Debugging Statecharts via Model-Code Traceability

Liang Guo and Abhik Roychoudhury

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

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

Keywords: Statecharts, Traceability, Debugging, Slicing

1 Introduction

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

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

1.3 Exceptions

Even if a statement or expression is syntactically correct, there might arise an error during its execution. For example, trying to open a file that does not exist, division by zero and so on. Such types of errors might disrupt the normal execution of the program and are called exceptions.

An exception is a Python object that represents an error. When an error occurs during the execution of a program, an exception is said to have been raised. Such an exception needs to be handled by the programmer so that the program does not terminate abnormally. Therefore, while designing a program, a programmer may anticipate such erroneous situations that may arise during its execution and can address them by including appropriate code to handle that exception.

It is to be noted that SyntaxError shown at Figures 1.1 and 1.3 is also an exception. But, all other exceptions are generated when a program is syntactically correct.

1.4 Built-in Exceptions

Commonly occurring exceptions are usually defined in the compiler/interpreter. These are called built-in exceptions.

Python's standard library is an extensive collection of built-in exceptions that deals with the commonly occurring errors (exceptions) by providing the standardized solutions for such errors. On the occurrence of any built-in exception, the appropriate exception handler code is executed which displays the reason along with the raised exception name. The programmer then has to take appropriate action to handle it. Some of the commonly occurring built-in exceptions that can be raised in Python are explained in Table 1.1.

Table 1.1 Built-in exceptions in Python

S. No	Name of the Built-in Exception	Explanation
1.	SyntaxError	It is raised when there is an error in the syntax of the Python code.
2.	ValueError	It is raised when a built-in method or operation receives an argument that has the right data type but mismatched or inappropriate values.
3.	IOError	It is raised when the file specified in a program statement cannot be opened.

Readability Metrics

A study involving 120 computer science students was conducted in an attempt to come up with a technique for modelling code based on its metrics of readability. The focus of the study was to assign a certain piece of code to the test subjects after which they completed a series of snippets, each snippet graded and corresponding to a unique aspect of the code. For instance, a typical snippet may query the subject about the difficulty of some specific part of the code, and request a grading on a scale from one to ten. The conclusion of the study was that it is very much viable to develop one such technique for modelling code and, in this case, one that shows a significant correlation with more conventional metrics. [2]

The effects of code styling on readability

File organisation, indentation, white space and naming conventions are just a few factors that can be grouped up and defined as code styling, with its purpose to make code more readable and easier to understand. Over half of the lifetime of a software is spent maintaining it and its usually maintained by more than one developer. This is all claimed in [1] where research was done on several open-source Java projects to find out whether the number of violations of a chosen code convention correlated with the projects readability score. They used the metric described in the previous section.

Eye-tracking

There are generally two different ways of monitoring eye movement using an eye-tracker, either measure the position of the eye relative to the head or the orientation of the eye in space. Eye-tracking technology has been used to analyse the way we look at things in a computer environment before. A study comparing three graph representation (force-directed, layer-based and orthogonal) layouts showed that a force-directed approach provided the best visualisation experience [8]. Another study looked at how different representation of number display (percentages versus fractions, digits versus words and rounding versus decimals) affected people with dyslexia [9].

Introduction

Code readability can be defined as a way of determining how easily code can be comprehended by the reader and is closely related to the life-cycle of any software product [1]. It revolves around aspects such as code maintainability, reuse and situations in which a programmer familiar with a code passes it on to another, perhaps less familiar, programmer; a situation in which it is important that this transition is smooth and that the new person can resume work with the code as smoothly as possible.
Using eye-tracking technology it is possible to analyse the readability of code by having subjects with various knowledge in programming participate in a range of experiments and by observing their behaviour and way of thinking towards some particular piece of code. Such knowledge is interesting as it allows for programming languages to further become more interactive and more easily understood by assigning them higher readability. A higher readability, in turn, improves the maintainability, reuse and general functionality of the code. [2]
This report aims to show the correlation between how code appears regarding styling, syntax and various other features that may affect readability and how easily the code may be interpreted by a programmer with concern to his/her level of programming skill. The skill level of the testing programmers will vary from 1 to 5+ years of programming studies/work experience. In order to collect and examine data and obtain relevant results, Eye tracking technology will be used to perform the experiments. More information about the tools and equipment used will be provided in the Methods section.

Problem Statement

The aim of this report is to explain and identify the correlations between how well a code is written with regards to code formatting, syntax, etc. and how easily the code can be read and interpreted by a programmer. By gathering data and information using forms and eye-tracking technology, the objective is to reach a conclusion as well as make a clear statement about whether code becomes easier to read and interpret or not. Thus, this report aims to answer the following query;

Can the readability and interpretation of code be visibly improved by adding features such as syntax highlighting, code indentation, comments and/or logical variable naming?

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

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

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

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

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

---

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

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

The Role of Algorithms in Computing

Algorithms play a crucial role in computing by providing a set of instructions for a computer to perform a specific task. They are used to **solve problems** and **carry out tasks** in computer systems, such as **sorting data**, **searching for information**, **image processing**, and much more. An algorithm defines the steps necessary **to produce the desired outcome**, and the computer follows the instructions to complete the task efficiently and accurately. The development of efficient algorithms is a central area of computer science and has significant impacts in various fields, from cryptography and finance to machine learning and robotics.

Algorithms are widely used in various industrial areas to improve efficiency, accuracy, and decision-making. Some of the key applications include:

1.Manufacturing: Algorithms are used to optimize production processes and supply chain management, reducing waste and

3.Healthcare: Algorithms are used to process and analyze medical images, assist in diagnosing diseases, and optimize treatment plans.4Retail: Algorithms are used for customer relationship management, personalized product recommendations, and pricing optimization.

4.Transportation: Algorithms are used to optimize routes for delivery and transportation, reducing fuel consumption and increasing delivery speed.

5.Energy: Algorithms are used to optimize energy generation, distribution, and consumption, reducing waste and increasing efficiency.

6.Security: Algorithms are used to detect and prevent security threats, such as hacking, fraud, and cyber-attacks.

In these and many other industries, algorithms

3.Healthcare: Algorithms are used to process and analyze medical images, assist in diagnosing diseases, and optimize treatment plans.4Retail: Algorithms are used for customer relationship management, personalized product recommendations, and pricing optimization.

4.Transportation: Algorithms are used to optimize routes for delivery and transportation, reducing fuel consumption and increasing delivery speed.

5.Energy: Algorithms are used to optimize energy generation, distribution, and consumption, reducing waste and increasing efficiency.

6.Security: Algorithms are used to detect and prevent security threats, such as hacking, fraud, and cyber-attacks.

In these and many other industries, algorithms [illegible] crucial role in automating tasks