Algebraic specification and verification with CafeOBJ

Part 3 – CloudSync

Norbert Preining

ESSLLI 2016 Bozen, August 2016

Observational Transition Systems

System specification with OTS

- describe the system as state machine (automaton)

System specification with OTS

- describe the system as state machine (automaton)
- one state is a set of observations
- describe the transitions of the system
- describe initial states

System specification with OTS

- describe the system as state machine (automaton)
- one state is a set of observations
- describe the transitions of the system
- describe initial states
- find an invariant of transitions that guarantees the target property

System specification with OTS

- describe the system as state machine (automaton)
- one state is a set of observations
- describe the transitions of the system
- describe initial states
- find an invariant of transitions that guarantees the target property
- base case of induction
  - find a finite set of covering state descriptions
  - show for those that if a state is initial then the invariant property holds

System specification with OTS

- describe the system as state machine (automaton)
- one state is a set of observations
- describe the transitions of the system
- describe initial states
- find an invariant of transitions that guarantees the target property
- base case of induction
  - find a finite set of covering state descriptions
  - show for those that if a state is initial then the invariant property holds
- step case of induction
  - find again a finite set of covering state descriptions for the left hand sides of the transitions
  - show that if the lhs of the transition satisfies the invariant condition, then also the rhs.

CloudSync

CloudSync in images

Cloud		
	state	idle
	stamp	n

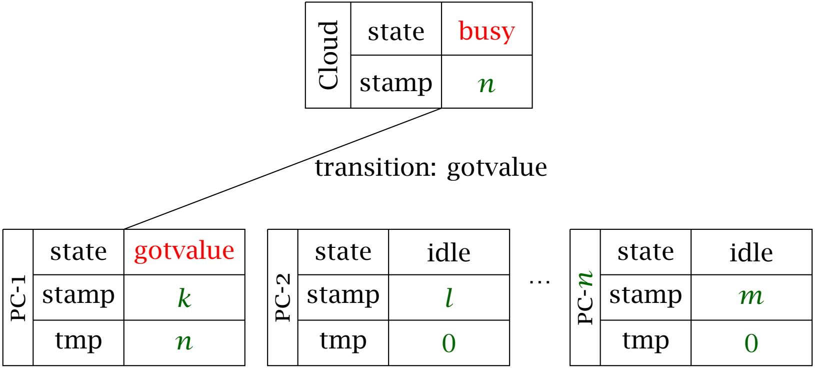
PC-1		
	state	idle
	stamp	k
	tmp	0

PC-2		
	state	idle
	stamp	l
	tmp	0

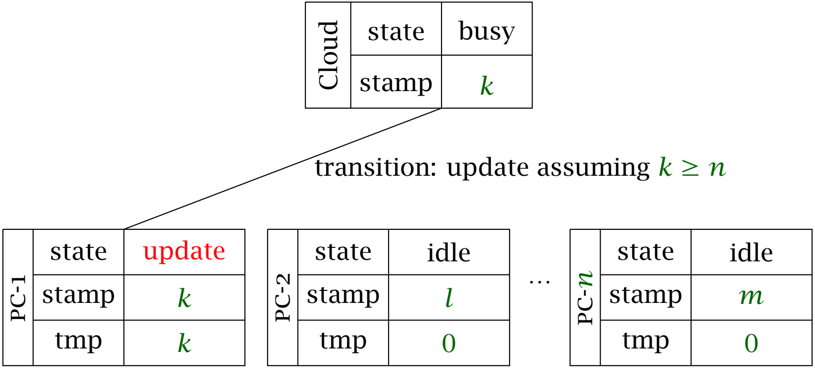
...

PC-n		
	state	idle
	stamp	m
	tmp	0

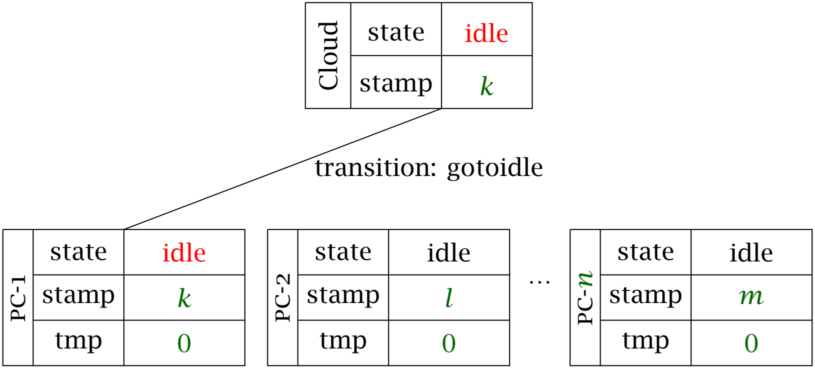
CloudSync in images



CloudSync in images



CloudSync in images



SPECIFICATION

ClLabel: {idlecl, busy}

```
mod! CLLABEL {
 [ClLabelLt < ClLabel]
 ops idlecl busy : -> ClLabelLt {constr} .
 eq (L1:ClLabelLt = L2:ClLabelLt) = (L1 == L2) .
}
```

SPECIFICATION

ClLabel: {idlecl, busy}
PcLabel: {idlepc, gotvalue, updated}

```
mod! PCLABEL {
 [PcLabelLt < PcLabel]
 ops idlepc gotvalue updated : -> PcLabelLt {constr} .
 eq (L1:PcLabelLt = L2:PcLabelLt) = (L1 == L2) .
}
```

Specification

ClLabel: $\{\text{idlecl}, \text{busy}\}$
PcLabel: $\{\text{idlepc}, \text{gotvalue}, \text{updated}\}$
ClState: $\text{ClLabel} \times \mathbb{N}$

```
mod! CLSTATE {
 pr(PAIR(NAT, CLLABEL{sort Elt -> ClLabel})*{
      sort Pair -> ClState, op fst -> fst.clstate,
      op snd -> snd.clstate })
}
```

Specification

ClLabel: $\{\text{idlecl}, \text{busy}\}$
PcLabel: $\{\text{idlepc}, \text{gotvalue}, \text{updated}\}$
ClState: $\text{ClLabel} \times \mathbb{N}$
PcState: $\text{PcLabel} \times \mathbb{N} \times \mathbb{N}$

```
mod! PCSTATE {
 pr(3TUPLE(NAT, NAT,
             PCLABEL{sort Elt -> PcLabel})*
                {sort 3Tuple -> PcState})
}
```

SPECIFICATION

ClLabel:	$\{\text{idlecl}, \text{busy}\}$
PcLabel:	$\{\text{idlepc}, \text{gotvalue}, \text{updated}\}$
ClState:	$\text{ClLabel} \times \mathbb{N}$
PcState:	$\text{PcLabel} \times \mathbb{N} \times \mathbb{N}$
PcStates:	$\text{MultiSet}(\text{PcState})$

```
mod! PCSTATES {
 pr(MULTISET(PCSTATE{sort Elt -> PcState})*
     {sort MultiSet -> PcStates})
}
```

Specification

ClLabel:	{idlecl, busy}
PcLabel:	{idlepc, gotvalue, updated}
ClState:	ClLabel $\times$ $\mathbb{N}$
PcState:	PcLabel $\times$ $\mathbb{N}$ $\times$ $\mathbb{N}$
PcStates:	MultiSet(PcState)
State:	ClState $\times$ PcStates

```
mod! STATE {
 pr(PAIR(CLSTATE{sort Elt -> ClState},PCSTATES
    {sort Elt -> PcStates})*{sort Pair -> State})
}
```

Transitions

GetValue: if PC and Cloud is idle, fetch Cloud value

Transitions

GetValue: if PC and Cloud is idle, fetch Cloud value

```
mod! GETVALUE { pr(STATE)
 trans[getvalue]:
   <
    < ClVal:Nat , idlecl > ,
    ( <<PcVal:Nat; OldClVal:Nat; idlepc>> S:PcStates)
   > =>
   <
    < ClVal , busy > ,
    ( <<PcVal; ClVal; gotvalue>> S)
   > .
}
```

Transitions

GetValue:	if PC and Cloud is idle, fetch Cloud value
Update:	update Cloud/PC according to larger value

Transitions

GetValue: if PC and Cloud is idle, fetch Cloud value
Update: update Cloud/PC according to larger value

```
mod! UPDATE { pr(STATE)
 trans[update]:
 <
  < ClVal:Nat , busy > ,
  (<<PcVal:Nat;GotClVal:Nat;gotvalue>> S:PcStates)
 > =>
 if PcVal <= GotClVal then
   < <ClVal,busy> ,(<<GotClVal;GotClVal;updated>> S)>
 else
   < <PcVal,busy> , (<< PcVal;PcVal;updated >> S) >
 fi .
}
```

Transitions

GetValue:	if PC and Cloud is idle, fetch Cloud value
Update:	update Cloud/PC according to larger value
GotoIdle:	both PC and Cloud go back to idle

Transitions

GetValue: if PC and Cloud is idle, fetch Cloud value
Update: update Cloud/PC according to larger value
GotoIdle: both PC and Cloud go back to idle

```
mod! GOTOIDLE {pr(STATE)
 trans[gotoidle]:
  <
   < ClVal:Nat ,busy > ,
   ( <<PcVal:Nat;OldClVal:Nat;updated >> S:PcStates)
  > =>
  < <ClVal, idlecl> , ( <<PcVal; OldClVal; idlepc>> S) > .
}
```

CloudSync

Final specification is combination of the three transitions
(included modules are shared!)

```
mod! CLOUD {
 pr(GETVALUE + UPDATE + GOTOIDLE)
}
```

CloudSync

Final specification is combination of the three transitions
(included modules are shared!)

```
mod! CLOUD {
 pr(GETVALUE + UPDATE + GOTOIDLE)
}
```

Goal

CloudSync

Final specification is combination of the three transitions (included modules are shared!)

```
mod! CLOUD {
 pr(GETVALUE + UPDATE + GOTOIDLE)
}
```

Goal

If PC is in updated state, then the values of the Cloud and the PC agree.

Verification

Hoare style proof

Verification

Hoare style proof

1) show invariant for all initial states

Verification

Hoare style proof

1) show invariant for all initial states
2) show that invariant is preserved over transitions

Verification

Hoare style proof

1) show invariant for all initial states
2) show that invariant is preserved over transitions

In details

- define a set of predicates

  $\text{initial} : \text{State} \mapsto \text{Bool}$

VERIFICATION

Hoare style proof

1) show invariant for all initial states
2) show that invariant is preserved over transitions

In details

- define a set of predicates

  initial : State $\mapsto$ Bool

- define a set of predicates

  invariant : State $\mapsto$ Bool

Verification

Hoare style proof

1) show invariant for all initial states
2) show that invariant is preserved over transitions

In details

- define a set of predicates

  $\text{initial} : \text{State} \mapsto \text{Bool}$

- define a set of predicates

  $\text{invariant} : \text{State} \mapsto \text{Bool}$

- show for all states

  $\forall S : \text{initial}(S) \to \text{invariant}(S)$

VERIFICATION

Hoare style proof

1) show invariant for all initial states
2) show that invariant is preserved over transitions

In details

- define a set of predicates

  $\text{initial} : \text{State} \mapsto \text{Bool}$

- define a set of predicates

  $\text{invariant} : \text{State} \mapsto \text{Bool}$

- show for all states

  $\forall S : \text{initial}(S) \rightarrow \text{invariant}(S)$

- show for all states

  $\forall S : \text{invariant}(S) \rightarrow \text{invariant}(S')$

where $S \mapsto S'$ is any transition

How to prove $\forall S$

Question

How to prove a statement like

$$\forall S : \text{initial}(S) \rightarrow \text{invariant}(S)$$

?

How to prove $\forall S$

Question

How to prove a statement like

$$\forall S : \text{initial}(S) \rightarrow \text{invariant}(S)$$

?

Answer

Show it for any element of a covering set of state expressions.

Covering set

most general: S (state variable) – every state is an instance of S

Covering set

most general: S (state variable) – every state is an instance of S

more general $\{S_1, \ldots, S_n\}$ such that

$$\forall S \exists S_i : S = \sigma(S_i)$$

i.e., every state term is an instance of one of the elements of the covering set

Proving with covering sets

Requirements for proving Hoare style

all transitions and predicates have to be *applicable* to terms of the covering set

Covering set

```
ops s1 s2 s3 s4 t1 t2 t3 t4 : -> State .
ops M N K : -> Nat . var PCS : PcStates .
eq s1 = < < N, idlecl > , ( << M; K; idlepc >> PCS ) > .
eq s2 = < < N, idlecl > , ( << M; K; gotvalue >> PCS ) > .
eq s3 = < < N, idlecl > , ( << M; K; updated >> PCS ) > .
eq t1 = < < N, busy > , ( << M; K; idlepc >> PCS ) > .
eq t2 = < < N, busy > , ( << M; K; gotvalue >> PCS ) > .
eq t3 = < < N, busy > , ( << M; K; updated >> PCS ) > .
```

Initial predicates

cl-is-idle: Cloud is initially idle

```
op cl-is-idle-name : -> PredName .
eq[cl-is-idle] : apply(cl-is-idle-name,S:State) =
              ( snd(fst(S)) = idlecl ) .
```

Initial predicates

cl-is-idle: Cloud is initially idle
pcs-are-idle: all PCs are initially idle

```
op pcs-are-idle-name : -> PredName .
eq[pcs-are-idle] : apply(pcs-are-idle-name,S:State) =
   zero-gotvalue(S) and zero-updated(S) .
```

Initial predicates

cl-is-idle: Cloud is initially idle
pcs-are-idle: all PCs are initially idle
init: cl-is-idle & pcs-are-idle

```
mod! INITIALSTATE {
 pr(INITPREDS)
 op init-name : -> PredNameSeq .
 eq init-name = cl-is-idle-name pcs-are-idle-name .
 pred init : State .
 eq init(S:State) = apply(init-name, S) .
}
```

Invariant predicates

goal: all PCs in updated state agree with Cloud

Invariant predicates

goal: all PCs in updated state agree with Cloud

if Cloud is idle then all PCs, too

only at most one PC is out of the idle state

all PCs in gotvalue state have their tmp value equal to the Cloud value

if Cloud is in busy state, then the value of the Cloud and the gotvalue of the Pcs agree

Hoare style in term reduction

initial step

```
red init(s1) implies invariant(s1) . -- OK
red init(s2) implies invariant(s2) . -- OK
red init(s3) implies invariant(s3) . -- OK
red init(t1) implies invariant(t1) . -- OK
red init(t2) implies invariant(t2) . -- OK
red init(t3) implies invariant(t3) . -- OK
```

Hoare style in term reduction

induction step search predicate

```
op inv-condition : State State -> Bool .
eq inv-condition(S, SS) =
  (not (
       S =(*,1)=>+ SS
       suchThat
       (not
         ((invariant(S) implies invariant(SS))
         == true)
       )
     )
  ) .
```

Hoare style in term reduction

induction step

```
red inv-condition(s1, SS) . -- OK
red inv-condition(s2, SS) . -- OK
red inv-condition(s3, SS) . -- OK
red inv-condition(t1, SS) . -- OK
--> The following condition does not reduce directly
--> to true, we will deal with it later on
red inv-condition(t2, SS) . -- BAD
red inv-condition(t3, SS) . -- OK
```

Hoare style in term reduction

induction step

```
red inv-condition(s1, SS) . -- OK
red inv-condition(s2, SS) . -- OK
red inv-condition(s3, SS) . -- OK
red inv-condition(t1, SS) . -- OK
--> The following condition does not reduce directly
--> to true, we will deal with it later on
red inv-condition(t2, SS) . -- BAD
red inv-condition(t3, SS) . -- OK
```

Rest of the invariant condition with case distinctions