Transforming UML 'Collaborating' Statecharts for Verification and Simulation*

Patrick O. Bobbie, Yiming Ji, and Lusheng Liang

School of Computing and Software Engineering
Southern Polytechnic State University (SPSU)
1100 S. Marietta Parkway, Marietta, GA 30060

[pbobbie, yji, llsh]@spsu.edu, Tel: 770-528-4284

Keywords: Model Checking, UML, XMI, database, Promela, SPIN

ABSTRACT

Due to the increasing complexity of real world problems, it is costly and difficult to validate today's software-intensive systems. The research reported in the paper describes our experiences in developing and applying a set of methodologies for specifying, verifying, and validating system temporal behavior expressed as UML statecharts. The methods combine such techniques/paradigms and technologies as UML, XMI, database, model checking, and simulation. The toolset we are developing accepts XMI input files as an intermediate metadata format. The metadata is then parsed and transformed into databases and related syntax-driven data structures. From the parsed data, we subsequently generate Promela code, which embodies the behavioral semantics and properties of the statechart elements. Compiling and executing Promela automatically invokes SPIN, the underlying temporal logic-based tool for checking the logical consistency of the statecharts' interactions and properties. We validate and demonstrate our methodology by modeling and simulation using both ArgoUML and Rhapsody™, respectively.

1. INTRODUCTION AND BACKGROUND

Due to the increasing complexity of real world problems, it is costly and difficult to validate today's software-intensive systems later in the software development cycle. An early validation requires extensive modeling, verification, and simulation using a combination of tools and techniques at the design stage of the cycle. Among the COTS tools, which we are using are SPIN and Promela. SPIN is a verification tool, which is based on linear temporal logic (LTL) and designed to analyze the logical consistency of concurrent systems, specifically for data communication protocols. SPIN runs atop Promela as its verification modeling language. Promela, an extended C-like language, has constructs for specifying system logical requirements and concurrent behavior.

Statecharts are a variant of finite-state machine models. The charts have one-to-one correspondence or association with UML class diagrams, and describe the dynamic behavior of the objects in a given class. Also, a statechart has modular, hierarchical, and structural properties for specifying and modeling the temporal and stimulus-response properties of real world entities.

We use ArgoUML[tm] to specify and model our target software systems. ArgoUML saves its files in XMI format. Generally, using XMI as an intermediate format to capture the structure of the statecharts enables an easy interchange of metadata between modeling tools (based on the OMG UML) and sharing of metadata repositories in distributed heterogeneous environments. XMI integrates three key industry standards: XML (eXtensible Markup Language, a W3C standard), UML (Unified Modeling Language, an OMG modeling standard), and MOF (Meta Object Facility, an OMG metamodeling and metadata repository standard).

Because XMI is standardized, our tools accept any XMI files (based on other OMG

*This work is supported by the Yamacraw Project[1]

modelers), as input, for subsequent verification (model checking). The verification process requires a transformation of the XMI files into database models to facilitate the traversal and development of an abstract syntax tree for code generation and verification.

There are a number of approaches for mapping XMI data to Promela program. Our solution is to take advantage of the provisions of database technologies. First, we build several relational tables corresponding to the UML statecharts (expressed in the XMI files). We then use different IDs as threads to represent the interconnections of the collaborative statecharts. One main advantage in using the tables is the preservation of the hierarchical structure of the model elements/statechart. Another advantage is the technique we use for parsing the XMI files, which, in the process, purges the redundancy often inserted into the file using various translators.

By extracting static and structural information from the tables, we dynamically embed SQL queries into a 'transformational' program to traverse all the states, transitions, and signal events and generate an abstract syntax tree for code generation. The resultant code from the transformation process is in Promela.

The rest of the paper is organized as follows: Section 2 describes the relevant UML statecharts. In section 3, we describe the overall architectural framework of our project. In section 4, we describe our experiences in parsing XMI file and generating intermediate databases. Section 5 is on the Promela-code generation and verification using SPIN. We conclude in section 6.

2. UML STATECHARTS

Figure 1 depicts a feedback system model for illustrating the methodologies of the research. The model is general enough for modeling various components or modules of feedback control, real-time systems. Thus, the model shows the expandability of our methods for modeling systems as statecharts, the analysis, verification and simulation.

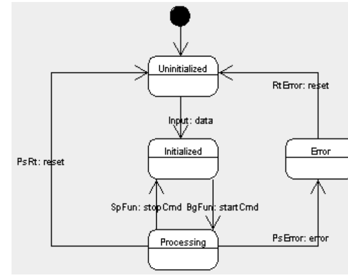

Figure 1: Simple feedback system

The model is expandable in that it can be viewed as a single unit (as a part of a complex system) or it can be viewed as a high-level system, which hides the detail control or behavioral logic inside each single state. For example, we expanded this model by incorporating hierarchy and concurrency into this unit, as shown in Fig 2 and Fig 3.

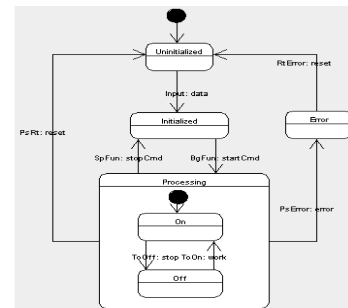

Figure 2: Feedback system with hierarchy

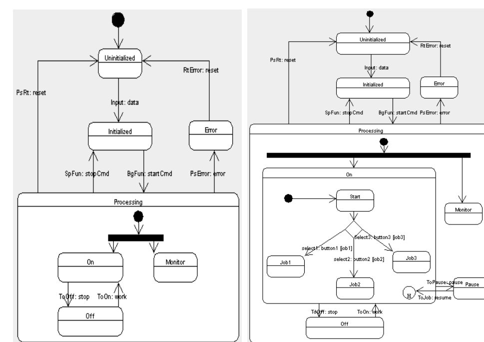

Figure 3: A more complex feedback system

In Figure 2, we expand the processing state as another composite state, which contains a switch. Figure 3 shows two stages of the model: the left model shows a concurrency element (the fork) and the two component subsystems – the *On* state and the *Monitor* state. In addition to these two states, the right model of Figure 3 includes a *condition-selection* state and a *history* state.

3. THE SYSTEM ARCHITECTURE

As far as we know, there are two different UML modeling tools, which generate XMI files from statecharts: one is ArgoUML and the other is Rhapsody™ 4.0. For availability and support in our environment, we use ArgoUML from Tigris [2]. We build the statecharts models in ArgoUML, and the ArgoUML simply generates the XMI file as an output to feed a DOM parser, which we have tailored to create an intermediate tabular representation between the XMI file and our target code. Figure 4 depicts the structure of the XMI files.

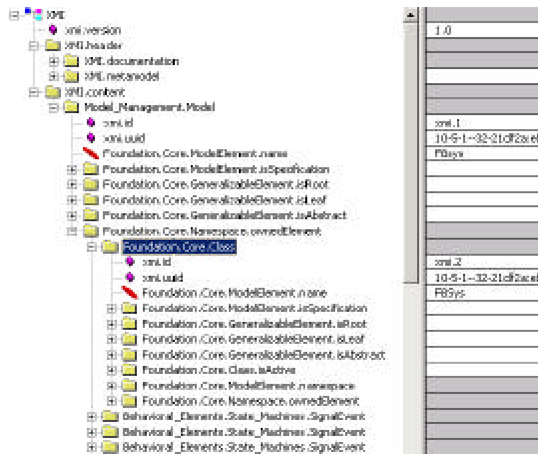

Figure 4: XMI structure for UML Statecharts

We used java Dom parser to parse this XMI file and store all the statecharts elements into set of tables.

Using the inherent definition of the statechart model element (of UML) we built tables for each state in the statechart in Figure 5 and used the 'unified ID' as a link-key to connect the tables. The approach allowed the preservation of the hierarchical structure of the UML statecharts. (See Figure 5 below.) In general, the leaf states, labeled 'States' in Figure 5, may represent either general or pseudo states. Therefore, the tables contain all the information specified in the underlying statechart.

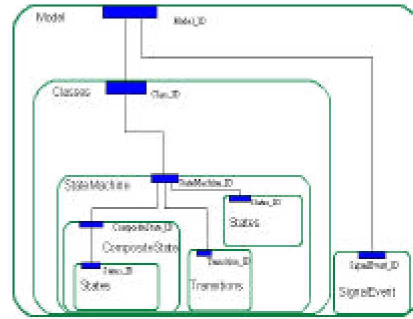

Figure 5: Tabular structure of the statecharts

We have built an SQL-based tool to extract such information as the states, constraints, protocols, properties, attributes, and data specified in the original statecharts from the tables. The extracted information is then used as syntactic elements in the Promela code that we eventually generate for verifying the correctness of the original statechart. Finally, we built a parser to generate the Promela code for the SPIN verification tool.

The whole process of our methodologies is depicted in figure 6.

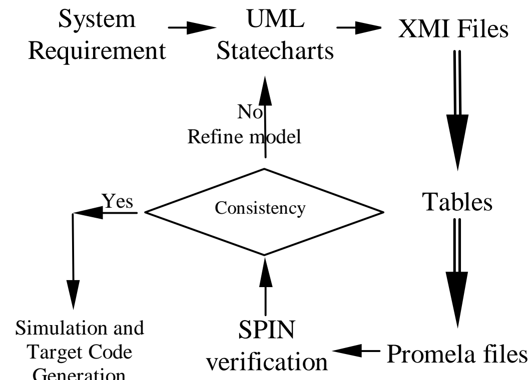

Figure 6: System architecture

4. BUILDING TABLES

As mentioned above, we build eight tables for the UML statechart in Figure 5. The table follows a "7+1" structure: one for each corresponding box/state and one for the pseudo-state. The root-table is the table for the top Model, and its model ID is the overall key for the whole system. All other elements are constructed as sub-tables and are controlled by this model ID. Thus these tables strictly

preserve the hierarchical structure of the UML statecharts.

Since a statechart may contain many different pseudo-states, like initial state, fork/join state, history state, and etc, it is convenient to put all these pseudo-states into one table. Figure 7 shows the pseudo-state table of our example system model.

PSEUDOSTATEID	ISSPECIFICATION	PSEUDOSTATEKIND	COMPOSITESTATE
xmi.22	false	fork	xmi.16
xmi.15	false	initial	xmi.4
xmi.25	false	initial	xmi.16

Figure 7: Pseudo-state table

As indicated before, different pseudo-states can be distinguished by their IDs, and the composite-state they belong to.

A composite-state is a state one level lower than the state-machine state, and it is itself a unit item that may contain sets of states or other composite-states. Figure 8 shows an example composite-table for a composite-state.

COMPOSITESTATEID	COMPOSITESTATENAME	ISSPECIFICATION	STATEMACHINEID
xmi.16	Processing	false	xmi.4
xmi.4	state machine top	false	xmi.3

Figure 8: Composite-state table

In general, a state may contain an entry, activity and exit. Each of these elements of the state embodies specific behavior. So the state tables for these are usually complex and big. Figure 9 shows an example state table.

STATE...	STATE...	ISSPE...	COMP...	ENTR...	ENTR...	ENTR...	ENTR...	ENTR...	ENTR...	EXITID	EXITIS...	EXITIS...	EXITS...	EXITS...	EXITS...
xmi.10	Initiali...	false	xmi.4	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
xmi.13	Error	false	xmi.4	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
xmi.17	On	false	xmi.16	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figure 9: State table

A transition corresponds to a transition edge on the statecharts diagram. A transition may contain items like trigger, guard and action, which are contained in the corresponding transition table. Figure 10 is an example transition table.

TRA...	TRA...	ISSP...	TRIG...	STAT...	SOU...	TAR...	GUA...	GUA...	GUA...	EXP...	EXP...	GUA...	EFF...	EFF...	EFF...	EFF...	ACTI...	ACTI...	ACTI...
xmi.6	Input	false	xmi....	xmi.3	xmi.5	xmi....	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
xmi.8	RtEr...	false	xmi....	xmi.3	xmi....	xmi.5	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
xmi...	PsF...	false	xmi...	xmi.3	xmi...	xmi...	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figure 10: Transition table

A signal-event is always associated with or related to a transition, and causes the triggering of the transition. Figure 11 is an example of a signal-event table.

SIGNALEVENTID	SIGNALEVENTNAME	ISSPECIFICATION	NAMESPACE	TRANSITIONID
xmi.27	data	false	xmi.1	xmi.6
xmi.53	start	false	xmi.1	NULL
xmi.54	startCmd	false	xmi.1	NULL

Figure 11: Signal-event table

The various tables described above serve as the bridge between the XMI files and the target code, which we generate for model checking. The tables also serve as a filter, which automatically eliminates the redundancies in the XMI files. Our methodologies are general in that, using the intermediate tables, we can generate code for other implementations, e.g., SMV-based implementations. In this research, we focused on Promela as the target language for code generation and eventual verification using the SPIN verification tool.

5. PROMELA FILE GENERATION AND SPIN VERIFICATION

Promela is a C-like language for specifying system behavior for model checking. SPIN is a tool for analyzing the logical consistency of concurrent systems, specifically for data communication protocols. Promela code is compiled into an intermediate rule-based program based on linear temporal logic (LTL). The resultant logic program is then verified by SPIN for correctness and consistency. Promela program constructs for specifying concurrent behaviors include *processes*, *message channels*, and *variables*. The process construct specifies the behavior of the system components. Channels and global variables define the environment in which the processes run.

For test runs, we associate each state with one process. The processes of the states run concurrently depending on the trigger events and condition guards. We also represent a fork

pseudo-state with a process, which retransmits the event from its input state to its sub-states.

To gain the hierarchical representation of the statecharts, our Promela programs are also constructed as set of blocks. Each composite-state corresponds to one block of processes. Only the top-level block can be initialized at the beginning. As events are generated and consumed, sub-blocks are triggered accordingly.

Events in the Promela code are extracted from the signal-event tables (see Figure 11). Events from the tables are associated with one particular ID number. Each event channel holds one message at a time. Thus, all process states communicate with each other by sending and retrieving message from the channels. Below is a typical Promela skeleton for the statecharts in Figure 3.

```
#define Processing  0
#define state_machine_top  1
#define startCmd  2
…

/*this is channel definition*/
chan eventChannel = [1] of{int};
chan forkChannel1 = [1] of{int};
chan forkChannel2 = [1] of{int};

/*this is globle variable*/
/* ..  */

proctype proc_Processing()
{
        eventChannel?startCmd;
        eventChannel!Processing;
        run On();
        run Off();
        run Monitor();
        run fork22();
}

proctype proc_state_machine_top()
{
        eventChannel!state_machine_top;
        run Initialized();
        run Error();
        run Uninitialized();
        run proc_Processing()
}

proctype Initialized()  {…}
proctype Error()  {…}
proctype On() {…}
proctype Off()  {…}
proctype Monitor() {…}
proctype Uninitialized() {…}

proctype fork22()
{
        eventChannel?Processing;
        forkChannel1!Processing;
        forkChannel2!Processing;
}

init
{
        run proc_state_machine_top()
}
```

The flow of execution of the processes in the above Promela program (automatically generated based on the XMI representation of the statechart in Figure 3) starts with the top-level *init* process, which initializes the process *proc_state_machine_top*, which is the outmost layer of the statecharts. The *proc_state_machine_top* further triggers all the states, which it encloses. The enclosed process states are the *uninitialized* process, *initialized* process, *error* process and another composite-state *proc_Processing* process. The proc_Processing process also initializes states inside it. The computation process permits a hierarchy or chain of process activation and execution. The root process, *proc_state_machine_top*, is the initial entry for the whole system. As a proof-of-concept, we used SPIN to check the logical correctness and the consistency of the specified constraints and properties in the original statechart, and expressed in the corresponding Promela program. Figure 12 is a screen shot of the output from using SPIN.

Figure 12: Screen shot for SPIN verification

We also developed a GUI wrapper as an interface to our system. (See Figure 13.)

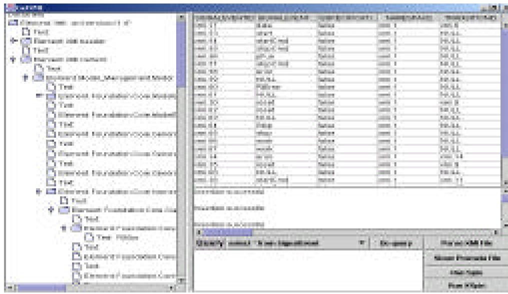

Figure 13: Screen shot for the GUI Interface

6. CONCLUSIONS

In this paper, we describe a set of methodologies for transforming UML collaborative statecharts for verification and simulation. (The simulation phase is currently ongoing within the Rhapsody™ environment.) In this paper, we focus on the techniques for building database tables and a set of translation tools for generating Promela programs from XMI representation of statecharts. We used an integrated development environment (IDE), which comprises ArgoUML, XMI, Java Dom parser, the Oracle™ database system, and SPIN. The IDE provided the necessary tools for the development of our parser and the Promela code generator. Our approach is based on generalized and expandable UML statechart models. Another contribution of our work is the introduction of concurrency among and within statechart models, and the mapping of the underlying behavioral structure into the corresponding Promela code. In this way, the correctness of the protocols (constraints/guards) governing the communication among the states of the statecharts can be verified.

The capability for an automatic generation of Promela program, coupled with the capability for verifying distributed embedded software systems in the early stages of the system design, is promising. We are currently working on additional plug-ins, which will interface with systems like Rhapsody.

REFERENCES

[1] http://www.yamacraw.org;
[2] http://argouml.tigris.org/servlets/ProjectHome;
[3] Holzmann, G. J. 1997. "The Model Checker SPIN", *IEEE Transactions on Software Engineering*, Vol. 23, No. 5 (May);
[4] OMG-2000. 2000. OMG *Unified Modeling Language Specification*, Version 1.3 First Edition. (http://www.omg.org);
[5] (OMG-XML 2000) OMG XML *Metadata Interchange (XMI) Specification*, Version 1.0 June 2000. (http://www.omg.org);
[6] XML and the Document Object Model (DOM),(http://java.sun.com/xml/jaxp/dist/1.1/docs/tutorial/dom/);
[7] A Quick Introduction to XML, (http://java.sun.com/xml/jaxp/dist/1.1/docs/tutorial/overview/1_xml.html);
[8] The Java™ Tutorial, A practical guide for programmers,(http://java.sun.com/docs/books/tutorial/index.html);
[9] ON-The-Fly, LTL MODEL CHECKING with SPIN, (http://netlib.bell-labs.com/netlib/spin/whatispin.html).

Biography: Dr. Patrick Otoo Bobbie is a Professor in the School of Computing and Software Engineering at Southern Poly State Univ., Marietta, GA. He is currently working with the Yamacraw Research group on Embedded Software modeling at Southern Polytechnic State University, Marietta, GA. He is researching methods and techniques for extending UML-based languages as well methodologies for processing intermediate model representations in XML/XMI to support embedded real-time software model verification. His focus is on mechanisms (temporal logic and theorem proving) for specifying the properties or constraints and model checking of distributed embedded software.