A Semantic Model for the State Machine in the Unified Modeling Language

Kevin Compton[1], James Huggins[3], and Wuwei Shen[1]*

[1] EECS Department, University of Michigan
1301 Beal Avenue, Ann Arbor, MI 48109-2122 USA
kjc, wwshen@eecs.umich.edu

[2] Computer Science Program, Kettering University
1700 W. Third Avenue, Flint, MI 48504-4898 USA
jhuggins@kettering.edu

1 Introduction

The Unified Modeling Language (UML) is becoming a standardized modeling notation for expressing object-oriented models and designs. UML is based on an intuitive and easy to understand diagrammatic notation. Since its birth, a lot of CASE tools for UML have been generated; but the specification of the language is "semi-formal", i.e. parts of it are specified with well-defined language while other parts have been described informally in English.

It is commonly admitted that a language should have a formal semantics to be unambiguous. Furthermore, the semantics must be precise if tools are to verify some properties on a model expressed in this language. A lot of methods have been presented to represent the semantics for the state machine in UML. Here we are looking for a method which can not only represent the semantics for the state machine but also execute and verify the semantic model.

There has been other work on verification tool for UML. The tool `vUML` [8], developed in Abo Akademi University in Finland, is used to verify a UML model by using the model checker SPIN. `vUML` translates the UML model into the SPIN input language PROMELA and then verifies the UML model. But the SPIN input language PROMELA is not a formal language which can be applied to give a semantic model. Therefore `vUML` deploys structural operation semantics to give the semantic model. This seems to us that it separates the verification model and semantic model. Fortunately Abstract State Machines can avoid this separation by unifying them into one model.

2 Abstract State Machine

Abstract State Machines (abbreviated as ASM) [1] were first presented by Prof. Y. Gurevich ten years ago. Since then, they have been successfully used in specifying and verifying a lot of software systems. For example, they have been used to give a semantic model for several different styles of programming languages, like C [7] etc.. We can use an ASM interpretor to execute these semantic model [2]. Additionally, Abstract State Machines are used to specify

* Partially supported by NSF grant CCR 95-04375.

and verify a lot of software application (e.g. [3]). As the first step of an ongoing project, we present an Abstract State Machine Model for the UML state machine in this paper.

An ASM program consists of the rules of the following forms:[1]

1. Update Rule: $f(\bar{s}) := t$ is a rule with function f. Here $\bar{s}$ is a tuple $(s_1, \ldots, s_n)$ of terms. The meaning of this rule is that the value of the function f at the value of the tuple $(s_1, \ldots, s_n)$ is set to the value of t.
2. Conditional Rule: If g is a boolean term and R_1 and R_2 are rules then **if** g **then** R_1 **else** R_2 **endif** is a rule. The meaning of this rule is that if the value of g is true then R_1 is fired; otherwise R_2 is fired.
3. Block: If R_1 and R_2 are rules then **do in-parallel** $R_1 R_2$ **enddo** is a rule with component R_1 and R_2. R_1 and R_2 are fired simultaneously if they are mutually consistent; otherwise do nothing. In a general ASM program like what we will give the rules in the following, we always omit the keyword **do in-parallel** and **enddo** .

In order to model a composite state in the UML state machine, we use a set of agents, each executing a set of ASM rules. To distinguish itself from the other agents, an agent `a` can interpret `Self` as `a`. More details about the distributed ASMs can be found in [6].

3 An ASM Model

Now we outline the idea about how to give an ASM model for a UML state machine diagram. The UML state machine we consider is based on [10]. All ASM rules given are general purpose. More details about this can be found in [5]. Because the interrupt caused by an event implicitly affects some nodes in the UML state machine, we extend it to an extended UML state machine diagram. In an extended UML state machine diagram, when an event occurs we just need to consider how to interrupt the activity associated with a node affected by that event; and we don't need to think about the other nodes. After obtaining an extended UML state machine diagram, we give a set of transition rules for all kinds of states in the extended UML state machine diagram. Next we consider how to derive an extended UML state machine diagram.

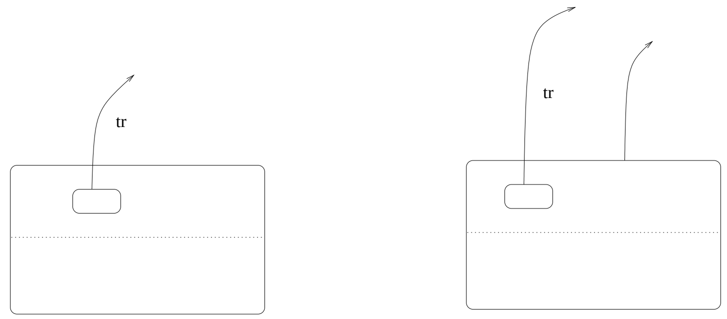
First we consider those transitions outgoing from a node. [2] In a UML state machine if an event associated with a transition occurs, then not only is the source node of that transition affected but those nodes, which are either subnodes of that transition's source node or the ones including the transition's source node, are affected as well. To explicitly denote these nodes affected by that transition, we add a transition for every affected node which is a source node of the new generated transition. In Figures 1 and 2 readers find how these new transitions are added.

On the other hand, we also add some transitions in an extended UML state machine diagram for those transitions coming into a composite node. The purpose for this is to explicitly give the target node for every transition coming into a composite node.

---

[1] In fact, an ASM program can be defined in a more liberal way. Some structures introduced like `let` rules make ASM program more easy to write and read and they can be translated to the basic rules.

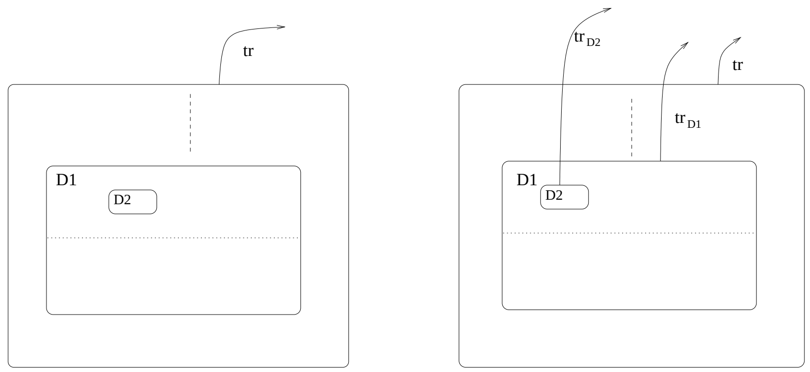
[2] Strictly speaking, state and transition are used in a UML state machine. And node and arc are used in an extended UML state machine diagram. But sometimes we use these names interchangeably.



(a) UML state machine diagram

(b) extended UML state machine diagram

Fig. 1. An arc is generated in (b) according to the arc tr in (a).



(a) UML state machine diagram

(b) extended UML state machine diagram

Fig. 2. Some arcs are generated in (b) according to the arc tr in (a).

If a transition enters a concurrent composite node and does not explicitly terminates on some of its regions, then we add some new transitions on these implicit regions, which is shown in Figure 3.

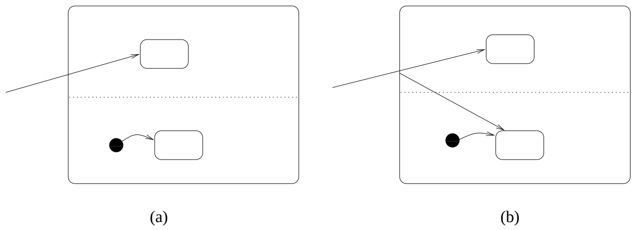


Fig. 3. (a) shows an transition in a state machine diagram terminating on a substate within a concurrent composite state. (b) shows a new arc is generated in the other region.

Having derived an extended UML state machine diagram, we consider how to give a dynamic model for a state machine in UML. To model a dynamic execution of a state machine in UML, two special function $CurArc$ and $CurNode$ are defined to denote the current active transition and node. The current node refers to a node whose actions are being executed. And the incoming transition is called an active transition.

Here we illustrate a simple node as an example to show that an ASM model for a UML state machine is correct. More details about an ASM model for a UML state machine can be found in [10]. The execution for a state is divided into three steps, one of which executes the entry actions, internal actions and exit actions respectively. If the guard condition associated with the current arc $CurArc$ is true, the node is a leaf node and the node's phase is *init*, then the entry action associated with the node is executed and at the same time, the node enters the phase *internal_exe*. The macro ISREACHABLE is used to check whether the guard condition associated with the current arc $CurArc$ is true or not. The macro `EXE_ENTRY_ACTION` is used to execute the entry actions. The ASM for this phase is shown in Fig 4. In order to save space we use `let` structure to represent that all appearances of `node` are replaced by `targetstate(CurArc)` in the ASM specification.

let *node =targetstate(CurArc)* **in**
    if *ISREACHABLE(node,CurArc)=true* **and** *IsSimple(node)=true* **and**
                Phase(node)=init **then**
        EXE_ENTRY_ACTION(node);
        CurNode(Self) := node;
        Phase = internal_exe;
    endif
endlet

Fig. 4. The initial phase for a simple state.

When an agent's control reaches a node and the node's phase is *internal_exe*, the internal actions and activity associated with the node are executed, which is denoted by macro `CREATE_ACT`. The ASM model for this phase is shown in Fig 5.

let *node = targetstate(CurArc)* **in**
  if *ISREACHABLE(CurArc(Self))=true* **and** *IsSimple(node)=true*
      and *Phase = internal_exe* **then**
    CREATE_ACT(node);
    Phase(node) := wait_for_exit;
  endif
endlet

Fig. 5. The internal phase for a simple state.

If a simple node is in state wait_for_exit and an arc, which results in a normal exit from the current node, is eligible to fire, the agent will execute the exit actions and entry actions along the arc. If there is more than one completion arc which is eligible to fire, a function $ChooseArc$ returns a highest priority arc, that is about to execute. The macro `ELIGIBLE2EXE` is used to denote whether a completion event is eligible to fire or not. The following are the definitions for the function $ChooseArc$ and macro `ELIGIBLE2EXE`.

ELIGIBLE2EXE(arc)=
IsTriggerless(arc)=true **and** *HasGenEvent(arc)=true* **and** *guard(arc)=true*

ChooseArc(node)= k *iff*
  ELIGIBLE2EXE$(outArc_k(node))$*=true* **and**
 ($\forall$ *i: ELIGIBLE2EXE*$(outArc_i(node))$*=true* $\Rightarrow$
  HighPriority$(ourArc_k(node),ourArc_i(node))$*=true*)

Fig. 6. The definitions for function ChooseArc and macro ELIGIBLE2EXE.

In order to execute the exit actions along the arc, we need to set some functions and the phase for the agent is also set to *arc*. All these actions are defined in the macro `EXIT_FROM_NODE`. And function $CurNode$ is set to $undef$ in that the agent is about to leave from that node. The ASM model for the last phase is shown in Fig 7.

The other interesting rule related to the UML state machine is how to implement the transition from one state into anther state. There are two ways to exit from one state into another: completion event and interruption event. If one of them is eligible to fire, then an agent needs to stop all the activities being executed and start executing all exit actions in the nodes from which the arc comes.

If a node, whose exit action is to be executed next, is a concurrent node, then we need to distinguish two cases. One is the execution for the exit action caused by an event belonging

```
let node = targetstate(CurArc) in
    if ISREACHABLE(node,CurArc) and IsSimple(node)=true and Phase = waiting_for_exit then
      let j=ChooseArc(node) in
        if (j != undef) then
          CurNode(Self) := undef;
          EXIT_FROM_NODE(outArc_j,node);
        endif
      endlet
    endif
endlet
```

Fig. 7. The last Phase for a simple state.

to that concurrent node and that concurrent node's exit action is to be first executed during the (abnormal) exit. The other is the execution for the exit action caused by the pass from its immediate subnode.

If the first case occurs, the agent needs to wait for all its child agents' executions of their exit actions to finish. If all the child agents finish their exit executions, then the agent kills all of its child agents (if they exist) and executes the action defined in EXE_EXIT_OUTWARDS. If the second case occurs, the agent stops the execution for exit action and waits for its parent agent to kill itself. When an agent finishes the execution for all the exit actions associated with an arc, we set function *CurMode* back to *node*, meaning a target node of the arc is a candidate to be executed in the next. The ASM specification is shown in Fig 8.

4 Conclusion

In this paper we outlined the ASM model for a UML state machine diagram. ASM specification uses simple syntax and its specification can be executed and verified as well.

References

1. Abstract State Machine Homepage: http://www.eecs.umich.edu/gasm.
2. Matthias Anlauff, XASM- An Extensible, Component-Based Abstract State Machines Language, Proceeding of Abstract State Machine Workshop, 2000.
3. Egon Börger. Why Use Evolving Algebras for Hardware and Software Engineering?", in M. Bartosek, J. Staudek, J. Wiedermann, eds., SOFSEM '95: Theory and Practice of Informatics, Springer Lecture Notes in Computer Science 1012, 1995, 236–271.
4. Egon Börger, A. Cavarra, and E. Riccobene. "An ASM Semantics for UML Activity Diagrams", in Teodor Rus, ed., Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000, Proceedings, Springer LNCS 1816, 2000, 293–308.
5. K. Compton, Y. Gurevich, J. Huggins, W. Shen. An Automatic Verification Tool for UML, Technical report, CSE-TR-423-00, University of Michigan, 2000.
6. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E.Börger, editor, Specification and Validation Methods, pages 9-36. Oxford University Press, 1995.

```
if CurMode(Self)=interrupt or CurMode(Self)=arc then
    if (head(NestStructure(Self)=undef) then
        CurMode(Self):= node;
        Action2ASM(action(CurArc(Self));
    else
        if (IsCompConcur(head(NestStructure(Self)))!=true) then
            EXE_EXIT_OUTWARDS(NestStructure(Self));
        else
            if CurNode!=head(NestStructure(Self)) then
                NestStructure(Self):=undef;    # deals with the second case
                Mod(Agent2Act(Self,head(NestStructure(Self))):=undef;
                CurMode(Self):=suspended;
            else    # deal with the first case
                if ∀a ∈ ChildAgent(Self): CurMode(a)=suspended then
                    EXE_EXIT_OUTWARDS(NestStructure(Self));
                    ∀c ∈ ChildAgent(Self): KILL(c);
                endif
            endif
        endif
    endif
endif
```

Fig. 8. The model for executing an arc.

7. James K. Huggins and Wuwei Shen, "The Static and Dynamic Semantics of C: Preliminary Version", Technical Report CPSC-1999-1, Computer Science Program, Kettering University, February 1999.
8. Johan Lilius and Ivn Porres Paltor, vUML: a Tool for Verifyng UML Models, TUCS Technical Report No. 272, May 1999.
9. J. Lilius, I. P. Paltor. Formalizing UML state machines for model checking, TUCS Technical Report No. 273, June 1999.
10. Rational Software Corporation, Unified Modeling Language (UML), version 1.3, http://www.rational.com, 1999.