Debugging Statecharts via Model-Code Traceability

Liang Guo and Abhik Roychoudhury

School of Computing, National University of Singapore.
{guol,abhik}@comp.nus.edu.sg

Abstract. Model-driven software development involves constructing behavioral models from informal English requirements. These models are then used to guide software construction. The compilation of behavioral models into software is the topic of many existing research works. There also exist a number of UML-based modeling tools which support such model compilation. In this paper, we show how Statechart models can be validated/debugged by (a) generating code from the Statechart models, (b) employing established software debugging methods like program slicing on the generated code, and (c) relating the program slice back to the Statechart level. Our study is presented concretely in terms of dynamic slicing of Java code produced from Statechart models. The slice produced at the code level is mapped back to the model level for enhanced design comprehension. We use the open-source JSlice tool for dynamic slicing of Java programs in our experiments. We present results on a wide variety of real-life control systems which are modeled as Statecharts (from the informal English requirements) and debugged using our methodology. We feel that our debugging methodology fits in well with design flows in model-driven software development.

Keywords: Statecharts, Traceability, Debugging, Slicing

1 Introduction

Model-driven software development is becoming increasingly popular. There exist many tools which enable design specification in terms of Unified Modeling Language (UML) diagrams. Subsequently code is generated from these diagrams either semi-automatically (as in Rhapsody from I-Logix [1] which compiles Statechart models into C/C++/Java code) or manually using the UML diagrams as guidance. Irrespective of whether the code is generated automatically or manually, some of the testing/dynamic analysis is done at the code level. At the UML level, usually verification methods like model checking are employed to check critical properties about the design.

If the testing/debugging of a piece of model-driven software reveals/explains an "unexpected program behavior" how do we reflect it at the model level? This requires us to maintain associations between model elements and code (which are built during code generation), and then exploit these associations to highlight the appropriate model elements which are responsible for the so-called unexpected behavior. We advocate such a method for debugging model-driven software in this paper. The benefits of relating the results of debugging model-driven software to the model level are obvious —

it enables design comprehension and debugging at the model level. Since most debugging tools work at the code level, this forms an important step in enabling model-driven software development.

To make our study concrete, we fix a modeling langauge and a debugging method — Statecharts [2] as the modeling language and dynamic slicing [3, 4] as the debugging method.[1] Given a program P and input I, the programmer provides a slicing criterion of the form (l, V), where l is a control location in the program and V is a set of program variables referenced at l. The purpose of slicing is to find out the statements in P which can affect the values of V at l via control/data flow, when P is executed with input I. Thus, if I is an offending test case (where the programmer is not happy with the observable values of certain variables), dynamic slicing can be performed and the resultant slice can be inspected (at the code level). However, at this stage, it might be important to reflect the results of slicing at a higher level, say at the model level — to understand the problem with the design. We address this issue in this paper.

We consider the situation where the design is modeled using class diagrams and Statecharts i.e. the behavior of each class is given by a Statechart and these Statecharts are *automatically compiled* into code in a standard programming language like Java. We present experimental results on a number of *real-life control systems* drawn from various application domains such as avionics, automotive and rail-transportation. These control systems are designed as Statecharts from which we automatically generate Java code (into which associations between model elements and lines of code are embedded). Subject to an observable error, the generated Java code is subjected to dynamic slicing. The resultant slice is mapped back to the model level, while preserving the Statechart's structure, orthogonality (multiple processes executing concurrently) and hierarchy.

One could argue that, if the models are executable and automatic compilation of models to code is feasible (as is the case for Statecharts) — the debugging should be done at the model level. Indeed, we could build a dynamic slicing tool directly for Statecharts.[2] However, to popularize such tools for debugging model-driven software will require a bigger shift in mind-set of programmers who are accustomed to debugging code written in standard programming languages. Moreover, debugging the implementation is a more focused activity, since it allows us to ignore the bugs in the model which do not appear in the implementation (since the implementation may have lesser behaviors than the model, as is the case when we compile Statecharts to sequential code).

In summary, this paper proposes a methodology for debugging model-driven software, in particular, code generated from executable models like Statecharts. Our proposed methods/tools focus on generating code with tags (to associate models and code), using existing tools and algorithms to debug the generated code and exploiting the model-code tags to reflect the debugging results at the model level. We feel that it is important to develop backward links between the three layers in software development — requirements, models and code. This article constitutes a step in this direction where we relate results of debugging at code level to the model level.

[1] The reason for choosing a *dynamic* analysis technique as the debugging method is obvious — it corresponds more closely to program debugging by trying out selected inputs.

[2] Static slicing of Statecharts has been studied in [5]. Direct simulation of statecharts has been discussed in [6].

2 State-of-the-art in Statechart Compilation

Compilation of Statecharts for generating code has been studied in many research articles. Some of these works, specifically those focusing on embedded system designs, give importance to generating efficient C/SystemC code from State diagrams [7, 8]. Certain other works (*e.g.*, [9] and, to a lesser extent, [10]) generate Java code from full-fledged UML designs consisting of Class Diagrams, State Diagrams and Collaboration Diagrams. *None* of these works support full-fledged model-code association, so lines of generated code cannot be easily mapped back to model elements. In fact, as we illustrate in the following via an example, even the commercial tools for Statechart modeling and code generation do not properly support association between Statechart models and generated code.

Rhapsody and Stateflow are two of the successful tools released by I-Logix[1] and MathWorks[11] respectively, which can generate code from Statechart models. Rhapsody supports all Statechart features and is capable of generating C, C++, and Java code. Stateflow supports Statechart models as part of a complete embedded system design. It supports most of the Statecharts' features, and can generate C code from Statecharts. Given a Statechart with sufficient details, all three tools (Rhapsody, Stateflow and our tool) are able to generate executable code supporting AND/OR-states and event broadcasting. Meanwhile, all three tools provide model-code association to some extent. All tools tag pieces of code with the corresponding Statechart elements information.

However, tags maintained by Rhapsody and Stateflow are not sufficient for supporting full model-code association. The purpose of tags in Rhapsody is to help users refer to model elements automatically while editing the generated code. The tags only associate actions (in transitions and states) and conditions (in transitions). *The code corresponding to events and transition firings is not tagged, and hence there is no direct association for these elements.* Stateflow generates tags on model structure for reference purpose only. Only state entry and state exit are tagged before and after each transition firing. *There is no association existing for events, transitions, actions and conditions.* When a transition is entering or leaving a composite state, all levels of states entered/exited are tagged, instead of the target/source (sub-)state only. Although it shows clearly the execution behavior of a composite state, it increases the difficulty in understanding the triggered transition as well as its source and target states.

The problem with incomplete tags for model element is, we cannot construct a complete trace of the Statechart execution, and hence no systematic analysis method can be applied. After the code is generated, we can perform debugging when an error is found. To enable a comprehensive understanding of the bug report at model level, the code-level bug report should be mapped back to model level. In both Rhapsody and Stateflow, since some model elements are not tagged for model-code association, the model-level bug report becomes incomplete. Our tool is able to build a full model-code association, and it maps bug report from code-level back to model-level.

In the following, we capture the capabilities of the existing tools as far as maintaining code to model backward associations is concerned. We use the popular Rail-car example developed by David Harel and Eran Gery in [12] to illustrate the differences. The example is drawn from the rail-transportation domain and has been widely used as a case study of UML-based system behavior modeling. In this example, there are a fixed

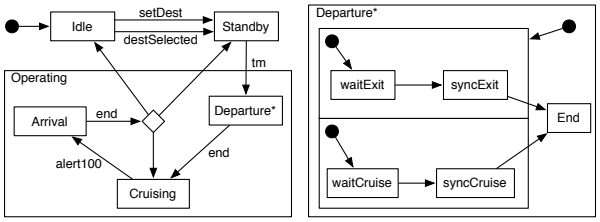


Fig. 1. Statechart fragment corresponding to `car` object. (a) the top-level Statechart, and (b) the details of composite state `Departure`. State `Arrival` is also a composite state, the details of which is not shown.

Our Tool		Rhapsody		Stateflow	
Element Type	Element Name	Element Type	Element Name	Element Type	Element Name
T_action	initial	T_action	initial		
E	destSelected				
T_fire	Idle2Standby				
E	tm				
T_fire	standby2departure			T_fire	standby2departure
:	:	:	:	:	:
:	:	:	:	:	:
S_entry	DepartureEnd	S_entry	DepartureEnd	S_entry	DepartureEnd
E	end				
T_fire	departure2cruising			T_fire	departure2cruising
E	alert100				
T_action	cruising2arrival	T_action	cruising2arrival		
T_fire	cruising2arrival			T_fire	cruising2arrival
:	:	:	:	:	:
:	:	:	:	:	:
S_entry	ArrivalEnd	S_entry	ArrivalEnd	S_entry	ArrivalEnd
E	end				
T_action	arrival2cond	T_action	arrival2cond		
T_condition	arrival2idle	T_condition	arrival2idle		
T_fire	arrival2idle			T_fire	arrival2idle
(a)		(b)		(c)	

Fig. 2. Model-level slices based on the code generated from (a) our tool, (b) Rhapsody, and (c) Stateflow. A dashed line shows a missing model element in the slice resulting from Rhapsody or Stateflow. "E", "T", and "S" appearing in "Element Type" denote "Event", "Transition", and "State". Model elements for `CarHandler` and details inside the states `Departure` and `Arrival` of `Car` are omitted for the ease of understanding.

number of terminals located along a cyclic path. Each adjacent pair of these terminals is connected by two rail tracks, one of which is for clockwise travel and another for anti-clockwise travel of the rail cars. There are several (a fixed number of) rail cars available for transporting passengers between the terminals. There is a control center which receives, processes and communicates data between various terminals and railcars. Each terminal has several car handlers to process transactions between the terminal and cars. More details about the example along with the class diagrams and Statecharts for each class appears in [12].

In particular, we consider the Statechart of a `car` object (shown in Figure 1). Suppose we have a car moving from a terminal to a neighboring terminal (its destination). In terms of the Statechart behavior, the car object is expected to visit states `Idle`, `Standby`, `Departure`, `Cruising`, `Arrival`, and back to `Idle`. Here we use slicing as the debugging method to study how the car finally comes back to state `Idle`. We

set the last occurrence of `Idle` [3] as the slicing criterion and perform slicing based on the car object. As shown in Figure 2, the model-level slice on column (a) is produced by mapping the code-level slice backward using our approach, while the slices on column (b) and (c) are from code generated by Rhapsody and Stateflow. Although code from all three tools have almost identical behavior, our tool is able to produce a complete model-level slice. More specifically, all events and transition-firings are missing in the slice resulting from Rhapsody, which contains only a sequence of actions executed and conditions checked. For example, since the transition between states `Idle` and `Standby` is missing, we have no idea which event - `setDest` or `destSelected` - triggers the car object transiting from `Idle` to `Standby`. In the slice resulting from Stateflow, the transition-firings are only reconstructed from state entry/exit information as well as the model structure. Here also, we cannot determine the transition triggered from state `Idle` to `Standby`. Note that the missing event here (`setDest` or `destSelected`) could be broadcast to other objects (running concurrently), thereby triggering transitions in other objects. Thus, not tracking these events hampers our understanding of the overall system behavior (and not just the behavior of the `car` object in question).

In summary, the existing tools do not maintain detailed model-code associations while generating code from Statecharts. Rhapsody only tags actions (which are executed as an effect of states/transitions) and conditions (which serve as the guard of transitions). Stateflow only tracks the states through which the Statechart moves. *None of the tools track the events which trigger the transitions and are broadcasted resulting in non-trivial communication patterns across the different concurrent objects represented by a Statechart*. These events are often responsible for "unexpected behaviors"; without considering them in our debugging methods (and bug reports) it would be impossible to comprehend concurrent system designs represented by Statecharts.

3 Overall Methodology

In this section, we present the methodology to trace design information between models and code. Specifically, our work consists of the following steps.

- *Forward code generation*. We automatically generate Java code from Statecharts while using appropriate tags to store model-code association information. The Java code can then be used to perform code-level analysis (*e.g.*, debugging via dynamic slicing).
- *Backward code-to-model mapping*. With the debugging result (bug report) from code analysis and the association information obtained, we perform a mapping to produce a model-level bug report, which is more tightly related to the Statechart and also smaller.
- *Hierarchical analysis result*. Although the model-level bug report is easier to understand than code-level report, it may still be large and complex. We utilize the important features of Statecharts (hierarchy/orthogonality) to re-structure the model-level bug report. Furthermore, we separate out the flow of different active objects (from the same class) whose behavior is captured by the same Statechart.

[3] We assume that the execution of Statechart model can be finished by entering an "End" state eventually.

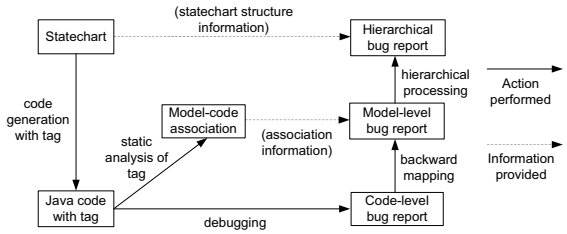


Fig. 3. Maintaining the traceability between model and code.

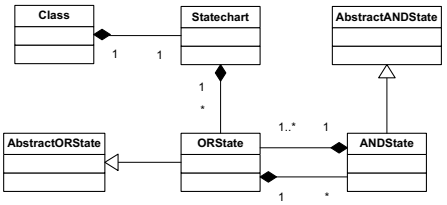


Fig. 4. Class diagram of Java code generated from Statecharts.

The whole methodology is summarized in Figure 3. When a Statechart model is available[4], we can generate code automatically. Since the code is generated completely from the model, we know exactly which part of code results from a particular model element. By tagging this piece of code with the corresponding model element information, we are able to derive the association between model and code. If we encounter an observable error while executing the code, we can use methods (such as slicing) to debug it. With the debugging result (code-level bug report), we can map the bug report backward to model-level by changing statements in code-level bug report to the corresponding model elements. To fully regain the structure of Statecharts, the model-level bug report can be re-organized. The re-organized hierarchical bug report maintains both the structure of Statechart as well as the elements in the original model-level bug report. We now elaborate the intricacies involved in each of these steps.

4 Code Generation

First we discuss how we can maintain tags between model elements and generated code during the process of code generation. It is worthwhile to note that, *we always translate a Statechart to a single-threaded Java program. Thus, event communication at the Statechart level gets translated to method calls at the code level.*

For each class of active objects in the system model, the corresponding Statechart is realized at the software level via several Java classes. As shown in Figure 4, a Statechart contains a set of OR-state classes. Meanwhile, an OR-state class may have several AND-state classes — where each AND-state class corresponds to a concurrently executing component. Each AND-state class may again contain different classes corresponding to the possible (OR-)states in which the system component (corresponding

[4] The states and transitions must be defined, and all appropriate triggers/conditions/actions must be available — such that the system is executable after generating code.

```
1.  public void trigger(Events event)
2.  {
3.      switch(event) {
4.          <% for each (transition t in current state) { %>
5.          case Events.<% transtion t's event %>:
6.              <% if (transition t has condition) { %>
7.              if(<% transition name %>_Condition()) {
8.              <% } %>
9.              <% if (transition t has action) { %>
10.             <% transition name %>_Action(event);
11.             <% } %>
12.             <% transition name%>_Fire();
13.             <% if(transition t has condition) { %>
14.             }
15.             <% } %>
16.             break;
17.         <% } //end for each %>
18.         default:
19.             for each (AND-State as contained) {
20.                 as.trigger(event);
21.             }
22.     }
23. }

24. <% for each (transition t in current state) { %>
25. <% if(transition t has action) { %>
26. /**
27. * @model type=transition_action name=<% transition name %>
28. */
29. private void <% transition name %>_Action(Object parameter) {
30.     <% transition t's action %>
31. }
32. <% } %>

33. <% for each (transition t in current state) { %>
34. /**
35. * @model type=transition_fire name=<% transition name %>
36. */
37. private void <% transition name %>_Fire() {
38.     Create target state object;
39.     make transition;
40. }
41. <% } %>
```

Fig. 5. A fragment of template used in code generation.

to the AND-state) can be in. The design of OR-states within an AND-state follows the State design pattern [13].

While generating code from Statechart models, we mark the lines of code corresponding to specific model elements with the model element name and type. The usual model element types correspond to events, states, transitions, conditions, actions and etc. Note that while generating Java code, each method only contains code for at most one model element. These markers or tags are inserted as *Javadoc* comments in the generated code in the form of: `@model type=`*type* `name=`*name* For example, if a method $meth$ in code corresponds to state $S2$ in a Statechart model, we insert the following comment before $meth$:

```
/**
*@model type=state name=S2
*/
```

The code generation mechanism is implemented using Eclipse framework, which is capable of emitting text files w.r.t. a set of templates and inputs to the templates. Figure 5 shows a fragment of a template used in generating an *ORState* class as in Figure 4, which is writing in pseudo code for ease of understanding. Line 1 - 23 represents the method to dispatch event, and line 24 - 32 and line 33 - 41 represents two methods for transition's action and transition firing respectively. Note that text contained in "$<\%$" and "$\%>$" is to be substitute with the real input - e.g. transition name, code for transition action, and etc. Other text is emitted as is. Each element is written as a method. For example, line 30 will be replaced with the code of transition action during generation. The tag for model element is written in the template as well, with appropriate names to be substitute. Line 26 - 28 shows such a tag for transition's action. Inserting tags as *Javadoc* comments at method level serves several purposes: (a) instead of inserting tag to every statement related to a model element, we greatly reduce the space overhead for tags; (b) Javadoc is a standard documentation format in Java program, and thus the generated tags can be easily processed by other design tools for their own analysis, and (c) it allows us to incrementally change the code, for minimal changes in the Statechart model.

Note that the tags in the generated code cannot be efficiently used for relating code-level bug reports to the model level. Indeed this is the main motivation of our work — debugging model-driven software such that the results of debugging can be shown and communicated to the designers at the model level. Since the tags are embedded inside the generated code as plain text, relating the lines in bug-reports to the model-level will involve expensive file accesses. Consequently, we use the tags in the generated code to build an in-memory representation of the model-code association. The association consists of tuples of the following form: `(Model element name, Element type, Java class file, Line numbers)` indexed by `(element name, type)` and `(class file, line numbers)` separately. Maintaining the model-code associations in-memory as well as in the file for generated code allows us to avoid regenerating the code for minor changes in the model.

Effect of incremental changes. The process of maintaining tags during code generation and building the in-memory model-code association is important for model-level debugging. Once the bugs are found and fixed at the model level, the changes need to be propagated to the generated code. This can be done automatically using the tags, provided the fixes at the model level do not add/remove any model elements. We note that often the bug-fixes involve correcting a wrong condition or a wrong action in the Statechart model. Such changes in the model level only *modify* model elements. These changes do not affect the tags, and thus do not require re-generating code from the modified model. In fact, as long as the structure of the Statechart model (the structure resulting from states and transitions) is not affected, there is no need to re-generate code from the Statechart. Instead we can use existing tags, to directly (and automatically) propagate the changes from the model level to the code level. The in-memory model-code associations can then be re-built on demand from the modified code.

5 Mapping Code-level Bug Reports to Statechart-level

We now elaborate the method for mapping the debugging results of the generated code back to the Statechart model level. Most debugging methods report a list of statements (the *bug report*) that are potentially related to the observable "error". These statements are at the level of the generated code. Recall that our model-code association stored in-memory contains tuples of the form

```
(Model element name, Element type, Java class file, Line #).
```

Thus, we can map a set of statements in the generated code to a set of model elements at the Statechart model level. This constitutes our preliminary model level bug report. The model-level bug report is smaller and more compact than the code-level bug report.

Take the example in Figure 1, where the "car" object visits states `Idle`, `Standby`, `Departure`, `Cruising`, `Arrival`, and back to `Idle` (the states inside composite states are not mentioned here). Suppose we set the last occurrence of `Idle` state as the slicing criterion, which is essentially translated to a number of lines in generated code passed to JSlice. The code-level bug report will consist of a set of `(Java class file, line number)` tuples. Apparently, a number of entries in the bug report

corresponds to one model element. By utilizing the model-code association, we can get a set of model elements as the model-level bug report. For this example, we will have:

```
State Idle, Transition Idle → Standby, State Standby,
       ..., State waitExit, State syncExit, ...
```

Note that in the model-level bug report, all related model elements are reported as a simple set. The hierarchy structure of Statechart is totally disregarded. The designer cannot figure out those more important (more suspicious) elements quickly from the element set. Thus, we need to further re-organize the model-level bug report.

Separating flows from different objects in a class We observe that debugging program generated from Statechart models differs in one significant way from normal debugging of sequential programs. A Statechart model M for a process class can capture the communication and control flow of *several* active objects running concurrently. This is because there might be several active objects in the class whose behavior is captured by M. Consequently, in the model-level bug report, it is important to separate out the relevant control flows of these different objects — so that the designer can trace the source of the observable "error". For example, if a state $S2$ appears in the model-level bug report, it might capture the visit of several active objects of the same class to the state $S2$ (each possibly multiple times). To separate out the control flows of the different objects, we can let our code-level debugging method return *a sequence of statement instances* rather than a *set of statements*. This is possible for popular debugging methods such as dynamic slicing [4, 3, 14] and fault localization [15]. The sequence of statement instances (call it σ_{code}) gets mapped to a sequence of model element instances (call it σ_{model}) using model-code associations. These model element instances may come from different objects; we can project σ_{model} to get the sequences of model element instances for the *different* active objects.

Hierarchical Bug reports Even after we project the model-level bug report for each active object, the bug report for objects are still sets of model elements, which may be huge compared to the entire model for the designer to inspect. In fact, we can go beyond the projection of model level bug report for active objects. Since a Statechart model has a hierarchy structure, the parent-children relationship of states can be formed as a tree automatically. Nodes in this hierarchy tree correspond to *OR-states* in Statechart. Children of a node n are OR-states directly contained by n's AND-states. Note that in the hierarchy tree we do not include AND-states. Usually the model designers are interested in how the model is executed - that is, how transitions are fired between OR-states. Building hierarchical bug report at code-level is studied in [16]. However, the organization of code-level hierarchical bug report may not correspond to the structure of Statechart. Thus, we need to build hierarchical bug report at model-level w.r.t the Statechart organization.

Given a model-level bug report (as a sequence of model element instances), we first project the report to get the sequence of model element instances for every object of class C. This sequence is projected further for each node of the hierarchy tree of Statechart model M for class C. This leads to a bug report which contains the structure of the Statechart model and enables greater design comprehension. Figure 6 shows the

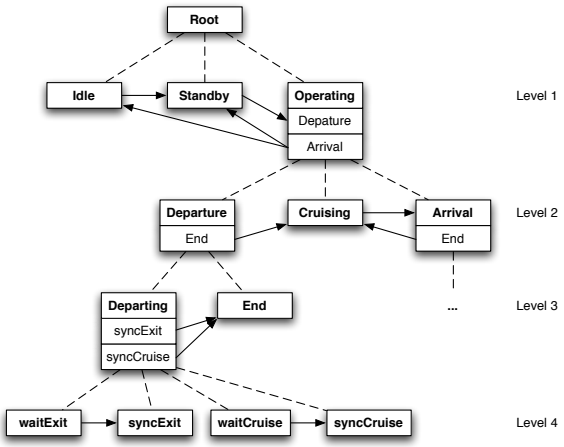


Fig. 6. Hierarchical bug report for the example in Figure 1.

hierarchical bug report (as a hierarchy tree) of Statechart example as in Figure 1. As the top level states in the statechart are `Idle`, `Standby`, and `Operating`, we have three nodes representing these three states at Level 1 in Figure 6. The node `Operating` can be further divided into three nodes as in Level 2, corresponding to the three OR-states contained by state `Operating`, and so on. At each level, we shows transitions (in bug reports) across nodes only. That is, transitions within a composite state is hided for current level, and can be examined by zooming into the composite state. Furthermore, each node (state) may selectively show *sub-states* where there exist cross-node transitions connecting them. For example, at Level 1 in Figure 6, we have transition connecting `Standby` and `Departure` (in `Operating`).The hierarchical bug report can be constructed as follows.

1. project model-level bug report to get the sequence of model element instances for every active object o (object-level bug report R_o);
2. build the hierarchy tree of states T_o for every active object o;
3. prune the hierarchy tree - for a sub-tree rooted at node n, if all nodes in the sub-tree are not in R_o, and no transition connecting them, we can prune this sub-tree in T_o;
4. connect nodes in T_o with all transitions in R_o, and expand node to show sub-states if any transition connects to them. In particular, for each transition $t \in R_o$, we connect it to two states/nodes s_1 and s_2, where
   - $parent(s_1) = parent(s_2)$, and
   - $ancestor(source\_state(t)) = s_1$, and
   - $ancestor(target\_state(t)) = s_2$.

By presenting this hierarchical bug report, the model designer can determine which model state is potentially buggy at higher level, and navigate inside to see the detailed transitions reported for that state, and so on. This approach is more effective to designer than being presented a *long* list of model elements.

6 Experimental Setup

In order to experimentally evaluate our methodology, we adopt and construct four statechart models. These models used are shown in Table 1. The third column shows the number of elements in the statechart model, counting OR-states, AND-states, transitions, actions, and conditions. Except for the RailCar example discussed in section 2, the other three models are based on real-life systems. The automated shuttle system [17] consists of several shuttles running on a railway network. They bid to transport passengers between two stations and earn money upon the completion of the transportation; meanwhile, the shuttles have to pay for the rail network usage. The weather control system is part of the Center TRACON Automation System (CTAS) [18] developed by NASA. It is used to control the air traffic at large airports. The weather controller contains a weather control panel dispatching weather status, a communication manager, and several clients receiving weather information. Such an update may succeed or fail and clients must respond with correct actions. The Media Oriented Systems Transport (MOST) [19] is a networking standard for multi-media devices (such as CD player) communicating in a car network. The network may contain up to 64 nodes, and each node corresponds to a multimedia device. These nodes are known as Network Slaves in MOST terminology. There is a special node called Network Master responsible for maintaining the network information in a central registry. The Network Master scans the whole network upon a change in the network status. Network Slaves may reply with valid or invalid information and further action must be performed (e.g. a re-scan). The MOST standard is currently maintained by the "MOST Cooperation", an umbrella organization consisting of various automotive companies and component manufacturers like BMW, Daimler-Chrysler and Audi.

For each of the above four models, we manually inject four to five bugs, resulting in four to five buggy versions (from each of which code is subsequently generated). These bugs can be categorized as follows.

- *Wrong control flow* - The bug affects states visited, including transition pointing to a wrong state, a condition is tightened or relaxed, or the event trigger of a transition is wrong. These correspond to "branch errors" in the generated code.
- *Wrong action* - The assignment to a variable in the action corresponding to a Statechart state/transition may be wrong. These correspond to "assignment errors" in the generated code.

Statechart	Description	# model elements
RailCar	A rail car system from [12]	121
ShuttleSystem	Shuttles transporting passengers between stations [17]	117
WeatherControl	Updating weather status to clients [18]	202
MOST	Networking standard of multimedia system in cars [19]	277

Table 1. Statechart models used in our experiment

– *Missing element* - The bug results from a missing transition, condition, or action. These correspond to "code missing errors" in the generated code. For bugs of this type, we define the bug in terms of elements existing in the Statechart model. Thus, if a condition or action is missing we mark the corresponding transition as buggy, and so on.

For each buggy version, we manually generate five to ten test cases which are failing runs with observable errors. In other words, the executions of these test cases are different w.r.t the correct version and the buggy version.

We choose *dynamic slicing* [3, 4] as the debugging method to produce code-level bug reports and perform backward mapping to model-level. Given a program P, input I, line of code l and set of variables V — dynamic slicing can find the statements/statement-instances of P which (directly or transitively via control or data flow) affect the value of V at l in the execution trace corresponding to I.

We modify and exploit the dynamic Java slicing tool JSlice [20, 21] from our previous work [14] to produce code-level slices. JSlice is an open-source tool which performs backwards dynamic slicing of sequential Java programs. Since backwards slicing requires storing of the execution trace, JSlice performs online compression during trace collection. The compressed trace representation is traversed without decompression during slicing. The program slices produced by JSlice are mapped back to model elements using the association between model entities and the generated code. The model-level slice is then further processed to produce hierarchical slices which correspond to the structure of the Statechart.

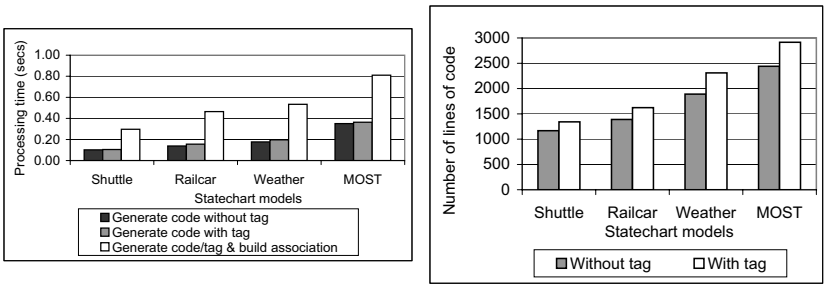
7 Experimental Results

We employ our tool on nineteen buggy program versions (for the four Statecharts in Table 1) to evaluate the efficiency and effectiveness of the methodology.

7.1 Code Generation

Given a Statchart model, we automatically generate a *single-threaded* Java program. While generating code from Statechart model, we also insert tags in generated Java files; the tags are processed to construct an in-memory structure representing association between model and code. Thus it is important to make sure the overhead of tags and building the in-memory association is small enough.

Figure 7(a) shows the time to generate code for the four models. For each model, it shows the time to generate code without tags, the time to generate code and tags, and the time to generate tagged code as well as the in-memory model-code association. The time overhead of tags in code generation is mainly for emitting into files (and writing to disk) and is largely system dependent. Among all models, the time required to generate code with tags increases 3% - 13%, compared with generating code without tag. From the figure, the time for generating code and tags is 34% - 45% of the total time. The remaining time is spent in building the in-memory associations. We recall



(a) (b)

Fig. 7. (a) Time to generate code and build model-code association. It compares the time to generate code without tag, time to generate code with tag, and time to generate code/tag and build association; and (b) The number of lines of code for four models.

that modifications to Statecharts which only modify model elements do not require regeneration of code. Thus, the overhead of code generation is usually incurred only once across several runs of debugging.

The size of generated code is shown in Figure 7(b). The increase in code size due to tags is low — 15% - 22%.

7.2 Dynamic Slicing

After we have the Java code and the model-code association information, we perform dynamic slicing on each of the nineteen buggy programs (corresponding to the four Statechart models). At the model level, we specify the slicing criterion as the last "wrong" state visited by a particular object (which gives the observable "error"). Since we actually perform slicing at code level, we specify the criterion as the corresponding state entry point (not necessarily the state entry action) in the code.

As mentioned earlier, each Statechart model has several buggy versions, and in each buggy version the slicing criterion is set based on the observable error. However, for dynamic slicing, apart from the slicing criterion, we also need inputs which exhibit the observable error in question. Hence corresponding to each buggy version, (at least) five test cases are chosen. *The experimental results (shown in Table 2) report all quantities corresponding to a buggy version as the average over all the test cases for that buggy version.* The goal for choosing different inputs for the slicing was to get rid of (or at least reduce) the influence of any specific program input on the overall results. Furthermore, the same bug may manifest itself as different observable errors for different inputs (leading to different slicing criteria).

The columns with heading "Slice Size" in Table 2 show the comparison of slice sizes. For all the buggy versions, the size of model-level slice is 12% to 25% of corresponding code-level slice. This is not surprising since a single model element may require a couple of lines of code to implement. The model-level slice is 27% to 47% compared with the total number of model elements, while the corresponding ratio for code-level slices is 17% to 30%. The larger ratio for model-level slices (as compared to code-level slices) is due to the same reason as above - when an element is included in

Model	Bug Type	Slice Size				Time (secs)	
		Code-level Slice	LOC	Model-level Slice	Total Elements	Map from Code-level	Build Hierarchy Slice
Shuttle System	1	316.2	1167	42.7	117	0.046	0.691
	1	334.8		43.5		0.039	0.609
	3	331.8		43.5		0.036	0.604
	2	282.0		37.5		0.027	0.591
	1	286.3		37.7		0.031	0.599
Railcar	2	412.8	1389	49.2	121	0.053	0.639
	3	405.3		47.0		0.044	0.613
	1	411.9		49.0		0.053	0.620
	1	414.0		48.4		0.045	0.607
Weather Control	1	353.7	1889	89.7	202	0.092	0.963
	1	324.8		78.2		0.090	0.985
	3	338.8		84.0		0.094	1.018
	1	376.4		94.6		0.097	1.016
	2	356.5		88.8		0.099	0.996
MOST	1	447.0	2440	74.3	277	0.118	1.009
	3	454.0		76.8		0.113	0.985
	1	491.1		92.0		0.194	1.058
	2	494.6		85.8		0.172	1.037
	1	466.0		81.3		0.133	1.028

Table 2. Summary of Experimental Results. Column 2 shows the type of bug, 1 - wrong control flow, 2 - wrong action, and 3 - missing element. The four columns under the heading "Slice Size" represent average size of code-level slices, total lines of code, average size of model-level slices, and total number of statechart elements. The two columns under the heading "Time" show the average dynamic analysis time, including time to map slice from code level and to build hierarchical slice.

the model-level slice, it is common that only a portion of corresponding code appears in the code-level slice.

The time to map code-level slice to model-level is shown in the first column under the heading "Time" in Table 2. We did not find significant differences across buggy versions of the same model. The average time to build hierarchical slice is shown in the second column under the heading "Time" in Table 2. It includes analyzing and constructing hierarchy tree for the Statechart and projecting the dynamic slice corresponding to the different nodes of the hierarchy tree. The time is almost same for each model, because reading the Statechart structure and constructing the tree needs a large amount of time.

Note that not all bugs can be found in dynamic slices. In our experiment, three of the nineteen buggy program versions (corresponding to the four Statecharts considered) had slices that do not contain the bug. For example, none of the dynamic slices contained the bug for the second buggy version of Shuttle System. Here, the condition of a choice transition was wrong and the corresponding transition never got fired. Thus, the error here occurred due to some portion of the model *not* being executed. Such errors cannot be located via dynamic slicing, and we need to employ techniques such as "relevant slicing" [22, 14].

8 Discussion

More and more software is not being produced in a hand-written manner. Indeed, in certain safety-critical domains (*e.g.*, avionics), developers are strongly encouraged to generate code from behavioral models. Consequently, we need new debugging method-

ologies. In this article, we have suggested the use of software debugging methods (such as dynamic slicing) on the code generated from behavioral models. The bug-report is then played back at the model level by exploiting the associations between program fragments and model elements, thereby achieving model debugging.

Acknowledgments This work was partially supported by a NUS research grant R252-000-321-112, and a Public Sector Funding research grant from A*STAR, Singapore.

References

1. Rhapsody tool. I-logix, inc. website: `http://www.ilogix.com`.
2. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
3. H. Agrawal and J. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1990.
4. B. Korel and J. W. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.
5. M.P.E. Heimdahl and M.W. Whalen. Reduction and slicing of hierarchical state machines. In *Intl. Symp. on Foundations of Software Engineering (FSE)*, 1997.
6. Y.A. Feldman and H. Schneider. Simulating reactive systems by deduction. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(2), 1993.
7. K.D. Nguyen, Z. Sun, P.S. Thiagarajan, and W-F. Wong. Model-driven SoC design via executable UML to systemc. In *IEEE Real-time Systems Symp. (RTSS)*, 2004.
8. A. Wasowski. On efficient program synthesis from statecharts. In *Intl. Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES)*, 2003.
9. H. J. Kohler, U. Nickel, J. Niere, and A. Zundorf. Integrating UML diagrams for production control systems. In *Intl. Conf. on Software engineering (ICSE)*, 2000.
10. W. Harrison, C. Barton, and M. Raghavachari. Mapping UML designs to Java. In *Intl. Conf. on Object-oriented Prog. Sys. and Languages (OOPSLA)*, 2000.
11. Stateflow tool. The MathWorks, inc. website: `http://www.mathworks.com`.
12. D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7), 1997.
13. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
14. T. Wang and A. Roychoudhury. Using compressed bytecode traces for slicing Java programs. In *Intl. Conf. on Software Engineering (ICSE)*, 2004.
15. L. Guo, A. Roychoudhury, and T. Wang. Accurately choosing execution runs for software fault localization. In *Compiler Construction (CC)*, 2006.
16. T. Wang and A. Roychoudhury. Hierarchical dynamic slicing. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2007.
17. Shuttle_Control_System. New rail-technology Paderborn. `http://wwwcs.uni-paderborn.de/cs/ag-schaefer/CaseStudies/ShuttleSystem`.
18. CTAS. Center TRACON automation system. `http://www.ctas.arc.nasa.gov`.
19. MOST Cooperation. `http://www.mostcooperation.com`.
20. JSlice: dynamic slicing tool for Java. T. Wang and A. Roychoudhury, National University of Singapore. website: `http://jslice.sourceforge.net`.
21. T. Wang and A. Roychoudhury. Dynamic slicing on Java bytecode traces. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2), 2008.
22. T. Gyimóthy, Á. Beszédes, and I. Forgács. An efficient relevant slicing method for debugging. In *7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 303–321, 1999.