A Toolset for Supporting UML Static and Dynamic Model Checking

Wuwei Shen * and Kevin Compton *
Dept. of EECS, The University of Michigan
{wwshen,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. These tools translate a UML model into a validation language such as PROMELA. However, they have some shortcomings: there is no proof of correctness (with respect to the UML semantics) for these tools; and there is no tool that supports validation for both the static and dynamic aspects of a UML model. In order to overcome these shortcomings, we present a toolset which 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. In addition, this toolset can be used to validate both the static and dynamic aspects of a 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. 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, represented by an object diagram, is included in 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 model includes some related snapshots or not.

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 not many research tools [6, 8] 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. Additionally, most existing tools 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 [5] 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.

ASMs have been applied to UML in a variety of ways, including the semantics for UML activity diagrams [2] and state machines [3]. The dynamic validation part in this toolset is mainly based on results previously announced in [4]. The paper is organized as follows. Section 2 will introduce the architecture of the toolset, including ASM specification for a dynamic part of a UML model. Section 3 will give a case study to show how this toolset works. We will draw some conclusions in section 4.

*Partially supported by NSF grant CCR 97-32735.

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

2 An Architecture of the Toolset

2.1 An Overview

To help software developers find problems early, our toolset provides the following features: syntax check, state check and dynamic behavior validation. Figure 1 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 [1]. The module `ASM Spec for dynamic diagrams` is used to check a dynamic property of a model by calling the ASM model checker.

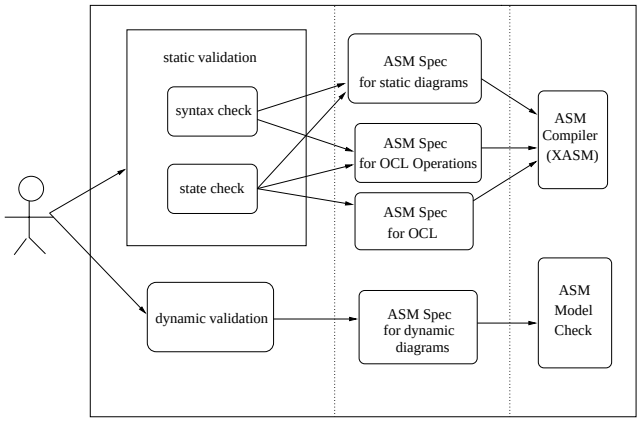

Figure 1. The Architecture of the toolset.

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 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. Due to space restriction, we illustrate dynamic diagrams to show how ASMs can be used to specify a UML model.

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. We will outline the ASM specification for a state chart diagram in the following and more details about it can be found in [4].

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 a transition. Figure 2 denotes the simplest state chart diagram and we just consider the ASM specification for this simple case and the ASM specifications for other composite states can be found in [4].

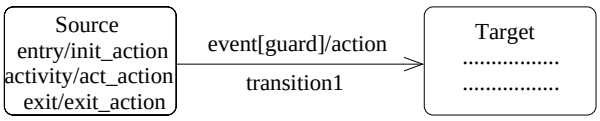

Figure 2. A simple state chart diagram.

Before giving the ASM specification, we introduce some functions used in our ASM specifications. The function $CurControl$ is used to denote the name of a current node or transition being executed. When a node is executed, the function $Phase$ is used to indicate different phases of a node execution. These phases are denoted *init*, *internal_exe*, and *exit*. Because UML does not provide a syntax for actions and activities, we adopt the standard programming language (such as C) structure to represent them; and this is not an important part in UML diagrams so we omit our ASM specification for these actions and activities in Figure 3.

For the simple state $Source$ shown in Figure 2, we execute the initial actions during the *init* phase. During the *internal_exe* phase, we execute the activities and at the same time we check the guard and event related to an outgoing transition. If the guard is true and event occurs[2], we enter the phase *exit_exe*. The last phase executes the exit actions and passes control to the corresponding outgoing transition by setting the function $CurControl$.

When control reaches a transition, we execute the actions associated with this transition because we have checked the guard and event associated with the outgoing transition. And at the same time we set the the control to the node named `Target` so that the actions and activities associated with node `Target` can be exe-

[2]We treat an event as a boolean variable and the *event=true* denotes that *event* occurs.

```
if (CurControl = Source and
        Phase(CurControl) = init) then
    execute the initial action
    Phase(CurControl) := internal_exe
elseif (CurControl = Source and
        Phase(CurControl) = internal_exe) then
    execute the activity
    if ( guard = true and event = true) then
        Phase(CurControl) := exit_exe
    endif
elseif (CurControl = Source and
        Phase(CurControl) = exit_exe) then
    execute the exit action
    CurControl := transition1
endif
            . . . , . . .
if (CurControl = transition1) then
    execute the action associated with this transition
    Phase(Target) := init
    CurControl := Target
endif
```

Figure 3. The XASM specification for a simple state.

cuted in the next step. The second part of Figure 3 outlines the ASM specification for a transition.

2.3 Case Study

Due to space restriction, we only show how the toolset works for the dynamic part of a UML model. According to the above schema, this toolset can accept a state chart diagram and then automatically generate 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 properties can be directly validated.

Here we present a traffic control problem [7] 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

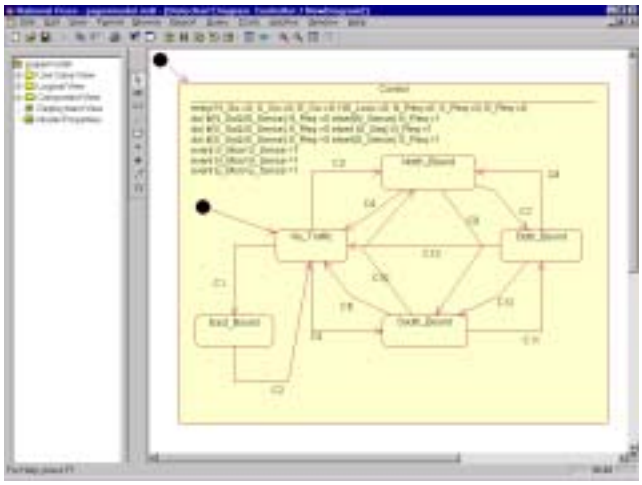

Figure 4. A state chart for the traffic light problem.

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 4, 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 4. 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

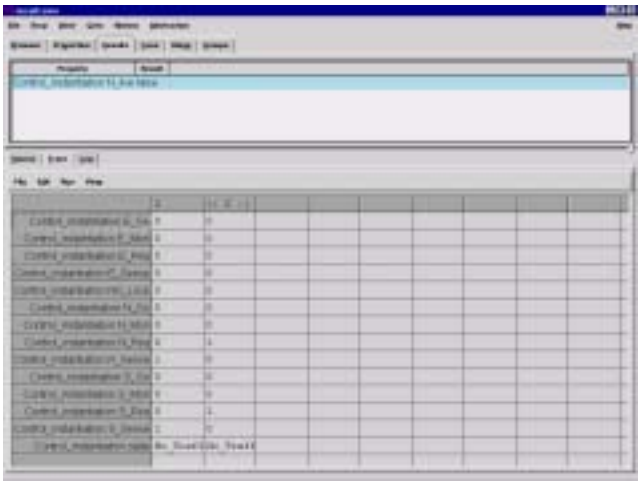

Figure 5. A counter example returned by the toolset for the traffic light problem.

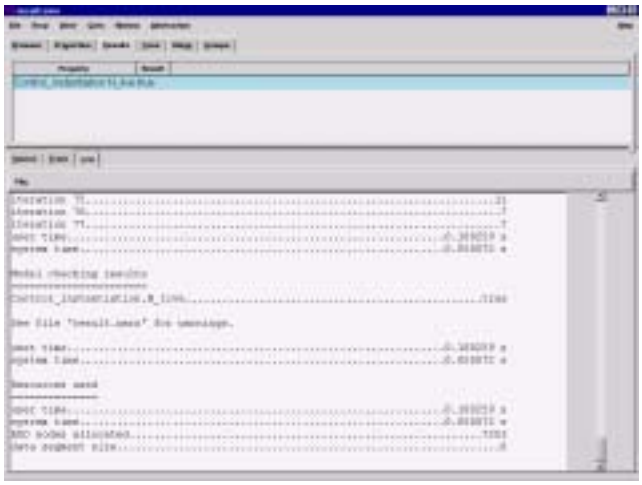

Figure 6. The saftey for the traffic light problem is passed by thetoolset.

checker to verify the above safety property, we have a counterexample shown in Figure 5. 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 5, 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 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 shown in Figure 6.

3 Conclusion

In this paper, we presented a toolset based on Abstract State Machines, which are both semantics and validation model. 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, we adopt the XMI format in this toolset so that this toolset can be used with any other UML commercial CASE tools.

Much future work is anticipated for this toolset. First we are expecting more other diagrams in UML to be included in this toolset. Additionally, We need more real applications to test the toolset. As for the toolset itself, we will build a common interface for the static and dynamic validation tools.

References

[1] Matthias Anlauff, XASM- An Extensible, Component-Based Abstract State Machines Language, in Y. Gurevich etc ed., Abstract State Machines - Theory and Applications, pp. 69-90, Springer LNCS 1912, 2000.

[2] Egon Börger, A. Cavarra, and E. Riccobene. "An ASM Semantics for UML Activity Diagrams", in Teodor Rus, ed., Algebraic Methodology and Software Technology, pp 293-308, Springer LNCS 1816, 200.

[3] Egon Börger, A. Cavarra, and E. Riccobene. Modeling the Dynamic of UML State Machines, in Y. Gurevich etc ed., Abstract State Machines - Theory and Applications, pp. 232-241, Springer LNCS 1912, 2000.

[4] K. Compton, Y. Gurevich, J. Huggins, W. Shen. An Automatic Verification Tool for UML, Technical report, CSE-TR-423-00, University of Michigan, 2000.

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

[6] Johan Lilius and Ivan Porres Paltor, Formalising UML State Machines for Model Checking, in R. France etc. ed. UML'99 - The Unified Modeling Language. Beyond the Standard, pp. 430–445, Springer LNCS 1723, 1999.

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

[8] Mark Richters and Martin Gogolla. Validating UML models and OCL constraints. UML 2000 - The Unified Modeling Language. Advancing the Standard, pp. 265-277, LNCS Vol. 1939. Springer, 2000.