Ticket0} (5)

- If we suppose that two processes are involved in Ticket0, the initial state is specified as follows:

```
mod TICKET-INIT is
  pr TICKET .
  op init : -> Sys .
  eq init
    = (pc[p1]: rs) (ticket[p1]: 0)
      (pc[p2]: rs) (ticket[p2]: 0)
      (vm: 0) (turn: 0) .
endm
```

34

Falsification of S_{Ticket0} with search

- The command to find a counterexample showing that S_{Ticket0} does not satisfy the mutex property.
 - search [1] in TICKET-INIT : init =>* (pc[p1]: cs) (pc[p2]: cs) (S:Sys) .
 Solution 1 (state 28)
 S:Sys --> vm: 2 turn: 0 (ticket[p1]: 0) ticket[p2]: 0
- The command to show the counterexample.
 - show path 28 .
 ... ===[...get...] ===>
 state1, ... (vm: 0) (ticket[p1]: 0) (ticket[p2]: 0) ...
 ... ===[...get...] ===>
 state1, ... (vm: 0) (ticket[p1]: 0) (ticket[p2]: 0) ...
 - A same ticket 0 is given to both p1 and p2 because the following is not performed atomically:
 rs: ticket[i] := vm;
 is: vm := vm + 1;

35

Modification of Ticket0 and S_{Ticket0}

<pre> Loop { Remainder Section rs: ticket[i] := vm; is: vm := vm + 1; ws: repeat until ticket[i] = turn; Critical Section cs: turn := turn + 1; } </pre>	<pre> Ticket1 rs: ticket[i] := vm + 1; </pre>
--	---

```

r1 [get] : (tv: N) (ticket[I]: X) (pc[I]: rs)
=>(tv: N) (ticket[I]: N) (pc[I]: is).
r1 [inc] : (pc[I]: is) (vm: N)
=> (pc[I]: ws) (vm: (N + 1)) .

```

```

r1 [get] : (tv: N) (ticket[I]: X) (pc[I]: rs)
=>(tv: (N + 1)) (ticket[I]: N) (pc[I]: ws).

```

S_{Ticket1}

36

Verification of S_{Ticket1} with search

- The command to verify that S_{Ticket1} satisfies the mutex property when two processes are involved.
– `search [1] in TICKET1-INIT : init =>*
 (pc[p1]: cs) (pc[p2]: cs) (S:Sys) .`
- But, it does not terminate because the reachable state of S_{Ticket1} is unbounded.
- So, we need to fix the maximum depth of the search, say 100.
– `search [1,100] in TICKET1-INIT : init
 =>* (pc[p1]: cs) (pc[p2]: cs) (S:Sys) .
 No solution.`