Protocol State Machines

From useocl

The UML allows to define so-called Protocol State Machines (PSM) to define the correct usage of the operations defined by a classifier. Using PSMs, the valid call-sequences of operations can be specified using not only states and transitions, but also using guards and post-conditions.

Definition in a USE model

To define a PSM in a USE model, the USE grammar allows a **statemachines** section inside of a class definition:

```
classDefinition ::=
  ["abstract"] "class" id [specialization]
  ["attributes"] { attributeDefinition }
  ["operations"] { operationDefinition }
  ["constraints"] { invariantClause }
  ["statemachines"] { stateMachine }
```

The state machine definition starts with the keyqord **psm** to be able to support other kinds of state machines in the future. After the **psm** keyword, a name for the state machine must be given. Next, the sates and the transitions can be specified. For a valid state machine one pseude state of kind *inital* and at least one custom state must be defined. Further, the initial state must have one outgoing transition to another state.

```
stateMachine ::=
  "psm" id
  "states" { stateDefinition }
  "transitions" { transitionDefinition }
  "end"
```

For a *normal* state, i. e., a state which is not a pseudostate, just a name needs to be provided. Pseudo states can be defined by the kind of the pseude state after a colon following the name. Normal states can further define an invariant which is a boolean OCL expression. The owning object can be accessed inside of the state invariant using the keyword **self**.

```
stateDefintion ::=
  id [":"stateType]
  [ "[" stateInvariant "]" ]
```

```
stateInvariant ::= expression
```

Currently, USE supports two types of pseudo states:

```
stateType ::= "initial" | "final"
```

For a transition, the source and target states are "linked" by an arrow (->). Only for the transition leaving the initial state, no trigger needs to be defined. Instead of leaving the trigger out, it can also be names *create*. For all other transitions USE requires to specify an operation owned by the classifier containing the protocol state machine as a trigger. A single operation can be defined for more than one outgoing transition of a state. The decision which transition to take is then left to the guards (pre conditions) and to the post conditions.

```
transitionDefinition ::=
  id "->" id
  [ "{"
    [ "[" guard "]" ]
    ( event | operation )
    [ "[" postCondition "]" ]
  "}" ]
```

Example

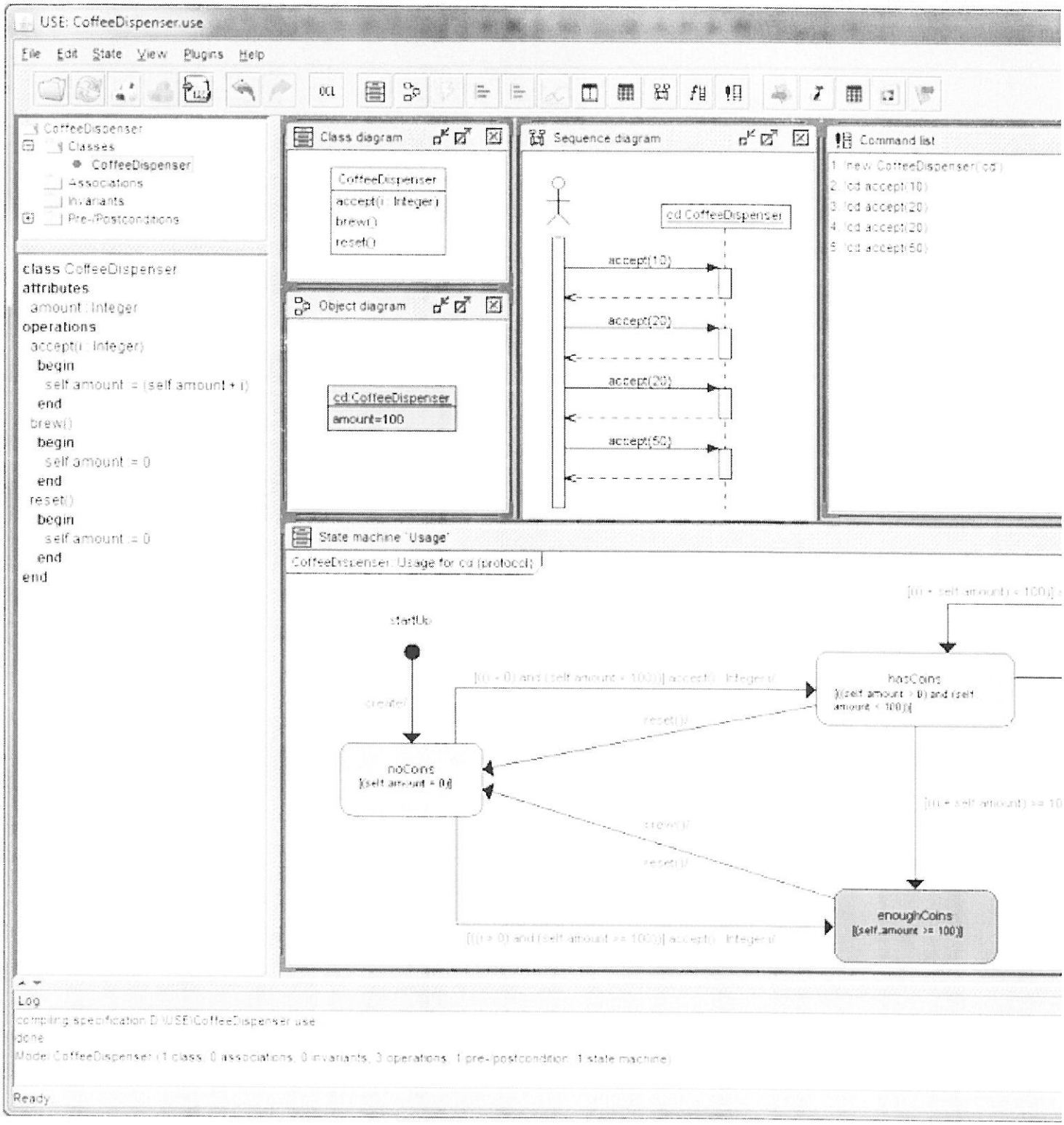
The following USE model defines a simple coffee dispenser that accepts coins and after retrieving enough coins, a coffee can be brewed. Any time until the machines brews, the user of the coffee dispenser can cancel its order and gets its money back.

```
model CoffeeDispenser
/*
 * A simple class for a coffee dispenser.
 */
class CoffeeDispenser
  attributes
    /* This attribute stores the money inserted into
       the dispenser.
       It is initialized with 0 during creation.*/
    amount : Integer init = 0

  operations
    /* A coin is inserted into the dispenser.
       Valid coins are: 10,20,50,100 and 200. */
    accept(i:Integer)
    begin
      self.amount := self.amount + i;
    end
    pre: let validCoins = Set{10,20,50,100,200} in validCoins->includes(i)

    /* A coffee should be brewed. Note, that there is
     * no pre condition. The state machine below handles
     * the correct amount.*/
    brew()
    begin
      self.amount := 0;
    end

    -- Return the coins currently in the dispenser
    reset()
    begin
      self.amount := 0;
    end
```

```
  statemachines
    /* This state machine describes the
     * "lifecycle" of the coffee dispenser.
     */
    psm Usage
      states
        -- The start node
        startUp:initial
        -- The initial state after creation.
        noCoins      [amount = 0]
        -- Some coins were inserted, but not enough.
        hasCoins     [amount > 0 and amount < 100]
        -- Enough coins, ready to brew.
        enoughCoins  [amount >= 100]
      transitions
        -- Define the first state after creation.
        startUp  -> noCoins     { create }
        -- A coin was inserted, but it wasn't enough.
        noCoins  -> hasCoins    { [i > 0 and amount < 100] accept() }
        -- Enough money was inserted.
        noCoins  -> enoughCoins { [i > 0 and amount >= 100] accept() }
        -- Abort
        hasCoins -> noCoins     { reset()  }
        -- Enough money was inserted.
        hasCoins -> enoughCoins { [i + amount >= 100] accept() }
        -- Another coin was inserted, but it wasn't enough.
        hasCoins -> hasCoins    { [i + amount < 100] accept() }
        -- Abort
        enoughCoins -> noCoins  { reset()  }
        -- Brew coffe, no change ;-)
        enoughCoins -> noCoins  { brew()   }
    end
end
```

The following screenshot shows the system state as it can be displayed in USE after executing the following commands:

```
!create cd:CoffeeDispenser
!cd.accept(10)
!cd.accept(20)
!cd.accept(20)
!cd.accept(50)
```



Coffee dispenser example in USE 3.1

Retrieved from "http://sourceforge.net/apps/mediawiki/useocl/index.php?title=Protocol_State_Machines"

- This page was last modified on 21 November 2013, at 13:01.