An Automatic Verification Tool for UML

Kevin Compton, Yuri Gurevich
James Huggins, Wuwei Shen

Feb. 25, 2000

Abstract

The Unified Modeling Language is becoming more and more popular in the software development. However because of its ambiguisity in its semantic model, few verification tool has been built. Abstract State Machines have been successfully applied in giving semantics for programming language like C. In this report, we try to use the Abstract State Machines to give a semantics model for UML and then use ASM Model Checker to design a verification tool for UML. Last we give a toy example to show how the verification tool works.

1 Software Crisis and Software Model

In the past thirty years, the computer industry has undergone dramatic changes that no other industry has experienced. With the surprising drops in hardware cost, software costs have shown an incredible growth in the computer industry. Although a lot of money has been invested in software development, a lot of software products still can not be used. In order to overcome these problems, people look for some methodologies and the waterfall model is one of them.

1.1 Waterfall Model

People are developing disciplines by which software development can pass through a series of stages. One of the disciplines is the "waterfall model". The traditional waterfall model divides software development into the following stages: requirements, design, coding, testing, and operations.

In the requirement stage, requirements for a software system to be developed are given. This stage includes analyzing the software problem and gives a complete specification of the desired software system. In the next stage a developer decomposes the software system into its actual constituent components, generating modules with its input, output and functions.

Then software development enters the third stage: coding. This stage transforms the modules defined during the design stage into a computer-understandable language. After then, the testing stage is entered; testers test the system according to the software requirements to uncover and remove "bugs". If there are no "bugs", then the software is delivered to a user.

From the above development it is obvious that the problems occurring in the requirement or design stage are uncovered or removed after the coding stage. Actually this results in a lot of money and time spent in the coding stage. To save the money and labor in software development, one question will be raised: is it possible for software developers to find the bugs at the early stage, for example just after the design stage or even before the design stage?

To answer the above question, we first need a good method to be used throughout the whole software development. As a good method, it should be able to deal with the more and more complicated software system. This good method should catch the main characteristics in the software industry. Additionally, it should be easy to understand so that it can be accepted in industry.

In order to find a good way, researchers presented a lot of methodologies to solve the problem in software industry. Now it seems that the object-oriented methodology has played a very important role in designing and analyzing software systems since it was first presented. Many object-oriented methods have been invented but users of these methods had trouble finding a modeling language that met their needs completely.

During the mid-1990s, Grady Booch and Ivar Jacobson, the two inventors of Object-Oriented Software Engineering, and James Rumbaugh, who presented the Object Modeling Technique, began to adopt ideas from each other's methods. This gave a birth to the Unified Modeling Language. Now the Unified Modeling Language has been widely accepted in industry.

After having a good language, we need a solid mathematical method to find bugs at the early stage of software development. Abstract State Machines are a good candidate to deal with this notoriously difficult problem. An abstract state machine has been used in many fields to show its strength in software specification and verification.

This proposal tries to present a tool to find bugs after the design stage so that designers can redesign their model. Designers iteratively deploy this verification tool till no bug is found. This can save time and labor spent in the later stages. We will use Abstract State Machines to give semantics for UML diagrams. Testing, once considered an important method for uncovering faults in unanticipated behaviors, is still viewed as inadequate. Additionally, tests designed for specific scenarios leave unexplored possible combinations of behavior that fall outside the anticipated patterns. To overcome these difficulties, a lot of formal methods are invented to prove a system correct. Model Checker [11] is one of them and it has been successfully applied in a lot of real applications. In this project we will use the Model Checker to do some property verification. The project combines an existed tool, which translates Abstract State Machines into a Model Checker(SMV) input language, and the Model Checker(SMV) tool to implement a UML verification tool.

1.2 Unified Modeling Language

The Unified Modeling Language(UML) is a standard language for writing software blueprints. The UML may be used to visualize , specify, construct and document the artifacts of a softwareintensive systems. UML has become a standardized notation for specifying complicated software systems.

With its birth in 1994, a lot CASE tools for UML have been generated up to now like Rational Rose, Microsoft Visual Modeler etc. All of these tools are very helpful in developing the complex softwares[1]. As far as I know, most of these tools are used to generate the executable code and do some static analysis check.

But UML itself is a very expressive and rich language. The UML models given by designers sometimes contain some behaviors not expected by the designers. How to check whether a UML model satisfies some specifications expected by designers is still a notoriously difficult problem and most of the UML tools have not touched this kind of problems.

To show whether a UML model satisfies a specification is not an easy task, and it becomes more challenging because providing a formal semantics for a language is always difficult. Although OMG provides the semantics for UML in English, it still suffers a lot of ambiguity in its meaning which results in a lot of research which has been done in this field. State machines in UML play a very crucial role in modeling software system behavior. In general an UML dynamic model can be represted by state machines. Activity diagrams and statechart diagrams are two special cases of state machines. As a major result of this proposal, we will give an ASM model for state machines in UML; therefor an ASM model for activity diagrams and statechart diagrams is also given.

1.3 The Role of Verification Tool in UML CASE Tools

Before introducing how a verification tool works, we introduce a UML CASE toolset and the role of a verification tool in the CASE toolset.

As mentioned before, now the software system is becoming more and more complicated and people are looking for some tools when they build a complex software system. And these tools are called CASE tools (Computer Aided Software Engineering tools). With the birth of UML, many CASE tools for UML are also generated. Now we illustrate a toolset provided by Rational Rose to demonstrate how it looks like and works.

In order to provide users to give a model for their software, Rational Rose provides some diagrams editors like class diagram editor and statechart diagram editor etc. Users can use these editors to give different diagrams for a software system. And all these diagrams define a model for a software system being built.

Having given a model by providing different diagrams, we can generate code that represents the model by using Rational Rose. Rational Rose provides several kinds of code generation. From a model Rational Rose can generate C++, Java, PowerBuilder, and Visual Basic. Having generated a code like C++ or Java etc., users can run their software system and try to find bugs in their system. If there are some bugs in the software system, users can redesign the system by

[1]Here "developing" refers to the stage after the design stage. More details about the tools can be found in the next section.

using some diagram editors to give a new model.

On the other hand, the ability to create a model from source code is becoming more and more important. This is the so called “reverse engineering”. Rational Rose provides a reverse engineering tool in its CASE toolset for UML. One of the advantages for using the reverse engineering is that visual modeling is easier for people to design a software system and find problems in the software design. Rational Rose provides the reverse engineering from C++, Java, PowerBuilder and Visual Basic etc. In the following diagram 1, we show the structure for UML CASE toolset provided by Rational Rose. Boxes with a rounded corner represent the tools in the UML CASE toolset; otherwise boxes represent an input or output structure to the tools.

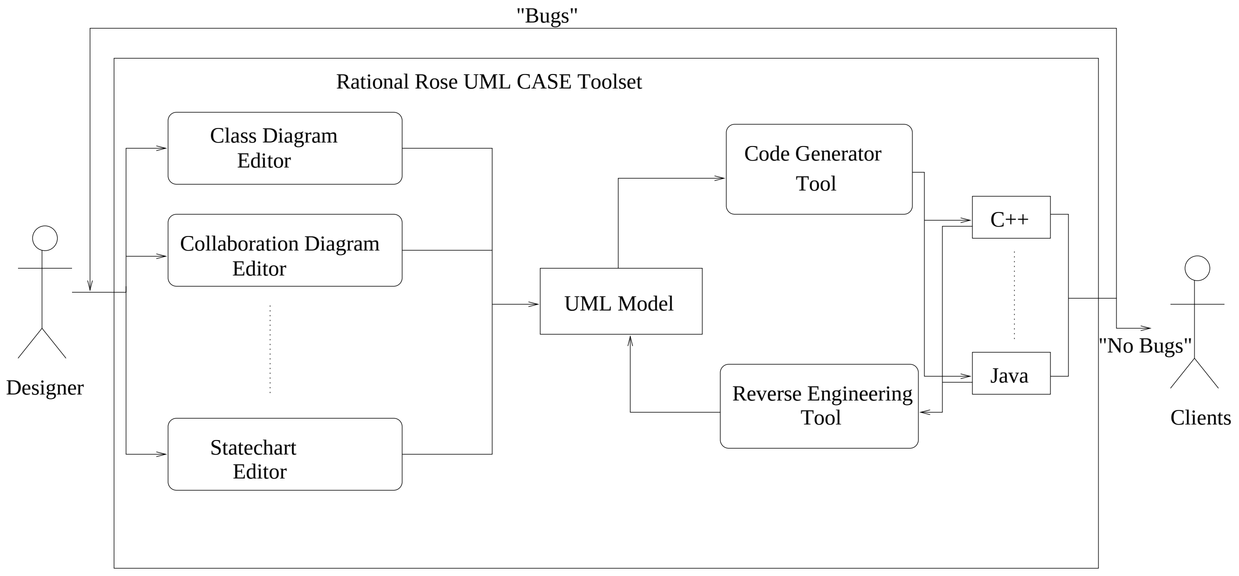


Figure 1: The Structure of a Rational Rose UML CASE Toolset.

However, if there is a verification tool which can verify a model provided by designers before it generates the code, it will provide a great benefit to designers. This can save the time and money spent in the code generation and testing. In general a verification tool accepts a model and specifications, given by designers, and then checks whether the model satisfies the specification. If the verification tool finds some problems, it returns some error information to designers and designers can redesign their system.

On the other hand, assuming we are given a source code and know some bugs in that code. But we are not clear about the design problems in that code. In order to try to find the problems, we can use the reverse engineering tool in Rational Rose to obtain a model for that buggy code. Then we can deploy the verification tool to find out the problems. By using this reverse engineering, we can find bugs existed in all the software we have had now. This will be of great importance to the whole computer world! In the following diagram 2 we show the role of a verification tool in UML CASE Toolset in Rational Rose.

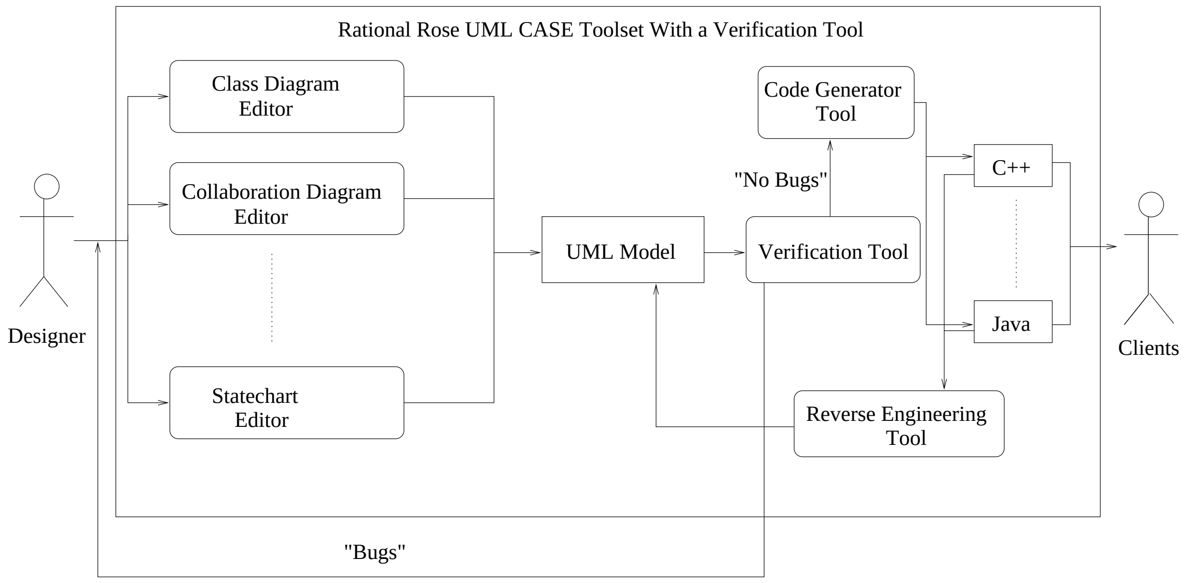


Figure 2: The Role of a Verification Tool in Rational Rose UML CASE Toolset.

1.4 How the tool works

In this thesis, we want to try to build a tool which can be used to check whether a UML model satisfies some properties. Before considering it, we need to find a method to give the semantics for UML, which is another hot topic in UML community. Abstract State Machines(ASM) were first presented by Prof. Gurevich ten years ago and it has been widely used in many aspects of computer systems, including the software engineering. One of the features is that although it has a very simple form, it is powerful in specifying and verifying a lot of complex computer systems.

In order to build a tool to verify a UML model, we will first give an ASM specification for UML models; then we use the ASM verifier to check whether the corresponding UML model satisfies some specifications given by designers. This tool is to be as automatic and transparent to the designers as possible. Therefore, any error information will be returned to a user as UML notations. The overall structure of this verification tool is shown in Figure 3. All boxes with rounded corner denote subtools and other boxes represent the input/output data.

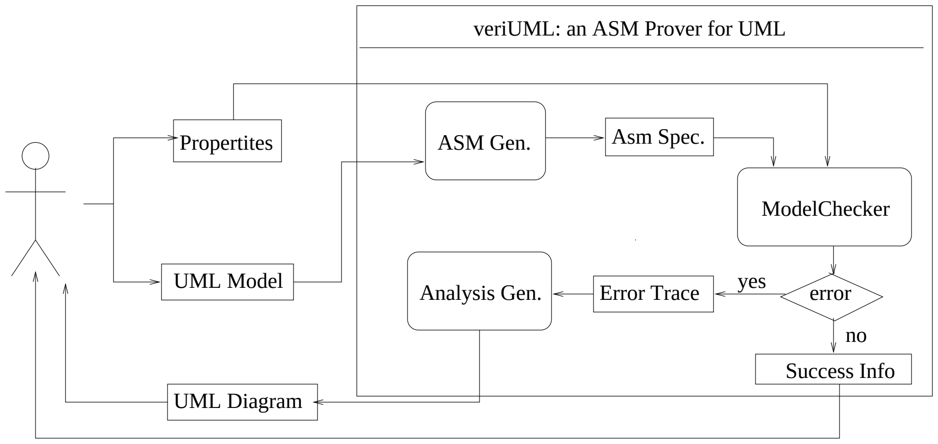


Figure 3: The Structure of the Verification Tool for UML.

In the following sections, we will discuss it in detail. Section 2 will give the reason why Abstract State Machines can be applied in modeling a UML model and we will illustrate this with a previous work: using Montages to give both the static and dynamic semantics for programming language C. Section 3 will focus on how the verification tool works. This includes how we can give the ASM specification for a UML state machine. In section 4, we will discuss a real example: Elevator Problem and show how the verification tool works on this example.

2 Why Use Abstract State Machines?

In this section[2], we will see how Abstract State Machines provide a powerful model for the programming language C. From this section, we are confident that Abstract State Machine can be used in modeling a UML model.

2.1 Introduction to ASM

In the following we describe an ASM model [5] which is sufficient to represent the semantics of C [7]. (ASMs have many features not presented here; see [5] for details.)

[2]Most of the work in this section has been reported in World Congress on Formal Method '99 and it is submitted to ASM 2000 conference.

The *signature* of an ASM A is a finite collection of function names, each name having a fixed arity. A *state* of A is a set, the *superuniverse*, together with interpretations of the function names in the signature. These interpretations are called *basic functions* of the state. A basic of function of arity r is an $r-$ary operation on the superuniverse. When $r = 0$, such a basic function is called a *distinguished element.* The superuniverse does not change as A evolves; the basic functions may. The superuniverse contains some distinct elements *true, false* and *undef* which are used to describe relations and partial functions. They are *logical constants*, whose names do not appear in the signature. In addition, we use equality as a logical constant.

A *universe* U is an important concept in ASMs. It is a special type of basic function: a unary relation usually identified with the set $\{x : U(x)\}$. ASMs provide some built-in universes such as the logic constant $Boolean = \{true, false\}$. When we define a function f from a universe U to a universe V, and write $f : U \rightarrow V$, we mean that f is a unary operation on the superuniverse such that $f(a) \in V$ for all $a \in U$ and $f(a) = undef$ otherwise. We can extend this notation to notations such as $f : U_1 \times U_2 \rightarrow V$ and $f : V$, which means the distinguished element f belongs to V. In addition the expression $f(a)$ can be written in the form $a.f$. For the general case, the expression $f(a_1, \ldots a_n)$ can be written in the form $a_1.f(a_2, \ldots, a_n)$.

There are three kinds of functions in ASMs. A function f is *dynamic* if f can be changed as the ASM evolves. Functions which are not dynamic are called *static. External* functions are syntactically static, but have their values determined by an oracle(that is, the outside world).

In principle, a program of A is a finite collection of rules, which are defined inductively in the following:

- Update Rules:

  $$f(\overline{s}) := t$$

  is a rule with *head* f.

  Here $\overline{s}$ is a tuple $(s_1, \ldots, s_r)$ of terms where r is the arity of f and $r \geq 0$. If f is relational, then the term t must be Boolean. To fire such a rule, change the value of f at the value of term $\overline{s}$ to the value of t.

- Conditional Rules: if g is a Boolean term and R_1, R_2 are rules then

  ```
  if    g    then    $R_1$
  else    $R_2$
  endif
  ```

  is a rule. To fire this rule at a given state A, examine the guard g. If g's value is true at A, then fire R_1; otherwise, fire R_2.

- Block: If R_1, R_2 are rules then

  ```
  do    in − parallel
  ```

```
        $R_1$
        $R_2$
    *enddo*
```

is a rule with *components* R_1, R_2. Do-in-parallel rules are called *blocks.*

Let r_1 and r_2 be update rules of the following forms:

$$r_1: \quad f_1(\overline{s_1}) := t_1 \qquad r_2: \quad f_2(\overline{s_2}) := t_2$$

r_1 and r_2 are said to be *mutually inconsistent* at a given state A if $f_1 = f_2$, and the values of $\overline{s_1}$ and $\overline{s_2}$ are equal but the values of t_1 and t_2 are not equal. Otherwise they are mutually consistent.

To fire a block R at a given state A, determine first if the update rules which will be fired in R_1 and R_2 are mutually consistent. If yes, then fire them simultaneously. If not, do nothing; R is inconsistent at A.

- Do-forall Rules: If v is a variable, $g(v)$ is a Boolean term and $R_0(v)$ is a rule, then

```
    *do*  *forall* $v$ :   $g(v)$
        $R_0(v)$
    *enddo*
```

is a rule with *head variable* v, *guard* $g(v)$ and *body* R_0. A do-forall rule is similar to the do-in-parallel rule, except that the components are not listed explicitly. Suppose R is the do-forall rule above. At a state A which maps every variable in R to a value, the components of R are the rules $R_0(a)$ where a is any element in the state A satisfying $g(a) = \mathit{true}$. To fire R at A, fire simultaneously all these $R_0(a)$ unless they are mutually inconsistent. In the latter case, do nothing.

2.2 Introduction to Montages

Montages[8] are a semi-visual formalism that allow unified and coherent specification of syntax, static analysis and semantics, and dynamic semantics. Generally speaking, for every syntax rule there is one corresponding Montage. In every Montage there can be four parts. The first three parts define the *static aspects* of the language, which refers to the work which can be done at compile time (such as static analysis), and the fourth part defines the *dynamic semantics* of the language.

The first part of a Montage is the *syntax rule.* In general a syntax rule has the form $n ::= E$ where n is a *nonterminal* symbol and E is a string of nonterminal and terminal symbols. We say this rule is *associated* with the nonterminal n. Terminal symbols are the symbols which do not appear on the left-hand side of any syntax rule. The terminal symbols are also called *tokens.* All nonterminal and terminal symbols are called *grammar entities* in the following. The universe *Node* consists of all grammar entities; according to the elements in the universe Node, it can be divided into two subuniverses *Token* and *Nontoken*. The grammar entities on the right hand side of the "::=" symbol are called the *descendants* of n. During the static analysis phase, we use the distinguished element *CN* to denote the token which is currently being analyzed.

The second part of a Montage is the data and control flow diagram, also called the *static analysis graph*, where the Montage describes the data and control flow functions among the nonterminal and terminal symbols defined in the syntax rule. *Control flow functions* describe the execution order among the grammar entities and *data flow functions* describe how values flow through these grammar entities. Control flow functions are represented by dotted arrows and data flow functions are represented by solid arrows. For brevity, we call dotted arrows *control arrows* and the solid arrows *data arrows*. Every data and control function has a name associated with it. Montages provide some special names for certain control functions. *I* and *T* represent the initial and terminal control flow for a Montage. They are used to connect local control flow information to global control flow. *NT*, another important control flow function, represents the token to be executed in the next step during the dynamic semantics computation. Boxes represent nonterminal symbols and ovals represent terminal symbols in the syntax rule. All of the names appearing in boxes or ovals start with "S-" followed by the corresponding grammar entities' name. They represent *selector functions* which allow one to select that grammar entity from its parent when a program is parsed. Besides these attributes one can define other static aspects which can not be represented directly by the flow diagram.

The third part of a Montage is the *static condition*; the conditions in this part should be satisfied during the static analysis phase, otherwise a syntax error will be reported by the static analysis phase. The fourth part of a Montage is the dynamic semantics for this syntax rule, written by ASM transition rules. Actually all the above four parts are translated into ASM and have the final result after running the interpretor for ASM.

2.3 Semantics of C

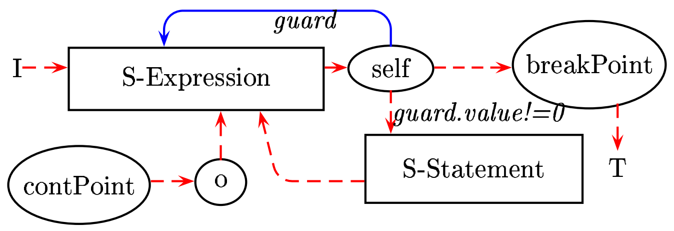
In this section, we will outline the C semantics by several examples. More details about this can be found in [6].

2.3.1 While Statements

We choose the `while` statement in the following to show how we use Montages to represent the semantics of statements in C. The Montage is shown in Figure 4; here we give an explanation of key points of the Montage.

In a `while` statement, the substatements in it are executed repeatedly so long as the value of the guard expression remains unequal to 0. This behavior is represented by the two control flow arrows emerging from the node labeled "self" in the static portion of the Montage; the arrow

Whilestatement ::= "while" "(" Expression ")" Statement



self. lastCtrlStm := EnclosingCtrlStm
EnclosingCtrlStm := self
–
EnclosingCtrlStm := lastCtrlStm

condition *IsArithmeticOrPointerType(self.S-Expression.Terminal)=true*

Figure 4: Montage for `while` Statements.

labeled "*guard.value*! = 0" is followed when that guard expression is true (*i.e.*, when the loop should continue), while the other arrow is followed when the guard is false.

The guard expression in the `while` statement must be of arithmetic or pointer type. This restriction is given in the condition part of its Montage; the function *IsArithmeticOrPointerType* has a detailed definition which we omit for brevity.

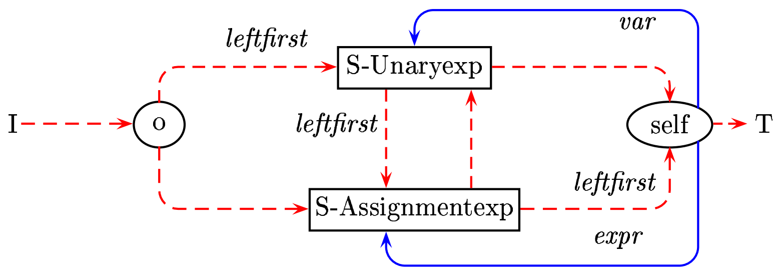
The semantics of the `while` statement becomes complicated when there is a jump statement within it. For example, a `break` statement terminates execution of the smallest enclosing loop or `switch` statement; a `continue` statement causes control to pass to the loop-continuation portion of the smallest enclosing loop statement. In the static part of the Montage, we use two ovals named "contPoint" and "breakPoint" to denote the two targets for a `continue` statement and a `break` statement respectively. Function $EnclosingCtrlStm : Node$ and $lastCtrlStm : Node \rightarrow Node$ are used to represent the smallest enclosing loop statement; we set (and reset) those functions as we process the loop's substatements, so that the Montages for those substatements will know where to direct control flow in those situations.

2.3.2 Assignment Expressions

We consider here the simplest form of the assignment statement in C, shown in Figure 5; again, we give an explanation of key points of the Montage.

The value of an assignment expression is the value stored in the left operand after the assignment has taken place. This is given in the dynamic part of the following Montage. The order in which the two operands is evaluated is ambiguous (*i.e.* under-specified by the definition

Assignment ::= **Unaryexp Assignop Assignmentexp**
Assignop = "="



```
staticType := S-Unaryexp.Terminal.staticType
if ((Primarycase1(S-Assignmentexp)=true) and
      (S-Assignmentexp.Terminal.staticType.arrayType=true)) then
      S-Assignmentexp.onlyArrayNameVisted := true
else
      S-Assignmentexp.onlyArrayNameVisted := false
endif
```

condition *(CompareType(S-Unaryexp.Terminal,*
S-Assignmentexp.Terminal,S-Assignop.Name)=true) and
(S-Unaryexp.Terminal.lvalue=true)

value := AssignOp(var, expr, S-Assignop.Name)

Figure 5: Montage for assignment expressions

.

of C); a function $leftfirst : Boolean$ is used to denote whether the left subexpression should be evaluated first or not.

The condition portion of the Montage makes use of a function $CompareType$ which ensures that both operands are of types which are compatible for assignment (*e.g.*, both are arithmetic values). The definition is lengthy but straightforward; we omit it for brevity.

A special case in handling assignment statements occurs when the right-hand expression in the statement represents the name of an array rather than a variable name; in such situations, we need a different value to be computed by the subexpressions in order to complete the operation. The function $onlyArrayNameVisited : Node \rightarrow Boolean$ is used to denote whether this case has occurred; the value of this function is used by other Montages while evaluating the subexpression in order to generate the correct value.

2.3.3 Additive Expressions

There are many mathematical expressions in C involving binary operators ("*", "+", "−", *etc.*) whose behaviors are similar. (We treat the bit-wise operators (e.g., |,&) as ordinary mathematical operators.) The Montage for addition expressions is shown in Figure 6; again, we comment on relevant features of the Montage.

To evaluate an addition expression, one evaluates both subexpressions (in some order) and combines the results appropriately. The control-flow for this Montage is thus similar to that for the assignment operator considered previously.

The combining operation is usually conventional addition; however, C overloads the addition operator by allowing addition of an integer i and pointer p, with the result being a pointer which is i units forward in memory from p. The dynamic rules for this Montage implement this overloading.

Determining the static type of this expression presents a couple of complications. One is the operator overloading just discussed; most of the rules in the static section distinguish between various integer/pointer operand combinations. The other complication comes with arithmetic conversions between arithmetic types; the function $ConvertName : Node \times Node \rightarrow Node$ encodes the complicated rules of [7] for this situation (again, we omit the details).

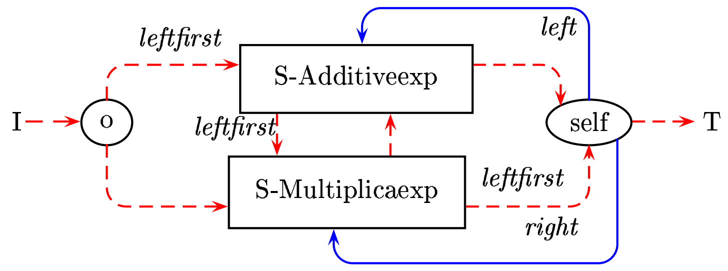
2.3.4 Function Definition

We consider the Montage for user-defined function definitions (as opposed to ASM functions) in Figure 7; we highlight relevant portions of the Montage below.

As one might expect, function definitions present a great deal of static information which must be analyzed; consequently, the static portion of the Montage is large. Here we discuss various portions of the static analysis.

User-defined functions (as opposed to ASM functions) are block-structured in C; the variables which are declared in a C-function can only be referenced within that function. Thus, when a variable is referenced, the corresponding declaration in the smallest enclosing block structure

Addexp ::= **Additiveexp AddOp Multiplicaexp**
AddOp = ”+”



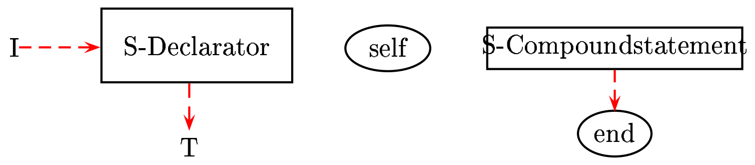
```
if(((S-Additiveexp.Terminal.staticType.pointType=true) or
    (S-Additiveexp.Terminal.staticType.arrayType=true)) and
    (S-Multiplicaexp.Terminal.staticType.Name=”int”))then
    staticType:=S-Additiveexp.Terminal.staticType
elseif(((S-Multiplicaexp.Terminal.staticType.pointType=true) or
           (S-Multiplicaexp.Terminal.staticType.arrayType=true)) and
         (S-Additiveexp.Terminal.staticType.Name=”int”)) then
    staticType:=S-Multiplicaexp.Terminal.staticType
else
    staticType:=ConvertName(S-Additiveexp.Terminal,S-Multiplicaexp.Terminal)
endif
constValue := Apply(S-Additiveexp.Terminal.constValue,
                    S-Multiplicaexp.Terminal.constValue,S-AddOp.Name)
```

condition *CompareArithType(S-Additiveexp.Terminal,S-Multiplicaexp.Terminal, S-AddOp.Name)=true*

```
if((left.staticType.arrayType=true) or
    (left.staticType.pointType=true)) then
    value := left.value + (right.value * left.staticType.staticType.typelength)
elseif((right.staticType.arrayType=true) or (right.staticType.pointType=true))then
    value := right.value + (left.value * right.staticType.staticType.typelength)
else
    value := left.value + right.value
endif
```

Figure 6: Montage for addition.

Funcdefcase2 ::= Declspecifier Declarator Compoundstatement



```
CurFundef := true
CurFunc := self
self.lastScope := EnclosingScope
EnclosingScope := self
FuncDef := true
–
CurFundef := false
FuncDef := undef – used to distinguish fun def and decl
CurFunc := undef
EnclosingScope := self.lastScope
do forall g in Gotostatement
   if Within(g, self)=true then
    g.gotoTarget := RecLabelLookup(g.S-Identifier.Name,g.scope)
   endif
enddo
let d=LookUpFuncDcl(S-Declarator.Initial.Name,self.lastScope) in
    if(S-Declarator.S-Parametertypelist$\neq$undef)then
         d.AddrToFunc := S-Declarator.S-Parametertypelist.Initial
         S-Declarator.S-Parametertypelist.nextStatement:=
                                 S-Compoundstatement.Initial
    else
         d.AddrToFunc := S-Compoundstatement.Initial
    endif
    d.funvarType := true
endlet
```

condition

```
do forall g in GotoStatement
    if Within(g, self)=true then
         RecLabelLookup(g.S-Identifier.Name,g.scope)$\neq$undef endif
enddo
```

```
@end:
StackTop := StackTop - 1
CT := RecCaller(StackTop)
```

Figure 7: Montage for function declarations.

is found. The function *EnclosingScope* : *Node* is used to denote the current block structure; the function *lastScope* : *Node* $\rightarrow$ *Node* is used to denote the outer block structure containing the current block structure. They are assigned before and after a C-function is called, thus maintaining a chain of enclosing scopes while permitting other Montages to know their enclosing scope immediately.

C permits the infamous `goto` statement within a function block. While processing that block, we build a function *RecLabelLookup* : *String* $\times$ *Node* $\rightarrow$ *Node* to indicate the node within the specified scope labeled with the specified string. At the point that static processing returns to this node, we will have seen all targets for `goto` statements; we thus can patch those `goto` statements with the locations of their targets. The condition part of the Montage ensures that all `goto` statements have defined targets.

In C, user-defined functions are stored in memory; the names of functions are treated as variable names, just like other variables. The value that is stored in a C-function variable is an implementation-dependent address which is used to transfer control to that function. In order to create an executable Montage for C, we will have to give meaning to this implementation-dependent value; we choose to store in this variable the Montage node corresponding to the declaration for this function. We use an additional function *AddrToFunc* : *Node* $\rightarrow$ *Node* to map the node where a function is declared to its initial definition node. In the static analysis, we set *AddrToFunc* to point to the first executable node for this function: either the list of parameters to be initialized (in the function declarator) if parameters exist, or the first statement of the function block otherwise.

At the same time, C functions may have several active incarnations at a given time during their execution. Thus we must have some means for storing multiple values of some ASM functions (*e.g.* "*value*") for a given token. We use a universe *Stack* comprising the positive integers for this purpose; a dynamic distinguished element *StackTop:Integer* is used to indicate the current top of the stack.

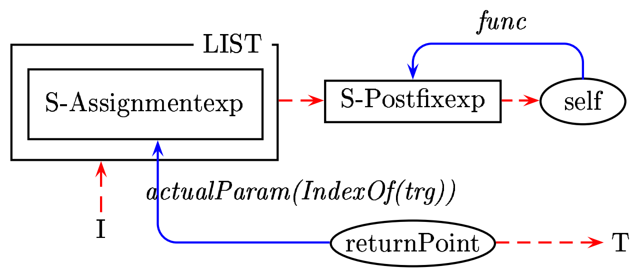
The dynamic semantics of the Montage decrements the value of *StackTop* because the C-function execution has finished. In order to make control transfer to the corresponding token after a C-function has been executed, we define a new function *RecCaller* : *Stack* $\rightarrow$ *Node* to store the node that calls this C-function at a given recursion level. We set the value of *RecCaller* when a C-function is called, which is shown in the Montage for C-function calls. Then the dynamic semantics transfers control to the node following the function call by using the function *RecCaller*. This same mechanism is also used by our Montage for `return` statements to handle function returns.

2.3.5 Function Call

The Montage for function calls is shown in Figure 8; again, we highlight the relevant features.

In C, a function call is an expression whose type is given by the return type of the function. An expression of type "function returning T" is usually converted to "pointer to function

Postfunpar ::= Postfixexp "(" Assignmentexp {"," Assignmentexp } ")"



```
if((S-Postfixexp.Terminal.staticType.pointType=true) and
    (S-Postfixexp.Terminal.staticType.staticType.funvarType=true))then
returnPoint.staticType:=S-Postfixexp.Terminal.staticType.staticType.static
else
    returnPoint.staticType:=S-Postfixexp.Terminal.staticType.staticType
endif
```

```
CT := MemoryValue(func.value).AddrToFunc
StackTop := StackTop + 1
RecCaller(StackTop+1) := returnPoint
```

Figure 8: Montage for function calls.

returning T". So we need to distinguish between the two cases when computing the type of a function.

We showed in the previous section how the *AddrToFunc* function is used to establish the correspondence between addresses where (user-defined) functions are stored and the first node in their Montage representations. To perform a function call, we use the *AddrToFunc* function to find the corresponding next node, increment the value of the stack, set the return node in the *RecCaller* function, and perform the branch. (Note that the arguments to the function call will have already been evaluated; those values will be copied into the callee when the callee's parameter nodes are visited.)

3 Behind the Verification

In this section we will discuss the structure of the verification tool (veriUML) and how it can be used to check whether a UML model satisfies a specification and other issues related to this tool veriUML.

3.1 How the tool works

Users of this tool are those designers who want to use UML to develop a complex software system. They can use any existing UML edit tool to provide a UML model as an input to this verification tool. At the same time, they need to provide specifications so that they can check whether the UML model satisfies these specifications.

The verification tool will use a UML model and some properties as its input; then it analyzes and checks them. If it finds something wrong, it will return its users some error information. The users don't need to know how this verification tool works behind them. As a result, what they will have, when a UML model does not satisfy a property, is some kinds of UML diagrams.

3.2 Structure of the Verifier

In order to provider UML designers a good tool to verify some requirements during the software specification phase, we design the verification tool which consists of the following subtools. The translation tool reads a UML model and translates it into an ASM model. The model check tool, which reads an ASM model and specifications provided by a designer, checks whether the ASM model, which is directly translated from the UML model, satisfies the specifications. The last subtool is an analysis tool which is used to analyze the result returned by the model checker tool. The verifier will return the result in both UML diagrams and textural form so that designers are not involved too much about the mathematics knowledge behind this tool.

The verifier reads a UML model, which is represented by UML diagrams, as its input. And the at the same time, it reads a requirement from a user who wants to verify whether the UML model satisfies the requirement. Generally speaking, to model a complex software by using UML diagrams, a user uses a class diagram, collaboration diagram and state machine. In the current implementation of the verifier we will use these diagrams as the input notation.

After translating a UML into an ASM model, the Model Checker takes the responsibility to analyze whether the requirements satisfy the ASM model. Actually Model Checker translates the ASM model into SMV which is used to implement the properties check. When an counter-example is found, the result will be output to the next subtool in the verifier so as to return to the user an error information in UML diagrams.

Last the analysis tool is used to accept the result returned by the Model Checker. If the result is a counter-example, then analysis tool returns the textural form and UML diagrams as well. According to the error information in these two forms, the user can redesign their system.

3.3 Translation Tool

The Translation Tool is used to translate a UML model into ASM specification. In order to give a model for a software system, a user of UML usually uses the class diagram, collaboration diagram and state machine[3]. Among these diagrams, state machine plays a crucial role in describing the software dynamic behavior. In this section we will see how we can give an ASM specification for a state machine diagram in UML.

3.3.1 State Machine in UML

A *state machine* is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its response to those events. It focuses on an object's dynamic behavior.

A state machine consists of the following elements. A *state* is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event. An *event* is a specification of a significant occurrence that has a location in time and space. It is an occurrence of a stimulus that can trigger a state transition. A *transition* is a relationship between two states indicating that an object in the first state will perform some actions and enter the second state when a specified event occurs and specified guards are satisfied.

A diagram of a `state machine` in UML generally consists of an initial state, a set of states and a set of arcs which connect the state. First let us introduce the basic elements in state machine.

1. **state**. A state[4] is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event. A state has several parts: name, entry/exit actions, internal transitions, substates and deferred events.

[3]In UML, a statechart diagram shows a state machine

[4]In the following unless it is explicitly stated, a "state" refers to a state of a state machine in UML.

A state is represented as follows. Entry/exit actions denote the actions executed on entering and leaving the state. The substates refer to a nested structure of a state, involving sequentially active or concurrently active substates. Figure 9 shows a standard state in a state machine diagram in UML. In the example shown in Figure 9, setMode(onTrack) is an entry action; setMode(offTrack) is an exit action and followTarget is an activity. Generally a state is represented by a box with a rounded corner. If a state does not include a substate, then this state is called a simple state; otherwise the state is called a composite state. A composite state can be divided into two types. One is concurrent composite state which includes several regions. Every region includes a set of states and regions are separated by dotted lines. The other is sequential composite state. It includes a set of states and no dotted line within the state. There are two special kinds of state, initial state and final state. Because there is no action in both initial and final state. So an initial state is represented by a bullet; a final state by a bullet surrounded by a circle. They are shown in Figure 10.

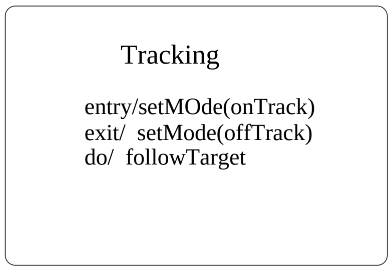


Figure 9: An example for a state in a state machine diagram.

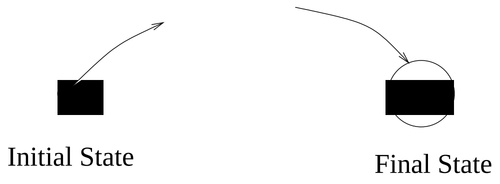


Figure 10: A initial and final state in a state machine diagram.

2. **transition** A transition is an arc between two states in the state machine diagram. It has

five parts: source state, event trigger, guard condition, action and target state. A source state denotes the state affected by the transition. An event trigger denotes the event whose reception by the object in the source state makes the transition eligible to fire if the guard is true. This guard is represented by a guard condition. An action denotes an executable atomic computation. A target state is a state which is active after the completion of the transition. Let us take a look at an example to see how a transition is represented in a state machine in UML. Assume *tr*=(src,event, guard,actions, target), the corresponding transition in a state machine is shown in 11.

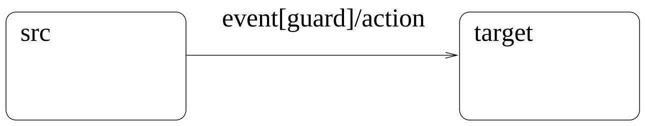


Figure 11: A transition *tr* in a state machine in UML.

Whenever a state is entered, it executes its entry action before any other action is executed. Conversely, when a state is exited, it executes its exit action as the final step prior to leaving the state.

The activity of a state represents the execution of a sequence of actions, which occurs while the state machine is in that state. An action in UML is an atomic computation, which is not interruptible. If the activity completes while the state is still active, it raise a completion event. In case where there is an outgoing completion transition, which has no explicit trigger event, the state will be exited. If the state is exited as a result of the firing of an outgoing transition before the completion of the activity, the activity is aborted prior to its completion.

In a state machine diagram in UML, there are two kinds of composite state. One is a sequential composite state and the other is a concurrent composite state. If a sequential composite state is entered, only one of its substates is active. If a concurrent composite state is entered, all of its regions are active.

If a transition terminates on the outside edge of a sequential composite state, then the entry action of that state is executed before the action associated with the initial transition. If the transition goes to a substate of the composite state, then that substate becomes active and its entry code is executed after the execution of the entry code of the composite state.

Whenever a concurrent composite state is entered, each one of its regions (concurrent sutstates) is also entered, either by default or explicitly. If a transition terminates on the edge of the composite state, then all the regions are entered using default entry. In this case the transition coming out from the initial node is taken. If the transition explicitly enters one or more regions, these regions are entered explicitly and the others by default.

When exiting from a sequential composite state, the active substate is exited recursively. This means that the exit actions are executed in sequence starting from the innermost active state.

When exiting from a concurrent state, each of its regions is exited. After that, the exit actions of the regions are executed.

3.3.2 Sketch of ASM Model for State Machine in UML

Before we dive into the details about an ASM model for the state machine in UML, we first give some general ideas about this model in this section.

First we extend a state machine diagram to an extended Abstract State Machine diagrams. The extension mainly focuses on adding some transitions between nodes in an extended Abstract State Machine diagram. First we consider those transitions coming from a node[5]. In UML state machine if an event associated with a transition occurs, then not only is the node, which 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 Figure 16 and 17 readers will find more details about these new transitions.

Therefore, in an extended Abstract 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.

On the other hand, we also add some transitions in an extended Abstract 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.

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 13.

Having derived an extended Abstract 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.

To model a node's execution, we give two kinds of agent. One is used to execute all actions associated with a node which is called a graphical agent, simply referred as an agent. The other is to execute the activity associated with a node and we call it an activity agent. When an activity agent finished its execution, it will trigger a completeness event to possibly pass control to the next node.

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

When an agent's control reaches a concurrent composite node, it creates a new agent for every region in that concurrent composite node. And every new agent executes a corresponding node within that region.

To model an interruption in UML, we let an agent generally sit at a either simple or concurrent composite node. The reason for an agent not sitting at a sequential composite node is that an agent's control is passed to a subnode of the sequential composite node when that composite node is reached.

If an interruption (an event) with a transition occurs and control sits at a simple node, then control stops the simple node's activities and executes its exit action. Next control passes to its immediate outside node to execute the outside node's exit action. All exit actions will be executed one by one until either the most outside source node for that transition[6] or a concurrent composite node is reached. Informally, a most outside node for a transition is the outside node which includes the source node for that transition but does not include the target node. A list of nodes, which have a nested structure relationship, are represented by a function called $NestStructure$.

If control sits at a concurrent composite node, all agents associated with that node will be killed and only does the agent associated with that concurrent composite node exist and it executes the exit action and all its outside node's exit actions. This implements the requirement for an interruption.

Because we extend the state machine in UML to an extended Abstract State Machine diagram and all transitions occur explicitly for every node, it is fairly easy to model the state machine diagram in a methodic way. In the next we will give the details about this model.

3.3.3 Signature for An Extended ASM Diagram

A UML user can use states and transitions to model an object dynamic behavior. Before giving the ASM specification for a state machine, we introduce an extended Abstract State Machine diagram (abbreviated as extended ASM diagram in the following) corresponding to every state machine diagram. In an extended ASM diagram we refer as a node a state in a state machine diagram. And an arc represents a transition in a state machine diagram. The universe NODE denotes all possible nodes whose corresponding states can possibly appear in the state machine diagram. A special element $TOPNODE : NODE$ denotes an imaginary top node which includes the whole extended ASM diagram.

In an extended ASM diagram, a node includes a name, entry action, exit action, internal transition, activity and deferred events[7]. Associated with a simple state are incoming transitions and outgoing transitions. We denote such a state as Node(name, entry, exit, internal, activity, deferred, $inArc_1, \ldots, inArc_n, outArc_1, \ldots, outArc_m$) where $inArc_i, i \in \{1, \ldots, n\}$, denote all the incoming transitions and $outArc_i, i \in \{1, \ldots, m\}$, all the outgoing transitions to and from

[6]The formal definition for a outside source node for a transition is defined in the following.

[7]At this moment, deferred events are not considered.

the node respectively. The rest parameters denote the entries in the corresponding state defined in a state machine.

Similarly, we denote a sequential composite state as Node(name, entry, exit, internal, activity, deferred,$inArc_1, \ldots, inArc_n$,nodes,$outArc_1, \ldots, outArc_m$) where nodes is a set of subnodes $\{D_1, \ldots, D_n\}$, each of which is an either simple or composite node. For any node D= Node($\ldots \{D_1, \ldots D_i \ldots, D_n\} \ldots$), a function $UpNode : NODE \rightarrow NODE$ assigns to a node its immediately enclosing node. A function $DownNode : NODE \rightarrow 2^{NODE}$ indicates a set of immediately enclosed nodes for a given node. Given the above example, we have $UpNode(D_i) = D$ and $DownNode(D) = \{D_1, \ldots, D_n\}$. D_i is called an immediate subnode of D. If a composite sequential node is active, exactly one of its subnodes is active.

We denote a concurrent composite node as Node(name, entry, exit, internal, activity, deferred, $inArc_1, \ldots, inArc_n, nodes_1, \ldots, nodes_n, outArc_1, \ldots, outArc_m$), where $nodes_i (i \in \{1, \ldots, n\})$ is a set of nodes $\{D_{i1}, \ldots, D_{ij_i}, \ldots, D_{im_i}\}$. D_{ij_i} is an either simple or composite node. $node_i$ is called a region[8] in that composite concurrent node. Every region is separated from the other regions by dotted lines. Every node D_{ij_i} in a region can be an either simple or composite node. If a node D is an instance for the above definition, then function $UpNode(D_{ij_i}) = D$ and $DownNode(D) = \{\ldots, D_{ij_i}, \ldots\}$ are defined. D_{ij_i} is called an immediate subnode of D. If a concurrent composite node is active, then all of its regions are active. A concurrent composite node or sequential composite node is also called a composite node.

A boolean function $IsSimple : NODE \rightarrow Boolean$ is used to indicate whether a node has subnodes or not. If a node a satisfies $IsSimple(a) = true$, then it means there is no subnode in a. We call it a `simple node` in the extended ASM diagram. Otherwise it includes subnodes which are either sequentially active or concurrently active. In order to distinguish these two cases, a function $IsCompSeq : NODE \rightarrow Boolean$ indicates whether a node is a sequential composite node or not. And a function $IsCompConcur : NODE \rightarrow Boolean$ denotes concurrent composite nodes.

Given two nodes, we can check whether these two nodes have an enclosing relation by a function $Including : NODE \times NODE \rightarrow Boolean$. Its definition is given in the following:

$Including(D_1, D_2) = true$ iff
$(UpNode(D_2) = D_1) \vee (\exists D : (UpNode(D_2) = D \wedge Including(D_1, D) = true))$ **or** $D_1 = D_2$

If two nodes D_1, D_2 satisfy $Including(D_1, D_2) = true$, then a function $UpChainNode : NODE \times NODE \rightarrow NODE^{\star}$ is used to indicate a chain of nodes between the two nodes.

$UpChainNode(D_1, D_2)=\{T_1, \ldots, T_n\}$ where
$T_1 = D_1, T_2 = UpNode(T_1), \ldots, T_n = UpNode(T_{n-1}) = D_2$

[8]In a state machine diagram there are no names for regions. Here we give names for convenience.

The universe ARC denotes all possible transitions appearing in a state machine diagram. An arc is denoted as Arc(sourcestate,event, guard,action,targetstate). Except for targeststate[9], the rest of parameters are the ones appearing in the corresponding transition.

The universe EVENT denotes all possible trigger events occuring in the arc. There are two types of arc in the extended ASM diagram. One is an arc with a trigger event and the other is without a trigger event. For those arcs which do not have any event, we call them completion arcs. A boolean function $IsTriggerless : ARC \rightarrow Boolean$ denotes this kind of arc. A completion arc is triggered by a completion event which is generated when all arc and entry actions and activities in the currently active node are complete. Given a completion arc, a function $HasGenEvent : ARC \rightarrow Boolean$ indicates whether a completion event has already generated. For each type of parameter occuring in either Node or Arc, we use a function *param* to yield the corresponding parameter.

For a given arc, we can denote as a node the *least common ancester*, which is denoted by a function $lca : ARC \rightarrow NODE$. A least common ancester node of an arc is the lowest composite node that contains the explicit source and target nodes of the arc. Its definition is shown in the following:

lca(ar)=a iff

$$\begin{aligned}&\forall b \in NODE : Including(a, targetstate(ar)) = true\wedge\\ &Including(a, sourcestate(ar)) = true\wedge\\ &Including(b, targetstate(ar)) = true\wedge\\ &Including(b, sourcestate(ar)) = true \Rightarrow\\ &Including(b, a) = true.\end{aligned}$$

Given an arc ar, the most outside source node of the arc ar is yielded by a function $OutMostSource : ARC \rightarrow NODE$. The node indicates a source node which is a subnode of the least common ancester node and its definition is as follows:

$$OutMostSource(ar) = D \text{ iff } Including(D, sorucestate(ar)) = true \wedge UpNode(D) = lca(ar)$$

A function $OutMostTarget : ARC \rightarrow NODE$ yields a most outside target node which is a subnode of the least common ancester.

$$OutMostTarget(ar) = D \text{ iff } Including(D, targetstate(ar)) = true \wedge UpNode(D) = lca(ar)$$

Figure 12 shows how to compute the above functions. Given an arc ar, we have lca(ar)=c, $OutMostSource(ar) = b$ and $OutMostTarget(ar) = d$.

[9]If an arc's corresponding transition has more than one target state, we replace it with a new arc in the extended ASM diagram so that every arc has only one target. This is shown in the following section.

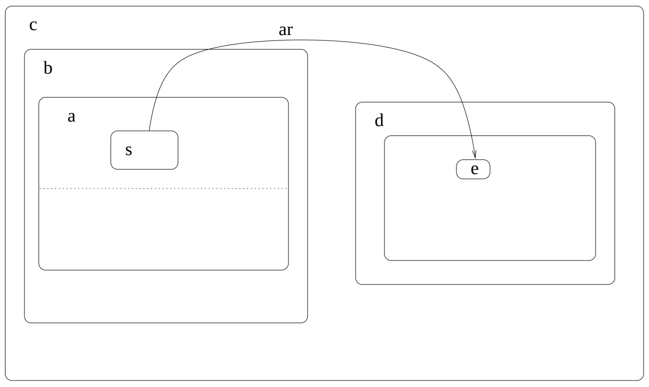


Figure 12: An example for the function $OutMostSource$ and $OutMostTarget$.

Now given a node and an arc which comes out from the node, a function $NestNodesFromArc : NODE \times ARC \rightarrow NODE^{\star}$ is used to indicate a chain of nodes between a source and its most outside source node. Its definition is as follows:

$NestNodesFromArc(node, ar) = UpChainNode(node, D)$, where $D = OutMostSource(ar)$ For example, in Figure 12 $NestNodesFromArc(s, ar) = \{s, a, b\}$.

Assuming $List : NODE^{\star}$ whose elements are represented as a list $\{D_1, D_2, \ldots, D_n\}$, we denote as the first element $head(List) = D_1$ and $tail(List) = \{D_2, \ldots, D_n\}$

3.3.4 An extended ASM diagram

Now we introduce how to derive an extended abstract state machine diagram from a state machine diagram. All the states in a state machine diagram are kept but some changes are made on some transitions in an extended ASM diagram. This changes can be divided into two separate parts. One is for incoming transitions and the other is for outgoing transitions. First we consider the changes to those incoming transitions.

1. step one: If an arc enters a concurrent composite node and does not terminate on a one or more regions, then we extend this arc so that it explicitly terminates on all its regions. We extend this arc which terminates on the node(s), which is the target node for the initial node within the implicitly region(s) for the concurrent composite node, see Figure 13. This kind of arc is called a fork arc, whose target nodes are more than one.

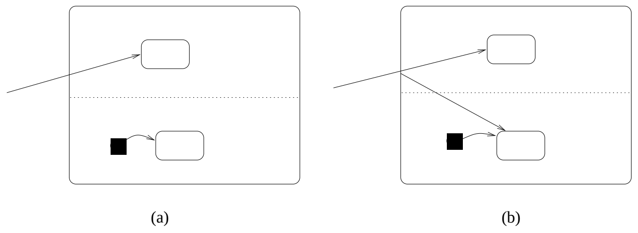


Figure 13: (a) shows an transition in a state machine diagram terminating on a substate within a concurrent composite state. (b) shows a new arc is generated in the other region.

2. step two: If an arc ar's corresponding transition in a state machine diagram enters a composite state and terminates on the edge of its substates, then the arc ar is replaced by a new arc whose target node is set to OutMostTarget(ar). Except for the target state, the new arc copies all parameters from the old arc ar. Furthermore, for every target node $t = targetstate(ar)$, including the arc ar having more than one target state, we add a set of new imaginary arcs based on the following rule. If $\forall d_1, d_2 \in UpChainNodes(t, OutMostTarget(ar)) : UpNode(d_2) = d_1$ and $(IsCompSeq(d_2) = true$ or $IsCompConcur(d_2) = true)$, then a new imaginary arc between d_1 and d_2 is generated. The new arc's source and target states are d_1 and d_2 respectively. All these imaginary arcs, used in the definition for a function *initArc*, indicate where control should go when a composite node is entered. All these imaginary arc's guards are set to true and no trigger event, actions in the arc. Figure 14 shows an example for this change. We use a dotted line ended by an arrow to represent the new imaginary arc. In this and the following Figures we denote *tr* as a transition name.

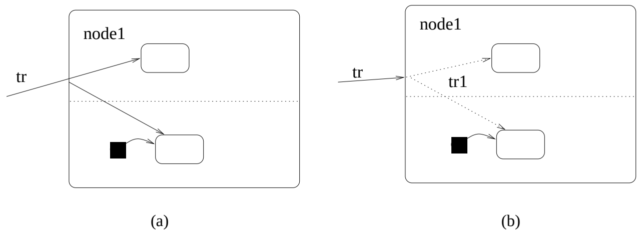


Figure 14: (b) shows some changes are made on an arc *tr* shown in (a). The new imaginary arcs are represented by dotted line ended by an arrow.

When there is more than one target state in an arc, there is only one arc to be generated if the two new imaginary arcs' source and target nodes are identical. For example, only one arc a is generated in the following case shown in Figure 15.

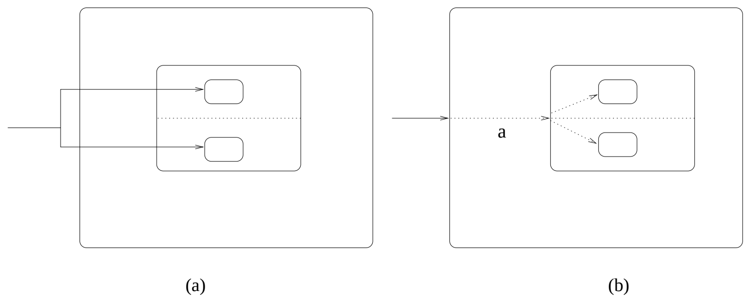


Figure 15: The only arc a is generated in (b) for a case shown in (a) .

Now we consider how to make changes to those outgoing arcs. All these changes are made based on the exit action associated with a node.

There are two ways to exit from a node in general. One is a normal exit. It satisfies two conditions. One condition is there is a completion arc which is an arc without any explicit event trigger. The other condition is: for a simple state, its activity is done; for a sequential composite node, besides finishing the activity defined in that composite node, the final node within that composite node is reached; for a concurrent composite state reaches, all its regions reach their final node as well as the activity defined in that composite node is done. When the second condition is satisfied, then the completion arc is fired if the guard associated with it is true. This is so called a normal exit.

An abnormal exit is another way to exit from a state. It occurs when an event associated with an arc happens and the arc's guard condition is true. This results in stopping the activities for those nodes affected by this arc. And those nodes' exit actions will be executed.

To reflect an abnormal exit, we add new arcs for a given arc based on the following two cases.

If an arc ar=(sourcestate,...,targetstate) satisfies sourcestate! = OutMostSource(lca(ar)), then $\forall D \in$ *UpChainNodes(sourcestate, OutMostSource(lca(ar)))* and IsCompConcur(D)=true we add a new arc from D ar_D=(D,...,targetstate) where the first parameter of ar is replaced by D. Except for the source node and guard condition, all the other parameters are kept in the new arc. The guard condition is generated by a boolean function whose name is the characters "guard_" followed by the associated arc name. For example, the condition for the new generated arc for the above ar is *guard_ar*. If the guard condition for the original arc ar becomes true and its event occurs, we set the function *guard_ar* to be true. Therefore all these newly generated arcs can be fired. This additional arc is shown in Figure 16.

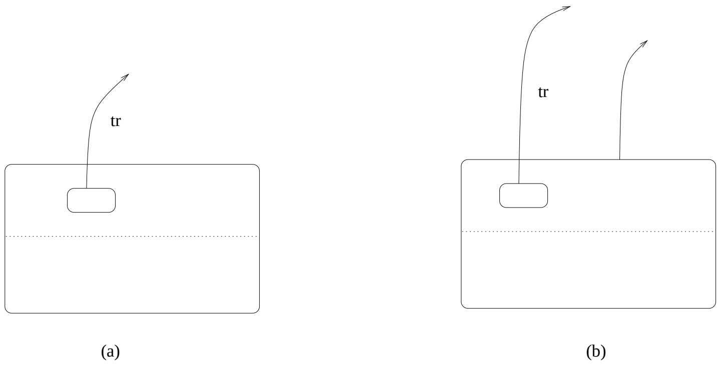


Figure 16: An arc is generated in (b) according to the arc *tr* in (a).

If an arc *tr*=(sourcestate,...,targetstate)'s source state is a composite state, then, $\forall D$: *IsSimple(D)=true* $\wedge$ Including(sourcestate,D)=true, we add a new arc originating from D tr_D=(D,...,targetstate). And for any $D' \in UpChainNodes(D, sourcestate) \wedge$ IsCompConcur(D')=true, we also add a new arc tr_D=(D',...,targetstate). Similarly except for the source node and the guard condition, all the other parameters are kept in the new arc. For the guard condition, we deal with it in the same way we mentioned before. This can be shown in the Figure 17. The reason for a set of new arcs being added is that those source nodes of the new arcs are affected by the event associated with the old arc.

Based on the above rules for adding new arcs, for a given state machine diagram, we can derive an extended abstract state machine diagram. For every node in the diagram, a function $evList : NODE \rightarrow EVENT^\star$ indicates a list of events which cause an abnormal exit during the execution. The arc, associated with a node, containing that event is yielded by a function $event2arcEVENT \times NODE \rightarrow ARC$. A function $NormalevList : NODE \rightarrow ARC^\star$ indicates a list of transitions with a node, which do not have any event. If a node's activity is done and serveral arcs associated with that node are eligible to be fired, a function $ChooseArc : NODE \rightarrow ARC$ is used to indicate which arc will be fired. And a external function $HighPriority$: $ARC \times ARC \rightarrow Boolean$ indicates whether the first parameter arc has a higher priority than the second one. The function $ChooseArc$ is shown in Figure 18.

Now we discuss which arc will be active when a composite node is initially entered. A function $initArc : ARC \times NODE \rightarrow ARC$ indicates which arc is active when the composite node is initially entered. The first parameter represents an incoming arc and the second parameter

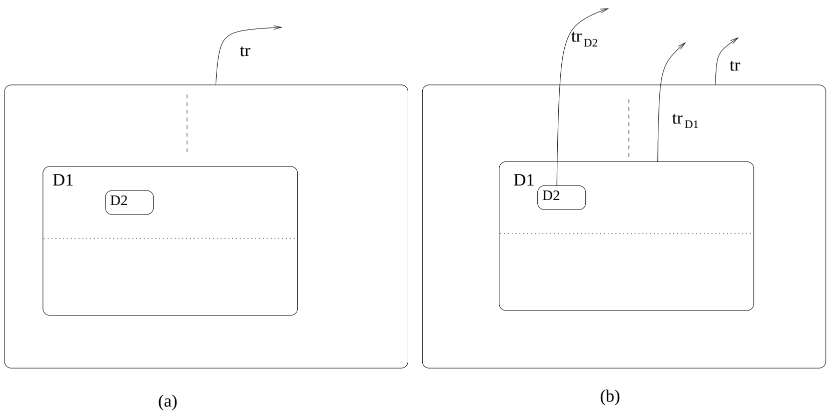


Figure 17: Some arcs are generated according in (b) according to the arc tr in (a).

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)

Figure 18: The definition for function ChooseArc(node).

refers to a region within that (concurrent) composite node. When an arc ar terminates on the edge of a sequential composite node n, then the arc coming out from an initial node is the value for the function initArc(ar, n). When an arc ar terminates on the edge of a concurrent composite node n, $node_i$ is used to get the next active arc within the region $node_i$[10]. If a new arc is generated in that region, then the new arc is the value for that function initArc; otherwise the arc coming from the initial node in that region is the value for the function initArc. For example in Figure 14(b), initArc(node1,tr)=tr1.

3.3.5 Agent and Activity Agent

There are two kinds of agent in the ASM model. The universe AGENT denotes all the abstract set of agents which move through the extended ASM diagram. In the following, "agent" always refers to an element in AGENT. The universe AGENT_ACT denotes agents which model an execution of an activity associated with a node. An activity agent is always created by an agent in AGENT and their relation is denoted by a function $Agent2Act$: AGENT $\times$ NODE $\rightarrow$ AGENT_ACT. A function $IsActAgent$: AGENT_ACT $\rightarrow$ Boolean indicates whether an agent belongs to the universe AGENT_ACT. An agent in the universe AGENT can create an activity agent in the universe AGENT_ACT for a given node by the following macro shown in Figure 19. $PerformActivity$ is another module used to execute the activities in a state machine diagram.

```
CREATE_ACT(node):
    extend AGENT with a
        Agent2Act(Self,node):=a;
        Mod(a):=PerformActivity(node);
    endextend
```

Figure 19: The macro definition for CREATE_ACT(com)

Figure 20 shows the general structure of the program executed by all activity agents whose details we are not interested in. Therefore we just give how it can trigger a completion event. An activity, which is composed of a set of actions, is also mapped into a list of ASM specifications. For a normal execution, at the end of an activity associated with a node, we add an additional ASM specification to set a function $HasGenEvent : ARC \rightarrow Boolean$ to be true, triggering complete events associated with a node. And we set function $act\_done : Boolean$, which is used to indicate whether an activity agent is done, to be true. All these are shown in Figure 20. In addition, $node_i, i \in \{1, 2, \ldots, n\}$ in Figure 20 denote all the nodes in an extended ASM diagram.

[10]We can regard $node_i$ as a composite node containing all the nodes within the region $node_i$.

```
PERFORMACT(node):
if $node=node_1$ then
            ⋮
        if $act\_done = true$ then
            do forall $ar \in NormalevList(node1)$:
                $HasGenEvent(ar):=true;$
            enddo
            $act\_done := false;$
            $Mod(Self) := undef;$
        endif
elseif $node=node_2$ then
            ⋮
endif
```

Figure 20: The ASM specification for an activity agent.

3.3.6 Rules of extended ASM diagrams

In order to represent the current arc the agent a's control lies in, a function $CurArc : AGENT \rightarrow ARC$ is defined and its updates will reflect the movement of the agent during the execution.

We divide the execution of an agent into the following different modes MODES={node, arc, interrupt, suspended, undef}. Mode *node* denotes an agent is executing a node; Mode *arc* indicates an agent is executing an arc. Mode *interrupt* indicates an agent is interrupted by some event. Mode *suspended* indicates an agent is suspended by its creator agent because of an interrupt. Mode *undef* indicates an agent does not exists any more. Function $CurMode : AGENT \rightarrow$ MODES is used to indicate current mode an agent is at.

For each node, its execution (by an agent) can be divided into three different phases. A function Phase:$NODE \rightarrow \{init, internal\_exe, wait\_for\_exit\}$ is used to indicate which phase a node is at. Before an agent's control reaches a node, it is in phase *init*. When control reaches it and entry action is being executed, the node enters phase *internal_exe*. During *internal_exe*, when the internal transition and activities start to execute, the node enters the third phase *wait_for_exit*. When the exit action associated with the node starts to execute, the node's phase is reset back to *init*. The only exception to this is a sequential composite node which is exited when a final node within that node is reached. A function $IsFinal : NODE \times NODE \rightarrow Boolean$ indicates whether a node is a final node in anther node. For example, $IsFinal(D_1, D_2) = true$ means that D_1 is a final node in D_2.

When an agent is to execute an arc, all the nodes' exit and enter actions affected by this arc are to be executed. A function $NestStructure : AGENT \rightarrow NODE^{\star}$ is used to indicate a chain of nodes whose exit actions are to be executed next. In addition, during an agent execution, a function $CurNode : AGENT \rightarrow NODE$ represents a node which is possibly interrupted by some event associated with an arc coming out from that node.

An agent is created by the following macro shown in Figure 21. All the children agents created by an agent a are denoted by function ChildAgent(a)=$\{a' \in$ AGENT $|$ parent(a')=a$\}$. a is called a parent of these child agents.

```
CREATE_AGENT(agent,node,arc):
CurMode(agent):=node;
CurArc(agent):= initArc(node,arc);
parent(agent):= Self;
Mod(agent):=Mod(Self);
```

Figure 21: The macro definition for CREATE_AGENT(agent,node,arc).

To kill an agent, we set several functions to "undef" shown in Figure 22.

```
KILL(a):
Mod(a):=undef;
CurMode(a):=undef;
parent(a):=undef;
```

Figure 22: The macro definition for KILL(a).

When an agent needs to execute an entry action associated with a node, we give the following macro in Figure 23. Besides executing the entry action, we assign Phase for that node to *internal_exe* because the entry action is executed during phase *init.* In addition, function $GName : Guard \rightarrow STRING$ is used to get a string name for a guard in an arc. And function $Concat : STRING \times STRING \rightarrow STRING$ is used to append the second parameter to the first parameter to produce a new string name. Function $OriginalGurd : Guard \rightarrow Boolean$ is used to denote whether a guard is a original one or not. If it is an original one, we set the corresponding guard condition to be true so that the new corresponding arcs associated with this arc will be fired. Function $Name2G : String \rightarrow Guard$ is used to get a guard from a name.

When an agent is about to exit from a node, we need to update some functions for this agent. These updates include to set functions $CurArc, CurMode$ and $NestStructure$ to corresponding new values. These updates are showned in Figure 24.

The following Figure 25 gives a definition for a marco EXE_EXIT_OUTWARDS(List) which is used to execute one exit action from the head of List. In addition, if a node execution is

```
EXE_ENTRY_ACTION(node):
let gname = GName(guard(node)) in
Action2ASM(entry(node));
Phase(node):=internal_exe;
if (OriginalGurd(guard(node))=true) then
      Name2G(Concat("Guard_",gname)) := true;
endif
endlet
```

Figure 23: The macro definition for EXE_ENTRY_ACTION(node).

```
EXIT_FROM_NODE(outArc,node):
CurArc(Self):=outArc(node);
CurMode(Self):=arc;
NestStructure(Self):=NestNodesFromArc(node,outArc(node));
```

Figure 24: The macro definition for EXIT_FROM_NODE(outArc, node).

caused by an arc which is newly generated in the ASM extended diagram, we should reset the guard condition to be false so that this arc can be fired later.

When an agent a is in mode *node*, indicated by the function $CurMode(a) = node$, we need the following macro $ISREACHABLE(arc)$ to check whether the target *node* associated with *arc* can be entered. The definition is given in Figure 26.

In addition, the macro ELIGIBLE2EXE(arc) is used for a completion arc to check whether the completion event is done and the guard associated with it is true. If both of them are true, then the completion arc is about to happen. This is shown in Figure 27.

First we consider the case when an agent's control reaches a simple node which is of the form: Node(name,entry,exit,internal,activity,$inArc_1, \ldots, inArc_n, outArc_1, \ldots, outArc_m$). If the guard condition associated with the current arc $CurArc$ is true, the node is a simple node and the node's phase is *init*, then besides the actions defined in EXE_ENTRY_ACTION we assign $CurNode$ to the current node. This is shown in Figure 28.

When a simple node is entered by an agent, we create a new agent to execute the activity defined in this simple node. The node then enters the last phase, waiting for a normal exit.

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[11]. If there are more than one completion arc which is eligible to fire, a function $ChooseArc$

[11]All the arcs shown in Figure 30, 33 and 36 are the completion arc.

```
**EXE_EXIT_OUTWARDS(List):**
let gname = GName(guard(node)) in
Action2ASM(exit(head(List)));
List := tail(List);
Phase(head(list)):=init;
do forall ev ∈ NormalevList(head(List))
        HasGenEvent(ev):=undef;
enddo
if (OriginalGurd(guard(node))!=true) then
        Name2G(Concat("Guard_", gname)) := false;
endif
endlet
```

Figure 25: The macro definition for EXE_EXIT_OUTWARDS.

```
ISREACHABLE(arc)=
CurMode(Self)=node and guard(arc)=true
```

Figure 26: The definition for ISREACHABLE.

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

Figure 27: The macro definition for ELIGIBLE2EXE(arc).

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

Figure 28: The initial phase for a simple state.

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

Figure 29: The internal phase for a simple state.

returns a highest priority arc, which is about to execute. In order to execute the exit acitons along the arc, a function *NestStructure* is set for this agent. The phase for the agent is also set to *arc*. And function *CurNode* is set to *undef* in that the agent is about to leave from that node.

```
let node = targetstate(CurArc) in
      if ISREACHABLE(CurArc(Self))=true and IsSimple(node)=true and
            Phase(node)=wait_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
```

Figure 30: The last phase for a simple node.

But the semantic becomes complicated when a node is not a simple node. For a sequential composite node, we assume it has the form: Node(Name, entry, exit, internal, activity, deferred, *subnodes*, *inArc*, *outArc*), where *subnodes* represents all the immediate subnodes which are composed of the sequential composite node. When the guard associated with the current arc is true and the composite node's phase is *init*, then the entry action is to be executed and the phase for that node is set to *internal_exe*. This is shown Figure 31.

After executing the entry action defined in the composite node, the agent creates a new activity agent to execute the activity and then sets *CurArc* to the arc given by function *initArc* and *CurMode* to *node* so that a subnode of that composite node is about to execute in the next.

```
let node = targetstate(CurArc) in
    if ISREACHABLE( CurArc) and Phase(node)=init and IsCompSeq(node)=true then
        EXE_ENTRY_ACTION(node);
    endif
endlet
```

Figure 31: The initial Phase for a sequential composite node.

This is shown in Figure 32.

```
let node = targetstate(CurArc) in
    if ISREACHABLE( CurArc) and Phase(node)=internal_exe and
            IsCompSeq(node)=true then
        CREATE_ACT(node);
        CurArc(Self) := initArc(node, CurArc);
        CurMode(Self) := node;
        Phase(node) := wait_for_exit;
    endif
endlet
```

Figure 32: The internal phase for a sequential composite node.

Now we consider how to exit from a sequential composite state. Here we consider a normal exit and we will discuss an abnormal exit caused by some event late. When an agent reaches the final state of a sequential composite state, and a completion arc coming out from the composite state is eligible to fire, we will execute the exit and entry actions affected by the arc. If the composite node is $TOPNODE$, it means that we have finished the execution for the extended ASM diagram, otherwise we execute the actions decided by the arc, using the macro EXIT_FROM_NODE. This ASM specification is shown in Figure 33.

Now we consider a concurrent composite node. A concurrent composite node is of form: Node(name,entry,exit,internal, activity, deferred, $inArc_1, \ldots, inArc_n$, $nodes_1, \ldots, nodes_n$, $outArc_1$, $\ldots$, $outArc_m$). When a composite concurrent node is entered by an agent whose phase is *init*, besides the actions defined in EXE_ENTRY_ACTION, we set function $CurNode$ to the current node. This is shown in Figure 34[12].

During the phase *internal_exe*, apart from creating a new activity agent as we have shown

[12]We can combine Figure 34 and Figure 28 into one if we add condition $IsCompConcur(node) = true$ in Figure 28.

```
let $node$ = $targetstate(CurArc)$ in
    if $ISREACHABLE(CurArc)$ and $Phase(node)=init$ and
                $IsFinal(node, UpNode(node))=true$ then
      if $IsCompSeq(UpNode(node))=true$ then
        if $UpNode(node) = TOPNODE$ then
            $CurMode(Self) := undef;$
        else
          let $i=ChooseArc(node)$ in
            if $(i!=undef)$ then
              $EXIT\_FROM\_NODE(outArc_i, UpNode(node));$
            endif
          endlet
        endif
      endif
    endif
endlet
```

Figure 33: The last phase for a final state in a composite sequential node.

```
let $node$ = $targetstate(CurArc)$ in
    if $ISREACHABLE(CurArc)$ and $Phase(node)=init$ and $IsCompConcur(node)=true$ then
        $CurNode(Self) := node;$
        $EXE\_ENTRY\_ACTION(node);$
    endif
endlet
```

Figure 34: The initial phase for a concurrent composite node.

above, we create a new agent for every region within that concurrent composite node to execute nodes in that region. This is shown in Figure 35.

```
let $node$ = $targetstate(CurArc)$ in
    if $ISREACHABLE(CurArc)$ and $Phase(node)=internal\_exe$ and
            $IsCompConcur(node)=true$ then
        $CREATE\_ACT(node)$;
        extend $AGENT$ with $c_1, \ldots, c_n$
            $CREATE\_AGENT(c_i, nodes_i, CurArc)$;
        endextend
        $Phase(node) := wait\_for\_exit$;
    endif
endlet
```

Figure 35: The internal phase for a concurrent composite node.

In the following we consider a normal exit from a concurrent composite node. Assuming an agent is waiting for a normal exit, if all its child agents are in their final states and a completion arc is eligible to fire, then the agent kills all its child agents and execute the actions defined in macro EXIT_FROM_NODE. This is shown in Figure 36.

Now let us consider the abnormal exit. During an agent execution, if an event occurs, which belongs to *evList* associated with the node the agent lies in, the agent's mode is set to *interrupt* immediately, indicating to stop its normal execution.

When an agent enters the mode *interrupt*, it needs to stop all the activities being executed and start to execute all exit actions in the nodes which the arc, whose event occurs, comes out from. There is no difference in executing the exit actions caused by either normal or abnormal exit. So we give one ASM specification in Figure 38 for both cases.

If a node, whose exit action is be executed next, is a composite concurrent node, then we need to distinguish two cases. One is the execution for the exit action is caused by an event belonging to that concurrent composite node and that composite node's exit action is to be first executed during the (abnormal) exit. The other is the execution for the exit action is caused by the pass from its immediate subnode.

If the first case occurs, the agent needs to wait for all its child agents execution for their exit actions to be done. If all the child agents finish their exit execution, then the agent kills all of its child agents (if exists) 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.

```
let $node$ = $targetstate(CurArc)$ in
    if $ISREACHABLE(CurArc)$ and $Phase(node)=wait\_for\_exit$ and
                $IsCompConcur(node)=true$ then
      if $\forall a_i \in SubAgent(Self) : IsFinal(targetNode(CurArc(a_i)),node)=true$ then
        let $i=ChooseArc(node)$ in
          if $(i!=undef)$ then
            do forall $a_i \in SubAgent(Self)$
                $KILL(a_i);$
            $EXIT\_FROM\_NODE(outArc_i,node);$
            enddo
          endif
        endlet
      endif
    endif
endlet
```

Figure 36: The last phase for a concurrent composite node.

```
if $occured(ev)$ and $ev \in evList(CurNode)$then
    $CurMode(Self) := interrupt;$
    $CurArc(Self) := event2arc(ev, CurNode);$
    $NestStructure(Self):=NestNodesFromArc(CurNode,arc(ev));$
endif
```

Figure 37: The model for an event occurs.

```
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; #agent stops exit execution
                Mod(Agent2Act(Self,head(NestStructure(Self))):=undef;
                CurMode(Self):=suspended;
            else
                if ∀a ∈ ChildAgent(Self): CurMode(a)=suspended then
                    EXE_EXIT_OUTWARDS(NestStructure(Self));
                    ∀c ∈ ChildAgent(Self): KILL(c);
                endif
            endif
        endif
    endif
endif
```

Figure 38: The model for executing an arc.

3.4 Verification Tool

The Verification Tool is the major tool in the verifier. We single out the SMV among all other methods as the following reason.

First SMV is also based on the transition systems. All the transitions internally are regarded as binary decision diagrams which leads to very efficient algorithms. The SMV checks a temporal logic formula against the system specification and outputs a counter-example if the system fails to meet the requirement.

Secondly, a translation tool from ASM to SMV has been implemented by a group in Germany [2]. We hope the verifier can take the advantage of the existed tool. In the following, let us take a look at the transformation schema from an ASM Model to a SMV model, showing why this schema can be applied to our verifier.

Kirsten Winter [12] reported the work about this transformation schema which covers the most ASM structures we will be using in this verifier. All these transformation schemas are shown in the following.

Update instructionR : $f(\bar{t}) := t_0$ will be translated into

$$ASSIGN next(l) := y$$

with the location $l = (f, Val_S(\bar{t}))$ and the value $y = Val_S(t_0))$

Guarded transition rule if g_0 then R_0 elseif g_1 $\ldots, \ldots$ elseif g_k then R_k endif is translated into the following:

$$\begin{array}{l} ASSIGN next(l) := \\ \qquad case \\ \qquad\qquad g_0 : y_0; \\ \qquad\qquad\quad \vdots \\ \qquad\qquad g_k : y_k; \\ \qquad\qquad 1 : l; \\ \qquad esac; \end{array}$$

where location $l = (f, Val_S(\bar{t}))$and values $y_i = Val_S(t_i)$ for all i.

The SMV language allows modules to be used as a unit to run in parallel. If we instantiate a module with the keyword `process`, then the semantics is running of interleaving concurrency: on two modules with the same parent run at the same time. However if `running` is used in a parameter of an instance of a module, then that module is always running when the parent module is. This is *true concurrency* in the sense of *simultaneous execution*.

In ASMs, agents and all the transition rules inside an agent are running concurrently. So in [12] all agents in ASM are translated into SMV modules that are instantiated without the keyword `process`, meaning a true concurrency is generated. In order to instantiate all modules, a module `main` is initially generated. All modules are running whenever the module `main` is running.

3.5 Analysis Tool

Because users of this tool are those engineers of object-oriented software systems, all the details about the implementation of this tool are hidden from the users. The analysis tool is used to build a bridge between these two aspects so as to make it easier for the engineers to use in the industry world. The analysis tool accepts the result from the verification tool and it returns some UML diagrams to the users if the verification tool finds some errors.

One of the diagrams which will return to the engineers is the sequence diagram in UML. The reason that we consider to represent the error is that it not only shows the time dimension but also gives the configuration for message exchanges. It is more easier for the engineers to find the problem in their design.

Using the collaboration diagram to represent errors in the design model is another way we are considering. Although the collaboration diagram represents the message exchange, it concentrates on the links between the objects. When a user can not find the problem in his design from the sequence diagram, he can also look at the collaboration diagram to investigate the problem possibly occurring among the objects.

Although a sequence diagram is equivalent to a collaboration diagram in UML, we provide the both two diagrams to satisfy the different users' preference in this tool.

4 A Toy Example: Elevator Problem

In this section, we show how the tool veriUML we will be building works when an UML model is given. Then we give a conclusion about this project.

4.1 Elevator Problem

First let us take a look at the elevator problem.

1. A product is to be installed to control elevators in a building with m floors. The problem concerns the logic required to move elevators between floors according to the following constraints: Each elevator has a set of m buttons, one for each floor. These illuminate when pressed and cause the elevator to visit the corresponding floor. The illumination is canceled when the elevator visits the corresponding floor.

2. Each floor, except the first floor and top floor has two buttons, one to request a up-elevator and one to request a down-elevator. These buttons illuminate when pressed. The illumination is canceled when an elevator visits the floor and then moves in the desired direction.

3. When a request is pressed at any certain floor, then the request will be served eventually.

4. When an elevator has no requests, it remains at its current floor with its doors closed.

In general, the design phase should produce the details about class diagrams, collaboration diagrams, sequence diagrams, statechart diagrams and activity diagrams. Here because of simplicity of this problem, we skip the activity diagram for the elevator problem.

4.2 A UML Model for Elevator Problem

Class diagrams show the static structure of the objects, their internal structure and their relationships. From the problem description, we know that there are following classes in the class diagram: Elevator, Ele_Controller, Button, Elevator_Button and Floor_Button. The class Ele_Controller controls the class Elevator. And the class Ele_Controller controls one Elevator. There are m Button classes associated with the class Ele_Controller. And the Class Button has two subclasses: Elevator_Button and Floor_Button. The class diagram is shown in Figure 39.

In the following we give the collaboration diagram for the elevator problem. Here to simplify the problem, we just give the collaboration diagram for serving door button. In general, control flows among the objects in the following. When a passenger at floor 3 presses the up button. The object floor3 sends a message to the object ele_controller to update a requested. The object ele_controller sends a message, like move, to the object elevator to move to the floor 3. When the elevator reaches floor 3, it sends a message *arrived* back to the object ele_controller. So the collaboration diagram for serving door button is shown in Figure 40.

Now we consider statechart diagrams for these objects. Statechart diagrams show a state machine which models the dynamic aspects of a system. It concentrates on the flow of control from state to state, so especially useful in modeling the lifetime of an object. First we consider the statechart diagram for the object ele_controller. There are two states in the object ele_controller. One is `waiting` and the other state is `working`. In state `waiting`, the object sits at there, waiting for a message possibly from the object floor_m, elevator and door. After receiving a message, the object ele_controller enters state `working` which means that it will send a message to the corresponding object according the message it received. This is shown in Figure 41.

Now we consider the statechart diagram for the object elevator. There are two states in it, `Idling` and `Moving`. In both states, the object receives the message `move` from the object ele_controller. After receiving this message, the object elevator always stays at state `Moving`. When the elevator reaches the floor, it sends a message `arrived` back to the controller. The Figure 42 shows the statechart diagram for object elevator.

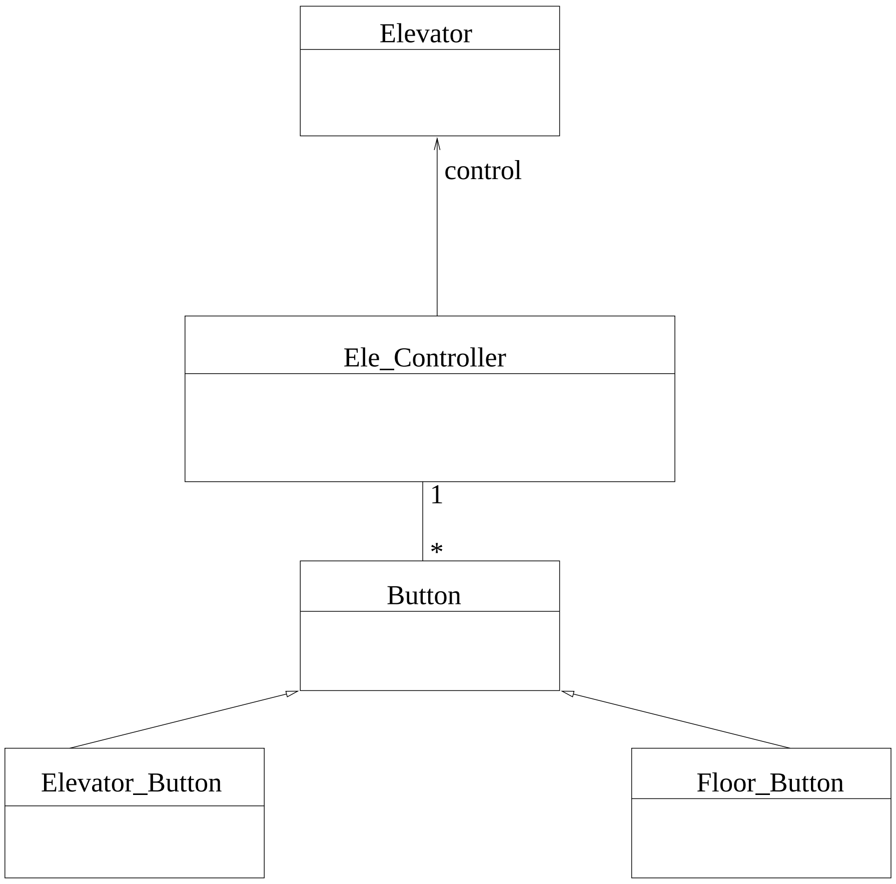


Figure 39: A class diagram for the elevator problem.

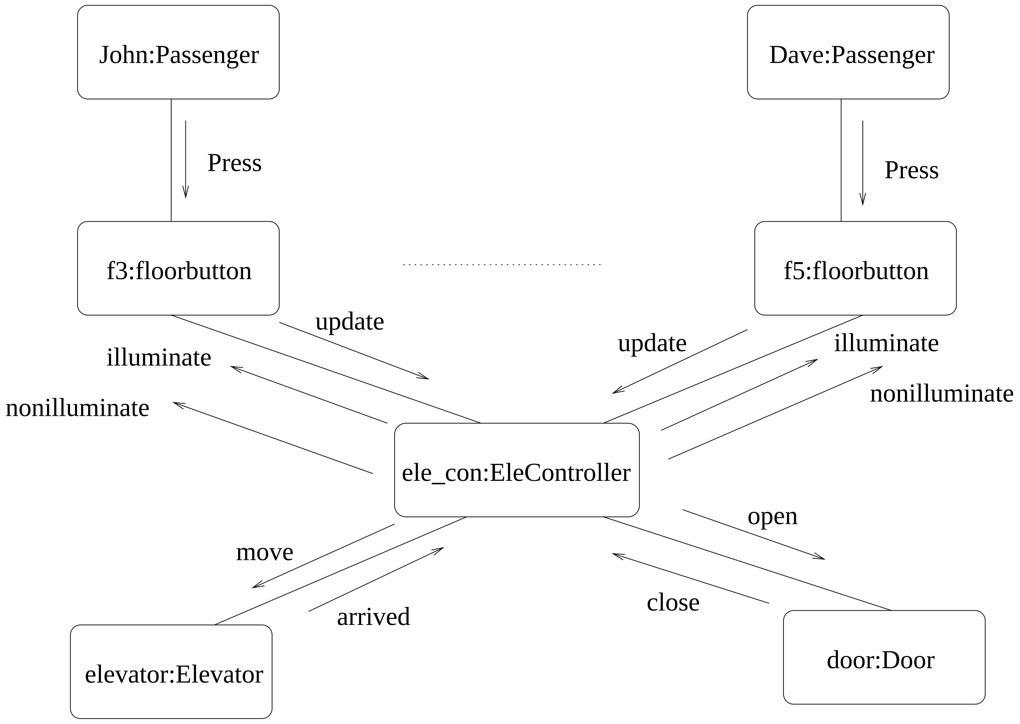


Figure 40: The collaboration diagram for the elevator problem.

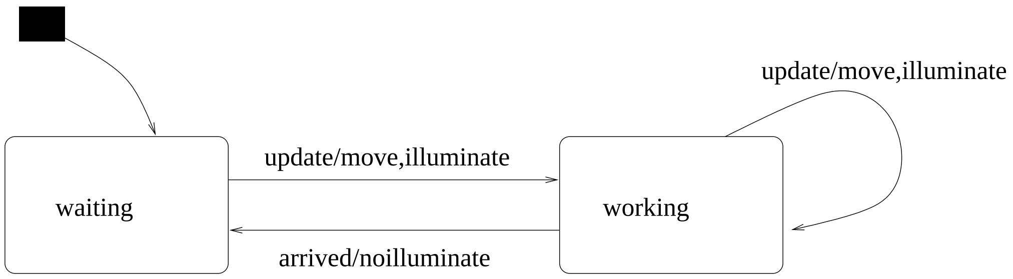


Figure 41: The statechart diagram for object ele_controller.

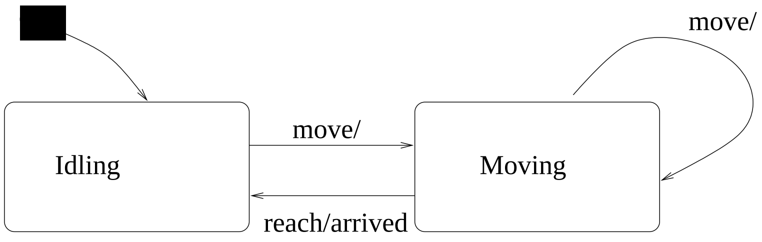


Figure 42: The statechart diagram for object elevator.

For the other parts in this model, we skip the diagrams for them. Additionally, designers should provide a specification which should be met by the model. For this elevator problem, designers can provide the following temporal logic formula as a specification:

$$\Box(press(x) \Rightarrow \Diamond open(x))$$

But in the above model combined with the above specifiction, a careful reader must have noticed a problem. If John presses the button on the third floor first, then the elevator moves to that floor. But during that movement, another passenger Dave presses the button on the fifth floor. The elevator control receives this message and updates its control system and sends a message to the elevator. The elevator receives the new command and moves to the fifth floor during its movement to the third floor. More worse, the elevator does not have the function to remember the previous commands. Therefore, the third floor will be never arrived until John presses the button again later. Obviously, it violates the requirement for the elevator problem.

Whenever an error is detected in SMV, it generates a trace showing how to produce the error. The tool analyses the error trace and represent it as a UML diagram which is returned to the designer. Figure 43 shows an error where the third floor will not be reached.

5 Conclusion

In this project as first step, we give the semantics for a state machine in UML. At the same time a group led by Prof. Egon Borger etc. was also working on an Abstract State Machine model for the State Machines. As first step they gave an ASM model for activity diagram in UML[3], the author read this paper and some of ideas are borrowed from that paper. But there are still a lot of work to do when giving an Abstract State Machine model for the state machine in UML because the difference between the activity diagram and state machine.

One of the major differences between the two diagrams is that both the source and target node for a transition in the state machine in UML may not be on the same level. This difference

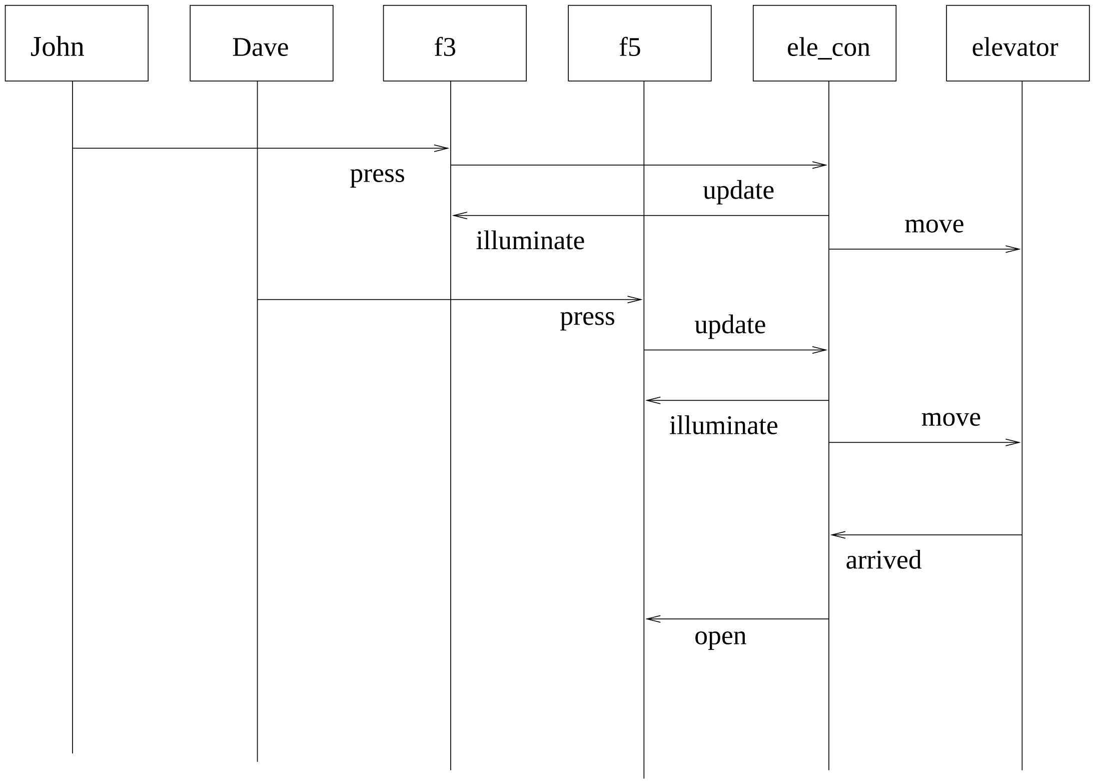


Figure 43: A sequence diagram for a counter-example.c

results in changes for modeling an interruption. To deal with the change, we extend the state machine diagram in UML to an extended Abstract State Machine diagram by adding some new arcs. Therefore it is easy to deal with an interruption whose source and target states are possibly at the different level.

Additionally, in the Abstract State Machine Model for the state machine in UML, all agents sit at either a simple node or composite concurrent node to wait for a possible interruption. This simplifies the interruption model for the state machine in UML. However, [3] does not have this in their model.

The other difference is that an activity can occur in any state in the state machine diagram in UML; however this is not allowed in the activity diagram in UML. To model its execution, we create a new agent to execute the activity associated with a state.

Last in the ASM model given in the above, we give more details about the implementation of some functions; for example, how to compute the functions like $NestStructure$. That is why we need to derive a ASM extended state machine diagram from the original one.

When we finished writing this proposal, we found another similar work related to our tool which is vUML[9]. That tool is being worked by a group in Finland. Although that tool is still being worked, most of the ideas between these two tools are quite similar. They receive UML diagrams as the input and return UML diagrams as the output if there exist some errors in the input specification. Similarly, they use another model checker tool, named SPIN, to verify some properties.

Because they use SPIN to verify the properties, they translate the UML diagrams into PROMELA, an input language of SPIN. But PROMELA is not a formal specification language which can be applied to model UML diagrams. Therefore, they use operational semantics to give the semantics model [10]. They separate a model from its verification.

However, in this project, we first give a semantic model for UML diagrams by using Abstract State Machines; then we can verify some properties from an Abstract State Machine model. Abstract State Machines connect a model to its verification, which shows the strength when using Abstract State Machines. It is this strength that distinguishes Abstract State Machines from the other methods, including the method used in vUML.

References

[1] Egon Börger. Integrating ASM into the Software Development Life Cycle. Journal of Universal Computer Science, vol. 3. no. 5(1997), p. 603-665.

[2] G. D. Castillo, K. Winter. Model Checking Support for the ASM High-Level Language. Technical Report tr-ri-99-209, Universität-GH Paderborn, June, 1999.

[3] Egon Börger, A. Cavarra, E. Riccobene. An ASM Semantics for UML Activity Diagrams.

[4] A. Davis. Software Requirements Analysis and Specification. 1990.

[5] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E.Börger, editor, Specification and Validation Methods, pages 9-36. Oxford University Press, 1995.

[6] 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.

[7] B. Kernighan, W. Brian, The C programming Language, 2nd edition, Prentice Hall, 1988.

[8] P. Kutter and A. Pierantonio. Montages: Specifications of Realistic Programming Languages. Journal of Universal Computer Science, 3(5):416-442, 1997.

[9] J. Lilius, I. P. Paltor. vUML: a Tool For Verifying UML Models, TUCS Technical Report No. 272, May 1999.

[10] J. Lilius, I. P. Paltor. Formalizing UML state machines for model checking, TUCS Technical Report No. 273, June 1999

[11] K. McMillan. Symbolic Model Checking. Kluwer Academic Publishers. 1993.

[12] K. Winter. Model Checking for Abstract State Machines. Journal of Universal Computer Science, vol. 3. no. 5(1997), p. 689-701.