UNDERSTANDING THE EFFECTS OF MODEL EVOLUTION THROUGH INCREMENTAL TEST CASE GENERATION FOR UML-RT MODELS

by

ERIC JAMES RAPOS

A thesis submitted to the School of Computing

in conformity with the requirements for

the degree of Master of Science

Queen's University

Kingston, Ontario, Canada

September, 2012

Abstract

Model driven development (MDD) is on the rise in software engineering and no more so than in the realm of real-time and embedded systems. Being able to leverage the code generation and validation techniques made available through MDD is worth exploring, and is the focus of much academic and industrial research. However given the iterative nature of MDD, the natural evolution of models causes test case generation to occur multiple times throughout a software modeling project. Currently, the existing process of regenerating test cases for a modified model of a system can be costly, inefficient, and even redundant.

The focus of this research was to achieve an improved understanding of the impact of typical model evolution steps on both the execution of the model and its test cases, and how this impact can be mitigated by reusing previously generated test cases.

In this thesis we use existing techniques for symbolic execution and test case generation to perform an analysis on example models and determine how evolution affects model artifacts; these findings were then used to classify evolution steps based on their impact. From these classifications, we were able to determine exactly how to perform updates to existing symbolic execution trees and test suites in order to obtain the resulting test suites using minimal computational resources whenever possible.

The approach was implemented in a software plugin, IncreTesCaGen, that is capable of incrementally generating test cases for a subset of UML-RT models by leveraging the existing testing artifacts (symbolic execution trees and test suites), as well as presenting additional analysis results to the user.

Finally, we present the results of an initial evaluation of our tool, which provides insight into the tool's performance, the effects of model evolution on execution and test case generation, as well as design tips to produce optimal models for evolution.

Acknowledgements

I would first like to take this opportunity to thank my supervisor, Juergen Dingel, for providing me the opportunity to study under him. It has been an excellent learning experience, both in terms of the material and for self-growth. His supervisory style allowed me to explore the work freely on my own, and provided guidance when needed, which was certainly a big part of the success of this work.

Secondly, I would like to thank the members of my lab, who have provided support in times of need. The MASE Group has been my home for the past two years, and the other members have made it a great place to be, so thank you Ernesto, Gehan, Amal, Tawhid, Eyrak, Nick, Mark and especially Karolina, who has been a big help to me in my research, using her valuable time to help me understand a concept, or working on a new implementation. Additionally, I would like to thank people in our extended lab, members of the STL whom I have shared office space with, and had the pleasure of socializing with in down times, so a big thanks to Dr. Scott Grant, Doug, Matthew, Andrew and Paul. There have been some interesting times in our lab and these people have made the experience that much better. There are too many people within the School of Computing to thank individually, but it is needed to be said that this place as a whole has certainly been a contributing factor to the amazing times I have had here.

I would like to thank my industrial sponsors for their support, for without them this work would not have been possible. Thanks to IBM Research, the Ontario Centers for Excellence, Malina Software Corporation, and the Natural Sciences and Engineering Research Council of Canada.

To my family, I cannot thank you enough for the support you have shown me, my mother Janet, and father Jim have been nothing but supportive of my graduate studies. To the friends I have made along the way, I can’t thank you enough for your support outside of the research, because that too is important, because as they say, all work and no play makes Eric go crazy, or something like that. I would also like to thank my roommate Jesse, who has been a great asset to have around, both academically and supportively. Throughout my Masters, there have been two close friends I have really come to count on for support, so thank you Melissa and Lizzie!

Table of Contents

List of Tables

List of Acronyms

EMF – Eclipse Modeling Framework

GUI – Graphical User Interface

IDE – Integrated Development Environment

MDD – Model-driven development

PC – Path Constraint

RSA-RTE – Rational Software Architect RealTime Edition

SAUML – Symbolic Analysis of UML-RT Models (Plugin implementing Symbolic Execution)

SET – Symbolic Execution Tree

UML – Unified Modeling Language

UML-RT – UML-RealTime

List of Figures

Chapter 1. Introduction

1.1 Motivation

Model-driven development (MDD) is an iterative form of software development, which presents challenges for test case generation. Due to this iterative nature, and the often minute evolution steps performed to enhance or fine-tune a software model, an accurate test suite for that model would require numerous changes during the development phases.

The problem of automated test case generation has long been investigated [8] [9] [10] [12] [13] [23] [24] [25] [26], and many elegant solutions exist, but in the presence of model changes there is often no need to regenerate an entire new test case.

Given the inefficiency of a complete regeneration of test cases after every evolution step, we aimed to reuse existing test cases in order to avoid any unnecessary generation. Working specifically with UML-RT models, we aimed to solve this problem by incrementally generating the test cases each time a model evolution step occurs. Instead of regenerating an entire test suite we concluded that examining the existing test cases and determining which tests need to be updated, removed or whether or not more tests were required might lead to a more efficient technique to generate test cases in the presence of model changes.

In addition to the effective implementation, motivation for this project stemmed from the desire to better understand the effects of model evolution. By determining how evolution steps impact execution and test case generation, we are be better able to understand how models evolve over time, and how this evolution can be supported efficiently.

1.2 Contribution

This thesis makes the following contributions: an implementation of incremental test case generation, a classification of model evolution steps and their effects on test cases, and an additional method of analysis for model evolution, including test case selection.

Because we first required a list of rules pertaining to each model evolution steps we first needed to classify the effects of model evolution on test case generation, and therefore symbolic execution as an intermediary step. We were able to complete a study that gave conclusive results of how each evolution step (a full suite defined within the research) affected both execution and tests, and these effects were documented for use in the tool, as well as in related research.

Using the classifications discovered, along with existing algorithms, a plugin for RSA-RTE was developed that successfully reuses existing test cases, making only the necessary modifications, thus reducing redundancy and improving efficiency.

Industry models are often extremely large, and the execution of test suites can take in the order of days to execute; this is undesirable. Using our tree differencing approach, we are able to further examine symbolic execution trees and perform different types of test case selection to avoid the entire re-execution of a test suite. This technique allows the user to decide on more restrictive criteria for the tests, and select them appropriately, while still ensuring that the appropriate amount of testing occurs. This reduction in the total number of tests run can greatly improve productivity in an industrial setting. One such method of test selection is to run only the newly generated or modified tests, and to skip execution of unaltered tests, which is made possible through a notification to the user of which tests have been altered or newly added. Additionally, we explore a method of evolution impact analysis on execution.

1.3 Chapter Overview

In Chapter 2 we present some of the background that our work is based on. We discuss the tools and technologies that have been used, and their specific relationship to this project. This section makes reference to related projects and their relevance to this work.

Chapter 3 serves as an overview of the research, providing a clear, high-level, description of the process. It includes a brief look at the methodology and theory behind the work itself. Finally, a summary of the work as a whole is presented.

Chapters 4 through 7 outline the research conducted in detail. Specifically, Chapter 4 details the initial model selection and analysis for classification of model evolutions; results are presented in terms of classifying these effects. Additionally, Chapter 4 looks into the initial test case generation and examination, providing insight into how it is used within the research. Chapter 5 focuses on the comparison of UML-RT models to determine the appropriate update action, specifically looking at the integrated differencing tool that is part of the existing development environment. Chapter 6 serves as a description of the actual update of symbolic execution trees and test suites. Chapter 7 discusses the development of the final tool using the work from the previous chapters.

Chapter 8 provides a methodology for, and the results of our tool validation. Interesting findings are related and a number of claims are made about model evolution in relation to tool performance.

Finally, in Chapter 9, we conclude with some observations, and summarize the work. We discuss some of the limitations and why they exist. Additionally, some possible future work is presented.

Chapter 2. Background and Related Work

2.1 UML-RT

UML-RT [7] is a real-time profile of the Unified Modeling Language (UML) [6], dealing specifically with modeling real-time interactions between a system and its surrounding environment.

The main difference between a traditional UML model representation and a UML-RT representation is the notion of capsules. A capsule is a singular component of a system that may interact with other capsules via protocols; each capsule's behaviour is modeled using a state machine. The main difference between UML-RT state machines and standard UML state machines is that UML-RT state-machines do not contain any orthogonal regions.

Two of the main artifacts of a UML-RT model are State-Machine Diagrams and Structure Diagrams. The state-machine defines behavior, and the structure diagram defines structure and connectivity. As stated, capsules are connected to other capsules via ports, which use protocols to communicate with each other. Capsules may also contain other capsules; this nesting is shown in the structure diagram of the model as well as in the main class diagram.

Transitions in UML-RT state machines are triggered by the occurrence of some event; the main type of event that triggers a transition is the receiving of a signal on a given port. Execution will wait until that specific signal is received on a port (which may be sent by an internal capsule or from an outside source); the state machine then executes the transition and execution continues. The next type of action that can trigger a transition is a timer (the introduction of the real-time aspect). Timers can be set to timeout after a given amount of time or recurring over a set interval, and this allows execution to react to time, in addition to signals. When a timer times out it sends a signal that is received by the state machine.

Because state machines require signals to be received to continue execution, it is evident that a state machine must be able to send signals as well. When connected to a protocol via a port, a capsule is able to send any of the predefined signals over the protocol to be received by a connected capsule.

For those unfamiliar with software models, specifically state-machines, an initial concern may arise from the inability to add code to a model to influence behavior; however this is not an issue. As with traditional state machines, UML-RT state machines allow the addition of action code throughout execution. Developers may place action code on transitions, as well as within a state as either entry or exit actions. This ability allows developers to further control how a state machine influences execution. The most common purposes of action code are: updating attribute values, sending signals on ports, or setting timers, however more options are possible.

2.1.1 Uses of UML-RT

UML-RT lends itself to the modeling of reactive systems quite nicely since they are systems that respond (react) to external events, including the passage of time.

Currently, much focus for reactive systems is on embedded systems. Current advances in technology such as mobile devices have led to a shift in focus that favours real-time modeling, and UML-RT is becoming more and more useful for this reason.

The first example of real world use of UML-RT, as mentioned above, is within mobile devices and communications [21]. There is a tremendous focus on development for mobile phones, tablets, and similar devices, all of which are suited for development using UML-RT. A mobile device reacts to user input in the form of button presses, outside stimulus such as incoming calls, as well as internal timers. This example will be further explored in detail in this section.

Another example of UML-RT's use is in development of automobile software [20]. A piece of machinery with a number of disjoint, but connected, systems that need to communicate and function with a precise accuracy is a piece of machinery where the software needs to be well designed and tested. UML-RT provides the perfect development environment to do this, by allowing simulation to occur on the model, and systematic testing as an attempt to avoid errors. In a safety critical system such as a brake control system in a vehicle, an error can cost lives.

2.1.2 Mobile Phone Example

In this section, we demonstrate the usefulness of UML-RT in modeling a real-time embedded system, specifically a mobile phone. This will be a simplified look at models for some of the behaviour of a mobile phone, in order to demonstrate some of the features of UML-RT in detail.

We begin by looking at the notion of timers, which are used in mobile devices to automatically turn off a display after a period of time, among other tasks. When thinking of this in terms of states, you would have a *displayOn* state and a *displayOff* state. The *displayOff* state would have a transition to *displayOn* that is triggered by a number of things such as a button press, an incoming call, or any other notification. There would also be a transition in the reverse direction, triggered by the user pressing a specific button to turn off the display, or the timer timing out. For this particular example, Figure 2.1.1 shows a state machine with the two states and the transitions between them (button press and timeout).

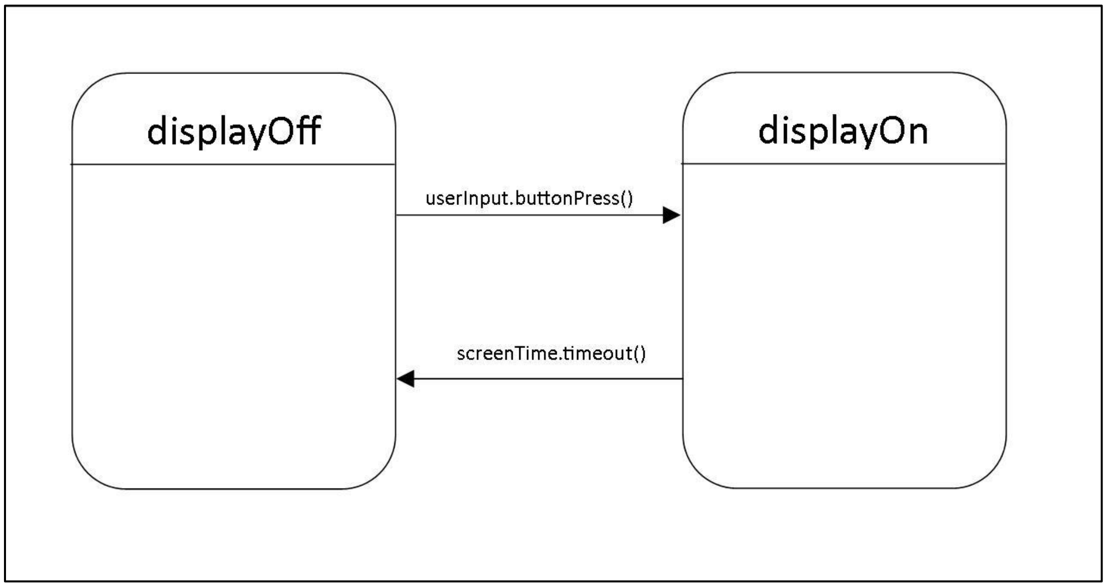


Figure 2.1.1 UML-RT State Machine showing Control of a Screen

Another feature of UML-RT is hierarchical states. The example that is used to demonstrate this shows the possible events that can occur from a *home* state when the display is on. The whole example takes place within the *displayOn* state described above. In this simplified example, we see that from the *home* state, execution can move into a *dialing* state by pressing a number button, a *menu* state by pressing the menu key, or an *incoming* state when a call is received on the network. From any of these states, execution returns to the *home* state by pressing the end call button. Figure 2.1.2 shows a very simplified version of events that can occur while the display is on.

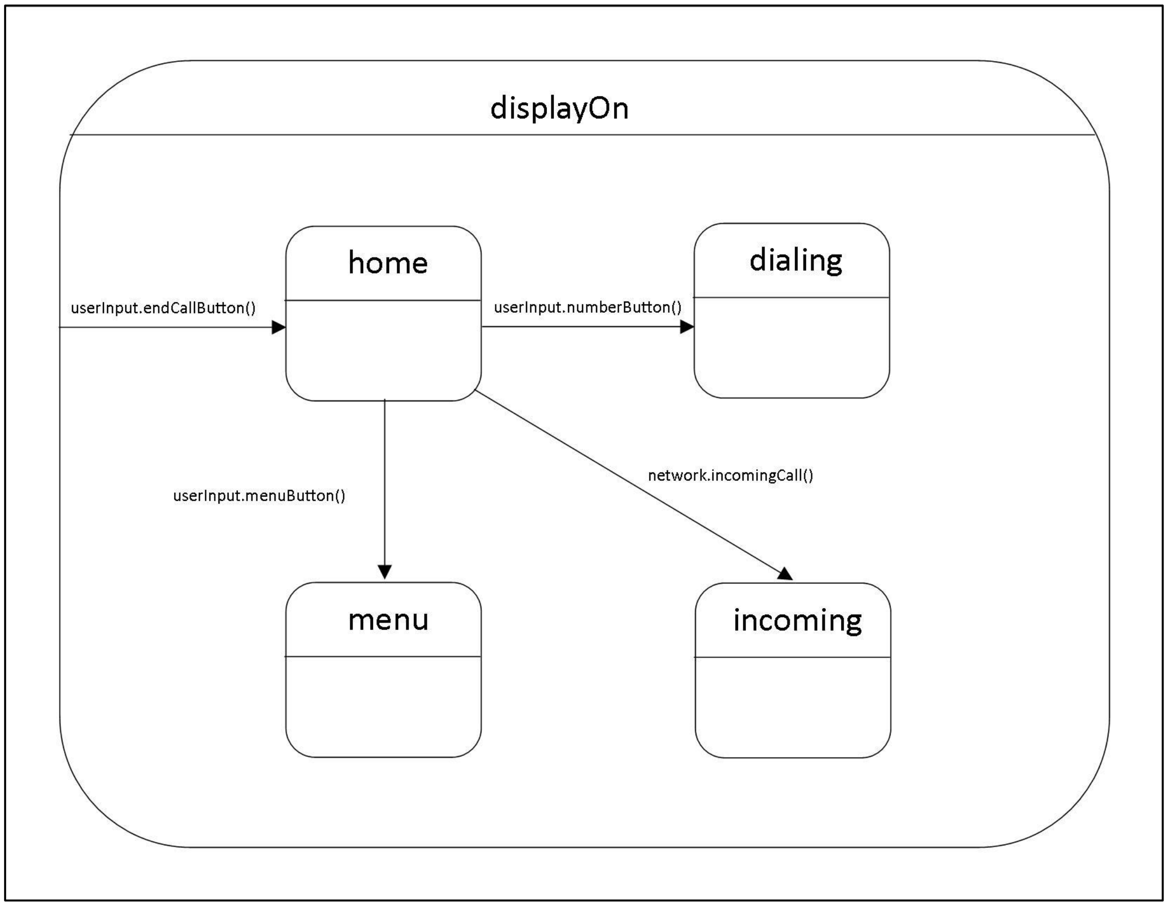


Figure 2.1.2 State Machine Showing Functionality When Screen is On

A benefit of using UML-RT to model this type of device comes from the ability to model communication between contained systems. This is shown by signals coming in on the port entitled *userInput*, and to this state machine, it is irrelevant what those signals mean or where they originated, just that it must react to them. However, on the development side, it can be quite relevant to know exactly what can send a particular signal, which is why design practices are extremely important. In this example, this can be shown through the containment of similar tasks occurring on a single protocol. Shown are both the *userInput* and *network* ports that connect to protocols dealing with these actions. Additionally, individual state machines are utilized to model individual components, focusing on certain behaviours, such as the examples above focusing only on the behaviour of the display.

Relationships between capsules are modeled using a structure diagram. For this simplified example, we show a *mobilePhone* capsule, and within it are two subsystems: the *display*, and the *userInputDevices*. These two capsules are then connected via ports over a protocol.

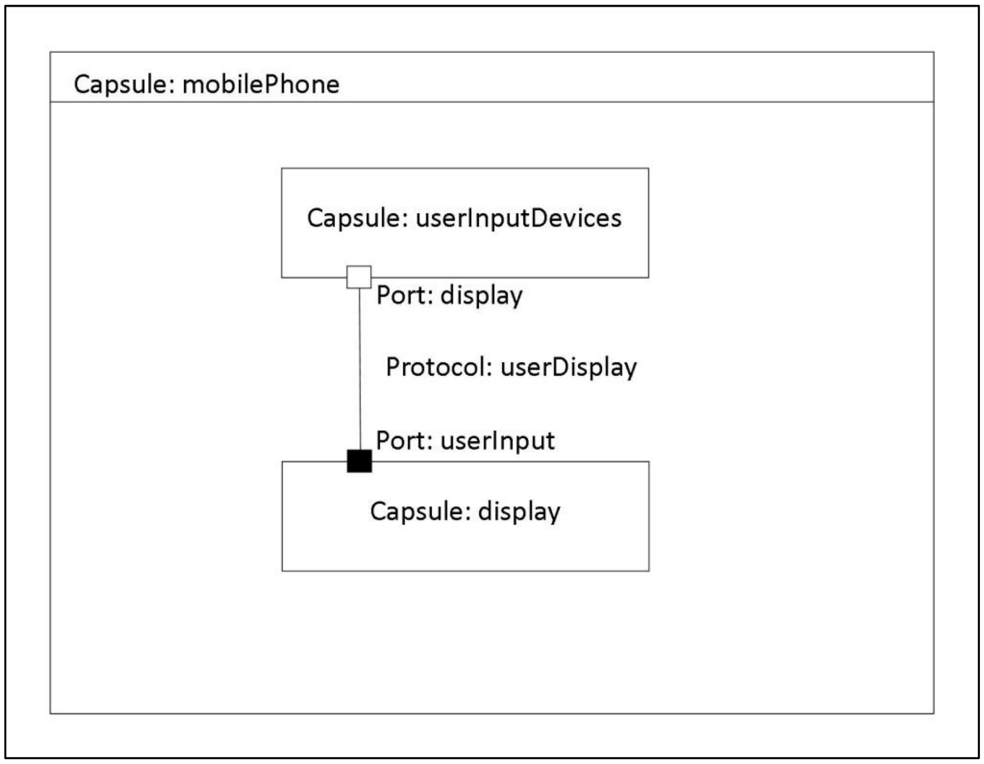


Figure 2.1.3 Structure Diagram for a Mobile Phone

Both of the contained capsules depicted in Figure 2.1.3 also have structure diagrams of their own that detail their internal structure, showing things such as the timer ports that are created, as well as the end points for any of the external protocols. For example, the port *userInput* would end inside of the *display* capsule at a state machine, meaning that the end point for signals on that protocol would be to influence behavior of the *display* state machine. Additionally ports can act as relay ports, connecting to internal capsules. For example, although not depicted above, there would be an external port on the *mobilePhone* capsule that would connect the phone to the network, this network port would then be connected to the *display* capsule; thus the signal is relayed by the *mobilePhone* to the *display*.

2.1.3 IBM Rational Software Architect – RealTime Edition

For this research the environment used for UML-RT development is IBM Rational Software Architect-RealTime Edition (IBM RSA-RTE) Version 8.0 [16], which is an Eclipse-based IDE. The implementation is a recent adaptation of Rational Software Corp.'s RoseRT [17] that was redesigned when IBM purchased Rational.

RSA-RTE uses the frameworks defined for RoseRT, which include a full set of libraries to implement timing, logging to terminals, sending and receiving signals over ports, and all required functionality of UML-RT.

RSA-RTE provides a GUI in which users can edit their models through a typical drag and drop methodology from a palette of model elements. Building on the success of the Eclipse IDE, RSA-RTE makes excellent use of the Properties tab to allow the user to modify all properties of any element. The Code View also allows the user easy access to input action code on transitions and entry and/or exit actions.

With the environment being Eclipse based, the idea of projects carries over, and users can have multiple models per project and the encapsulation provides for excellent organization. Another benefit of RSA-RTE being Eclipse based is the ability to develop and use plugins to enhance functionality of the tool, which we have done through development of our tool, IncreTesCaGen.

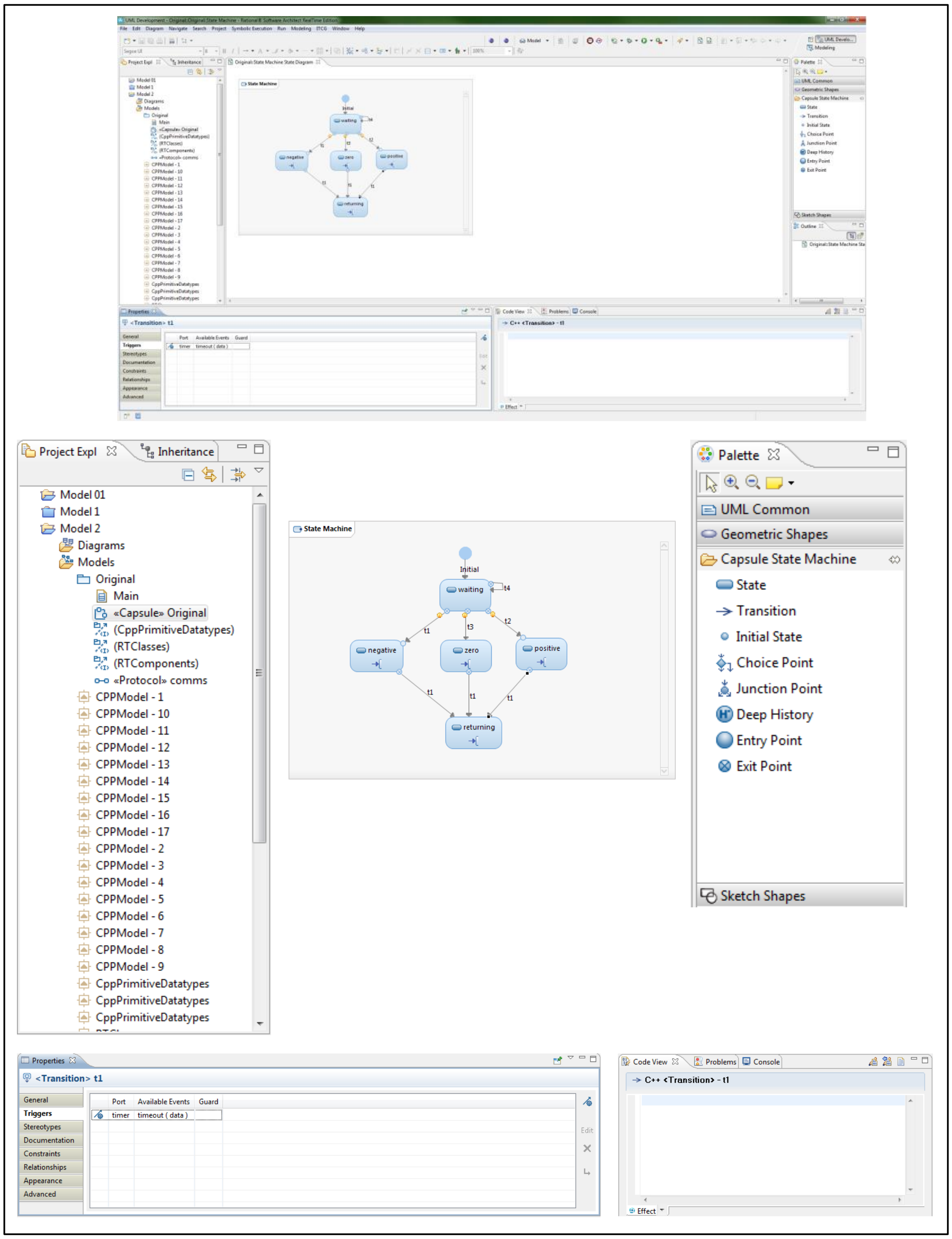


Figure 2.1.4 RSA-RTE Environment (With Enlarged Features)

Figure 2.1.4 shows the full RSA-RTE environment. For those familiar with Eclipse this will look very similar, with a few differences. The main area (center) of this view is showing a state machine that contains five states and the transitions between them. The area on the top right is the palette, which contains the model elements that can be inserted into the state machine (state, transition, initial state, etc.). The area on the top left is the project explorer, which is native to Eclipse, and shows the organization of models into projects, and the elements of each model, including available diagrams. The bottom left area shows the properties view, where the user can edit properties of the currently selected object; in Figure 2.1.4 a transition is selected, and the current view shows the triggers for that transition. The last area of the tool shown on the bottom right is the code view window, which is where action code or guards on transitions can be written. As stated a transition is selected, and this transition does not contain any action code, hence the code view window is empty.

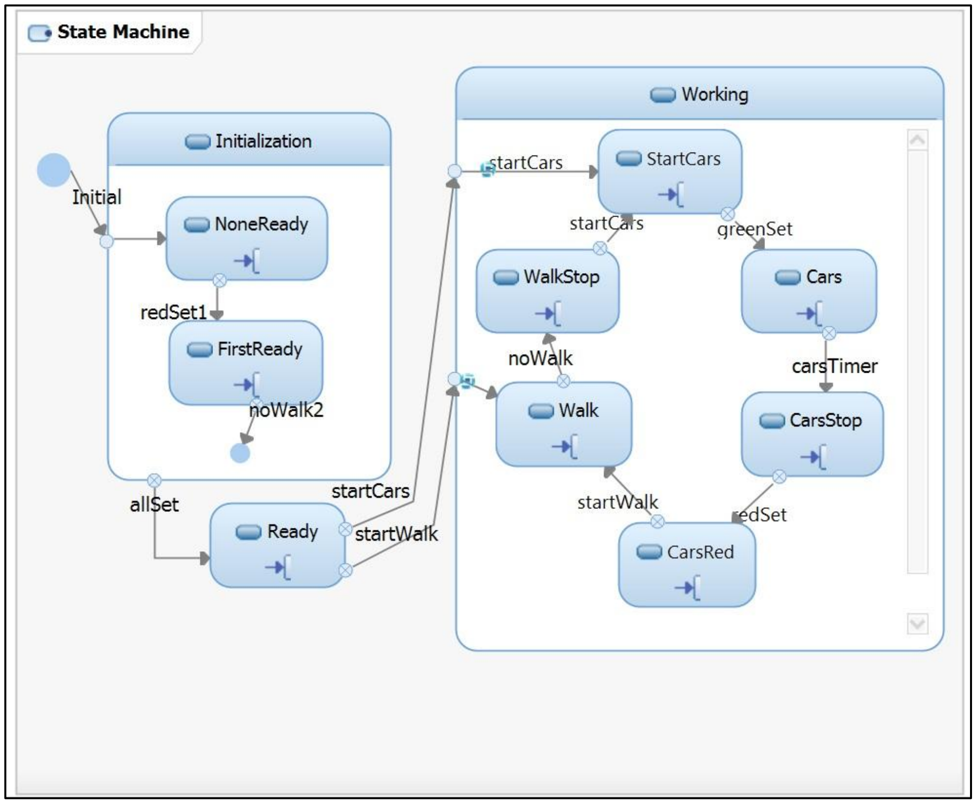


Figure 2.1.5 RSA-RTE State Machine

Figure 2.1.5 shows a closer view of a UML-RT state machine as implemented in RSA-RTE. This view shows the full state machine diagram, including indication of entry code on many of the states, shown by a right facing arrow and the left half of a state, as well as action code on a transition being shown with a blue circle on the transition itself, as seen on the *startCars* transition located in the upper left area of the *Working* state.

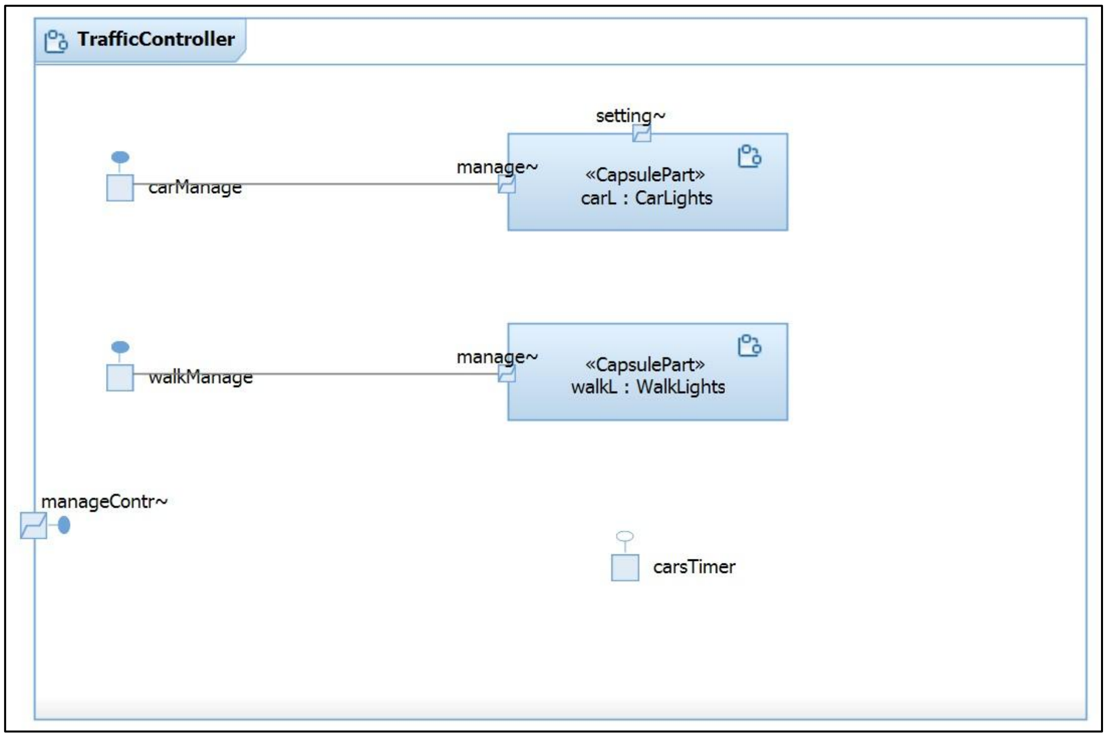


Figure 2.1.6 RSA-RTE Structure Diagram

Figure 2.1.6 shows an RSA-RTE structure diagram as implemented by the tool. A blue oval shape indicates that a port ends at a state machine; the three blue ovals in the diagram each indicate that any signals received on those ports are handled by the state machine for the *TrafficController*. The two capsules shown are contained capsules of the *TrafficController*, which communicate with each other via the ports shown. Additionally note that there is a *carsTimer* port, with a white oval, and this is a special notation for ports such as timers or logs.

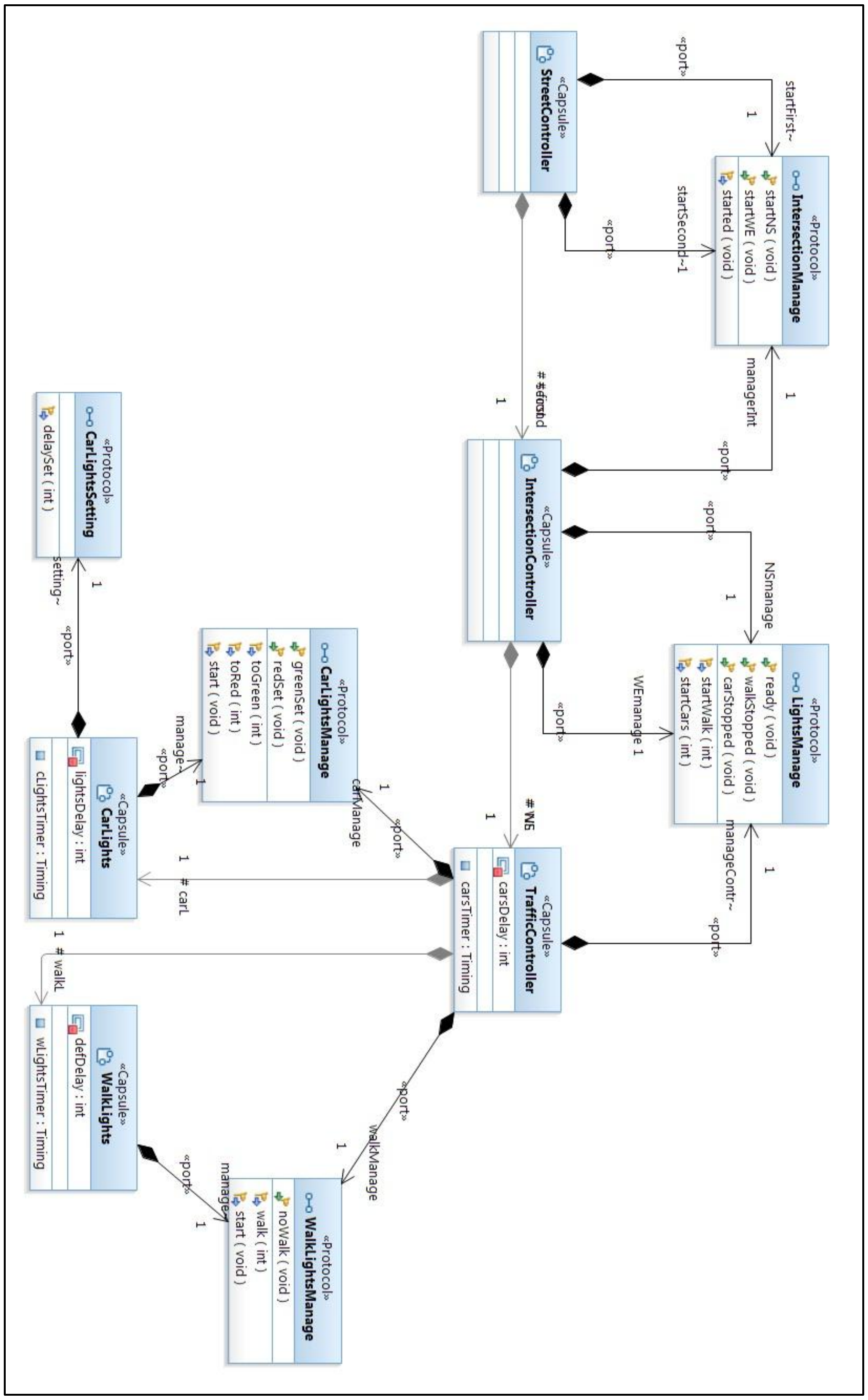


Figure 2.1.7 RSA-RTE Class Diagram

Figure 2.1.7 shows a main class diagram in RSA-RTE, showing all relations between all elements in the model. It details all of the capsules, the protocols they use (including multiplicity), containment of capsules, the ports on which a certain protocol is used, and all of the properties for each of these elements, such as signals for protocols, and additional ports and attributes for capsules.

As an end result, it is important to note that the final artifact produced by RSA-RTE is generated code. The tool is capable of generating executable code (Java or C++) from the developed models.

2.2 Symbolic Execution

Briefly, symbolic execution is a method used to simulate execution of a piece of software, using symbolic variables/symbols in place of actual input values [13]. A program begins at an initial symbolic state, and as each line of code is symbolically executed, a test occurs to see if the symbolic state has changed, and if so a new state is recorded. This is repeated for the entire program and the output is a series of paths that execution can take, dependent on the potential values of the symbolic variables.

When conditional statements arise within the code, this can often create branching within a program's execution, and this is handled within symbolic execution by generating multiple outgoing paths from a given state, and from that point on in execution, path constraints are placed on the symbolic variable that led to that particular branch.

Because of this branching, and the sequential order that is implied by the symbolic execution, the final output of symbolic execution is a tree, which is why they are known as symbolic execution trees, or SETs.

```
0  int method(int x, int y){
1        int z = 0;
2        if (x > 0)
3              z = x;
4        z = z + y;
5        return z;
   }
```

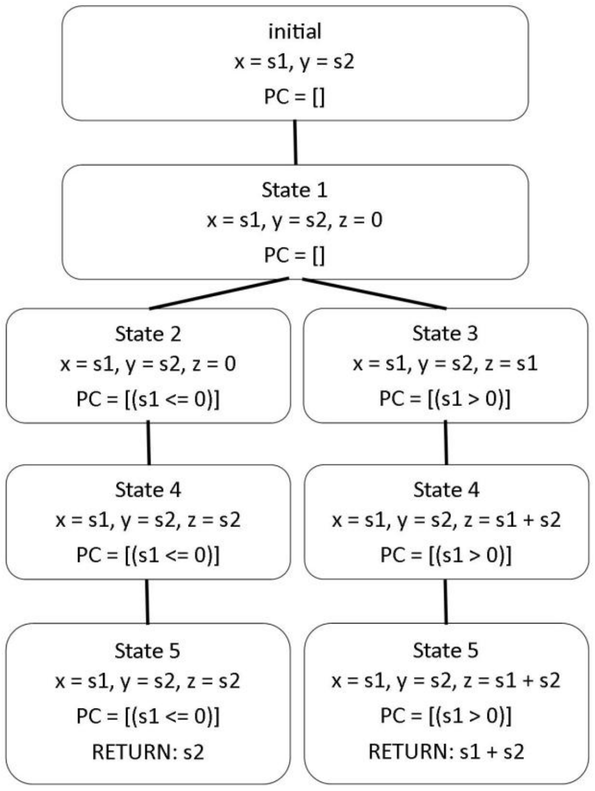


Figure 2.2.1 Example Symbolic Execution Tree

Figure 2.2.1 shows a simple example method, and the resulting symbolic execution for that method. Execution begins at line 0, or the initial state with no path constraint; the two attributes are assigned a symbolic value, denoted by s1 and s2 in this example. Execution moves to State 1, which is representative of symbolically executing line 1, which declares and initializes a variable z to the value 0, which is reflected in that state. Upon reaching line 2, there is a condition on one of the values which happens to be assigned a symbol, and since the value at execution is unknown, a branch is created for both possibilities (greater than 0, and less than or equal to zero). This leads us to two possible symbolic states, State 2 or State 3, and each of these states update their path constraints (PC) to show the constraints on the symbol s1. Line 3 is only executable if execution reaches State 3, and this is shown by updating the value of z to the symbol assigned to x. In State 2 none of the values change. From this point on in execution, each line of code must be executed in each branch, so line 4 is applied to both State 2 and State 3 and the valuations updated accordingly; notice the path constraints carry down in the tree. This is repeated until finally we reach line 5, the return statement, where the symbolic states indicate what the value being returned is, based on the symbols. Note that the constraints on symbols then denote the domain of possible values for the symbol.

In relation to the work of this thesis, recent work has been presented by Person et al. [22] on directed incremental symbolic execution. They present a study of the effects of program evolution on symbolic execution, which provides a solid background for our work, which includes our own version of incrementally performing symbolic execution. Their work deals with symbolic execution of source code through analysis of Control Flow Graphs (CFG), and not of state machines, but the resulting SETs provide the same expressiveness, and potential for test case generation as those utilized in this thesis. The procedure is very similar to the work we present, in that it takes as input an original CFG and a modified CFG, as well as a list of differences (either source code line or abstract syntax tree difference); these inputs are used to create a mapping between the base CFG and the modified CFG in order to incrementally symbolically execute the modified program by only exploring areas impacted by the changes made.

2.2.1 Symbolic Execution of UML-RT Models

Research is currently being conducted within our lab on symbolically executing UML-RT State-Machines [1] for the purpose of analysis of models [2] [3] rather than code. As with symbolic execution of code, when symbolically executing a UML-RT state machine, a symbolic execution tree (SET) is produced. A SET contains all possible execution paths of the model, based on symbolic inputs, as well as any constraints on those inputs.

Execution tends to become more complex with the shift to UML-RT models, as the non-linear execution allows for more branching to occur within a tree based on possible triggers for transitions. One place where the idea of symbolic execution does become easier to understand is the parallel drawn between the states of the model, and the symbolic states of the SET. Each time a new state is entered within the model a new symbolic state is created, and the transitions between symbolic states become closely associated with the transitions between states in the UML-RT model. It is however important to differentiate between a state and a symbolic state; a state within a state machine is akin to a program location, whereas a symbolic state is an instance representation, that contains the current active location (which state machine state is currently active), a list of the current valuations of attributes, a list of the current path constraints, and a number of enabled outgoing transitions. While symbolic states are named by the same name as the state they represent, they are not equal to, and merely an instance representation of them.

SETs in general are used for reachability testing, path constraint evaluation and of course test case generation [8][13]. Due to the nature of symbolic execution it is very useful in generating entire test suites for any given software, including the UML-RT State Machines that we are working with, which is why this medium was chosen for this work.

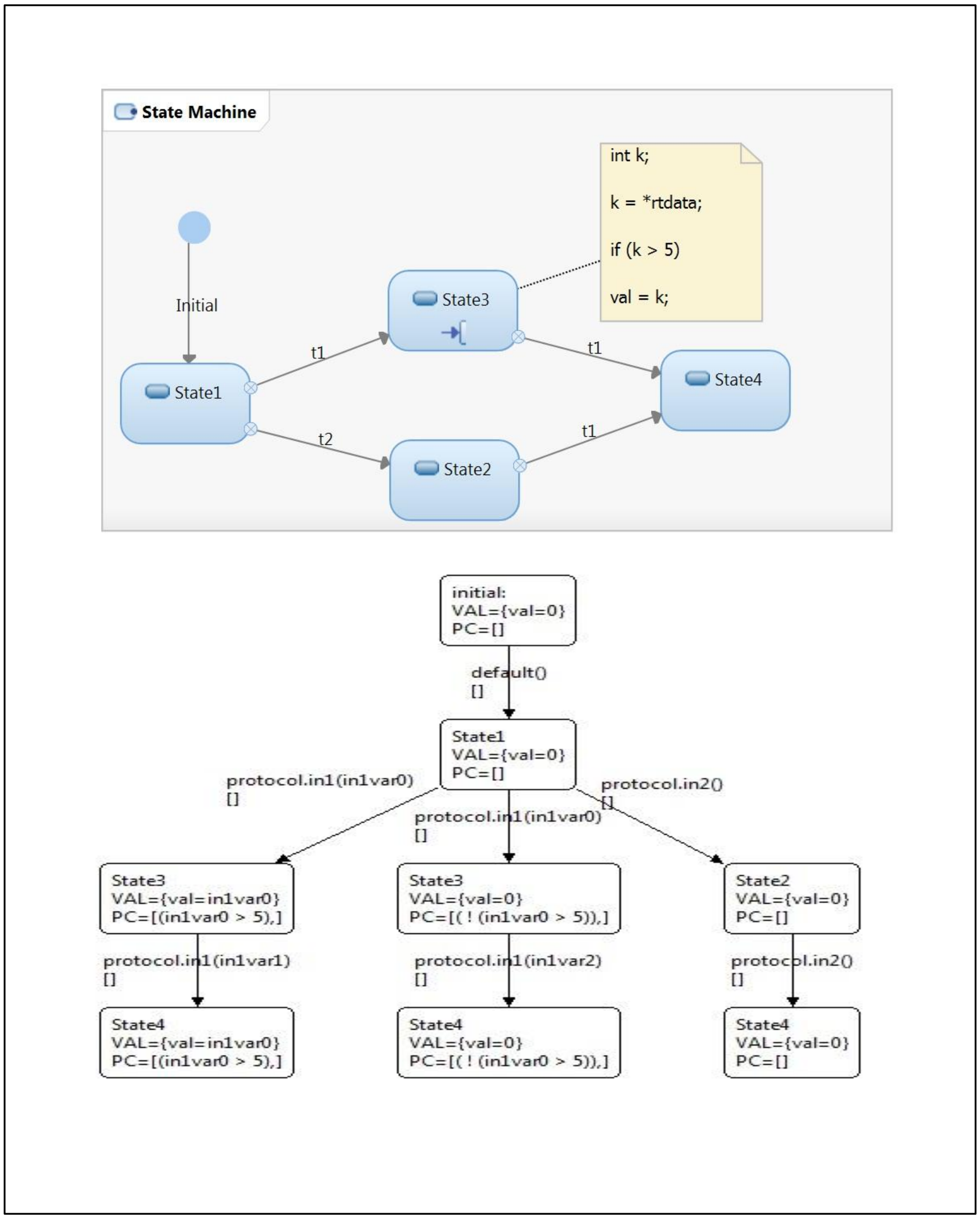


Figure 2.2.2 Example SET for a UML-RT Model

Figure 2.2.2 shows an example of a SET that has been generated from a UML-RT state machine. It is a simple example which illustrates the close relation between the states of the model and the symbolic states of the generated tree. It also shows how execution can branch due to differing transitions from a state, as well as conditional statements within action code. The assignment statement `k = *rtdata;` is the RSA-RTE syntax for assigning the data parameter from a signal that has been received.

From State 1, there are two possible execution paths within the state machine (t1 or t2), based on receiving different inputs on a port, but the SET shows three paths, due to a branching caused by the conditional statement in the entry action of State 3 (code shown in the note). This creates two symbolic instances of State 3 that can be entered, one where the condition is true ($k > 5$), and one where it is false ($k <= 5$). Any constraints on symbolic variables are recorded and will propagate through the tree.

There are number of limitations imposed on the symbolic execution of state machines by the current implementation [1] that affect the expressiveness of the resulting SET. One such limitation is the handling of loops in the state machine, which can cause a state-explosion within the SET for even the simplest of loops. The way this was handled in the implementation was to only execute the loop a bounded number of times. Another effective solution to this problem is the use of subsumption of states. A state is said to be subsumed by a similar state higher in the tree when it shares the same state machine state, same values assigned to attributes, and the same or a less tightly constrained list of path constraints. What this essentially means is that if a state is subsumed by another state there is no need to continue expanding the tree at that point, as it will continue to expand in the same manner as from the state which subsumed it. This allows for state machines that do not have a definite end state to still result in a finite tree.

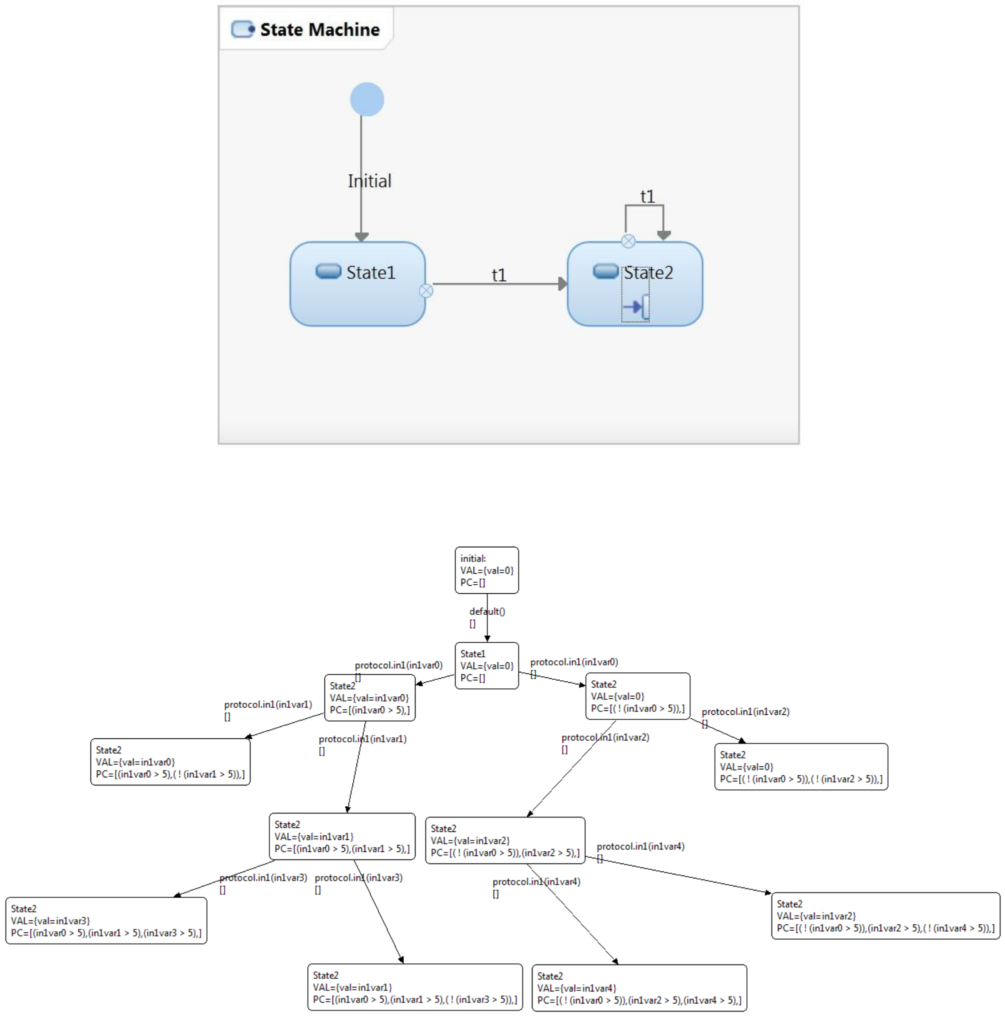


Figure 2.2.3 State Subsumption

Figure 2.2.3 shows an example of state subsumption in the generation of SETs. There exist only two states in the state machine; however one of them contains a self-transition that can execute multiple times. State 2 contains the same entry code that was found in State 3 in Figure 2.2.2 meaning that the conditional statement occurs on each loop, branching out after each new symbolic state. It is easy to deduce that this would result in an infinite tree; however we show that a tree that is essentially infinite can be represented finitely due to the subsumption of states.

Another limitation that is faced by symbolic execution of UML-RT models is the time aspect; it becomes near impossible to properly represent all possible interleavings of a timer expiring with any of the other possible events. The way this implementation deals with this limitation is to always assume the timer will timeout before any signal is received, meaning in a state where there are two or more outgoing transitions, one based on a timeout and the others on any input signal, the input signals will never be received as the timer will always be first, assuming it is initially set to timeout at some point.

2.2.2 Uses in Incremental Test Case Generation

Due to the ability to outline all possible execution paths, symbolic execution makes for an excellent method of generating test cases [8][13], but this will be discussed further in Section 2.3. In addition to its usefulness in test case generation, one of the main reasons symbolic execution was chosen as a medium for incremental test case generation was for its analysis properties [2]; the ability to examine execution paths and determine appropriate actions when modifying existing test cases is a valuable contribution. Throughout the classification of model evolution steps, symbolic execution provided an excellent vehicle to examine how model changes propagated through execution into test case generation.

2.3 Model Based Testing

Software, just like any other product, requires testing; nothing can be deemed correct without first testing its functionality. For physical products this can be as simple as using it in all of its intended uses, as well as the unintended ones to test for errors, and coming to a conclusion that the product is satisfactory. In software this process is slightly more complicated, and often formalized; testing software has become an extremely important aspect of development.

The emergence of model driven development has led to an increased demand for accurate model-based testing techniques. Since the primary artifacts for development are models, and code is generated from those models, to test the software code needed to be generated each time an update was made; this is problematic in MDD due to its iterative nature. The solution to this was to have the ability to execute models through simulation, and then to test the models as opposed to simply the code. This process became widely known as model-based testing. It is important to note that it is also possible to use the generated tests for the model to directly test the real system, as long as the code-generation step itself is trusted.

The ability to test the model itself was a great advantage as it meant only needing to generate the code once full testing had been completed on the model, ensuring its correctness and completeness. This meant that primary testing was isolated to simulation of the model.

In their work [9], Gnesi et al. discuss methods for a formal test case generation for UML statecharts. Their work is an excellent example of using models as the primary artifact for test case generation, and the similarities between UML statecharts, and the UML-RT models used in our work provides a validation of the usefulness of this type of test case generation. While their methodology differs in the absence of symbolic execution, the test suites that are developed are similar in nature to those of our work, presenting a series of inputs designed to pragmatically test execution of the statechart. Similar work dealing with generation of tests for statecharts is presented by Bogdanov et al. [12] demonstrating another formal testing notation for UML statecharts. They use the notion of a state cover (visit all states) to determine the necessary tests for a given statechart.

Work even more closely related to ours is presented by Lee et al. [10], concerning test generation for event-driven, real-time systems, also using symbolic execution as a medium for analysis. Using symbolic execution of Modecharts to create a time annotated symbolic execution tree allows them to generate tests that not only react to inputs, but to time as well. In addition, they explore a different options for completeness of test suites using symbolic execution; noting that the cycles created by reactive systems can allow an almost infinite number of possible input sequences, they explore both an “all-event-sequences” coverage criteria which generates tests for all sequences, as well as an “all-cyclic-event-sequences” coverage criteria, which ensures that all repeated event cycles are included at least once. The second method matches the type of completeness criteria explored within our work. This work provided an excellent basis for our initial test case generation by following their method of collecting all paths in a generated SET to form a complete suite of tests; this was a necessary step in incrementally generating test cases.

Pasareanu et al. [11] present a comprehensive survey on work closely related to the work presented in this thesis, demonstrating the usefulness of symbolic execution in the automatic generation of test cases. While the survey presents significant amounts of work in relation to symbolic execution and testing of source code (as opposed to models) the techniques are useful in model-based testing using symbolic execution. State subsumption is further explained, along with another SET scaling technique known as path merging. The test sequences referenced for source code testing are a number of method calls, which in the code-generated from a model, is exactly what would be required to test the system, and this similarity demonstrates promise for our work.

Another area of interest in model-based testing is the analysis capabilities provided by the process. In this thesis we discuss test case selection using SETs, which involves analyzing the generated SETs and selecting tests based on some given criteria. T. Jeron presents work on this topic [14], providing a methodology for symbolic model-based test selection. The work presents methods for conformance testing, and the completeness of test suites created through test case selection on tests generated for a model.

2.3.1 Test Case Generation

What constitutes a test case will vary from system to system, but in its simplest form, it will be a series of events that will lead to a certain execution path, given certain conditions. Values for attributes and parameters can be generated based on any constraints on them, and supplied to the program for a test execution.

It would be sufficient to examine a piece of software or a model and manually come up with a number of tests that will take the program through a number of different executions, but this process can often lead to overlooking a particular case that leads to an error in the software or even an ideal case to ensure functionality; additionally it may be impossible/infeasible. It is for these reasons that automated test case generation became an area of focus. The ability to provide a program or model of a system, and have a complete set of tests automatically generated to test all desired executions was something that was desired by software development teams. There exists a number of possible methods for generating test cases automatically which have been developed due to the understanding that generating tests by hand selection is error prone and time consuming.

However, as with any new advance, it is not without its own issues. One of the major problems that exist in test case generation is the fact that because automation more often than not employs a systematic method, a full test suite must be generated each time, meaning that work is often done many times more than necessary. It is because of this that we chose to pursue incremental test case generation, which would allow automated test case generation to reuse as much as possible of existing tests during the iterative development cycle.

2.3.2 Completeness of Test Suites

One of the most difficult aspects of generating test cases is determining when a test suite is complete. Test cases can be designed to test requirements, and a test for each of the requirements may be sufficient. The question of what constitutes a complete test case is one that is often dependent on the project itself or on the development team.

There are a number of completeness criteria, many of which are artifact specific. For example when working with code, tests can aim for statement coverage, block coverage, decision coverage, or loop coverage, each of which has some criteria to ensure that intended tests are covered. In MDD, this type of criteria is not possible as you are working with the models themselves; model-based criteria are needed. In a similar fashion to the code based testing, a large emphasis was placed on the artifacts, and working specifically for tests for state machines, two main types emerged: state coverage and transition coverage.

As their names suggest they simply require a number of test cases sufficient to ensure that each reachable state is reached and executed for state coverage, and similarly each reachable transition for transition coverage. It is for this reason that a more complete testing method has been proposed.

It is desirable to be able to test each possible execution path through the state machine. This is where the usefulness of symbolic execution becomes evident; a symbolic execution tree contains every possible execution path (some simplifications are usually made). With the recent developments in symbolic execution of UML-RT state machines [1][2], we felt it was an obvious choice for this work.

The completeness criteria chosen for our work is based on path coverage, which will be explained further in Section 2.3.3, but mainly focuses on leveraging the use of symbolic execution trees of UML-RT state machines to examine execution paths.

2.3.3 Using Symbolic Execution to Generate Test Suites

One of the forms of analysis on symbolic execution [1][2] is the automated generation of test suites for the UML-RT model. A test suite for a UML-RT state machine consists of a number of test cases, which are a series of inputs that will cause one finite execution of the state machine over some path. These inputs can be any number of things ranging from inputs over ports, or the expiration of a set timer; no matter the input, different interleavings will (likely) result in different executions.

Our implementation of test case generation using symbolic execution is explained further in Section 4.2.1, but briefly, it consists of first symbolically executing the model using an existing tool [3], and then traversing through the generated tree to obtain a number of execution paths, based on inputs. For each of these paths obtained, a number of symbolic variables are provided as inputs, but in order for a test case to be executable; these values need to be replaced with concrete values. This is achieved through solving the path constraints on all symbolic variables using the constraint solver, Choco [15]. These solved values are then substituted in for any symbolic variables, thus the generated tests are executable on the model, well as the generated real system.

When using symbolic execution to generate test suites, the satisfaction of the completeness criterion becomes evident through the artifacts available; as a generated SET shows all possible execution paths through the state machine, generating a test suite using path coverage as the completeness criteria becomes the preferred choice. By creating a test for each possible path in the SET, we are guaranteeing that a test is created for every possible execution of the model; this is of course due to the nature of symbolic execution.

Using the state machine and SET shown in Figure 2.2.2, we present an example of a test suite generated using symbolic execution. Given the generated SET, we observe there are three paths of execution through the three, which will be described using the inputs to move from one symbolic state to another, as the inputs are what make up each test case. The three paths and the associated path constraints are:

1. default( ) → protocol.in1(in1var0) → protocol.in1(in1var1)
    a. PC = {(in1var0 > 5)}
2. default( ) → protocol.in1(in1var0) → protocol.in1(in1var2)
    a. PC = {(!(in1var0 > 5)}
3. default( ) → protocol.in2( ) → protocol.in2( )
    a. PC = { }

Now that each path is obtained, the symbolic variables need to be replaced by concrete values, so for each symbolic variable (in1var0, in1var1, in1var2) they are passed to the constraint solver, Choco [15], along with the path constraints for the current path of execution. Symbolic variables with no constraints are evaluated to a default value of zero. Since the path constraints are additive the list of constraints from any leaf node will be a full set of constraints for that execution. When each set of constraints is solved, and values determined for the symbolic variables, we substitute the concrete values into the paths of inputs and obtain the three following test cases:

1. default( ) → protocol.in1(6) → protocol.in1(0)
2. default( ) → protocol.in1(0) → protocol.in1(0)
3. default( ) → protocol.in2( ) → protocol.in2( )

Note that all three tests are different and yield different executions of the model. These three tests provide path coverage of the model, and ensure that a complete test suite is automatically generated, using symbolic execution as a means for testing.

2.4 Related Work

In addition to the background work required for this thesis, we will also present related works, consisting of similar efforts that demonstrate the state of the art in incremental test case generation. We present several works, and discuss similarities of, and differences with, the work presented within this thesis.

Chittimalli and Harrold have presented their work on Recomputing Coverage Information to Assist Regression Testing [24], which shares motivational aspects with our work. Based on the realization that "Software systems continually evolve during development and maintenance.", for many of the same reasons cited in our work, they felt the need to pursue a method of reusing existing test cases to save in the expensive process of regression testing. By leveraging the existing test cases, they aimed to reduce the load on test generation, while still yielding the same results. The main area of importance presented in their work is that performing the analysis to determine which tests will give the same results when run on a new version of a model, will allow test case generation (and running of tests) to occur for only impacted areas of the program. In their studies, they found that the time to run the selected tests, plus the time to run their tool (ReCover) to generate and select tests, was consistently lower than the time required of rerunning all tests for a new version by a significant amount. These validation results were very promising to us, showing that there is indeed potential for gain in incremental test case generation.

Bates and Horwitz present work in incremental program testing using program dependence graphs [25]. As with our work, along with that of Chittimalli and Harold, motivation stems from the ability to reuse existing test suites to test a modified version of program. Based on some modification, they aimed to create "an adequate test suite for the modified program that reuses as many files from the old test suite as possible." In addition to simply creating the adequate test suite, they also were able to identify the impact on the remaining tests, to determine whether or not they need to be rerun on the modified program. The similarity of goals was shown in their aim to reduce the time required to perform the tests on a modified program, as well as to "avoid unproductive testing", which parallels our motivation of reducing the redundancy that this creates. While their implementation and the testing environment vary greatly from ours, the validation is in the aims presented; the desire to reduce redundancy through leveraging existing test cases is an important area of focus.

Uzuncaova et al. present a technique for Incremental Test Case Generation for Software Product Lines [23]. As with the other presented works, the main idea behind the implementation is similar, while the actual implementation differs greatly. The aim to reuse existing test cases exists, but more emphasis is placed on evolving the actual test cases to become representative of the new version, as opposed to simply selecting the test cases from the existing suite that still hold and filling gaps by generating new tests; the focus is on refinement using constraint solving. Their initial evaluation of the tool shows significant gains in the incremental method over the conventional method, suggesting again that the use of incremental test case generation as a test generation technique is beneficial, and the further exploration is warranted.

Mirzaaghaei et al. [26] present similar work in their study of test case adaption to support evolution. In much the same vein as our work and the others discussed here, the motivation stems from the realization that as a program evolves, some test cases often become invalidated. The aim is to reuse knowledge from existing test cases to adapt the applicable test cases and to generate new ones when necessary. While the approach focuses on source code, as opposed to models as in our case, the goals and general process is largely the same. They follow a similar process in first determining the difference between the versions, and then adapting test cases using the appropriate evolution algorithms. An analysis of their implementation shows that they were able to adapt test cases with high accuracy and performance that is comparable to other test generation approaches.

The main takeaway from reviewing these related works is that the motivation for incremental test case generation remains consistent across domains, and the need for this technique is not isolated to our work. The realization that the development process is iterative, and often leads to new versions of software rapidly being produced, has led to the understanding that reuse of existing test suites is imperative. While there are a number of different approaches presented, the notion of selecting the appropriate test cases to carry forward to a new version of a test suite is seen across the implementations; these reused tests are also often examined to determine whether or not they need to be run on the new version. Also, the concept of generating new tests for only the impacted functionality is presented throughout these works; by determining the impact of the change, tests can be generated solely for the affected areas of the software. Lastly, the largely positive results obtained through evaluation of the presented implementations provides excellent validation that this technique is a worthwhile effort.

Our approach was developed to provide a solution to these common goals, specific to UML-RT and the environment in which we are working. The shared motivation and general technique allow the presentation of a worthwhile implementation to a confirmed problem in the test case generation area.

2.5 Summary

UML-RT [7] is a real-time profile of UML [6] used mainly for real-time embedded systems, due to its reactive implementation. It varies slightly from UML in the fact that it contains the notion of capsules, which contain state machines detailing behaviour, and structure diagrams detailing composition and structure. The increasing popularity of MDD and greater need for real-time software has led to UML-RT becoming an emerging design method, and while there are a number of implementations of it, we have chosen IBM's Rational Software Architect RealTime Edition [16] as the environment for our exploration and implementation.

Symbolic Execution provides developers with a symbolic representation of execution, in the form of a Symbolic Execution Tree (SET) which can be used for a number of forms of analysis, test case generation being among them, and is the primary reason for of our use of symbolic execution. SETs provide an excellent starting point for test case generation for UML-RT models, as well as a built in completeness criteria in creating tests for path coverage.

There are a number of methods for test case generation available, and some of these were explored, including the method that was chosen for our work, which is the use of the SETs to generate a series of inputs for the model, in order to achieve path coverage testing.

Throughout the chapter, a number of works were explored in relation to the work presented, in hopes of providing additional background and support of the relevance of the work completed.

Chapter 3. Overview

3.1 Model Selection and Discovery Phase

3.1.1 Input Test Model Selection and Types of Changes

When evolving a model, there are (in the simplest form) three types of changes: adding new elements, modifying existing elements in some manner, or removing existing elements from the model. Thus, these are the three top-level categories of model evolution we will work with. For each of these categories of evolutions, we can manipulate the following model elements: states, transitions, parameters to inputs, entry and action code, and attributes. By applying each of the evolution categories (addition, modification, or deletion) to each of those elements of UML-RT models, we cover a number of possible evolution steps for any given model. This is how the evolution steps were chosen.

To create the original models, we needed to create a number of models that would produce different sizes of SETs and test suites, and were of differing complexity. The necessity of these variations is that all of these factors contribute to the resulting test suites. Five models were created for this purpose and are described in detail in Appendix A – Example Models.

3.1.2 Initial Test Case Generation and Examination

Initial test case generation is necessary for two reasons: first the original test suite is used as a base for the evolution to the new test suite, and secondly we aimed to analyze the evolved test cases to determine the effects of evolution on test cases in order to implement these updates when possible. The initial test case generation is performed using the original SET (as defined in the previous chapter), by traversing through the tree in a depth-first manner. Beginning at the root, we traverse each possible path of the tree, moving from symbolic state to symbolic state through a series of transitions that are based on the transitions of the UML-RT state machine. As each transition is taken, the required inputs and events, and the constraints on the parameters are recorded in sequence for later use. At the end of each path, when a leaf node is reached, this constitutes a full execution, and the path constraints on any input variables are solved using the Choco Constraint Solver [15]. The solved values are substituted in where the symbolic variables once were, and the ordered events are presented as a full test case. This is repeated for each path, and therefore the generated test suite will provide path coverage for all executions.

The goal of the exploration of test cases and SETs for the evolved models is to collect knowledge about how test suites and SETs are impacted for certain model evolution steps. Beginning with an original model and the set of evolved models for it, we symbolically executed each model, and then proceeded to generate the test cases. The generated test cases were then compared with the original to determine the effects of evolution. These effects were noted, including any interesting patterns and anomalies.

The differencing of the SETs also proved to be useful, as it allowed us to determine the highest point in the tree (conversely the earliest point in execution) that the evolved model differs, along all paths. This in turn allows us to find the tests which are changed and those which are not affected. Given that execution will occur in exactly the same manner for the remaining test cases, they do not need to be run again, and this can be a significant gain in the efficiency of testing.

We were able to leverage this knowledge throughout development of the tool to assist in determining the appropriate action to take, and in writing the update functions for the evolution classifications. It is this initial generation and examination that provided us with the benchmarks that were needed for validation as well.

3.2 Differencing UML-RT Models and Determining Appropriate Actions

The first step necessary to incrementally generate tests for an evolved model is to determine exactly how that model has evolved. By differencing the models, using RSA-RTE's [16] built in comparison tool, we are able to sort through the list of differences, and determine the appropriate action (if any) that will cause the existing SET and Test Suite to be representative of the newly evolved model.

The built in differencing tool reports a very comprehensive list of differences, so the first thing that was done was to filter out all differences that are irrelevant to the updating of the SET and the test suite. From that point, the differences are sorted based on their evolution classification, and are then operated on accordingly.

3.3 Updating the Existing SET and Test Suite

Based on the types of differences found by the differencing tool, there are a number of options that can be taken to make the required updates to the SET and Test Suite of the original model. The best case scenario would be to directly update the resources based solely on the information obtained from differencing the models, but this is also fairly uncommon. Another option is to perform partial symbolic execution, in order to obtain subtrees of the entire SET and combine these as necessary with the existing tree. Using this new tree, we can obtain the corresponding test suite by simply regenerating the test suite using the test case generation algorithm discussed earlier. The third possible option is to symbolically execute the new model, which is something we aim to avoid at all times, but in some cases may be the only option. Just as with the partial subtree generation, once the new tree is obtained the test suite is generated by traversing the tree and creating the series of inputs for each test.

A goal of this work is to reduce the need for symbolic execution, as it is the most costly operation in the test case generation; the third option is certainly one that we wish to avoid at all costs, and the second option should be avoided whenever possible.

3.3.1 Direct Updates

When possible, it is our goal to be able to directly update the existing SET and Test Suite to be reflective of the updated model, as this removes any necessity to symbolically execute the new model. Without having to symbolically execute the model, there is a substantial reduction in the amount of time required for test case generation.

This type of direct update is possible in a select few of our evolution classifications, but remains consistent across example models, and has proven to result in a significant gain.

3.3.2 Partial Symbolic Execution

When possible, we aim to obtain the SET through partial subtree generation, which gives us an end result of an updated SET that is representative of the newly evolved model. The main use of this SET is for the generation of the new test suite, but it also can be used for analysis as well, in that differencing two SETs will show you paths along which changes occur, and can highlight important information about execution.

The analysis phase is entirely optional, and is performed to provide additional information to the user. Given a SET for a model, and one for another model with one evolution step performed to it, we have two trees with what is assumed to be a large amount of similarity and a few select differences. Using a tree differencing algorithm where we compare the two SETs using a breadth-first method, we are able to determine the highest level differences along all paths in the tree. If along a given path the leaves of both SETs are reached, there is no difference in that execution path. By the nature of execution trees, when a difference is found it is possible, and probable, that the change will propagate through the rest of that path. When a difference is found, it is recorded and the search continues with the rest of the tree until all paths have been fully explored.

Once the new test suite is generated from the new SET, the test suite is compared to the original test suite to determine which tests in the new test suite are new or modified. The importance of this is that the tests that remain do not need to be rerun, as the execution paths they follow have not changed. This allows the user to only run the necessary test cases, essentially cutting back on the testing phase.

3.3.3 Full Symbolic Execution

This option is chosen when partial symbolic execution is impossible. Because of this the full SET must be generated from scratch, and it is then used for the same purposes described for the SETs generated via partial symbolic execution.

3.4 Tool Development

The last thing that will be discussed in this chapter is the final tool development, and tying together of all of the research. Chapter 7 will outline this process, and highlight some of the features of the tool as a complete implementation.

3.4.1 Application of Classifications

As one of the main goals of the research was to classify the model evolution steps, it was imperative that the final implementation consider this. While the tool itself does not provide notification to the user of these classifications, the ability to use them in the incremental generation of test cases, as part of the decision making process applies them in a practical setting.

3.4.2 Regeneration as Necessary

This can be seen as an extension of the application of the classifications; however it merits a further explanation on its own. One of the main goals of any analysis software should be to determine when your tool is going to be effective, and when it is not. Our tool does exactly that, and in the instances where the use of any of our incremental techniques would not provide any improvement over the existing test case generation method or simply cannot create the same results, the tool reverts to using the original technique.

3.4.3 SETs and Tree Differencing for Analysis

Another benefit of drawing all of the research into a single tool is that the internal algorithms that are used in the test case generation process are useful in other ways. One prevalent example of this is the differencing of Symbolic Execution Trees, in that the information obtained from this process can be quite useful in a number of ways, including but not limited to: test case selection, and evolution impact analysis.

3.4.4 Notification of Updates

One of the major features of IncreTesCaGen is its reporting function. In addition to incrementally generating the new test suite, a number of differencing results are displayed to the user that can be leveraged for a number of different improvements to the development cycle. One example that we will discuss further is the altered test case notification.

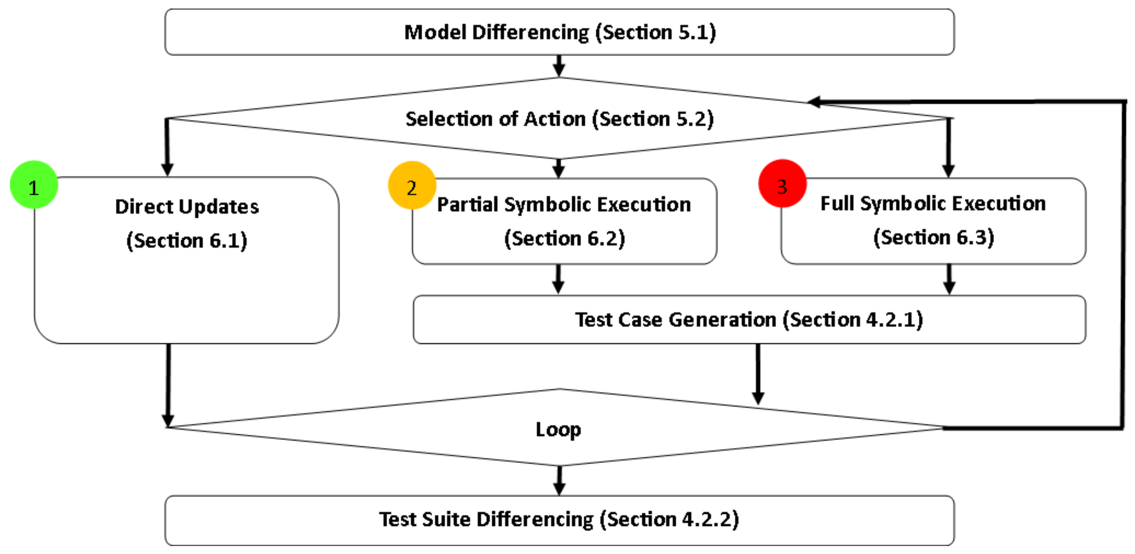
3.5 Summary

Although there are a number of different steps and options to the process of incrementally generating test cases, the process can be simplified and streamlined with ease, and that is what this section intended to do.

The first step is to determine the difference between the original and evolved model; using the differencing tool we are able to determine the type of difference (add, modify, delete) and the model element that has changed. Given this difference, we select the appropriate action (direct updates to the SET and test suite, partial symbolic execution of the model, or full symbolic execution of the model).

Once the new test suite is obtained, a differencing of test suites is performed to report to the user which tests have been affected by the update, thus informing them which tests need to be run. The full new SET and test suite are saved for future use and updates.

The full process is outlined in Figure 3.5.1.


Model Differencing (Section 5.1)
Selection of Action (Section 5.2)
1
Direct Updates (Section 6.1)
2
Partial Symbolic Execution (Section 6.2)
3
Full Symbolic Execution (Section 6.3)
Test Case Generation (Section 4.2.1)
Loop
Test Suite Differencing (Section 4.2.2)

Title	Desciption	Inputs	Outputs
Model Differencing	Use RSA-RTE's internal differencing tool to identify differences between original and evolved model.	2 RSA-RTE Models	List of Differences between Models
Selection of Action	Based on difference, select best option.	Current Difference	Decision of Action
Direct Updates	Directly update the SET and test suite using information available from differencing models	Original SET and Tests	Updated SET and Tests
Partial Symbolic Execution	When not possible to directly update, second preference is partial symbolic execution.	Original SET and New Model	New SET
Full Symbolic Execution	As a last resort, we may need to fully symbolically execute the entire model.	New Model	New SET
Test Case Generation	A full test suite is generated for the newly obtained SET.	New SET	New Generated Test Suite
Loop	Continue looping if more differences remain.	List Of Differences	Decision of whether or not to loop again
Test Suite Differencing	Given the new test suite, we difference it with the original test suite and determine which of the test cases have changed or been added or removed, so we know which tests need to be run, and which do not.	Original and New Test Suites	List of added and/or removed tests

Figure 3.5.1 Describing Approach to Incremental Test Case Generation

Chapter 4. Model Selection and Discovery Phase

4.1 Input Test Model Selection and Types of Changes

4.1.1 Standard Evolutions Steps

In order to proceed with incremental test case generation, the first step was to define "evolving a model". The following are the list of questions that we needed to answer: What defines an evolution? How atomic would these changes be? What changes will we examine?

Initially, we decided to look specifically at what types of actions can occur to a model. In the simplest form, there are essentially only three things a developer can do to a model: we can add a new element to the model, we can modify an existing element in some manner, or we can delete an existing element from the model. These three high level actions became the basis for our classification of standard evolution steps, but more was needed to constitute a classification.

When dealing with UML-RT models, the items that can be added, modified, or deleted are the model elements, which are: states, transitions, entry code, transition action code, triggers to transitions, and parameters on signals. There are other elements to UML-RT models, but these elements were the ones chosen for this research, as they are predominant elements.

Once we defined the three actions and a number of elements they can be performed on, we then created a number of plausible evolution steps that were used throughout the research for testing, implementation and validation. Some were purposefully excluded due to their similarities to existing examples, and in one case (addition of action code) two examples were used as adding code to send an output signal, and adding code that alters the value of an attribute were thought to be significantly distinct.

Table 1 defines the 14 model evolutions that were used throughout the research.

Evolution Step	Description
1. Add State	Add a new state to the state machine. This alone has no effect on execution, so in order to proceed with testing, a single transition to this state is added as well, meaning that a new reachable state has been added.
2. Modify State	An existing state is modified in some manner. For these particular examples, this refers to a renaming of the state.
3. Delete State	A state from within the state machine is deleted. Any transitions to or from this state are automatically deleted along with it by the tool.
4. Add Transition	A transition is added between two existing states within the state machine.
5. Modify Transition	An existing transition is modified in some manner. There are two types of modifications that were used to demonstrate this evolution: a change of the transition's target state, and a change of the trigger for that transition.
6. Delete Transition	A transition is deleted from the state machine.
7. Add Entry Code to a State	Entry code is added to an existing state.
8. Modify Entry Code on a State	Existing entry code is modified in some manner.
9. Delete Entry Code from a State	Existing entry code is removed from a state entirely.
10. Add Action Code on a Transition (send output)	Action code is added to a transition that will send output on a protocol.
11. Add Action Code on a Transition (modify val)	Action code is added to a transition that will update the value of one of the attributes of the capsule.
12. Add a Parameter to a Signal	A parameter is added to an incoming signal of a protocol. The signal initially was a void signal, but an interger parameter is added.
13. Delete a Parameter from a Signal	The parameter is removed from an incoming signal of a protocol. The signal initially had an integer parameter, but is now a void signal.
14. Modify Initial Value of an Attribute	The initial value of one of the capsule attributes is changed.

Table 1 Model Evolution Steps Used

4.1.2 Original Models Used

The next step was to build a suite of models on which to apply these evolutions. In considering this, we factored in things such as model size, model complexity, and resulting test suite size. We also had to determine how many different models we were going to work with.

In the end we ended up choosing five models that are explained in full detail in Appendix A – Example Models. The models varied in the criteria mentioned above, and were chosen as a reasonable representation of generic models. Some were arbitrarily simple, while others were artificially complex, and one was aimed at simulating a real life scenario.

It is clear that these five models are not a full representation of all models in existence, but we feel that they provide a sampling of the types and complexities of a number of models that we would see in use. It is clear that more examples would always be better, and real life examples would be excellent, but for an initial evaluation of our work, we feel that they are sufficient.

4.1.3 Applying Evolutions

With a set of initial models and the evolutions steps decided, the next step was to apply these 14 evolutions to each of the five models, thus creating 70 newly evolved models that would be used for examination in order to determine effects of evolution, and later for validation of the tool.

The specific evolutions that were performed on each model can be found in Appendix A – Example Models. It outlines for each model and evolution step which update was chosen; additional information includes the element which was updates, along with the new values for the changed attribute.

4.2 Initial Test Case Generation and Examination

Although the goal of the tool and the research as a whole is to avoid the full symbolic execution and test case generation, it was necessary to perform for exploratory purposes and to determine how to implement the updates. For each of the evolved models we used the symbolic execution engine to obtain the SET, and then used the test case generation engine to produce a complete test suite. For each evolved SET and test suite, they were compared to the originals, to determine the exact changes for a given evolution step. The hope was to discover patterns across the five different models for each evolution step meaning that it affected symbolic execution and testing consistently.

4.2.1 Initial Test Case Generation

In this section we will discuss the methodology that was used for generating the initial tests, and how we arrived at the decisions that were made.

There are a number of different definitions of a test case, or test suite, and the first thing we needed to decide was which of these fit our criteria, and what would constitute a test case/test suite in the context of incrementally generating test cases for UML-RT models. Recall that when performing background research we had previously determined that path coverage was the best option for a completeness criteria for our test suites due to the ease of determining this from a SET. From there we had to determine the method of testing and also consider options for implementation, allowing for the incremental generation to take place and make sense within the context.

Since the models are representations of real-time reactive systems that function based on reacting to events such as timers and input signals, a test case for a capsule model would need to be an ordered series of events. This would include parameters on the input signals and the input values if applicable, as well as timers timing out at given times, which are special input signals from an internal framework port.

However, in addition to simply providing outputs and observing the behaviour (execution passes through the states as expected), the added ability of UML-RT to send outputs over ports allowed us to also produce expected output for each input, such that each output (or set of outputs) is associated with the input which caused it to be sent.

Given all of these decisions, we finally settled on the definition of test cases and test suites that we would be using for the research. We would be performing input testing, with output validation, using SET path coverage as a completeness criterion.

In order to implement our “test suite” the first thing we did was create an Ecore model [18] of a test suite that would be used for our experiments. The added bonus of creating an Ecore model was the automatic generation of the classes, and a visual editor.

The model shown in Figure 4.2.1 depicts the class diagram for a test suite. A *TestSuite* is made up of some number of *TestCases*, each designated by a *CaseNumber*. A *TestCase* is some number of ordered *Inputs*, which can be either a *signal* over a *port*, a *timer*, or the default input which is automatically assigned to the transition from the initial state; these special inputs are assigned as the *signal*, with a null value for the *port*. Each *Input* may also have a *parameter*, which is an *Expression*, meaning it can either be a *Constant* or a *SymbolicVariable*. The final test cases only make use of constants as parameters, as symbolic variables are meaningless in test cases. The SymbolicVariables are used throughout the generation process, and stored in the *unsolvedParam* field for later use. The same is also true for values of *Outputs*. During the generation process, the *unsolvedParm* values are solved using the constraint solver Choco [15], and the actual values are given as parameters.

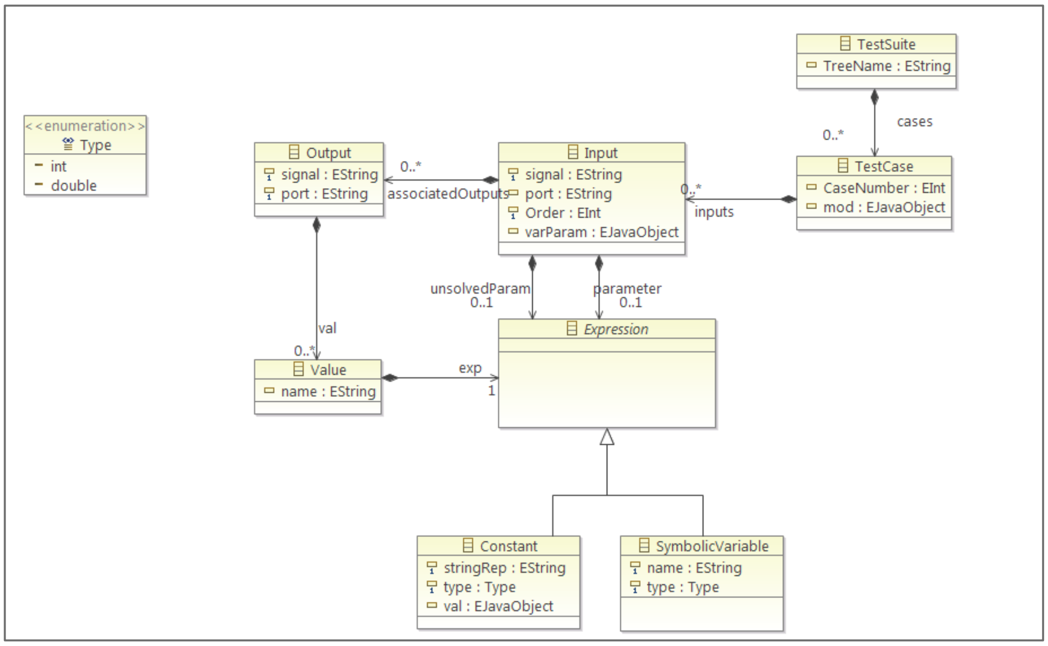
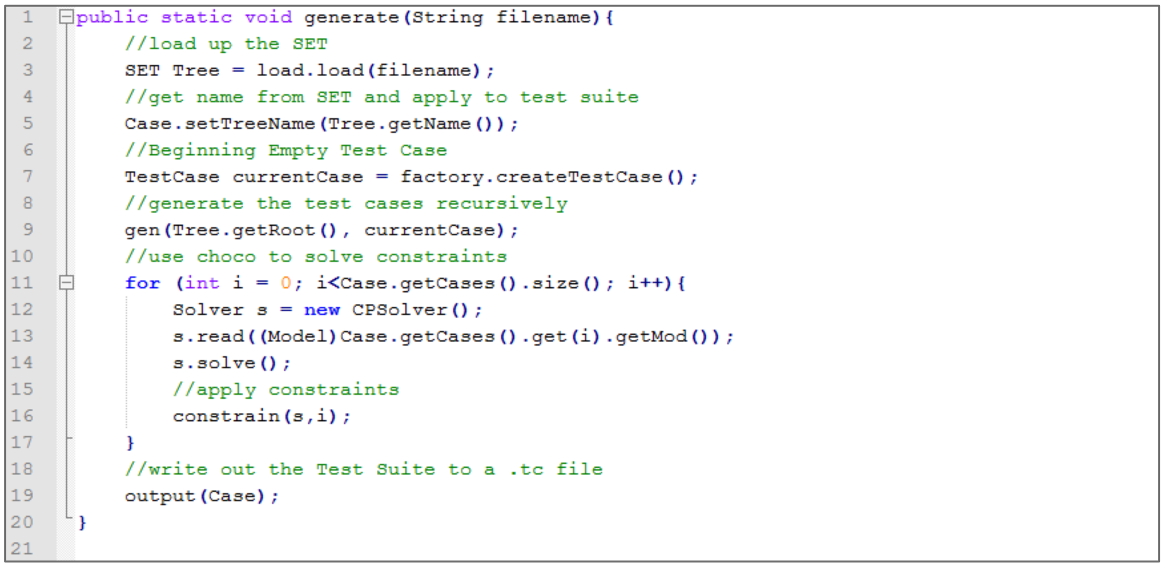


Figure 4.2.1 Ecore Model of a Test Suite

In the next section, we will briefly explore our implementation of test case generation, and provide an example of a test suite.

4.2.2 Implementation

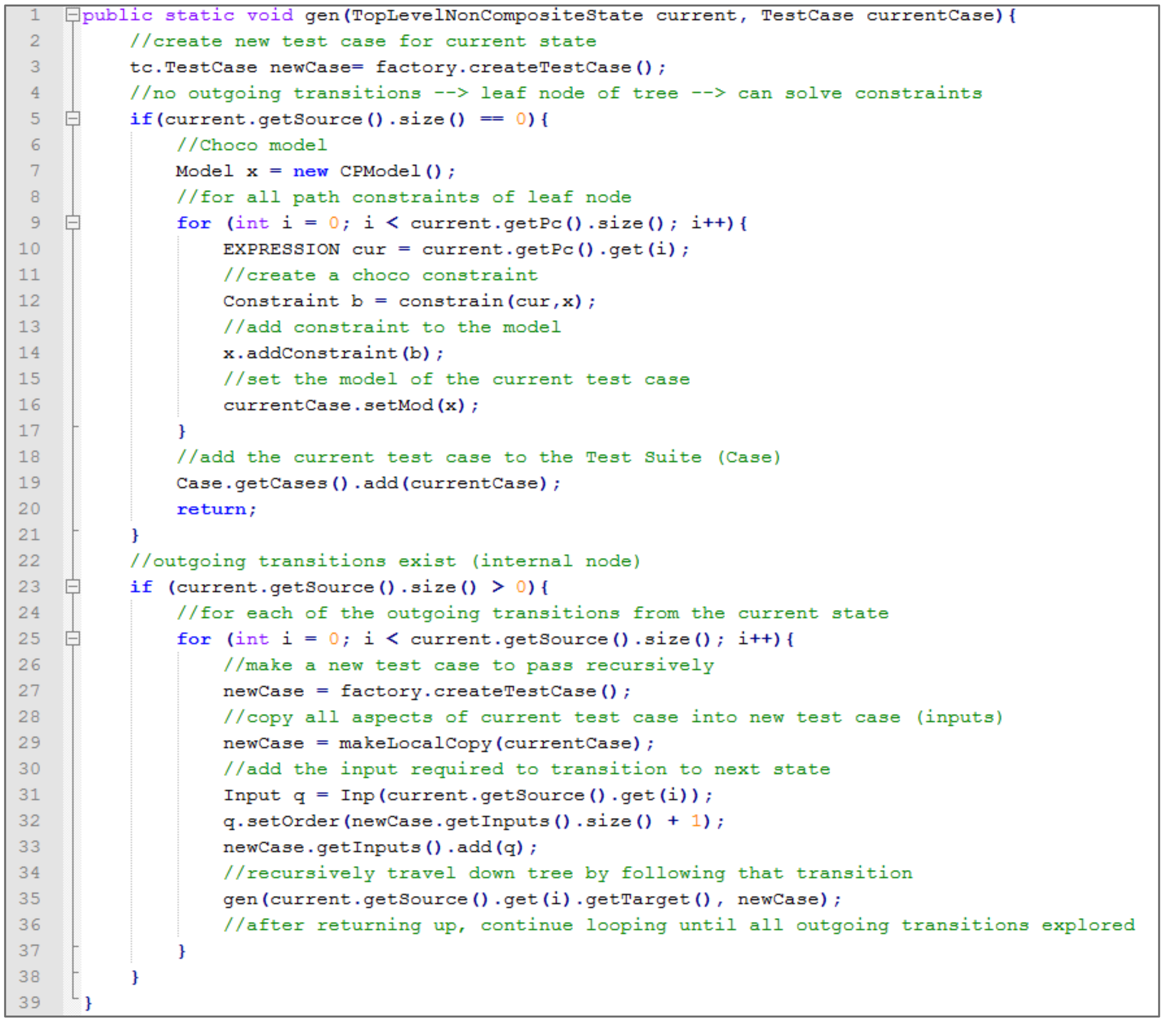
Figure 4.2.2 shows the *generate* method, which is the main method called to generate a test suite. The algorithm begins with the initialization of variables, and instantiating the test suite itself (lines 2 – 7). At this point, the gen method is called, which will be described in detail later in this section, but it is used to generate the tests for the provided tree (line 9). After the test suite is generated, for each of the test cases (line 11), the Choco constraint solver [15] is used to solve the constraints on symbolic variables (lines 12-14), and these variables are then applied to the test case (line 16), meaning the solved values are substituted in for the symbolic variables. Finally the completed test suite is output to a file (line 19).



```
1   public static void generate(String filename){
2       //load up the SET
3       SET Tree = load.load(filename);
4       //get name from SET and apply to test suite
5       Case.setTreeName(Tree.getName());
6       //Beginning Empty Test Case
7       TestCase currentCase = factory.createTestCase();
8       //generate the test cases recursively
9       gen(Tree.getRoot(), currentCase);
10      //use choco to solve constraints
11      for (int i = 0; i<Case.getCases().size(); i++){
12          Solver s = new CPSolver();
13          s.read((Model)Case.getCases().get(i).getMod());
14          s.solve();
15          //apply constraints
16          constrain(s,i);
17      }
18      //write out the Test Suite to a .tc file
19      output(Case);
20  }
21
```

Figure 4.2.2 generate Algorithm to generate tests and solve constraints

Figure 4.2.3 shows the *gen* method, which is a recursive method that is used to generate the test suite for a given SET. The parameters to this method are *current*, which is a state within the SET (initially the root state), and *currentCase*, which is the test case up until the current state, along the path taken to get there (line 1). The first thing the recursive algorithm does is create a new test case to represent any new tests generated by branching in the SET (line 3). From here there are two things that can happen: there are no outgoing transitions in the current state (lines 5-21), or there are outgoing transitions to explore (lines 23-39).



```
1  public static void gen(TopLevelNonCompositeState current, TestCase currentCase){
2      //create new test case for current state
3      tc.TestCase newCase= factory.createTestCase();
4      //no outgoing transitions --> leaf node of tree --> can solve constraints
5      if(current.getSource().size() == 0){
6          //Choco model
7          Model x = new CPModel();
8          //for all path constraints of leaf node
9          for (int i = 0; i < current.getPc().size(); i++){
10             EXPRESSION cur = current.getPc().get(i);
11             //create a choco constraint
12             Constraint b = constrain(cur,x);
13             //add constraint to the model
14             x.addConstraint(b);
15             //set the model of the current test case
16             currentCase.setMod(x);
17         }
18         //add the current test case to the Test Suite (Case)
19         Case.getCases().add(currentCase);
20         return;
21     }
22     //outgoing transitions exist (internal node)
23     if (current.getSource().size() > 0){
24         //for each of the outgoing transitions from the current state
25         for (int i = 0; i < current.getSource().size(); i++){
26             //make a new test case to pass recursively
27             newCase = factory.createTestCase();
28             //copy all aspects of current test case into new test case (inputs)
29             newCase = makeLocalCopy(currentCase);
30             //add the input required to transition to next state
31             Input q = Inp(current.getSource().get(i));
32             q.setOrder(newCase.getInputs().size() + 1);
33             newCase.getInputs().add(q);
34             //recursively travel down tree by following that transition
35             gen(current.getSource().get(i).getTarget(), newCase);
36             //after returning up, continue looping until all outgoing transitions explored
37         }
38     }
39 }
```

Figure 4.2.3 gen Algorithm for recursively generating test cases

If there are no further outgoing transitions from the current state, this means that execution has reached a leaf of the tree (the end of an execution path). The first thing done here is the creation of a Choco Model object, which is a container for the path constraints (line 7). For each constraint in the leaf state (line 9), we add that constraint to the model to be solved later (lines 10-16), and finally, we add the current test case to the test suite (line 19).

If there are outgoing transitions, this means that there are more inputs to add to the path, and more to explore of the SET. For each of the outgoing transitions from the current state (line 25), we create a local copy of the test case up until the current point in exploration (lines 27-29), and then we add the input signal for the outgoing transition to the local copy of the test case (lines 31-33), and then recursively generate the remainder of that path by calling the gen method (line35).

This recursive method allows for a depth-first search of the entire SET, and results in complete path coverage, and a test case for each of the possible paths of execution.

4.2.3 Example Test Suite

To better understand our implementation of test suites, an example will help provide clarity. Using the Model 2 from Appendix A – Example Models as our working example, the first artifact would be the SET for that model. It is a fairly simple model, and the SET is also quite simple.

Figure 4.2.4 shows the generated SET for the state machine. The number of paths (four) is important to note, because this will also be the total number of test cases generated by the tool. Before jumping directly to the generated test suite, let us explore the SET further. The first thing to note is that from the *initial* state there is only one possible transition, meaning that every test case will begin with the *default* transition as an input.

From the *waiting* state, which is reached by taking the *default* transition, there are four possible options, meaning there are four branches that need to be explored. Within the algorithm, this would translate to four local copies being made of the single transition test case, and recursively exploring these four options. One of the options leads to a shorter path than the others, containing only one more additional transition/input, making its total length two inputs. The others all contain three inputs.

Taking all of these observations into account, it becomes clear exactly what the test suite should look like for this SET: four test cases, three of length three, and one of length two, and all tests will begin with the default transition. In larger more complex SETs it becomes less clear what the resulting test suite will look like, but the theory behind it remains the same.

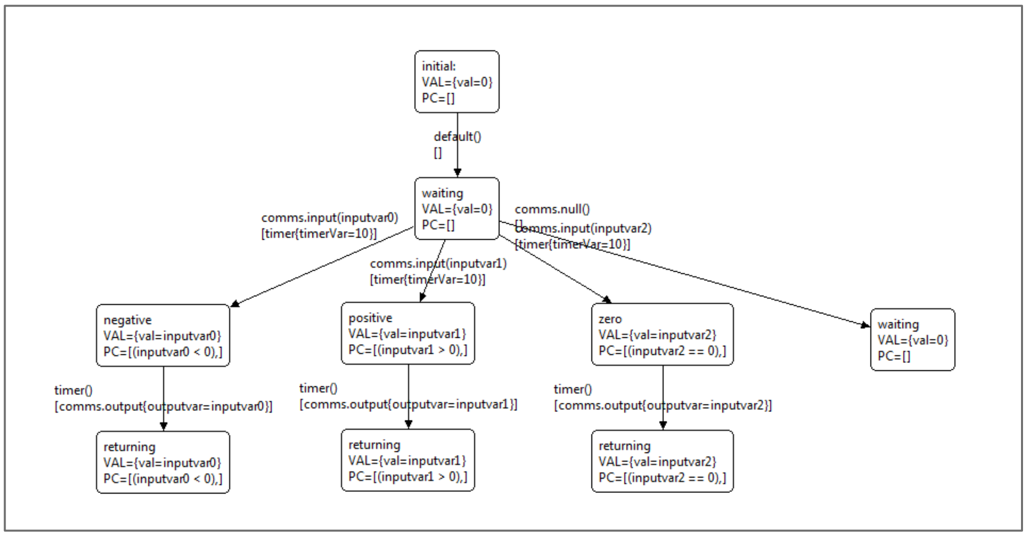


Figure 4.2.4 SET for Model 2

Figure 4.2.5 shows the test suite for Model 2, generated by our test case generation algorithm in textual form. Looking closely at it, we can confirm that the observations made from the SET have held true. The test suite contains exactly four test cases, three of which contain three inputs, and one that contains two inputs. Additionally they all begin with the default signal as input.

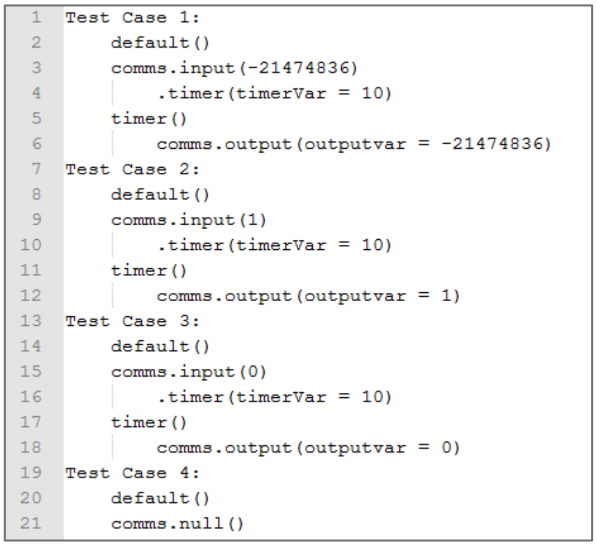

```
Test Case 1:
    default()
    comms.input(-21474836)
        .timer(timerVar = 10)
    timer()
        comms.output(outputvar = -21474836)
Test Case 2:
    default()
    comms.input(1)
        .timer(timerVar = 10)
    timer()
        comms.output(outputvar = 1)
Test Case 3:
    default()
    comms.input(0)
        .timer(timerVar = 10)
    timer()
        comms.output(outputvar = 0)
Test Case 4:
    default()
    comms.null()
```

Figure 4.2.5 Generated Test Suite for Model 2

4.2.4 Test Case Examination

With an implementation for test case generation complete, it was time to begin exploring the effects of model evolution on test cases, and the best way to accomplish this was to compare the evolved test suites to the original test suite, noting differences and comparing these for consistencies across models.

This was done by implementing a straightforward test suite comparison algorithm that would compare each of the original test cases to each of the modified test cases until either (a) an exact match is found, or (b) all are explored and no match is found. If no match is found, the test case is noted as not being in the new test suite, and therefore removed. The exact same process is repeated for each test case in the modified test suite, to determine which test cases are not in the original test suite, and therefore added. One thing to note is that this does not highlight test cases that are modified; in such a case, it is listed as removed from the original and added to the new test suite, so this comparison requires further examination.

Much like the tree differencing this is used as an exploratory step to help determine classification of the effects of model evolution steps on test suites, but is also used in the final tool as a method of reporting to the user which tests have changes and/or remained the same. This reporting step is discussed further in Section 4.2.5.

During this exploratory phase of the research the differencing is used to highlight effects on test suites, noting whether the evolution step affected all test cases, some particular subset of the test cases, or none of the tests. These observations were recorded and further explored to implement the updating function of the tool.

It was during this phase that the results began to show what we were hoping to see, and there were patterns that began to emerge across models. These findings are discussed in Section 4.3.

4.2.5 Test Case Reuse

As stated, in addition to being a useful comparison tool to examine effects of model evolution, the differencing of test suites is useful for the reuse of existing test cases. Once a new test suite is obtained, by comparing the new test suite to the original, the user is presented with a list of test cases that do not appear in the original test suite.

This list identifies the reduced number of tests that are required to be run on the new model, assuming the original test suite was run on the original model. The reasoning for this is that if the original test suite was successfully run on the original model, the evolution step performed has not affected the execution paths tested by the remaining test cases, and to test them again would be redundant. By running only the newly generated tests, we are still ensuring the correctness of the model, but are not running any unnecessary tests. In large industrial models, this could save hours of testing during each iteration of development, as only a small fraction of tests may need to be run with each evolution as opposed to an entire test suite.

4.3 Findings

In this section, we present the findings of the exploration of the effects of model evolution on both SETs and Test Suites, and conclude by presenting three possible classifications for the model evolution steps presented in Section 4.1.1, and which classification each of them falls into.

We began by looking at the impacts of the changes on the SETs and Test Suites with a goal of grouping the evolution steps based on the type of operation required to make the update to the original SET and Test Suite. The most interesting of our findings was that the deletion of states and/or transitions has a predictable effect on the SET and Test Suite of the evolved model; subtrees of the SET are completely removed at the point where the removed state or transition is found. This finding meant that the updating of the SETs and Test Suites for these types of evolution steps (numbers 3 and 6) would not require any new symbolic execution, and simply only removing portions of the SET and parts of the existing test suite.

There are two other evolution steps of interest that stand out from the others due to the fact that they also do not require symbolic execution to make updates to the SET or the test suite. The first of these is number 2 (the modification of a state by renaming it). Since the name of the state has no effect on the execution, the test suite does not change in any way, and the only change needed to the SET is to replace any occurrence of the original state name with the new state name. The second such instance is number 12 (adding of a parameter to an input signal). This action alone does not affect the execution, as the parameter has not yet been referenced in any guards or action code; because of this, any time it is referenced in test cases, its value is insignificant, and can thus be assigned the default value of zero each time. This is done to both the SET and the Test Suite, globally replacing any occurrence of the input signal, which originally had no parameter with the same input signal with parameter of an unconstrained symbolic variable, replaced in the test cases with the default value of 0.

This classification aims to determine, when producing the updated SET and Test Suite, whether it requires a direct update to the existing artifacts, partial subtree generation (generating some subtree to replace existing subtree), or regeneration of the whole SET. Based on these three tool-specific classifications, Table 2 outlines which model evolution step falls into which category. It is worth noting the fact that only one evolution step fell into the full regeneration classification, this is significant, meaning that it is the only time that a full regeneration is consistently required. These classifications were used as a basis for the decision making shown in Figure 3.5.1.

Evolution Step	Classification of Update Operation
1. Add State	Partial Subtree
2. Modify State	Direct Update
3. Delete State	Direct Update
4. Add Transition	Partial Subtree
5. Modify Transition	Partial Subtree
6. Delete Transition	Direct Update
7. Add Entry Code to a State	Partial Subtree
8. Modify Entry Code on a State	Partial Subtree
9. Remove Entry Code from a State	Partial Subtree
10. Add Action Code on a Transition (send output)	Partial Subtree
11. Add Action Code on a Transition (modify val)	Partial Subtree
12. Add a Parameter to a Signal	Direct Update
13. Remove a Parameter from a Signal	Partial Subtree
14. Change Initial Value of an Attribute	Full Regeneration

Table 2 Classification of Update Operations

4.4 Summary

This chapter aimed to highlight the initial discovery phase for the research. It began by presenting the evolution steps chosen for exploration, and the five example models that we would be applying these evolution steps to, and performing the analysis and examination on.

From there, we presented our implementation of test cases, by providing our definition of a test suite in the context of the research, and showing the Ecore class diagram representing a Test Suite. The algorithm used to generate test suites from a given SET is presented and explained in detail.

The next portion of the chapter dealt with comparing test suites, and how this was used both in discovery, and in the final implementation as a method of test case reuse. The fact that comparing test cases can determine which tests need to be run provides a huge gain in industrial models, preventing the redundant rerunning of unaffected test cases.

Finally this chapter presents our findings of comparing the original model, SET, and Test Suite to their evolved counterparts. We present the three classifications of the impact of model evolution effects, specifically the updates required of the SETs and Test Suites: the ability to directly update SETs and test suites (Direct Update), the need to perform partial symbolic execution to generate subtrees of the entire tree (Partial Subtree), and the need to regenerate the entire SET (Full Regeneration).

Chapter 5. Differencing UML-RT Models and Determining Appropriate Actions

This chapter will outline the first steps taken in incrementally generating the SET and Test Cases for an evolved model: comparing the two models, and determining the appropriate action to take. The first thing we will explore is comparing the two models, looking at the tool we use and the way results are presented to the user. From there we will discuss the examination of these results, and the decision engine that is used to determine which action to use.

5.1 Differencing Tool

The first step in incrementally generating test cases is to determine which evolution step (if any) has been performed.

While exploring the development environment for UML-RT, Rational Software Architect Real-Time Edition (RSA-RTE) [16], we discovered that it has its own implementation of model differencing that is built upon the EMF Compare [19] framework, with additions for the real-time content. In order to reduce the workload of developing our tool we explored this further in order to leverage the existing tool's differencing power. We were pleased to discover that the default difference tool was able to provide us with all of the necessary details to determine exactly how a model version differs from a previous version.

The compare function takes in two models, and returns a list of differences that can be iterated over, and each difference explored. Since the differencing tool can provide some differences that do not affect the SET or test suite in any way, such as visual changes to the model (line sizes, shape dimensions, etc.), these are first pruned from the list, and the remaining differences are explored further.

Each difference in the list contains a difference type, which can either be addition, change or deletion, which aligns perfectly with our evolution steps. The second thing that each difference contains is a pointer to the actual item of difference in each model; this allows us to determine what has changed. Lastly, each return difference contains a reference to the original artifact of change and the newly updated artifact, either of which may be null in the case of addition or deletion.

5.2 Selection of Action Based on Differences

With a pruned list of differences provided by the built-in RSA-RTE differencing tool, the next step is to determine which action(s) should be performed to obtain the updated SET and test suite. For this we return to the two sets of classifications obtained in Chapter 4 and identify which type of update we will need to perform.

If the first difference returned from the tool outlines that there is an addition of a new transition, then the method *addTransition* is called, which implements a version of the partial symbolic execution to generate a new SET and generates a test suite. The method is provided with the reference to the added transition for exploratory purposes; the same is true for any of the other differences.

There are also a number of differences where there is a direct overlap in what must occur. The way that these overlaps were discovered was by examining the classifications and finding areas of commonality between like elements. For example, the evolutions for adding entry code, modifying entry code, and deleting entry code are all of the same classification (partial subtree), and all pertain to the same element; this commonality means that the implementation for all three of these is exactly the same, and only one method was required. The same is true for the two additions of action code, since they both require the same updating of the SET and test suite. On the other hand, there are actually two different implementations for the modify transition evolution step, as there are two different types of differences that can be reported by a modification to a transition: the target and the trigger may be changed, and due to this there are two methods that needed to be created to perform these two types of updates.

The implementations of the updates themselves will be discussed in further detail in Chapter 6; however it is important to note that whenever possible, when creating the decision engine, we grouped similar differences together, so as to reduce the amount of work later on in the process.

Another possibility during this phase of decision is that the differencing tool may have returned multiple differences for a pair of models; one such example of this would be the removal of a state from a state machine. When a state is deleted, the transitions to and from this state are also automatically removed; therefore a state with one incoming and one outgoing transition that is deleted would result in three differences being reported by the tool. This is handled by iterating over the list of differences individually and performing the required updates for each change. There is also some built in optimization to avoid redundant work; take for example the state deletion discussed previously. Each time a deletion of a transition is reported, the source and target are checked to determine if either of them is equal to a deleted state; if so, the difference is skipped, as it will be dealt with by the update function for the deleted state. The iteration over this list of differences allows for multiple smaller changes to occur, but they are still handled as individual atomic changes.

5.3 Summary

This chapter highlighted the first step in incrementally generating SETs and test cases, which is to determine the differences between the original model and the evolved model, to determine the appropriate action wherever possible.

This is done by using the built-in comparison tool within RSA-RTE, which returns a list of differences that can be iterated over, and operated on as necessary.

Also as part of this phase we used the classifications provided in Chapter 4, to attempt to group like differences together into the same update functions, to reduce the workload later on in development. One such example of this is the addition, modification, or deletion of state entry code, all being part of the same classification in both sets, and requiring the same updates to the SET and test suite.

Chapter 6. Updating the Existing SET and Test Suite

Given the classifications defined and observed in Chapter 4 and the differences discovered in Chapter 5, we were finally able to focus on the main area of the work, updating the SET and the test suite from the original model so that they are reflective of the new evolved model.

As previously presented, there are three possible ways of updating the SET and test suite for an evolved model: directly updating if possible, performing partial symbolic execution on the necessary portions of the model and generating tests from the resulting combined tree, or a full regeneration of the SET and generating tests from the new tree.

The following sections will discuss these options in more detail, while also providing listings of the methods that are used to implement the updates.

6.1 Direct Updates

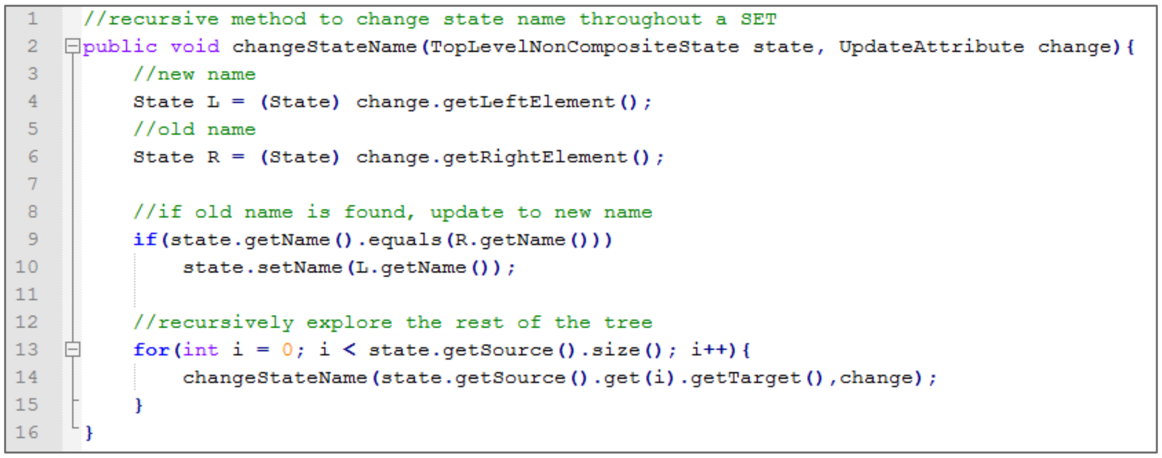
This option is by far the most favourable as it removes the necessity for any symbolic execution, which is the most expensive portion of test case generation.

From the classifications discussed earlier, it was clear that there are exactly four updates that fit into this category. These four evolution steps are: modification of a state (particularly state name), deleting a state, deleting a transition, and adding a parameter to an input signal. Each of these requires a very different approach, and each will be discussed within this section.

6.1.1 Modify a State

A modification to a state simply means changing the name of a state to reflect some update to the model. This is, by itself, not a large change, but is indeed a common change as a model evolves; the name of a state is likely to change to reflect functionality.

Figure 6.1.1 shows the *changeStateName* algorithm that is used to make the necessary updates to the SET. Since there is no change at all to the test suite, this is completely ignored. The algorithm recursively explores the SET in a depth-first method, checking each state name, and if it finds the name that has been changed, it replaces it with the new one, and then continues exploring.



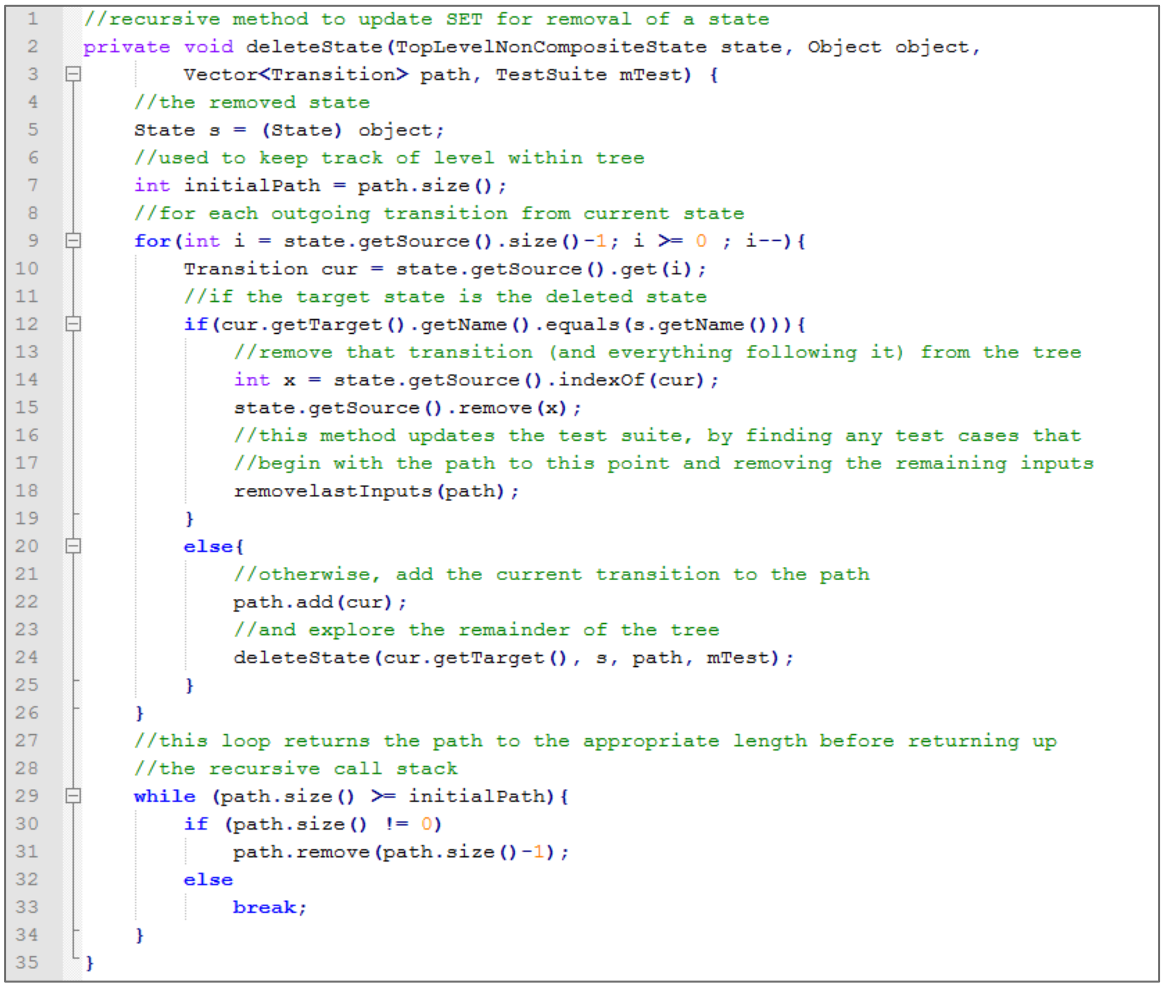
```java
//recursive method to change state name throughout a SET
public void changeStateName(TopLevelNonCompositeState state, UpdateAttribute change){
    //new name
    State L = (State) change.getLeftElement();
    //old name
    State R = (State) change.getRightElement();

    //if old name is found, update to new name
    if(state.getName().equals(R.getName()))
        state.setName(L.getName());

    //recursively explore the rest of the tree
    for(int i = 0; i < state.getSource().size(); i++){
        changeStateName(state.getSource().get(i).getTarget(),change);
    }
}
```

Figure 6.1.1 changeStateName Algorithm

6.1.2 Delete a State

Deleting a state from a model means that execution may no longer reach that state, and therefore any test cases that result in passing through that state are no longer valid. To update a SET to reflect this, we explore the tree until a symbolic state in which there is an enabled transition to a symbolic state matching the deleted state is found, and remove the transition to that symbolic state from the SET, and therefore everything below it in the tree. This process is repeated for the entire tree and the resulting SET is representative of the evolved model. To make the necessary updates to the test suite, when a difference is found, and subtrees are deleted, a call is made, passing the path to the symbolic state to a helper method that removes the remaining inputs from any test case that begins with inputs matching the path to that point. This however, typically results in duplicate test cases in the test suite, but this is handled by the calling function, which goes through the test suite and removes any duplicates.

Figure 6.1.2 shows this process in more detail.



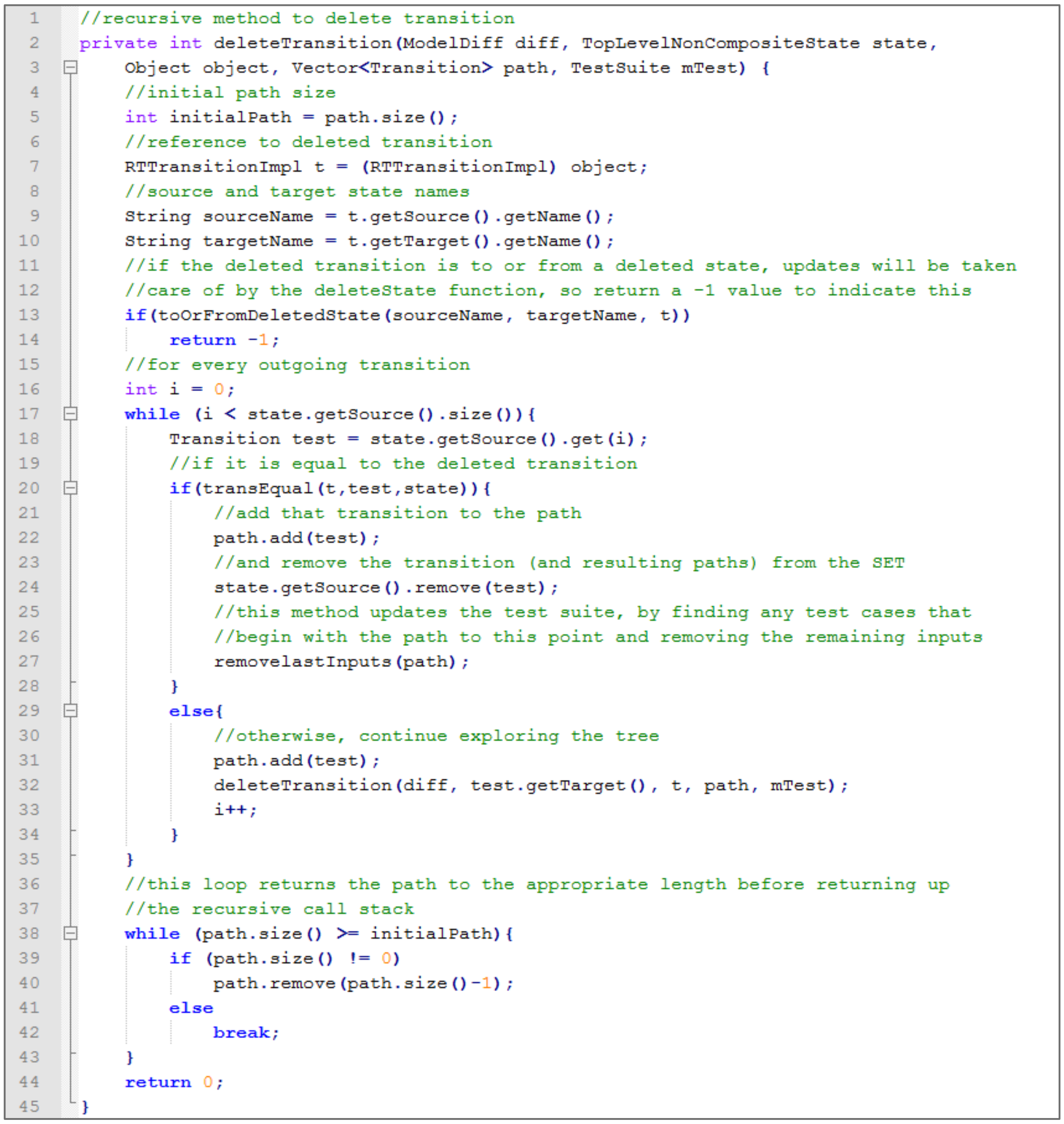
```java
//recursive method to update SET for removal of a state
private void deleteState(TopLevelNonCompositeState state, Object object,
        Vector<Transition> path, TestSuite mTest) {
    //the removed state
    State s = (State) object;
    //used to keep track of level within tree
    int initialPath = path.size();
    //for each outgoing transition from current state
    for(int i = state.getSource().size()-1; i >= 0 ; i--){
        Transition cur = state.getSource().get(i);
        //if the target state is the deleted state
        if(cur.getTarget().getName().equals(s.getName())){
            //remove that transition (and everything following it) from the tree
            int x = state.getSource().indexOf(cur);
            state.getSource().remove(x);
            //this method updates the test suite, by finding any test cases that
            //begin with the path to this point and removing the remaining inputs
            removelastInputs(path);
        }
        else{
            //otherwise, add the current transition to the path
            path.add(cur);
            //and explore the remainder of the tree
            deleteState(cur.getTarget(), s, path, mTest);
        }
    }
    //this loop returns the path to the appropriate length before returning up
    //the recursive call stack
    while (path.size() >= initialPath){
        if (path.size() != 0)
            path.remove(path.size()-1);
        else
            break;
    }
}
```

Figure 6.1.2 deleteState Algorithm

6.1.3 Deleting a Transition

This update is very similar to deleting a state, in that a deleted transition can no longer be taken, so therefore any time that exact transition is taken within the original SET, it is then removed in the updated SET. The process for finding it is the same as any other, in that we recursively traverse the SET until the difference is found.

Figure 6.1.3 depicts the algorithm used to update the SET and Test Cases when a transition has been deleted. Much like the state deletion, the difference is found, and any subtrees that take the deleted transition are deleted, with test cases being shortened as necessary, and the calling method removes any duplicates.



```java
//recursive method to delete transition
private int deleteTransition(ModelDiff diff, TopLevelNonCompositeState state,
    Object object, Vector<Transition> path, TestSuite mTest) {
    //initial path size
    int initialPath = path.size();
    //reference to deleted transition
    RTTransitionImpl t = (RTTransitionImpl) object;
    //source and target state names
    String sourceName = t.getSource().getName();
    String targetName = t.getTarget().getName();
    //if the deleted transition is to or from a deleted state, updates will be taken
    //care of by the deleteState function, so return a -1 value to indicate this
    if(toOrFromDeletedState(sourceName, targetName, t))
        return -1;
    //for every outgoing transition
    int i = 0;
    while (i < state.getSource().size()){
        Transition test = state.getSource().get(i);
        //if it is equal to the deleted transition
        if(transEqual(t,test,state)){
            //add that transition to the path
            path.add(test);
            //and remove the transition (and resulting paths) from the SET
            state.getSource().remove(test);
            //this method updates the test suite, by finding any test cases that
            //begin with the path to this point and removing the remaining inputs
            removelastInputs(path);
        }
        else{
            //otherwise, continue exploring the tree
            path.add(test);
            deleteTransition(diff, test.getTarget(), t, path, mTest);
            i++;
        }
    }
    //this loop returns the path to the appropriate length before returning up
    //the recursive call stack
    while (path.size() >= initialPath){
        if (path.size() != 0)
            path.remove(path.size()-1);
        else
            break;
    }
    return 0;
}
```

Figure 6.1.3 deleteTransition Algorithm

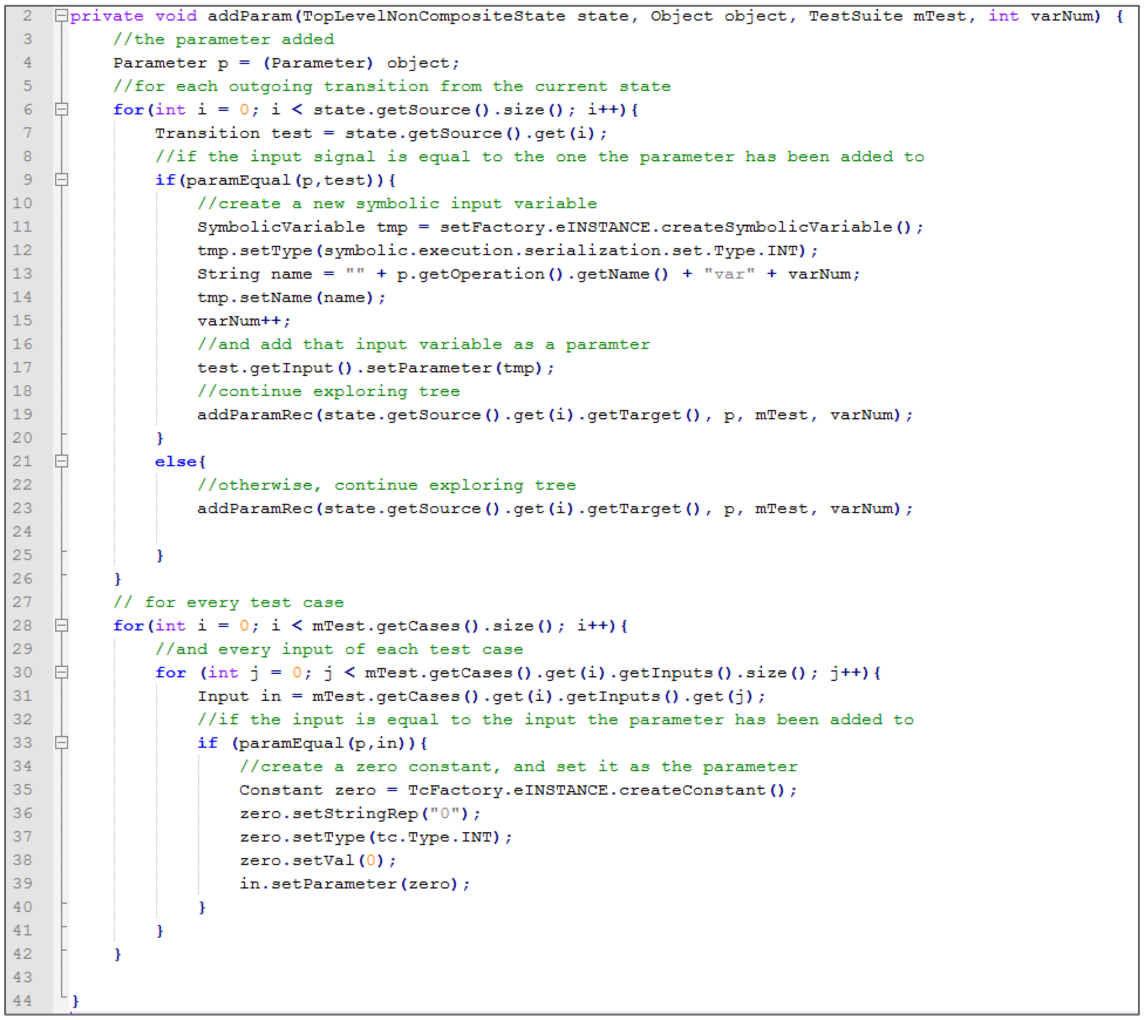
6.1.4 Adding a Parameter

The fourth and final evolution step that allows for direct updates to the SET and test suite is the addition of a parameter on an input signal. This is a uniform change, meaning that it happens for every occurrence of the signal that it has been added to, regardless of context. This means that updates can be applied directly to the SET and test suite without needing to perform symbolic execution to determine the effects of the update.

Figure 6.1.4 outlines the algorithm used to update the SET and test suite to reflect this update. The first thing to note is that the method shown is the top level algorithm, which makes calls to a version slightly different, containing only the first for loop, as the second for loop only needs to be run once, at the end of exploration, to make the updates to the test suite.

The way the update works is much like any other, in that it begins by traversing through the tree, looking for the element that has been changed (in this case a transition where the input has had a parameter added), and when it is found, it simply adds a symbolic variable as the parameter.

The last thing that is done, as previously mentioned, is that once the whole tree has been explored and symbolic parameters have been added, updates to the test suite need to occur. The way this is done is to inspect each individual input for each test case, and if an input is triggered by the signal with the added parameter, a parameter of zero is added. The reason we can be certain that the parameter is 0 is that there would be no constraints placed on the parameter at this point, as the only change made was introducing it. A symbolic variable with no constraints is always solved to be the constant zero (0).



```java
private void addParam(TopLevelNonCompositeState state, Object object, TestSuite mTest, int varNum) {
    //the parameter added
    Parameter p = (Parameter) object;
    //for each outgoing transition from the current state
    for(int i = 0; i < state.getSource().size(); i++){
        Transition test = state.getSource().get(i);
        //if the input signal is equal to the one the parameter has been added to
        if(paramEqual(p,test)){
            //create a new symbolic input variable
            SymbolicVariable tmp = setFactory.eINSTANCE.createSymbolicVariable();
            tmp.setType(symbolic.execution.serialization.set.Type.INT);
            String name = "" + p.getOperation().getName() + "var" + varNum;
            tmp.setName(name);
            varNum++;
            //and add that input variable as a paramter
            test.getInput().setParameter(tmp);
            //continue exploring tree
            addParamRec(state.getSource().get(i).getTarget(), p, mTest, varNum);
        }
        else{
            //otherwise, continue exploring tree
            addParamRec(state.getSource().get(i).getTarget(), p, mTest, varNum);

        }
    }
    // for every test case
    for(int i = 0; i < mTest.getCases().size(); i++){
        //and every input of each test case
        for (int j = 0; j < mTest.getCases().get(i).getInputs().size(); j++){
            Input in = mTest.getCases().get(i).getInputs().get(j);
            //if the input is equal to the input the parameter has been added to
            if (paramEqual(p,in)){
                //create a zero constant, and set it as the parameter
                Constant zero = TcFactory.eINSTANCE.createConstant();
                zero.setStringRep("0");
                zero.setType(tc.Type.INT);
                zero.setVal(0);
                in.setParameter(zero);
            }
        }
    }

}
```

Figure 6.1.4 addParameter Algorithm

6.2 Partial Symbolic Execution

Partial Symbolic Execution refers to the generation of some part of a whole SET; a subtree from within an entire tree. The subtree represents all possible execution paths from a provided symbolic state; much like full symbolic execution provides all execution paths from the initial state. The way this is accomplished is that a depth-first search is performed on the original SET, looking for all symbolic states impacted by the difference reported by the tool, and these symbolic states (or the ones preceding them in some cases) are used as the roots for partial symbolic execution. The newly obtained subtrees are then used to replace the initial subtrees, and the result is a SET reflective of the new model.

The theory behind this implementation is that some number of partial symbolic executions will ultimately require less time and resources than a full symbolic execution, excepting of course corner cases that will result in the partial symbolic execution taking place from the initial state.

To argue that the use of partial symbolic execution is beneficial to incremental test case generation, we provide a simple example.

Figure 6.2.1 shows a simple SET that will be used as our original model SET. Execution may proceed from symbolic state A to either B or C, both of which proceed to either D or E, and finally from either D or E, execution returns to A, which in this case creates a subsumption relationship with the A at the root of the tree.

To demonstrate the effectiveness of partial symbolic execution, the evolution step that is performed is the addition of a transition to a newly added state F within the state machine that is reachable from state E only. Within the SET, both occurrences of symbolic state E are able to transition to a symbolic state F. It is evident that the newly generated SET would look similar to the original, with a two-way branching from symbolic state E, to both A and F, and this full generation would require generating a tree containing a total of 13 symbolic states. However, if we search the original tree for the known difference (addition of new transition to new state), we find that State E is the point where execution diverges, by passing symbolic state E as the root for partial symbolic execution, it will generate a subtree containing only three symbolic states; this being repeated a second time, when the second occurrence of symbolic state E is found. These two generated subtrees, totaling six symbolic states, are shown in Figure 6.2.2.

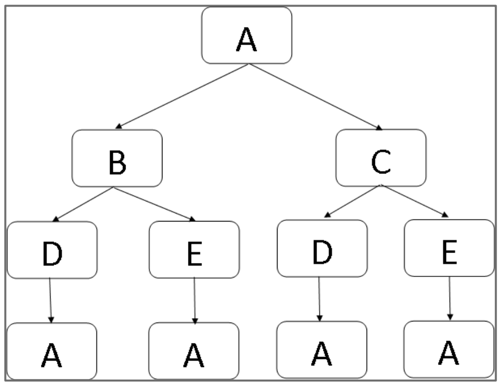


Figure 6.2.1 Simple SET

By replacing the original instances of symbolic state E with the newly generated ones, we can obtain the full SET for the newly evolved model at a reduced amount of workload and time. The final SET can be seen in Figure 6.2.3.

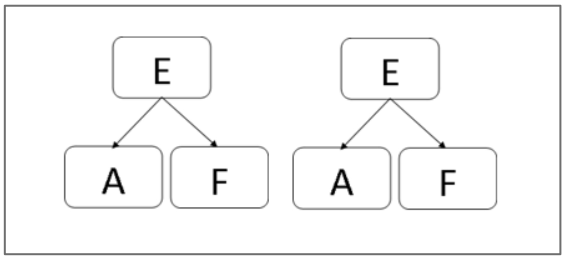


Figure 6.2.2 Two Generated Subtrees

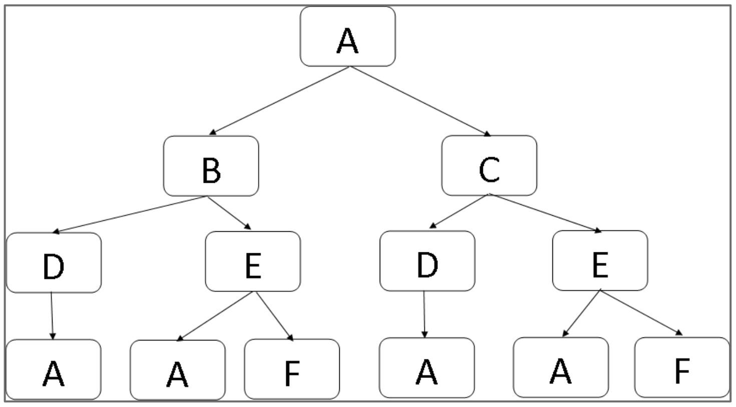


Figure 6.2.3 Evolved SET Including Subtrees

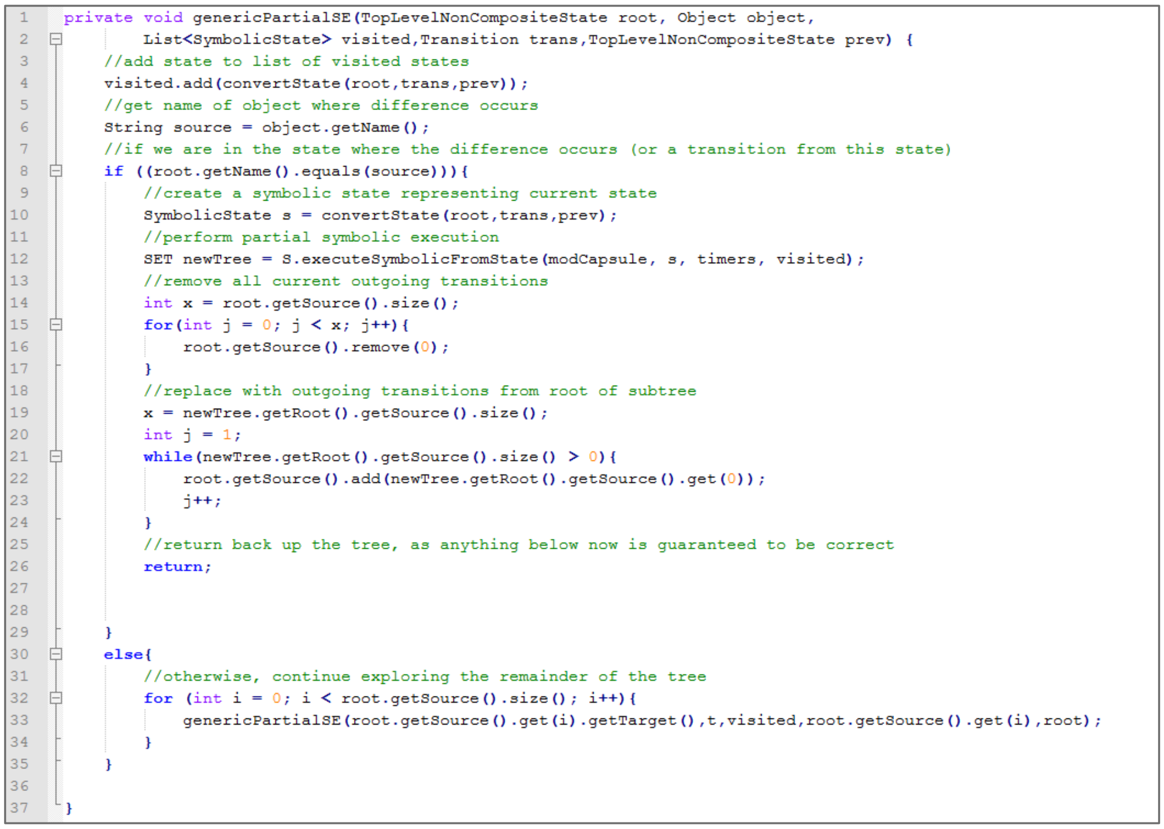
One realization that we faced when choosing to pursue this method of obtaining the new SET was that the additional overhead associated with differencing the original models (model comparison and iterating over the list of differences), and locating those differences within the original SET would be problematic and the overall savings may not have been worth-while when compared to simply regenerating the full SET. The thing to keep in mind is that this example shows that only six states needed to be generated as opposed to 13, but in larger models, the savings can become even more substantial.

6.2.1 Implementation

To implement partial symbolic execution where necessary, a number of methods were created for the evolution steps that are able to leverage partial symbolic execution. These different methods are largely similar, and only exist as different methods to allow for future optimizations to specific cases. In this section we will present a generalized algorithm used in a number of cases, and specify which methods implement this algorithm.

The first thing to note is that there are two steps to these types of updates: obtaining the new SET, and generating the new Test Suite. Obtaining the SET will be discussed in detail, but the second step of generating the test suite requires no further explanation, as we make use of the test generation algorithm presented in Section 4.2.1 to fully generate the new test suite, as this is still the most cost-effective way of obtaining the updated test suite.

Figure 6.2.4 shows the generalized recursive algorithm that is used for partial symbolic execution as an implementation for generating a SET. The parameters to it (and all implementations of it) are the current state being explored, the actual object of difference from the model, the list of states that have already been visited, the incoming transition to the current state, and the previous state in the execution path.



```
1   private void genericPartialSE(TopLevelNonCompositeState root, Object object,
2           List<SymbolicState> visited,Transition trans,TopLevelNonCompositeState prev) {
3       //add state to list of visited states
4       visited.add(convertState(root,trans,prev));
5       //get name of object where difference occurs
6       String source = object.getName();
7       //if we are in the state where the difference occurs (or a transition from this state)
8       if ((root.getName().equals(source))){
9           //create a symbolic state representing current state
10          SymbolicState s = convertState(root,trans,prev);
11          //perform partial symbolic execution
12          SET newTree = S.executeSymbolicFromState(modCapsule, s, timers, visited);
13          //remove all current outgoing transitions
14          int x = root.getSource().size();
15          for(int j = 0; j < x; j++){
16              root.getSource().remove(0);
17          }
18          //replace with outgoing transitions from root of subtree
19          x = newTree.getRoot().getSource().size();
20          int j = 1;
21          while(newTree.getRoot().getSource().size() > 0){
22              root.getSource().add(newTree.getRoot().getSource().get(0));
23              j++;
24          }
25          //return back up the tree, as anything below now is guaranteed to be correct
26          return;
27
28
29      }
30      else{
31          //otherwise, continue exploring the remainder of the tree
32          for (int i = 0; i < root.getSource().size(); i++){
33              genericPartialSE(root.getSource().get(i).getTarget(),t,visited,root.getSource().get(i),root);
34          }
35      }
36
37  }
```

Figure 6.2.4 Generalized Algorithm used for Partial Symbolic Execution

The reason the list of visited states is necessary, is that it is passed to the partial symbolic execution method, and used to assist the subsumption checking; since the generation is not originally aware of visited states, by passing a list of already visited states to it, it has some sense of the exploration history, and more options for subsumption. For the same reason the previous state and incoming transitions are required of each method; they simply increase the likelihood of finding a matched state for subsumption.

To explore this further, we will step through the algorithm and explain the purpose and aim of each step. The first thing that is done is that the current state is added to the list of visited states (line 4), which is necessary for subsumption, as explained above. The next thing done is to figure out the name of the state where the difference occurs (line 6) and we compare that to the name of the current state (line 8). If the two names are equal, the affected state has been found, and modifications need to be made to the tree (lines 9-29) rooted at this state. The first thing that is done is that the current state is converted into a symbolic state (line 10) that is then passed as the root of the partial symbolic execution (line 12). All of the existing outgoing transitions are removed from the current state (lines 14-17) and replaced with the transitions from the root state of the resulting partial SET (lines 19-26). In the event that the name of the current state does not match, exploration continues for each of the outgoing transitions from the current state, further down the tree (lines 31-34).

As stated, the new SET is then used to generate the new test suite, and the results are presented to the user and saved to the file system.

Since the algorithm that was outlined was a generalization, it is important to explore the individual implementations of used within the tool. Table 3 provides a list of the six methods that use the algorithm outlined in Figure 6.2.4. Although there are a number of other evolution steps that use partial symbolic execution, there are some that are similar enough that the implementations are the same, and there was no need to duplicate the same code.

Implementations of Partial Symbolic Execution
addTransition
addTransitionBehaviour
addStateBehaviour
changeTransitionTrigger
changeTransitionTarget
deleteParam

Table 3 Methods Using Generalized Partial Symbolic Execution Algorithm

6.3 Full Symbolic Execution

The third and final (and least beneficial) option for updating the SET and test suite is to symbolically execute the entire model and use the SET to generate the test cases. As it is the goal to avoid symbolic execution whenever possible, this is of course the last resort. Our exploration findings only indicated one evolution step where this type of update is necessary every time: a change to the initial value of an attribute. This is a change where even though there may be no effect on execution, there is no way of determining this with certainty from the existing SET.

Given the overhead provided by comparing the original and evolved model, and then the cost of a full symbolic execution, there will be absolutely no possibility of our implementation outperforming the existing method, and this is a realization we were well aware of during exploration and implementation, however given that only one case of the 14 evolution steps requires a full symbolic execution every time, this was encouraging.

The last item of note is that it is possible for any evolution step implemented using partial symbolic execution to actually require a full symbolic execution, should the difference occur at the root state of the tree. This is indeed a possibility, but even in these cases it is still considered a partial symbolic execution.

The one method that implements full symbolic execution is the *changeInitialValue* method and it contains two calls: one to symbolically execute the model, and one to generate the test suite.

6.4 Summary

Given that there are three different methods of updating a SET and test suite to be reflective of an evolved model, and they vary in computational cost, it is always a benefit to determine the least cost option for any evolution step; we were able to do exactly that in our implementation. In any case where direct updates are possible, they are performed; when they are not possible a new SET is obtained, either by partial (preferred) or full symbolic execution, and the resulting SET is used to generate test cases.

Table 4 shows the 14 evolution steps, and the operation classifications from Chapter 4, as well as the method within the code that implements each update. Note that evolution step 1 does not have a method for adding a state as this has no effect directly, but the transition added does, and this calls addTransition. Additionally, evolution step 5 has two implementations for the two different ways a transition can be altered. Lastly you will note that a number of steps use the same methods having to do with behaviour/action code, and this is because they share the same implementation.

Evolution Step	Classification of Update Operation	Implemented Method
1. Add State	Partial Symbolic Execution	addTransition (nothing for state)
2. Modify State	DirectUpdate	changeStateName
3. Delete State	DirectUpdate	deleteState
4. Add Transition	Partial Symbolic Execution	addTransition
5. Modify Transition	Partial Symbolic Execution	changeTransitionTarger or changeTransitionTrigger
6. Delete Transition	DirectUpdate	deleteTransition
7. Add Entry Code to a State	Partial Symbolic Execution	addStateBehaviour
8. Modify Entry Code on a State	Partial Symbolic Execution	addStateBehaviour
9. Remove Entry Code from a State	Partial Symbolic Execution	addStateBehaviour
10. Add Action Code on a Transition (send output)	Partial Symbolic Execution	addTransitionBehaviour
11. Add Action Code on a Transition (modify val)	Partial Symbolic Execution	addTransitionBehaviour
12. Add a Parameter to a Signal	DirectUpdate	addParam
13. Remove a Parameter from a Signal	Partial Symbolic Execution	deleteParam
14. Change Initial Value of an Attribute	Full Symbolic Execution	changeInitialValue

Table 4 Implemented Methods for Classifications

A strong decision engine using the classifications discovered allows for the best possible option to be performed in each case.

Chapter 7. Tool Development

The chosen method of implementation for incremental test case generation was a plugin for IBM's Rational Software Architect RealTime Edition [16] called IncreTesCaGen. We took the numerous algorithms and implementations for things such as test case generation, SET and test suite differencing, and combined them with a slightly modified version of the existing plugin for symbolic execution and analysis of UML-RT Models (SAUML 0.0.1) [3], to create a single plugin capable of performing incremental test case generation for a UML-RT model, provided the original artifacts are available to the plugin.

The program flow of the entire plugin is shown in Figure 3.5.1, and is rather linear, with the exception of the decision on which method is chosen to implement the updating of the SET and Test Suite.

In addition to the full implementation, and its ability to provide the updated test suite to the user, the final tool possesses the capability to provide other useful information to the user, as well as serves as a representation of the achievement of our more abstract goal of classifying the effects of model evolution. Throughout the remainder of this chapter, these additional benefits will be explored in detail.

7.1 Application of Classifications

One of the main goals of this research from early on was to obtain an understanding of the effects of model evolution on execution and testing, and to classify these effects into categories of classifications. While the tool does not explicitly state to which class each update belongs to and how the updates were performed, the tool itself is an implementation of these classifications. We were able to understand how certain types of evolution affected the resulting artifacts, and were able to group like results together; this is demonstrated through using the plugin to advance tests cases along with the evolved model.

The classifications as they exist currently could be further refined, but in their current state are significantly reflective of the types of updates possible to a UML-RT model. The similarities that exist between the effects of model evolutions allowed for a very succinct classification that led to an implementation and realization of our original goal.

7.2 Regeneration as Necessary

Another benefit of the final implementation that comes from the classifications, and the decision on how to update the test suite and SET, is the ability to identify the cases where the original method is the best option, and to regenerate the SET, or some portion of it, as necessary.

It is impossible for a tool to ever do everything that a previous implementation can do (let alone be better at everything). Due to this it is important to be able to leverage existing techniques in times where it is beneficial to the user; when a difference is found where updates are not possible, and partial symbolic execution is not possible, the tool is able to access the original symbolic execution plugin [3] to perform the generation of the SET, and then continues by generating the test suite from this SET.

7.3 SETs and Tree Differencing for Analysis

This specific feature of the plugin is not something that is available by default, but given a few tweaks to the source code, it can become a very useful analysis tool for model evolution. The ability to compare the original SET and the new SET is something that can provide insightful information to the user about the effects of the evolution they have performed.

One of the uses for SET differencing in terms of analysis is locating divergence points within the SETs; by finding the highest point within the tree (earliest point in execution) where a difference occurs, we can determine which portions of execution remain unchanged, and which contain possible changes. Due to the nature of SETs, any sections of a SET after a difference point can be (and likely are) different, meaning that the point of divergence is useful for information such as fault finding or variable value assignments.

In addition to simply determining effects on execution, differencing SETs and determining which paths differ and which remain unchanged is very useful in terms of test case selection [14]. Oftentimes a test suite may be far too large to even run all of the tests, let alone generate them all, which is why it is imperative to select test cases based on what has changed from an original model. Additionally, other methods of test case selection based on input variables or tree sectioning are possible.

Here we will provide two examples of test case selection methods based on analysis of the newly obtained SET. The first being a subset selection, and the other is variable bounding.

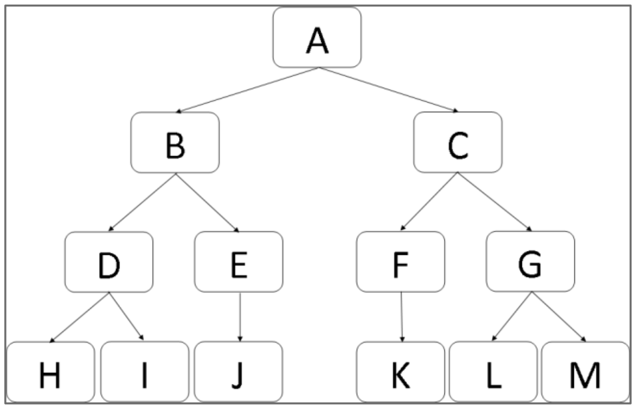


Figure 7.3.1 Sample SET for Test Selection

Figure 7.3.1 shows a simple SET that will be used for our examples. Given this SET, a full test suite would consist of six test cases, one to take you to each leaf node, and each test would pass through four states. The six cases for this example would be as follows:

1. A→B→D→H
2. A→B→D→I
3. A→B→E→J
4. A→C→F→K
5. A→C→G→L
6. A→C→G→M

The first method of test case selection is a systematic method, which deals with moving up some number of levels in the SET, and only directly following the path to that point, and then randomly selecting a given number (usually one) of test cases from the possible tests that begin with that path. For this example, if we were to specify moving up one level in the tree, and then selecting one test at random, we would only generate four tests, instead of six, meaning only two-thirds of the tests are generated and run; however they still provide an accurate representation of all tests. For example, 4 possible tests that would be generated using this example would be:

1. A→B→D→H
2. A→B→E→J
3. A→C→F→K
4. A→C→G→M

While it is true that this method does not end up with the entire test suite, it provides a close approximation, testing some subset of the execution paths (dependent on user), and still allowing for some downsizing of the test suite when necessary.

The second method of test case selection that is worth exploring is variable bounding. Since the SET branches based on both input signals, as well as conditionals, it is entirely possible to select test cases based on a certain path constraint. Say for example the branching from state A to state B or C is based on an if statement on a value, where the branch to B occurs if the value is less than zero, and the branch to C occurs if it is greater than or equal to zero. If we wanted to select test cases where this input variable could not take on a negative value, we could easily do this, and we would be left with only three test cases based on this selection. These three cases would be:

1. A→C→F→K
2. A→C→G→L
3. A→C→G→M

The last item of relevance in terms of using SETs and tree differencing for analysis is the ability to perform an impact analysis on changes. We developed a very simple approximation that can provide a numerical value approximating how much of a test suite is affected by a given set of changes. By differencing the trees and finding all of the highest difference points you can assign values and the total effect is reported.

The number of original test cases (known at the time of differencing), is assigned to the root node of the tree, as a change to that node in the tree would affect that number of tests. From there the tree is traversed and values are assigned to each node until a difference is found, and the current value of that node is added to the total difference amount; once the entire tree is searched, the total difference amount is divided by the total number of tests to determine the percentage of impact of the changes. The following equation shows how the value of impact of a state is calculated, with the base case being that the root node is equal to the number of test cases.

$$valueOfEachChildOfN = \left\lceil \frac{currentValOfN}{numOfChildrenOfN} \right\rceil$$

To better understand an example is provided in Figure 7.3.2, which outlines the impact values of each state instead of a state name. As you will note, this is indeed an approximation, and due to the inclusion of the ceiling function in the formula, it becomes an over approximation, as demonstrated by the two leaf nodes with a value of two, implying that a change at that point would affect two test cases, instead of the actual one that test it does affect.

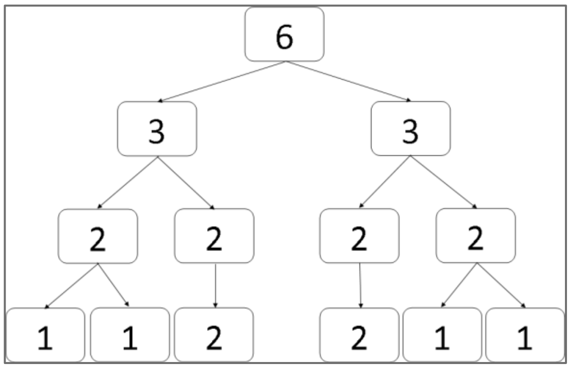


Figure 7.3.2 Sample SET for Impact Analysis

To show the total impact, another example is provided, using the same tree, but with points of differences noted by being highlighted in red. Figure 7.3.3 shows these differences, having weights of three and two, for a combined total of five. The impact analysis shows that the difference affects five of six test cases, or approximately 83% of the test cases and this information can be quite useful.

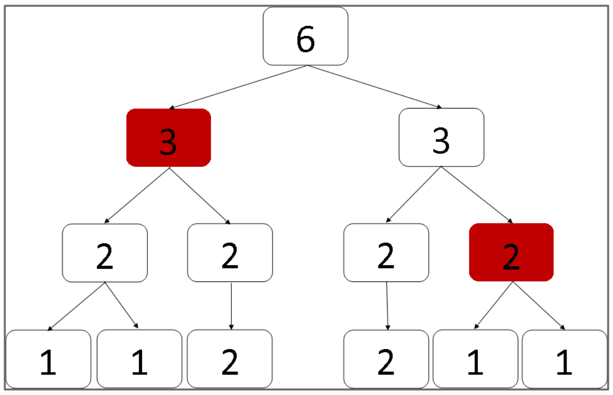


Figure 7.3.3 Sample SET Highlighting Differences

One important item to note is that this algorithm for weighting states is only an approximation, and it best suited for SETs that have a relatively equal distribution, which is certainly not always the case. It is entirely possible that this method can cause an abnormally high impact due to over approximation. Figure 7.3.4 demonstrates this problem where branching occurs once to a leaf node, assigning it a value of 50, but it actually only affects one test case. This is a drawback of the approximation, but if this is understood, then the approximation is still a very useful and valid tool.

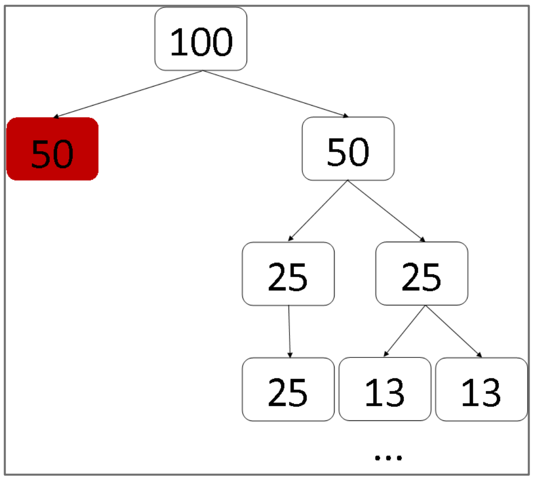


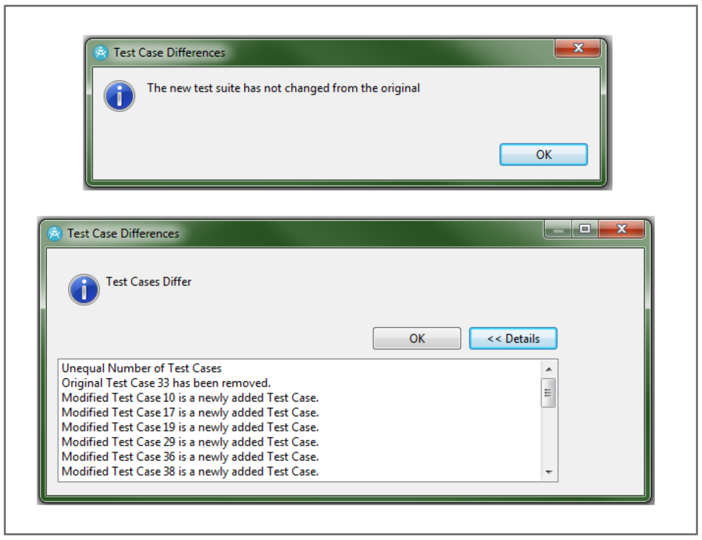
Figure 7.3.4 Demonstration of Misapproximation

7.4 Notification of Updates

The last benefit of the tool implementation is the ability to notify the user of test case updates with specific references to which tests were removed from the original test suite, which test cases were added into the new test suite, or if there was no change to the test suite.

The direct report to the user is useful for an extended methodology of test case selection, in that since it is known which cases are newly added, and therefore which tests have not changed, the user can choose to only run the new test cases, and skip over the unaltered ones. This allows the user to run only the essential tests representative of any changes in execution.

This information is communicated to the user via a dialog box in the tool that states that either the test cases have not changed or that there are differences and provides additional details. Figure 7.4.1 shows two examples of dialog boxes that the tool presents to the user. The first reports that the test suite has not changed from the original, and the second states the test cases differ, and then specifies which test(s) have been removed, and which have been added, so the user can choose to run only the newly added tests.



The new test suite has not changed from the original

Test Cases Differ

```
Unequal Number of Test Cases
Original Test Case 33 has been removed.
Modified Test Case 10 is a newly added Test Case.
Modified Test Case 17 is a newly added Test Case.
Modified Test Case 19 is a newly added Test Case.
Modified Test Case 29 is a newly added Test Case.
Modified Test Case 36 is a newly added Test Case.
Modified Test Case 38 is a newly added Test Case.
```

Figure 7.4.1 Example Dialog Boxes

7.5 Summary

After completing the research into the effects of model evolution on testing, and implementing incremental test case generation, it was imperative to bring all of these features together into one tool that allows the user to not only obtain the updated artifacts for their model, but to also analyze the differences, and be able to determine with more detail exactly how their evolution step has affected the model's SET and test suite. It is due to this requirement that the final tool IncreTesCaGen, a plugin for RSA-RTE [16], included some of these features by default, and the option to implement the others.

The plugin boasts a final reporting mechanism so that developers are made aware of exactly which tests have changed and can decide which tests to run based on that.

Additionally the analysis features of the plugin allow for further investigation in areas such as test case selection through SET examination, by choosing a method of systematically selecting tests.

Another analysis feature of the tool is the change impact analysis that provides an approximation of the overall effect of a change on the test suite by assigning values to each state in a SET and totaling all of the states where a difference occurs, and determining what percentage of the whole test suite is affected.

The last purpose the tool serves, in addition to the obvious implementation of incremental test case generation, is that it is a concrete representation of the classification of the effects of model evolution steps on SETs and Test Suites that were presented in Chapter 4.

Overall, the tool development allowed us to present the work in one compact implementation with the ability for further analysis should the user wish to explore the evolution further.

Chapter 8. Validation

8.1 Procedure and Results

8.1.1 Correctness Validation Procedure

The first step in validating our implementation was to ensure correctness of the SETs and Test Suites generated by the tool. This was done by directly comparing them to the SETs and Test suites that are generated by full symbolic execution, and performing the test case generation algorithm on that SET. Having the initial models to compare to allowed a simple differencing to determine if the output produced IncreTesCaGen was correct.

To do this each of the five original models found in Appendix A – Example Models were run through this process, and each of their 14 evolutions as well, meaning that a total of 75 SETs and test suites of varying size and complexity (including originals) were generated as comparison for the output of our implementation of incremental test case generation.

The next step was to run each of the 70 evolved models through IncreTesCaGen and to compare the results with the baseline outputs. Having done this, approximately 85% of the outputs were identical to those of the baseline; the remaining outputs were minor exceptions. These exceptions were explored further and it was determined that the only differences were to some of the cases where partial symbolic execution was used and subsumption between two states was not detected. These resulting test cases are not incorrect, and do cover all of the paths in the baseline test suites, but may contain a few additional redundant inputs at the end of a test case. These are an acceptable addition as they maintain the same testing, plus a slight bit of redundancy.

Based on these comparisons, it is safe to conclude that on a correctness criterion, our implementation was able to correctly produce the same results as our baseline.

8.1.2 Performance Validation Procedure

The next area of validation for our tool was in terms of performance vs. the original method, and the metric that we chose to measure performance was the time taken to completely generate the new SET and Test Suite for the evolved model.

Just as with correctness, we needed a benchmark to compare to, and this was again accomplished by running the same five example models through the symbolic execution engine and test case generation algorithm, with a timer to measure the time of execution to the nearest one-thousandth of a second. These times were recorded as the benchmark time for the tool, and were the “time to beat”. It should be noted that the times reported in this section and in Appendix B – Timing Values are not the only data points, and in fact are averages of three executions for each value, in order to mitigate the variance on the time.

All runs were completed on the same computer, a 2.66GHz Intel® Core™ 2 Quad CPU with 4GB of RAM, running the 64-bit version of Windows 7. Timings were then recorded for our incremental implementation as well, using the same three measure rule, on the same machine as the benchmark to remove as much variability as possible.

8.1.3 Results

For each evolution step for each model, the results that are available are: benchmark performance, tool performance, and the percentage of gain/loss. The individual timing values for each model can be found in Appendix B – Timing Values in chart form for further inspection.

Table 5 outlines the gain or loss, in terms of seconds, as well as percentage of benchmark time, for each evolution step (rows) on the five example models (columns), as well as the averages. Note that a number of the evolution steps do indeed have a positive average (average gain) but the majority show a negative average (average loss). This will be explored further in Section 8.2, with explanations as to why this has occurred.

Evolution Step	Model 1		Model 2		Model 3		Model 4		Model 5		Average	
	Sec	%	Sec	%	Sec	%	Sec	%	Sec	%	Sec	%
Evolution 1	28.80	26.7%	-0.59	-13.7%	1.19	13.2%	-35.85	-476.5%	-0.33	-5.5%	**-1.36**	**-91.1%**
Evolution 2	64.10	55.7%	1.78	36.9%	2.74	35.0%	1.14	15.6%	3.99	55.6%	**14.75**	**39.7%**
Evolution 3	17.84	26.1%	1.44	26.4%	2.42	30.7%	0.32	5.6%	0.60	11.2%	**4.52**	**20.0%**
Evolution 4	36.63	31.6%	-1.17	-26.2%	-0.49	-3.4%	-136.18	-1479.4%	0.09	1.4%	**-20.22**	**-295.2%**
Evolution 5	21.21	23.3%	-1.05	-21.3%	-1.62	-21.8%	-12.56	-206.3%	0.30	5.3%	**1.26**	**-44.2%**
Evolution 6	35.53	34.1%	1.01	22.5%	4.05	42.1%	1.94	31.3%	1.86	33.1%	**8.88**	**32.6%**
Evolution 7	33.54	28.6%	-1.80	-45.2%	0.32	3.9%	-1.45	-21.6%	-0.51	-9.0%	**6.02**	**-8.7%**
Evolution 8	38.93	34.2%	-1.86	-46.3%	0.57	5.4%	-0.41	-5.6%	-0.33	-5.7%	**7.38**	**-3.6%**
Evolution 9	-15.79	-35.1%	-1.15	-30.2%	-12.44	-124.7%	-0.71	-13.6%	1.77	26.4%	**-5.66**	**-35.4%**
Evolution 10	39.97	33.8%	0.24	4.3%	-17.81	-251.7%	-0.40	-5.5%	-4.48	-88.7%	**3.50**	**-61.6%**
Evolution 11	37.91	32.4%	-0.79	-18.4%	-80.80	-951.2%	-32.53	-443.9%	0.39	5.8%	**-15.17**	**-275.1%**
Evolution 12	65.61	56.0%	0.83	19.7%	3.29	34.6%	3.15	39.8%	2.43	38.9%	**15.06**	**37.8%**
Evolution 13	-22.22	-249.6%	-1.78	-45.4%	-3.08	-65.9%	-2.46	-60.5%	-1.05	-19.6%	**-6.12**	**-88.2%**
Evolution 14	-3.30	-2.9%	-1.62	-38.4%	-4.38	-53.1%	-1.72	-22.0%	-1.41	-25.9%	**-2.48**	**-28.5%**
Average	**27.05**	**6.8%**	**-0.46**	**-12.5%**	**-7.57**	**-93.4%**	**-15.55**	**-188.8%**	**0.24**	**1.7%**		

Table 5 Tool Performance - Positive values indicate an improvement over the benchmark (negative indicates loss). Sec is the amount of time gained or lost, and the percentage of the overall gain/loss is shown

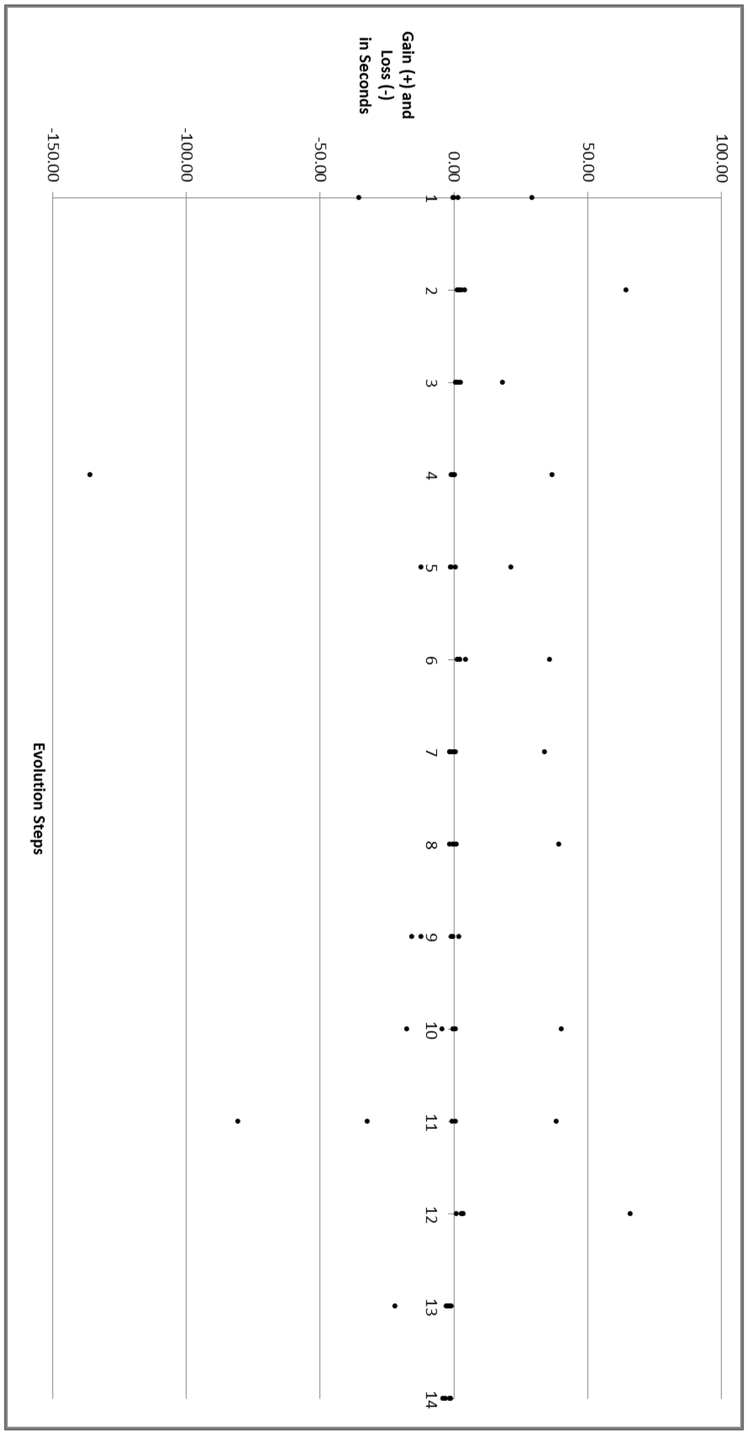


Figure 8.1.1 Tool Performance Graph (Evolution Steps) - Positive values indicate an improvement over the benchmark (negative indicate loss).

Figure 8.1.1 plots the gain and/or loss of the tool versus the benchmark in seconds for the five models for each evolution step. The graph provides slightly more information than the values alone, as the graph is able to communicate that even though the average was a loss for most of the evolution steps, for all but two of them there exist input models for which our technique was able to outperform the benchmark and demonstrate a gain. It is these cases that provide interesting findings which will be discussed in detail in the next section.

Figure 8.1.2 shows a plot for each of the five models, showing much of the same patterns. Even though the averages for three of the five are below zero, all five are showing a gain as their maximum value. It is made clear that models 4 and 5 both contain negative outliers.

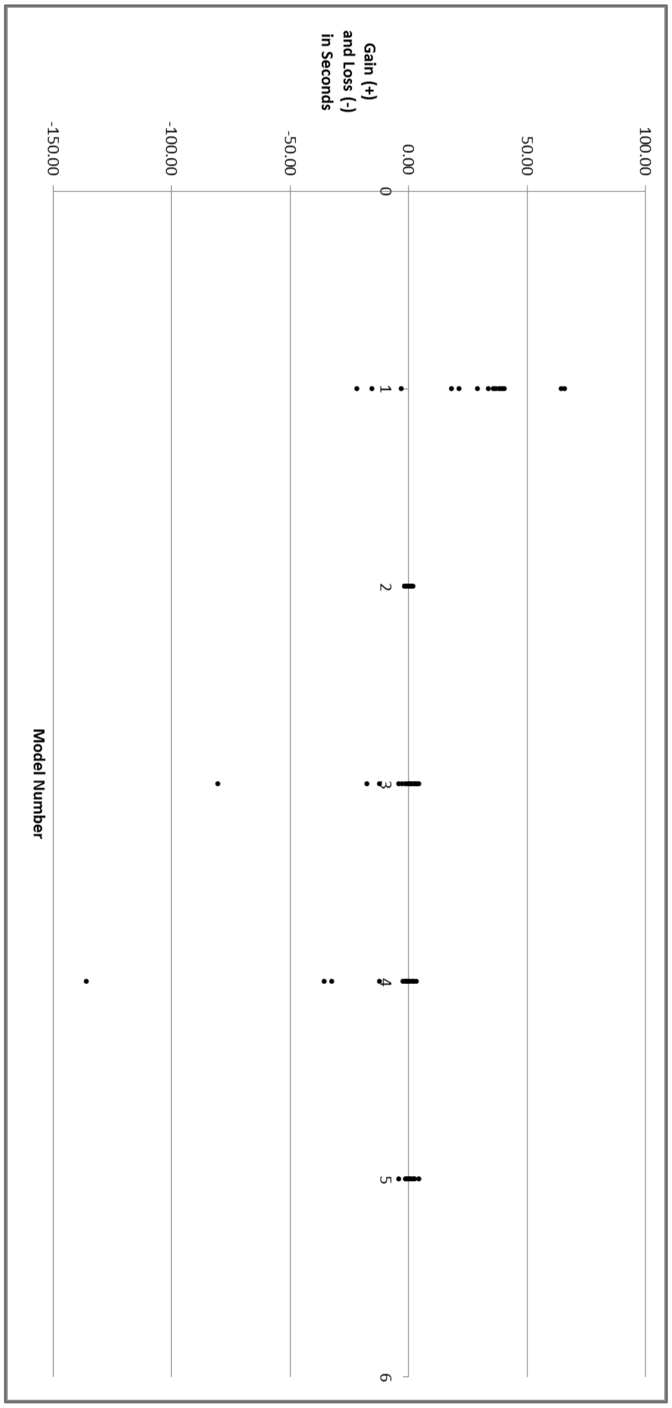


Figure 8.1.2 Tool Performance Graph (Model Number) - Positive values indicate an improvement over the benchmark (negative indicate loss).

8.2 Findings

8.2.1 Evolution Steps

The first point of interest that has been highlighted by the results is the clear clustering of the results based on the initial classifications of evolution steps. The four evolution steps that allowed for direct updates consistently performed better than the benchmark in every example. The evolution steps requiring partial symbolic execution all performed in the middle range, with some of their times being faster than the bench mark and some being much worse, and the average sitting around break even (slightly on the loss side). Finally the one evolution step requiring a full symbolic execution consistently performed worse than the tool in every case. These exact clusters are encouraging support for the classifications defined in Chapter 4, with the single exception of evolution step 13 (removal of a parameter) which also consistently performed worse than the benchmark. The reasoning for this is that the parameter that was removed was used in the initial state, or very early on in execution, meaning that it was basically a full regeneration each time; therefore it would essentially become part of the full regeneration classification. Table 6 highlights the relationship between the classifications as presented and the average gain/loss of the tool in terms of performance.

Evolution Step	Classification of Update Operation	Average % Gain/Loss
1. Add State	Partial Symbolic Execution	-91.14%
2. Modify State	DirectUpdate	39.74%
3. Delete State	DirectUpdate	20.01%
4. Add Transition	Partial Symbolic Execution	-295.19%
5. Modify Transition	Partial Symbolic Execution	-44.15%
6. Delete Transition	DirectUpdate	32.63%
7. Add Entry Code to a State	Partial Symbolic Execution	-8.66%
8. Modify Entry Code on a State	Partial Symbolic Execution	-3.59%
9. Delete Entry Code from a State	Partial Symbolic Execution	-35.44%
10. Add Action Code on a Transition (send output)	Partial Symbolic Execution	-61.56%
11. Add Action Code on a Transition (modify val)	Partial Symbolic Execution	-275.06%
12. Add a Parameter to a Signal	DirectUpdate	37.80%
13. Delete a Parameter from a Signal	Partial Symbolic Execution	-88.21%
14. Modify Initial Value of an Attribute	Full Symbolic Execution	-28.45%

Table 6 Relationship Between Classification and Gain/Loss

These findings are indicative of the point we were trying to prove, being that the symbolic execution is the most time consuming portion of the test case generation, and whenever it can be avoided, the gain increases. This shows that the following changes to a model are the least costly in terms of test case evolution: changing the name of a state, removing a state entirely, removing a transition entirely, or adding a parameter to an input signal, whereas the most costly updates are to remove a parameter from an input signal, or to change the initial value of an attribute. The remainder of the evolution steps each depend on other factors to determine the effectiveness of the tool, and those will be explored further below.

8.2.2 Model Design

Having explored the relationship between the evolution step and the performance of the tool it is interesting to investigate the connection between the model itself and the performance of the tool, in order to determine the types of models where the tool performs better, and the types of models where it will perform consistently at a loss. As was the case with each of the evolution steps, there are some models that perform consistently well, and some that were poor across the board, the only distinction with this analysis is that there is no one model that outperformed the benchmark in every case, nor one that was worse in every case. This lack of a clear cut result is admittedly not encouraging, but the results are still interesting upon further inspection.

The first area of note is Model 1, the model that performed better than the benchmark in 11 of the 14 tests, and two of the three times when a loss was found, it was by rather small amounts. The reason this result is noteworthy is the size of the test suite generated from this model; our largest example, generating 6,820 test cases for the original model. While the model is linear in its execution, containing no loops (meaning the SET generated does not require the subsumption checking to remove extraneous states) the overall performance is still impressive, saving from 23.29% to 56.05% in the cases where it saved time over the benchmark. This suggests that in large models without looping the tool is going to be much more effective than the benchmark method for test case generation.

The issue created by looping within a model, either from loops in the state machine or within code, is that they tend to produce SETs that contain a larger number of subsumed states, whereas a linear model does not. The problem that arises from this is that when performing partial symbolic execution on models of this type, states can be visited more often than necessary, and the subsumption checking may not produce the same results as a full symbolic execution. This is why models containing large numbers of complex loops have shown to perform worse than those of a simpler design.

8.2.3 Model Size

Interestingly enough, the next piece of information obtained from the results seems to imply that the tool performs better on larger models, as opposed to trivial models.

The theory behind this claim is that in larger models, the size of the impacted parts will tend to be smaller relative to the overall model size. By leveraging partial symbolic execution the overhead remains small and only subtrees are generated as needed.

To provide numerical evidence of this, we present the times and gain/loss % for two models that did not perform overly well in cases using partial symbolic execution, showing that as the model increased in size, for the most part the tool performance improved. Table 7 shows these values for Model 4 as the smaller model, and Model 3 as the larger model, both of which share a similar composition, along with the comparison of improvement of the larger model over the smaller model. The evolution steps using partial symbolic execution are highlighted in yellow, as they are the areas of interest for this comparison.

	Model 4			Model 3			
	Benchmark	Tool	Gain/Loss	Benchmark	Tool	Gain/Loss	Improvement?
1	7.523	43.37	-476.50%	9.031	7.842	13.17%	489.66%
2	7.314	6.174	15.59%	7.825	5.09	34.95%	19.37%
3	5.647	5.328	5.65%	7.881	5.462	30.69%	25.05%
4	9.205	145.382	-1479.38%	14.232	14.722	-3.44%	1475.94%
5	6.088	18.647	-206.29%	7.435	9.054	-21.78%	184.52%
6	6.198	4.258	31.30%	9.604	5.558	42.13%	10.83%
7	6.728	8.182	-21.61%	8.266	7.942	3.92%	25.53%
8	7.387	7.801	-5.60%	10.465	9.896	5.44%	11.04%
9	5.188	5.896	-13.65%	9.979	22.419	-124.66%	-111.01%
10	7.29	7.691	-5.50%	7.078	24.892	-251.68%	-246.18%
11	7.327	39.855	-443.95%	8.495	89.298	-951.18%	-507.24%
12	7.91	4.761	39.81%	9.519	6.229	34.56%	-5.25%
13	4.062	6.518	-60.46%	4.667	7.743	-65.91%	-5.45%
14	7.822	9.543	-22.00%	8.236	12.611	-53.12%	-31.12%

Table 7 Comparison of Model Size on Computation Time for Model 4 and Model 3

For this comparison we will exclude evolution step 13 from any further investigation for the reasons stated in the previous section. Of the eight remaining evolution steps for comparison, five of them show an improvement as the size of the model increases, three of which show a very substantial improvement. These results are very encouraging for the claim that the tool performs better on a larger model than a smaller model; the counter-supportive numbers can likely be attributed to another factor that plays into the efficiency of the tool which will be discussed in the next section.

Figure 8.2.1 shows an emerging trend of gain increasing with model size. There are two distinct clusters shown between similar models in terms of size and complexity (note models 3 and 4 fall within the same cluster). Within these clusters, as the size increases the performance (in this case in terms of seconds) increases as well. It is important to note that the horizontal axis is logarithmic, so the lines drawn show signs of a logarithmic correlation.

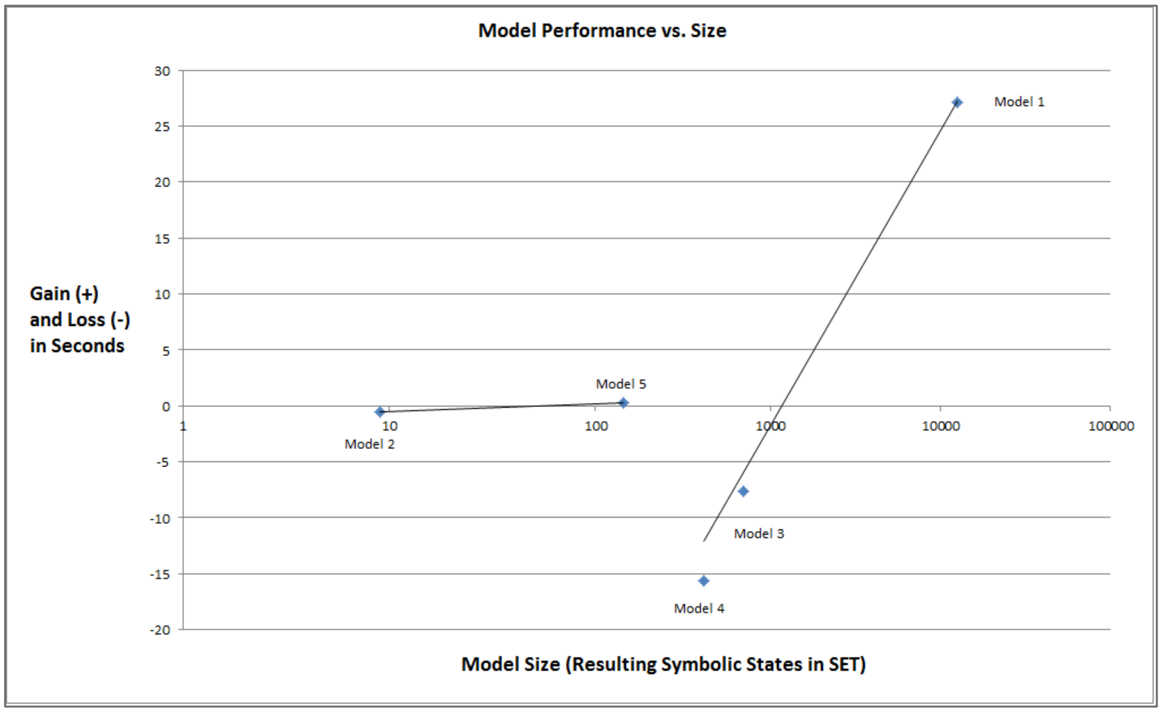


Figure 8.2.1 Model Performance vs. Size

8.2.4 Location of Change

The last area of interest in terms of findings is the location of the change within the UML-RT model, and its representative location with the SET. Intuitively evident is the fact that location plays a big role in the performance of the tool; a change early on in execution (near the top of the SET) is much more costly than one later on in execution (near the bottom of a tree). The reasoning behind this is that the higher in the tree the difference occurs, the larger the subtree that is generated by partial symbolic execution will need to be. To show this, refer to example Model 3, which is a fairly large model with a number of states and looping, and we will be looking specifically at two evolution steps performed at different extremes of the tree: the addition of a new state reachable from State16 (evolution step 1), and the addition of action code to a transition between State3 and State4 (evolution step 10). The first evolution takes place at a location such that it will only require generation of subtrees from leaf nodes of the original tree, whereas the second will require generation of large subtrees early on in execution. Table 8 shows the times for these two evolution steps for both the benchmark and the tool, and you can see that the change occurring later on in the model (1) performed at a gain, while the one with the change early on in the model (10) performed at a significant loss. The interesting factor in this is that because even though the difference in evolution step occurs more times than the one in evolution step 10, meaning that it actually performs partial symbolic execution more times than in evolution step 10, it still outperforms it, due to the size of the trees generated.

	Benchmark Time	Tool Performance	%Gain/Loss
1	9.031	7.842	13.17%
10	7.078	24.892	-251.68%

Table 8 Performance for Model 3 (selected locations of change)

This type of result is demonstrated in a number of other models as well, such as in Model 2, where evolution step 7 affects the waiting state which is early on in the model performed at a loss of 45.2%, and evolution step 10 which effects the transition from zero to returning which is late in the model performs at a gain of 4.28%.

From these findings it stands to reason that changes early on in a SET have a larger impact on the generation of test cases than those later on in execution.

8.3 Summary

Based on the results of the validation, a number of findings have been observed in regards to the performance of the tool and more generally, the effects of model evolutions on symbolic execution and test case generation.

The first finding has to do with the type of model evolution step, and the classifications of those model evolution steps. In all cases the evolution steps classified as “Direct Update” outperformed the benchmark, thus confirming that the use of symbolic execution should be avoided whenever possible. Additionally the evolution step that was classified as “Full Symbolic Execution” consistently performed worse than the benchmark.

The next conclusion that we were able to come to was that the design of the model plays a big part in the efficiency of the tool. Models that are lacking cycles and other complex design patters performed better than those that were complicated; an example of this is Model 1, which is the largest model, and contains no cycles in execution. With Model 1, the tool performed better than the benchmark in all but three of the evolution steps. The model which performed second best of the five examples was Model 5 which contains a number of cycles, which as previously explained cause issues in partial symbolic execution, but overall was simpler than Models 3 and 4 which were full of cycles.

Another area that plays a role in the effect of model evolution on SET and test case generation is model size. This is shown first and foremost by Model 1, which is by far the largest of the models and performed the best of them all. However, as previously stated Model 1 was also a simple model, so excluding Model 1 from the model size analysis was necessary to provide useful findings. Using Model 4 as the small model and Model 3 as the larger model, we compared their performances, and in a majority of the evolution steps using partial symbolic execution, the larger model performed better than the smaller model. Possible reasoning for this was explained, having to do with the overhead of the process becoming less significant as a model increases in size.

The last interesting finding had to do with location of change, and how that affected the efficiency of the tool. We were able to conclude that changes early on in execution have a larger impact on the SET and tests than those later on in execution, due to the propagating nature of SETs, and an effect on the actual generation due to the sizes of the subtrees that are generated. An earlier change requires a large subtree generation, whereas a later change requires a smaller generation (in most cases).

Based on all of these findings we can conclude that the tool would perform best on a large, simple model, where the change occurs later on in execution, or is a change in the “Direct Update” classification. This is held true, as the two highest gains across all models and all evolution steps are within Model 1, and are evolution steps 2 and 12, with gains of 55.72% and 56.05% respectively. When we exclude that classification, the largest gains are still found in Model 1, which is our largest simplest model. Excluding Model 1 because of its simplicity, while still excluding the “Direct Update” classification, the largest gain is evolution step 9 in Model 5, which is a change later on in execution; this provides proof of the importance of location of change. Conversely, an example of poor model that developers would hope to avoid in development, would be a smaller to medium sized model, containing a number of loops, and the evolution performed would be one of the two consistently poor performing steps (modify initial value or remove parameter), or some other step requiring partial symbolic execution from an early on state in the SET. Keeping these two extremes in mind during development in testing can provide useful insight in to the best ways to approach development of a model of a system.

Chapter 9. Conclusion

9.1 Summary

The first of the main goals of this work was to develop a tool capable of incrementally generating test cases for UML-RT models, which would use existing artifacts from the original model; this was accomplished through the use of several individual components being combined together into IncreTesCaGen, a single plugin for IBM Rational Software Architect RealTime Edition [16]. The tool takes as input, an original model and the symbolic execution tree for that model, as well as a full test suite for that model, along with an evolved model, and performs a comparison on the two models. Based on the results of our comparative study, the tool makes the appropriate updates to the SET and test suite such that they become reflective of the new model. One of the main goals in this implementation was to avoid symbolic execution whenever possible, as this is the most costly operation in the test case generation process. The end result was a tool that avoided symbolic execution whenever possible, and utilized an adapted partial symbolic execution in cases where that would be sufficient in updating the SET, which is then used to generate the new test cases.

The second of our goals was to better understand the effects of model evolution on models, and on their execution and testing. This was completed through a study of five example models (found in Appendix A – Example Models), in which the models were all symbolically executed and had test cases generated, where all of these artifacts would act as the original artifacts for 14 evolutions that were performed on each model. After the model evolution was performed, the evolved models were then symbolically executed and ran through the test case generation algorithm, resulting in 70 sets of artifacts. These were then compared with the originals, to determine how evolution affected the artifacts, in hopes of observing patterns that would help provide classifications for the evolution steps. In this classification exercise we came up with three categories of updates that encompass all of the evolution steps that were studied: Direct Update (evolution steps where updates directly to the SET and Test Suite were possible), Partial Symbolic Execution (evolution steps that required some portion(s) of the SET to be regenerated based on the observed differences between models), and Full Symbolic Execution (evolution steps that resulted in drastic changes that require a full regeneration of the SET).

There was additional information to distill in terms of model evolution, and this was retrieved through our tool validation; an analysis of performance of the developed tool. As part of the validation step, we were able to distill additional information regarding which types of updates consistently performed well, and those that consistently underperformed in comparison to the benchmark. In addition to performance data for the types of evolution steps, interesting patterns emerged in relation to the model itself, and how well certain model design aspects perform, such as model size, model complexity, and location of change. Based on the findings we were able to highlight what would make a "perfect model" in terms of evolution, including the type of evolutions to perform. This "perfect model" would be larger in size, contain no (or a very small number of loops), and the evolution step would be one of the changes that fall into the Direct Updates category. Obviously not all models can be of this design, and not all updates will be in that category, but the findings allow developers to understand the effects of design and evolution, and the hope is that this will allow for development to be tailored more in the direction of model evolution, as this is of importance in the iterative design paradigm of model-driven development.

9.2 Alternate Implementation Options Explored

As is the case with any development process, there are often a number of alternative options to arrive at a similar end product, but the best option is the one that is ultimately chosen. However it is often useful to look at some of the alternatives and describe why that method was not chosen in the final implementation. One such design choice for this work was the exclusion of an algorithm to incrementally generate the test cases based on two SETs and differencing them. This of course was only useful in the cases where direct updates were not possible, the gain from those types of evolutions was much more than this method could produce. Parts of this work were presented in a work-in-progress poster [5], but the work has since been excluded for the reasons we will discuss.

The following steps outline the process taken to perform incremental test case generation:

1. Obtain SET for new model (either partial or full symbolic execution)
2. Difference the two SETs to determine highest level of difference
3. Identify “path to difference”
4. Remove any test cases from test suite beginning with the “path to difference”
5. Generate new tests from the identified difference point of new SET
    a. Solve any new constraints introduced for each case
6. Prepend “path to difference” to all newly generated test cases
7. Add newly completed test cases to test suite for complete suite for new model

Figure 9.2.1 shows two SETs, (A) being the original model and (B) being the evolved model, with the differences between them shown by highlighting the difference points in red in SET (B). The numbers identify the transitions required to move from one symbolic state to another, meaning for example, the left-most test case would follow the path 1 → 3 → 7. The yellow paths that are the same in each tree are examples of the prefixes to the differences found, meaning that once the differences have been recorded, all of the test cases that begin with the transition chains 1 → 4 → 10, or 2 → 6 are removed. In this example three test cases would be removed of the eight total. The next step is to generate test case stubs from the difference points, referring to the two red states. These newly generated stubs, which in this case only contain one transition, are then appended to the prefix for that difference, resulting in the four following new test cases:

1. 1 → 4 → 10 → 16
2. 2 → 6 → 13
3. 2 → 6 → 14
4. 2 → 6 → 15

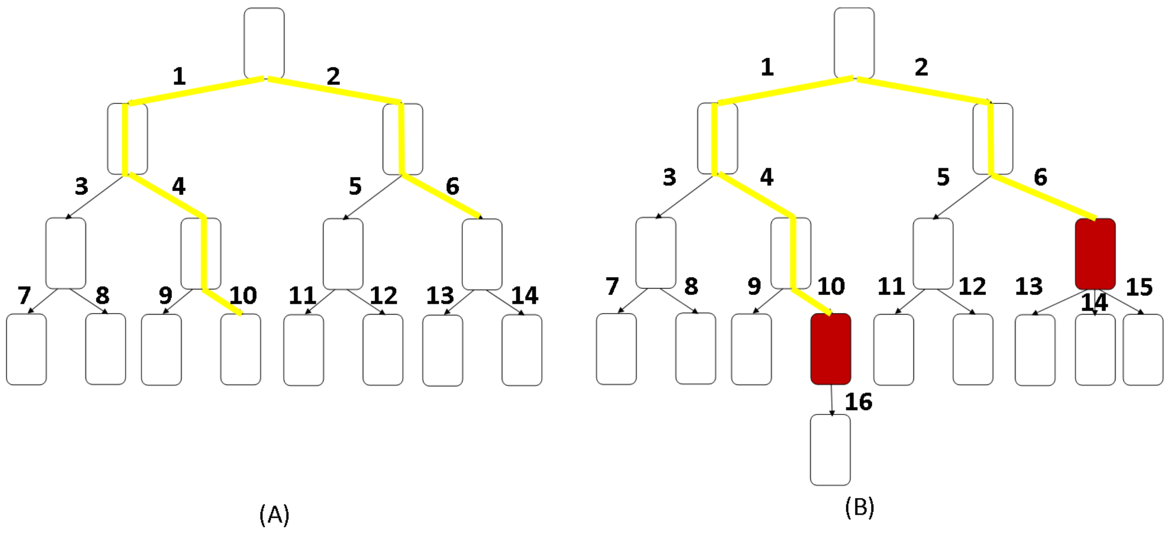


Figure 9.2.1 Two Different Trees Showing Prefixes to Differences

The four new tests are added to the remaining five test cases, and a complete test suite with nine test cases is the result.

The reason this method was ultimately not included in tool development was that there was very minimal opportunity for gain, and even the possibly of a loss in performance over simply regenerating all of the test cases using the generation algorithm. The reasoning behind this is the algorithm used to generate test cases is a recursive depth first search of the tree to collect the series of inputs for each test, followed by solving any constraints for that particular branch in the tree. While the proposed method of incrementally generating the tests also involved a recursive depth first search of the tree to find differences and searched all the way to leaf nodes in the cases where no difference was found. At an initial glance this method shows gain in the fact that it does not explore beyond the differences during this search, and while this is true, those remaining branches are explored during the generation of the new tests from that point, and any savings is lost immediately. Additionally, since it is a comparison of two trees, the traversal up until that point is done two times, once for each tree. The only place there was even the potential for any gain was the reduced amount of constraint solving that occurs as a result of this method, which only necessitates solving constraints for the new branches of the SET, as opposed to all paths. However, this poses an additional problem, in that the new execution paths in the tree may have imposed additional constraints on variables that were solved for earlier use and are no longer valid, and this is not reflected through this algorithm.

While the proposed incremental generation idea showed promise, it was ultimately excluded from the tool, however its removal does not preclude further development in the future and inclusion at a later time.

9.3 Limitations

As is the case with all software, throughout design and development a number of events occur that lead to some portion of the system needing to be scaled back or limited in some way; this tool was no exception. This section aims at summarizing limitations due to various causes, but mainly limitations imposed to ensure research progress.

The first, and possibly largest limitation of the current tool is that it works only for simple, single capsule models, and does not function for communicating and contained capsules. These types of models are more predominant and common amongst real development examples; however implementation does not include their use. The main reason for this implementation was to ensure progress of the tool development, as these types of models were initially included in design, and early implementations incorporated their additional information, such as the Ecore model for SETs that was created, as it contains four types of states to differentiate containment of states and whether a state was a composite state or not. As stated, later on in development it was decided to not include these features, but the framework for them still remains.

There were a number of small limitations that came from limitations in two of the other tools used during development: RSA-RTE [16], and SAUML [3]. Since our implementation uses the internal differencing tool from RSA-RTE, we were bound by the functionality of the differencing tool, and the results that it presents developers with were sometimes not overly succinct. For example, if a hierarchical state is added to a state machine, containing some number of states and transitions within it, the differencing tool only reported that the top level state was added to the model, and further investigation is required to determine this; however with no indication, this is problematic. While this is not a major concern for our development, as the current set of evolution steps contains simple one item changes, it prevents further work being completed for evolution steps such as hierarchical composition, and limits the types of changes traceable by the tool. In terms symbolic execution tool (SAUML), there are a number of limitations imposed that propagate through to this tool as well. For example, the tool limits the languages that can be used as action code to either Java or C++, and does not allow for any other language on transitions or as entry code. Additionally, the types of variables are limited to only integer or double types, and do not include strings or boolean values. The last limitation imposed by SAUML deals again with more complex models, which although are not included in our testing, is still a limitation; the tool does not provide functionality for features of UML-RT such as dynamic ports and plugin capsules which are features of UML-RT that deal with dynamic updating of execution at runtime, such as connecting to a port or instantiating a new optional capsule respectively.

9.4 Future Work

There are a number of areas of this work that allow for improvements and advancements that will improve the functionality and performance of the tool; this section aims to discuss what they are and initial ideas for their implementation, if applicable.

The first area of future work is simply expanding the tool to function with more complex and real-life models; currently the tool works for a small subset of UML-RT models, and the goal is that it would eventually work on any model supplied. The current area where this work would be beneficial is to include communicating and contained capsules, as well as dynamic runtime changes within the model. The runtime limitations have been imposed by the SAUML tool used for symbolic execution, but presumably development for this tool may eventually move to include those features, and we would certainly leverage that functionality with this tool. In order to implement the updates to include communicating and contained capsule models, the framework exists for this, and more work would be needed for the test case generation, but many of the other facets such as tree differencing and symbolic execution already are capable of these types of models. The main theme among this possible future work is an expansion of the functionality in terms of input models, whether it is in terms of size or complexity, or the addition of features of UML-RT that are not yet included.

Another area of future work that has been considered is the further investigation and implementation of the incremental test case generation method described in Section 9.2. With further investigation and time, it could be possible to update this implementation such that the gain becomes substantial enough to merit using it in place of the existing test case generation algorithm.

Additionally, the idea of exploring other methods for test cade generation has been considered. Instead of using symbolic execution as a method of generating a test suite that will create tests for every path, it may be worth while exploring other options such as state or transition coverage. While this may lead to a loss in analysis, and overall testing, the removal of the dependency on symbolic execution may provide substantial gain.

We also feel that work could be done in terms of optimizing the existing process, specifically within the decision making step. Currently in a list of change, they will be performed in the order presented by the RSA-RTE differencing tool (with minor exceptions) and it would be beneficial to look at optimizations in this area, such that less computation is done overall, by ordering the updates in a specific way, and grouping like actions together. Some initial optimizations have been made in our implementation, but there is certainly room for additional improvements.

References

[1] K. Zurowska and J. Dingel. "Symbolic Execution of UML-RT State Machines". 27th ACM Symposium on Applied Computing, Track on Software Verification and Testing (SAC-SVT'12). Riva del Garda, Italy, March 25-29, 2012.

[2] K. Zurowska and J. Dingel. "SAUML - a Tool for Symbolic Analysis of UML-RT Models". Tool Demonstration Paper. 26th IEEE/ACM International Conference On Automated Software Engineering (ASE'11).

[3] SAUML 0.0.1- a Tool for Symbolic Analysis of UML-RT Models - http://research.cs.queensu.ca/~mase/sauml.html

[4] K. Zurowska and J. Dingel. "Symbolic Execution of UML-RT State Machines". Technical Report 2011-578, School of Computing, Queen’s University, June, 2011.

[5] E.J. Rapos and J. Dingel. “Incremental Test Case Generation for UML-RT Models Using Symbolic Execution”. Poster and Extended Abstract. Fifth International Conference on Software Testing, Verification, and Validation (ICST 2012). Montreal, Canada, April 17-21, 2012.

[6] Unified Modeling Language (UML 2.0) Superstructure. http://www.uml.org

[7] B. Selic. “Using UML for Modeling Complex Real-Time Systems”. Muller, F., Bestavros, A. (eds.) LCTES 1998. LNCS, vol. 1474, pp. 250–260. Springer, Heidelberg (1998)

[8] L. Frantzen, J. Tretmans, and T. Willemse. “Test generation based on symbolic specifications”. FATES 2004 (LNCS 3395), pages 1 – 15, 2004.

[9] S. Gnesi, D. Latella, and M. Massink. “Formal test-case generation for UML statecharts”. ICECCS 2004, pages 75–84, 2004.

[10] N.H. Lee, S.D. Cha. “Generating test sequence using symbolic execution for event driven real-time systems”. Microproc. and Microsys. 27, 523–531 (2003)

[11] C. Pasareanu, W. Visser. “A survey of new trends in symbolic execution for software testing and analysis”. Journal on Software Tools for Technology Transfer 11 (2009)

[12] K. Bogdanov, M. Holcombe, and H. Singh. “Automated test set generation for statecharts”. FM-Trends 1998 (LNCS 1641), pages 107 – 21, 1998.

[13] J. King. “Symbolic execution and program testing”. Communications of the ACM, 19(7):385 – 94, 1976.

[14] T. Jeron. “Symbolic Model-based Test Selection”. ENTCS, 240:167 – 184, 2009.

[15] Choco Constraint Solver - http://www.emn.fr/z-info/choco-solver/

[16] IBM Rational Software Architect RealTime Edition - http://www-947.ibm.com/support/entry/portal/overview//software/rational/rational_software_architect_realtime_edition

[17] IBM Rational Rose RealTime - ftp://ftp.software.ibm.com/software/rational/docs/documentation/manuals/rosert.html

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

[19] EMF Compare - http://www.eclipse.org/emf/compare/

[20] H. Liu, Y. Lei, J. Zhang. “A Modeling Approach for Development of An Automotive AMT ECU Software With UML-RT”. International Conference on Software Engineering and Data Mining (SEDM 2010), June 23-25, 2010

[21] D. Herzberg, “UML-RT as a Candidate for Modeling Embedded Real-Time Systems in the Telecommunication Domain”. Proceedings of the 2nd international conference on The unified modeling language (UML'99), pages 330-338, 1999

[22] S. Person, G. Yang, N. Rungta, S. Khurshid, “Directed Incremental Symbolic Execution” 32nd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2011), San Jose, California, June 4-8, 2011.

[23] E. Uzuncaova, S. Khurshid, D. S. Batory, “Incremental Test Generation for Software Product Lines”. IEEE Transactions on Software Engineering (TSE) Volume 36, Number 3, pages 309-322, 2010

[24] P. K. Chittimalli, M. J. Harrold, “Recomputing Coverage Information to Assist Regression Testing”. IEEE Transactions on Software Engineering (TSE) Volume 35, Number 4, pages 452-469, 2009

[25] S. Bates, S. Horwitz, “Incremental Program Testing Using Program Dependence Graphs”, 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL ’93), New York, NY, USA, 1993

[26] M. Mirzaaghaei, F. Pastore and M. Pezze, “Supporting Test Suite Evolution through Test Case Adaptation”, Fifth International Conference on Software Testing, Verification, and Validation (ICST 2012), Montreal, QC, Canada, 2012.

Appendix A – Example Models

LEGEND

For all models, the following are the definitions for the labels describing the models.

Loops –Does the Model Contain Loops?
States – Number of States in the State Machine
Transitions – Number of Transitions in the State Machine
SET Size – Number of Symbolic States in the SET
Test Suite Size – Number of Test Cases in the Test Suite

MODEL 1

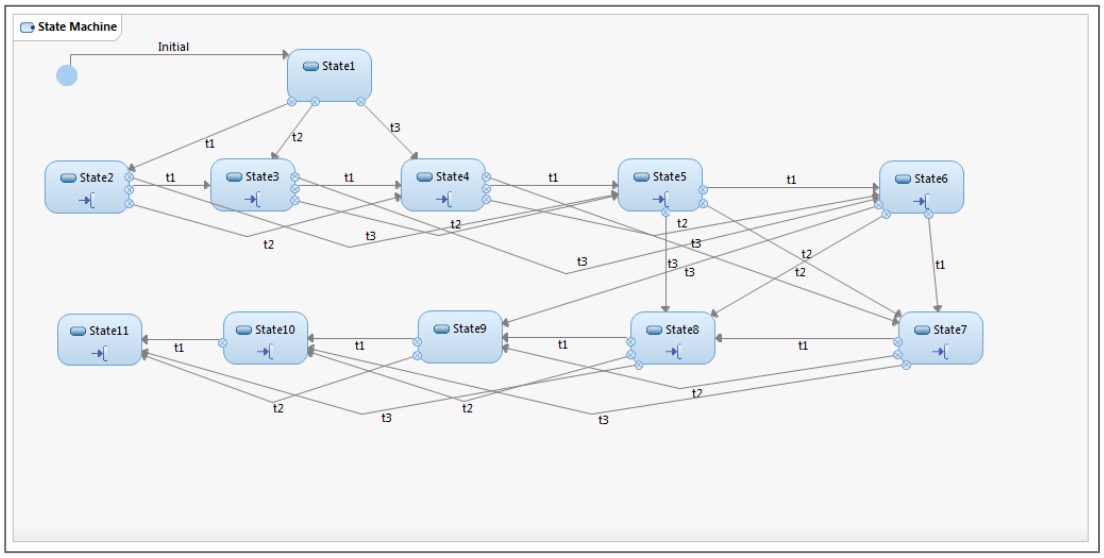


Figure 9.4.1 State Machine for Model 1

Model 1	
Loops	No
States	11
Transitions	27
SET Size	12576
Test Suite Size	6820
Additional Details	
Created as a simple model with unrealistic style, as an example of a model with no looping/cycles	

Table 9 Details for Model 1

Evolution Step	More details
1. Add State	Added "new" state reachable from State3 via comms.dummy()
2. Modify State	changed name of State4 to State40
3. Delete State	Deleted State9
4. Add Transition	added transition from state2 to State11 with trigger of comms.in5()
5. Modify Transition	changed target of transition from State2 to State4 to be State10
6. Delete Transition	Delete transition from State10 to State11
7. Add Entry Code to a State	Add sending of blank to state1
8. Modify Entry Code on a State	changed entry code to state4 (removed else branch)
9. Delete Entry Code from a State	removed entry code from State7
10. Add Action Code on a Transition (send output)	Add sending of blank to transition between State7 and State8
11. Add Action Code on a Transition (modify val)	Added assignment of val1 to 1000 on transition from State2 to State4
12. Add a Parameter to a Signal	added int param to dummy
13. Delete a Parameter from a Signal	removed param from in1
14. Modify Initial Value of an Attribute	changed initial value of val1 from 0 to 1

Table 10 Detailed Evolutions for Model 1

MODEL 2

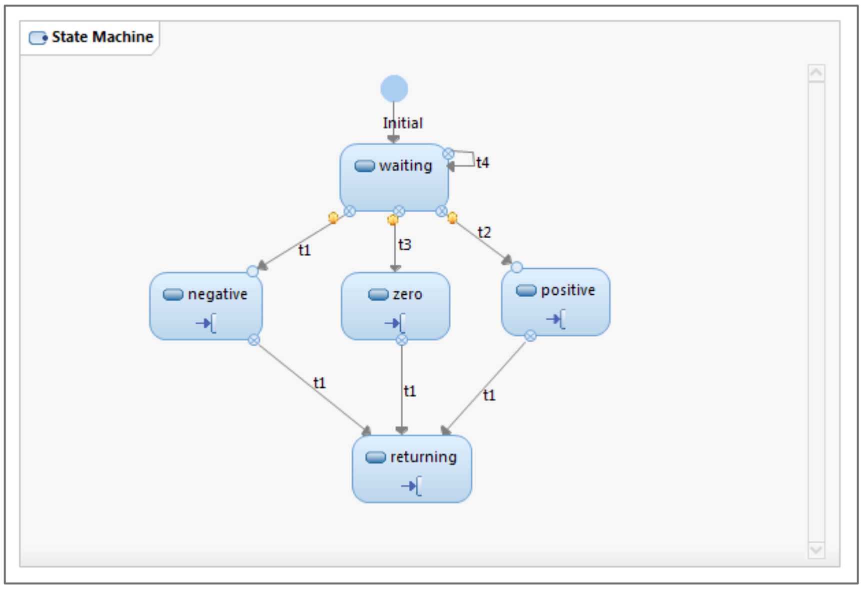


Figure 9.4.2 State Machine for Model 2

Model 2	
Loops	Yes
States	5
Transitions	7
SET Size	9
Test Suite Size	4
Additional Details	
Extremely trivial example used as the proof of concept for many of the tests. The results from this model are bound to be the least reflective of tool performance.	

Table 11 Details for Model 2

Evolution Step	**More details**
1. Add State	added a new state reachable from returning state called done, reachable via null transition
2. Modify State	changed the name of returning to EndState
3. Delete State	deleted the returning state
4. Add Transition	added a transition from returning back up to waiting, basically allowing functionality to loop, this should drastically increase the tree size
5. Modify Transition	chnaged target of self loop to become the returning state
6. Delete Transition	deleted transition from zero to returning
7. Add Entry Code to a State	change value of val upon entering waiting state
8. Modify Entry Code on a State	change entry code on state zero to assign val to 0 as opposed to the input value, since we are already certain that it is 0, no change in execution but improves efficiency
9. Delete Entry Code from a State	remove entry code to returning state (the sending of output)
10. Add Action Code on a Transition (send output)	send a null signal on transition from zero state
11. Add Action Code on a Transition (modify val)	modify val by setting it to zero on the transition out of negative.
12. Add a Parameter to a Signal	added a parameter to null. adding a parameter to a signal means that any time that input is taken a symbolic variable is now passed as a parameter, the effect on this should alter every instance of this call by having 0 as the input.
13. Delete a Parameter from a Signal	removed parameter from input signal (also needed to remove references to this parameter that caused errors in symbolic execution)
14. Modify Initial Value of an Attribute	change initial value of val to 5 only effects value in initial and waiting states, rest of tree should be uneffected

Table 12 Detailed Evolutions for Model 2

MODEL 3

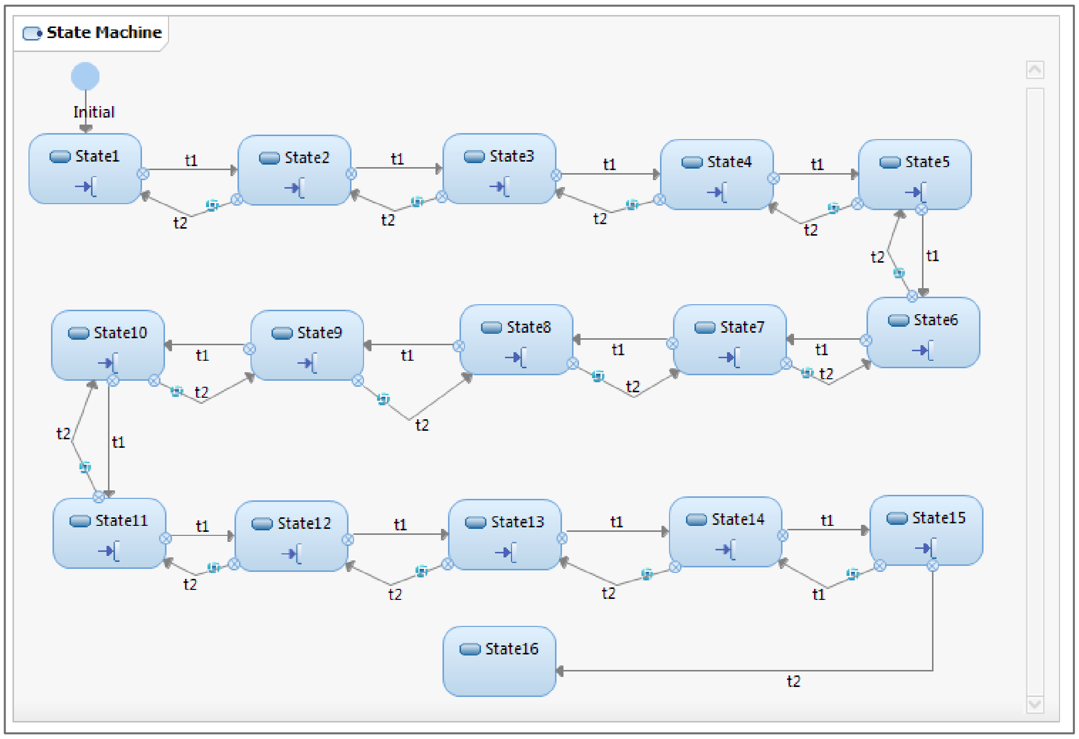


Figure 9.4.3 State Machine for Model 3

Model 3	
Loops	Yes
States	16
Transitions	29
SET Size	689
Test Suite Size	439
Additional Details	
A mid sized model that is fairly complex in design, containing a number of loops.	

Table 13 Details for Model 3

Evolution Step	**More details**
1. Add State	Added a reachable state 17 from 16
2. Modify State	change the name of State 13 to State 135
3. Delete State	Deleted State 10
4. Add Transition	Added transition from State2 to State11 with a trigger of comms.in1()
5. Modify Transition	changed trigger of transition from State1 to State2 to in1, instead of in2
6. Delete Transition	Deleted the transition from State 14 to State 13
7. Add Entry Code to a State	Added entry code to State16 to set val to 0.
8. Modify Entry Code on a State	Changed the entry code of State2 to send 0 instead of val in the first branch of the conditional
9. Delete Entry Code from a State	remove entry code from State4
10. Add Action Code on a Transition (send output)	added "comms.out2().send()" to transition between State3 and State4
11. Add Action Code on a Transition (modify val)	added "val = 0" to transition between State6 and State7
12. Add a Parameter to a Signal	added a parameter to in2. adding a parameter to a signal means that any time that input is taken a symbolic variable is now passed as a parameter, the effect on this should alter every instance of this call by having 0 as the input.
13. Delete a Parameter from a Signal	remove the parameter to in1
14. Modify Initial Value of an Attribute	change initial value of val to 5

Table 14 Detailed Evolutions for Model 3

MODEL 4

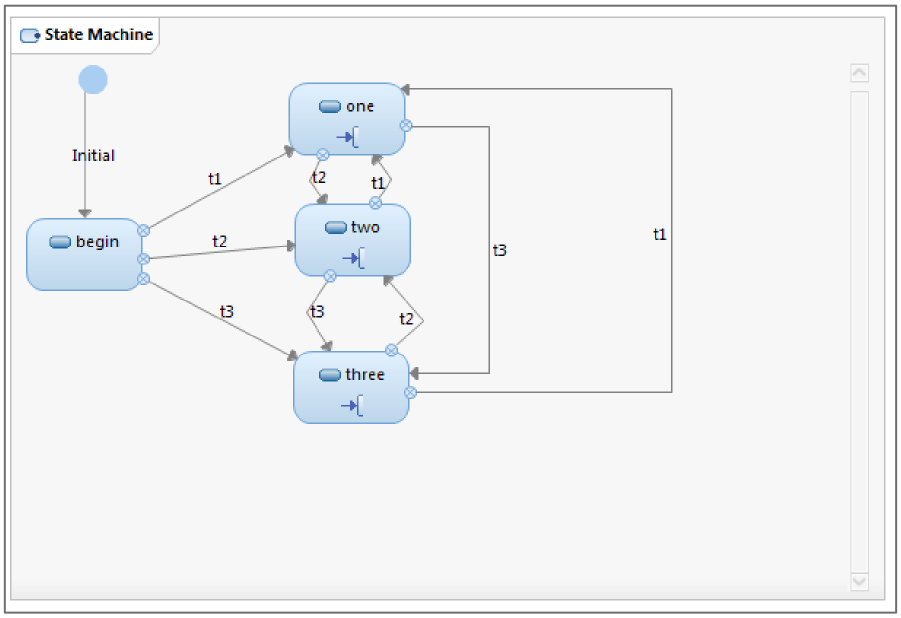


Figure 9.4.4 State Machine for Model 4

Model 4	
Loops	Yes
States	4
Transitions	9
SET Size	403
Test Suite Size	288
Additional Details	
A model similar in composition to Model 3, but of a smaller size. Useful in comparison of the effects of model size on performance.	

Table 15 Details for Model 4

Evolution Step	More details
1. Add State	added a reachable state doubleThree reachable from three via an in3 transition
2. Modify State	changed three to 3
3. Delete State	removed state three
4. Add Transition	added transition from state three to begin
5. Modify Transition	changed the trigger of the transition from three to one from in1 to in3
6. Delete Transition	removed transition from three to two
7. Add Entry Code to a State	added entry code to begin that decreases value of val by 1
8. Modify Entry Code on a State	changed entry code of three, instead of outputting val in out1, it is changed to 0
9. Delete Entry Code from a State	removed entry code from one
10. Add Action Code on a Transition (send output)	added an output of out2 to the transition from begin to three
11. Add Action Code on a Transition (modify val)	added an assignment of val to 0 to the transition from three to one
12. Add a Parameter to a Signal	added an int param to in3
13. Delete a Parameter from a Signal	removed the int param from in2
14. Modify Initial Value of an Attribute	changed the initial value of val from 0 to -1

Table 16 Detailed Evolutions for Model 4

MODEL 5

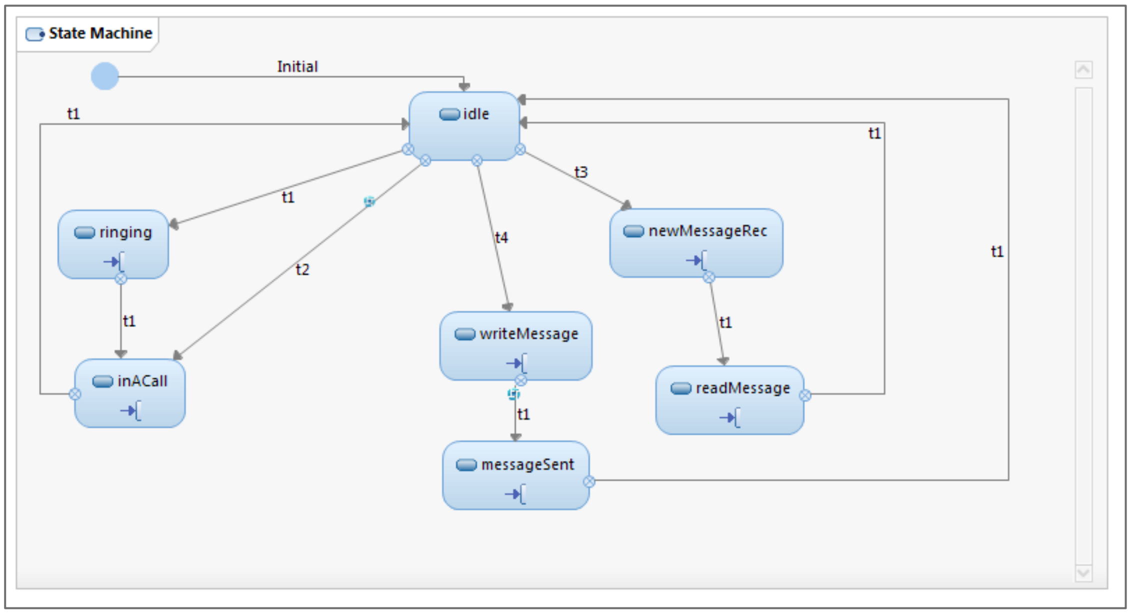


Figure 9.4.5 State Machine for Model 5

Model 5	
Loops	Yes
States	7
Transitions	10
SET Size	137
Test Suite Size	66
Additional Details	
An attempt at a more realistic example, using actual names for states and transitions, in order to demonstrate practical application.	

Table 17 Details for Model 5

Evolution Step	More details
1. Add State	added dummy state reachable from idle, via a ping input
2. Modify State	changed idle to homeScreen
3. Delete State	removed ringing state (simulate inavailability of incoming calls)
4. Add Transition	add transition from idle to idle, with ping input
5. Modify Transition	changed transition that went from messageSent to idle, to have a target of ringing (not practical, but shows change of transition)
6. Delete Transition	delete transition from messageSent to idle
7. Add Entry Code to a State	added sending of notify to entry of idle
8. Modify Entry Code on a State	change the sign from < to > in the entry code of ringing
9. Delete Entry Code from a State	removed entry code from readMessage
10. Add Action Code on a Transition (send output)	added sending of notify on transition from newMessageRec to readMessage
11. Add Action Code on a Transition (modify val)	added setting of val to 1000 on transition from idle to ringing
12. Add a Parameter to a Signal	added integer parameter to makeCall signal
13. Delete a Parameter from a Signal	removed parameter from incomingCall
14. Modify Initial Value of an Attribute	changed initial value of val1 from 7 to 70

Table 18 Detailed Evolutions for Model 5

Appendix B – Timing Values

Evolution Step	Benchmark Time (sec)	Tool Performance (sec)	%Gain/Loss
1	107.698	78.901	26.74%
2	115.041	50.942	55.72%
3	68.404	50.569	26.07%
4	115.788	79.16	31.63%
5	91.086	69.872	23.29%
6	104.089	68.562	34.13%
7	117.176	83.638	28.62%
8	113.792	74.863	34.21%
9	44.928	60.717	-35.14%
10	118.172	78.202	33.82%
11	116.836	78.923	32.45%
12	117.057	51.451	56.05%
13	8.9	31.118	-249.64%
14	114.047	117.348	-2.89%
		Average	6.79%

Table 19 Performance for Model 1

Evolution Step	Benchmark Time (sec)	Tool Performance (sec)	%Gain/Loss
1	4.303	4.891	-13.66%
2	4.824	3.044	36.90%
3	5.451	4.012	26.40%
4	4.45	5.616	-26.20%
5	4.92	5.968	-21.30%
6	4.492	3.482	22.48%
7	3.978	5.776	-45.20%
8	4.012	5.87	-46.31%
9	3.827	4.981	-30.15%
10	5.627	5.386	4.28%
11	4.312	5.105	-18.39%
12	4.217	3.388	19.66%
13	3.93	5.714	-45.39%
14	4.216	5.835	-38.40%
		Average	-12.52%

Table 20 Performance for Model 2

Evolution Step	Benchmark Time (sec)	Tool Performance (sec)	%Gain/Loss
1	9.031	7.842	13.17%
2	7.825	5.09	34.95%
3	7.881	5.462	30.69%
4	14.232	14.722	-3.44%
5	7.435	9.054	-21.78%
6	9.604	5.558	42.13%
7	8.266	7.942	3.92%
8	10.465	9.896	5.44%
9	9.979	22.419	-124.66%
10	7.078	24.892	-251.68%
11	8.495	89.298	-951.18%
12	9.519	6.229	34.56%
13	4.667	7.743	-65.91%
14	8.236	12.611	-53.12%
		Average	-93.35%

Table 21 Performance for Model 3

Evolution Step	Benchmark Time (sec)	Tool Performance (sec)	%Gain/Loss
1	7.523	43.37	-476.50%
2	7.314	6.174	15.59%
3	5.647	5.328	5.65%
4	9.205	145.382	-1479.38%
5	6.088	18.647	-206.29%
6	6.198	4.258	31.30%
7	6.728	8.182	-21.61%
8	7.387	7.801	-5.60%
9	5.188	5.896	-13.65%
10	7.29	7.691	-5.50%
11	7.327	39.855	-443.95%
12	7.91	4.761	39.81%
13	4.062	6.518	-60.46%
14	7.822	9.543	-22.00%
		Average	-188.76%

Table 22 Performance for Model 4

Evolution Step	Benchmark Time (sec)	Tool Performance (sec)	%Gain/Loss
1	5.985	6.312	-5.46%
2	7.184	3.193	55.55%
3	5.31	4.714	11.22%
4	6.325	6.235	1.42%
5	5.575	5.279	5.31%
6	5.604	3.749	33.10%
7	5.658	6.169	-9.03%
8	5.793	6.123	-5.70%
9	6.708	4.936	26.42%
10	5.052	9.534	-88.72%
11	6.663	6.278	5.78%
12	6.247	3.815	38.93%
13	5.327	6.373	-19.64%
14	5.446	6.854	-25.85%
		Average	1.67%

Table 23 Performance for Model 5