

Dynamic Semantic Checking for UML Models in the IIOSS System

Motoyuki Sano^{*,**)}

*) Open Technologies Corporation
Koishikawa, Bunkyo-ku
Tokyo 112-0002, Japan
sano@opentech.co.jp

Teruo Hikita^{**)}

***) Dept. of Computer Sci., Meiji University
Higashimita, Tama-ku
Kawasaki 214-8571, Japan
hikita@cs.meiji.ac.jp

ABSTRACT

In order to incorporate semantic checking into UML, we have created a mechanism which performs model simulation on UML diagrams, including statechart, activity and sequence diagrams. This helps designers in developing correct models at early stages of software design.

Keywords

Activity diagram, behavioral diagram, debugger, dynamic semantic checking, object-oriented design, sequence diagram, simulation, semantic checking, statechart diagram, UML.

INTRODUCTION

The modeling language UML (Unified Modeling Language) developed by OMG (Object Management Group) is now becoming a standard tool in order to probe and obtain the user's needs and to design object-oriented software systems [3],[13]. Various kinds of diagrams designed in graphical editors help system analysis and software architects to do their work. Most of UML editing tools have a diagram-syntax checker, but fail to examine diagrams' behaviors, which requires careful semantic analysis of diagrams. In order to incorporate semantic checking into UML we have created a mechanism which performs model simulation on a diagram by taking its system specifications into account.

Because models are written in a higher level in abstraction than programs, simulating models only with information described with them is difficult in general. At the same time a modeling tool must have a capability to handle large-scale systems for practical use.

In this paper we exhibit the framework for simulation on UML behavioral diagrams including statechart, sequence and activity diagrams. To be more specific, the mechanism has the following five new features:

- (i) The simulator works not only on statecharts which are mostly used to simulate models, but on all four behavioral diagrams. This helps collaborating work with different kind of diagrams with different views on same systems.
- (ii) A message passing functionality between model objects. With it, our system can simulate large-scale complex models which are hard to describe without resorting to combinatorial uses of more than one diagram.
- (iii) A correspondence functionality between model objects

in different types of diagrams. This helps a user to check consistency between different diagrams.

(iv) A functionality that can pass messages between a model object and a Java program during simulation. This helps to promote software reuse.

(v) An interactive debugger that can provide software designers with a functionality to control the way how the simulation ought to proceed. The debugger in a way carries out simulation onto a model. It solicits an input from a user when it detects an object with multiple outgoing relations.

We introduce these five new features mainly for two reasons. One is that the previous method of using only statechart diagrams for semantic checking or model simulation is inadequate, since in general, statecharts are suited for detailed descriptions of system behavior within a single object, while activity diagrams and sequence diagrams are more appropriate for describing global or initial description of a system (see e.g., [3]).

The second reason is that the system descriptions in the early stages of software development are inherently vague and imperfect, but still model checking or simulating its dynamic behaviors is useful not only for correcting errors but for clarifying difficult and subtle points in designing the models.

The dynamic semantic checking capability described in this paper has been developed on the IIOSS system (Integrated Inter-exchangeable Object-modeling and Simulation System), enhanced as its model debugging facility.

PREVIOUS WORK

In this section we overview previous work on the two topics of model simulation and integrated tools.

Model Simulation

The User Interface Prototyper (UP) was one of the early model simulators [15]. It simulates both statechart diagrams and conceptual process diagrams written in Entity Relationship models, by defining actions as the attributes of objects in diagrams.

There are now many software tools with model simulation capability. Most of them are used when developing realtime or embedded software systems. Because the notions such as timing and realtimeness are important in those systems, the

tools usually employ the following steps: 1) one writes state transition diagrams or tables in detail, 2) compiles them to build an executable program, and 3) simulates and checks the model by invoking the program. These tools mostly use their own diagrams rather than UML diagrams [11],[12], but recently they seem to be moving to conform to UML diagrams.

Recent Research on Simulation and UML

The UME/Ruby is a modeling aid which helps to check internal and external behaviors by simulating statechart diagrams together with checking concurrency, and it also has a unique capability of generating a sequence diagram as a simulation log [4]. Mocha is a model checker based on the hierarchical state-transition graphs, with features such as graphical user interface and a scripting language [1].

Since the current version of UML does not have enough capability to describe realtimeness, each vendor seems to be expanding the UML syntax which allows to describe more specific actions as well as concurrency and timing for real-time and embedded systems. Realtime UML [2] and eUML (Embedded UML) [16] are the examples of such work. OMG is also working on the issue along with those tool vendors. The (future) standard is called the Action Semantics for the UML [6],[7] and is defined on top of UML1.4 standard [8], but the standardization process is still undergoing.

Integrated Tools and IIOSS

The Rose from Rational Software is one of the well-known integrated tools in object-oriented software development [5]. This tool has many features, such as a model editor, source code generator, C++ support, and round trip capability. However, it does not have a diagram simulation capability.

IIOSS is an object-oriented design and development tool based on UML, developed by the first author and the others, supported by the Information-Technology Promotion Agency, and is available in open source license [9],[14].

Model Editing Facility (MEF) and Model Debugging Facility (MDF) of IIOSS are for modeling and verification in requirement analysis phase and design phase. The system also provides supports for such facilities as translation between models and source programs, interface design for GUI, database and network systems. It also has facility for storing and managing model information. Our work in this paper are related with the facility MDF together with MEF.

CONCEPTUAL MODEL OF OBJECTS AND SIMULATION

In this section the conceptual model and view of objects and simulation of our new approach are given.

Classification of Objects

Fig. 1 shows our terms and classification of objects in our approach.

Model	Model Object	
	UML Model Object	Virtual Java Object
Program	has Skeleton	
	Skeleton Object	Java Program Object
	Program Object	

Figure 1: Classification of objects in IIOSS

1. *model object*
A model object is an object which is described as a model element. It is either a UML model object or a virtual Java object.
2. *UML model object*
A UML model object is an object which is described as a model element and cannot be executed.
3. *virtual Java object*
A virtual Java object is an object in a model world to handle a corresponding Java program object. It has Java object information as well as UML model object information, and is used to invoke the Java class while simulation.
4. *program object*
A program object is written as a Java program which can be invoked on a certain platform.
5. *skeleton object*
A skeleton object is a Java class file generated from a UML model object, which is expected to be used later as a core of an actual program.
6. *Java object*
A Java object is a Java class. Once the system reads in a Java class file and imports it into a UML class, the user can use it as a model element.

New Features for Simulation

(i) Simulating behavioral diagrams

Since there are four kinds of behavioral diagrams which are used to describe system's dynamic behavior, the system works not only on statecharts but on all the four behavioral diagrams, in contrast with other tools which work only on statecharts.

(ii) Message passing between objects

The capability of sending and receiving messages between class objects is one of the most basic concepts on object oriented software programming to realize "object" and its independence. To realize this in modeling and simulation, the system has a message passing facility between model objects. The objects are not necessarily in a single diagram.

The simulator evaluates a message specified as an attribute of a model element, with a command starting with the character “^” as an action. The format of an action is:

^ ClassName . ActionName (arguments)

The user can specify it in an effect association of the transition object in a statechart diagram. An *effect association* is an attribute in which the user can specify actions when a transition is made.

This allows a user to simulate models closer to programs, and also helps to develop large scale models. Since software system is getting bigger recently, the capability of simulating large-scale complex models is inevitable.

(iii) Simulating on diagrams with different types

There are four types of behavioral diagrams to describe program behaviors in UML specification. To use four behavior diagrams more effectively, the system has the correspondence feature. The user can define a *correspondence* between a model object and other model objects on another diagram. This allows the user to specify decomposition or elaboration of model elements. During simulation the system shows a link.

What a user needs to do in order to use this functionality is: the user selects one object in a diagram and chooses “Start Corresponding One” in popup menu, then selects an object on any diagram. The user also needs to specify “End Corresponding” point in the same manner. Note that the user can select more than one model element as the end point. When the simulator arrives at the first object, the simulator jumps to the corresponding objects specified as “Start Corresponding” to continue simulation. Once the simulator reaches “End Corresponding” object, the simulator returns to the first object.

(iv) Simulating with real programs

The system has a capability to handle a real Java program as a virtual model element. What the user needs to use a Java program is to register it to a model element. After that the user can send a message from the model element to the Java program to use it.

When the simulator arrives at the virtual model element connected to a Java program while simulating models, it actually invokes the Java program with a message as a parameter. When the execution of the Java program was finished, the system will receive a message from the Java program and can use the return value to continue the simulation.

(v) Asking decision to the user

Usually models are written more abstractly than real programs. The user may want to simulate models before specifying its details sufficiently. When a model element has more than one outgoing relation, the system automatically asks the user to make a decision without specifying stubs if it cannot

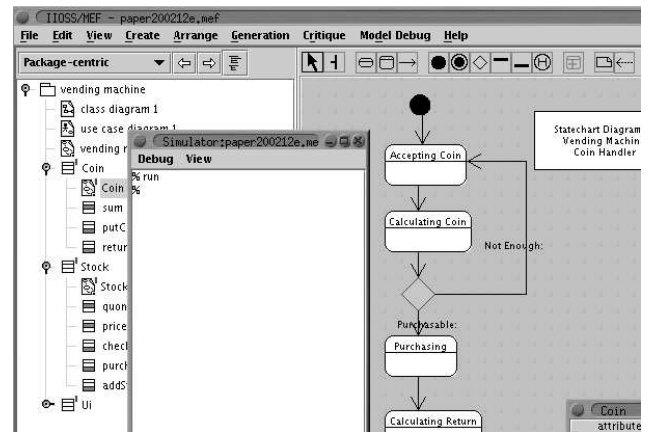


Figure 2: MDF in IIOSS

find where to go next. The system helps the user by showing the possible ways to help him/her.

Process of Simulation of UML Models

The UML models are simulated in the following steps.

Step 1. The user selects a starting behavioral diagram to simulate and then starts simulation by specifying the label “model debug.” Two new windows appear: one is the command window to control simulation, and the other the history window to show all logged messages. The user can choose a class diagram to simulate statechart diagram corresponding to the class. The user can set break-points on model elements. The system stops at the points when the simulation arrives at them.

Step 2. The system starts simulation by the user’s direction, and finds the start point. If variables described in a diagram are set or referred to during simulation, a window appears showing a list of variables. The current model element is highlighted during simulation.

Step 3. Simulation proceeds in each of the following cases.

(i) Transition within a single diagram

A transition from a current model object to the next model object is made if the model object has one and only one outgoing transition. For example, in a statechart diagram the model object is either a state, a box or a transition with an arrow. If the current model object is a state and has only one transition or the current model object is a transition, then it should be connected to a state at the end of an arrow.

If the model object has a break-point, the system stops simulation at the point and ask the user about the next move. If there is no break-point but the element has a guard condition, then the system evaluates the condition and makes a transition. A guard condition is a conditional expression defined as an attribute of an object. It is evaluated when the instance

of an object is performed. If the result is true, a transition will occur.

After that and if the simulator still cannot decide the transition uniquely, a list of possible transitions is displayed to help the user to make decision, and he/she can choose one of the transitions from the list.

(ii) Transition to another diagram by message passing

If an object has a message sending to another object, the system performs the transition according to the message, and moves the current object to the object which has received the message. This object receiving the message may be in a different diagram. The current diagram is changed also when processing moves to another diagram.

(iii) Transition to another diagram of different type by corresponding objects

If an object has an attribute called *correspondence*, the system performs the transition while simulation. A current diagram is moved when processing moves to another diagram. The simulator returns to the original diagram when it reaches the last object of the correspondence.

(iv) Invocation of a Java program (class)

When a model object under processing has the information that it is a virtual Java object, the corresponding Java program object on a real file is invoked. After this program is completed, it returns to the model object which has called it. If there is a return value from the Java program it is passed to the model object.

Step 4. When the simulation arrives at an end position, the system stops. When it is not an end, the simulator returns back to Step 3 to continue simulation.

Fig. 2 shows a screenshot of the IIOSS MDF during simulating a diagram.

Limitation on real objects

The system currently supports only Java program as a program which the system can invoke. Moreover, the argument types are limited as follows:

void, Boolean, char, byte, string, short, int, long,
float, double

Because of this limitation the system cannot handle reference objects, and also a real object cannot invoke a model object.

IMPLEMENTATION OF SIMULATION

In this section we state some important points in implementation for simulation.

Simulation Resources and Engines

The simulator works on top of MEF, the UML editor subsystem in the IIOSS system. The simulator deploys the information needed to simulate on what we call *Simulation Layer*. The simulation layer is a space in which all the instances in the simulation are working. The system shares the information that the editor MEF has without copying them. In

the way above, the user can work easier and simpler, since the user may not need to reinvoke the simulator after he/she changes some of diagrams.

When the simulator starts, the package named *org.iioss.mdf* is activated. It obtains information for model objects and their attributes described in the current diagram using a package named *uci.uml*, the UML editor (which was originally developed at Univ. of California, Irvine). Then it deploys all the information concerning resources of diagram elements onto the simulation layer. And it also builds a table to simulate, such as the current diagram and the current model object, as well as a set of breakpoints, variables and their values. It also handles pointers to program objects, commands interpretation, input and output by the user, a message passing emulator, an invocation of Java program, and receiving its return value with possible type conversion.

When the simulator moves from the current model element to the next element, it checks the relationship of the current model element. If it finds an outgoing relationship without violating any constraints,

- . it reverses the view of the current model object,
- . it moves the current simulation position to the next model object,
- . it reverses the view of the new current model object, and then
- . it sleeps for two seconds for padding.

If it cannot find one outgoing branch, it asks the user to make a decision.

We have actually prepared two different simulator engines, one for statechart and activity diagrams, and the other for sequence and collaboration diagrams, owing to the basic differences of behavioral natures between the two groups of diagrams.

Message Passing between Model Objects

The simulator has a message passing emulation capability in the package *org.iioss.mdf*. When the simulator finds a message to another model element described in an attribute of the current model element, the system passes the message to the emulator together with variables, their values, the pointer of the destination model element, the name of the diagram and the name of the model element.

The message passing emulator finds the destination element and passes variables and values to it. Then the system changes the current diagram and the current model element to the destination element if the message passing emulator succeeds. The system shows the new current diagram as the current diagram by asking the change to the *uci.uml* package. The system also activates this feature to jump to another diagram if the current model object has a correspondence.

EXAMPLE OF SIMULATION

To show the flavor of the simulation a small example of simulation of a model with an activity diagram and two state-

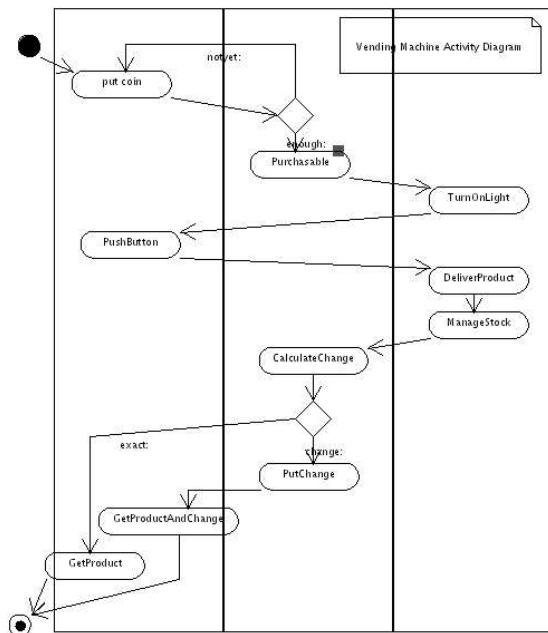


Figure 3: Activity diagram [A] of SVM

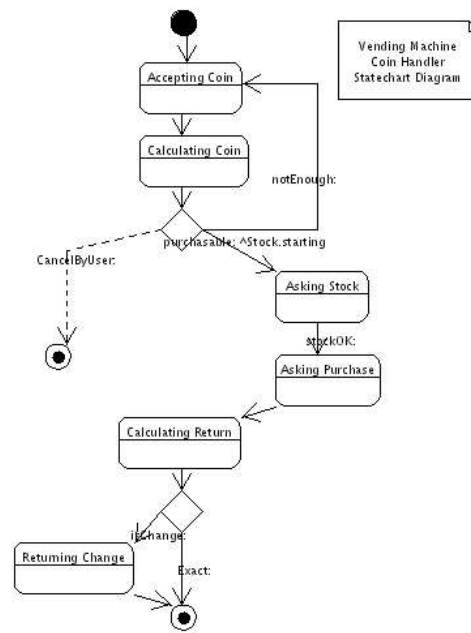


Figure 4: Statechart diagram [B] of SVM coin handler

chart diagrams are presented. The model is derived from a description of a simple vending machine.

Diagrams of the Model

There are three diagrams: an activity diagram of the Simple Vending Machine (SVM for short) [A] of Fig. 3, a statechart diagram of Coin Handler of SVM [B] of Fig. 4, and another statechart diagram of Stock Handler of SVM [C] of Fig. 5.

The activity diagram [A] shows common human activities while purchasing a product from a vending machine. The statechart diagram [B] shows how the vending machine SVM handles deposits and changes of coins, and the statechart [C] shows how SVM manages its stocks. Observe that in these diagrams the levels of abstraction are not exactly the same.

Simulation of SVM

We choose the activity diagram [A] for the starting one of simulating the model. To start simulation choose “Start” button from “Model debug” in the menu bar. Then two new windows appear; one is the Simulator Window which is used to control simulation, and the other the Trace Log Window for showing all logging messages. Simulation starts when the user selects “Run” from the Debug menu (or types “run” in the Simulator Window).

The simulator finds the begin state (a large dot) in the diagram [A] and starts simulation. From now on it always show the current position by highlighting the object. Next it moves the current position to an ActionState object as a result of following an arrow of outgoing transition. The current object now moves to the object “put coin.” All variable names

referred to and values set in the current diagram are always shown in a table corresponding to the diagram.

Then the simulator moves to a branch object, the diamond, meaning “EnoughDeposit.” The branch has two outgoing transitions “enough” and “not yet.” Since there is no information given in this diagram, e.g., diagrams describing user’s action, what kind of coin the user puts, nor prices of stocks in the vending machine, the system cannot find any attributes (conditions) which should be taken. Thus it suspends the simulation and asks the user for a decision, showing the possible selections in the popup menu. In this example “enough” and “not yet” are shown in the popup menu, and if the user chooses “enough,” then the system resumes the simulation

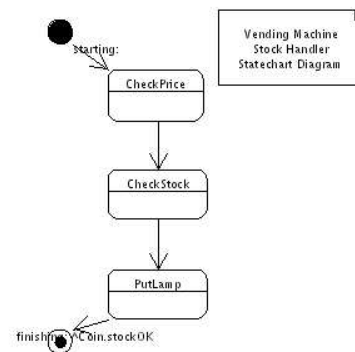


Figure 5: Statechart diagram [C] of SVM stock handler

and moves to the next object.

When the system reaches to “Purchasable” the icon has a small red rectangle on the top right corner which means that the object has an attribute “correspondence”. The simulator now jumps to the corresponding objects specified as “Start Corresponding” to continue simulation. When the simulator later reaches one of “End Corresponding” objects, the simulator returns to the previous object “Purchasable”. Now the current diagram moves to diagram [B]. The system finds the begin state in [B] which is specified as “Start Corresponding”, and continues simulation.

The user can inspect in any time all the necessary diagrams by double-clicking “diagram” tab attached to the diagram, which is useful to separate a current diagram from the main diagram window.

When the simulator arrives at “Purchasable” object in [B], it has an attribute of effect association shown in a list of attributes. In general, when the system moves to a model element which has an effect association, it evaluates the association and invokes the result as an action. In this case the result is to jump to diagram [C]. Then the system shows diagram [C] replacing diagram [B]. When the system reaches the end state of [C], it returns to “Purchasable” in [B] and continues simulation. And finally when the simulator returns to the end of activity diagram [A] it stops simulation with success.

In this specification consisting of the three diagrams, assume that we add, in the state-chart diagram [B], an arrow expressed in a dotted line in Figure 4 standing for the case of cancellation of buying an item. This change is of course legitimate within the single diagram [B]. However, when we simulate the three diagrams as a single system with this addition, we would face and find a strange behavior in the activity diagram [A] since the specification of [A] does not yet conform to the change, so that we can correct [A] in order to match the change of [B]. This is a typical example of an advantage of simulating a specification consisting of more than one diagram.

One may use a Java class program for the Stock Handler, or may have a skeleton object that has been specified in a class diagram of the model. The user can seek whether the model has a skeleton object by pointing the Stock Handler class and select “Type” in the “Object config” menu. In this case the Stock Handler class has a Java program as a skeleton object. When the simulator moves to the class in the statechart diagram, the system invokes the Java program.

CONCLUDING REMARK

The new mechanism using the five capabilities of dynamic simulation can give an advance warning to a software designer as to where semantic anomalies may lie within the

model. Because a software designer can find out potential semantic problems of his/her design well before actual programming starts, our mechanism seems to have a practical application in software development.

REFERENCES

1. R. Alur, et al. : jMocha: A model-checking tool that exploits design structure, Proc. 23rd Annual IEEE/ACM Int. Conf. Software Engineering, IEEE Computer Society Press, pp.835–836, 2001.
2. B. Douglass : Real-Time UML, Second Ed., Addison Wesley Longman, 2000.
3. M. Fowler, and K. Scott : UML Distilled, 2nd Ed., Addison Wesley Longman, 2000.
4. K. Ikeda, and T. Kishi : Modeling environment for beginners (in Japanese), in OO Symposium 2002, Kindaikagakusha, 2002.
5. I. Jacobson, G. Booch, and J. Rumbaugh : The Unified Software Development Process, Addison Wesley Longman, 1999.
6. S. J. Mellor, S. Tockey, R. Arthaud, and P. Leblanc : Software-platform-independent, precise action specifications for UML, UML99 at Colorado, USA, 1999.
7. Object Management Group : Action Semantics for the UML, OMG ad/01-03-01, 2001, OMG TC March 2001.
8. Object Management Group : OMG Unified Modeling Language Specification Version 1.4, OMG ad/01-02-13, 2001.
9. Open Technologies : Specification of the Development of Object Oriented and Prototyping Environment (in Japanese), Information-technology Promotion Agency, 1999.
10. M. Sano, M. Yamada, and T. Hikita : Semantic checking for the UML models in the IIOSS system (in Japanese), OO Symposium 2002, pp.109–112, Kindaikagakusha, 2002.
11. S. Shlaer, and S. Mellor : Object-Oriented Systems Analysis – Modeling the World in Data, Yourdon Press, 1988.
12. L. Starr : How to build Shlaer-Mellor Object Models, Prentice Hall, 1996.
13. P. Stevens, and R. Pooley : Using UML – Software Engineering with Objects and Components, Pearson Education, 1999, 2000.
14. S. Suzuki, A. Kurahone, M. Sano, and I. Kakinohana : IIOSS (in Japanese), Ascii, Tokyo, 2001.
15. Software Engineering Research Laboratory, ICS Department, University of Hawaii at Manoa : UP (User Interface Prototyper), <http://www.ics.hawaii.edu/>, 1987.
16. H. Watanabe, K. Horimatsu, M. Watanabe, and K. Watamoribe : Embedded UML (in Japanese), Shoeisha, 2002.