Introduction to Specification & Verification in CafeOBJ
3. Observational Transition Systems

Kazuhiro Ogata
(JAIST)

Gist

• Observational transition systems, or OTSs are mathematical model (transition systems, or state machines) of systems.
• Tiny examples are specified/modeled as OTSs.
• It is proved that the mutual exclusion property holds for a mutual exclusion protocol using an atomic instruction.
Roadmap

• Some more concepts on CafeOBJ
  – hidden sorts, behavioral operators
  – behavioral specifications
• Observational Transition Systems (OTSs)
  – Two tiny examples are modeled as OTSs.
• A mutual exclusion protocol using an atomic instruction.

A Simple State Machine (1)

• Let us consider a state machine INCXY whose transition diagram can be depicted as follows:
A Simple State Machine (2)

• INCXY is specified in CafeOBJ as follows:

```obj
mod* INCXY { pr(NAT)
  *[Sys]*
  op init : -> Sys
  bops x y : Sys -> Nat
  bops inc-x inc-y : Sys -> Sys
  var S : Sys
  -- init
  eq x(init) = 0 .
  eq y(init) = 0 .
  -- for inc-x
  eq x(inc-x(S)) = x(S) + 1 .
  eq y(inc-x(S)) = y(S) .
  -- for inc-y
  eq x(inc-y(S)) = x(S) .
  eq y(inc-y(S)) = y(S) + 1 .
}
```

• Hidden sort Sys represents the state space.
• Constant init denotes an arbitrary initial state.
• Operators x & y are used to observe the values of x & y.
• Operators inc-x & inc-y correspond to transitions inc-x & inc-y.
• The 1st two equations define an arbitrary initial state init.
• The next two equations define transition inc-x.
• The last two equations define transition inc-y.

Hidden Sorts

• Two types of sorts:
  – Visible sorts correspond to collections of data values (or data types).
    Nat corresponds to the set of natural numbers.
  – Hidden sorts correspond to collections of states (or state spaces) of state machines.
    Sys corresponds to the state space of INCXY.
Behavioral Operators

- An operator whose arity contains hidden sorts is called a *behavioral operator*.
  - A behavioral operator whose arity contains exactly one hidden sort and whose coarity is a visible sort is called an *observation function*.
    x and y are observation functions in INCXY.
  - A behavioral operator whose arity contains exactly one hidden sort and whose coarity is the same hidden sort is called a *transition function*.
    inc-x and inc-y are transition functions in INCXY.

Behavioral Specifications

- *Behavioral specifications* are algebraic specifications of (distributed) systems.
  - The whole state space is denoted as a hidden sort.
  - States are characterized by observation functions.
  - State transitions correspond to transition functions.
- Behavioral specifications generalize conventional transition system in that even if every observation function returns a same value for tow states in a behavioral specification, the two states may be different.
Another Simple State Machine (1)

• Let us consider a state machine INCXS whose transition diagram can be depicted as follows:

Another Simple State Machine (2)

• INCXS is specified in CafeOBJ as follows:

``` OBJ
mod* INCXS {
pr(PNAT)
*[Sys]*
op init : -> Sys
bop x : Sys Nat -> Nat
bop inc-x : Sys Nat -> Sys
var S : Sys vars I J : Nat
eq x(init,I) = 0 .
eq x(inc-x(S,I),J) = if I = J then x(S,I) + s(0)
else x(S,J) fi .
}
```

• Sys is the hidden sort denoting the state space of INCXS.
• Constant init is used to denote an arbitrary initial state.
• Operator x is an observation function, which is used to observe the value of each xi.
• Operator inc-x is a transition function, which corresponds to each transition inc-x.
• init is defined in the 1st equation, saying that each xi is zero.
• inc-x is defined in the 2nd equation, saying that in state inc-x(S,I), xi is incremented and xj is not for any other j.
Observational Transition Systems (1)

• *Observational transition systems*, or OTSs are mathematical models of (distributed) systems.
• OTSs can be regarded as a proper sub-class of behavioral specifications, corresponding to conventional transition systems.
• When every observation returns a same value for two states in an OTS, the two states are equal with respect to the OTS.

Observational Transition Systems (2)

• An OTS $S$ consists of the following:
  – $O$: A set of observation functions.
    \[ \text{bop } o : \text{Sys } V_{o1} \ldots \rightarrow V_o \]
    For two states $s_1,s_2:\text{Sys}$, $s_1$ is equal to $s_2$ wrt $S$ iff
    \[ o(s_1,x_{o1},\ldots) = o(s_2,x_{o1},\ldots) \text{ for all } o \in O, x_{o1} : V_{o1}, \ldots \]
  – $I : I \subseteq \text{Sys}$
  – $T$: A set of transition functions.
    \[ \text{bop } t : \text{Sys } V_{t1} \ldots \rightarrow \text{Sys} \]
    Each $t$ has a condition called the effective condition:
    \[ \text{bop } c-t : \text{Sys } V_{t1} \ldots \rightarrow \text{Bool} \]
    If $c-t(s,y_{t1},\ldots)$ does not hold, $t(s,y_{t1},\ldots)$ is equal to $s$ wrt $S$. 
Observational Transition Systems (3)

• A transition function \( t \in T \) in an OTS \( S \) are defined as follows.
  – For each \( o \in O \), we have an equation like
    \[
    \text{ceq } o \left( t\left( S, Y_{t1}, \ldots \right), X_{o1}, \ldots \right) = o-t\left( S, Y_{t1}, \ldots, X_{o1}, \ldots \right)
    \]
    \[
    \text{if } c-t\left( S, Y_{t1}, \ldots \right) .
    \]
    \( o-t\left( S, Y_{t1}, \ldots, X_{o1}, \ldots \right) \) does not consists of any transition functions.
  – If \( c-t\left( S, Y_{t1}, \ldots \right) \) always holds or \( t \) does not change the value returned by \( o \), we can omit the condition.
    \[
    \text{eq } o \left( t\left( S, Y_{t1}, \ldots \right), X_{o1}, \ldots \right) = o-t\left( S, Y_{t1}, \ldots, X_{o1}, \ldots \right) .
    \]
  – We have one more equation like
    \[
    \text{bceq } t\left( S, Y_{t1}, \ldots \right) = S \text{ if not } c-t\left( S, Y_{t1}, \ldots \right) .
    \]

A Mutual Exclusion Protocol
Using an Atomic Instruction (1)

• The program executed by each process:

```
Loop {
    Remainder Section
    ws: repeat while fetch\&store(locked,true);
    Critical Section
    cs: locked := false;
}
```

• \( locked \) is a Boolean variable shared by all processes.
• \( \text{fetch\&store}(x,v) \) atomically sets the variable \( x \) to the value \( v \) and returns the old value of \( x \).
• Initially, \( locked \) is false, and each process is at label \( ws \).
• The protocol is called \textit{Mutex}. 
A Mutual Exclusion Protocol Using an Atomic Instruction (2)

- The state transitions of each process can be depicted as follows:

  if \( \neg \text{fetch\&store}(\text{locked}, \text{true}) \)

  \[ ws \quad \xrightarrow{\text{if}} \quad cs \]

  \( \text{locked} := \text{false} \)

Modeling Mutex as an OTS (1)

- Two observation functions
  - \( \text{bop pc} : \text{Sys Pid} \rightarrow \text{Label} \)
  - \( \text{bop locked} : \text{Sys} \rightarrow \text{Bool} \)
  
  For a state \( s \) and a process ID \( i \), \( \text{pc}(s, i) \) returns the location (label \( \text{rs} \) or \( \text{cs} \)) of process \( i \) and \( \text{locked}(s) \) returns the value of \( \text{locked} \) in state \( s \)

- A state can be depicted as follows.

  \[ \ldots, \text{pc}[i]: l_i, \ldots, \text{pc}[j]: l_j, \ldots, \text{locked}: b \]

- Two transition functions
  - \( \text{bop try} : \text{Sys Pid} \rightarrow \text{Sys} \)
  - \( \text{bop exit} : \text{Sys Pid} \rightarrow \text{Sys} \)
  
  \( \text{try} \) corresponds to one iteration of the loop at label \( \text{ws} \) and \( \text{exit} \) corresponds to the assignment at label \( \text{cs} \).
Modeling Mutex as an OTS (2)

- An arbitrary initial state is declared and defined as follows:
  
  \[
  \begin{align*}
  \text{op } \text{init} & : \rightarrow \text{Sys} \\
  \text{eq } \text{pc}(\text{init}, I) & = \text{rs} . \\
  \text{eq } \text{locked}(\text{init}) & = \text{false} .
  \end{align*}
  \]

Modeling Mutex as an OTS (3)

- Transition function \text{try} is defined as follows:
  
  \[
  \begin{align*}
  \text{op } \text{c-try} & : \text{Sys Pid } \rightarrow \text{Bool} \\
  \text{eq } \text{c-try}(S, I) & = (\text{pc}(S, I) = \text{rs} \text{ and not } \text{locked}(S)) . \\
  \text{ceq } \text{pc}(\text{try}(S, I), J) & = (\text{if } I = J \text{ then } \text{cs} \text{ else } \text{pc}(S, J) \text{ fi}) \\
  \text{if } \text{c-try}(S, I) . \\
  \text{ceq } \text{locked}(\text{try}(S, I)) & = \text{true if } \text{c-try}(S, I) . \\
  \text{bceq } \text{try}(S, I) & = S \text{ if not } \text{c-try}(S, I) .
  \end{align*}
  \]
Modeling Mutex as an OTS (4)

- Transition function `try` is defined as follows:

```plaintext
op c-exit : Sys Pid -> Bool
eq c-exit(S,I) = (pc(S,I) = cs).
--
ceq pc(exit(S,I),J)
  = (if I = J then rs else pc(S,J) fi)
if c-exit(S,I).
ceq locked(exit(S,I)) = false if c-exit(S,I).
bceq exit(S,I) = S if not c-exit(S,I).
```

Modeling Mutex as an OTS (5)

- The transition diagram of the OTS is like
Invariant Properties

• **Reachable states** wrt an OTS $S$ are inductively defined as follows:
  – Each initial state $s_0 \in I$ is reachable wrt $S$.
  – If $s$ is reachable wrt $S$, then $t(s, y_{i1}, \ldots)$ is also reachable wrt $S$ for all $t \in T, y_{i1} : V_{i1}, \ldots$

Let $R_S$ be the set of all reachable states wrt $S$.

• A state predicate $p : Sys \rightarrow \text{Bool}$ is **invariant** (or is called an **invariant property**) wrt $S$ if $p$ holds in each reachable state, i.e. $\forall s \in R_S p(s)$.

---

Proof by Induction on Reachable States

• For a state predicate $p : Sys \rightarrow \text{Bool}$, the following two formulas are equivalent:
  1. $\forall s \in R_S p(s)$.
  2. $p(s_0)$ for all $s_0 \in I$, and $p(s) \Rightarrow p(t(s, y_{i1}, \ldots))$ for all $s \in R_S, t \in T, y_{i1} : V_{i1}, \ldots$

• Therefore, to prove (1), it suffices to show
  (i) Base case: $p(s_0)$ for an arbitrary initial state $s_0 \in I$.
  (ii) Induction case: $p(t(s, y_{i1}, \ldots))$ assuming $p(s)$ for an arbitrary $s \in R_S, t \in T, y_{i1} : V_{i1}, \ldots$

• (i) is called the **base case**, and (ii) the **induction case**.
• $p(s)$ are called the **induction hypotheses**.
Mutual Exclusion Property

- The *mutual exclusion property* is that *there are at most one process in the critical section at any given moment.*
- One desired property satisfied by mutual exclusion protocols such as Mutex.
- The property is rephrased as “if there are processes in the critical section, then those processes are identical.”
- Then, the property is formalized as the invariant property $\forall s \in R_{SMutex}^* \text{mutex}(s)$, where mutex is defined as follows:
  $$\forall i,j : \text{Pid.}(\text{pc}(s,i) = \text{cs} \land \text{pc}(s,j) \Rightarrow i = j)$$

Proof of the Mutex Property (1)

- The following module is first declared:

```plaintext
mod INV {
  pr(MUTEX)
  op s : -> Sys
  ops i j : -> Pid
  op inv1 : Sys Pid Pid -> Bool
  var S : Sys vars I J : Pid
  eq invi(S, I, J)
    = (pc(S, I) = cs and pc(S, J) = cs implies I = J) .
}
```

$\forall s \in R_{P Mutex}^* \text{mutex}(s)$
Proof of the Mutex Property (2)

• The following module is first declared:

```plaintext
mod ISTEP {  
pr(INV)  
op s' : -> Sys  
op istep1 : -> Bool  
eq istep1 = inv1(s,i,j) implies inv1(s',i,j) .  
}
```

• In each proof passage, $s'$ is replaced with a concrete successor state such as \( \text{try}(s,k) \), where \( k \) is a constant to denote an arbitrary process ID.

Proof of the Mutex Property (3)

• The proof passage of the base case is as follows:

```plaintext
open INV  
red inv1(init,i,j) .  
close  
```

CafeOBJ returns true for this proof passage.
Proof of the Mutex Property (4)

• The initial proof passage of the induction case where \texttt{try} is taken into account is as follows:

\begin{verbatim}
open ISTEP
  op k : -> Pid .
  eq s' = try(s,k) .
  red istep1 .
close
\end{verbatim}

• The case is split into two sub-cases based on the effective condition \texttt{c-try}.

\begin{verbatim}
open ISTEP
  op k : -> Pid .
  eq c-try(s,k) = true .
  eq s' = try(s,k) .
  red istep1 .
close
\end{verbatim}

CafeOBJ returns neither \texttt{true} nor \texttt{false}.

\begin{verbatim}
open ISTEP
  op k : -> Pid .
  eq c-try(s,k) = false .
  eq s' = try(s,k) .
  red istep1 .
close
\end{verbatim}

CafeOBJ returns \texttt{true}.

Proof of the Mutex Property (5)

• Instead of \texttt{c-try}(s,k) = true, we use the two equations as follows:

\begin{verbatim}
open ISTEP
  op k : -> Pid .
  eq c-try(s,k) = true .
  eq pc(s,k) = rs .
  eq locked(s) = false .
  eq s' = try(s,k) .
  red istep1 .
close
\end{verbatim} \hspace{1cm} \begin{verbatim}
open ISTEP
  op k : -> Pid .
  eq c-try(s,k) = false .
  eq s' = try(s,k) .
  red istep1 .
close
\end{verbatim}

(1) \hspace{1cm} (2)

• This is because (1) can be deduced from (2) by rewriting, but (2) cannot be deduced from (1) by rewriting, although (2) can be deduced from (1) by equational reasoning.

• Propositions \( k = i \) and \( k = j \) appear in the result returned by CafeOBJ for the proof passage, and then they are used to split the case into four sub-cases.

• CafeOBJ returns \texttt{true} for Case 1 & Case 4, but neither \texttt{true} nor \texttt{false} for Case 2 & Case 3.

\begin{tabular}{|c|c|c|}
  \hline
  \textbf{Case} & \textbf{\( k = i \)} & \textbf{\( k = j \)} \\
  \hline
  \textbf{Case 1} & true & true \\
  \textbf{Case 2} & true & false \\
  \textbf{Case 3} & false & true \\
  \textbf{Case 4} & false & false \\
  \hline
\end{tabular}
Proof of the Mutex Property (6)

- The proof passage of Case 2 is as follows:

```plaintext
pen ISTEP
op k : -> Pid .
  -- eq c-try(s,k) = true .
  eq pc(s,k) = rs .
  eq locked(s) = false .
  --
  eq i = k .
  eq (j = k) = false .
  eq s' = try(s,k) .
  red istep1 .
close
```

- CafeOBJ returns true for Case 2-1, but false for Case 2-2.

- CafeOBJ returns \((pc(s,j) = cs)\) xor true for the proof passage.

- Then, Case 2 is split into two sub-cases based the proposition \(pc(s,j) = cs\).

<table>
<thead>
<tr>
<th>Case 2-1</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 2-2</td>
<td>true</td>
</tr>
</tbody>
</table>

Proof of the Mutex Property (7)

- The proof passage of Case 2-2 is as follows:

```plaintext
pen ISTEP
op k : -> Pid .
  -- eq c-try(s,k) = true .
  eq pc(s,k) = rs .
  eq locked(s) = false .
  --
  eq i = k .
  eq (j = k) = false .
  eq pc(s,j) = cs .
  eq s' = try(s,k) .
  red istep1 .
close
```

- The five equations characterize Case 2-2.

- For the success of the proof, any state characterized by the five equations must be unreachable.

- A lemma can be conjectured from the five equations by basically combining them with conjunctions, negating the result, and replacing constants denoting arbitrary values with variables.

- Instead of use of \(i = k\), \(k\) is replaced with \(i\) in the rest of the equations.

\[
eq inv2(S,I,J) = not(pc(S,I) = rs and pc(S,J) = cs and not locked(S) and not(I = J)) .
\]
Proof of the Mutex Property (8)

- inv2 is used as follows:

```
pen ISTEP
  op k : -> Pid .
  -- eq c-try(s,k) = true .
  eq pc(s,k) = rs .
  eq locked(s) = false .
  --
  eq i = k .
  eq (j = k) = false .
  eq pc(s,j) = cs .
  eq s' = try(s,k) .
  red inv2(s,i,j) implies istep1 .
close
```

CafeOBJ returns true for the proof passage.

Proof of the Mutex Property (9)

- Case 3 can be discharged in the same way.
- Case 3 is first split into two sub-cases; one sub-case needs inv2.

```
pen ISTEP
  op k : -> Pid .
  -- eq c-try(s,k) = true .
  eq pc(s,k) = rs .
  eq locked(s) = false .
  --
  eq (i = k) = false .
  eq j = k .
  eq (pc(s,i) = cs) = false .
  eq s' = try(s,k) .
  red istep1 .
Close
```

```
pen ISTEP
  op k : -> Pid .
  -- eq c-try(s,k) = true .
  eq pc(s,k) = rs .
  eq locked(s) = false .
  --
  eq (i = k) = false .
  eq j = k .
  eq pc(s,i) = cs .
  eq s' = try(s,k) .
  red inv2(s,j,i) implies istep1 .
close
```

CafeOBJ return true for both proof passages.
Proof of the Mutex Property (10)

• The induction case where exit is taken into account can be proved likewise, which does not need to use any lemmas. All you need to do is case splitting.
• inv2 can be proved likewise. The proof needs to use inv1 as a lemma.

Exercise

• Complete the proof.