A Abstract State Machines and ASMETA

The *Abstract State Machine* (ASM) formal method [9,8] is a rigorous approach to software and system design, analysis, and verification. It provides a mathematical framework for modeling complex systems, capturing their behavior in a precise and abstract manner.

ASMs are based on the concept of state machines: each *state* represents a possible configuration of the system in terms of an algebraic structure (i.e., domains of objects with functions defined on them), and *transitions* between states are given by firing well-defined rules, i.e., rules constructed by a given and fixed number of *rule constructors*. At each computation step, all transition rules are executed in parallel by leading to simultaneous (consistent) updates of a number of locations – i.e., memory units defined as pairs (*function-name*, *list-of-parameter-values*)–, and therefore changing functions interpretation from one state to the next one. Location *updates* are given as assignments of the form $loc := v$, where *loc* is a location and v its new value. The rule constructors used for our modeling are those for guarded updates (`if-then`, `switch-case`), parallel updates (`par`), nondeterministic updates (`choose`), abbreviations (`let`).

Functions that are not updated by rule transitions are *static*. Those updated are *dynamic*, and distinguished in *monitored* (read by the machine and modified by the environment), and *controlled* (read and written by the machine).

An ASM *computation* (or *run*) is defined as a finite or infinite sequence $S_0, S_1, \ldots, S_n, \ldots$ of states of the machine, where S_0 is an initial state and each S_{n+1} is obtained from S_n by firing the unique *main rule* which in turn could fire other transitions rules.

ASMETA [7] is an Eclipse-based set of integrated tools tailored to the ASM formal method. It provides a textual user-friendly pseudo-code format to write ASM models, the `AsmetaL` notation, with an editor and a syntax checker. An ASMETA model has a well-formed structure as reported in Listing A.1 for the Auction smart contract specification: a model is composed by an `import` section that allows the import of external libraries and user-defined models that provide predefined domains, functions, and rules; a `signature` section to declare specific domains and functions; a `definitions` section to define specific static domains and functions, transition rules, state invariants, and properties to be verified (if any); it also encompasses the `main rule`, which is the starting point of the machine computation and that may trigger other transition rules; finally, the `init` section defines the values for the model's initial state.

A core part of the ASMETA toolset is the `AsmetaS` simulator to execute models. It offers a rich set of functionalities, including support for a wide range of validation tasks, from invariant checking, consistent update checking, and random simulation, to interactive simulation modes. For more powerful scenario-based validation, the `AsmetaV` tool uses the `AsmetaS` simulator along with the `AValLa` language to express scenarios as sequences of actor actions and expected machine reactions, enabling comprehensive validation of execution scenarios on ASM models. Formal verification of ASM models is handled by the `AsmetaSMV` tool, which translates ASM models into input for the NuSMV model checker.

```
asm Auction

import ../../lib/asmeta/StandardLibrary
import ../../lib/solidity/EVMLibrary

signature:
  dynamic controlled currentFrontrunner : User
  dynamic controlled currentBid : MoneyAmount
  dynamic controlled owner : User
  static auction : User
  static user2 : User
  static bid : Function
  static destroy : Function

definitions:
  // DESTROY FUNCTION RULE
  rule r_Destroy =
    if executing_function(current_layer) = destroy then
      switch instruction_pointer(current_layer)
        case 0 :
          r_Selfdestruct[owner]
      endswitch
    endif

  // BID FUNCTION RULE
  rule r_Bid =
    if executing_function(current_layer) = bid then
      switch instruction_pointer(current_layer)
        case 0 :
          r_Require[amount(current_layer) > currentBid]
        case 1 :
          if not isUndef(currentFrontrunner) then
            instruction_pointer(current_layer) := instruction_pointer(current_layer) + 1
          else
            instruction_pointer(current_layer) := 4
          endif
        case 2 :
          r_Transaction[auction, currentFrontrunner, currentBid, none]
        case 3 :
          r_Require[exception]
        case 4 :
          par
            currentFrontrunner := sender(current_layer)
            instruction_pointer(current_layer) := instruction_pointer(current_layer) + 1
          endpar
        case 5 :
          par
            currentBid := amount(current_layer)
            instruction_pointer(current_layer) := instruction_pointer(current_layer) + 1
          endpar
        case 6 :
          r_Ret[]
      endswitch
    endif

  // FALLBACK FUNCTION RULE
  rule r_Fallback =
    if executing_function(current_layer) != bid and executing_function(current_layer) != destroy then
      switch instruction_pointer(current_layer)
```

```
        case 0 :
          r_Require[false]
      endswitch
    endif

  // MAIN
  main rule r_Main =
    if current_layer = 0 then
      if not exception then
        let ($s = random_sender) in
          let ($r = random_receiver) in
            let ($n = random_amount) in
              let ($f = random_function) in
                if (not is_contract($s)) then
                  r_Transaction[$s, $r, $n, $f]
                else exception := true
                endif
              endlet
            endlet
          endlet
        endlet
      endif
    else
      if executing_contract(current_layer) = auction then
        par
          r_Destroy[]
          r_Bid[]
          r_Fallback[]
        endpar
      endif
    endif

default init s0:

  // LIBRARY FUNCTION INITIZLIZATIONS
  function executing_function ($sl in StackLayer) = none
  function executing_contract ($cl in StackLayer) = user
  function instruction_pointer ($sl in StackLayer) = 0
  function current_layer = 0
  function balance($c in User) = 3
  function destroyed($u in User) = false
  function payable($f in Function) =
    switch $f
      case destroy : false
      case bid : true
      otherwise false
    endswitch
  function exception = false
  function is_contract ($u in User) =
    switch $u
      case user : false
      case user2 : false
      otherwise true
    endswitch

  // MODEL FUNCTION INITIALIZATION
  function owner = user2
  function currentBid = 0
```

Listing A.1: Complete Auction Model

B List of Contracts invariants

Contracts		Invariants
Auction	A_1	The `destroy` function can only be called by the owner of the contract
	A_2	If a call is made to the `bid` function and a `current_frontrunner` already exists, the previously deposited money is returned to it
	A_3	If a call is made to the `bid` function with a `msg.value` greater than `current_bid` then the caller becomes the new `current_frontrunner`
	A_4	If a call is made to the `destroy` function, all the money in the contract goes to the owner
StateDao	B_1	If there was no exception and the contract is not running, the contract's state is INITIALSTATE
	B_2	If a call to `deposit` is made with a `msg.sender` value greater than 0 then it does not raise an exception
	B_3	An exception is not raised even if a call to `deposit` is made and the balance of `state_dao` is greater or equal than 12
	B_4	There is always at least one balance that is greater than the corresponding `customer_balance`
	B_5	If there was no exception and the contract is not running, the balance of `state_dao` is less than 12
Airdrop	C_1	Even if a call to `receive_airdrop` is made and no exceptions are raised, the value for `msg.sender` of `received_airdrop` remains false
	C_2	If a call to `receive_airdrop` is made from an account with `received_airdrop` set to 0, an exception is not raised
	C_3	Not all users received the airdrop
Crowdfund	D_1	If a call to `donate` is made, and no exceptions have been raised, then `donors(msg.sender)` is greater than 0
	D_2	Even if a call to donate is made and the donation phase is over, an exception is not raised
	D_3	If a call to `withdraw` completes without any exceptions being raised, then the sender was the owner of the contract
	D_4	After a call to `reclaim`, if no exceptions are raised, then the value of donors for the sender is 0
KotET	E_1	Every time a user becomes king it must be a different user from the previous king
	E_2	It is not possible for the balance of the contract to reach 0
	E_3	`claim_price` cannot be greater than all user balances
	E_4	If a call to the Kotet `fallback` is made with an amount greater than or equal to `claim_price` an exception is not raised
Baz	F_1	Not all the states are set to true

Table 3: The list of proposed invariants for each smart contract