Intermediary Module Incorporation Technique For Verification (Tool Description)

Vaibhav Shrivastava
Motilal Nehru National Institute of Technology, Allahabad, U.P. 211004, India +(91) 9935751724
vaibhav87s@gmail.com

Ankit Marwaha
Motilal Nehru National Institute of Technology, Allahabad, U.P. 211004, India +(91) 9008477558
mailmarwa@gmail.com

ABSTRACT
With the advent of complex software model development, emphasis has shifted to revision of an existing model instead of building it from scratch. The visual model, for generation of executable code, requires an intermediate representation that is reconstructed during model revision. This leads to a processing overhead when revisions increase manifold. Making incremental modifications, directly to the intermediate representation of the original model, can eliminate this drawback. Thus, time and space needed to parse the model and extract the details for each revision is saved.

In this paper, we implement a revision-incorporating technique of a software design model into an existing version. Initially, the Code Extractor parses the XML file representing the UML model, and extracts the relevant information into a set of tables. A Graphical Interface introduces revisions to the original model. The Module Binder combines them with the original tables to create a new set of tables for the revised model, eliminating the need to parse the entire XML file again. Finally, Validation module helps the designer verify the behavior of the revised model. If satisfied, the designer can approve the model for the code generation of software development.

The proposed technique is being implemented as a tool. It forms a part of our semester project. Presently, we are using Visual Paradigm for UML modeling, but it can be done using any other platform. A user-friendly interface for including the revisions to the model has been developed. Our tool can be used to select a suitable revision from a set of revised models by helping the designer to find variations from the desired behavior.

Categories and Subject Descriptors
D.2.13 [**Software Engineering**]: Reusable Software – *Reuse models.* D.2.4 [**Software Engineering**]: Software/Program Verification – *Model checking*

1 INTRODUCTION
Over the recent years, systems were growing rapidly in complexity and required more and more collaboration and a solid durable design quality. Early modeling Methodologies provided a way to cope with complexity, encourage collaboration and improve design in all aspects of Software development. Proliferation of various isolated modeling solutions and related problems led to development of UML as a well established standard.

1.1 Important Aspect
Relationships between the components of a system are grasped more easily when the design is represented graphically using a modeling language. Both the static and dynamic aspects of a program can be represented using a UML model.

With the use of automated test suites one can verify the accuracy of a model. A fully executable UML mode can be deployed to multiple platforms using different technologies.

During the design stage there are many changes in the model. At the final end we do receive an impeccable model which can be used by the above software for verification. There is no facility which would be useful in the preliminary stages.

1.2 Revision Problem
Most of the current software developments require the developers to incorporate new incremental features to existing models, rather than engineering softwares from base. These new features are modular in nature, so that the internal functioning of this new module can be looked upon specifically without having to refer to the existing combined remaining modules (residual model). This module is in fact a component that interacts with the residual model through interfaces. The modules possess the properties like low coupling and high cohesion. This involves inclusion of modules right from the design phase.

1.3 Usage
UML modeling tools generally support the generation of skeletal implementation code either directly or by exporting models in a standardized format, such as XMI, that can be used by third-party tools. Tools for the generation of code from model descriptions are valuable in helping developers maintain consistency between a model and its implementation, which may involve a large number of source files compared to size of the model.

Now when a module is added to the existing UML the UML may be appended, but the entire UML model is transformed to an XML file that has to be parsed again. As such, the previously extracted information is not utilized and instead the steps are repeated as if we create a structure each time from scratch. This is an overhead which is a bane if the new model being incorporated undergoes repetitive modifications, if there are flaws in the design of the new module.

We have developed a tool that would obtain only the relevant data from the new module with the help of a user interface and fuse it with the previously extracted data structure. The form based interface facilitates the user to feed in necessary values that are the specifications of the new module. The key features required to combine this new module with the existing model are also input through the interface such that, the module is combined at the appropriate point/points with the existing model.

This eliminates parsing and uses the combined structure to test for the validity of the module and its incorporation. The design could be finalized if behavior and activity of the model is sound. These modules may be implemented in different languages and may execute on different hardware. In this paper, we explain the proposed technique and working of the tool that has been developed. For explanation we have referred to the usage of State Diagrams throughout the paper.

2 PROPOSED TECHNIQUE

Various commercial platforms like Rational Rose and Visual Paradigm support UML diagram modeling and also XMI generation. The platforms generate a XML file which contains all the data represented by the UML model and all details in concern.

For Example, if the UML is a State Diagram, the relevant elements in the XML file would be States and the attributes would be the name, id, preconditions, post-conditions

The outgoing and incoming transitions would be in sub-elements to-transitions and from-transitions respectively of the element state.

For different platforms the structure of the XML page shall be different although basically the information stored will be the same. The difference in the element and attribute names can be easily understood from the DTDs.

The Code Extractor module parses the XML file. The implemented module is specific to the UML tool due to the platform dependence of the XML file as described above. However, with minor changes, the module could be tailored for any other UML tool.

The Code Extractor Module extracts the information form the XML file and stores them in a set of dynamically declared tables. Different UML diagrams have different elements which have different attributes. Hence there is diagram specific information.

The new module to be affixed with this existing model is fed in by the user through the User interface that has been provided for direct fusing of the module with the extracted data of the model. The specifications for the internal actions, transitions and states of the new module are inserted by the user through the interface. This includes all information about the states of the new module, their properties and the transitions between these states. Along with this information, transitions into or out off this new module are fed in through a dialog box, which require the user to input the pre-conditions and post-conditions. This signifies the states which are the corresponding end in the existing model for these external transitions from/to the module being appended. The user could supply the exit post-conditions for outgoing transitions and the entry pre-conditions for the incoming transitions (for this new module) as described in detailed later when the User Interface Discussion takes place. This graphical modular data represented in the interface are interpreted by the module binder and fused directly to the existing tabular structure created earlier.

The appended set of tables is the internal representation which serves as the input for the validation module. This validation module takes fresh scenarios as input along with the appended set of tables and tracks down the behavior of the combined model diagram.

Finally, *Validation* module helps the designer verify the behavior of the revised model. The implementation of the functionality of the model is totally based on methods which correspond to the various events which occur. Hence, whenever there is a message pass in the sequence diagram the corresponding event gets called, updating the values of variables. Conditions regarding the states and the changes in states and variables can be displayed to the user, thus enabling him to see the complete flow of the program from his object oriented design.

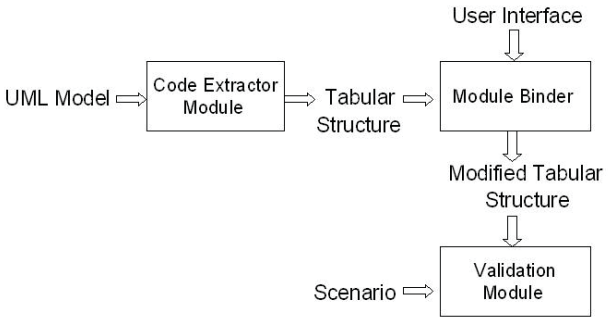

Figure 2-1. Modular Design of the Tool

The output displayed could be another UML 2.0 diagram that tracks the behavioral aspect and depicts the User perspective (Sequence, Activity and Use Case), skeletal code or even an executable code (if this tabular structure is modified such that it conforms to an input to a commercial tool for automatic code generation).

3 IMPLEMENTATION

Though the concept can be used on any tool as stated earlier, we have implemented the working particularly for Visual Paradigm. The corresponding XML file created for the UML 2.0 Diagram is parsed with the help of the JDOM parser [2]. The input is a state diagram for our tool. The state diagram consists of States and Transitions. Transitions would possess transition identification number, name, from-state identification number and to-state identification number. Thus a table tuple entry relates a unique transition (represented by the transition id) with a starting state (represented by a the state id in the column from-state) with a finishing state (represented by the state id in the column to-state) as shown in table 3-1.

Table 3-1. Transition Table Information

Transition id	From-state	To-state

States would have data namely, Name, identification number, a bit indicating whether or not state has sub-states, Entry, Exit and Do Activity , each of which is categorized further into, Pre-conditions, Body and Post-conditions. These are the properties of the state. The State structure also contains a linked list of Transition id and the to-state. This linked list is a list of all the transition ids that leave the particular state along with the state id (stored in the to-state) to which the transition proceeds as shown in table 3-2. This structure has been developed in Java Eclipse (version:3.1) platform[3]. The new incremented module is coalesced by data supplied through a form based interface which we have developed in Java.

Table 3-2. State Table Information

State Id	State Name	Complex Bit*	Entry	Do Activity	Exit

Transition id	To-state id	Link to next node

It is the interpretation done by the Module Binder of the data fed in through the interface that unifies it to the above existing tabular structure and removes the parsing overhead (had the new module been appended in the UML 2.0 tool itself). Using the table of existing states, the Binder adds the various transitions and includes new states of the new module to as specified by the user to the former table.

The Validation Module takes some input cases, which could be values of input variables and exhaustive scenarios that demonstrate the effective behavior of the tabular Structure and hence the Combined Module. The Validation process is visible to the user, as a flow diagram of the state diagrams.

The semantics of UML statecharts allow for the possibility of non-determinism in state transitions: Multiple transitions, triggered by the same event, may be enabled for firing from the same source state. This research does not handle such cases as we assume the statecharts to be deterministic. Though non-determinism may also result from concurrent state machines interacting with one another, testing the interaction among concurrent state machines belong to the realm of integration testing and is thus out of the scope of this research.

3.1 Illustration using an example

The proposed idea and the working of the tool could be realized more effectively using a real life state diagram as an example. Figure 3.1-1 shows a state diagram for a simple ATM model.

3.2 Initial Model

State A : The initial state of the ATM machine (waiting for a card input).

State B : Card entered .

State C : Cash Withdrawal.

State D : User authenticated.

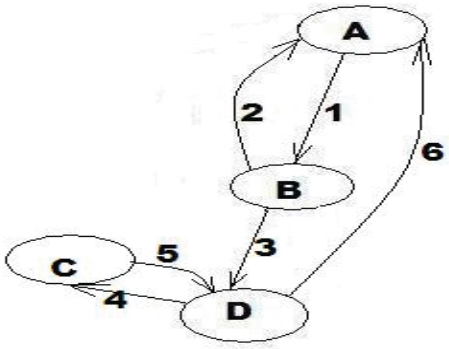

Figure 3.1-1. A Simple ATM model

As the card is entered (an event), there is a transition from state A to state B. Now state B has two self-initiated events. If the card is found out to be invalid, the transition 2 takes place otherwise, if the card is found out to be authentic, the transition 3 to state D takes place. At state D the user can initiate one of the two possible events. The logout event triggers the transition 6 or the withdraw cash event triggers the transition 4. At state C, the user feeds in the required amount and the corresponding cash is delivered to the user. Of course, this model does not consider the details like verifying the semantics such as if the withdrawal amount is greater than the balance as the emphasis is on the model as an example to visualize the tool advantages and not a fool proof transcript of the working of a secure ATM.

This model is represented in the UML tool. The XML representation of this model is parsed to obtain all the relevant data in the tabular data structure as represented in tables 3-1 and 3-2.

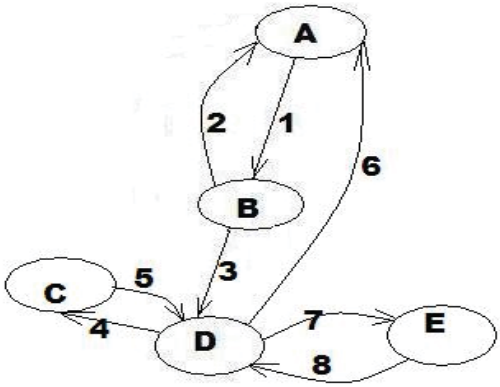

Figure 3.1-2 A modified ATM model with Balance-Inquiry facility

3.3 New Module

State A : The initial state of the ATM machine (waiting for a card input).

State B : Card entered.

State C : Cash Withdrawal.

State D : User authenticated.

New State E : Balance Inquiry.

Now, this working model has been implemented in an ATM which is working in a way it is desired to. But supposing, a situation arises to provide a facility to the user to inquire the present balance in the account before withdrawing the cash. This requires the model to be appended as shown in figure 3.1-2 which is the original model combined with the Balance-Inquiry model. In the logged in state (State D), an additional outgoing transition 7 which is triggered if the user wishes to inquire the balance. At state E, the balance is displayed on the screen and the user can trigger action 8 by pressing a "back" button which facilitates in a transition back to the logged in state home.

Redrawing of the model would definitely take time and for larger commercial models too, even appending the UML 2.0 diagram possesses the parsing overhead. As explained in the proposed technique, the solution is to directly add the data relevant to the Balance-Inquiry module using form-based user interface. In the Interface, we enter the State E and all its properties that would have been entered had state been incorporated in the UML diagram. These would include values needed in table 3-2.

3.4 Transition Incorporation

In order to combine this state (which is the module Balance-Inquiry in this case) with the existing model, we use the theory that a Transition exists from a state X to a state Y if the state id of X is in the Entry pre-conditions of state Y and the state id of Y lies in the Exit post-conditions of state X.. Thus the module may be appended (which may have as many number of states as it requires). But the state from which transition to this module takes place must have a state of this module in its exit post-conditions and so should be the correspondence for transitions leaving the module. Thus for this Balance-Inquiry module, the Exit post conditions of state E would be State D and the Entry Pre-Conditions of the state E would be State D, as evident from the diagram. Similarly data for State D is updated with it including state E in its Entry Preconditions and exit post conditions.

3.5 Validation

The main implementation issue is as to how the state changes. There is a current state variable which is initialized to the first state. Based upon this initial value of a state, we scan every event which occurs and check whether this corresponds to the exit conditions of the current state, the entry conditions of some other state. Also, if there exists a transition between these two states, then it is evident that the event causes a state change. Thus one can obtain the value of the new state. Hence as the model is being executed one can actually realize whether the flow from one state to another is conforming to the idea as presented by the user. Thus the new model incorporating the module can be validated as per the aim of the new model.

4 APPLICATIONS AND ADVANTAGES

The prime apparent advantage is that parsing is eliminated when a new module is incorporated. This procedure is carried out if the design model has not been confirmed by the developer. Thus it provides a methodology for verification of a designed module that is combined. This factor relies on the technique that previously extracted data is utilized. For coarse-grained or module-level concurrency, where modules are independent units of computation that interact by few calls, show evident results, though modules that interact greatly with the rest of the program would demand the user to carefully link the interaction paths.

For an easier and faster implementation of changes, the developer can actually test his changes which are to be incorporated with the main model. A situation wherein the developer has to make all the changes, only to find certain scenarios and left uncovered would be undesirable, for he would have to again remodel the changes in the main module. There can also be scenario wherein there are several models for the same implementation and the developer would like to test and compare how functionally correct and complete each of the model is. We believe our tool will become useful under these circumstances.

5 FUTURE DEVELOPMENT

There is immense scope for the development of this proposal. Though the implementation has been accomplished for state diagrams yet exhaustive exploitation of this technique could widen the usage for all the remaining UML 2.0 diagrams. The Structure made for the storage of extracted data could be modified such that it serves as an input to an existing third party tool. Thus, code could be generated without any XML file being developed. The input which is an XML file can be modified which may lead to a more suitable format for data extraction. With more experience and effort further developments may lead to the Interface that has been developed become instrumental as a primary tool for UML modeling.

6 REFERENCES

[1] "Model Driven Architecture", http://www.omg.org/mda

[2] "Processing XML with Java" Addison Wesley by Elliotte Rusty Harold

[3] Eclipse Modeling Framework (EMF) http://www.eclipse.org/modeling/emf/?project=emf

[4] I. A. Niaz and J. Tanaka , G. 1997 An Object-Oriented Approach To Generate Java Code From UML Statecharts. International Journal of Computer & Information Science. Vol. 6, no.2 2005.

[5] ArgoUML - "Code Generation from Statecharts specified in UML" Tiziana Allegrini.

[6] "An UML-XML-RDB Model Mapping Solution for Facilitating Information Standardization and Sharing in Construction Industry"I-Chen Wu and Shang-Hsien Hsieh

[7] "A Proposal for a Code Generator based on XML and Code Templates" Andreas Rausch

[8] "Generic XMI-Based UML Model Transformations" Jernej Kovse, Theo Härd