Abstract Operational Semantics for Use Case Maps

Jameleddine Hassine[1], Juergen Rilling[1], and Rachida Dssouli[2]

[1] Department of Computer Science, Concordia University, Montreal, Canada
{j_hassin, rilling}@cs.concordia.ca
[2] Concordia Institute for Information Systems Engineering, Montreal, Canada
dssouli@ciise.concordia.ca

Abstract. Scenario-driven requirement specifications are widely used to capture and represent functional requirements. Use Case Maps (UCM) is being standardized as part of the User Requirements Notation (URN), the most recent addition to ITU−T's family of languages. UCM models allow the description of functional requirements and high-level designs at early stages of the development process. Recognizing the importance of having a well defined semantic, we propose, in this paper, a concise and rigorous formal semantics for Use Case Maps (UCM). The proposed formal semantics addresses UCM's operational semantics and covers the key language functional constructs. These semantics are defined in terms of Multi-Agent Abstract State Machines that describes how UCM specifications are executed and eliminates ambiguities hidden in the informal language definition. The resulting operational semantics are embedded in an ASM-UCM simulation engine and are expressed in AsmL, an advanced ASM-based executable specification language. The proposed ASM-UCM engine provides an environment for executing and simulating UCM specifications. We illustrate our approach using an example of a simplified call connection.

Keywords: Use Case Maps, user requirements notation, abstract state machines, formal semantics, simulation, AsmL.

1 Introduction

In the early stages of common development processes, system functionalities are defined in terms of informal requirements and visual descriptions. Scenario-driven approaches, although often semiformal, are widely accepted because of their intuitive syntax and semantics.(Amyot and Eberlein [4] provide an extensive survey of fifteen scenario notations)

Use Case Maps [12] is one of these scenario based languages that has gained momentum in recent years within the software requirements and specification community. Use Case Maps (UCMs) can be applied to capture and integrate functional requirements in terms of causal scenarios representing behavioral aspects at a higher level of abstraction, and to provide the stakeholders with guidance and reasoning about the system-wide functionalities and behavior. Use Case Maps [19] are part of a new proposal to ITU−T for a User Requirements Notation (URN) [18]. UCM notation has been baptized URN−FR, while another and complementary component for non-functional

F. Wang (Ed.): FORTE 2005, LNCS 3731, pp. 366–380, 2005.
© IFIP International Federation for Information Processing 2005

requirements is called URN−NFR. UCMs have been useful in a number of areas: Design and validation of telecommunication and distributed systems [2,3], detection and avoidance of undesirable feature interactions [13,21], evaluation of architectural alternatives [20], and performance evaluation [22].

However, the general lack of formalism and accuracy in requirement languages can cause ambiguities and misinterpretations of the specifications expressed by these languages and limit their use. This ambiguity can be removed by adding formal semantics to requirement specification languages. Moreover, this added formalism allows for a verification of specifications and their properties .

Currently, the UCM abstract syntax and static semantics are informally defined in an XML Document Type Definition [5]. However, to date the precise meaning of its execution semantics has not been captured.

In this paper, we present a formal operational semantics of UCM language in terms of *Abstract State Machines*(ASM) [16]. ASMs have been used to specify a wide variety of programming languages in particular C++ [24] and Java [10], logic programming languages such as Prolog [9] and its variants, and hardware languages such as VHDL [8]. ASMs have been also used to define the operational semantics of UML activity diagrams [7] and the formal definition of ITU−T standard SDL 2000 [15].

We tried to make this paper self-contained. In the next section, we provide an overview of the Use Case Maps notation. In section 3, we briefly introduce the basic concepts and notions of Abstract State Machines used in this paper. Section 4 gives the ASM models for Use Case Maps. In section 5, we provide an ASM-UCM engine, written in AsmL language [6], for simulating and executing UCM specifications. In section 6, we describe one possible scenario execution of the ASM model for the simplified call connection introduced in section 2.3. Finally, section 7 contains a brief discussion and a conclusion.

2 Use Case Maps Notation

2.1 Introduction

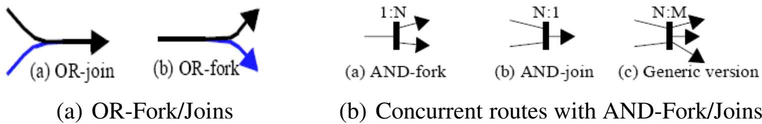
The Use Case Maps notation is a high level scenario based modeling technique used to specify functional requirements and high-level designs for various reactive and distributed systems. UCMs expressed by a simple visual notation allows for an abstract description of scenarios in terms of causal relationships between responsibilities (e.g. operation, action, task, function, etc.) along paths allocated to a set of components. These relationships are said to be causal because they involve concurrency, partial ordering of activities, and they link causes (e.g., preconditions and triggering events) to effects (e.g. postconditions and resulting events). With the UCM notation, scenarios are expressed above the level of messages exchanged between components, hence, they are not necessarily bound to a specific underlying structure (such UCMs are called *Unbound* UCMs). One of the strengths of UCMs is their ability to integrate a number of scenarios together (in a map-like diagram), and to reason about the architecture and its behavior over a set of scenarios. UCM specifications may be refined into more detailed models such as MSCs [20]. These detailed models may be transformed then into concrete implementations (possibly through automated code generation).

In the following section, we describe and illustrate the UCM core path notation. For a detailed description of the language, the reader is invited to consult [12] and [23].

2.2 UCM Functional Notation

A basic UCM path contains at least the following constructs: start points, responsibilities and end points. **Start points.** The execution of a scenario path begins at a start point. A start point is represented as a filled circle representing preconditions and/or triggering events. **Responsibilities.** Responsibilities are abstract activities that can be refined in terms of functions, tasks, procedures, events. Responsibilities are represented as crosses. **End points.** The execution of a path terminates at an end point. End points are represented as bars indicating post conditions and/or resulting effects.

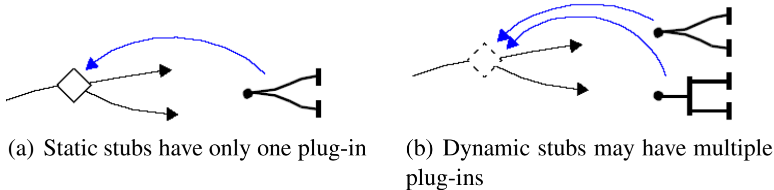
UCMs help in structuring and integrating scenarios in various ways− sequentially, as alternatives (with OR-forks/joins as illustrated in Figure 1(a)) or concurrently (with AND-forks/joins as illustrated in Figure 1(b)). **OR-Forks.** represent a path where scenarios split as two or more alternative paths. An OR-Fork has one incoming hyperedge and two or more outgoing ones. Conditions (Boolean expression called guard) can be attached to alternative paths. **OR-Joins.** capture the merging of two or more independent scenario paths. **AND-Forks.** split a single control into two or more concurrent control. **AND-Joins.** capture the synchronization of two or more concurrent scenario paths.



(a) OR-Fork/Joins (b) Concurrent routes with AND-Fork/Joins

Fig. 1. Structuring Scenarios

When maps become too complex to be represented as one single UCM, a mechanism for defining and structuring sub-maps becomes necessary. Path details can be hidden in sub-diagrams called plug-ins, contained in stubs (diamonds) on a path.



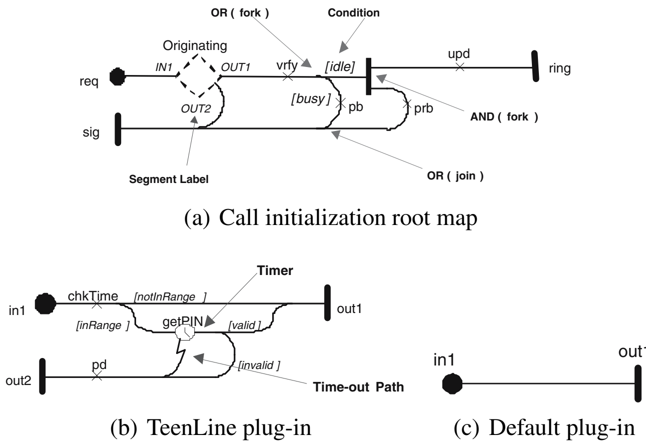
(a) Static stubs have only one plug-in (b) Dynamic stubs may have multiple plug-ins

Fig. 2. Stubs and plug-ins

2.3 Use Case Maps Example

Figure 3(a) shows a simplified call connection phase of a telephony system with one user-subscribed feature, *TeenLine* feature. This UCM is a modified version of the model originally introduced in [19]. The originating user can subscribe to the TeenLine feature which restricts outgoing calls based on the time of day (i.e., hours when homework should be the primary activity). This can be overridden on a per-call basis by anyone with the proper personal identification number. The causal path is initiated through the start point *req*. The dynamic stub *Originating* has two plug-ins:

- Default plug-in that represents how the basic call reacts in the absence of TeenLine feature (Figure 3(c)).
- TeenLine plug-in (Figure 3(b)) checks the current time (chkTime) and, if in the predefined range, requires a valid personal identification number (PIN) to be provided in a timely fashion for the call initiation to continue. If an invalid PIN is provided, or if a time-out occurs, then a denied reply is prepared (pd).



(a) Call initialization root map

(b) TeenLine plug-in (c) Default plug-in

Fig. 3. UCM example

The stub selection policy is based on the global variable *SubTL* which determines whether the originating user has subscribed to TeenLine feature:

- *SubTL=true* → TeenLine plug-in.
- *SubTL=false* → Default plug-in.

Other global variables of the specification include also *getPIN* and *valid*. The binding relationship connects the stub path segments of the parent map to the start/end points of the plug-in. In our example, the binding relationship for the TeenLine plug-in is : {<TeenLine, IN1, in1>, <TeenLine ,OUT1, out1>,<TeenLine, OUT2, out2>} connects the stub path segments of the parent map to the start/end points of the plug-in. If the call is allowed, the system then verifies whether the called party is busy or idle (vrfy). The idle path splits the control into two concurrent paths: Ringing (ring) and

signaling (sig) the occurrence of a prepared ringback reply (prb). In the case that the busy path is selected, it will result in the signaling of a prepared busy reply (pb).

3 Abstract State Machines

This section introduces some basic notions of ASM [16], that will be employed for the construction of our UCM model. For a rigorous mathematical definition of the semantic foundations of ASMs, we however refer to [11,16].

Abstract State Machines define a state-based computational model, where computations (runs) are finite or infinite sequences of states $\{S_i\}$ obtained from a given initial state S_0 by repeatedly executing transitions δ_i.

$$S_0 \xrightarrow{\delta_1} S_1 \xrightarrow{\delta_2} S_2 \quad \cdots \quad \xrightarrow{\delta_n} S_n$$

An ASM A is defined over a fixed vocabulary V, a finite collection of function names and relation names. Each function symbol has a fixed arity *n* and type $T_1,\ldots,T_n \rightarrow$ T where T_i and T are basic types. Names in V may be **(1) Static:** having the same (fixed) interpretation in each computation state of A **(2) Dynamic:** where function names can be altered by transitions fired in a computation step or, **(3) External:** its interpretation is determined by the environment (thus, not controlled by the system).

Given a vocabulary, A is defined by its program P and a set of distinguished initial states S_0. The program P consists of transition rules and specifies possible state transitions of A in terms of finite sets of local function *updates* on a given global state. Such transitions are atomic actions. A transition rule that describes the modification of the functions from one state to the next has the following form:
if Condition then <Updates> else <Updates> endif, where *Updates* is a set of function updates $f(t_1,t_2,\ldots,t_n) := t$ which are simultaneously executed when Condition is true. A state transition is performed by firing a set of rules in one step. Each function update changes a value at a specific location given by the left-hand-side of the update.

ASMs are multi-sorted based on the notion of universes. We presume the standard mathematic universes of Booleans, integers, lists, etc. as well as the standard operations on them such as the usual Boolean operations ($\wedge$, $\vee$ .etc.). A universe can be dynamically extended with individual objects by:
extend Universe with v <Rule> end extend, where *v* is a variable which is bound by the extend constructor.

The choose constructor defines an arbitrary selection of one element in a universe: *choose v in Universe <Rule> end choose*, where *v* is non-deterministically selected from the given universe. The choose constructor can be qualified by a condition.

A distributed ASM (called also *Multi-Agent ASM*) involves a collection of agents that perform their computation concurrently. The agents are elements of a dynamic universe AGENT that may grow and shrink over a run. Each agent $a \in$AGENT is viewed as an object of class AGENT and can identify itself by means of a special nullary function *me*:AGENT. The program of an agent *a* is a method of the class AGENT. The state of *a* (given by all fields of a) evolves in sequential steps with each invocation of its program. We assume that there is one program, prog, shared by all agents.

4 ASM Models for Use Case Maps

The definition of the ASM formal semantics of UCM consists of associating each UCM construct with an ASM which models its behavior. In this section, we associate an ASM signature to each UCM construct then we assign execution rules to them.

4.1 Signature of UCM Constructs

The UCM maps are modeled using the abstract sets: *StartPoint, EndPoint, Responsibility, AND-Fork, AND-Join, OR-Fork, OR-Join, Stub* and *Timer*. We define also the abstract set *HyperEdge* that represents the set of hyperedges connecting UCM constructs.

Start Points are of the form *StartPoint(PreCondition-set, TriggerringEvent-set, StartLabel, in, out)* where the parameter *PreConditions-set* is a list of conditions that must be satisfied in order for the scenario to be enabled (if no precondition is specified, then by default it is set to true). The parameter *TriggeringEvents-set* is a list that gives the set of events that can initiate the scenario along a path. One event is sufficient for triggering the scenario. The parameter *StartLabel* denotes the label of the start point. A start point should not have an incoming edge except when connected to an end point (called a waiting place). In such situation, we use the parameter *in* $\in$ *HyperEdge* to represent the connection with an end point. The parameter *out* $\in$ *HyperEdge* is the (unique) outgoing hyperedge.

End Points are of the form *EndPoint(PostCondition-set, ResultingEvent-set, EndLabel, in, out)* where the parameter *PostConditions-set* is a list of conditions that must be satisfied once the scenario is completed. The parameter *ResultingEvent-set* is a list that gives the set of events that result from the completion of the scenario path. The parameter *EndLabel* denotes the label of the end point; the parameter *in* $\in$ *HyperEdge* is the (unique) incoming hyperedge. End points have no target hyperedge except when connected to a start point (i.e. a waiting place). In such a case, *out* $\in$ *HyperEdge* represents such connection.

Responsibilities are of the form *Responsibility(in, Resp, out)* where *in* $\in$ *HyperEdge* is the incoming hyperedge, *Resp* is the responsibility to be executed (to be defined by a set of simultaneous ASM function updates), and *out* $\in$ *HyperEdge* is the outgoing hyperedge. A responsibility is connected to only one source hyperedge and to one target hyperedge.

OR-Forks are of the form *OR-Fork(in,* $[Cond_i]_{i\leq n}$, $[out_i]_{i\leq n}$*)* where *in* denotes the incoming hyperedge, $[Cond_i]_{i\leq n}$ is a finite sequence of Boolean expressions, and $[out_i]_{i\leq n}$ is a sequence of outgoing hyperedges.

OR-Joins are of the form *OR-Join(*$\{in_i\}_{i\leq n}$*, out)* where $\{in_i\}_{i\leq n}$ denotes the incoming hyperedges and, *out* is the outgoing hyperedge.

AND-Forks are of the form *AND-Fork(in,* $\{out_i\}_{i\leq n}$*)* where *in* denotes the incoming hyperedge, and $\{out_i\}_{i\leq n}$ is a sequence of outgoing hyperedges.

AND-Joins are of the form *AND-Join(*$\{in_i\}_{i\leq n}$*, out)* where $\{in_i\}_{i\leq n}$ denotes the incoming hyperedges, and *out* is the outgoing hyperedge.

Timers are of the form *Timer(in, TriggerringEvent-set, out, out_timeout)* where *in* denotes the incoming hyperedge. The parameter *TriggeringEvents-set* is the list that

gives the set of events that can trigger the continuation path (i.e. represented by *out*) and the parameter *out_timeout* $\in$ *HyperEdge* denotes the timeout path.

Stubs have the form *Stub(*$\{entry_i\}_{i\leq n}$, $\{exit_j\}_{j\leq m}$*, isDynamic,* $[Cond_k]_{k\leq l}$, $[plugin_k]_{k\leq l}$*)* where $\{entry_i\}_{i\leq n}$ and $\{exit_j\}_{j\leq m}$ denote respectively the set of the stub entry and exit points. *isDynamic* indicates whether the stub is dynamic or static. Dynamic stubs may contain multiple plug-ins, $[plugin_k]_{k\leq l}$ whose selection can be determined at run-time according to a selection-policy specified by the sequence of Boolean expressions $[Cond_k]_{k\leq l}$. The sequence Cond is empty for static stubs (i.e. *isDynamic=false*).

For each UCM construct we use a (static) function Param which, when applied to constructs yields the parameter. For example *in(StartPoint)* yields the incoming hyperedge of the construct StartPoint. We often suppress parameters notationally.

We formalize UCM maps by an abstract set MAPS. It contains the root map (i.e. the main UCM map) and all its submaps (i.e. plug-ins).

The nesting structure of a UCM specification is encoded in the following functions:

- UpMap: MAPS $\rightarrow$ MAPS $\cup$ $\{undef\}$, assigns to a plug-in its immediately enclosing map, if any. We assume that this function yields *undef* for the root map which is not enclosed in any map. Thus, *UpMap(rootMap)=undef.*
- StubBinding:$\{\{entry_i\}\cup\{EndPoints\}\}$ $\times$ MAPS $\rightarrow\{\{StartPoints\}\cup\{exit_j\}\}$ specifies how a plug-in $\in$MAPS is bound to a stub. The path segments that are connected to the stub need to be bound to the paths of the plug-ins in order to express continuity. This is done through explicit binding. An entry hyperedge joins a stub entry with a start point from the plug-in. An exit hyperedge joins a stub exit with an end point from the same plug-in.

In the following section, we define the ASM rules that define the operational semantics used to express the UCM control constructs.

4.2 ASM Rules of UCM Constructs

Let AGENT be the abstract set of agents *a* which move through their associated UCM map, by executing the UCM construct at the current active hyperedge, i.e. the hyperedge where the agent's control lies.

Every agent can mainly be characterized by three dynamic functions:

- active: AGENT$\rightarrow$HyperEdge represents the identifier of the active hyperedge leading to the next UCM construct to be executed.
- mode: AGENT$\rightarrow$ $\{running, inactive\}$. An agent may be running in normal mode or inactive once the agent has finished its computation.
- level: AGENT$\rightarrow$MAPS gives the submap that the agent is currently traversing.

For the root map, it is required that there is an agent for each starting point, in running mode with active hyperedges positioned on the corresponding start points of the root map (i.e. active=in(StartPoint)). The creation of the initial ASM agents, their initialization and the initialization of the global variables used in the scenario definitions represent the initialization phase.

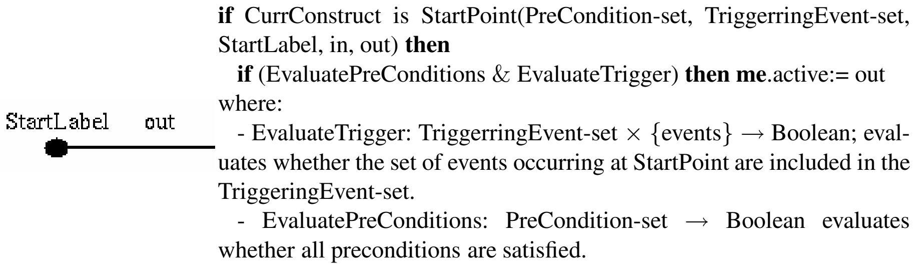


Fig. 4. Rule Start Point

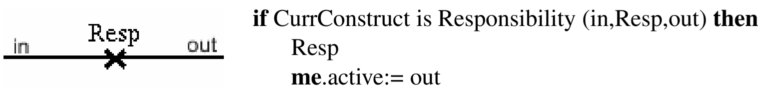


Fig. 5. Rule Responsibility

Typically, a running agent has to look at the target of its currently active hyperedge to determine the next action. CurrConstruct denotes the current UCM construct to be executed, i.e. the UCM construct where **me**.active=in(construct)∧**me**.mode=running.

In the following, we assign ASM execution rules to UCM constructs.

Start Points. If the control is on the hyperedge *in(StartPoint)*, the *PreCondition-set* is satisfied and there occurs at least one event from the *triggeringEvent-set*, then the start point is triggered and the control passes to the outgoing hyperedge of the StartPoint (Otherwise nothing happens and the control stays at the StartPoint). Figure 4 describes the start point rule.

Responsibilities. Responsibilites represent atomic actions, not to be decomposable, and their execution is not interruptible. If the control is on the hyperedge *in(Responsibility)* then *Resp* is performed and the control passes to the outgoing hyperedge.

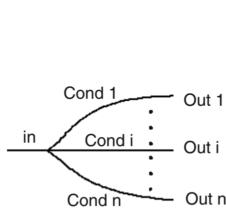
If the control is on the incoming hyperedge of an OR-Fork, the conditions are evaluated and the control passes to the hyperedge associated to the true condition. If more than one condition evaluates to true (i.e. nondeterministic choice), the control passes randomly to one of the outgoing hyperedges associated to the true conditions. Figure 6 illustrates the OR-Fork rule.

When one or many flows reach an OR-Join, the control passes to the outgoing hyperedge. Figure 7 illustrates the OR-Join rule.

Note:An UCM loop can be modeled as an OR-Fork followed by an OR-Join. Their respective rules should be executed once encountered.

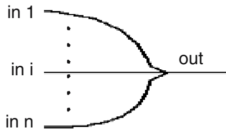
When the control is on an hyperedge entering an AND-Fork synchronization bar, then the flow is split into two or more flows of control. The currently running agent creates the necessary new subagents and sets their mode to running, then sets its mode to inactive. Each new ASM subagent inherits the program for executing UCMs, and its control is started on the associated outgoing hyperedge of the AND-Fork.

When many subagents running in parallel reach an AND-Join, their parallel flow must be joined. When all incoming hyperedges become active, a new agent is created



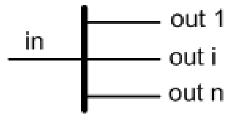
if CurrConstruct is OR-Fork(in, $[Cond_i]_{i \leq n}$,$[out_i]_{i \leq n}$)
then if NonDeterministicChoice($[Cond_i]_{i \leq n}$) **then**
me.active:= (**choose** out_i **in** $[out_k]_{k \leq l}$)
else **if** $Cond_1$ **then me**.active:=out_1
...
if $Cond_n$ **then me**.active:= out_n
where NonDeterministicChoice:{Cond}→Boolean is a dynamic function that checks whether more than one condition evaluates to true and $[out_k]_{k \leq l}$ is the sequence of hyperedges associated to satisfied conditions.

Fig. 6. Rule OR-Fork



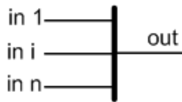
if CurrConstruct is OR-Join($\{in_i\}_{i \leq n}$, out) **then**
me.active:= out

Fig. 7. Rule OR-Join



if CurrConstruct is AND-Fork(in, $\{out_i\}_{i \leq n}$) **then**
me.mode:=inactive
extend AGENT with $a_1, \ldots, a_n$
do for alla_i, 1 $\leq$ i $\leq$ n
a_i.mode := running
a_i.active := out_i

Fig. 8. Rule AND-Fork



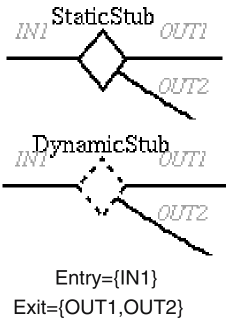
if CurrConstruct is AND-Join($\{in_i\}_{i \leq n}$, out)
then if not ($\forall a_1$,...,a_n $in_i = active(a_i)$) **then**
me.mode:= inactive
else **me**.mode:= inactive
extend AGENT with a_{n+1}
a_{n+1}.active:= out
a_{n+1}.mode:= running

Fig. 9. Rule AND-Join

and the control passes to the outgoing hyperedge. The last agent arriving to the AND-Join will fire the rule. Inactive agents are deleted after each rule's execution. For the clarity's sake, we have omitted the *Garbage Collection* from all our ASM rules.

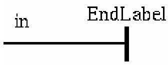
Once the control reaches a stub, the control passes to the selected plug-in and the execution continues following the UCM semantics. No extra agents are needed to execute a *Stub* unless the selected plug-in contains a concurrent flow.

When the control reaches an end point, two cases should be considered, depending on whether the end point is inside a plug-in or part of the root map:



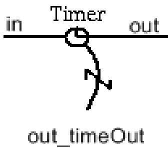
if CurrConstruct is Stub($\{entry_i\}_{i\leq n}$,$\{exit_j\}_{j\leq m}$, isDynamic, $[Cond_k]_{k\leq l}$,$[plugin_k]_{k\leq l}$) **then**
if not(isDynamic) **then** add(plugin, me.level) to MapHierarchy
me.level := plugin
me.active := in(StubBinding($entry_i$, plugin)
else add(plugin, **me**.level) to MapHierarchy
me.level := SelectionPolicy($(Cond_k)_{k\leq l}$))
me.active := in(StubBinding($entry_i$, SelectionPolicy($Cond_k)_{k\leq l}$))
Where SelectionPolicy:{Cond} $\rightarrow$ $MAPS$ is the selection policy function.

Fig. 10. Rule Stub



if CurrConstruct is EndPoint(PostCondition-set,ResultingEvent-set, EndLabel, in, out)**then if** UpMap(me.level)≠undef) **then**
me.active:= out(StubBinding(EndPoint, **me**.level))
elseif out≠undef **then me**.active := out
else me.mode:= inactive

Fig. 11. Rule End Point



if CurrConstruct is Timer(in, TriggerringEvent-set, out, out_timeout) **then**
if (Triggered) **then me**.active:= out
else me.active := out_timeout
where Triggered: TriggerringEvent-set→Boolean determines whether a trigger occurs within a predefined time frame.

Fig. 12. Rule Timer

1. End point is inside a plug-in: the control passes to the stub's exit point bound to the plug-in end point.
2. End point is part of the root map: the control passes to the out hyperedge if any (e.g. a waiting place) otherwise the running agent is stopped.

The exit from nested maps should be performed in the correct order of the stub structure. However, one control may exit the stub while another one is still inside the stub.

The timer rule is very similar to a basic OR-Fork rule with only two disjoint branches (out and out_timeOut).

5 ASM-UCM Simulation Engine

The ASM-UCM simulation engine is designed for simulating and executing UCM specifications. It is written in AsmL [6], a high level executable specification language developed by the Foundations of Software Engineering (FSE) group at Microsoft Research. AsmL is integrated with Microsoft's software development, documentation and runtime

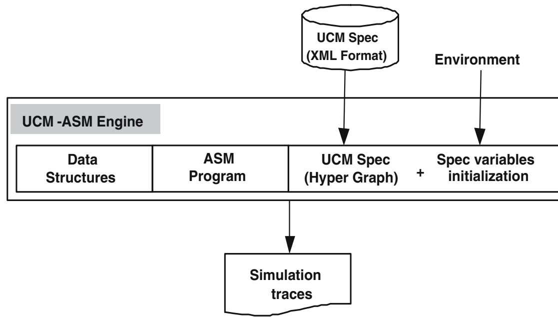


Fig. 13. ASM-UCM Simulation Engine Architecture

Enum Mode
  running
  inactive
structure UCMElement
  source as UCMConstruct
  hyper as HyperEdge
  target as UCMConstruct
type Maps = Set of UCMElement
structure UCMConstruc
case SP_Construct
  in_hy as HyperEdge
  out_hy as HyperEdge
  label as String
  preCondition as Boolean
	case R_Construct
	  in_hy as HyperEdge
	  out_hy as HyperEdge
	  label as String
	case OF_Construct
	  in_hy as HyperEdge
	  Selec as Set of OR_Selection
	case Stub_Construct
	  entry_hy as Set of HyperEdge
	  exit_hy as Set of HyperEdge
	  Selec_plugin as Set of Stub_Selection
	  Binding_Relation as Set of Stub_Binding
	  label as String
	case . . .

Fig. 14. Data Structures

environments including Visual Studio, Word and Component Object Model (COM). It has full .NET interoperability. Figure 13 shows the structure of the ASM-UCM simulation engine, which is composed of the following three components:

5.1 UCM Specification

In order to apply ASM rules defined in section 4, the UCM specification (originally described in XML format) should be translated into a hyper graph format where constructs are connected using hyperedges. For this purpose, we define a UCM specification as a hyper graph: SPEC = (C, H, λ) where:

- C is the set of UCM constructs composed of sets of typed constructs.
- H is the set of hyperedges
- λ is a transition relation (path connection) defined as: $\lambda = C \times H \times C$

Note: The translation from the XML format to hyper-graph format is done manually. Before a simulation can be run, the specification's global variables are initialized.

class Agent
const id as String
var active as HyperEdge
var mode as Mode
var level as Maps
 	
Program()
step
until me.mode = inactive do
choose h in level where HyperExists(active, GetInHyperEdge(h.source))
match (h.source)
// Rule of Start Point
SP_Construct (a,b,c,d): me.active := b
// Rule of Responsibility
R_Construct (a,b,c): Execute(h.source)
me.active := b
// Rule of OR-Fork ...
	main()
	step
	forall s in StartPoints
	let ag=new Agent(label(s), in(s), running, RootMap, init_stub)
	ag.Program()

Fig. 15. ASM-UCM program

5.2 Data Structures

The data structures maintained by the ASM-UCM engine are AsmL structures and dynamic sets. They encode the attribute information of UCM constructs and the structures that handle the dynamic flow of execution. The listing below shows part of the AsmL data structures used in ASM-UCM simulation engine. For instance, *Mode* is a static universe where each element is a static nullary function, *UCMElement* represents the structure of the transition relation λ, and *UCMConstruct* structure incorporates many case statements as a way of organizing different variant of UCM constructs.

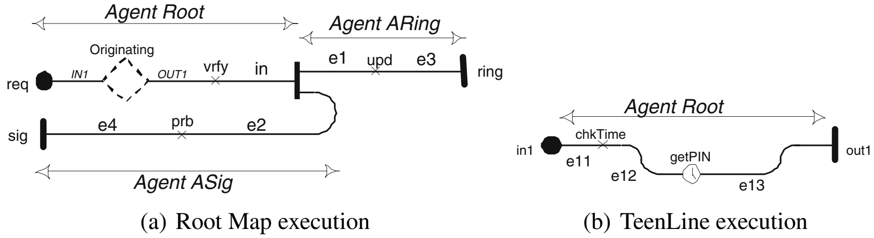
5.3 ASM Program

The listing below illustrates the class *Agent* and the main program of the ASM-UCM simulation engine.

6 ASM Execution of the Simplified Call Connection

In this section, we will describe one possible scenario execution of the ASM model for the simplified call connection introduced in section 2.3.

During the initialization phase the main agent *Root* is created and the global variables are initialized (i.e.,SubTL=true;InRange=true;getPIN=true;valid=true). The start point *req* rule is executed, and the control goes to the hyperedge *IN1*. The Stub rule is then executed. The function *SelectionPolicy* selects the plug-in *TeenLine* and the control passes to *in(StubBinding(IN1, SelectionPolicy(SubTL)))* which is the incoming hyperedge of the startpoint in_1. Responsibility *ChkTime* is observed, control passes to hyperedge *e12*, then the timer is triggered and, a valid PIN is entered. When the control



(a) Root Map execution (b) TeenLine execution

Fig. 16. Execution Trace

reaches the end point out_1, the EndPoint rule is executed, and the control passes to *out(StubBinding(EndPoint,level))* which is the hyperedge *OUT1*. Then the responsibility *vrf* is observed, and the control passes to the incoming hyperedge of the AND-Fork (i.e. hyperedge *in*). When the main agent *Root* reaches the AND-Fork, it creates two new agents *ARing* and *ASig* and changes it mode to *inactive*. These two agents start their execution respectively at the AND-Fork's upper and lower outgoing hyperedges (i.e. respectively hyperedges *e1* and *e2*). In our implementation, agents *ARing* and *ASign* evolve in an interleaving mode. Agent *ARing* executes responsibility *upd* and terminates while agent *ASig* executes *prb* then terminates. An ASM scheduler may be designed to have concurrent agents behave in true concurrency mode. Choosing the suitable concurrent execution semantics depends on the application domain and the design choices.

7 Discussion and Conclusion

In this paper, we have presented a formal operational semantics for Use Case Maps language based on Multi-Agent Abstract State Machines. Our ASM model provides a concise semantics of UCM functional constructs and describes precisely the control semantics.

Our approach based on ASM is more abstract and more flexible than the one given in [1] in terms of LOTOS [17]. Indeed, our ASM rules can be easily modified to accommodate language evolution. Considering new semantics for a UCM construct, result in changing the corresponding ASM rule without modifying the original specification. While in [1], one needs to redesign the mapping between UCM to LOTOS and to regenerate the LOTOS specification. Moreover, our ASM-UCM simulation engine may support different concurrency semantics at minimal cost. Agents may behave either in interleaving semantics with atomic actions (i.e. comparable to LOTOS processes) or in true concurrency mode. The choice of the suitable alternative depends on the application domain and the ASM program (i.e., ASM Scheduler) is designed accordingly.

We showed that ASMs are, in general, suitable to provide a formal representation of Use Case Maps constructs. The proposed semantics can be seen as a complementary, unambiguous documentation approach that provides additional insights of the UCM language and its notation, as well as a basis for future formal verification of UCM. As part of our future work, we will investigate the use of ASM model checking technique [14] to verify UCM specifications.

References

1. Amyot D., Formalization of Timethreads Using LOTOS. Master Thesis, Department of Computer Science, University of Ottawa, Canada, 1994.
2. Amyot D. and Andrade R., Description of wireless intelligent network services with Use Case Maps, SBRC'99, 17th Simpósio Brasileiro de Redes de Computadores, Salvador, Brazil, May 1999, pp. 418-433.
3. Amyot D., Buhr R.J.A., Gray T. and Logrippo L., Use Case Maps for the Capture and Validation of Distributed Systems Requirements. RE'99, Fourth IEEE International Symposium on Requirements Engineering, Limerick, Ireland, June 1999,44-53. http://www.UseCaseMaps.org/pub/re99.pdf
4. Amyot D. and Eberlein A., An Evaluation of Scenario Notations and Construction Approaches for Telecommunication Systems Development. In: Telecommunications Systems Journal, 24:1, 61-94, September 2003.
5. Amyot D. and Miga, A., Use Case Maps Document Type Definition 0.19. Working document, June 2000. http://www.UseCaseMaps.org/xml/
6. AsmL for Microsoft.Net, http://www.research.microsoft.com/foundations/asml, 2003
7. Börger E., Cavarra A. and Riccobene E., An ASM Semantics for UML Activity Diagrams. In T. Rus, editor, Proc. Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, LNCS 1826. Springer, 2000.
8. Börger E., Glässer U. and Muller W., Formal Definition of an Abstract VHDL'93 Simulator By EA-Machines. In C. Delgado Kloos and P. T. Breuer, eds., Formal Semantics for VHDL, 107-139. Kluwer Academic Publishers, 1995.
9. Börger E. and Rosenzweig D., A mathematical definition of full Prolog. In Science of Computer Programming, vol. 24, 249-286. North-Holland, 1994.
10. Börger E. and Schulte W., Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In L. Brim, J. Gruska, and J. Zlatuska, editors, Mathematical Foundations of Computer Science, MFCS 98, Lecture Notes in Computer Science. Springer, 1998.
11. Börger E. and Stärk R., Abstract State Machines: A Method for High-Level System Design and Analysis. Springer-Verlag, 2003.
12. Buhr, R. J. A., Use Case Maps as Architectural Entities for Complex Systems. In: IEEE Transactions on Software Engineering, 24(12) (Dec. 1998) 1131-1155.
13. Buhr R. J. A., Elammari M., Gray T. and Mankovski S., Applying Use Case Maps to multiagent systems: A feature interaction example. In 31st Annual Hawaii International Conference on System Sciences, 1998.
14. Del Castillo G. and Winter K., Model checking support for the ASM high-level language. In S. Graf and M. Schwartzbach, editors, 6th International Conference for Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000), volume 1785 of LNCS, pp 331-346. Springer-Verlag, 2000.
15. Eschbach R., Glässer U., Gotzhein R., von Löwis M. and Prinz A., Formal Definition of SDL−2000 - Compiling and Running SDL Specifications as ASM Models. In Journal of Universal Computer Science, 7 (11): 1025-1050, Springer Pub. Co., Nov. 2001.
16. Gurevich Y., Evolving algebra 1993: Lipari guide. In E. Börger, editor, Specification and Validation Methods. Oxford University Press, Oxford, 1995.
17. ISO, Information Processing Systems, OSI: LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. IS 8807, Geneva, 1989.
18. ITU-T, Recommendation Z.150, User Requirements Notation (URN)- Language Requirements and Framework, Geneva, Switzerland. http://www.UseCaseMaps.org/urn/
19. ITU-T, URN Focus Group (2002), Draft Rec. Z.152 - UCM: Use Case Map Notation (UCM). Geneva.

20. Miga A., Amyot D., Bordeleau F., Cameron C. and Woodside M., Deriving Message Sequence Charts from Use Case Maps Scenario Specifications. Tenth SDL Forum (SDL'01), Copenhagen, 2001. LNCS 2078, 268-287.
21. Nakamura N., Kikuno T., Hassine J., and Logrippo L., Feature Interaction Filtering with Use Case Maps at Requirements Stage. In: Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'00), Glasgow, Scotland, UK, May 2000.
22. Petriu. D. C. and Woodside M., Software Performance Models from System Scenarios in Use Case Maps, Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools, p.141-158, April 14-17, 2002.
23. Use Case Maps Web Page and UCM Users Group, 1999. http://www.UseCaseMaps.org
24. Wallace C., The Semantics of the C++ Programming Language. In E. Börger, editor, Specification and Validation Methods. Oxford University Press, 1995.