

I636: Specification and Verification of Distributed Systems

13. Proof Score Approach to Verification

Kazuhiro Ogata & Kokichi Futatsugi

Outline

- *Proof scores* are proof plans written in an algebraic specification language.
- This lecture describes how to write proof scores of invariant properties in Maude by exemplifying the verification that a mutual exclusion protocol satisfies the mutual exclusion property.

Specifications of OTSs for Theorem Proving

- Each $t \in T$ is defined in equations as follows:
 - For each $o \in O$,
 - $\text{ceq } o(t(S, Y_1, \dots, Y_n), X_1, \dots, X_m) = E \text{ if } c\text{-}t(S, Y_1, \dots, Y_n) \text{ .}$
 - where E is a term that consists of observers and variables appearing in the LHS, but does not contain any transitions.
 - When E is $o(S, X_1, \dots, X_m)$, the following unconditional equation can be used:
 - $\text{eq } o(t(S, Y_1, \dots, Y_n), X_1, \dots, X_m) = o(S, X_1, \dots, X_m) \text{ .}$
 - When $c\text{-}t(S, Y_1, \dots, Y_n)$ always holds, the following unconditional equation can be used:
 - $\text{eq } o(t(S, Y_1, \dots, Y_n), X_1, \dots, X_m) = E \text{ .}$
 - There is one more equation:
 - $\text{ceq } t(S, Y_1, \dots, Y_n) = S \text{ if } c\text{-}t(S, Y_1, \dots, Y_n) \text{ .}$
 - When $c\text{-}t(S, Y_1, \dots, Y_n)$ always holds, this equation can be omitted.

3

Proof Scores of Invariant Properties (1)

- Let us suppose that we prove $\forall u \in \mathbf{R}_S. p(u)$, where p is $\forall d_1 : D_1. \dots \forall d_l : D_l. P(u, d_1, \dots, d_l)$ and u, d_1, \dots, d_l are all the variables appearing in p .
- $\forall u \in \mathbf{R}_S. p(u)$ is proved by induction on u .
- The proof is conducted by writing proof scores and executing them.
- Two modules **INV** and **ISTEP** are first declared:
 - **INV**: p is specified.
 - **ISTEP**: Formulas to prove in induction cases are specified.
- The proof score of $\forall u \in \mathbf{R}_S. p(u)$ is written by doing two things:
 - *Case splitting*
 - *Conjecture of lemmas*

When a lemma is conjectured, the lemma is specified in **INV** and **ISTEP**.
- The proof scores of all lemmas are written.

4

Proof Scores of Invariant Properties (2)

- $\text{MOD-}S$ is the module in which an OTS S is specified.
- The constant s is used to denote an arbitrary state.
- The constant z_i is used to denote an arbitrary value of V_i .

```
fmod INV is
  pr MOD- $S$  .
  *** arbitrary values
  op s : -> Sys .
  op  $z_1$  : ->  $V_1$  . ... op  $z_l$  : ->  $V_l$  . ...
  *** invariant properties
  op inv1 : Sys  $V_1$  ...  $V_l$  -> Bool . ...
  *** Maude variables
  var S : Sys . var  $Z_1$  :  $V_1$  . ... var  $Z_l$  :  $V_l$  . ...
  *** definitions
  eq inv1(S,  $Z_1, \dots, Z_l$ ) = P(S,  $Z_1, \dots, Z_l$ ) .
  ...
endfm
```

5

Proof Scores of Invariant Properties (3)

- $\text{inv1}(s', z_1, \dots, z_n)$ is the formula to prove in each induction case.
- $\text{inv1}(s, z_1, \dots, z_n)$ is an instance of the induction hypothesis.
- Since the instance is often used, istep1 is defined as this.

```
fmod ISTEP is
  pr INV .
  *** arbitrary successor state
  op s' : -> Sys .
  *** formulas to prove in induction cases
  op istep1 : -> Bool . ...
  *** definitions
  eq istep1
  = inv1(s,  $z_1, \dots, z_l$ ) implies inv1(s',  $z_1, \dots, z_l$ ) .
  ...
endfm
```

6

Proof Scores of Invariant Properties (4)

- Since Maude does not provide `open` & `close` commands, functional modules are used to write proof scores.
- Basic fragments of a proof score are called *proof passages*.
- Writing proof scores in Maude, a proof passage consists of one functional module and one reduction command.

7

Proof Scores of Invariant Properties (5)

- The proof of the base case is like

```
fmod BASE is
  pr INV .
endfm
red inv1 (init, z1, ..., zl) .
```
- It is often the case that Maude returns `true` for the proof passage.
- If Maude does not return `true`, we need to split the case (the proof passage) into multiple sub-cases.
- A typical case splitting is to choose a proposition found in the result and split the case into two sub-cases based on the proposition: (1) one where the proposition is true and the (2) other where the proposition is false.

8

Proof Scores of Invariant Properties (6)

- When a proposition b is used for the case splitting, the proof passage is split into the two proof passages:

```

*** b
fmod BASE is
  pr INV .
  *** assumptions
  eq b = true .
endfm
red inv1(init, z1, ..., zi) .

*** ~b
fmod BASE is
  pr INV .
  *** assumptions
  eq b = false .
endfm
red inv1(init, z1, ..., zi) .

```

- A comment for each proof passage should be used to make it clear that the proof passage corresponds to which case.
- Case splitting is done until Maude returns either `true` or `false` for each proof passage.
- If `true` is returned for each proof passage, the proof of the base case is successful.
- If `false` is returned for some proof passage, you may need to use some lemmas or you disprove the invariant property.

9

Proof Scores of Invariant Properties (7)

- In the induction case, the proof passage for each transition t is like

```

fmod STEP is
  pr ISTEP .
  *** arbitrary values
  op x1 : -> V11 . ... op xn : -> V1n .
  *** assumptions
  *** successor state
  eq s' = t(s, x1, ..., xn) .
endfm
red istep1 .

```

- It is often the case that Maude returns neither `true` nor `false` for the proof passage.
- We often need to do case splitting.

10

Proof Scores of Invariant Properties (8)

- When $c-t(s, x_1, \dots, x_n)$ does not always hold, $c-t(s, x_1, \dots, x_n)$ is first used to split the proof passage.

```

*** c-t(s, x1, ..., xn)
fmod STEP is
  pr ISTEP .
  *** arbitrary values
  op x1 : -> V11 .
  ... op xn : -> Vm .
  *** assumptions
  eq c-t(s, x1, ..., xn) = true .
  *** successor state
  eq s' = t(s, x1, ..., xn) .
endfm
red istep1 .

*** ~c-t(s, x1, ..., xn)
fmod STEP is
  pr ISTEP .
  *** arbitrary values
  op x1 : -> V11 .
  ... op xn : -> Vm .
  *** assumptions
  eq c-t(s, x1, ..., xn) = false .
  *** successor state
  eq s' = t(s, x1, ..., xn) .
endfm
red istep1 .

```

- true must be returned for the case where $c-t(s, x_1, \dots, x_n)$ is false b/c t changes nothing essentially in the case.
- It is often the case that Maude returns neither true nor false for the other case.

Proof Scores of Invariant Properties (9)

- We suppose that $c-t$ is defined as follows:
eq $c-t(S, X_1, \dots, X_n) = c_1$ and ... and c_k .
- The proof passage for the case where $c-t(s, x_1, \dots, x_n)$ is true is modified as follows:

```

*** c-t(s, x1, ..., xn)
fmod STEP is
  pr ISTEP .
  *** arbitrary values
  op x1 : -> V11 .
  ... op xn : -> Vm .
  *** assumptions
  --- eq c-t(s, x1, ..., xn) = true .
  eq c1 = true . ... eq ck = true .
  ---
  *** successor state
  eq s' = t(s, x1, ..., xn) .
endfm
red istep1 .

```

- It is still often the case that Maude returns neither true nor false.
- If that is the case, case splitting should be done like the base case.
- Basically, case splitting is repeated until Maude returns either true or false for each proof passage.
- If true is returned for a proof passage, the corresponding case is discharged.
- If false is returned for a proof passage, the corresponding case may represent unreachable states and then we need to use lemmas to discharge the case.

Proof Scores of Invariant Properties (10)

- We suppose that Maude returns `false` for the proof passage:

```

*** c-t(s, x1, ..., xn)
fmod STEP is
  pr ISTEP .
  *** arbitrary values
  op x1 : -> Vt1 .
  ... op xn : -> Vtn .
  *** assumptions
  eq1 ... eqa
  *** successor state
  eq s' = t(s, x1, ..., xn) .
endfm
red istep1 .

```

One possible lemma candidate is
`not(q1 and ... and qa)`
 where each q_i is
 • $l = r$ if eq_i is `eq l = r .`,
 • l if eq_i is `eq l = true .`, or
 • `not l` if eq_i is `eq l = false .`,
 and each constant denoting an arbitrary value is replaced with the corresponding variable.

Generally, a lemma is a state predicate that implies the candidate.

When lemmas lem_1, \dots, lem_b are used, the following command is used:

```
red (lem1 and ... and lem_b) implies istep1 .
```

13

Verification that Ticket satisfies the Mutual Exclusion Property

- Ticket is a mutual exclusion protocol.
- Ticket is modeled as an OTS S_{Ticket} .
- S_{Ticket} is specified in Maude for theorem proving.
- It is verified that S_{Ticket} satisfies the mutual exclusion property by writing proof scores in Maude and executing them with Maude.

14

Ticket: A Mutual Exclusion Protocol

- The program used by each process i in Ticket is as follows:

```

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

- $atomicInc(x)$ atomically does the following:
 - $ticket[i] := x$;
 - $x := x + 1$;
- Initially,
 - Each process i is in the remainder section (at label rs),
 - $vm = 0$,
 - $turn = 0$, and
 - $ticket[i] = 0$ for each i .

Note that “rs: $ticket[i] := vm + 1$,” is used on page 36 of lecture note 5, but should be “rs: $ticket[i] := atomicInc(vm)$.”

15

OTS S_{Ticket} (1)

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

```

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

16

OTS S_{Ticket} (2)

- get is defined as follows:

$vm(\text{get}(u,i)) = vm(u) + 1$ **if** $c\text{-get}(u,i)$
 $turn(\text{get}(u,i)) = turn(u)$ **if** $c\text{-get}(u,i)$
 $ticket(\text{get}(u,i),j) = (\text{if } i = j \text{ then } vm(u) \text{ else } ticket(u,j))$ **if** $c\text{-get}(u,i)$
 $pc(\text{get}(u,i),j) = (\text{if } i = j \text{ then } ws \text{ else } pc(u,j))$ **if** $c\text{-get}(u,i)$
 $get(u,i) =_{S_{\text{Ticket}}} u$ **if** $\neg c\text{-get}(u,i)$
where $c\text{-get}(u,i)$ is $pc(u,i) = rs$.

```

Loop {
  Remainder Section
  rs:  $ticket[i] := \text{atomicInc}(vm)$ ;
  ws: repeat until  $ticket[i] = turn$ ;
  Critical Section
  cs:  $turn := turn + 1$ ;
}
  
```

- try is defined as follows:

$vm(\text{try}(u,i)) = vm(u)$ **if** $c\text{-try}(u,i)$
 $turn(\text{try}(u,i)) = turn(u)$ **if** $c\text{-try}(u,i)$
 $ticket(\text{try}(u,i),j) = ticket(u,j)$ **if** $c\text{-try}(u,i)$
 $pc(\text{try}(u,i),j) = (\text{if } i = j \text{ then } cs \text{ else } pc(u,j))$ **if** $c\text{-try}(u,i)$
 $\text{try}(u,i) =_{S_{\text{Ticket}}} u$ **if** $\neg c\text{-try}(u,i)$
where $c\text{-try}(u,i)$ is $pc(u,i) = rs \wedge ticket(u,i) = turn(u)$.

17

OTS S_{Ticket} (2)

- exit is defined as follows:

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

```

Loop {
  Remainder Section
  rs:  $ticket[i] := \text{atomicInc}(vm)$ ;
  ws: repeat until  $ticket[i] = turn$ ;
  Critical Section
  cs:  $turn := turn + 1$ ;
}
  
```

18

Data Used in S_{Ticket}

- Boolean values: Denoted by `Bool`; Specified in the built-in module `BOOL`.
- Natural numbers: Denoted by `Nat`; Basically specified in the built-in module `NAT`.
- Process Identifiers: Denoted by `Pid`; Specified in `PID`.
- Labels: Denoted by `Label`; Specified in `LABEL`.

19

Natural Numbers

- The following module is declared:

```
fmod NAT+ is pr NAT .
  sort Nat+ .  subsort Nat < Nat+ .
  op _+_ : Nat+ Nat+ -> Nat+ [ditto] .
  op _<_ : Nat+ Nat+ -> Bool [ditto] .
  op _=_ : Nat+ Nat+ -> Bool [comm] .
  vars X Y : Nat+ .  vars M N : Nat .
  eq (X = X) = true .
  eq (M = N) = (M == N) .
  *** Lemmas
  eq X < X = false .
  ceq X < 1 + Y = true if X < Y or X = Y .
endfm
```

Subsort overloading

} ditto specifies that the operator inherits all attributes from the corresponding operator declared before.

Two lemmas are used in the verification.

- `Nat` is used to express natural number instances.
- `Nat+` is used to express (arbitrary) natural numbers.
- `_=_` is used to compare natural number instances, but `_==_` is used to compare (arbitrary) natural numbers.

20

Process Identifiers & Labels

- The following modules are declared:

```
fmod PID is  sort Pid .
  op == : Pid Pid -> Bool [comm] .
  var I : Pid .
  eq (I = I) = true .
endfm

fmod LABEL is  sorts LabelInst Label .
  subsort LabelInst < Label .
  ops rs ws cs : -> LabelInst .
  op == : Label Label -> Bool [comm] .
  var L : Label .  vars LI1 LI2 : LabelInst .
  eq (L = L) = true .
  eq (LI1 = LI2) = (LI1 == LI2) .
endfm
```

21

Specification of $S_{\text{Ticket}} (1)$

- The following operators are declared:

```
*** an arbitrary initial state
op init : -> Sys .
*** observers
op vm    : Sys -> Nat+ .
op turn  : Sys -> Nat+ .
op ticket : Sys Pid -> Nat+ .
op pc    : Sys Pid -> Label .
*** transitions
op get   : Sys Pid -> Sys .
op try   : Sys Pid -> Sys .
op exit  : Sys Pid -> Sys .
```

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

- The equations defining an arbitrary initial state:

```
eq vm(init) = 0 .
eq turn(init) = 0 .
eq ticket(init,I) = 0 .
eq pc(init,I) = rs .
```

22

Specification of S_{Ticket} (2)

- The equations defining transitions:

```

op c-get : Sys Pid -> Bool .
eq c-get(S,I) = (pc(S,I) = rs) .
***
ceq vm(get(S,I)) = vm(S) + 1 if c-get(S,I) .
eq turn(get(S,I)) = turn(S) .
ceq ticket(get(S,I),J)
  = (if I = J then vm(S) else ticket(S,J) fi)
  if c-get(S,I) .
ceq pc(get(S,I),J) = (if I = J then ws else pc(S,J) fi)
  if c-get(S,I) .
ceq get(S,I) = S if not c-get(S,I) .

```

```

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

```

23

Specification of S_{Ticket} (3)

- The equations defining transitions (cont):

```

op c-try : Sys Pid -> Bool .
eq c-try(S,I)
  = (pc(S,I) = ws and ticket(S,I) = turn(S)) .
***
eq vm(try(S,I)) = vm(S) .
eq turn(try(S,I)) = turn(S) .
eq ticket(try(S,I),J) = ticket(S,J) .
ceq pc(try(S,I),J) = (if I = J then cs else pc(S,J) fi)
  if c-try(S,I) .
ceq try(S,I) = S if not c-try(S,I) .

```

```

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

```

24

Specification of S_{Ticket} (4)

- The equations defining transitions:

```
op c-exit : Sys Pid -> Bool .
eq c-exit(S,I) = (pc(S,I) = cs) .
***
eq vm(exit(S,I)) = vm(S) .
ceq turn(exit(S,I)) = turn(S) + 1 if c-exit(S,I) .
eq ticket(exit(S,I),J) = ticket(S,J) .
ceq pc(exit(S,I),J) = (if I = J then rs else pc(S,J) fi)
    if c-exit(S,I) .
ceq exit(S,I) = S if not c-exit(S,I) .
```

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

25

Verification (1)

- The following are declared in the module INV:

```
op s : -> Sys . ops i j : -> Pid .
op invl : Sys Pid Pid -> Bool .
eq invl(S,I,J)
  = (pc(S,I) = cs and pc(S,J) = cs) implies (I = J) .
```

- The following are declared in the module ISTEP:

```
op s' : -> Sys .
op istep1 : -> Bool .
eq istep1 = invl(s,i,j) implies invl(s',i,j) .
```

26

Verification (2)

- Let us consider the proof passage:

```
fmod STEP is
pr ISTEP .
**** arbitrary values
op k : -> Pid .
*** assumptions
--- eq c-try(s,k) = true .
eq pc(s,k) = ws .
eq ticket(s,k) = turn(s) .
---
*** successor state
eq s' = try(s,k) .
endfm
red istep1 .
```

- Maude returns neither true nor false.
- We need to do case splitting.
- The result contains $i = k$.
- $i = k$ is used to split the case into two sub-cases.
- But, Maude still returns neither true nor false for both sub-cases.
- The result contains $j = k$.
- $i = k \& j = k$ are used to split the case into four sub-cases: (1) $i = k \& j = k$, (2) $i = k \& j \neq k$, (3) $i \neq k \& j = k$, and (4) $i \neq k \& j \neq k$.
- Maude returns true for (1) and (4).
- Maude returns true xor $cs = pc(s, j)$ for (2).
- $cs = pc(s, j)$ is used to split case (2) into two subcases: (2.1) true case & (2.2) false case.
- Maude returns false for (2.1) and true for (2.2).

27

Verification (3)

- Let us consider the proof passage:

```
fmod STEP is
pr ISTEP .
**** arbitrary values
op k : -> Pid .
*** assumptions
--- eq c-try(s,k) = true .
eq pc(s,k) = ws .
eq ticket(s,k) = turn(s) .
---
eq i = k .
eq (j = k) = false .
eq pc(s,j) = cs .
*** successor state
eq s' = try(s,k) .
endfm
red istep1 .
```

- Maude returns false.
- A lemma should be conjectured.
- One possible lemma can be conjectured by combining all assumptions with conjunction and negating it.
- The following is added to INV:


```
eq inv2(S,I,J)
= not(pc(S,I) = ws and
      ticket(S,I) = turn(S) and
      not(J = I) and pc(S,J) = cs) .
```
- The following is added to ISTEP:


```
eq istep2 = inv2(s,i,j)
      implies inv2(s',i,j) .
```
- $inv2(s,i,j)$ implies $istep1$ reduces to true.
- Case (3) $i \neq k \& j = k$ can be treated as well.

28

Verification (4)

- `inv1` can be proved with case splitting and one lemma `inv2`.
- When we write the proof score of `inv2`, we notice that we need to conjecture two lemmas as we did for `inv1`.
- The two lemmas are as follows

```
eq inv3(S,I,J)
= not(pc(S,I) = rs and not(J = I) and
      turn(S) = vm(S) and pc(S,J) = cs) .
eq inv4(S,I,J)
= not(pc(S,I) = ws and pc(S,J) = ws and
      ticket(S,I) = turn(S) and
      ticket(S,J) = turn(S) and not(I = J)) .
```

29

Verification (5)

- Let us consider the proof passage:

```
fmod STEP is
pr ISTEP .
**** arbitrary values
op k : -> Pid .
*** assumptions
--- eq c-get(s,k) = true .
eq pc(s,k) = rs .
---
eq (i = k) = false . eq (j = k) = false .
eq (i = j) = false . eq (turn(s) = vm(s)) = false .
---
eq turn(s) = 1 + vm(s) .
eq pc(s,i) = rs . eq pc(s,j) = cs .
---
*** successor state
eq s' = get(s,k) .
endfm
*** check
red istep3 .
```

- Maude returns `vm(s) = 1 + vm(s)`, which should be false.
- We need a lemma.

30

Verification (6)

- The lemma is
 $\text{not}(\text{pc}(s, i) = \text{rs} \text{ and } \text{not}(J = I) \text{ and } \text{turn}(S) = 1 + \text{vm}(S) \text{ and } \text{pc}(S, J) = \text{cs})$
- But, since inv3 is
 $\text{not}(\text{pc}(S, I) = \text{rs} \text{ and } \text{not}(J = I) \text{ and } \text{turn}(S) = \text{vm}(S) \text{ and } \text{pc}(S, J) = \text{cs})$,
it seems that the proof of the lemma needs another lemma, which is
 $\text{not}(\text{pc}(s, i) = \text{rs} \text{ and } \text{not}(J = I) \text{ and } \text{turn}(S) = 2 + \text{vm}(S) \text{ and } \text{pc}(S, J) = \text{cs})$
- So, we need to strengthen inv3 as follows:
 $\text{eq inv3}(S, J)$
 $= \text{not}(\text{pc}(S, J) = \text{cs} \text{ and } \text{not}(\text{turn}(S) < \text{vm}(S)))$.
- Note that the new inv3 implies the old inv3 .

31

Verification (7)

- The proof of inv3 needs two lemmas.
- One is inv1 .
- The other is as follows:
 $\text{eq inv5}(S, J)$
 $= \text{not}(\text{pc}(S, J) = \text{ws} \text{ and } \text{ticket}(S, J) = \text{turn}(S) \text{ and } \text{not}(\text{turn}(S) < \text{vm}(S)))$.

32

Verification (8)

- When we write the proof score of `inv4`, we notice that `inv4` should be strengthened as we did for `inv3`.
- The strengthened `inv4` is as follows:
eq `inv4(S, I, J)`
= `not(pc(S, I) = ws and pc(S, J) = ws and
ticket(S, I) = ticket(S, J) and not(I = J))` .
- When we write the proof score of the new `inv4`, we also notice that `inv5` should be strengthened.
- The strengthened `inv5` is as follows:
eq `inv5(S, J)`
= `not(pc(S, J) = ws and
not(ticket(S, J) < vm(S)))` .

33

Verification (9)

- The proof of `inv4` needs one lemma, which is `inv5`.
- The proof of `inv5` does not need any lemmas.

34