A Toolset for Supporting UML Static and Dynamic Model Checking

Wuwei Shen*
Dept of Computer Science, Western Michigan University
wwshen@cs.wmich.edu

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

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

Abstract

The Unified Modeling Language has become widely accepted as a standard in software development. Several tools have been produced to support UML model validation. However, most of them support either static or dynamic model checking; and no tools support to check both static and dynamic aspects of a UML model . But a UML model should include the static and dynamic aspects of a software system. Furthermore, these UML tools translate a UML model into a validation language such as PROMELA. But they have some shortcomings: there is no proof of correctness (with respect to the UML semantics) for these tools. In order to overcome these shortcomings, we present a toolset which can validate both static and dynamic aspects of a model; and this toolset is based on the semantic model using Abstract State Machines. Since the toolset is derived from the semantic model, the toolset is correct with respect to the semantic model.

1 Introduction

The Unified Modeling Language (UML) is becoming a standardized modeling notation for expressing object-oriented models and designs. More and more, software developers are using UML to model their software in the early stages of software development. But helping developers design a correct software system is still a challenging problem. Investigations have shown that errors are introduced more often during the initial software development stages such as requirement and design stage [6]. These errors will cost much more money to fix during the late software development stages than during the stage when they are introduced. With the advent of UML, it is possible to find a methodology to detect errors when they first appear during software development.

The semantics of UML is described in a meta-model, which consists of three views: Abstract Syntax, Well-formedness rules and Semantics. The Abstract Syntax is provided as a model described in a subset of UML, i.e. class diagrams together with a supporting natural language description. The Well-formedness rules are provided using the Object Constraint Language. Finally the Semantics are described primarily in natural language.

Based on the metamodel of UML, we apply Abstract State Machines in giving the semantics for the above three views in this project. We give the ASM semantics for class diagrams, Object Constraint Language and the semantics parts for UML in our tools. Therefore, a user can have syntax checking for a UML model by comparing it with the UML metamodel.

The architecture of the UML is based on the four-layer metamodel structure, which consists of the following layers: user objects, model, meta-model and meta-metamodel. According to the UML document [16], a UML model defines a language to describe an information domain; however, user objects are an instance of a model, which defines a specific information domain. Because the instance of the model is usually represented by an object diagram, we give the ASM specification for object diagrams in UML in our tool. Based on the ASM semantics for UML class diagrams and object diagrams, the tool can help a user check whether a specific information domain is valid according to the UML model the user gives in the class diagram.

A UML model usually includes both static and dynamic aspects so as to completely model a real application. In general, the static aspect of a model can be represented by the static diagrams in UML, such as class diagrams, together with some constraints written in the Object Constraint Language (OCL); the dynamic aspect of a model can be given by the UML dynamic diagrams such as state machine diagrams[1] or activity diagrams.

We think that any tool supporting the validation of a UML model should include static and dynamic validation. First, static validation can be used to check whether a model is syntactically valid, i.e., whether the model satisfies the UML meta-model including the well-formedness rules given by OCL. On the other hand, as the application becomes more complicated, it is harder for a developer to find whether some state (specific information domain) is valid according to the model which (s)he is developing. The second function for the static validation is that it can help a developer check whether his/her UML model includes some specific information domain.

*Partially supported by NSF grant CCR 95-04375.

[1] We use the term "state chart diagrams" and "state machine diagrams" interchangeably.

After designing a static structure of a model, a developer can specify dynamic behavior for a class and this kind of behavior can be represented by UML dynamic diagrams such as state chart diagrams. Dynamic validation is used to check whether the dynamic aspect of a model satisfies some important properties such as safety or liveness.

There are few research tools [10] and [13] available to support either static or dynamic validation. One of the reasons most tools do not support model validation is the lack of the formal semantics of UML and OCL. Generally, research work to support UML model validation usually includes the following two steps. First, researchers present a formal semantics for a diagram or language in which they will work; and then, according to the formal semantics, they either translate the diagram or language into some language supporting the validation or use some programming language to execute the diagram or language. One of the problems in the above tools is that the researchers have not given a proof of correctness (with respect to the UML semantics) for these tools, although the validation model they assume is the same as the semantic model.

After we investigated existing validation tools, we found that another common weakness is that most of them have their own unique representations for a model. They do not support XMI which is an XML Metadata Interchange Format and therefore these tools can not be used with many UML commercial tools such as Rational Rose. For this reason, these tools' applications are greatly reduced.

In this paper we will introduce a new toolset which tries to overcome the weaknesses in previous validation tools. First this toolset supports validation of both the static and dynamic aspects of a model. This is the first time (at least to our knowledge) that a UML toolset supports both aspects of a model. Secondly, the validation of a model is totally based on the semantic model given by Abstract State Machines which is also a validation model. Because we use the same model, all validations are completely consistent with our semantic model for UML diagrams. Last, XMI is used to represent a model in this toolset so that no matter what kind of UML CASE tools a software developer uses, (s)he can validate a model if the model can be translated into XMI format.

In the following sections, we will first introduce how to generate ASM specifications for a UML model. We will give an overview about the toolset and its functions in section 3. In section 4 we will draw some conclusions.

2 Abstract State Machines and ASM Specifications for a UML Model

Abstract State Machines (ASMs) [7] were first presented by Gurevich ten years ago. Since then, they have been successfully used in specifying and verifying many software systems. For example, they have been used to give a semantic model for several programming languages such as C. Furthermore ASMs have been applied to UML in a variety of ways. Börger et al have applied ASMs to provide semantics for UML activity diagrams and state machines ([3] and [4]), and are currently working on simulation tools for UML statecharts [5]. The dynamic validation part in this toolset is based on results previously announced in [15].

An ASM program consists of some rules which can be found in [7]. There are several tools available to support ASMs. In this project, we use XASM [2] to execute ASM specifications. XASM, an extensible ASM, realizes a component-based modularization concept based on the notion of external function as defined in ASMs. Compared with other ASM tools, we think XASM may be suited for high-level validation of UML models.

Another important tool supporting ASM is a model checker. This ASM model checker is based on the SMV model checker. Although our ASM model checker is built on the SMV model checker, it is quite different from other model checker because it can be used to validate some properties totally based on the semantic model given by ASMs. Therefore our validation model is the same as the semantic model. But most other model checkers do not have this advantage. More detailed introduction to ASMs and their tools can be found in [7] and [1].

Because our ASM specification for a UML model consists of the static and dynamic parts, we will outline the two parts of ASM specification in the following two subsections.

2.1 ASM Specifications for a static part of a UML Model

Our ASM specification for the UML meta-model consists of three parts: class diagram, OCL and the semantics. We have a set of rules which maps the structures in class diagrams and OCL into a set of ASM specifications. Because the semantics part of UML is written in English, we translate it into the corresponding ASM specifications. Now the tool is supporting the core package and state machine part in the UML document [16].

As an example, we illustrate how to check whether a state (a specific domain) is a valid instance of a UML model (a language describing a domain.). Similarly, a syntactically valid UML model can be checked in the same way in our tool because a UML model is an instance of the UML meta-model. Because object diagrams are an important way to represent user objects, we give our ASM specification for object diagrams besides class diagrams and OCL in the tool. Due to space restriction, readers are referred to [2] for more details about the XASM specifications. In general we define some functions in our ASM specification for class diagrams and set values for these functions when ASM specifications for an object diagram are given. We will illustrate how to give ASM specifications for a static aspect of a UML model by using the diagrams shown in Figure 1 so as to give readers a flavor about how the tool works. In Figure 1(a) there are two classes and we define a universe named *Class_Name* which contains these two elements. See Figure 2.

For any class, there are some attribute and method definitions. For example in class $Control$ there are attribute definitions such as $Name : String$. In the ASM specification, we give a function definition for each attribute definition. We omit the type definitions for attributes because types can be decided at running time in the current XASM compiler version. A parameter in the attribute function definition denotes an object possibly derived from the associated class. Thus the ASM specification for the attribute $Name$ can be found in Figure 2.

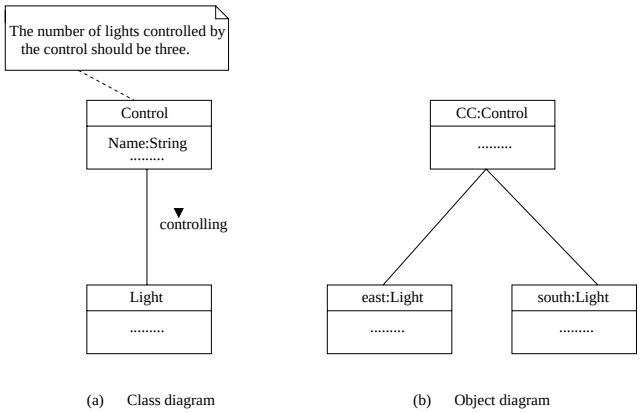

Figure 1. A class diagram and object diagram for a model.

The other important relationship in a class diagram is an association which provides a way for object navigations. There are several ways to navigate from a specific object to the other objects through an association in a class diagram in UML. We use relations (boolean functions) in XASM to represent all kinds of the navigations. The parameters in the relation represent the objects derived from the associated classes. In Figure 2 we define relations for all possibilities for navigations appearing in a class diagram. Besides the association, we can define ASM function to represent the generalization and dependency relationship in a class diagram.

After giving a static aspect of a model, software developers can give an object diagram as state (an instance) of the model. In order to set up the relation between a class and its instantiated objects in the state, we use a list structure in XASM. In Figure 2, we give the XASM specifications for the object diagram in Figure 1 (b). In this part, we first give a universe definition for all objects and define functions setting up the relation between a class and its corresponding objects. Then we set the relation for those associations defined in the class diagram.

Another important part of static validation is the Object Constraint Language. We define ASM specifications in a library for all the operations in OCL. This library is very important and it is used not only when an OCL expression is called but also when some structures in a class diagram are met. For example, we can call the library when we check the multiplicity for an association. However due to space restriction, we skip the details, which can be found in [15].

2.2 ASM Specifications for a dynamic part of a UML Model

A dynamic part of a UML model is usually represented by dynamic diagrams in UML such as state chart diagrams and activity diagrams. In the current version of the toolset, we consider state chart diagrams as a dynamic aspect of a UML model. The ASM specification for a state chart diagram consists of two different parts. One is used to represent a state and the other to represent

$universe\ Class\_Name = \{Control, Light\}$
$function\ Name(\_);$
$relations\ control\_light(\_,\_), control(\_,\_), light(\_,\_)$
$\quad \ldots, \ldots, \ldots$
$universe\ Object\_Name = \{CC, east, south\}$
$function\ control\_obj\_list -> List$
$function\ light\_obj\_list -> List$
$\quad \ldots, \ldots, \ldots$
$\quad control\_obj\_list := cons(CC,nil)$
$\quad light\_obj\_list := cons(east,cons(south,nil))$
$\quad \ldots, \ldots, \ldots$
$\quad controlling(CC, east) := true$
$\quad controlling(CC, south) := true$
$\quad light(CC, east) := true$
$\quad light(CC, south) := true$

Figure 2. The part of the XASM specification for an object diagram in Figure 1(b).

a transition. Due to space restriction, we skip the details which can be found in [15].

3 Validation of a UML Model

Based on the schema for generating ASM specifications for a UML model, we build a toolset helping software developers find errors during their early stages of software development. We will give a structure of the toolset and then introduce the functions for this toolset in the following.

3.1 An Architecture of the Toolset

To help software developers find problems early, our toolset provides the following features: syntax check, state check and dynamic behavior validation. Figure 3 gives an architecture for the toolset. For static validation, the modules `syntax check` and `state check` are used and they call the module `ASM spec for static diagrams` and `ASM spec for constraints` which are based on the executable ASM compiler XASM. The module `ASM Spec for dynamic diagrams` is used to check a dynamic property of a model by calling the ASM model checker.

The module `ASM spec for OCL operations` is used to provide ASM specifications for all OCL operations. Because either the well-formedness rules or the constraints in a model are written in OCL, this module is used as a standard XASM specification library for the OCL operations. This module is called when the module `syntax check` or `state check` is used in the toolset.

The module `ASM spec for static diagrams` provides the ASM specification for the static diagrams for UML. In the current version of the toolset, only ASM specifications for class diagrams and object diagrams are provided. When either

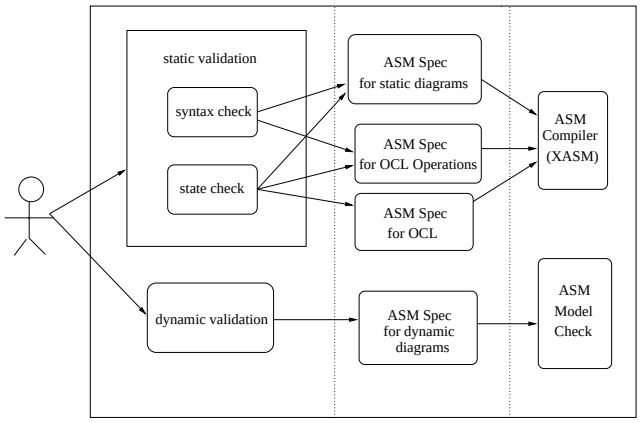

Figure 3. The Architecture of the toolset.

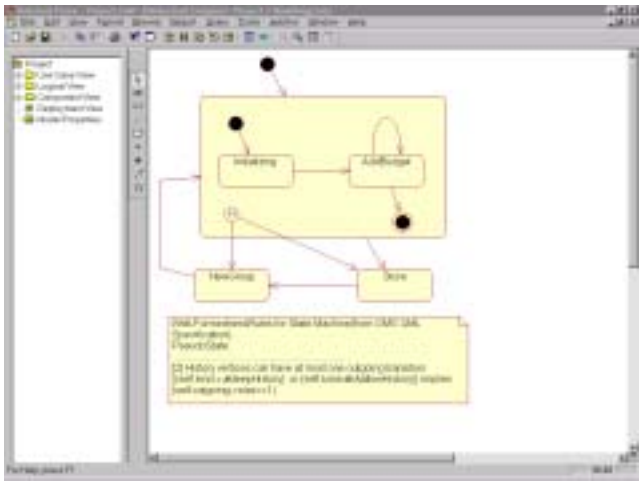

Figure 4. An example for static checking for a model.

syntax check or state check is used, the toolset calls the ASM compiler XASM to execute the ASM specifications generated by this module and the module `ASM spec for OCL operations`.

The module `ASM Spec for constraints` is used to provide the ASM specification for constraints in a model. Because the constraints written in OCL are included in the static diagram of a model, the module `ASM spec for static diagram` takes all constraints from the model and stores them into a data structure where the module `ASM spec for OCL` can retrieve these constraints and check their syntax. If there is no syntax error, the module translates the constraints into the ASM specification. When doing a state check, the toolset calls the ASM compiler XASM to run the ASM specifications generated by the above three modules.

The module `ASM for dynamic diagrams` concentrates on the state chart diagram in UML. This module gives the ASM specification for the state chart diagram and then calls the module `ASM model checker` to verify some properties such as liveness and safety. The module returns some information about the result especially when some errors are found. Software developers can redesign their model according to the result.

3.2 Validation of a static part of a model

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 the model is a syntactically correct model, OMG uses class diagrams combined with well-formedness rules written in OCL to give the meta-model for the UML. Therefore the static validation includes two steps. The first step is a syntax check, i.e., comparison of the model a software developer has designed with class diagrams in the meta-model for UML. Then the toolset will determine whether the model satisfies all the relevant well-formedness rules. If some errors are found during the above two steps, they will be returned to the developer.

Figure 4 is a contrived example represented by a state chart diagram and this diagram is edited by Rational Rose. In this diagram there are two outgoing transitions from a history state. But according to the well-formed rules given for the state machine part in [16], History vertices can have at most one outgoing transition and the diagram in Figure 4 violates this rule.

Another important aspect for a UML static validation is to check whether some state, represented by an object diagram, is included in a UML model or not. The idea behind this kind of validation is that after a developer designs a static model for a software system, (s)he has some states in his(her) mind related to this model. Sometimes these states must be included in the model, this toolset can be used to find this kind of inclusion. If these states are excluded, the developer must redesign the model to contain these important states. Similarly this toolset can be used to check the exclusion of some erroneous states in a UML model.

3.3 Validation of the dynamic part of a model

When a developer develops a software system, (s)he usually considers not only the static part of a model but the dynamic part as well. State chart diagrams in UML are used to represent the dynamic behavior of a class. In order to support to find errors in the dynamic aspect of a UML model, this toolset can be used to validate some properties based on the ASM model checker.

According to the above schema, this toolset can accept a state chart diagram and then automatically generates ASM specifications. These ASM specifications can be sent to the ASM model checker, which is based on SMV model checker. Therefore some properties such as safety property can be directly validated.

Here we present a traffic control problem [12] as an example to show how the toolset works for the dynamic validation for a model. Before diving into the example, let us briefly take a look at the problem description.

There is a controller that operates the traffic lights at an intersection where a two-way street running north and south intersects a one-way street running east. The control includes three monitors, sensors and sets of the traffic lights (red and green). Each monitor,

sensor and set of traffic lights take responsibility for one direction. When a monitor detects incoming traffic, it sends a message to the sensor, which sends a request to the controller. According to the requests from different directions, the controller will send a signal to the corresponding traffic light (either red or green) so as to avoid collisions and make sure no traffic waits at a red light forever.

The controller has three traffic sensor inputs: N_Sense, S_Sense and E_Sense, indicating incoming traffic in the north, south and east direction respectively. These sensor inputs are triggered by the hardware, monitoring the three different directions. These east, north and south direction's monitors are called E_Monitor, N_Monitor and S_Monitor respectively. We treat these monitors as events in the UML state chart diagram. The three internal registers N_Sense, S_Sense and E_Sense are set when the corresponding monitor detects incoming traffic. The outputs N_Go, S_Go and E_Go are used to indicate that a green light should be given to traffic in each of the three directions. In addition, the register *NS_Lock* is set when traffic is enabled in the north or south directions, and prevents east-going traffic from being enabled. The three bits *N_Req*, *S_Req*, *E_Req* are used to latch the traffic sensor inputs. All these inputs, output and internal registers are treated as Boolean variables in the UML model.

In Figure 5, we give a state chart for the controller. There are five states in this model. They represent no-traffic, south-bound traffic, north-bound traffic and east-bound traffic. In order to make the diagram clear, we use C1, . . ., C13 to denote all the transitions in Figure 5. Due to space, we just explain transitions for the state associated with the no-traffic. In the current version of the toolset, we borrow the notation from SMV to represent the transitions and properties. The transition C3 is used to show the transition from No-traffic to North-bound. The idea is that if there is a north-bound request but no south-bound and east-bound traffic request and no traffic for north-bound, then we set the variable for the north-bound traffic and the register for the north-south bound traffic. This can be defined as follows: [N_Req & ˜ N_Go & ˜ E_Req & ˜ S_Req]/N_Go:=1;NS_Lock:=1. Similarly we can give the transition C5 provided that north-bound and south-bound variable are switched according to the transition C3.

The "liveness" property for this example, which says that if the traffic sensor is on for a given direction, then the corresponding light is eventually on; thus, no traffic waits forever at a red light. Here we are interested in north-bound liveness, shown as follows:

```
N_Live:  assert G (N_Moniter —> F N_Go)
```

A careful reader can find the north-bound liveness is violated according to the transition C3 and C5 because that model does not consider the case when the south and north-bound traffic requests are set to true at the same time. When we use the ASM model checker to verify the above safety property, we have a counterexample shown in Figure 6. Because the ASM model checker is built on the SMV model checker, the result returned to a developer is the same as the one given by the SMV model checker. In the bottom of Figure 6, the left column represents the variables defined in our UML model together with state. The first line gives a possible sequence number. A value for a variable at a given number in a sequence is given in the box decided by the corresponding line and column. In this example |:2:| represents the sequence number 2

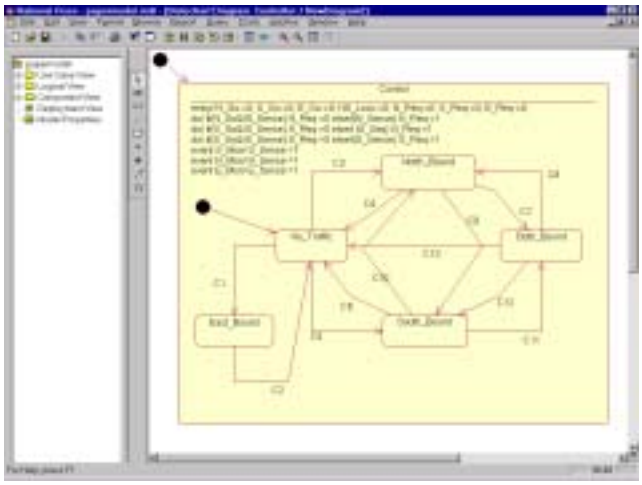

Figure 5. A state chart for the traffic light problem.

will repeat forever, i.e., all variables' values will not change when both south-bound and north-bound requests are set to true.

In order to correct this problem, we can add a new variable `flag` to deal with the both south and north-bound requests. The idea is that we give a priority for one direction request by checking the value of `flag`. When both south- and north- bound requests occur, one direction traffic can go through and we flip the value of `flag` so that the opposite direction traffic can pass when both north and south-bound requests will be set in the next time. Similarly we redefine the transition C5. Then we can verify the liveness property without any error.

4 Conclusion

In this paper, we presented a toolset based on Abstract State Machines. The ideas behind this toolset are different than those behind other research validation tools. One difference is that this toolset uses the semantic model to do the validations. Thus any error found in the validation model is also an error according to the semantic model. On the other hand, with UML becoming more dominant in software development, designing just a static or dynamic aspect of a model is becoming obsolete. Therefore, we have built a toolset which can validate both aspects of a model. This helps software developers find errors in their model design as soon as possible. Last, almost all of the UML CASE tools have unique internal representations. The OMG presented XMI, trying to specify an open information interchange model that is intended to give developers working with different UML CASE tools a means to exchange models in a standardized way. Based on this consideration, we adopt the XMI format in this toolset so that this toolset can be used with any other UML commercial CASE tools. The ideas presented in this paper have been implemented in the toolset.

When we were writing this paper, we were told of another project, called Alcoa [8], which shares many similarities with ours. Alcoa uses a new specification language Alloy [9], based on Z, and

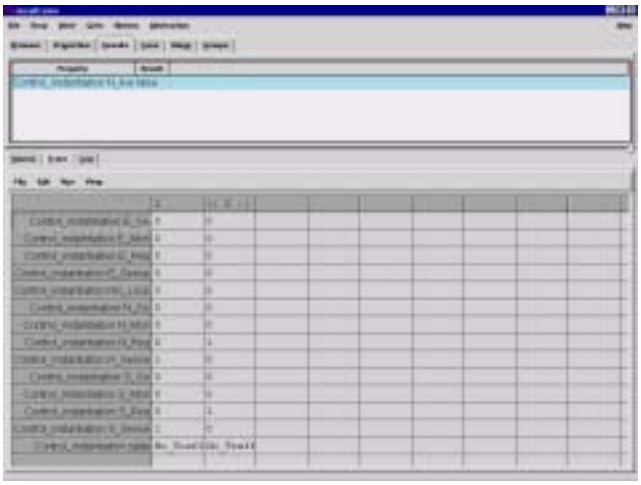

Figure 6. A counter example returned by the toolset for the traffic light problem.

it, like UML, can be used to give a model for early software development. Receiving a model given by Alloy, Alcoa can provide two kinds of analysis by translating the model into a boolean formula which is handed to a SAT solver. After running the SAT solver, Alcoa can check whether constraints associated with the Alloy model is either too weak or too strong. These kinds of checking are similar to our static validation in the toolset. We can find these kinds of errors by running OCL constraints based on ASM specification. But our validation tool can verify some dynamic properties associated with a class in addition to static validation.

In fact, we think ASMs can become a good candidate for a specification language for software development because they have many features Alloy has. However, UML has become so widely accepted and it is unlikely that any new notation will replace it soon. Accordingly we are building a validation tool for UML based on ASMs.

Much future work is anticipated for this toolset. First this toolset is far from maturity. In this paper, we just show that Abstract State Machines can be used in building such a toolset. We need more real applications to test the toolset. As for the toolset itself, we still have a lot of work to do. To support the validation of user objects level, we give our ASM specification for object diagrams in the current version of the tool. But we will consider the ASM specification for other diagrams in UML related to objects such as the interaction diagrams in our tool. By checking the inconsistency in these diagrams, we hope that the tool can help a developer find more errors during the design phase. In addition, we will build a common interface for the static and dynamic validation tools.

Acknowledgement The first author appreciates the conversation and emails with Mark Richters about his USE tool and the meta-model for UML.

References

[1] Abstract State Machine Homepage edited by J. Huggins: *http://www.eecs.umich.edu/gasm.*

[2] Matthias Anlauff, XASM- An Extensible, Component-Based Abstract State Machines Language, Proceeding of Abstract State Machine Workshop, 2000.

[3] E. Börger, A. Cavarra, and E. Riccobene. "An ASM Semantics for UML Activity Diagrams", 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000, Proceedings, Springer LNCS 1816, 2000, 293–308.

[4] Egon Börger, A. Cavarra, and E. Riccobene. Modeling the Dynamic of UML State Machines, Proceeding of International Workshop, ASM 2000, Monte Verita, Switzerland, volume 1912 of LNCS, pp. 232-241, Springer Verlad, Mar. 2000.

[5] Alessandra Cavarra, E. Riccobene. Simulating UML Statecharts, Formal Methods and Tools for Computer Science, Proceeding of EUROCAST 2001, pp 224-227, Gran Canaria, Spain Feb. 2001.

[6] R. Bourdeau and B. Cheng, A Formal Semantics for Object Model Diagrams, IEEE Transactions on Software Engineering, Vol. 21, No. 10, pp 799-821, Oct., 1995

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

[8] D. Jackson, I. Schechter and I. Shlyakhter, Aloca: The Alloy Constraint Analyzer, Proc. International Conference on Software Engineering, Limerick, Ireland, June 2000.

[9] D. Jackson, Alloy: A lightweight Object Modeling Notation, *http://sdg.lcs.mit.edu/d̃nj/publications.html*, July 2000.

[10] Johan Lilius and Ivan Porres Paltor, vUML: a Tool for Verifying UML Models , TUCS Technical Report No. 272, May 1999.

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

[12] Ken L. McMillan. Getting Started with SMV. http://www-cad.eecs.berkeley.edu/ kenmcmil/smv/.

[13] Mark Richters and Martin Gogolla. Validating UML models and OCL constraints. UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, LNCS Vol. 1939. Springer, 2000.

[14] Wuwei Shen, Kevin Compton, James Huggins. A Validation Method for a UML Model Based on Abstract State Machines. Formal Methods and Tools for Computer Science, Proceeding of EUROCAST 2001, pp 220-223, Gran Canaria, Spain Feb. 2001.

[15] W. Shen. The Application of Abstract State Machines in Software Engineering, PHD thesis, Dept of EECS, The University of Michigan, Sep 2001.

[16] Rational Software Corporation, Unified Modeling Language (UML), version 1.3, *http://www.rational.com*, 1999.