An ASM Operational Semantics for Use Case Maps

Jameleddine Hassine, Juergen Rilling
Department of Computer Science
Concordia University
Montreal, QC, Canada, H3G 1M8
{j_hassin, rilling}@cs.concordia.ca

Rachida Dssouli
Concordia Institute for Information Systems Engineering
Concordia University
Montreal, QC, Canada, H3G 1M8
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 work, a concise and rigorous formal semantics for Use Case Maps, defined in terms of Multi-Agent Abstract State Machines. The proposed formal semantics addresses UCM's operational semantics and provides a sound basis for executing UCM specifications using simulation tools and supporting formal verification.

1 Introduction

Use Case Maps [3] is a scenario based language 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. Use Case Maps are part of a new proposal to ITU−T for a User Requirements Notation (URN) [3].

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.

Currently, the UCM abstract syntax and static semantics are informally defined in an XML Document Type Definition [3]. However, to date the precise meaning of its execution semantics has not been captured. In this work, we present a formal operational semantics of UCM in terms of *Abstract State Machines*(ASM) [2].

2 Abstract State Machines

Abstract State Machines [1, 2] 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:

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

ASMs states are multi-sorted first-order structures, i.e. sets with relations and functions. The transition relation is specified by guarded function updates, called rules, describing the modification of the functions from one state to the next:
if Condition then <Updates> else <Updates> endif, where *Updates* is a set of function updates *f*(t_1,t_2,...,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.

Multi-Agent ASM [1] are used to model concurrent systems. The system consists of many agents of the abstract type AGENT. Each agent a ∈ AGENT executes its own program *prog(a)* and can identify itself by means of a special nullary function *me*.

3 ASM models for Use Case Maps

Defining the ASM formal semantics for Use Case Maps consists of associating a signature and an ASM execution rule to each UCM construct. We formalize UCM maps by an abstract set MAPS, which contains the main UCM map and all its plug-ins (i.e. submaps). The nesting structure of a UCM specification is encoded in the following functions:
- UpMap: assigns to a plug-in its immediately enclosing map, if any.
- StubBinding: specifies how a plug-in is bound to a stub. 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.

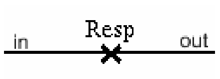
It is required that we create an agent for each start point. A running agent has to look at the target of its currently active hyperedge to determine the next construct to be executed, denoted by CurrConstruct. An agent can mainly be characterized by three functions:

- active: AGENT→HyperEdge represents the active hyperedge.
- mode: AGENT→ $\{running, inactive\}$. An agent may be running or inactive once the agent has finished its computation.
- level: AGENT→MAPS gives the submap that the agent is currently traversing.

In the following, we provide a signatures and execution rules to UCM constructs.

StartPoint(in, PreConditions, TrigEvent-list, Label, out). If the control is on the start point, the preconditions are satisfied and there occurs at least one event from the *trigEvent-list*, then the start point is triggered and the control passes to the outgoing hyperedge *out*.

Responsibility(in, Resp, out). If the control is on the hyperedge *in* then *Resp* is performed and the control passes to the outgoing hyperedge. Figure 1 illustrates the Responsibility rule.



```
if (CurrConstruct = Responsibility(in, Resp, out)) then Resp
        me.active:= out
```

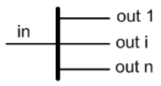
Figure 1. Rule Responsibility

OR-Fork(in, $\{Cond_i\}_{i\leq n}$, $\{out_i\}_{i\leq n}$). If the control is on the incoming hyperedge, the conditions are evaluated and the control passes to the hyperedge associated to the true condition. If more than one condition evaluates to true the control passes randomly to one of the outgoing hyperedges with true conditions.

OR-Join($\{in_i\}_{i\leq n}$, out). When one or many flows reach an OR-Join, the control passes to the outgoing hyperedge *out*.

AND-Fork(in, $\{out_i\}_{i\leq n}$). When the control is on hyperedge *in* then the flow is split into *i* concurrent flows. The currently running agent creates the necessary new agents and sets their mode to running. Created agents inherit the program for executing UCMs, and their execution is started on the associated outgoing hyperedges $\{out_i\}_{i\leq n}$. Figure 2 illustrates the AND-Fork 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.

AND-Join($\{in_i\}_{i\leq n}$, out). When many agents running in parallel reach an AND-Join, their parallel flow must be joined. If all the incoming hyperedges are active, then a new agent is created and control passes to the outgoing hyperedge *out*.



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

Figure 2. Rule AND-Fork

Stub($\{entry_i\}_{i\leq n}$, $\{exit_j\}_{j\leq m}$, isDynamic, $\{Cond_k\}_{k\leq l}$, $\{plugin_k\}_{k\leq l}$). Once the control reaches a stub, the control passes to the start point of the selected plug-in and the execution continues inside the plug-in.

EndPoint(PostConditions, ResultingEvent-list, Label, in, out). We consider two cases:

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

4 Conclusion

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 and flexible semantics of UCM functional constructs and describes precisely the control semantics. 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.

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.

References

[1] Börger E. and Stärk R., Abstract State Machines: A Method for High-Level System Design and Analysis. Springer-Verlag, 2003.

[2] Gurevich Y., Evolving algebra 1993: Lipari guide. In E. Börger, editor, Specification and Validation Methods. Oxford University Press, Oxford, 1995.

[3] ITU-T, URN Focus Group (2002), Draft Rec. Z.152 - UCM: Use Case Map Notation (UCM). Geneva.