# DEBUGGING STATECHARTS MODELS VIA MODEL-CODE TRACEBILITY

GUO LIANG

(B.Comp, National University of Singapore)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2008

# ACKNOWLEDGEMENTS

I would like to thank a lot of people for their guidance and help. I sincerely acknowledge all those whom I mention, and apology to anybody whom I might have forgotten.

Firstly, I express my sincere thanks to my supervisor, Dr. Abhik Roychoudhury, for his valuable advice and guidance. I really appreciate his support in both academics and life during my graduate study, and providing me the opportunity to work with him in the area of software debugging.

I have special thanks to my parents and family for their love, encouragement and understanding. They have been very supportive throughout my studies.

I am grateful to my friends for their support and friendship. I thank my friends Wang Tao, Ju Lei, Wang Fanru, Liu Shanshan, Shen Ren, Huang Wenfan, Liu Yang, and Li Jia to name a few.

I also thank the administrative staffs in School of Computing, National University of Singapore for their supports during my study. The work presented in this thesis was partially supported by a research grant from the Agency of Science, Technology and Research (A*STAR) under Public Sector Funding.

# TABLE OF CONTENTS

# SUMMARY

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 thesis, 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.

First, our study is presented concretely in terms of dynamic slicing of sequential 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.

The existing dynamic slicing tool JSlice only supports basic features of Java languages. However, most real programs utilize advanced Java language features including exception, reflection, and multi-threading. We further extend JSlice tool to support full Java language by integrating the above three features into it. Meanwhile, with the support of multi-threading in the concrete code-level analysis tool JSlice, we enhance our code generation methodology of Statechart models to produce multi-threaded Java code. Compared to sequential code generated, multi-threaded

program lifts the restriction imposed on Statechart behavior where concurrent states are serialized in sequential programs. With the support of advanced language features, both code generation tool and JSlice greatly extend their usability.

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 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 [43] 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 thesis. 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 [21] [1] as the modeling language and dynamic slicing [4, 28] as the debugging method.[2] 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 values of certain variables that can be observed easily - e.g. through program output), 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 thesis.

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

[1] In this thesis we use Statecharts and UML State Diagram interchangeably. In terms of Statecharts definition and behavioral model, we follow the UML Specification 2.0 [35].

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

and lines of code are embedded). Subject to an observable error [3], the generated Java code is subjected to dynamic slicing. The resultant slice is mapped back to the model level, while preserving the original 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.[4] However, to popularize such tools for debugging model-driven software may require a shift in mind-set of programmers who are accustomed to debugging code written in standard programming languages. More importantly, there exists a vast wealth of mature algorithms/tools for software debugging, which we would like to re-use while developing debugging methods for model-driven software.

We choose a dynamic slicing tool JSlice recently developed [45, 11] as the code-level debugging tool. It executes and performs dynamic slicing analysis on Java programs. Given the criteria as statements and variables involved, it outputs a list of statements affecting the criteria. In the first part of the study, we present the effort involved in generating *sequential* Java code from Statechart models, performing code-level dynamic slicing on generated code, and mapping code-level result back to model-level. In fact, most code-level (semi-)automatic debugging tools do not support advanced language features like multi-threading, including the dynamic slicing tool JSlice we

[3]An observable error means the program behaves abnormally - producing incorrect output or performing unexpected action, which can be identified by programmer. In fact, a programmer only considers debugging a program if he/she observes an error.

[4]Static slicing of Statecharts has been studied in [24]. Direct simulation of statecharts (possibly for debugging) has been discussed in [12].

choose. By generating sequential program, we are still able to fully demonstrate our methodology, and study the feasibility of integrating model-level design tools and code-level analysis tools. This has been presented in [18].

However, many real models and real programs require multi-threading and other support. To extend the usability of code generation methodology and JSlice tool, we further enhance both tools to support many advanced language features. For JSlice, we add support of exception, reflection, and multi-threading, and thus extend to full Java language support. Exception and reflection produce "gaps" in the execution in the point of view of slicing, where the normal execution is suspended and additional actions are performed making the execution resumes from a (different) point. We also need to distinguish between threads to perform slicing and to consider the effect among them. For code generation tool, we also extend it to generate multi-threaded Java program, which conforms to the Statechart behavior standard more closely, by removing the constraint imposed during serialization.

In summary, this thesis 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 thesis constitutes a further step in this direction where we extend both the methodology and underlying code-level analysis tool to support more advanced features.

# CHAPTER 2

# THE DYNAMIC SLICING TOOL - JSLICE

Testing and debugging is a common activity in program development life cycle, and most of time it is difficult and time consuming. During testing we identify a program execution as incorrect by (a) some exception occurs at a statement, or (b) the output / intermediate result is incorrect. Usually the above statement where the error occurs is not the buggy statement. This is true even for some obvious errors. For example, in the case of a variable $v_1$ referencing to an inaccessible memory address and crashes the program, the buggy statement could be assigning a wrong value to another variable $v_2$ at the very beginning, which is involved in calculating the value of $v_1$.

Given a program $p$, developer tests it using a set of testcases $T = \{(i, o_e)\}$, where each pair of $(i, o_e)$ is the input $i$ and expected output $o_e$. The program $p$ contains error if for some input $(i, o_e)$, the observed result $o_r$ is not the same as $o_e$. In order to debug $p$, the developer needs to examine $p$'s states with input $i$ leading to erroneous observation. Traditionally, the debugging approaches could be:

- Inserting printing function at various locations in $p$ to display the program state including relevant variables, call stacks, and etc.
- Using conventional debugger (e.g. GDB [15], JDB [16]) to set breakpoints, execute program in steps, and examine states more easily.

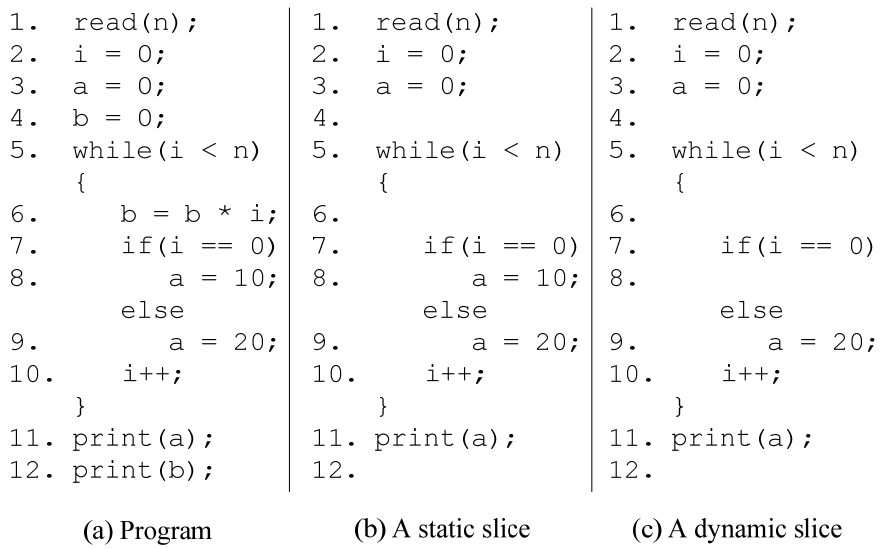



**Figure 2.1:** A slicing example. (a) is the program. (b) is the static slice with variable $a$ at line 11 as criterion. (c) is the dynamic slice with input n = 2 and variable $a$ at the first occurrence of line 11 as criterion.

These approaches help developer to hypothesize a subset of program statements which are likely to be the buggy statements, and then to confirm each statement by examining its state. However, these conventional approaches only provide mechanism to examine program states, and still require developer's manual intervention to locate the buggy statements. In other word, they cannot make any conclusion further from program states.

A number of automated debugging techniques are proposed to increase the degree of automation in debugging. Most of them analyze testing result and program states to provide a subset of program statements which are suspicious, or a limited number of statements expected to be buggy statements. Two major techniques out of these are slicing and test based fault localization.

- **Slicing** is the technique to reduce the set of program statements by excluding

statements that are not relevant to the error [1, 2, 3, 4, 5, 25, 28, 29, 32, 42, 45, 47, 50, 51]. Slicing algorithm requires slicing criteria as the starting point. A slicing criterion is a variable $v$ at some statement / location $l$, and is usually the observed error. Starting from criteria, slicing algorithm searches through control and data dependence in program dependence graph to include all statements traversed. Slice can be computed backward or forward. Backward slice contains statements that directly or indirectly affect the criteria; while forward slice contains statements that are (transitively) dependent on the criteria. Backward slice is usually interested in terms of program debugging. Slicing can also be performed on static program (static slicing) or on program execution trace (dynamic slicing). Figure 2.1 shows an example of static and dynamic slicing performed on the same piece of code, by examining static / dynamic dependencies respectively. Note that static slice is computed w.r.t. *all executions* of the program, and dynamic slice is computed w.r.t. a particular execution with given input. Dynamic slicing is usually more interested to programmers as it analyzes a particular execution with erroneous output and produces less statements compared to static slicing.

- **Test based fault localization** techniques take a different approach. Instead of examining the program or an execution, these techniques compare the failing runs (i.e. executions with erroneous behaviors) and successful runs (i.e. executions without erroneous behaviors) [6, 7, 17, 19, 25, 36, 38, 37, 46, 52, 53]. The successful runs can be obtained either by selecting some inputs from a test case

pool, or by alternating the branch or state of a failing run. The difference $diff$ in terms of different statements executed, different dependencies, or different program states are generated. The rational is the $diff$ must be related to the observed error, as applying $diff$ to failing runs will produce successful runs.

## 2.1 Dynamic Slicing

In this thesis, we focus on using dynamic slicing as the debugging technique. Generally dynamic slice includes the closure of dynamic control and data dependencies from the slicing criterion. Assuming $\beta$ represents an occurrence of the statement $stmt(\beta)$, dynamic control and data dependencies can be defined as follows.

**Definition 2.1. Dynamic Control Dependency** *The statement occurrence $\beta$ is dynamically control dependent on an earlier statement occurrence $\beta'$ iff*

1. *$stmt(\beta)$ is statically control dependent[1] on $stmt(\beta')$, and*

2. *$\nexists\, \beta''$ between $\beta$ and $\beta'$ where $stmt(\beta)$ is statically control dependent on $stmt(\beta'')$.*

**Definition 2.2. Dynamic Data Dependency** *The statement occurrence $\beta$ is dynamically data dependent on an earlier statement occurrence $\beta'$ iff*

1. *$\beta$ uses a variable $v$, and*

2. *$\beta'$ defines the same variable $v$, and*

3. *the variable $v$ is not defined by any statement occurrence between $\beta$ and $\beta'$.*

[1] *Static control dependence is defined in [13] using the notion of post-dominators in the control flow graph.*

Dynamic control and data dependencies can be captured by *Dynamic Dependence Graph* (DDG) [4]. Each node in DDG represents an occurrence of a statement, while each edge represents dynamic data / control dependency. Then the dynamic slice can be defined as follows.

**Definition 2.3. Dynamic Slice** *for slicing criteria consists of all statements whose occurrence nodes can be reached from the nodes representing the slicing criteria in the DDG.*

## 2.2 The JSlice Tool

JSlice [11, 45] is a framework to perform dynamic slicing on Java programs, with the infrastructure showing on Figure 2.2. JSlice framework consists of a front end (GUI) and a back end. The back end is the core component which collects execution trace and performs dynamic slicing.

Given a Java program to be debugged (usually a program with unexpected output), the programmer can use JSlice to find out the relevant statements w.r.t. the unexpected output. As the first step (Phase 1 in Figure 2.2), the programmer selects the dynamic slicing criteria via GUI, which can be one or multiple Java statements. Each statement can be further specified whether he/she is interested in the last occurrence or all occurrences during program execution. Then (Phase 2 in Figure 2.2) the programmer invokes JSlice back-end through GUI to perform slicing. It uses the Java Virtual Machine to execute the program and collect bytecode trace in compact form, and then performs slicing w.r.t. the criteria specified previously. The resulting bytecode level slice is then mapped to source code level (statement level) according

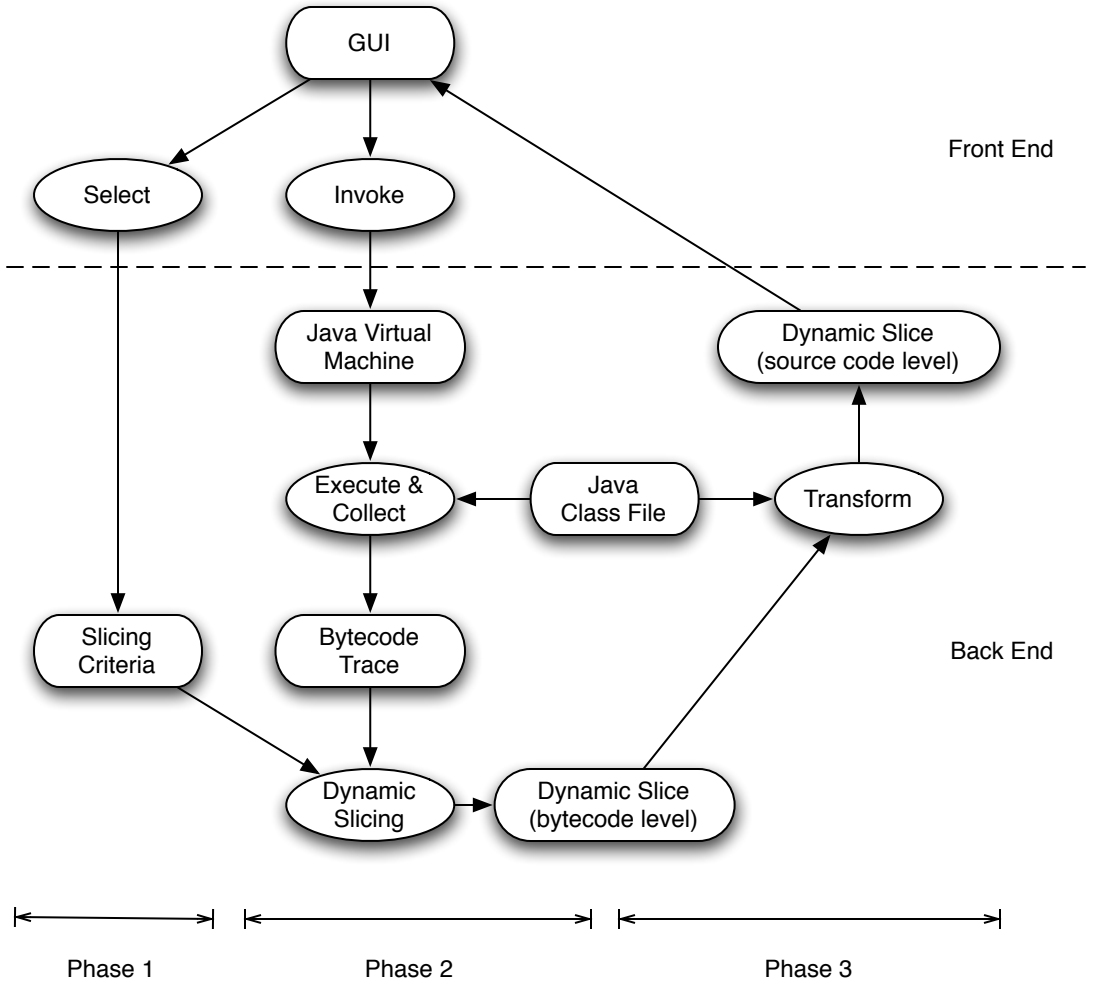



**Figure 2.2:** The infrastructure of JSlice. Phase 1: Select slicing criteria. Phase 2: Perform dynamic slicing. Phase 3: Display slicing result.

to the Java class file (Phase 3 in Figure 2.2) and highlighted in GUI.

JSlice (back end) is developed by modifying an existing Java Virtual Machine (JVM) - Kaffe [26], with the capability of collecting trace and slicing program.

- **Trace Collection.** The bytecode trace collection is the foundation of JSlice infrastructure (as in Figure 2.2). For medium to large size programs, the bytecode trace would be huge. Thus, JSlice compacts bytecode trace on-the-fly during program execution. First, bytecodes not corresponding to heap memory

access and control transfer (e.g. method invocation) are not stored in trace, as their operands are fixed and can be discovered from Java class file. For bytecodes to be traced, the sequence of addresses used by them is stored compactly. Since these addresses typically have highly repeated patterns, JSlice uses a variant of well-known lossless data compression algorithm SEQUITUR [33] (called RLESe) to store them in compressed form. Another important advantage of RLESe is the compressed addresses can be accessed without decompression.

- **Slicing Algorithm.** JSlice employs a goal-directed backward slicing algorithm, which analyzes the compact bytecode trace starting from the occurrences of bytecodes in the slicing criteria. During slicing, it maintains: (a) the dynamic slice $\varphi$, (b) a set of variables $\delta$ which has been used by bytecode in $\varphi$ but not been defined in bytecodes traversed, and (c) a set of bytecode occurrences $\gamma \subseteq \varphi$ where the bytecode occurrences they dynamically control dependent on are not traversed yet. Given a slicing criterion as $(l, v)$ ($l$ is a bytecode occurrence and $v$ is a variable), initially we have $\varphi = \gamma = \{l\}$ and $\delta = \{v\}$. For each bytecode occurrence $\beta$ traversed,

  - if there exists any bytecode occurrence in $\gamma$ which is dynamically control dependent on $\beta$, these bytecode occrrences are removed from $\gamma$. Then variables used by $\beta$ are inserted into $\delta$, and $\beta$ is inserted into $\varphi$ and $\gamma$. This essentially checks dynamic control dependencies.
  - if $\beta$ defines a variable $v_\beta$ and $v_\beta \in \delta$, we have $v_\beta$ removed from $\delta$, and variables used by $\beta$ are inserted into $\delta$. $\beta$ is inserted into $\varphi$ and $\gamma$. This

finds the locations of variable definitions for variables used at earlier stage of traversal (later stage of program execution) to resolve dynamic data dependencies.

During backward traversal, JSlice also simulates stack operations to capture data dependencies introduced by data access via stack.

## 2.3 JSlice Extension

JSlice was first developed by Wang et al. [45] at National University of Singapore. As described above, the first JSlice version extends a Java Virtual Machine (Kaffe) to provide dynamic slicing function. Given a slicing criteria, it produces dynamic slice for Java program with basic Java language features. Although many Java programs are supported by JSlice, it lacks support of more advanced language features including exception, reflection (with Java Native Interface), and multi-threading.

Most real programs contain these advanced features. Thus it is necessary to support programs with all Java features. We have extended JSlice to support all features in Java language [2] including exception, reflection and multi-threading. That is, the new JSlice is able to collect the trace for programs containing these features, and to perform slicing w.r.t the trace.

Exception occurs when there is computation error determined by JVM (internal exception) or executing program (external exception). When an exception is thrown, if there is exception handler in a method in the call stack, the execution jumps to the begin of exception handler, and resumes executing the exception handler and

[2] All Java language features of Java version 1.4 are supported.

normal program code follows. In order to reach the exception handler possibly in the middle of call stack, several methods on top of it must be popped, which leaves an execution gap compared to normal execution - only one method is popped each time when its `return` statement is executed. To support slicing w.r.t exception, we need to explicitly record the type of exception, methods popped and their operand stacks.

Reflection provides a mechanism to access a variable or execute a method, where the exact variable or method is only known at runtime. JVM supports reflection using Java Native Interface, which traps into JVM's internal structure to locate the required variable or method. In the case of method invocation with reflection, after locating the method using native (C) code, the bytecodes of the Java method will be executed. Thus we have Java code and native code executing alternatively in reflection. However, since we are tracing Java program execution at JVM level, we are not able to trace the details of native code. For variable access, we need to save the variable address; and for method invocation, we need to record the Java method executed by native code (i.e. to be executed through reflection), and link their parameters and return values (since these are just passed between the calling Java method and the callee Java method through native code).

In order to support multi-threading, we need to maintain several call stacks and operand stacks, one for each thread. We also need to record the relative access sequence to shared variables among threads. During slicing, we should perform backward traversal along each thread, make sure the order of shared variable accesses among threads are preserved, and also to identify dependencies between threads.

The detailed JSlice extension is further discussed in Chapter 6.

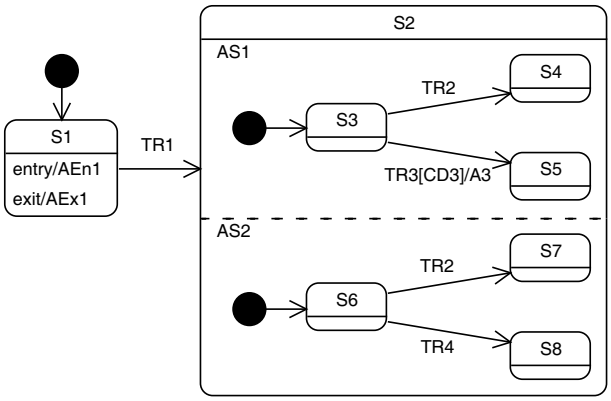
# CHAPTER 3

# BACKGROUND ON STATECHARTS

Statecharts were originally developed by David Harel for reactive systems [21] and have subsequently been integrated into UML specification as one of the major behavioral diagram types. Statecharts extend traditional finite state machines with three main features — hierarchy (OR-states), orthogonality (AND-states) and broadcast communication. Hierarchy is used to present a large state machine at different levels of abstraction. Orthogonality allows the different system components as separate state machines (running concurrently), rather than constructing their concurrent composition. Finally, broadcast communication is used for modeling event interactions among concurrent components.

Figure 3.1(a) shows a Statechart example. Initially, the system enters state $S1$, and the entry action $AEn1$ (of $S1$) is executed. After event $TR1$ is received, the system exits state $S1$ and executes the exit action $AEx1$ (of $S1$). Then it enters state $S2$ by following the transition on $TR1$. Since $S2$ is an orthogonal state with two AND-states $AS1$ and $AS2$, both states $S3$ and $S6$ are entered. This means that there are two concurrently executing components $AS1$, $AS2$ — one in state $S3$ and the other in state $S6$. At this point,

- if event $TR2$ is received, the system leaves both states $S3$ and $S6$, and enters states $S4$ and $S7$.




(a)

$$
\begin{aligned}
M &= \{C+\} \\
C &= \{Attr*,\ Meth*,\ SC?\} \\
SC &= \{E*,\ S+,\ T+\} \\
S &= \{AS*,\ A_{entry}?,\ A_{exit}?\} \\
AS &= \{S+,\ T+\} \\
T &= \{TR,\ CD?,\ A?\}
\end{aligned}
$$

(b)

**Figure 3.1:** (a) An example of Statechart; and (b) Statechart model structure. Suppose we have model $M$, class $C$, attribute $Attr$, method $Meth$, Statechart $SC$, event $E$, (OR-)state $S$, AND-state $AS$, transition $T$, trigger $TR$, condition $CD$, and action $A$; specifically, $A_{entry}$ and $A_{exit}$ are entry and exit actions of state $S$. * denotes zero or more such elements can be contained; + denotes one or more such elements can be contained; and ? denotes the element is optional.

- if event $TR3$ is received and the condition $CD3$ evaluates to true, in $AS1$ state $S3$ is exited and the action $A3$ is executed. Then state $S5$ is entered. State $S6$ in $AS2$ remains unchanged. Similar semantics apply when event $TR4$ is received.

In Figure 3.1(b) we outline the constituent elements of Statecharts. A class $C$ in the model $M$ may contain a Statechart $SC$, and each Statechart $SC$ contains some (OR-)states $S$, some transitions $T$, and all possible events $E$. A simple state has no AND-state, while a composite state may have one or more AND-states. *A composite state with two or more AND-states is also an orthogonal state, where the AND-states (AND-components) are running concurrently.* Both simple and composite states may optionally have entry action $A_{entry}$ and exit action $A_{exit}$. An AND-state $AS$ also contains a set of OR-states $S$ and transitions $T$. A normal transition $T$ connecting two

(OR-)states has a trigger $TR$ specifying which event fires the transition. Optionally, it may contain a condition $CD$ to guard the firing of the transition and an action $A$ to execute whenever the transition is fired. The model may also contain special transitions - join, fork, and choice transitions (see [21] for details).

# CHAPTER 4

# 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 [34, 49]. Certain other works (*e.g.*, [27] and, to a lesser extent, [23]) 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[43] and MathWorks[44] 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,

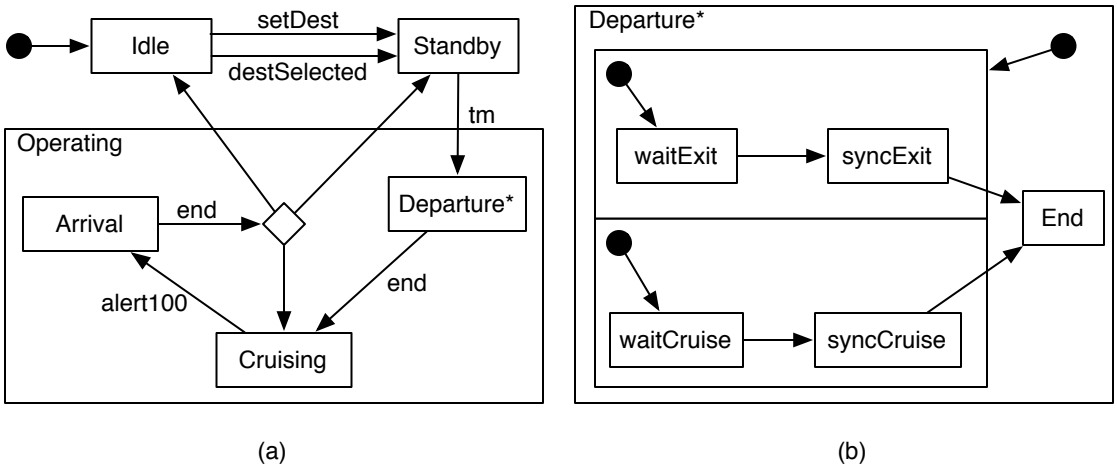



**Figure 4.1:** Statechart fragment corresponding to `car` object. (a) the top-level Statechart, and (b) the details of composite state `Departure`. State `Arrival` is also a composite state, the details of which is not shown.

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 [22] 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 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

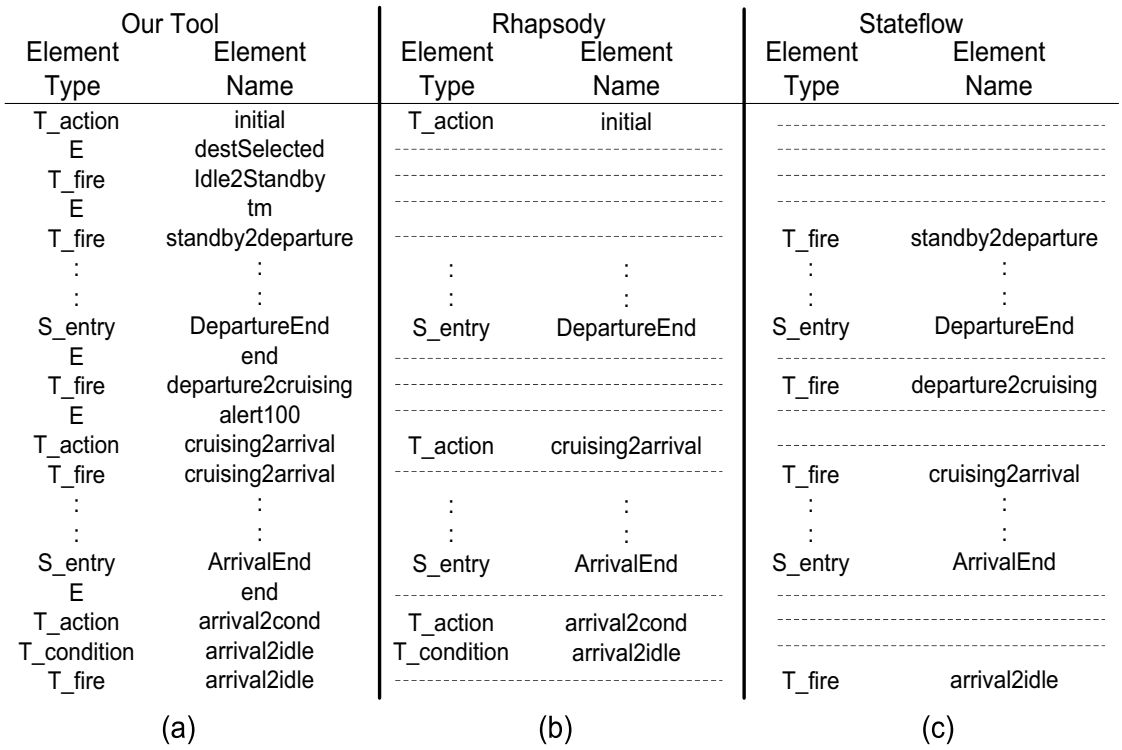



**Figure 4.2:** Model-level slices based on the code generated from (a) our tool, (b) Rhapsody, and (c) Stateflow. A dashed line shows a missing model element in the slice resulting from Rhapsody or Stateflow. "E", "T", and "S" appearing in "Element Type" denote "Event", "Transition", and "State". Model elements for `CarHandler` and details inside the states `Departure` and `Arrival` of `Car` are omitted for the ease of understanding.

between the terminal and cars. More details about the example along with the class diagrams and Statecharts for each class appears in [22].

In particular, we consider the Statechart of a `car` object (shown in Figure 4.1). 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` [1] as the slicing criterion and perform slicing based on the car object. As shown in Figure 4.2, 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.

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

*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.

# CHAPTER 5

# MODEL-CODE TRACEBILITY

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.

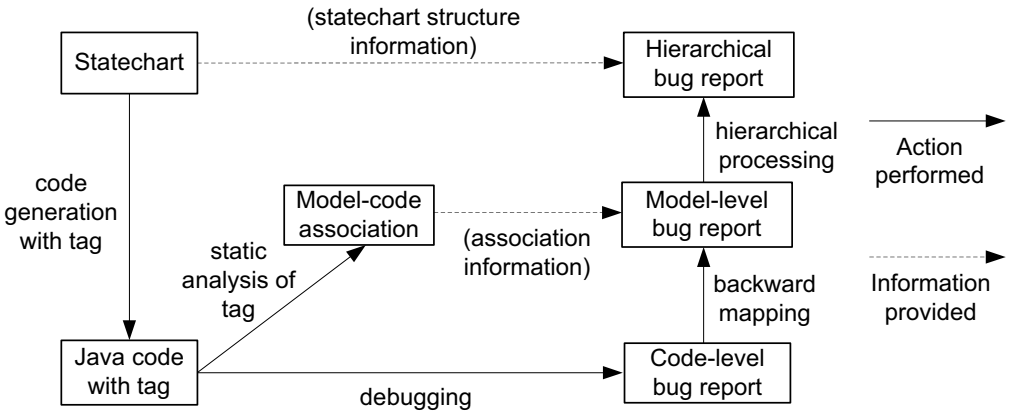



**Figure 5.1:** A brief representation of maintaining the traceability between model and code.

The whole methodology is summarized in Figure 5.1. When a Statechart model is available[1], 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 code-level analysis tools (such as slicing) to debug it. With the debugging result (code-level bug report), we map the bug report backward to model-level by replacing all statements corresponding to a model element in code-level bug report with the model element. 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. A preliminary report is presented in [18].

[1]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.

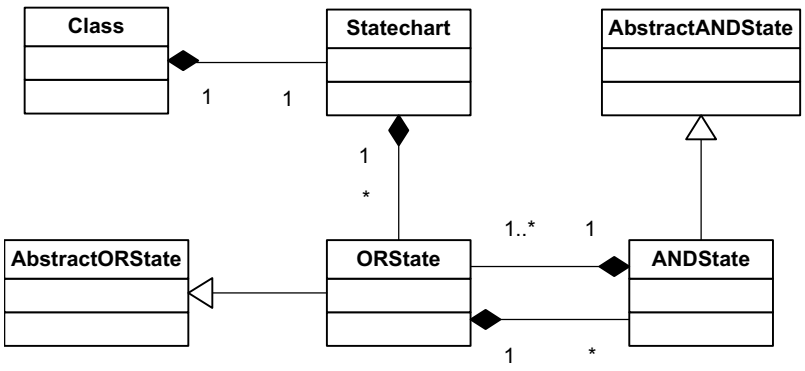



**Figure 5.2:** Class diagram of Java code generated from Statecharts.

## 5.1 Code Generation

First we discuss how we can maintain tags between model elements and generated code during the process of code generation. *In this chapter, we present our methodology by translating a Statechart to a single-threaded Java program. Thus, event communication at the Statechart level gets translated to method calls at the code level. Translating Statechart to multi-threaded program is discussed in Chapter 6. It is worthwhile to note that how do we translate Statecharts to code does not affect the method to build model-code association and to map code-level result to model-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 5.2, 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 to the AND-state) can be in. The design of OR-states within an AND-state follows the State design pattern [14].

```
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. <% } %>
```

**Figure 5.3:** A fragment of template used in code generation.

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 5.3 shows a fragment of a template used in generating an *ORState* class as in Figure 5.2, 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:

- instead of inserting tag to every statement related to a model element, we greatly reduce the space overhead for tags;
- 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;
- 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.

## 5.2 Debugging the Generated Code

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 numbers)`.

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.

Taking the example in Figure 4.1, where the "car" object visited 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 [28, 4, 45] and fault localization [19]. The sequence of statement instances (call it $\sigma_{code}$) gets mapped to a sequence of model element instances (call it $\sigma_{model}$) using model-code associations. These model element instances may come from different objects; we can

project $\sigma_{model}$ 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. Since all AND-states become active when their parent OR-state is active, there is no terminology of sequential transitions between AND-states.

Building hierarchical bug report at code-level is studied in [47]. 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

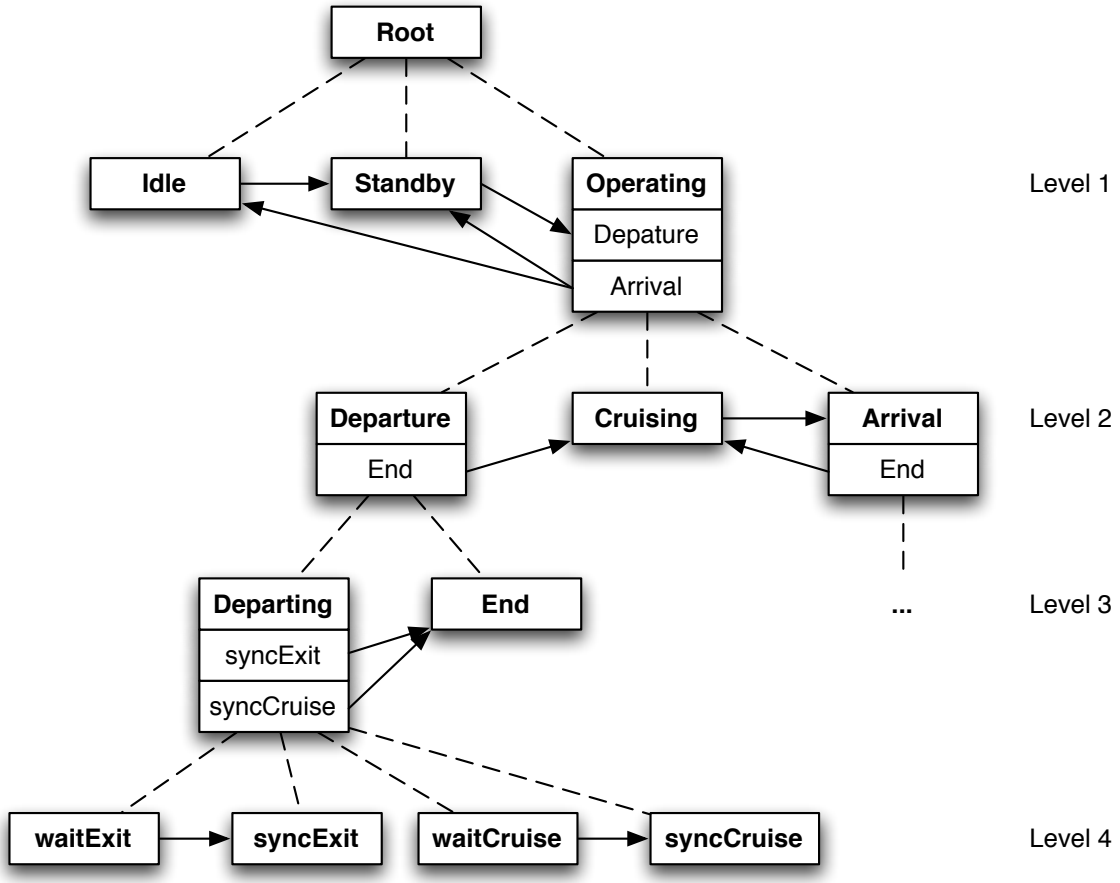



**Figure 5.4:** Hierarchical bug report for the example in Figure 4.1.

structure of the Statechart model and enables greater design comprehension. Figure 5.4 shows the hierarchical bug report (as a hierarchy tree) of Statechart example as in Figure 4.1. 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 5.4. 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 5.4, we have transition connecting `Standby` and `Departure` (in `Operating`).

The hierarchical bug report can be constructed as:

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.

# CHAPTER 6

# EXTENSION FOR ADVANCED PROGRAM FEATURES

In previous chapter we discussed model-code traceability for sequential programs. That is, by assuming the underlying code-level debugging tool can only process sequential program (which is true for many code-level debugging tools), our code generation tool generates sequential Java program for Statechart. Even if the Statechart contains AND-states that run concurrently in Statechart models, we manage to serialize the execution of concurrent AND-states by implementing event triggers as method calls. These method calls are properly arranged such that they follows the specification of Statechart execution model.

However, the generation of sequential code is only limited by the underlying code-level debugging tools. There is no obstacle preventing us from generating concurrent code from Statechart model. As we have extended the JSlice tool with the ability to analyze multi-threaded Java programs, we can generate concurrent code containing threads to maximize the code performance. It also means that generated code exposes the uncertainty of event triggers through threading, which complies with Statechart execution model exactly. In the following, we will discuss the support of concurrent code generation and concurrent (and other) extensions to dynamic slicing.

## 6.1 Concurrent Program Code Generation For Statechart

As implied by the characteristic of Statechart, we use threads to realize concurrent AND-states. When an (OR-)state $s$ containing several AND-states becomes active, all its AND-states are active. Thus we create a thread for each AND-state, and these threads terminate when the (OR-)state $s$ becomes inactive [1].

The event triggers are handled through a centralized *Event Manager*. As shown in Figure 6.1(a), the event manager is associated with a dispatching table, which contains the mapping between events and threads (AND-states). All events generated internally and externally are sent to the event manager, which is responsible to dispatch the event $e$ to all states containing transitions to be triggered by $e$. Since we use threads to implement AND-states, the event manager effectively dispatches $e$ to a number of threads found in the dispatching table. The event manager also has the mechanism for threads to register/de-register themselves for an event.

When a thread enters a state $s$, it registers itself to the event manager with the list of events $E$ that can trigger transitions at current state $s$. Then it will wait for event manager to dispatch an event $e \in E$ generated internally or externally. By receiving an event $e$, the thread could proceed to make transition to next state. Figure 6.1(b) shows a fragment of a Statechart. When state $s_0$ becomes active, both its AND-states $as_1$ and $as_2$ are active and two threads $t_1$ and $t_2$ are created respectively. Note that for $as_1$ and $as_2$, state $s_1$ and $s_4$ are active. Then $t_1$ registers itself with event $\{e_1, e_2\}$ to

[1] The OR-state and all of its AND-states are active simultaneously. It is impossible that one of the AND-states become inactive while other AND-states directly contained by the same OR-state are active.

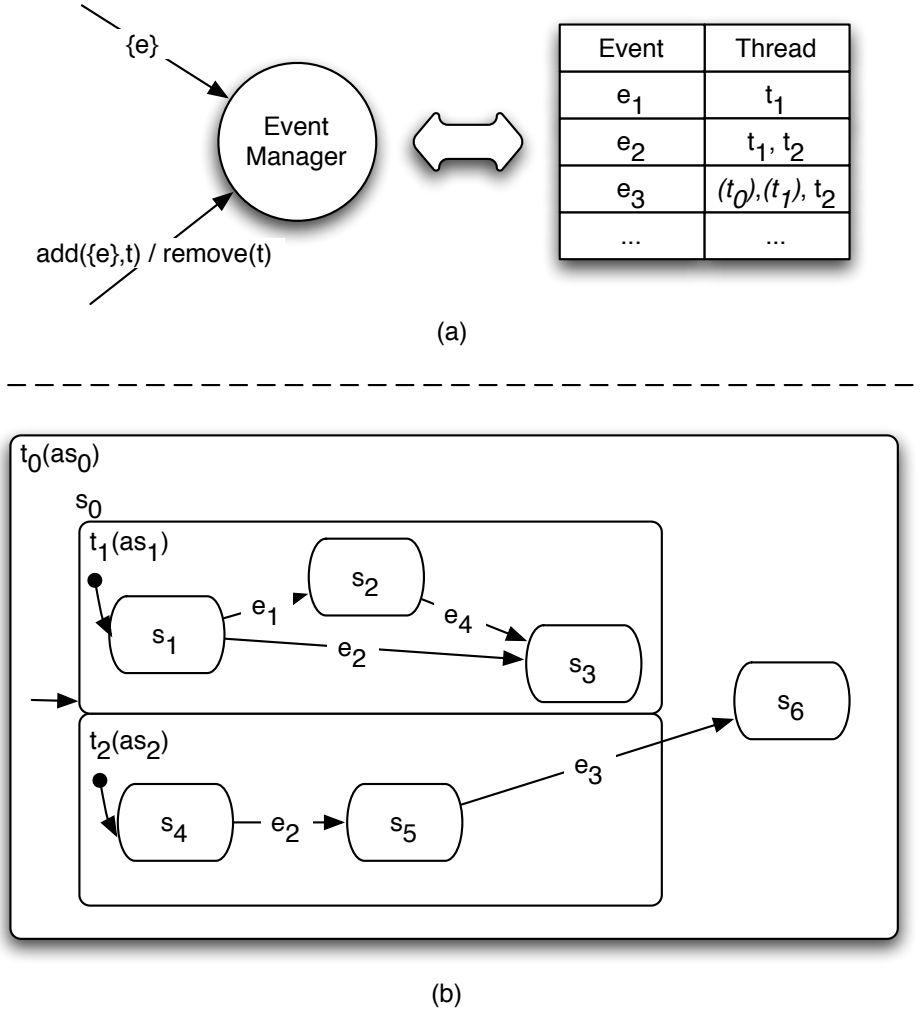



**Figure 6.1:** The event manager in generated concurrent code.

event manager, and $t_2$ registers itself with event $\{e_2\}$, resulting the dispatching table as in Figure 6.1(a) (first two entries in dispatching table). Suppose event $e_1$ is received by event manager, it will find $t_1$ to dispatch, where $t_1$ could make the transition to state $s_2$. After event manager dispatches $e_1$ to $t_1$, it will remove all entries of $t_1$ from the dispatching table, since $t_1$ will make an transition and invalidate all $t_1$'s entries in the table.

The dispatching of events by event manager is implemented through semaphore.

A semaphore has a counter initially setting to 0, and provides two operations - UP and DOWN. UP operation increases the counter by 1 atomically; and DOWN operation decreases the counter by 1 atomically. If the counter is 0 before DOWN operation, the calling thread is blocked until an UP operation is invoked by another thread. From thread perspective, a thread $t_1$ can wait (and block) for some event by invoking DOWN operation. Another thread $t_2$ can signal $t_1$ by invoking UP operation on the same semaphore. In the context of Statechart code generation, we have a semaphore for each thread. In the example shown in Figure 6.1, after thread $t_1$ registers to event manager, it invokes DOWN operation on its semaphore $sem_1$ and is blocked. After event manager receives $e_1$, it invokes UP operation on $sem_1$, to wake up thread $t_1$ to make the transition.

**Implementation choice of semaphore.** When we discuss thread signaling mechanism, we have several choices. One is to use semaphore, and another is to use wait/signal mechanism. The most relevant difference between two mechanisms is: with wait/signal, if a thread is signaled before it waits, the signal is lost and it will keep on waiting. In our case, a thread waits after it registers to event manager. However, these two actions are not executed in one atomic step. It is possible that event manager receives the event and signals the thread between these two actions, resulting the thread waits without being signaled. Thus, the semaphore makes sure the thread can always be signaled after it registers to event manager.

If a thread is waiting in a state with only one outgoing transition, the event manager is just required to signal the thread. However, since a state may have two or more outgoing transitions, the event manager must provide the actual event to

the signaled thread as well, for it to trigger the correct transition. We also need to make sure when an AND-state triggers transition going out of it, all other AND-states contained within the same OR-state are deactivated as well. As shown in Figure 6.1, suppose $t_2$ enters state $s_5$, it will register itself, $t_1$ and their super state $t_0$ with event $e_3$ in event manager ($t_0$ and $t_1$ in the third row of table is shown in italic indicating they are registered by other thread). Upon event $e_3$ occurs,

- $t_1$ and $t_2$ execute exit actions (if any) of current states and terminate.
- $t_0$ is woke up from state $s_0$ and enters $s_6$.

## 6.2 Slicing with Advanced Features

JSlice is capable of producing dynamic slice for Java programs, and it supports major features of Java programming languages. However, it lacks support of advanced features - exception, reflection, and multi-threading. We have extended JSlice with full Java programming language (version 1.4) support by implementing the above three features. Implementing these features is important as most real programs utilize one or more of them.

### 6.2.1 Exception

When a program violates any semantic constraint of the Java programming language, the Java virtual machine throws an exception to signal this error [53]. Meanwhile, a Java program may also explicitly throw an exception indicating error encountered to the program. This exception causes a non-local control transfer from the point where exception occurred to the exception handler (if any) specified by programmer. During

the trace of program execution, we must store this non-local control transfer in order to simulate it reversely from handler to the point exception occurred during backward traversal of dynamic slicing. The execution transfer from exception point to handler may require popping up method invocations from call stack if the handler does not reside in the same method of exception. The Java Virtual Machine pops methods from call stack one by one until it finds the appropriate handler, and continues to execute the handler [2].

To make sure we can traverse backward, for each exception occurred we maintain a list of methods popped and the type of exception (thrown by JVM or program). Suppose the exception occurred in method $m_0$ and handled in $m_h$, we keep the method sequence as $meth\_pop = (m_0, m_1, \ldots, m_{h-1}, m_h)$, where methods $m_0$ to $m_{h-1}$ are popped, and the program counter of $m_h$ revises from invocation to $m_{h-1}$ to the handler. For each $m_i$, we maintain:

- the class name of $m_i$, and
- the method name of $m_i$, and
- the signature of $m_i$, and
- the last executed bytecode of $m_i$, and
- the size of operand stack of $m_i$ before it is popped or revised.

Thus during backward traversal, we could construct the call stack with correct methods, program counters, and (sizes of) operand stacks, when we reach the beginning

[2]Java Virtual Machine will exit if an appropriate handler does not present.

of exception handler.

Exception introduces dynamic control and/or data dependencies between the bytecode throwing the exception and the exception handler catching it. [41]

- There is dynamic control dependency since the execution of handler is dependent on the occurrence of exception (i.e. the bytecode throwing exception).
- There could also be data dependency if the exception is explicitly thrown by program using "`throw`" statement. This is because it will push an exception object to be thrown into the operand stack which could be used by the handler. Thus we need to record the type of exception, and if it is thrown by program, we maintain the proper data dependency w.r.t the exception object by pushing it into the operand stack of method throwing exception during backward traversal.

### 6.2.2 Reflection

Reflection enables Java program to: (a) access class structure and object fields of a selected class/object at runtime, (b) create a runtime-specified object, and (c) invoke a runtime-selected method. Most of these capabilities are implemented by calling native methods (written in C) in JVM. In general, JSlice which works at JVM level cannot trace within native methods, and thus we cannot support slicing on native method. However, we can support reflections in JSlice as we can get the object fields to be accessed, or the Java method to be invoked by native method.

**Accessing class structure and object fields.** These reflection methods do not call native method but access the internal structure of JVM. Thus we can trace these accesses similar to bytecode tracing. For example, for object field access, we can find

the object and the field from the parameter of reflection call, and record them for data dependencies analysis involving the field.

**Create runtime-specified objects and invoke runtime-selected methods.** These reflection calls first invoke certain native methods, which further invoke corresponding Java object constructor / Java method. Furthermore, the parameters for the Java method together with the method name are passed to the native calls as parameters. Although we cannot trace native method, we can map between the parameters passed to native method and parameters passed to subsequent Java method. The return value (if any) are mapped as well. These mapping of parameters and return values is to trace data dependencies across reflection calls.

Note that in normal method invocation, we do not need to explicitly record the callee method, as this information has been compiled into Java class file. However for reflection call, the information is not available statically, so we need to record the indirect Java callee (class name, method name, and method signature) through native method, and attach this information to method invocation bytecode in caller.

It is common that the reflection method is invoked several times, and in each invocation the intermediate native method calls several Java methods in sequence. During tracing, we use a stack to trace the bytecode instances in Java method calling native method, and for each Java method called by the native method, attach its information to the bytecode instance on top of the stack. For example, as shown in Figure 6.2(a), we have a reflection call to invoke callee method `A.f()`, which further uses reflection call to invoke another callee `B.f()`. Note that `invokeMethod()` and `invoke()` in both occurrences refer to the same Java/native method. Figure 6.2(b)

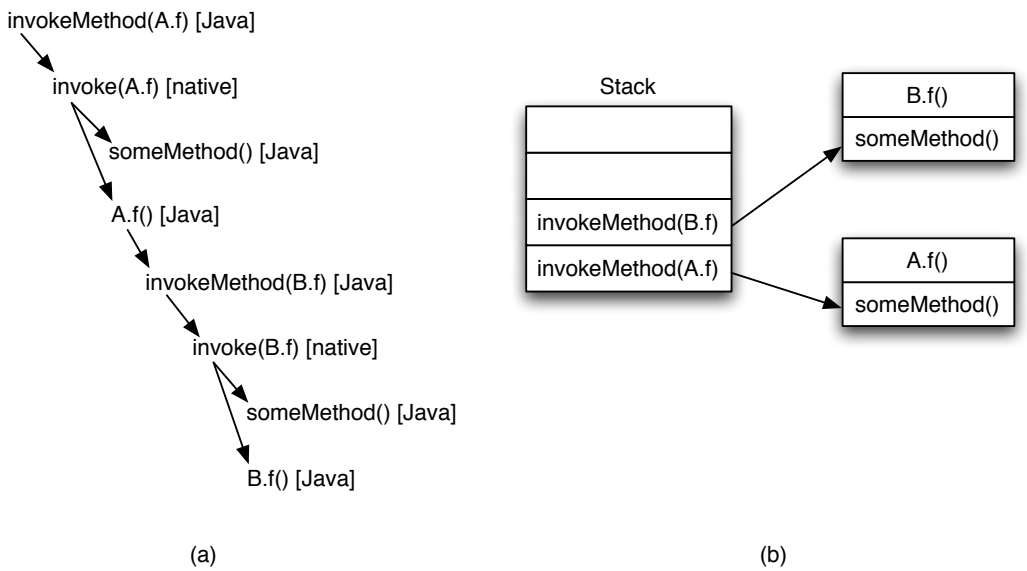



**Figure 6.2:** The example of multi-level and multi-callee in reflection invocation.

shows how we record reflection calls using stack. After we enters `A.f()`, we have `invokeMethod(A.f)` in the stack pointing to the list of callees (`someMethod()` and `A.f()`). Right after we enters `B.f()`, we have the stack shown as 6.2(b). After the reflection calls finished, the two invocation lists are attached to the bytecode $b_i$ calling `invoke()` in `invokeMethod()`. During slicing both times we reach the bytecode $b_i$, we push both `someMethod()` and `X.f()` to the call stack. So that we can simulate backward traversal covering all Java methods executed.

### 6.2.3 Multi-threading

We also extend JSlice to support multi-threaded Java programs. The trace collection for multi-threaded Java program is similar to single-threaded program. For single threaded tracing, each bytecode executed has its control and data flow trace stored compactly. In multi-threading tracing, we still store a bytecode's control / data trace for all threads in that bytecode, in order to reduce the overhead introduced

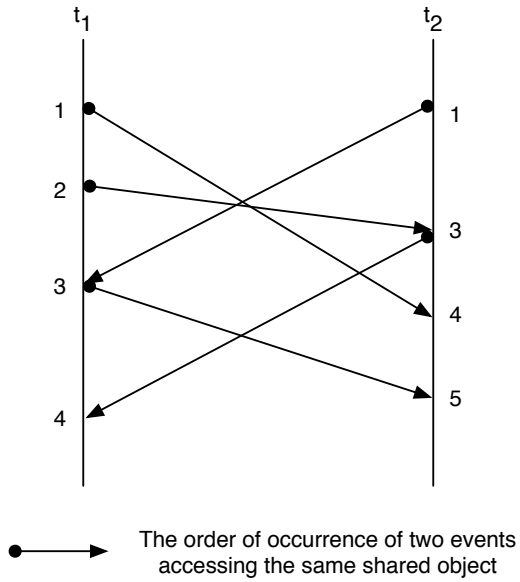



**Figure 6.3:** An example of events time stamps in multi-threaded dynamic slicing.

for maintaining separate trace for each thread. The difference is we maintain one call stack and one operand stack for each thread separately. However, threads often communicate with each other through inter-thread events, such as shared variable access, wait/notify, mutex and semaphore. The order of these events is required for dynamic slicing to detect dynamic dependencies between threads.

We use a method similar to [31] (Levrouw et al.) to trace the inter-thread events, which is based on the Lamport Clocks [30]. In Levrouw's approach, each thread $t$ has a scalar time stamp $c_t$ and each object $o$ has a scalar time stamp $c_o$. When a thread $t$ accesses a shared object $o$, this event is recorded with time stamp $c_e = max(c_t, c_o)+1$, where $max$ returns the maximum value of the two inputs. The time stamp $c_t$ and $c_o$ for $t$ and $o$ respectively are also set to $c_e$. This imposes a partial order on any two inter-thread events accessing the same shared object. For two such events $e_i$ and $e_j$, if $e_i$ occurs before $e_j$ ($e_i < e_j$), we have $e_i < e_j \Rightarrow c_{e_i} < c_{e_j}$. Figure 6.3 shows an example

of recorded time stamps for events occurred in two threads $t_1$ and $t_2$. The numbers are recorded time stamps, and the arrow indicates the order of two events accessing the same shared object. For two events connected by arrow, we must maintain their relative order during backward traversal. These orders can be captured by comparing their time stamps.

During backward traversal, we can retain the order of inter-thread events using time stamps recorded. That is, for any two inter-thread events $e_i$ and $e_j$ with $c_i < c_j$, we enforce $e_j$ to occur before $e_i$ as in backward traversal. Note that this may introduce additional event order constraints. It is possible that even if $c_i < c_j$, the execution order of corresponding events $e_i$ and $e_j$ are not constrained (e.g. these two events do not access the same shared object). However, these additional orders will not cause any deadlock in the backward traversal. As in Figure 6.3, although the events $e^3_{t_1}$ and $e^4_{t_2}$ are not ordered, we will still force $e^4_{t_2}$ occurring before $e^3_{t_1}$ in backward traversal.

It can be further optimized to reduce the trace size. Levrouw et al. show that it is not necessary to trace all time stamps to record the partial order. In particular, for an event $e$ of thread $t$ accessing object $o$, we only need to trace the increment of $c_t$ before and after the event $e$, if $c_t < c_o$. In other cases, we do not need to record the time stamp as $c_t$ and $c_o$ increments by 1 which is default. During backward traversal, if an event do not have traced time stamp, we can obtain its time stamp by decrementing $c_t$ by 1. Otherwise, we decrement $c_t$ by the recorded value.

The dynamic slicing algorithm for multi-threaded programs is similar to that for single-threaded programs as well. However there are several differences. The algorithm maintains operand stack and call stack for each traced thread. Although

the bytecode trace is recorded during execution w.r.t. multiple threads, the slice is computed in a single thread (the last thread finishing execution). At any specific time, only one traced thread with its operand stack and call stack is active. At the beginning of backward traversal, we first activate a thread (the main thread) and traverse it backward. During the traversal, we check the recorded time stamp for every bytecode accessing object (potential inter-thread event). When we cannot continue traversing the active thread due to time stamp constraints, we switch to another thread where traversal is not blocked by time stamp constraints. That is, we stop traversing a thread if there are inter-thread events (bytecodes) from other threads with bigger time stamps. Meanwhile, besides dynamic control and data dependencies, we also consider inter-thread dependencies, such as dependencies due to wait/notify calls.

**Handling of `System.exit()`.** Java Virtual Machine provides a system-level method `System.exit()` which terminates the program execution and exits JVM immediately. When this method is invoked by a program, we need to terminate its execution, and perform slicing before JVM exits. With concurrent programs, there is one more step - we should also terminate other threads immediately beside the thread calling `exit()`, since this is the expected behavior without slicing. When `exit()` is called by a thread $t$:

1. thread $t$ informs all other threads that they should terminate;
2. all threads stop executions and clean up;
3. the last stopping thread performs slicing.

# CHAPTER 7

# EXPERIMENTS

## 7.1 Experimental Setup

In order to experimentally evaluate our methodology, we adopt and construct four Statechart models for the evaluation of sequential code generation and bug report backward association. These models used are shown in Table 7.1. 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 Chapter 4, the other three models are based on real-life systems. The automated shuttle system [40] 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) [9] 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) [8] 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

| Statechart | Description | # model elements |
|---|---|---|
| RailCar | A rail car system from [22] | 121 |
| ShuttleSystem | Shuttles transporting passengers between stations [40] | 117 |
| WeatherControl | Updating weather status to clients [9] | 202 |
| MOST | Networking standard of multimedia system in cars [8] | 277 |

**Table 7.1:** Statechart models used in our experiment

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.

- *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 construct 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* [4, 28] 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 exploit the dynamic Java slicing tool JSlice [11, 48] from our previous work [45] to produce code-level slices. As discussed in earlier chapter, JSlice is an open-source tool which performs backward dynamic slicing of sequential Java programs. Since backward 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.

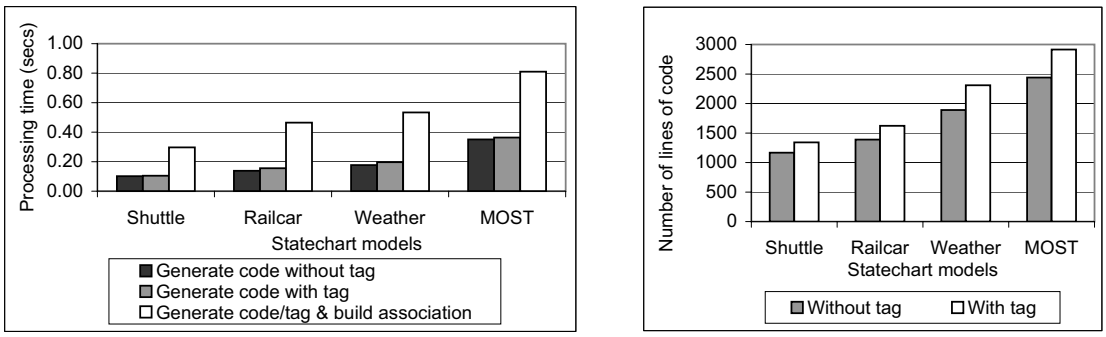
In addition, we also choose 2 smaller Statechart examples to evaluate concurrent dynamic slicing. We do not report experimental results for concurrent code generation and association as we will work on the full implementation of concurrent code generation in next stage. One Statechart example is Airline Tickets Issuing and another one is Bank Account Simulation, consisting of 56 and 41 model elements respectively. These two are adopted from the concurrent benchmark suite from IBM Research [39]. The Airline Tickets Issuing example simulates several agents selling a fixed number of air tickets for a flight. Every agent checks if there is available ticket and sells one repeatedly. Since all agents sell concurrently, proper locking is expected to make sure the number of tickets sold does not exceed total number of tickets available. The Bank Account Simulation example has similar behavior where several people access their accounts concurrently with deposit, withdraw and transfer operations. For each model, we manually inject two bugs to evaluate the effectiveness of dynamic slicing.

## 7.2 Experimental Results

For experiment on sequential code generation, we employ our tool on nineteen buggy program versions (for the four Statecharts in Table 7.1) to evaluate the efficiency and effectiveness of the methodology. We first consider the efficiency of generating sequential code with tags.

### 7.2.1 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




(a) (b)

**Figure 7.1:** Experimental results for sequential code generation. (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.

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 7.1(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 that modifications to Statecharts which only modify model elements do not require re-generation 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 7.1(b). The increase in code size due to tags is low — 15% - 22%.

### 7.2.2 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 7.2) 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). In the following, we discuss only the average slice size and times for each buggy version. This is particularly so, because we did not find significant differences in slice size and times across different inputs of a buggy program version.

| 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 7.2:** Summary of experimental results for sequential dynamic slicing. 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 columns with heading "Slice Size" in Table 7.2 show the comparison of slice sizes. The slice size for code-level bug report is the number of statements contained; while the slice size of model-level bug report is the number of model elements contained. 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 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 7.2. 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 7.2. 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 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. *Although the condition can be included in dynamic slice, this is misleading as the reason why the model behaves incorrectly is due to the transition guarded by that condition is not fired.* Thus, the error here occurred due to some portion of the model *not* being executed. Such errors cannot be found in dynamic slicing, and we need to employ techniques such as "relevant slicing" [20, 45].

### 7.2.3 Concurrent Dynamic Slicing

We also perform dynamic slicing on the two concurrent examples. We first manually construct corresponding Java code following the methodology described in Chapter

| Model | Bug Type | Slice Size | | | | Slicing Time (secs) | |
|---|---|---|---|---|---|---|---|
| | | Code-level Slice | LOC | Model-level Slice | Total Elements | Sequential | Concurrent |
| Airline Ticket | 1 | 210.5 | 683 | 22.0 | 56 | 0.912 | 1.231 |
| | 2 | 237.0 | | 25.0 | | 0.985 | 1.326 |
| Bank Account | 1 | 167.0 | 526 | 18.0 | 41 | 0.746 | 1.000 |
| | 2 | 171.5 | | 19.0 | | 0.771 | 1.018 |

**Table 7.3:** Summary of Experimental Results for Concurrent Programs.

6. We expect the time to generate concurrent code and build model-code association will be a little larger compared to sequential code generation, since there will be extra code generated for event manager and threads handling. The manually written Java programs for the two examples have 683 and 526 LOC.

Similarly, we specify slicing criterion as the last "wrong" state visited by an object, with corresponding code level statement as criterion input to JSlice. We employ the same methodology to choose criteria and program input as in the experiment for sequential dynamic slicing. For each buggy version, we apply two test cases leading to failing executions (execution with unexpected result).

The experiment results are shown in Table 7.3. It also reports quantities as average over all test cases for each buggy version, since there is no significant difference across test cases. The columns with heading "Slice Size" show the comparison of slice sizes on code-level and model-level. Compared with sequential programs, they have similar ratio w.r.t (a) the size of code-level slice over LOC, and (b) the size of model-level slice over number of model elements. No matter we generate sequential code or concurrent code (from the same model), we generate a piece of code for every model element. Meanwhile, the generated programs have same operational behavior, since

both follows the Statechart behavior model. Thus, the slice size is mainly dependent on (a) program, (b) program input, and (c) slicing criterion.

The time to produce dynamic slicing is shown under "Slicing Time". Quantities under "Sequential" are obtained by using JSlice without concurrent extension to slice sequential code generated from models; while "Concurrent" shows the time required for JSlice with concurrent extension to slice concurrent code. For concurrent programs, since we need to record the event time stamps in tracing and check them in slicing, the time overhead has been increased around 34% compared to sequential code. For large concurrent program, we expect a slightly higher overhead increment.

# CHAPTER 8

# DISCUSSION

More and more software is not being produced in a hand-written manner. Indeed, in certain safety-critical domains such as avionics, the developers are strongly encouraged to generate code from behavioral models. Consequently, we need new software debugging and comprehension methodologies. In this thesis, we have suggested the use of well-established 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.

Currently, we have developed a prototype for model-code associations in the context of Statecharts and Java. Dynamic slicing of the Java code results in a slice of the Statechart model being highlighted to the designer. In terms of future work, we can think of many avenues. First of all, we can complete the full implementation of generating multi-threaded programs from Statecharts, and perform comprehensive evaluations of it together with multi-threaded dynamic slicing. Since Statechart models support concurrent execution of processes, generating sequential code only captures a subset of the behaviors allowed by the Statechart model. By analyzing sequential code, if we find any bugs, they amount to bugs in the Statechart model.

However, we may not be able to find certain bugs in the Statechart model by analyzing sequential code, simply because those buggy behaviors are not even captured in the sequential code. As future work, with the full implementation of concurrent code generation, we can generate multi-threaded code from Statecharts and slice the multi-threaded code.

Secondly, one can try out model debugging using debugging methods other than dynamic slicing (such as relevant slicing or fault localization).

Additionally, we can examine whether our approach for debugging Statecharts can be extended to debug full-fledged UML models. Usage of our model-code associations for debugging of code generated from full-fledged UML descriptions and relating back the bug report to the UML level — remains a possible next step.

Finally, a similar approach can be adopted to build association between informal requirements and formal models. Given a requirement to a system, if it is informally stated in English, the problem of relating models to requirements can be harder. Similar to the spirit of this thesis, one could try to see whether the results of model-based testing (where the test-cases are obtained by exploring formal executable models) can be reflected back to the English language requirements. Even though this sounds like an impossible task, we note that in many application domains (such as avionics) the English language requirements are well-structured. They are given as "rules" on event ordering of the form "if x happens then y,z,w eventually happen" (see [9] for an example of such a requirements document.) Clearly such rules in English can be seen as temporal properties or even as specifications in executable visual formalisms like Live Sequence Charts [10]. This makes the task of backward association between

design models (possibly given as Statecharts) and the informal requirements (which can be visualized as Live Sequence Charts) more achievable.

# REFERENCES


[1] Agrawal, H., *Towards Automatic Debugging of Computer Programs.* PhD thesis, Purdue University, 1991.

[2] Agrawal, H., DeMillo, R. A., and Spafford, E. H., "Dynamic slicing in the presence of unconstrained pointers," in *Proceedings of the ACM Symposium on Testing*, pp. 60–73, 1991.

[3] Agrawal, H., DeMillo, R. A., and Spafford, E. H., "Debugging with dynamic slicing and backtracking," *Software - Practice and Experience (SPE)*, vol. 23, pp. 589–616, 1993.

[4] Agrawal, H. and Horgan, J., "Dynamic program slicing," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1990.

[5] Agrawal, H., Horgan, J., Krauser, E., and London, S., "Incremental regression testing," in *International Conference on Software Maintenance (ICSM)*, pp. 348–357, 1993.

[6] Choi, J.-D. and Zeller, A., "Isolating failure-inducing thread schedules," in *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, 2002.

[7] Cleve, H. and Zeller, A., "Locating causes of program failures," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2005.

[8] Cooperation, M. `http://www.mostcooperation.com`.

[9] CTAS, "Center TRACON automation system." `http://www.ctas.arc.nasa.gov`.

[10] Damm, W. and Harel, D., "LSCs: Breathing life into message sequence charts," *Formal Methods in System Design*, 2001.

[11] dynamic slicing tool for Java, J., "T. Wang and A. Roychoudhury and L. Guo, National University of Singapore." website: `http://jslice.sourceforge.net`.

[12] Feldman, Y. and Schneider, H., "Simulating reactive systems by deduction," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 2, no. 2, 1993.

[13] Ferrante, J., Ottenstein, K., and Warren, J., "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.

[14] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns*. Addison-Wesley, 1995.

[15] "The gnu project debugger." website: `http://www.gnu.org/software/gdb/gdb.html`.

[16] "The java debugger." website: `http://java.sun.com/`.

[17] Groce, A. and Visser, W., "What went wrong: Explaining counterexamples," in *SPIN Workshop on Model Checking of Software*, pp. 121–135, 2003.

[18] Guo, L. and Roychoudhury, A., "Software model backward association," in *Asian Working Conference on Verified Software (AWCVS)*, 2006.

[19] Guo, L., Roychoudhury, A., and Wang, T., "Accurately choosing execution runs for software fault localization," in *Compiler Construction (CC)*, 2006.

[20] Gyimóthy, T., Beszédes, A., and Forgács, I., "An efficient relevant slicing method for debugging," in *7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 303–321, 1999.

[21] Harel, D., "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[22] Harel, D. and Gery, E., "Executable object modeling with statecharts," *IEEE Computer*, vol. 30, no. 7, 1997.

[23] Harrison, W., Barton, C., and Raghavachari, M., "Mapping UML designs to Java," in *Intl. Conf. on Object-oriented Prog. Sys. and Languages (OOPSLA)*, 2000.

[24] Heimdahl, M. and Whalen, M., "Reduction and slicing of hierarchical state machines," in *Intl. Symp. on Foundations of Software Engineering (FSE)*, 1997.

[25] Jones, J. A., Harrold, M. J., and Stasko, J., "Visualization of test information to assist fault localization," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, pp. 467–477, 2002.

[26] "The kaffe Java virtual machine." website: `http://www.kaffe.org`.

[27] Kohler, H. J., Nickel, U., Niere, J., and Zundorf, A., "Integrating UML diagrams for production control systems," in *Intl. Conf. on Software engineering (ICSE)*, 2000.

[28] Korel, B. and Laski, J. W., "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, 1988.

[29] Korel, B. and Rilling, J., "Application of dynamic slicing in program debugging," in *International Workshop on Automatic Debugging*, 1997.

[30] Lamport, L., "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, pp. 558–565, 1997.

[31] Levrouw, L. J., Audenaert, K. M. R., and Campenhout, J. M., "A new trace and replay system for shared memory programs based on lamport clocks," in *Euromicro Workshop on Parallel and Distributed Processing*, pp. 471–478, 1994.

[32] Lucia, A. D., "Program slicing: Methods and applications," in *IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 142–149, 2001.

[33] Nevill-Manning, C. G. and Witten, I. H., "Linear-time, incremental hierarchy inference for compression," in *Data Commpression Conference (DCC)*, pp. 3–11, 1997.

[34] Nguyen, K., Sun, Z., Thiagarajan, P., and Wong, W.-F., "Model-driven SoC design via executable UML to systemc," in *IEEE Real-time Systems Symp. (RTSS)*, 2004.

[35] Object Management Group, Inc, "UML Specification." `http://www.uml.org`.

[36] Pytlik, B., Renieris, M., Krishnamurthi, S., and Reiss, S. P., "Automated fault localization using potential invariants," *CoRR*, vol. cs.SE/0310040, Oct, 2003.

[37] Renieris, M. and Reiss, S. P., "Fault localization with nearest neighbor queries," in *Automated Software Engineering (ASE)*, pp. 30–39, 2003.

[38] Reps, T. W., Ball, T., Das, M., and Larus, J. R., "The use of program profiling for software maintenance with applications to the year 2000 problem," in *ACM SIGSOFT Symp. on the Foundations of Software Engg. (FSE)*, 1997.

[39] Research, I., "Concurrent benchmark." website: `https://qp.research.ibm.com/concurrency_testing`.

[40] Shuttle_Control_System, "New rail-technology Paderborn." `http://wwwcs.uni-paderborn.de/cs/ag-schaefer/CaseStudies/ShuttleSystem`.

[41] Sinha, S. and Harrold, M., "Analysis and testing or programs with exception handling constructs," *IEEE Transactions on Software Engineering*, vol. 26, no. 9, pp. 849–871, 2000.

[42] Tip, F., "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, no. 3, pp. 121–189, 1995.

[43] tool, R., "I-logix, inc.." website: `http://www.ilogix.com`.

[44] tool, S., "The MathWorks, inc.." website: `http://www.mathworks.com`.

[45] Wang, T. and Roychoudhury, A., "Using compressed bytecode traces for slicing Java programs," in *Intl. Conf. on Software Engineering (ICSE)*, 2004.

[46] Wang, T. and Roychoudhury, A., "Automated path generation for software fault localization," in *ACM/IEEE International Conference on Automated Software Engineering (ASE), Short Paper*, 2005.

[47] Wang, T. and Roychoudhury, A., "Hierarchical dynamic slicing," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2007.

[48] Wang, T. and Roychoudhury, A., "Dynamic slicing on Java bytecode traces," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, To appear.

[49] Wasowski, A., "On efficient program synthesis from statecharts," in *Intl. Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES)*, 2003.

[50] Weiser, M., "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 352–357, 1984.

[51] Xu, B., Chen, Z., and Yang, H., "Dynamic slicing object-oriented programs for debugging," in *IEEE International Workshop on Source Code Analysis and Manipulation*, 2002.

[52] Zeller, A., "Isolating cause-effect chains from computer programs," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pp. 1–10, 2002.

[53] Zeller, A. and Hildebrandt, R., "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, 2002.