A Method of Implementing UML Virtual Machines With Some Constraints Based on Abstract State Machines

Wuwei Shen
Department of Computer Science
Western Michigan University
wwshen@cs.wmiche.du

Kevin Compton
Dept of EECS, University of Michgan
kjc@eecs.umich.edu

James Huggins
Computer Science Program, Kettering University
jhuggins@kettering.edu

Abstract

UML has become a standard language for designing software systems. To help software developers design a correct UML model for a software system has become an important goal for many UML CASE tools. In this paper, we propose a new UML virtual machine based on Abstract State Machines. We combine the UML meta-model, UML model and user objects model into one under the ASM virtual machine for UML. Since the ASM virtual machine for UML supports OCL, software developers can precisely design a software model and then find some errors such as inconsistency in the model with the help of the ASM virtual machine.

1. Introduction

The Unified Modelling Language (UML) [12] has become a standard notation for software system modeling, mostly because the same conceptual framework and the same notation can be used from requirement and specification through design to implementation. One of the main advantages of UML is that it incorporates modeling notations to represent different views of a software system. However these different views given by different UML diagrams can result in some errors in a model.

Errors in the early software development phases using UML fall into the following two categories. The first is a violation of the UML meta-model. As an instance of the UML meta-model, a software model given by a set of UML diagrams should satisfy all restrictions in the UML meta-model. The restrictions consists of graphical restrictions, given in class diagrams in the meta-model, and well-formedness rules, which are written in the Object Constraint Language (OCL). A model given by a software designer is said to be syntactically correct if it satisfies not only the syntax given in class diagrams in the UML meta model but all the well-formedness rules as well. Any violation of the syntax or well-formednesss rules should be regarded as an error and therefore reported to a software designer.

The second kind of errors is an inconsistency between a software model an its corresponding object model (an instance of the software model). When a software designer designs a software system, (s)he first gives a model by presenting some UML diagrams and some constraints associated with these diagrams. At the same time the designer should have some reasonable snapshots related to this model. The snapshots are usually denoted by UML diagrams related to objects such as object diagrams, representing a state in a software model. Sometimes some important snapshots are not included in the model, i.e. they do not satisfy the constraints given in a UML model. Helping software designers to find this kind of exclusion is also a goal for our ASM virtual machine. If an exclusion of some reasonable snapshot is detected, then the software designer should redesign the model to contain it. In addition, a designer can test whether a undesirable system state is included in a model being designed. If included, the designer also should redesign the model to delete the undesirable state.

Thanks to the four-level architecture of the UML model system, we can find errors in a UML model and its corresponding specific information model under the ASM virtual machine. A model given by class diagrams in UML can be mapped to the ASM virtual machine. Combined with support of OCL, our ASM virtual machine can map all models in the last three levels to ASM specification. Consequently, software developers can compare any two consecutive level

1530-1362/03 $17.00 © 2003 IEEE

models, trying to find any error in a lower level model based on its immediate upper level model in the UML model system. Thus the ASM virtual machine provides short feedback cycles and makes UML model prototyping sufficient so that a software developer can find errors in his/her model as quickly as possible.

Furthermore, Abstract State Machines, as a formal method, have been applied in many software and hardware systems [6, 4, 5] etc.. Unlike some other formal methods, ASMs are easy to learn and write. Microsoft is investing time and labor in an environment supporting ASMs [10]. Due to this reason, we decide to build an ASM virtual machine for UML. This virtual machine can provide not only a semantic model for UML but a direct executable environment for UML as well.

The content of this paper is structured as follows. Section 2 introduces the architecture of the UML model system. Section 3 introduces Abstract State Machines. Section 4 gives some implementation issues about the ASM virtual machine. Section 5 show some results when we apply the ASM virtual machine to some UML models. Section 6 compares our virtual machines with others and draws some conclusion about this work.

2. The Architecture of the UML Model System

Like a programming language, we need some rules to represent whether a UML model is a syntactically correct software system model. The Object Management Group (OMG) defines UML in a four-layer meta-model architecture, called the M0, M1, M2 and M3-level. This architecture provides a mechanism to let software developers design a correct model for a software system. Furthermore, this architecture makes it possible to check whether a model in one level is a valid instance of its immediate upper level model. Figure 1 gives the four-level modelling architecture. We give more explanations about the four-layer meta-model architecture in the following.

1. M0-Level: In this level the user objects, such as *smith*, are given. The objects in this level represent an instance of a M1-level model and they define a specific information domain.

2. M1-Level: In this level all classes defined by a developer are given. All these classes and the relations between them give a language to describe an information domain. They give a model for a software system.

3. M2-Level: In this level all objects are used to define a language for specifying a modeling language. This level is also called the meta-model level.

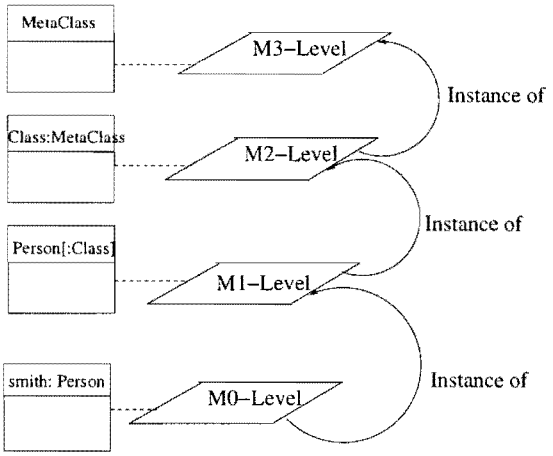

Figure 1. The relation between the four-level modeling architecture

4. M3-Level: In this level all objects are used to define a language for specifying a meta-model. This level is also called the meta-meta-model level.

In the UML definition, the M3-level is used to describe the UML meta-model (M2-level), the M2-level (UML meta-model) is used to describe a software system model (M1-level) and the M1-level (UML model) represents a running system which is an M0-level. The four-level structure provides us a mechanism to define and understand a software system given in UML .

For example in Figure 1, a software developer can define a user class `Person` (an M1-Level object) which is an instance of object `Class` in the M2-level. Because the user class `Person` is an instance of the M2-level object, it must satisfy all constraints given for the object `Class` in the M2-level. Similarly, at run-time, the user class can be instantiated and some user objects (like *smith*) can be generated. These objects are M0-level objects which satisfy all the constraints given for the user class in the M1-level.

Due to the above four-level architecture, any changes in one level can cause the next lower level model to change accordingly. In most UML CASE tools like Rational Rose, a software developer first uses some editor to define user classes which are M1-level objects and some relations among these classes. Finally a software system can be designed by the use of a UML CASE tool. This software system is actually an instance of the M2-level model.

Using constraints in software systems has become more and more important. Graphical notation cannot be enough to completely represent a software system. Constraints can not only provide restrictions on a system but also be used as

assertions in the software system. These assertions can be placed in important locations in a software system to provide a mechanism to deal with unexpected events. In short, constraints play an important role in improving software quality. However, most of UML CASE tools do not support OCL. Therefore most of them cannot check all syntax errors in a UML model.

3. Abstract State Machines and Its Virtual Machine for UML

Abstract State Machines were first presented by Prof. Yuri Gurevich [11] more than ten years ago. ASMs have been widely used in specifying and verifying many computer systems.

Before introducing the ASM virtual machine, we give some definitions related to ASMs. The *signature* of an ASM $\mathcal{A}$ is a finite collection of function names, each name having a fixed arity. A *state* of $\mathcal{A}$ 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 $\mathcal{A}$ 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 $\mathcal{A}$ 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

  $$\begin{array}{l} if \quad g \quad then \quad R_1 \\ else \quad R_2 \\ endif \end{array}$$

  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

  $$\begin{array}{l} do \quad in-parallel \\ \qquad R_1 \\ \qquad R_2 \\ enddo \end{array}$$

  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

  $$\begin{array}{l} do \quad forall\ v : \quad g(v) \\ \qquad R_0(v) \\ enddo \end{array}$$

  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 $g(a) = true$. To fire R at A, fire simultaneously all these $R_0(a)$ unless they are mutually inconsistent. In the latter case, do nothing.

In applications, a program is usually a block of rules referred to as rules of the program. Intuitively, a run is a sequence of states where each state is obtained deterministically from its predecessor by executing the updates described by the program. But algorithms need not be deterministic; they may depend on input from the outside world which is effectively non-deterministic. One mechanism for representing nondeterminism is through *external* functions. The intuition is that external functions have their values determined by an oracle outside of our control, who may change the value of such a function at any time. We require an external function have consistent values within a given state; it may, however, change arbitrarily between states.

Our notion of run restricts attention to non-external functions. A *run* of a program $\mathcal{P}$ is a sequence states $S_0, S_1, \ldots$ where each S_i is obtained from S_{i-1} by firing $\mathcal{P}$ at S_{i-1}.

ASMs have been applied in UML in many ways. Börger et al have applied ASMs to provide semantics for UML activity diagrams and state machines ([3] and [2]). A simulation tool for UML statecharts [7] was reported.

4. Implementation of the ASM Virtual Machine for UML

As an implementation of the ASM virtual machine for UML, we choose XASM [1] as our ASM specification language. The ASM virtual machine accepts a UML model which is using XMI, the OMG standard for representing UML models using XML [13]. Due to this reason, the ASM virtual machine can accept a UML model edited by any UML CASE tool which can provide the translation into XML.

The advantages of the ASM virtual machine are as follows:

1. Efficiency: ASMs are a formal method which can be executed. The ASM virtual machine not only gives the semantic model for any level in the UML model system but also provides a means to check whether one level model is a correct instance of the next upper level model by running the ASM virtual machine. Unlike most of the UML CASE tools, the ASM virtual machine combines the semantic, design and runtime environment into one; it thus requires less time to run a model. Therefore the ASM virtual machine makes rapid UML model checking possible and efficient.

2. Precision: Although ASMs are a formal method, ASM specification uses an extremely simple syntax, which can be read even by novices as a form of pseudo-code. It is pretty easy to check the precision of a system ASMs describe. Furthermore, the ASM semantics is represented by classic mathematic structures which are a well-understood, precise model. The precision of a system ASMs describe can also be formally checked.

According to the UML model system, the ASM virtual machine for UML consists of two parts. The first part gives the ASM specification for any two consecutive level models in the last three levels in the four-level architecture. This part maps any model in the two consecutive levels into ASM specification under one ASM virtual machine. This schema makes it possible to map the objects in different levels, such as the M1-level and M2-level, into ASM entities under the same environment, i.e. the ASM virtual machine environment, shown in Figure 2.

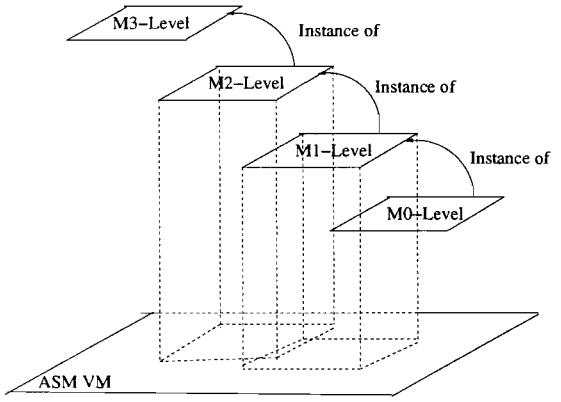

Figure 2. The relation between the UML four-level modeling architecture and the ASM virtual machine

The second part of the ASM virtual machine supports OCL. In fact, the UML meta-model (M2-level) is defined by a set of class diagrams and the OCL constraints which are associated with the M2-level objects. There are two kinds of constraints in the UML meta-model. One is given by graphical notations. For example, the multiplicity in an association is one kind of constraints, which restricts the number of its instance objects in the next lower level. The other kind of constraints usually cannot be represented by graphical notation. Instead, these constraints can be written in the Object Constraint Language. To support to check a model instance , it is very important for the ASM virtual machine to implement an ASM model for both the graphical notation and OCL.

The structure of the ASM virtual machine is given in Figure 3. It mainly consists of two modules. One module deals with the graphical models given in the Mi-level (i=1,2) and its next lower level. In this part, a set of ASM specification has been generated. The second module in the ASM

virtual machine is used to give ASM specification for the constraints given in a software system. Last, running all the ASM specifications generated from these two modules can give a software developer direct feedback about his/her model.

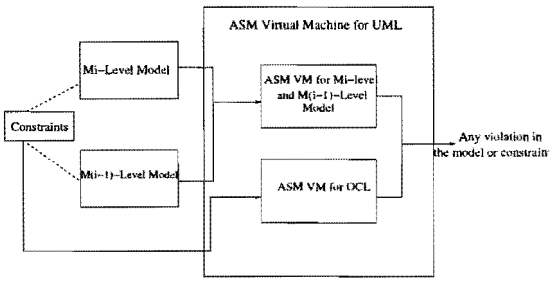

Figure 3. The structure of the ASM virtual machine

4.1 An ASM Model for UML Diagrams

The ASM virtual machine first describes the structure of the UML diagrams. To represent the structure of a Mi-level (i=1,2) model, we usually define universes to include all class names and associations in the model, declare functions to represent the corresponding attributes defined in a class, and declare functions to denote relationships between classes in the ASM virtual machine.

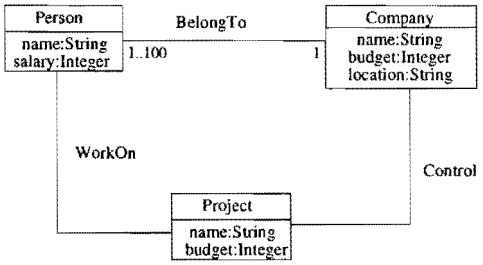

Figure 4. A UML class diagram

Based on the M1-level model (UML model) shown in Figure 4, we define a universe named Class which includes all class names in a class diagram in the ASM virtual machine. So we have the following ASM specification:

universe Class = {Company, Project, Person}

where *Class* is a universe name. For every class, there are attributes and method definitions. In the requirement analysis and design phase it is not necessary to consider method definitions, only attributes. Therefore, we define a new unary function for every attribute in the ASM virtual machine. The function name is the same as the attribute name except that the function name is written in capital letters. The unary function takes every object derived from the class as its parameter. For example, in the above class diagram, we have the following attribute function declarations for the class *Person*:

function NAME(x:Person) → String;
function SALARY(s:Person) → Int;

In the above ASM specification, the class name *Person* is used as a type because when the ASM virtual machine accepts an instance of the model it generates a universe for class *Person* which includes all objects derived from the class *Person*.

Besides the classes, a relationship between two classes can be represented by an association. An association is drawn as a line connecting two classes and the line is labelled with the name of the association, the multiplicities of the two classes (number of objects of the given class) participating in the association, and the roles that an instance of each class takes within the association. All of these labels are optional.

To give a flavor about the ASM specification for an association, we use the diagram shown in Figure 4 as an example. There are three association links in Figure 4. To represent an association link between the two classes, we first define a universe to include all association links in the diagram. For the association links in Figure 4, we have the following ASM specification:

universe Association = {BelongTo, Control, WorkOn}

If an association name is omitted, we use the class names related to the association to represent the association link as follows, assuming that *BelongTo* is omitted:

universe Association={BelongTo,Person_Company_Association, WorkOn}

Besides the association universe declaration, we define several functions to represent the association link between the two classes. For the association link *BelongTo* in Figure 4, the following ASM specification can be generated:

function Is_Association_Link(obj1:Class, link: Association, obj2:Class) → Boolean
Is_Association_Link(Person,BelongTo,Company):=true;

The above ASM specification gives the ASM specification for the association link *BelongTo* between the class *Person* and *Company*. However, sometimes the association name can be omitted and we can use the two associated classes' name to represent them as follows:

Is_Association_Link(Person,Person_Company_Assoication,Company):=true

Starting from a specific instance of a class (object), we can navigate an association on a class diagram to refer to other instances of a class and their properties. To represent this feature, we can declare a function as follows:

function objects_linked(x:Person,l:Association,y:Company) → Boolean

where *x* and *y* denote an object derived from class *Person* and *Company* respectively and *l* denotes the instance of an association link *BelongTo*. There are several ways to refer to other objects. One is to use a role name; when a role name is missing at one of the ends of an association, the name of the class at the other association end, starting with a lowercase character, is used as the role name. To implement the navigation on a class diagram, we declare a function returning boolean value in ASM for every role name and class name starting with a lowercase character in our ASM specification. The first and third parameters in the function denote two objects derived from the two classes at the two ends of an association and the second parameter denote the instance of the association link. Usually an association can be traversed in two directions; therefore two ASM functions are defined to represent this two-directional traversal. For the association *BelongTo* in Figure 4, we can define the following ASM specification:

function link_navigation(x:Company,l:Association,y:Person)→ Boolean
function link_navigation(y:Person,l:Association,x:Company) → Boolean

However in some cases, an association cannot be traversed by an object at either end of the association. To indicate unidirectional traversal, the line connecting the two classes is replaced by an arrow pointing in the direction of traversal. In that case, we skip the corresponding function declaration.

An important property for an association is the multiplicities defined in the association link. It gives the restrictions on the number of objects which can participate in the association. To represent this feature, we can declare some functions in the ASM virtual machine. For example, for the association *BelongTo* at the end of class *Person* in Figure 4 we can define the following functions:

function Low_Number(l:Association, x:Class) → Int
Low_Number(BelongTo, Person) := 1
function Upper_Number(l:Association, x:Class) → Int
Upper_Number(BelongTo, Person) := 100

In order to count the number of the objects generated, we should declare another ASM function as follows:

function current_num_association(x:BelongTo, l: Association, y:Person) → Int;

Besides the association, another important relationship between two classes is the generalization. By using the generalization relationship a child class can inherit some attributes or methods defined in its parent class and objects of the child class may be used anywhere the parent may appear. To represent this relationship, we also define a function as follows:

function GeneralizationRelation(x:ParentClass,y:ChildClass)→ Boolean

where the first parameter denotes the parent class and the second the child class.

4.2 An ASM Model for OCL

OCL can be used to represent constraints in a model. In order to support complete rapid UML model prototyping, we must provide ASM specifications for OCL in the ASM virtual machine. OCL consists of two parts: type and operation. The type consists of predefined types, including basis types, collection types, and user-defined model types. The user-defined model types are defined by the UML models. Each class or type in a UML model is automatically a type in OCL.

We translate the basic types in OCL into the corresponding XASM types. For the collection types and model types, we define a list structure containing all these elements in XASM.

Since the operations related to the basic types in OCL can be always found in their counterpart operators in XASM, we will consider ASM specifications for the collection-related types and their operations.

The collection-related types include `Set`, `Bag` and `Sequence`. The `Set` is the mathematical set, containing elements without duplicates. A `Bag` is a collection with duplicates allowed; one object can be an element of a bag many times. There is no ordering defined on the elements in a bag. A `Sequence` is a collection in which the elements

are ordered. An element may be part of a sequence more than once.

To deal with the operations related to the collection-related types, we use a `constructor` in XASM to represent all possible elements in the collection-related types. First we define a universe as follows:

universe List = {*nil*, *cons*{_,_}}

To give ASM specifications for specific collection-related types such as a model type, we need to give a function definition. This function's value is a list of elements.

After defining a function related to a collection-related type, we can deal with the operations on this type. But there are some differences among the collection-related types. We have slight changes in our ASM specification for these types. For example, when we deal with `Set`, the above function will be redefined so that all duplicate elements are eliminated. Then we can perform the collection-related operations on their operands. In the following we give an ASM specification for the operation `append` related to a `Sequence`; others can be derived in a similar way.[1] The syntax for `append` is as follows: `l->append(ele:T): Sequence(T)`. The meaning for this operation is that the operation returns a sequence of elements consisting of all elements of *sequence l* followed by *ele*. Figure 5 shows the ASM specification for `append`.

```
asm append(ele, l) -> List
   used as function in MAIN
is
   functions result -> List, s -> List, temp -> List,
step -> Int
   init
      result <- cons(ele,nil)
      step <- 0
      temp <- nil   s <- l
   endinit
   if (step=0) then
      if ( s =~ cons(&hd,&tl) ) then
       temp := cons( &hd, temp)
       s := &tl
       if (&tl=~nil) then
         step := 1
       endif
      endif
   elseif (step =1) then
      if (temp =~ cons(&hd,&tl)) then
       result := cons( &hd, temp)
       temp := &tl
       if (&tl=~nil) then
         step := 2
       endif
      endif
   elseif (step = 2) then
      return result
   endif
endasm
```

Figure 5. ASM specification for `append` operation in `sequence`.

All ASM specifications for OCL are given in a library file. These ASM specifications may be called when the ASM virtual machine executes a constraint.

4.3 ASM Virtual Machine Execution

After the ASM virtual machine accepts an Mi-level model (i=1,2) with some constraints, it can be running if the next lower model is provided. For example, after accepting a model (M1-level) given in Figure 4, the ASM can implement a rapid model check if an instance of that model is provided. Usually a software developer can use an object diagram to represent this instance.

The ASM virtual machine first declares a universe which includes all the objects derived from the same class name. Besides the declaration of some universes, we also need to declare a function to link all elements from the same class. This lets us use some operations such as `cons` provided in XASM conveniently. For example, for class *Person* in Figure 4, we have a function definition as follows:

function Person_list -> List

Assuming there exist three elements frank, john and smith in an stance of the model shown in Figure 4 then we can write the following ASM specification:

Person_list:=cons(frank,cons(john,cons(smith,nil)));

Furthermore, all functions which are defined in the M1-level model and related to the objects in its next lower level model will be assigned according to its instance model. For example, the associate link functions defined for objects are assigned according to the link in the model instance. At the same time the function counting the multiplicity for the association link is updated.

After declaring and assigning all the functions according to the M1-level model and its instance model, the ASM virtual machine starts to check all constraints in the M1-level model, including graphical constraints such as the multiplicity for an association, and the other OCL constraints given in the M1-level model.

When the ASM virtual machine checks the syntax of a UML model (the M1-level model), it compares the UML

[1] Interested readers are referred to [?] for more details.

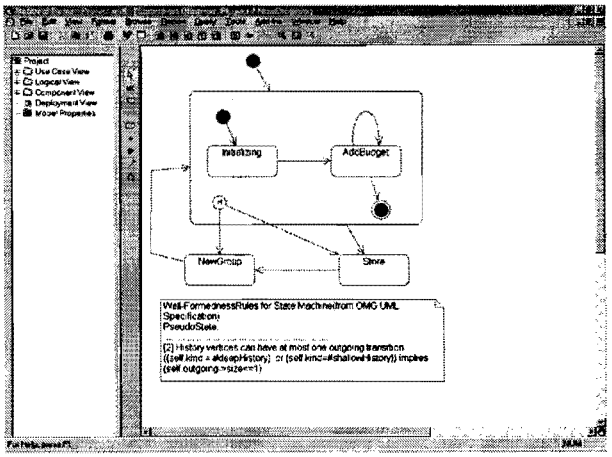

Figure 6. A model used to compare with the UML meta-model.

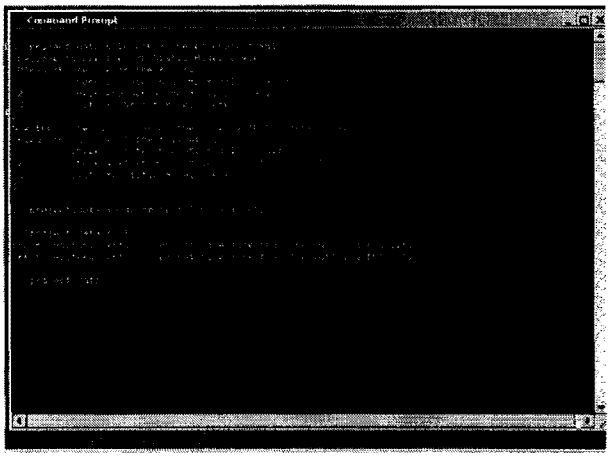

Figure 7. A result for checking the model shown in Figure 6.

model with the UML meta-model (the M2-level). First the ASM virtual machine accepts the UML meta-model and generates the ASM specification according to the above schema. After then, the ASM virtual machine can receive a UML model, which is an instance of the M2-level model. Because any object in M1-level should be an instance of an object in the M2-level, the ASM virtual machine assigns some values to the related functions defined by the UML meta-model. Last, the ASM virtual machine can be running to check all the constraints given in the UML meta-model and find errors in the UML model.

5 Examples

When software developers develop a software system, they first design a model for the problem by employing UML diagrams. In order to make sure that a model is a syntactically correct model, software developers can check whether the model satisfies all constraints given in the UML meta-model by running the ASM virtual machine which compares the model software developers have designed with class diagrams in the meta-model for the UML. If errors are found during the checking, they will be returned to the developers.

Now let us take a look at the following example in Figure 6 by using Rational Rose. In order to give a flavor of how the toolset works, we give a contrived example represented by the state chart diagram.

From the example, there are two outgoing transitions from a history state. But according to the well-formed rules given for the state machine part in [12], you can find the rules for history vertices as follows:

. . . , . . . , . . .

[2] History vertices can have at most one outgoing transition

$(self.kind = \#deepHistory or self.kind = \#shallow-History)$ **implies** (self.outgoing−>size <= 1)

After a developer sets up this model, he can use the toolset to validate the static aspect of the model. The tool first compares the state machine diagram with the class diagram given for the state machine meta-model. After then it checks all well-formedness rules associated with the state machine part. A violation of the well-formedness rule about the history vertices can be found, shown in Figure 7.

Figure 8 shows another UML example. Associated with the class diagram in Figure 8 we add some constraints which are written in English, followed by its OCL expressions. Since the Rose does not support OCL, we use a comment box where we write constraints written in English and OCL. One of the purposes of this example is to check whether some states are included in the model. The state is represented by an object diagram, shown in Figure 9. Having a static model given by a class diagram and its instantiation given by an object diagram, we can check whether the object diagram is a correct state for a given model.

Because there is no link between the object *jay* and *ComputerDealer* in Figure 9, Figure 10 shows an error message about the violation against the constraint i5. According to this error information, a software designer can reevaluate

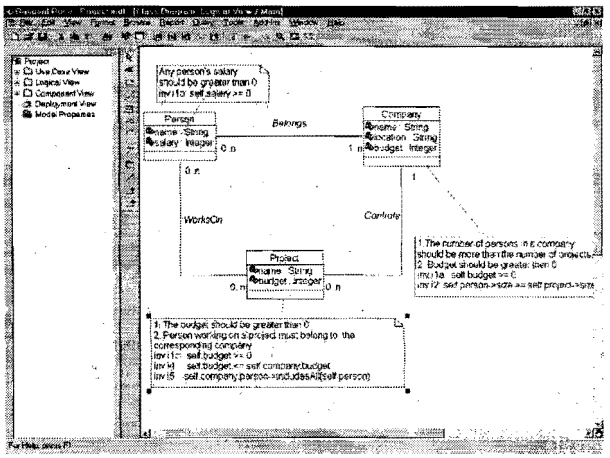

Figure 8. A class diagram for a model.

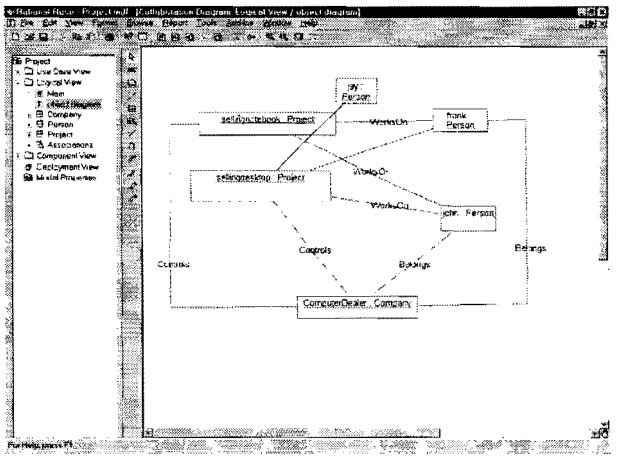

Figure 9. An instance diagram for the model shown in 8.

the model.

6. Comparison and Conclusion

We have built a prototype system for the ASM virtual machine. The results we have got are very amazing. All syntax errors, for example the number of the outgoing transition for a history vertex should not exceed one, have been detected in a UML model even though some UML CASE tools such as Rational Rose cannot find them. With software systems becoming more complicated, the ASM virtual machine can play an important role in checking the validation of an instance of an M1-level model.

In short, the ASM virtual machine provides a feature which many programming language compilers do; our ASM virtual machine supports the syntax check for UML diagrams based on the four-level architecture. Similarly, a software developer can compare his/her M0-level model with the M1-level model and therefore it is possible to make rapid model checking. Additionally, our method combines the semantic model, design model and runtime model into one environment, i.e. the ASM virtual machine for UML. This makes rapid UML model checking efficient. A software developer can receive some feedback immediately without any code-generation provided by most UML CASE tools.

This work was first motivated by the work done in [14]. The USE system provides a way to support UML and OCL based on Java. It maps the objects in an M1-level model into the corresponding Java classes and then runs OCL expressions to find some violations in the M1-level model. An interesting application is that it can validate the UML meta-model. However, this work separates the design environment from its runtime environment (Java runtime environment). Furthermore because the runtime environment is written in Java, which cannot be used as a semantic model, the authors in paper [9] use set theory to give a semantic model for OCL. Additionally, the USE system has its own internal representation for a UML model, which does not support XMI. Therefore its application is greatly reduced.

[15] presents an idea which is similar to ours. The authors see the problems in the code-generation approach adopted by most UML CASE tools. They map an M1-level model into a diagram which consists of "Logical Object and Classes" and "Physical Classes". The part " Logical Object and Classes" is used to represent the logical structure of the objects, and the part "Physical Classes" realizes the logic architecture by using Java. The most important difference between these two virtual machines is that our virtual machine is based on a formal method, i.e. ASMs; but theirs is not. Furthermore, our virtual machine supports OCL and therefore any constraints associated with an Mi-level (i=0,1,2,3) model can be supported in our virtual machine. [15] briefly mentions the support of OCL in their virtual machine, but it is unclear to us whether they have the same features as we do.

UML has been presented to outline a software system during the early phase in the software development. Therefore UML provides some diagrams to specify the behavior of classes. We are working on adding more diagrams in the ASM virtual machine. These diagrams include sequence diagrams and statechart diagrams. We will work on the event in these diagrams and check whether these events are sat-

Figure 10. A result for checking an inconsistency between Figure 8 and Figure 9.

isfied with some constraints given in the other diagrams in UML.

Furthermore, we plan to add statechart diagrams into the ASM virtual machine to describe the behavior of a class based on our previous work [8]. Although statechart diagrams are presented in UML, their role in the methods or operations of a class is not clear. We are seeking the relation between the implementation of the methods or operations and the state chart diagrams.

On the other hand, we consider supporting Action Language which can be used to give more details about the behavior of classes. Considering the pre- or post- conditions which can be defined in the methods or operations, our ASM virtual machine will support these features in order to make rapid model prototype more sufficient and efficient.

References

[1] M. Anlauff. XASM- an extensible, component-based abstract state machines langu age. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines - Theory and Applications*, volume 1912 of *LNCS*, pages 69–90. Springer.

[2] E. Börger, A. Cavarra, and E. Riccobene. An ASM semantics for UML activity diagrams. In *Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000*, volume 1816 of *LNCS*. Springer.

[3] E. Börger, A. Cavarra, and E. Riccobene. Modeling the dynamic of UML state machines. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines - Theory and Applications*, volume 1912 of *LNCS*, pages 223–241. Springer.

[4] E. Börger, I. Durdanovic, and D.Rosenzweig. Occam: Specification and compiler correctness. part i: Simple mathematical interpreters. In E. O. U. Montanari, editor, *Proceedings of PROCOMET'94 (IFIP Working Conference on Programming Concepts, Methods and Calculi)*, pages 489–508, North-Holland, 1994.

[5] E. Börger, U. Glässer, and W. Muller. The semantics of behavioral vhdl'93 descriptions. In *Proceedings of EURO-DAC'94/EURO-VHDL'94*. IEEE Press, 1994.

[6] E. Börger and W. Schulte. Programmer friendly modular definition of the semantics of java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in LNCS, 1998.

[7] A. Cavarra and E. Riccobene. Simulating uml statecharts. In *Proceeding of EUROCAST 2001, pp 224-227, Gran Canaria, Spain Feb. 2001*.

[8] K. Compton, Y. Gurevich, J. Huggins, and W. Shen. An automatic verification tool for UML. Technical Report Technical report, CSE-TR-423-00, Dept. of EECS, University of Michigan, May, 2000.

[9] M. Gogolla and M. Richters. On constraints and queries in UML. In M. Schader and A. Korthaus, editors, *The Unified Modeling Language – Technical Aspects and Applications*, pages 109–121. Physica-Verlag, Heidelberg, 1998.

[10] M. F. Group. Introducing asml: A tutorial for the abstract state machine language. Technical report, Microsoft FSE Group, Dec, 2001.

[11] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[12] OMG Unified Modeling Language Specification, version 1.3, June 1999.

[13] OMG XML Metadata Interchange (XMI) Specification. OMG, 2000.

[14] M. Richters and M. Gogolla. Validating UML models and OCL constraints. In *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000*, volume 1939 of *LNCS*, pages 265–277. Springer, 2000.

[15] D. Riehle, S. Fraleigh, D. Bucka-Lassen, and N. Omorogbe. The architecture of a UML virtual machine. In *Proceedings of the OOPSLA '01 conference on Object Oriented Programming Systems Languages and Applications*, pages 327–341, 2001.