Abstract

Model Driven Engineering (MDE) is an iterative and incremental software development process. Supporting the analysis and the verification of software systems developed following the MDE paradigm requires to adopt incrementality when carrying out these crucial tasks in a more optimized way.

Communicating State Machines are one of the various formalisms used in MDE tools to model and describe the behavior of distributed, concurrent and real-time reactive systems (e.g., automotive and avionics systems). Modeling the overall behavior of such systems is carried out in a modular way and on different levels of abstraction (i.e., it starts with modeling the behavior of the individual objects in the system first then modeling the interaction between these objects). Similarly, analyzing and verifying the correctness of the developed models to ensure their quality and their integrity is performed on two main levels. The intra-level is used to analyze the correctness of the individual models in isolation of the others, while the inter-level is used to analyze the overall interoperability of those that are communicating with each other.

One way to facilitate the analysis of the overall behavior of a system of communicating state machines is to build the global state space (also known as the global reachability tree) of the system. This process is very expensive and in some cases it may suffer from the state explosion problem. Symbolic execution is a technique that
can be used to construct an abstract and a bounded version of the system global state space that is known as a symbolic execution tree (SET), yet the size of the generated trees can be very large especially with big and complex systems that are composed of multiple objects. As the system evolves, one way to avoid regenerating the entire SET and repeating any SET-based analyses that have been already conducted is to utilize the previous SET and its analysis results in optimizing the process of generating the SET of the system after the change. In this thesis, we propose two optimization techniques to direct the successive runs of the symbolic execution technique towards the impacted parts of an evolving state machine model using memoization (MSE) and dependency analysis (DSE), respectively. The evaluation results of both techniques showed significant reduction in some cases compared with the standard symbolic execution technique.
Acknowledgments

All sincere praises and gratitude are due to Almighty God, the Most Gracious and the Most Merciful, who bestowed His guidance, support and knowledge upon me all the way through my life’s journey.

With all my respect, gratitude and appreciation, I would like to deeply thank my supervisor, Prof. Juergen Dingel, for his exceptional advice, encouragement, patience and support all the way through my PhD studies. It was a great privilege working with him. I also thank my PhD supervisory committee: Prof. Jim R. Cordy and Prof. Mohamed Zulkernine.

I am also very grateful to my PhD defense examination committee: Prof. Marsh Chechik, Prof. Ying Zou, Prof. Jim R. Cordy and Prof. David Lamb for reviewing my PhD work and providing valuable feedback.

Special thanks are due to all the great people of the Queen’s School of Computing who have been very helpful to me in one way or the other during the past five years, including the faculty, staff, administration and my former and current colleagues of the MASE research group and STL Lab.

Last but not least, I would like to deeply thank my beloved family for their endless love, support and encouragement.

This work was partially funded by NSERC, the Natural Sciences and Engineering
Research Council of Canada, as part of the NECSIS Automotive Partnership with General Motors, IBM Canada, and Malina Software Corp.
Statement of Originality

I, Amal Khalil, certify that all of the work described within this thesis is the original work of the author. Any published (or unpublished) ideas and/or techniques from the work of others are fully acknowledged in accordance with standard referencing practices. Earlier versions of some parts of the work reported in this thesis have previously appeared as [51, 52, 53, 54].
Contents

Abstract i

Acknowledgments iii

Statement of Originality v

Contents vi

List of Tables ix

List of Figures xi

Chapter 1: Introduction 1

1.1 Summary ................................................. 1
1.2 Motivation ................................................. 2
1.3 Thesis Statement and Scope of Research ....................... 3
  1.3.1 Thesis Statement ...................................... 3
  1.3.2 Scope of Research ..................................... 3
1.4 Contributions .............................................. 4
1.5 Organization of Thesis ..................................... 4

Chapter 2: Background 5

2.1 Modeling and Analysis in the IBM Rational Rhapsody Developer MDE Tool ............................................. 5
  2.1.1 Rhapsody Object Model Diagrams .......................... 6
  2.1.2 Rhapsody Statecharts ..................................... 8
  2.1.3 Analysis and Verification of State-based Models ........... 9
2.2 Symbolic Execution ........................................... 10
2.3 Dependence Analysis ......................................... 14
  2.3.1 Control Dependence ..................................... 15
  2.3.2 Data Dependence ....................................... 16
  2.3.3 Communication Dependence ............................... 17
2.4 Model Differencing ........................................... 18
2.5 Model Transformation ....................................... 20
2.6 Summary ......................................................... 22

Chapter 3: Related Work ........................................ 24
3.1 Two Ways to Reconcile Analysis and Evolution Efficiently 24
3.2 Analysis and Verification of Statechart-like Models .......... 28
3.3 Incremental Analysis and Verification Techniques for Evolving Programs 29
3.4 Incremental Analysis and Verification in Model-based Development . 35
3.5 Summary ......................................................... 40

Chapter 4: Standard Symbolic Execution of Rhapsody Statecharts 41
4.1 Standard Symbolic Execution of Individual Rhapsody Statecharts . 41
4.1.1 Statecharts-to-Mealy-like-Machines Transformation: SC2MLM 43
4.1.2 Symbolic Execution of Mealy-like Machine: MLM2SET .... 48
4.2 Symbolic Execution of a Model of Communicating Rhapsody Statecharts 54
4.2.1 Rhapsody Model of Communicating Statecharts-to-Communicating Mealy-like-Machines Transformation: RhapM2CMLM . . 55
4.2.2 Symbolic Execution of Communicating Mealy-like Machines: CMLM2GSET ........................................... 59
4.3 Summary ......................................................... 63

Chapter 5: Optimizing Symbolic Execution of Evolving Rhapsody Statecharts 66
5.1 Motivating Example ........................................... 66
5.1.1 Optimization Via Reuse – An Artifact-oriented Evolution Support Approach ........................................... 68
5.1.2 Optimization Via Reduction – An Analysis-oriented Evolution Support Approach ........................................... 73
5.2 Proposed Symbolic Execution Optimizations for Evolving Rhapsody Statecharts ........................................... 76
5.2.1 Memoization-based Symbolic Execution (MSE) .......... 76
5.2.2 Dependency-based Symbolic Execution (DSE) .......... 92
5.3 MSE or DSE? .................................................... 100
5.4 Summary ......................................................... 101

Chapter 6: Implementation ........................................ 102
6.1 Implementation Decisions .................................... 102
6.2 Interaction With KLEE ....................................... 106
6.3 Implementation of the MQL & TGL-based Transformation Components113
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4</td>
<td>Interaction with the IBM Rational Rhapsody DiffMerge and its Related Components</td>
<td>115</td>
</tr>
<tr>
<td>6.5</td>
<td>Summary</td>
<td>118</td>
</tr>
<tr>
<td><strong>Chapter 7: Evaluation</strong></td>
<td></td>
<td>119</td>
</tr>
<tr>
<td>7.1</td>
<td>Research Questions and Variables of Interest</td>
<td>119</td>
</tr>
<tr>
<td>7.2</td>
<td>Case Study Artifacts</td>
<td>121</td>
</tr>
<tr>
<td>7.3</td>
<td>Evaluation Setup</td>
<td>122</td>
</tr>
<tr>
<td>7.4</td>
<td>Results and Analysis</td>
<td>128</td>
</tr>
<tr>
<td>7.5</td>
<td>Threats to Validity</td>
<td>140</td>
</tr>
<tr>
<td>7.6</td>
<td>Summary</td>
<td>140</td>
</tr>
<tr>
<td><strong>Chapter 8: Conclusions and Future Work</strong></td>
<td></td>
<td>142</td>
</tr>
<tr>
<td>Bibliography</td>
<td></td>
<td>146</td>
</tr>
<tr>
<td><strong>Appendix A: Ecore meta-model of CMLMs</strong></td>
<td></td>
<td>161</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Components’ tasks and their corresponding algorithms’ steps . . . . . 87

7.1 Characteristics of the artifacts used in our evaluation . . . . . . . 123
7.2 Performance of standard SE on the base version of the four models:
   AQS, LGS, ACCS and FABP . . . . . . . . . . . . . . . . . . . . . . 123
7.3 Results of MSE and DSE on the AQS example . . . . . . . . . . . . 127
7.4 Results of MSE and DSE on the LGS example . . . . . . . . . . . . 128
7.5 Results of MSE and DSE on the ACCS example . . . . . . . . . . . 129
7.6 Results of MSE and DSE on the versions of the FABP example with
   Modification Changes . . . . . . . . . . . . . . . . . . . . . . . . . 130
7.7 Results of MSE and DSE on the versions of the FABP example with
   Addition Changes . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131
7.8 Results of MSE and DSE on the versions of the FABP example with
   Deletion Changes . . . . . . . . . . . . . . . . . . . . . . . . . . . . 132
7.9 Summary of statistical measures of the effectiveness of MSE and DSE
   for the four models: AQS, LGS, ACCS and FABP with respect to the
   savings in execution time and in the number of symbolic states and
   execution paths . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 133
7.10 Summary of statistical measures of the effectiveness of MSE and DSE for the FABP model per change type with respect to the savings in time, in the numbers of symbolic states and in the numbers of execution paths.
List of Figures

2.1 The base version $AB^{v_0}$ of an example model $AB$ of two communicating objects ........................................ 7
2.2 Symbolic execution of programs - Example 1 ........................................ 12
2.3 Symbolic execution of programs - Example 2 ........................................ 13
2.4 Dependency graph of system $AB^{v_0}$ ........................................ 18

4.1 Standard Symbolic Execution (SE) of Individual and Communicating Rhapsody Statecharts ........................................ 42
4.2 An Example of a Rhapsody Statechart and its Corresponding MLM Representation ........................................ 45
4.3 SET of the MLM Model in Figure 4.2(b) (Highlighted Levels: 1, 2 and 3) ........................................ 54
4.4 GSET($AB^{v_0}$) – Global Symbolic Execution Tree for the Example Model $AB^{v_0}$ in Figure 2.1 ........................................ 64

5.1 Two versions of an example Statechart ........................................ 67
5.2 SET($v_0$) – Symbolic Execution Tree for the Statechart in Figure 5.1(a) 68
5.3 SET($v_1$) – Symbolic execution tree for the Statechart in Figure 5.1(b) 69
5.4 A modified model version $AB^{v_1}$ and its dependency graph .................. 70
5.5 MGSET($AB^V_1$) – The parts that are different from GSET($AB^V_0$) in Figure 4.4 are highlighted (in red).

5.6 DGSET($AB^V_1$) – The pruned parts resulting from the partial exploration of transition $\text{itsA}_T1$ are grayed out.

5.7 Memoization-based Symbolic Execution (MSE) proposed architecture

5.8 An example SET with nodes 2 and 5 are marked as “needing an update” with respect to some of their outgoing branches. Nodes are labeled based on their creation order.

5.9 A demonstration of the Initial Nodes Update Strategy used in the implementation of an earlier version of MSE [53, 54]. Nodes are labelled based on their creation order and leading zeros in a node label indicate the number of reruns of the SE technique that created the node.

5.10 A demonstration of the Adapted Nodes Update Strategy used in the implementation of the current version of MSE as presented in Algorithms 7 and 8 – One-level depth exploration with marking generated nodes (if not subsumed by a pre-visited node) as “needing an update”. Nodes are labelled based on their creation order and leading zeros in a node label indicate the number of reruns of the SE technique that created the node.

5.11 Comparison between our two nodes update strategies showing (on the left) a demonstration of undetected subsumption Relation (SR) between nodes 02 and 001 resulting in having extra nodes 04 and 05.
5.12 A demonstration of a possible case of undetected subsumption relation (SR) between nodes 5 and 001 resulting in having extra nodes 02, 03, 06 and 07 .......................................................... 91

5.13 Dependency-based Symbolic Execution (DSE) proposed architecture . 93

6.1 Implementation illustration of our developed components . . . . . . . 103

6.2 An example illustrating an execution of the pseudo-code function Symb-
olicExecutionOfCode(A*, val, G*, pc) ................................. 109

7.1 An example of two differently sized SETs of the same model, each of
which has a different branch order. Note the reduction in the number
of nodes in the SET in (b) compared to the one in (a) as a result of the
subsumption relation (SR) that is detected in the SET in (b) between
node “j” and the previously visited node “l” which cannot exist if the
creation order of both nodes is reversed as it is the case in the SET in
(a). ................................................................. 136

A.1 CMLMs Ecore Meta-model – Part 1/2 ..................................... 162

A.2 CMLMs Ecore Meta-model – Part 2/2 ..................................... 163
Chapter 1

Introduction

1.1 Summary

Model Driven Engineering (MDE) is a \textit{model-centric} software engineering approach that aims at improving the productivity and the quality of software artifacts by focusing on models as first-class artifacts in place of code. MDE has been widely used for over a decade in many domains such as automotive and telecommunication industries. \textit{Iterative-incremental} development and \textit{model-based analysis} are central to MDE in which artifacts typically undergo several iterations and refinements during their lifetime that may require changes to their initial design versions. As these models evolve, it is necessary to assess their quality by repeating the analysis and the verification of these models after every iteration or refinement. This process, if not optimized, can be very tedious and time consuming.

State machines are a visual state-based formalism used to describe the behavior of reactive systems and they are found, with some variation, in every MDE tool (e.g., IBM Rational Rhapsody’s Statecharts, IBM Rational Software Architect’s UML-RT or MathWorks Simulink’s Stateflow).
1.2. MOTIVATION

Symbolic execution is a well-known analysis technique that is used to analyze the execution paths of behavioral software artifacts (e.g., programs [56] and state-based models [35, 99]). The output of the analysis is a symbolic execution tree (SET) which provides the basis for various types of analysis and verification. One of the key challenges of symbolic execution is scalability, especially when applied to big, complex artifacts where the size of the output SET becomes very large. Repeating the entire analysis even after small changes is not the best solution.

This thesis describes research investigating two complementary optimization techniques that leverage the similarities between state machine versions to reduce the cost of symbolic execution of the evolved version.

1.2 Motivation

This research is motivated by a number of facts. First, symbolic execution has been shown to be a very powerful method for the analysis of programs and there are already several commercial code analysis tools built based on it (e.g., CodeSonar [26]). Similarly, the technique has been adopted and applied in the context of state-based models (e.g., the IAR visualSTATE - Verificator [92]). Second, there is an interest from our industrial partner, General Motors Corporation (GM), to improve the model-level analysis capabilities of the IBM Rhapsody tool. Third, research on optimizing the symbolic execution of evolving programs has been recently addressed [80, 95], however to the best of our knowledge we are the first to consider such optimizations for the symbolic execution of evolving state machines.
1.3 Thesis Statement and Scope of Research

1.3.1 Thesis Statement

The efficiency and cost of symbolic execution of evolving state machines can be measurably improved using incremental analysis.

1.3.2 Scope of Research

Since we aim to use IBM Rhapsody Statecharts as the input state-based models to build a proof of concept implementation of our proposed techniques, we will only use IBM related tools to develop the methodology presented in the research plan section. Nevertheless, we believe that the same methodology can be easily applied to other contexts.

Rhapsody Statecharts offer a full set of advanced features for specifying reactive behavior. Supporting the entire set of these features is beyond the scope of this work, however, we have selected a subset that contains the most frequently used features. This includes composite and orthogonal states, inter-level transitions (also known as group transitions) and condition and junction connectors. History connectors are an example of an advanced feature that is not supported.

The prototype implementation of our proposed techniques is based on the KLEE symbolic execution engine [16]. Since KLEE supports only C code, we consider only a subset of Java or C++ action code that has the same syntax as C code, including basic assignment statements, conditional statements and iterative statements. The data types supported are the basic data types including characters, Booleans and integers. We also consider the statements used in action code for sending events.

The Statechart models to be used in the evaluation were selected in accordance
1.4 Contributions

The proposed research aims to improve the current state-of-the-art in the area of model-based analysis in an evolutionary software development environment. Our contributions specifically include (1) the development, formalization and proof-of-concept implementation of the proposed optimization techniques mentioned in Chapter 5, (2) an evaluation that will provide the results showing the benefits of our research methodology and (3) a concrete example of research that can enrich the analysis capabilities of an existing MDE tool.

1.5 Organization of Thesis

This section outlines the structure of the thesis document.

Chapter 2 gives an overview of the background. Chapter 3 discusses the related work. Chapter 4 presents our standard symbolic execution technique for individuals and communicating Rhapsody Statecharts. Chapter 5 demonstrates the optimization ideas employed in this research. Chapter 6 explains the implementation details of the proposed work. Chapter 7 outlines the evaluation procedure and presents and discusses the evaluation results. Chapter 8 concludes the work presented in this thesis and outlines future work.
Chapter 2

Background

In this chapter we provide the background information of the main technologies used in this thesis. First, we introduce the modeling context used and the types of models considered. Then, we present an overview of symbolic execution and dependence analysis. Finally, we specify the tools we use to realize model differencing and model transformation tasks.

2.1 Modeling and Analysis in the IBM Rational Rhapsody Developer MDE Tool

The IBM Rational Rhapsody Developer framework [83] is one of the commercial tools that support model-driven development, where models are the primary artifacts for software development and are iteratively refined until code can be automatically generated by the tool. The tool is heavily used in practice (e.g., in the automotive industry). It has a number of specialized models (or diagrams) used to describe a software system at different levels of abstraction. We refer only to the two types of models we use in this work: Object Model Diagram (OMD) and Statecharts. A detailed description of all features supported by these models is outside the scope of
this work. For simplicity, only a limited set of these features is used in the illustrative example shown in Figure 2.1.

2.1.1 Rhhapsody Object Model Diagrams

The Object Model Diagrams (OMD) are used to specify the structure and the static relationships of the classes and objects in a software system. Rhapsody’s OMDs serve, to some extent, the same purpose as the UML’s Composite Structure Diagrams, Class Diagrams and Object Diagrams. The top level object in an OMD is usually a composite class that represents the root class of the system and which is also considered as the system container. The IBM Rational Rhapsody code generator directly translates the elements and relationships modeled in an OMD into source code in a specified high-level language (e.g., C, C++, C#, Java or Ada). Figure 2.1(a) shows a system $AB$ that is composed of two communicating objects: $itsA$ and $itsB$. Object $itsA$ is of class $A$ that has an integer attribute $x$ that is initialized to 0 and a protocol of a set of three messages: $e1$, $e2$ and $e3$. Similarly, object $itsB$ is of class $B$ that has an integer attribute $y$ that is initialized to 5 and a protocol of three messages: $e4$, $e5$ and $e6$. Message $e1$ has an integer argument $e1Arg1$ and message $e5$ has an integer argument $e5Arg1$. The behavior of each object is modeled by the Rhapsody Statecharts shown in Figures 2.1(b) and 2.1(c), respectively.
2.1. MODELING AND ANALYSIS IN THE IBM RATIONAL RHAPSODY DEVELOPER MDE TOOL

(a) The Object Model Diagram (OMD) of our simple model $AB$

(b) The Statechart of object itsA in (a)

(c) The Statechart of object itsB in (a)

Figure 2.1: The base version $AB^{V0}$ of an example model $AB$ of two communicating objects
2.1.2 Rhapsody Statecharts

Rhapsody Statecharts\(^1\) (also known as Harel’s Statecharts [41]) are a visual state-based formalism implemented in the IBM Rational Rhapsody framework to describe the behavior of reactive systems. They extend conventional Mealy Machines – a type of Finite State Machines (FSMs) that perform their action only on firing transitions – with advanced features such as hierarchical states, orthogonal regions and action code in states and transitions. An action is a reactive behavior associated with a state or a transition that can be carried out when, for instance, entering a state, leaving a state, or when firing a transition. Examples of these actions include updating the values of the machine variables or producing an output to be sent to the environment via events. Rhapsody allows the use of C, C++ or Java to express these actions. A transition is defined in the general format: \textit{event[guard]/action}, that specifies the firing event of the transition, an optional guard condition and an optional action code. Events are stimuli delivered to the Statechart from the environment or other objects in the system and they can have optional user-defined parameters (also called arguments). Timeout events are internal events which can be generated using timers within the Statecharts. In contrast with the machine variables, which are considered as global variables that can be accessed anywhere within the Statechart, event arguments are considered as local variables that have a limited scope that is restricted only to the guard and the action code of the transitions that receive their event. A guard is an optional Boolean expression that can be defined over machine variables and events arguments. A transition is fired when its source state is active, its corresponding event is received, its guard evaluates to true and no higher priority transition is enabled.

\(^1\)The term “Statechart” is used exchangeably with the term “State Machine” in this document.
When a transition is fired, its action is executed and, consequently, the system goes to the transition’s target state. Besides the standard assignment and conditional statements that can be used in describing the action code of a transition, there is a domain-specific send event statement which is used to generate an output event to be sent to a specified object. An example of such statement is shown in the action Java code of transition itsA_T2 in Figure 2.1(b), which is \texttt{itsB.gen(new e5(5))}; that is used to generate event \texttt{e5} with the event’s argument \texttt{e5Arg1} set to 5 to object \texttt{itsB}. Accordingly, the argument value of the event \texttt{e5} is stored in a variable called \texttt{params} that can be accessed using the statement \texttt{params.e5Arg1}.

Apart from whether the given Statecharts satisfy their specification, there are a list of basic and intuitive design rules any given Statechart model should satisfy. This includes the reachability of states, the satisfiability of guards and the availability of input events. The task of verifying whether these rules are satisfied looks very simple for Statecharts like the ones in Figures 2.1(b) and 2.1(c) but it gets more complex as these models get larger and when more advanced modeling features are used, which is the typical case when building industrial real-time systems.

### 2.1.3 Analysis and Verification of State-based Models

Existing analysis methods to analyze state-based models are divided into two categories: \textit{model-level analysis} and \textit{code-level analysis}. The motivation of the code-level analysis approaches stems from the fact that these models are executable and we can easily generate their corresponding code and use some of the model checkers for C/C++ or Java code or one of the static analysis tools would help in finding possible problems at the code-level. However, it is difficult for a general static analysis tool to
2.2. SYMBOLIC EXECUTION

figure out the high-level intent of any piece of code we feed to it and therefore they are typically incapable of revealing model-level problems (e.g., unreachable states and dead ends). Also, it is not always the case that we would be able to generate the code, especially with incomplete specifications. Additionally, the results provided from a code-level analysis are typically more detailed which usually makes them less intuitive for designers and modellers. Other challenges include the difficulty of tracing back the errors generated in the code to its source at the model level and to adapt existing analysis tools (e.g., model checkers, symbolic execution engines or static analysis tools) to handle domain-specific statements found in the generated code. For these reasons, we believe that model-level analysis is the most appropriate way to handle this task. Therefore, the motivation of our research is to highlight the importance of the types of analysis or verification to be carried out on the model level.

2.2 Symbolic Execution

Symbolic execution is a program analysis technique that allows the execution of programs in a parametric way using symbolic inputs to derive precise characterizations of their properties and their execution paths. It has first introduced in the 1970’s by Lori A. Clarke [25] and James C. King [56] for program testing. Since 2003, much research effort has been devoted to improve the effectiveness, the efficiency and the applicability of the traditional technique [18, 55, 96].

The main idea is to substitute program inputs with symbolic values and then execute the program parametrically such that: 1) the values of all program variables are computed as symbolic expressions over the symbolic input values, and 2) all possible execution paths are explored. The result from the symbolic execution of a
program is a tree-based structure called symbolic execution tree (SET). The nodes of a SET represent the symbolic program states and the edges represent the transitions between these states. Each program symbolic state consists of a program location, the set of program variables and their symbolic valuations, and a path constraint (PC) which is the conjunction of all the logical constraints collected over the program variables to reach that program location. The feasible paths of a SET characterize all the distinct execution paths of a program. Decision procedures and SMT solvers are used to check the satisfiability of each path constraint (PC).

The resulting symbolic execution tree forms the basis for many program analysis, verification and testing activities (e.g., bug finding, dead code detection, invariants checking, program equivalence checking, regression analysis and test case generation). For instance, it can be used to optimize the testing process by generating the minimum number of test cases required to inspect all feasible program paths. Figure 2.2(b) (resp., 2.2(d)) shows the SET of the sample code in Figure 2.2(a) (resp., 2.2(c)). The exploration is carried out in a depth-first manner. Examples of other exploration strategies are breadth first, random or heuristic based exploration. Solving the path constraints of the leaf nodes of each SET will provide us with the sample input values that can be used to test each feasible path. Program locations that do not have any corresponding symbolic states in the program SET indicate dead code. For example, program locations 8-9 in the code snippet `foo V2` are dead code.

One of the biggest challenges in symbolic execution is the path explosion problem especially for large programs and programs with unbounded loops and recursion. Examples of proposed solutions include: setting upper bounds for loops, summarizing loop effects, employing some state matching criteria (e.g., state subsumption)
2.2. SYMBOLIC EXECUTION

(a) Code Snippet of foo_V1

```plaintext
def foo_V1(x, y):
    if x > y:
        x = x - y;
    else:
        x = y - x;
    if x > 0:
        x++;
    else:
        x--;
    return x;
```

(b) SET of foo_V1

(c) Code Snippet of foo_V2

```plaintext
def foo_V2(x, y):
    if x > y:
        x = x - y;
    else:
        x = y - x;
    if x > 0:
        x++;
    else:  // dead code
        x--;  // dead code
    return x;
```

(d) SET of foo_V2

Figure 2.2: Symbolic execution of programs - Example 1
2.2. SYMBOLIC EXECUTION

Figure 2.3: Symbolic execution of programs - Example 2

for pruning redundant paths and reducing the state space, using heuristics for path finding to achieve some user-defined coverage criteria or dividing a program into independent parts and run the symbolic execution for each part in parallel [96]. For example, Figure 2.3(b) shows a finite version of the SET of the code snippet in Figure 2.3(a) [18]. This finitized SET version is obtained by performing a state matching technique called subsumption checking to determine when a symbolic state is revisited and hence pruning the exploration at this point. For any given two symbolic states \( SS_i \) and \( SS_j \), \( SS_j \) is subsumed by \( SS_i \) if both symbolic states represent the same program location and if the concretization of \( SS_j \) is included in the concretization of \( SS_i \). For example, in Figure 2.3(b), the symbolic state \( SS_7 \) is subsumed by the symbolic state \( SS_4 \) and hence the exploration of the symbolic state \( SS_7 \) can be avoided.

(a) Code Snippet of `testme_inf` [18]  
(b) A finite SET of `testme_inf`
Other challenges include: 1) the limitation of constraint solvers in solving very complex and non-linear constraints and 2) the inability to handle external library calls. Examples of proposed solutions include: Concolic Symbolic Execution (an variant of the traditional technique that enables the use of concrete values for unmanagable symbolic expressions) [17, 38], performing constraints simplification and providing models to simulate or abstract the behavior of external modules.

Examples of existing symbolic execution tools are jCUTE and Java PathFinder (JPF) for Java, EXE and its successor KLEE for C and C++, and Pex for the .NET Framework.

Symbolic execution is not restricted only to programs; it has been also applied to a variety of state-based models including Input Output Symbolic Transition Systems [35], Statecharts [89], UML State Machines [6] and recently UML-RT State Machines [99].

2.3 Dependence Analysis

Dependence analysis has been extensively studied in research areas such as program refactoring, parallelization and optimization. It also supports many activities in software maintenance such as comprehension, slicing, impact analysis and regression test reduction. Dependence analysis has been primarily studied in the context of programming languages where dependences are defined between statements and variables using their Control Flow Graphs (CFGs). Eventually, the same concept has been adapted for other types of software artifacts to capture potential interactions within such artifacts, e.g., Extended Finite State Machines (EFSMs) [3, 60, 81] where interactions are defined between transitions since they are the “active” elements of
2.3. DEPENDENCE ANALYSIS

these types of models. Examples of dependences are control dependence, data dependence and communication dependence.

In our setting, we transform Rhapsody Statecharts into an EFSMs-like representation called Mealy-like Machines (MLMs) (the formal definition on an MLM is presented in Subsection 4.1.1), therefore we can adopt the definitions of dependencies from conventional EFSMs. In the sequel, we will refer to a set of definitions for computing these dependences as adopted from [22] based on the work presented in [3, 60, 81].

2.3.1 Control Dependence

Classical definitions of control dependence in sequential programs state that a statement $s_j$ is control dependent on a statement $s_i$ if statement $s_i$ is a conditional that affects the execution of statement $s_j$. For example, in an if-then-else construct, statements in the two branches of the conditional statement are control dependent on the predicate. Since state-based formalisms differ from sequential programs, some adapted definitions are given to capture the control dependence in such formalisms. The one that applies to state machines with possible non-termination is called Non-Termination Sensitive Control Dependence (NTSCD) (see Definition 2), which is given in terms of maximal paths (see Definition 1).

**Definition 1. (Maximal Path).** A path in a state machine is defined as a finite sequence of consecutive transitions that are organized such that no state is visited more than once. A path is called maximal if it terminates in an end state (state with no outgoing transitions) or it terminates in a cycle or it reaches the initial state again. A cycle is a set of transitions that forms a strongly connected component.
Definition 2. (Non-Termination Sensitive Control Dependence (NTSCD)).

\[ t_1 \xrightarrow{\text{NTSCD}} t_j \]

means that \( t_j \) is non-termination sensitive control dependent on a transition \( t_1 \) iff:

1. for all paths \( \pi \in \text{MaximalPaths}(\text{targetState}(t_1)) \), the \( \text{sourceState}(t_j) \) belongs to \( \pi \);

2. \( t_1 \) has at least one sibling \( t_k \) and there exists a path \( \pi' \in \text{MaximalPaths}(\text{sourceState}(t_k)) \) such that \( \text{sourceState}(t_j) \) does not belong to \( \pi' \);

where \( \text{MaximalPaths}(s) \) is the set of paths that have \( s \) as the source state of the first transition on each path and \( t_k \) is said to be a sibling transition of \( t_1 \) if \( \text{sourceState}(t_k) = \text{sourceState}(t_1) \).

2.3.2 Data Dependence

Typical data dependence definitions are given in terms of variable definitions and uses. In the context of state machines, a variable is used on a transition if its value appears in the guard of the transition or appears on the right side of an assignment-statement or in the Boolean expression of an if-statement, or in an output-statement in the action code of the transition. A variable is defined on a transition if it appears on the left hand side of an assignment-statement in the action code of the transition. Based on this, the following definition is given [3, 22].
Definition 3. (Data Dependence (DD)). $t_i \xrightarrow{\text{DD}} t_k$ means that $t_k$ is data dependent on $t_i$ with respect to variable $v$ if:

1. $v \in \text{Def}(t_i)$, where $\text{Def}(t_i)$ is the set of variables defined by the action code of transition $t_i$;

2. $v \in \text{Use}(t_k)$, where $\text{Use}(t_k)$ is the set of variables used in the guard or the action code of transition $t_k$;

3. there exists a path in the state machine from $t_i$ to the $\text{targetState}(t_k)$ along which $v$ is not modified.

2.3.3 Communication Dependence

Two transitions are called communication dependent iff: 1) each transition belongs to a different Mealy-like Machine (MLM) (the formal definition on an MLM is presented in Subsection 4.1.1) and 2) one transition is triggered by an event that is generated by the action code of the second transition [3, 22].

Definition 4. (Communication Dependence (COMD)). Given two transitions $t_i$ and $t_k$, $t_i \xrightarrow{\text{COMD}} t_k$ means that $t_k$ is communication dependent on $t_i$ iff:

1. $t_i \in T(\text{MLM}_i)$ and $t_k \in T(\text{MLM}_k)$ where $i \neq k$;

2. $e(t_k)$ – the event (or trigger) of $t_k$ is generated by the action code of $t_i$.

The detailed steps for computing the maximal paths, the non-terminating sensitive control dependence (NTSCD) and the data dependence (DD) in a Mealy-like Machine as well as the communication dependence (COMD) in a model of communicating Mealy-like Machines are shown in Algorithm 1, Algorithm 2, Algorithm 3 and
2.4. MODEL DIFFERENCING

Algorithm 4, respectively. In our work we consider only data and communication dependences, however we listed the others for completeness. Figure 2.4 depicts the data and communication dependences that exist between the Statecharts describing the behavior of the two objects \textit{itsA} and \textit{itsB} of System \textit{AB} shown in Figure 2.1.

![Figure 2.4: Dependency graph of system AB](image)

2.4 Model Differencing

Model differencing is the practice of identifying and locating the changes between different versions of a model. As models evolve, keeping track of their changes is essential for their maintainability during their lifetime. Existing approaches and tools for model differencing depend on some similarity-based matching criteria to guide the search process [87]. Some of them consider the syntactic similarities between models elements and others try to search for the semantic similarities as well [73]. In our work, we use the IBM Rational Rhapsody DiffMerge tool to find the differences between Rhapsody Statecharts. The tool compares Rhapsody projects that belong to the same ancestor (i.e., they are all models originating from the same base version) and generates a difference report with the differences between the matched elements in each project. As we notice, the tool depends on some globally unique identifiers
Algorithm 1 Maximal Paths Computation

Inputs:
1: An MLM representation of a Rhapsody Statechart model \(MLM = (S, T, V, s_0)\) that is defined in terms of states \(S\), transitions \(T\), variables \(V\) and an initial state \(s_0 \in S\)
2: A mapping, \(\text{outTrans} : S \rightarrow \mathcal{P}(T)\), of states to their outgoing transitions

Output:
1: \(\text{maximalPaths}(MLM)\)

Steps:
1: \(\text{endStates}(MLM) \leftarrow \emptyset\)
2: \(\text{cycles}(MLM) \leftarrow \emptyset\)
3: \(\text{maximalPaths}(MLM) \leftarrow \emptyset\)
4: \(\text{for all } (s \in S) \land (\text{outTrans}(s) = \emptyset) \text{ do}\)
5: \(\text{endStates}(MLM) \leftarrow \text{endStates}(MLM) \cup \{s\}\)
6: \(\text{end for}\)
7: //Explore the statechart in depth-first-search manner
8: Push the initial state \(s_0\) into a stack \(st\)
9: Mark \(\text{outTrans}(s_0)\) as unexplored
10: while \(st\) is not empty do
11: \(\text{currentStateToProcess} \leftarrow \text{topElement}(st)\)
12: \(\text{for all } (t_o \in T) \land (t_o \in \text{outTrans(\text{currentStateToProcess}))} \land (t_o\text{ is still unexplored}) \text{ do}\)
13: \(\text{if } (\text{targetState}(t_o) = s_0) \lor (\text{targetState}(t_o) \in \text{endStates}(MLM)) \lor (\text{targetState}(t_o) \in \text{cycles}(MLM)) \text{ then}\)
14: Report a maximal path \(\pi \leftarrow st + \text{targetState}(t_o)\)
15: \(\text{maximalPaths}(MLM) \leftarrow \text{maximalPaths}(MLM) \cup \{\pi\}\)
16: Mark \(t_o\) as explored
17: \(\text{else if } \text{targetState}(t_o) \in st \text{ then}\)
18: Report a cycle \(\mathcal{O} \leftarrow st\text{.subList(indexOf(targetState(t_o)), size(st))) + targetState(t_o)}\)
19: \(\text{cycles}(MLM) \leftarrow \text{cycles}(MLM) \cup \{\mathcal{O}\}\)
20: Report a maximal path \(\pi \leftarrow st\text{.subList(0, size(st))} + \text{targetState}(t_o)\)
21: \(\text{maximalPaths}(MLM) \leftarrow \text{maximalPaths}(MLM) \cup \{\pi\}\)
22: Mark \(t_o\) as explored
23: \(\text{else}\)
24: Push \(\text{targetState}(t_o)\) into \(st\)
25: Mark \(\text{outTrans(\text{targetState}(t_o))}\) as unexplored
26: Mark \(t_o\) as explored
27: Break
28: \(\text{end if}\)
29: \(\text{end for}\)
30: Pop \(\text{currentStateToProcess}\) from \(st\)
31: \(\text{end while}\)

which are assigned internally to model elements to match between the elements in each model and then to find the differences between the matched ones.
2.5. MODEL TRANSFORMATION

Algorithm 2 Non-terminating Sensitive Control Dependence (NTSCD) Computation

Inputs:
1. An MLM representation of a Rhapsody Statechart model $MLM = (S, T, V, s_0)$ that is defined in terms of states $S$, transitions $T$, variables $V$ and an initial state $s_0 \in S$
2. $\text{maximalPaths}(MLM)$

Output:
1. A mapping, $CD : T \rightarrow \mathcal{P}(T)$, of transitions to their control dependent transitions

Steps:
1. // Finding $\text{maximalPaths}(T)$
2. for all $t \in T$ do
3. for all $\pi \in \text{maximalPaths}(MLM)$ do
4. if $\pi\.indexOf(\text{targetState}(t)) = \pi\.indexOf(\text{sourceState}(t)) + 1$ then
5. $\text{maximalPaths}(t) \leftarrow \text{maximalPaths}(t) \cup \{\pi\}$
6. end if
7. end for
8. end for
9. // Finding $CD(T)$
10. for all $(t_i \in T) \land (\text{siblingTransitions}(t_i).\text{size()} \geq 1)$ do
11. $CD(t_i) \leftarrow \emptyset$
12. for all $(s \in S) \land (s \in \text{maximalPaths}(t_i))$ do
13. if $(t_k \in T) \land (t_k \in \text{siblingTransitions}(t_i)) \land (\exists \pi \in \text{maximalPaths}(t_k)) \land (s \notin \pi)$ then
14. for all $(t_o \in T) \land (t_o \in \text{outTrans}(s))$ do
15. $CD(t_i) \leftarrow CD(t_i) \cup \{t_o\}$
16. end for
17. end if
18. end for
19. end for

2.5 Model Transformation

Model transformation is the technology that is used in the area of MDE to convert models to other software artifacts (e.g., code) [87]. When the transformation is carried out to convert a source model to a target model and both models conform to the same metamodel, we call it an endogenous transformation. This type of model transformation is used to perform tasks such as model refactoring and optimization in general. On the other hand when the transformation is carried out to convert a source model to a target model and both models conform to different metamodels, we call this an exogenous transformation. This type of transformation is used handle tasks such as code generation, reverse engineering and migration. In our setting, we
Algorithm 3 Data Dependence (DD) Computation in an MLM

Inputs:
1. An MLM representation of a Rhapsody Statechart model $MLM = (S, T, V, s_0)$ that is defined in terms of states $S$, transitions $T$, variables $V$ and an initial state $s_0 \in S$
2. A mapping, $\text{Use} : T \rightarrow \mathcal{P}(V)$, of transitions to the set of variables each transition uses
3. A mapping, $\text{Def} : T \rightarrow \mathcal{P}(V)$, of transitions to the set of variables each transition defines

Output:
1. A mapping, $\text{DD} : T \rightarrow \mathcal{P}(T)$, of transitions to their data dependent transitions with respect to $V$

Steps:
1. for all $t \in T$ such that: $(\exists v \in V) \land (v \in \text{Use}(t))$ // i.e., for all transitions that use at least one variable do
2. for all $(v \in V) \land (v \in \text{Use}(t))$ // i.e., for all the variables that are used in $t$ do
3. // Apply backward depth-first-search to find the first transitions defining the variable $v$
4. Push the $\text{sourceState}(t)$ into a stack $st$
5. while $st$ is not empty do
6. $\text{currentStateToProcess} \leftarrow \text{topElement}(st)$
7. for all $(t_x \in T) \land (t_x \in \text{incomingTransition}(\text{currentStateToProcess})) \land (t_x \text{ has not yet been explored})$ do
8. $\text{DD}(t_x) \leftarrow \emptyset$
9. if $v \in \text{Def}(t_x)$ // i.e., $v$ is defined in $t_x$ then
10. $\text{DD}(t_x) \leftarrow \text{DD}(t_x) \cup \{t\}$ w.r.t. $v$
11. Mark $t_x$ as explored
12. else
13. Mark $t_x$ as explored
14. Push the $\text{sourceState}(t_x)$ into $st$
15. Break
16. end if
17. end for
18. end while
19. end for
20. end for

Algorithm 4 Communication Dependence (COMD) Computation

Inputs:
1. A global model/system of $n$ asynchronously communicating Mealy-like-Machines $M = (MLM^M, Q^M, s_0^M, V^M, V_0^M)$ with $MLM^M = (MLM_1, MLM_2, ..., MLM_n)$ and $MLM_i = (S_i, V_i, E_i, E_A_i, T_i, s_{0i}, V_{0i})$ is the $i^{th}$ Mealy-like Machine in $M$
2. A mapping, $\text{Send} : T \rightarrow \mathcal{P}(E^*)$, of all transitions in $MLM^M$ to the set of output events each transition may generate as a result of evaluating/executing their action code

Output:
1. A mapping, $\text{COMD} : T \rightarrow \mathcal{P}(T)$, of transitions to their communication dependent transitions

Steps:
1. for all $MLM_i \in MLM^M$ do
2. for all $t_x \in T_i$ do
3. $\text{COMD}(t_x) \leftarrow \emptyset$
4. for all $t_y \in T$ do
5. if $e(t_x) \in \text{Send}(t_y) \land MLM(t_x) \neq MLM(t_y)$ then
6. $\text{COMD}(t_x) \leftarrow \text{COMD}(t_x) \cup \{t_y\}$
7. end if
8. end for
9. end for
10. end for
need to perform exogenous transformations to transform Rhapsody Statecharts to our internal MLM representation such that we can symbolically execute the model. Therefore, we use the model transformation technologies offered by the IBM Rational Rhapsody Developer RulesComposer Add On. The add-on is built based on the Eclipse Modeling Framework (EMF) and it provides the infrastructure required to perform common MDE services such as model and metamodel manipulation, model-to-model transformation and model-to-text generation. The language used to define model transformations is called Model Query Language (MQL) and the language used to define text templates is called Text Generation Language (TGL) and they are both imperative and proprietary languages. MQL is also used to launch text generations and to define the expressions used within TGL.

2.6 Summary

In this chapter we have presented the background information. First, we have introduced the modeling context used and briefly described the two types of models considered. Namely, the Object Model Diagrams for describing the structural aspects of the system and the Statecharts for describing the behavioral aspects of the system. We also have showed the basics of two analysis techniques: symbolic execution and dependence analysis. We have introduced symbolic execution of programs with examples showing some of the most common applications of the technique and explaining the state matching criteria used to reduce the state space of programs with loops. Additionally, we have provided the definitions and the algorithms used to compute the three types of dependences that exist between transitions in EFSMs-like representations. Last, we have defined and specified the model differencing and the model
transformation tools used in the context of our work.
Chapter 3

Related Work

In this chapter we first introduce a general description of our classification of the two most common ways used in the literature to reconcile analysis and evolution efficiently in software development: reused-based (artifact-oriented) reconciliation and reduction-based (analysis-oriented) reconciliation [29]. Second, we discuss related work in the context of the proposed research and based on the two classified reconciliation methods. We review studies that (1) propose approaches for the analysis and the verification of Statecharts-like models and highlight those that are based on symbolic execution, (2) propose incremental analysis or verification techniques for evolving programs and highlight those which are built specifically to improve the efficiency of symbolic execution, and (3) motivate and realize the concept of incremental verification in the context of model-based development.

3.1 Two Ways to Reconcile Analysis and Evolution Efficiently

Analysis and verification are crucial and expensive activities in the development of real-time and safety-critical software systems. The prerequisite preparation required to run these activities is not trivial in terms of the time it costs and the technologies
it uses. The output from such prerequisite preparation process is a special type of artifact that represents one or more views of the system under development in the input language of the analysis or the verification method to be used. The generation of these analysis-related artifacts is usually done using some transformation or translation technologies that allow the semantic mapping between the artifacts used to build the system and those that are used to analyze and verify it. The output from running the analysis or verification process on these auxiliary or intermediate artifacts is used to verify or disprove the correctness of the analyzed properties. Classical analysis methods were focused on analyzing source code related artifacts (both statically and dynamically) and on the intra-level as well as the inter-level. The intra-level is used to analyze the correctness of the individual artifacts in isolation of the others, while the inter-level is used to analyze the overall interoperability of those models that are communicating with others. With the emergence of model-driven engineering and agile methods, analysis techniques have been extended to analyze the more abstract architecture artifacts (both structural and behavioral). Examples of static analysis-related artifacts and their usages include 1) control flow graphs for static analysis and optimization [57], 2) dependency graphs for change impact analysis and slicing [3] and for regression test selection [10], 3) theorem prover formalization for formal verification [9], and 4) symbolic execution trees for reachability analysis, invariant checking and test case generation [77, 35]. Examples of dynamic analysis-related artifacts include run-time and simulation execution traces which can be used for performance and configuration analysis and run-time errors detection [72].

With the iterative and incremental development of modern development life-cycles, software systems are continuously evolving. Maintaining and analyzing an
ever-changing system is essential for its quality and continuity, yet it is a very expensive process and requires considerable efforts. Looking for effective techniques to optimize the analysis activity of these systems as they evolve is vital and it starts with a better identification and understanding of the subject change (i.e., the type of the change and its location) as well as the type of the analysis to be carried out and of course taking into consideration whether a previous analysis of the same type has been conducted or not. For instance, some refactoring changes such as consistent renaming should not invalidate a previous analysis result but would require a refactoring of the previously created analysis-related artifacts to maintain the correctness of the result. On the other hand, some corrective, modification, addition or deletion changes may or may not invalidate a previous analysis result and hence require a complete or partial regeneration of the analysis-related artifacts and rerunning the analysis on the newly generated analysis-related artifacts.

In the sequel we discuss two different scenarios where optimization opportunities (or reconciliations) may exist.

1. **Reused-based (Artifact-oriented) Reconciliation:** The first scenario is the case where analysis-related artifacts are regularly retained (i.e., not discarded) and they can be used to run different types of analyses (e.g., symbolic execution trees and dependency graphs). In this case there is an opportunity to optimize the process of re-generating these analysis-related artifacts after a change by reusing the unaffected parts of the already existing ones and regenerating only the change-related parts. The output from this process are complete updated analysis-related artifacts which are consistent with the system in its most recent state (i.e., after the change) and can be used to run
any of its related analyses. By distinguishing the newly generated parts from the reused parts, we can also optimize the process of re-running any previously conducted analyses such that only the newly generated parts of these analysis-related artifacts are considered. Incremental model transformation [43], extreme model checking [44], memoized symbolic execution [95] and regression model checking [93] are all examples of existing approaches that are based on this reconciliation strategy.

2. Reduction-based (Analysis-oriented) Reconciliation: The second scenario is the case where analysis-related artifacts are discarded but there is certainty about the validity of the system in its previous state (i.e., before the change) with respect to a specific type of analysis and the goal is to find the most effective way to re-run the same type of analysis on the system in its current state (i.e., after the change). In this case, one possible opportunity for optimization is to rerun the analysis in a reduced mode or context taking into consideration only the changed parts and their dependencies. The output from this process are partial, yet sufficient, analysis-related artifacts that can be used to run only the same type of analysis and to conclude the overall correctness of the system after the change even though the new analysis does not cover the whole system. Existing work for incremental equivalence checking [5, 33, 39, 40, 78, 88], directed incremental symbolic execution [80] and incremental property checking [94] employ this idea.
3.2 Analysis and Verification of Statechart-like Models

**Model Checking Based Approaches:** The majority of existing analysis methods for Statechart-like models (e.g., UML state machines, UML-RT, STATEMATE and Stateflow) require a translation to the input language of existing formal verification tools, mainly model checkers such as SPIN [75] or SMV [24]. Most of the time, the translation process is non-trivial and it restricts the support of the key features of the model to those supported by the model checker to be used and hence makes integration and use with modern MDE tools difficult. Another category of approaches analyze the C or Java code generated from these models and use one of the model checkers for these languages (e.g., Java Pathfinder [74] or CBMC — a Bounded Model Checker for C and C++ [71]). On the other hand, there are fewer approaches that intend to provide domain-specific analysis capabilities by developing domain-specific model checkers that are tailored to the modeling language of interest and hence support more sophisticated features and reduce the semantic gap between the modeling language and the specification language of a general-purpose model checker (e.g., the verification methods implemented within the FUJABA Real-Time tool [37] or the VIS model checker in a very early version of the Rhapsody tool [85]).

**Symbolic Execution Based Approaches:** Besides the model checking-based approaches, there are several approaches that adopt the symbolic execution analysis technique of programs and use it in the context of state-based models including:

- Modechart specifications (a variation of Harel Statecharts that incorporates timing constraints in the models) for test sequences generation [66];

- Labelled Transition Systems (LTSs) for test case generation [34] and test case
3.3. INCREMENTAL ANALYSIS AND VERIFICATION TECHNIQUES FOR EVOLVING PROGRAMS

selection [46];

- Input Output Symbolic Transition Systems (IOSTSs) [35] for test purpose definition;

- STATEMATE statecharts [89] and UML State Machines [6] for verifying temporal properties of the subject models;

- Simulink/Stateflows [76] for analysis and test case generation of flight software using Java PathFinder and Symbolic PathFinder;

- UML-RT State Machines [99] to support a variety of analyses for this type of models.

Our research belongs to this category of approaches. We use symbolic execution to support the analysis of a slightly different type of state-based models which are Rhapsody Statecharts. Our technique differs from others in that 1) it reuses the off-the-shelf symbolic execution engine KLEE [16] and 2) it supports the incremental analysis of the subject models as they evolve.

3.3 Incremental Analysis and Verification Techniques for Evolving Programs

A comprehensive survey of existing work in this area of research is outside the scope of our study. Therefore, we only provide examples of two categories of work that use similar ideas as ours in building their incrementality approaches, namely reuse-based (or artifact-oriented) incremental approaches and reduction-based (or analysis-oriented) incremental approaches. We noticed that the majority of existing work belongs to the first category.
Reused-based (Artifact-oriented) Incremental Approaches: In the first category of approaches incrementality is achieved by reusing analysis-related artifacts. Examples of analysis-related artifacts that have been reused in the literature are state space representations (e.g., abstract reachability trees and symbolic execution trees) [44, 62, 93, 27], control flow graphs and dependency graphs [63, 27], constraint solving results [16, 95, 91, 47, 4], counter example traces [8, 94], abstraction precisions [7], some compressed representation in the form of hash codes of verified properties [59], and function summaries [86].

In analogy to the idea of extreme programming that advocates the use of regression testing side-by-side with stepwise software development, the idea of extreme model checking is proposed in [44] to support regression verification. The proposed approach extended the lazy abstraction algorithm implemented in the unbounded model checker BLAST for C programs to support incremental checking of safety properties. This is done by caching and reusing the abstract reachability trees resulting from a previous model checking run. A similar approach is presented in [62] to perform an incremental state-space exploration for evolving Java programs by recording and reusing the state-space resulting from a previous model checking run to speed up the exploration process of subsequent program versions. The realization of this approach is built in the context of the Java PathFinder (JPF) symbolic execution framework. Inspired by this latter work, a technique of regression model checking is presented in [93]. Two modes of operations are defined: recording mode and pruning mode. Recording mode is used to compute and store the state space and state coverage resulting from exploring a new program location. When a change is made to a program whose state space before the change has previously been computed and recorded, the model checker operates
in the pruning mode where a change impact analysis is used to identify the impacted elements in the old state space that need to be updated. Only program locations related to these impacted elements are going to be re-explored during the recording operation mode of the model checker.

The idea of memoizing and reusing the path constraints found in the symbolic execution tree (SET) of a Java program to optimize the symbolic execution of the program’s next revision is introduced in [95]. Reusing a previous SET will avoid the re-execution of common paths between the new version of a program and its previous one and hence speed up the symbolic execution of the new version.

The idea of caching and reusing constraint solving proofs in the context of symbolic execution is first used in the implementation of the symbolic execution engine KLEE [16]. The goal is to reduce the number of calls to the constraint solver while running the symbolic execution and hence mitigating some of the scalability issues related to this technique. The same idea has been further explored and applied in the context of other symbolic execution engines such as JPF [91, 47], Crest and JBSE [4]. In all such work, a repository of constraint proofs is created that is augmented with a number of powerful simplification techniques to facilitate the optimal reuse of constraints including slicing, normalization and canonicalization. This constraint reusing mechanism can be applied to optimize the symbolic execution of similar or different programs.

The idea of reusing previous model checking outcomes including the counter-examples of failed properties (also called verification witnesses) is presented in [8] to optimize the following re-verification runs and in [94] to perform incremental property checking. A key goal in such approaches is to reduce the number of properties to be
re-checked after an update has been made to the set of properties required for a given program.

Different types of analysis can be performed in the context on model checking (e.g., explicit-value analysis and predicate analysis). Each type needs to be explicitly defined and supplied to the model checker to determine the scope of the analysis (e.g., location-scoped, function-scoped or global-scoped) and to guide the abstraction process followed by the model checker to create a proper and sufficient abstraction of the model state space. The definitions of such analysis-based requirements are called program precisions which can be recorded and reused for speeding up the subsequent analysis of an evolving program as presented in [7]. Intuitively, the effectiveness of such an approach depends on the class of program changes introduced in a program’s next revision which may or may not require refinement of old precisions.

The idea of saving and reusing a compressed representation (hash codes) of the parts of the model parse tree relevant to a given property is presented in [59] for efficient regression verification. When a change is committed in the system, a new hash code is computed and is compared with the hash code value generated from the previous verification run. If both values are identical then there is no need to re-verify the given property, otherwise a re-verification is needed.

In [86], an incremental upgrade checking approach is presented and implemented in the context of bounded model checking (BMC). It reuses the function summaries generated from model checking a previous program version to check the validity of their corresponding modified functions in the new program version. If such validity check fails for a given modified function, an upgrade check is performed to create a new valid summary to replace the invalid one.
A whole program path profiling technique is presented in [61] that uses execution traces collected from running a system test suite to generate a compact graphical representation in a form of directed acyclic graph (DAG) of the control flow of the entire system including inter-procedural paths. The generated DAG can be used to perform more precise program analysis such as dynamic impact analysis [64]. An incremental version of such dynamic impact analysis is presented in [63] which reuses the DAG of a previous release of a system to incrementally maintain the DAG of the system as it evolves (i.e., a change is made to its behavior or its test suite).

In [27], an incremental approach for inter-procedural analysis of safety properties is presented. The main idea depends on reusing previous derivation graphs resulting from the synchronous product of program control flow graphs and a checking automaton describing some safety properties to speed up the re-analysis of the program after a change has been made to its specification or to its safety properties.

**Reduction-based (Analysis-oriented) Incremental Approaches:** On the other hand, in the second category of approaches incrementality is achieved by reducing the scope of the new analysis to focus mainly on the parts of the system that are impacted by the change given that 1) some analysis has already been conducted on the system before the change and 2) this analysis proved the system’s correctness with respect to some properties. Examples of these approaches are built on the assumption that the previous version of the program is verified and is correct and hence the task of re-verifying the program after the change is either limited to the impacted parts [80, 84, 94] or is reduced to an equivalence check between the impacted parts of both program versions [88, 39, 40, 78, 5, 33] to conclude the correctness of the modified program version. Program refactoring is an example of an application that
can benefit from equivalence checking approaches.

In [80], an intra-procedural level analysis technique called DiSE (directed incremental symbolic execution) is proposed which uses static analysis and change impact analysis to determine the differences between program versions and the impact of these differences on other locations in the program, and uses this information to direct the symbolic execution to only explore these impacted locations. Analyzing the symbolic paths that characterize the behavior of impacted locations helps in discovering bugs and regression testing (test case selection and augmentation). Another important and practical application of this technique is presented in [79] for monitoring safety-critical systems. Extending DiSE with inter-procedural analysis capabilities is proposed in [84]. A DiSE-based strategy is used in [94] in the cases where code changes occur along with the update of properties. In such cases, only the impacted parts of the program are considered when analyzing the set of updated properties.

In order to reduce the overhead of performing a complete semantics equivalence check of two compared programs, the notion of uninterpreted functions is employed in [88, 39, 40] to abstract the parts of the programs that are syntactically equivalent and have same inputs. In this case a semantics equivalence check is only applied between the corresponding parts that have not been abstracted. An obvious challenge in this technique is to determine which parts correspond to each other.

The notion of uninterpreted functions is also used in [78] in place of common parts of two program versions which are then analyzed by an extended form of symbolic execution to generate a set of symbolic summaries of behavior that reflects the differences between the two program versions. Analyzing these summaries will reveal equivalency, partial equivalency and deltas in the two versions which can be used for
refactoring assurance, change characterization and test suite evolution.

In [5], a change impact analysis is applied to two related program versions to compute the sets of impacted statements in each version and the set of impacted symbolic summaries (constraints) resulting from the symbolic execution of each program version. Checking the equivalence of program versions depends on whether their impacted symbolic summaries are equivalent or not.

A partition-based regression verification is proposed in [12] where the input space of two program versions is partitioned, using a form of symbolic execution in conjunction with random testing, into equivalence-revealing partitions (those that generate the same output in both versions) and difference-revealing partitions. Proving the absence of regression errors is carried out by checking the difference-revealing partitions.

The idea of reducing the equivalence checking of similar programs to a comparison of their deduced predicates summaries is presented in [33]. Proving behavior equivalence of two programs is inferred from the level of coupling found between their predicates.

3.4 Incremental Analysis and Verification in Model-based Development

In his statement paper “Evolution, Adaptation, and the Quest for Incrementality”, Ghezzi [36] argues that supporting software evolution requires building incremental methods and tools to speed up the maintenance process with the focus on the analysis and the verification activities. An incremental approach in such contexts would try to characterize exactly what has been changed and reuse (as much as possible) the results of previous processing steps in the steps that must be re-run after the change.
The motivation for this is twofold: time efficiency and scalability. Given the iterative
development approach suggested by MDD, we believe that this vision needs to be
employed by modern analysis and verification methods.

Surveying the literature for existing work on model-based incremental analysis and
verification methods shows two categories of work: change-based incremental analy-
sis and verification methods and compositional incremental analysis and verification
methods.

**Change-based Incremental Analysis or Verification Methods:** We notice
that the majority of existing work in this category focuses on incremental model-based
test case generation and regression testing [45, 58, 32, 19, 90, 82, 68]. In [45], a global
state reduction technique based on dead and live variables has been used to construct
and to incrementally update (upon a change), on-the-fly, the global reachable state
space of communication protocols modeled as a set of Extended Communicating Fi-
nite State Machines (ECFSMs). The generated global state space is then used to
verify the correctness of such systems. The idea of incremental test case generation
is presented in [32] for testing communication protocols modeled as Finite State Ma-
chines (FSMs). The goal is to test only the modified parts (i.e., transitions) of an
evolving FSM specification. The experimental results provided showed significant
gains, especially when the percentage of the modifications is not higher than 20 %
of the original specification. The same idea has also been explored in the context of
software product lines [90] using SAT-based analysis and UML-RT models [82] using
symbolic execution. In [90], properties of features are defined as first-order logic for-

mulas in Alloy which are then automatically mapped to a transformation that defines
an incremental refinement of test suites. In [82], symbolic execution trees of the subject models (i.e., a model and its evolved versions) are manipulated to incrementally generate their test suites. Techniques for model-based regression test reduction and regression test generation of extended finite state machines (EFSMs) are presented in [58, 19] using an incremental manipulation of the control and data dependence graphs of these models.

The idea of incremental dependency graph adaptation and incremental slice computation is proposed in [67] for software product lines based on the concept of delta modeling [23] where a family of product variants are defined in terms of a core product and a set of deltas describing the changes to be made to the core product such that a product variant is obtained. In the same context, a framework for incremental regression-based testing is proposed in [68] to efficiently test the selected product variants.

Incremental techniques for consistency checking of UML models are presented in [31] using an automatic scope validation of consistency rules instantiation and in [11] using a change impact analysis on the dependencies between model elements and inconsistency rules. Both approaches automatically detect the model changes that violate user-defined consistency rules during their validation.

Our research shares the same motivation as this category of existing work, however it differs in the analysis technique employed and type of models involved.

**Compositional Incremental Analysis or Verification Methods:** In contrast to the change-based incremental verification methods, compositionality-based incremental analysis or verification methods focus on optimizing the analysis or verification process of large systems using different strategies including assume-guarantee
3.4. INCREMENTAL ANALYSIS AND VERIFICATION IN MODEL-BASED DEVELOPMENT

reasoning and compositional minimization (or compositional abstraction) [97]. A key aspect of these strategies is to alleviate some of the scalability issues (e.g., state explosion problem) encountered when analyzing large systems. The main idea of the assume-guarantee reasoning strategy is to make use of the modular structure of the system by breaking down the verification process into sub-processes, each of which is concerned with the verification of one component. A global conclusion about the verification of the whole system should be incrementally derived from the verification of its components without the need to construct the global state space of the system. On the other hand, the compositional minimization or abstraction strategy depends on building a reduced version of the global state space of the system by composing a minimized or abstracted version of the local state space of the individual system components.

Examples of compositional reasoning approaches are presented in:

- [15] for UML-RT models using domain-specific communication patterns and the RAVEN real-time model checker;

- [28] for software product lines using an automata-based model checking algorithm;

- [49] for component-based software systems based on an assume-guarantee model checker.

Examples of compositional minimization or abstraction approaches are presented in:

- [42] for integration testing of component-based applications whose behavior is modeled as UML Statecharts using some heuristic reduction rules;
[14] for verifying systems of asynchronously communicating extended finite state machines (CEFSMs) using a partial product algorithm (or a partial composition technique);

[69] for verifying systems of distributed programs modeled as synchronously communicating finite state machines using a hierarchy-based compositional minimization algorithm;

[50] for communicating state machines with infinite data space by clustering bisimilar states that have different data values into a finite set of equivalent classes and hence reducing the system state space;

[1, 13] for communication protocols modeled by finite state machines communicating through FIFO queues using symbolic reachability analysis and loop-first search analysis, respectively;

[21, 20] for labeled transition systems (LTS) by leveraging precise information about the context and interface constraints for the different components in the system and hence eliminating the presence of possible unreachable states in the resulting compositional reachability analysis of the whole system;

[98] for communicating UML-RT state machines by computing the symbolic execution trees of individual state machines and then composing them to construct a composite symbolic execution tree which can be used to run multiple system-level analyses.

Our research adopts the notion of abstraction presented above to construct an abstracted version of the global state space of the system to be analyzed using symbolic
execution. This construction is performed on-the-fly for the whole system and not incrementally as in the work presented in this category.

3.5 Summary

In this chapter we have outlined the categories of research work that are relevant to our research. First, we have shown examples of approaches for the analysis and the verification of Statecharts-like models. Next, we have presented incremental analysis and verification techniques for evolving programs. Finally, we have reviewed some of the techniques that implement incremental analysis and verification of behavioral models.

Our work is inspired by the category of work presented in Section 3.3 that is built specifically to improve the efficiency of symbolic execution when it is reapplied to an evolving version of a program, e.g., the work on incremental state-space exploration (ISSE) [62], regression model checking [93], memoized symbolic execution [95] and directed incremental symbolic execution (DiSE) [80]. In contrast to the work in [62, 93, 95] which reuses previous model checking results and constraint solving proofs generated from analyzing a predecessor version of a Java program, we reuse symbolic execution trees generated from analyzing a predecessor version of a Rhapsody Statechart. Also in contrast to the work in [80] which performs an intra-procedural regression analysis of an evolved Java code, we perform a regression symbolic analysis of an evolved version of a Rhapsody Statechart.
Chapter 4

Standard Symbolic Execution of Rhapsody Statecharts

In this chapter, we first explain how we build the symbolic execution tree of individual Rhapsody Statecharts. Then, we explain how we extend it to consider a model of communicating Statecharts. Figure 4.1 shows the architecture of an implementation of both techniques.

4.1 Standard Symbolic Execution of Individual Rhapsody Statecharts

In Chapter 3, we referred to some existing work that adopts the idea of symbolic execution of programs and transfers it to state-based models. We found the approach of Zurowska and Dingel [99] is the most relevant to our work as it has been applied to a type of models, namely UML-RT state machines, that shares many features with Rhapsody Statecharts. Therefore, we followed a similar procedure to build ours. The two core components of our standard symbolic execution technique of individual Rhapsody Statecharts are shown in Figure 4.1. The first component,
4.1. STANDARD SYMBOLIC EXECUTION OF INDIVIDUAL RHAPSODY STATECHARTS

Figure 4.1: Standard Symbolic Execution (SE) of Individual and Communicating Rhapsody Statecharts

“SC2MLM Transformation”, is a transformation that transforms an individual Statechart into our custom Mealy-like Machine (MLM) representation. The second component, “KLEE-based MLM2SET Generation”, is a symbolic executor that traverses the Mealy-like Machine model and symbolically executes the action code encountered in each transition to build its symbolic state space. In contrast to the work in [99], our intermediate machine representation of Rhapsody Statecharts takes the form of Mealy-like Machines instead of their Functional Finite State Machines (FFSMs). Additionally, our symbolic execution module is based on an off-the-shelf symbolic execution engine, KLEE [16], to symbolically execute action code encountered in the Statecharts, whereas an in-house symbolic execution engine is used in [99]. The reason for choosing the Mealy-like Machine formalism is to have actions associated only with transitions, and we do this in such a way that it preserves the behavior of
4.1. STANDARD SYMBOLIC EXECUTION OF INDIVIDUAL RHAPSODY STATECHARTS

Statecharts.

4.1.1 Statecharts-to-Mealy-like-Machines Transformation: SC2MLM

We first define our custom Mealy-like Machine formalism that is used to represent a semantically equivalent flattened version of a Rhapsody Statechart. The basic structure of a Mealy-like Machine consists of a set of global variables (sometimes called attributes), a set of simple states with one of them marked as an initial state, and a set of transitions between these states. Simple states in Mealy-like Machines do not have entry or exit actions. Transitions are characterized in the same way as in Rhapsody Statecharts by the event that triggered them, an optional guard and an action that occurs upon firing them. More advanced features that are found in Rhapsody Statecharts such as composite states, concurrent states, states with entry and exit actions and choice points (also called condition connectors or OR-connectors, which allow a transition to branch depending on the value of a guard) need to be mapped to fit the structure of Mealy-like Machines. Our current transformation supports the mapping of these specific features. For example:

- Composite and concurrent states and their outgoing group transitions (formally called group transitions or high-level transitions) are flattened into simple ones;

- Choice points along with their incoming and outgoing branches are replaced by newly created transitions connecting the source state of each choice point with its target states;

- Entry actions of each state are added at the end of the action code of all its incoming transitions; and
Exit actions of each state are added at the beginning of the action code of all its outgoing transitions.

We followed the semantics of the Rhapsody Statecharts that is used by the code generator of the IBM Rational Rhapsody tool to develop our transformation. We performed exhaustive manual checking and inspection for the set of models that we used to validate the correctness of our transformation. Conducting a formal testing or verification study for this task is outside the scope of this research and it is subject for future work.

Figure 4.2 shows an example of a Rhapsody Statechart with the subset of features that we consider in this work and its Mealy-like Machine representation.

**Definition 5.** (Mealy-like Machine)

Formally, a Mealy-like Machine is a tuple $\text{MLM} = (S, V, E, \text{EA}, T, s_0, V_0)$, where:

- $S$ is a nonempty finite set of states;
- $V$ is a set of typed global variables (these are the attributes of the class of objects whose behavior is modeled by the MLM);
- $E$ is a nonempty finite set of events (also called triggers or signals) by which the machine communicates with its environment; they are divided into disjoint sets of input events $E^{\text{inp}}$, output events $E^{\text{out}}$, and internal events $E^{\text{int}}$, and can optionally have arguments (also called parameters), $\text{EA}$;
- $\text{EA}$ is a finite set of event arguments that is disjoint with $V$;
4.1. STANDARD SYMBOLIC EXECUTION OF INDIVIDUAL RHAPSODY STATECHARTS

(a) An Example of a Rhapsody Statechart as Adapted from [48]

(b) The MLM Representation of the Rhapsody Statechart in (a)

Figure 4.2: An Example of a Rhapsody Statechart and its Corresponding MLM Representation
4.1. STANDARD SYMBOLIC EXECUTION OF INDIVIDUAL RHAPSODY STATECHARTS

- $T$ is a set of transitions connecting the states in $S$. A transition is a tuple $t = (s, e, eA, G, A, s', DD)$, where:

  - $s$ and $s'$ are the source state and the target state, respectively;
  - $e$ is an input or internal event in $E^{inp}$ or $E^{int}$;
  - $eA$ is a subset of $EA$ representing the arguments of $e$;
  - $G$ is a Boolean expression (condition) defined over a subset of $V \cup eA$ and is called the guard of $t$;
  - $A$ is a fragment of action code written in C, C++ or Java and is called the action of $t$; statements in this fragment can be assignment expressions, conditional statements, iterations or a special type of statement used to generate some output events to be sent to the environment; the execution of this fragment may result in updating the values of a subset of $V$, constraining some of the variables in $V$, or the parameters in $eA$, or it may cause a sequence of output events in $E^{out}$ to be sent;
  - $DD$ is a subset of the transitions in $T$ that have a data dependency with $t$ with respect to the variables in $V$ (the initialization of this variable occurs as a result of computing the data dependency between the transitions in $T$);
  - $s_0 \in S$ is the initial state.
  - $V_0$ is an initial valuation (or initialization) of the variables in $V$.

**Example 1.** For the example of the MLM model presented in Figure 4.2(b), we can identify the following:
– S = {S0, S1_S0, S1_S1, S1_S2, S1_S3};

– V = {a, b, c};

– E_{inp} = \{setTriangleSides(x, y, z), checkPositiveSides(), checkIsTriangle(), checkTriangleType(), reset()\};

– E_{out} = \{evStartChecking(), evValidInput(), evInvalidInput(), evIsTriangle(), evIsNotTriangle(), evEquilateral(), evScalene(), evIsosceles()\};

– E_{int} = \emptyset;

– EA = \{x, y, z\};

– s_0 = S0;

– V_0 = \{a=0, b=0, c=0\};

– T1 = (S0, setTriangleSides, \{x, y, z\}, \emptyset, \{itsOutput.gen(new evStartChecking()); a=params.x; b=params.y; c=params.z;\}, S1_S0, \emptyset);

– T2 = (S1_S0, checkPositiveSides, \emptyset, \{(x>0)\&\&(b>0)\&\&(c>0)\}, \{itsOutput.gen(new evValidOutput());\}, S1_S1, \{T1\});

– ...

– T10 = (S1_S3, reset, \emptyset, \emptyset, \{a=0;b=0;c=0;\}, S0, \emptyset).
4.1.2 Symbolic Execution of Mealy-like Machine: MLM2SET

The main idea is the same as for the symbolic execution of programs with some variations that reflect the different features of Mealy-like Machines. For instance, states in MLMs are similar to program locations (i.e., program counters), transitions in MLMs are similar to program statements, and event arguments in MLMs are similar to program arguments (which both represent the input data passed from the environment). Just as symbolic execution of programs replaces the concrete values of all input variables to a program by symbolic values, so too will symbolic execution of MLMs replace the concrete values of all event arguments received by a MLM by a unique set of symbolic values. Consequently, both symbolic execution techniques are able to symbolically trace the given artifact (either a program or a MLM) and compute the constraints and the variable updates associated with each execution path. In this case, both constraints and variable updates are defined over symbolic values. The output from symbolic execution is represented by a symbolic execution tree (SET). The nodes in this tree represent program locations (or MLM states) with the symbolic valuations of program variables (or MLM variables) at these locations and the path constraints collected to reach these locations; therefore, we call them symbolic. The edges in this tree are links between symbolic locations, and they reflect the control flow of the execution. For the symbolic execution of programs, these edges usually have no labels; however, in the symbolic execution of MLMs they are labeled with the event causing the transition (with the symbolic values substituting their arguments if any) and also the sequence of events resulting from the execution of the transition action code (along with their arguments, if any). We call these edges
4.1. STANDARD SYMBOLIC EXECUTION OF INDIVIDUAL RHAPSODY STATECHARTS

symbolic transitions. The root of the tree is the node representing the initial state of the MLM with the MLM variables set to their initial values and the path constraint set to “true” or empty.

Definition 6. (Symbolic Execution Tree)

Formally, a symbolic execution tree of a Mealy-Like-Machine, MLM = (S, V, E, EA, T, s₀, V₀), is a tuple SET = (SS, L, ST, ss₀) where:

- SS is a finite set of symbolic states; a symbolic state is a tuple ss = (s, val, pc) where,
  - s is a state in S;
  - val is the symbolic valuation of machine variables V;
  - pc is the path constraint collected to reach state s.

- L is a finite set of labels; a label is a tuple l = (e, Vˢ, σ, out) where,
  - e is an input or internal event in E^{inp} or E^{int};
  - Vˢ is a set of symbolic variables, different from V and EA;
  - σ is a mapping from the original event argument names eA ⊂ EA to the new set of variables names in Vˢ;
  - out is a sequence of some output events e₁^{out}, e₂^{out}, ..., eₙ^{out} in E^{out} and the symbolic valuations of their arguments e₁^{out}A, e₂^{out}A, ..., eₙ^{out}A in EA (if any).

¹Please note that we sometimes use the terms “symbolic transition”, “symbolic path” and “execution path” to mean the same thing, however, more generally, the term “execution path” refers to a sequence of one or more consecutive symbolic transitions in a given symbolic execution tree (SET).
4.1. STANDARD SYMBOLIC EXECUTION OF INDIVIDUAL RHAPSODY STATECHARTS

- $ST$ is a finite set of symbolic transitions defining the transition relation between symbolic states; a symbolic transition is a tuple $st = (ss, l, ss')$ where,
  - $ss$ is the source symbolic state;
  - $l$ is the label of the symbolic transition;
  - $ss'$ is the target symbolic state.

- $ss_0$ is the initial symbolic state.

In the remainder of this section, we provide the details of our algorithm to symbolically execute MLMs as listed in Algorithm 5.

The first task in this algorithm concerns the exploration of the MLM, which is done in a breadth-first-search fashion\(^2\), starting from the initial state and proceeding through each of its outgoing transitions, and so on, until all states have been visited. Since some MLMs may have loops, we need to limit the exploration by some criterion. The criterion that we consider here is to check if a symbolic state is subsumed by some other previously explored symbolic state; if so, we should not explore it again.

**Definition 7.** (Symbolic States Subsumption [2])

Formally, a symbolic state $ss = (s, val, pc)$ is subsumed by another symbolic state $ss' = (s', val', pc')$ iff:

1. $s = s'$;

2. $val = val'$;

3. $pc$ is included in $pc'$ (i.e., $pc \Rightarrow pc'$ or $pc$ is at least as constraining as $pc'$).

\(^2\)Depth-first-search would be another possible alternative.
4.1. STANDARD SYMBOLIC EXECUTION OF INDIVIDUAL
RHAPSODY STATECHARTS

Algorithm 5 Symbolic Execution (SE) of MLMs

Input:
1: A Mealy-Like Machine MLM = (S, V, E, EA, T, s0, V0)

Output:
1: A Symbolic Execution Tree SET = (SS, L, ST, ss0)

Steps:
1: ss0 ← (s0, v0, true)
2: SS ← {ss0}
3: Create a queue q
4: q ← [ss0]
5: //Explore the MLM in Breadth-First-Search fashion
6: while q is not empty do
7: Remove the first element ss = (s, val, pc) from q
8: for all outgoing transitions t = (s, e, eA, G, A, s') of s do
9: Create a unique set Vs of new variables in one-to-one relation with the transition's event arguments eA
10: Create a map σ between eA and Vs
11: Substitute every occurrence of a variable x ∈ eA in the guard G and the action code A by σ(x) to obtain
   Gs ← σ(G) and As ← σ(A)
12: if Gs is satisfiable given val and pc then
13:   feasiblePaths ← SymbolicExecutionOfCode(As, val, Gs, pc)
14:   for all feasiblePath = (val', pc', out') ∈ feasiblePaths do
15:     Create a new symbolic state, ss' = (s', val', pc')
16:     if ss' is not subsumed by any previously generated symbolic state in SS then
17:       SS ← SS ∪ {ss'}
18:       Create a new label, l' = (e, Vs, σ, out')
19:       L ← L ∪ {l'}
20:       Create a new symbolic transition, st' = (ss, l', ss')
21:       ST ← ST ∪ {st'}
22:       q ← enqueue(q, ss')
23:     end if
24:   end for
25: end if
26: end for
27: end while

The second task is the symbolic execution of transitions. In this task, we perform
the following steps for each transition: 1) We create and assign a unique set of
symbolic variables to replace the event arguments of the transition, if any (Lines 9-
10); 2) We substitute the occurrences of these event arguments in the guard expression
and the action code statements by their assigned symbolic values (Line 11); 3) We
check if the guard is satisfiable with respect to the variable updates and the path
constraint of the current symbolic state (Line 12); if so, 4) We use symbolic execution
of programs to execute the updated action code of the transition; the result from the
symbolic execution engine\(^3\) is a set of variable valuation and path constraints that trigger different feasible paths in that code (Line 13); 5) We use the results from the previous step to both create a new set of symbolic states representing the target state of this transition and also to label the edges to these new symbolic states (Lines 15-19).

**Example 2.** The complete SET of the Mealy-like Machine in Figure 4.2(b) is shown in Figure 4.3. As we notice, the SET has depth 6, and it has 14 symbolic execution paths identified by the leaves in the tree. In the following, we show the detailed steps for generating the first three levels of the tree, based on Algorithm 5.

As it is shown, the root of the tree is the symbolic state \(SS1 = (S0, (a=0, b=0, c=0), [])\). Exploring state \(S0\) of \(SS1\) will symbolically execute its outgoing transition \(T1\), which creates new symbolic values \(X\), \(Y\) and \(Z\), assigns them to the arguments \(x\), \(y\) and \(z\) of the event \(\text{setTriangleSides}\), updates the machine variables \(a\), \(b\) and \(c\) to have the new symbolic values of the \(\text{setTriangleSides}\) event arguments, creates a new symbolic state \(SS2 = (S1\_S0, (a=X, b=Y, c=Z), [])\), and finally forms the label for the symbolic transition (i.e., the edge) connecting \(SS1\) and \(SS2\) with the transition event \(e = [\text{setTriangleSides}(X, Y, Z)]\) and the output \(out = [\text{itsOutput.evStartChecking()}]\). Because there is no guard associated with the transition \(T1\) and there are no conditional statements in its action code, there is no update to be made to the path constraint of the symbolic state \(SS2\). The next step is exploring state \(S0\_S1\) of \(SS2\) which will symbolically execute its outgoing transitions, \(T2\), \(T3\) and \(T4\), in sequence. Symbolically executing \(T2\) creates a new symbolic state \(SS3 = (S1\_S1, (a=X, b=Y, c=Z), [])\).

\(^3\)We use the popular open-source symbolic execution engine KLEE [16].
4.1. STANDARD SYMBOLIC EXECUTION OF INDIVIDUAL RHAPSODY STATECHARTS

\[(!(X<0) \&\& !(Y<0) \&\& !(Z<0))]\) and forms the label for the edge between SS2 and SS3 with the transition event \(e = [\text{checkPositiveSides()}]\) and the output \(out = [\text{itsOutput.evValidInput()]}\). As we notice, the value of the path constraint of SS3 reflects the guard condition of transition T2. In contrast with T2, symbolically executing T3 creates three symbolic states representing state S1, S2, which are SS4, SS5 and SS6. All have the same variable valuations \(val = (a=X, b=Y, c=Z)\) but have different path constraints representing all feasible cases for the guard condition of T3. The edges connecting SS3 with SS4, SS5 and SS6 are all labeled with the input event \(e = [\text{checkPositiveSide()}]\) and the output \(out = [\text{itsOutput.evInvalidInput()]}\). Finally, symbolically executing T4 creates a new symbolic state \(SS7 = (S0, (a=0, b=0, c=0), [])\) and forms the label for the edge between symbolic state SS2 and symbolic state SS7 with the transition input event \(e = [\text{reset()}]\) and the output \(out = []\). The difference between the symbolic state SS7 and the other symbolic states SS4, SS5 and SS6 is that we find that there is another symbolic state, SS1, that has already been explored before, and it subsumes SS7, therefore no further exploration is needed for SS7.

We can notice that the symbolic execution of a transition may result in zero, one or many symbolic edges depending on the satisfiability of the guard condition and action code of the transition. Therefore, we refer to transitions with more than one symbolic edge in the resulting SET as multi-path transitions (e.g., transitions T3 and T4 are multi-path transitions).

The resulting SET can be used to run many \textit{intra-level} types of analyses (e.g., reachability analysis and guard analysis) to ensure the correctness of the individual objects in isolation of the others.
Figure 4.3: SET of the MLM Model in Figure 4.2(b) (Highlighted Levels: 1, 2 and 3)

4.2 Symbolic Execution of a Model of Communicating Rhapsody Statecharts

In the previous subsection, we showed how to symbolically execute individual Statecharts. In this section we present how we extended our standard symbolic execution technique to consider a model of communicating Statecharts. The goal here is to facilitate the analysis of the overall behavior of the system and the interoperability between its communicating objects. Our approach is based on an on-the-fly exploration of all interleaving Statecharts in a given model to construct the global state space of the system represented by a more general symbolic execution tree called a
4.2. SYMBOLIC EXECUTION OF A MODEL OF COMMUNICATING RHAPSODY STATECHARTS

The two components of our standard symbolic execution technique of a model of communicating Statecharts are shown in Figure 4.1, which are the “RhapM2CMLM Transformation” and the “KLEE-based CMLM2GSET Generation”. Each component generalizes the functionality provided by its corresponding one for individual Statecharts. This generalization incorporates the extra features present in the communication setting. For instance, we need to access the Object Model Diagrams of a Rhapsody model to identify the set of communicating objects in the model and to determine the communication topologies between these objects. This requires the extension of our previous MLM formalism to represent a collection of communicating MLMs and to represent their communication topologies. We also need to extend our previous symbolic execution tree definition to represent, instead, a global symbolic execution tree. In such global symbolic execution trees, symbolic model states are used in place of simple symbolic states.

4.2.1 Rhapsody Model of Communicating Statecharts-to-Communicating Mealy-like-Machines Transformation: RhapM2CMLM

The input to this component is a Rhapsody model represented by its Object Model Diagrams (OMDs) and its Statecharts and the output is a Communicating Mealy-like Machines representation. Our formalism for Communicating Mealy-like Machines is shown below and it uses the Mealy-like Machine (MLM) formalism presented in Subsection 4.1.1 (please see Definition 5). For illustration, we refer to the model example shown in Chapter 2, Figure 2.1. For simplicity, we consider only simple states and simple transitions in the Statecharts used in this example, yet our implementation
4.2. SYMBOLIC EXECUTION OF A MODEL OF COMMUNICATING RHAPSODY STATECHARTS

supports Statecharts with more complex features such as hierarchical states, orthogonal regions, choice points and group transitions.

**Definition 8.** (Communicating Mealy-like-Machines)

Formally, a global model of \( n \) asynchronously communicating Mealy-like-Machines is a tuple \( M = (MLM^M, Q^M, s_0^M, V^M, \nu_0^M) \) where:

- \( MLM^M = (MLM_1, MLM_2, ..., MLM_n) \) where \( MLM_i = (S_i, V_i, E_i, EA_i, T_i, s_{0i}, V_{0i}) \) is the \( i^{th} \) Mealy-like Machine in \( M \) that is defined as follows:

  - \( S_i \) is a nonempty finite set of *states* and \( S_1 \cap S_2 \cap ... \cap S_n = \emptyset \);
  - \( V_i \) is a set of typed global *variables* (these are the attributes of the class of objects whose behavior is modeled by the MLM);
  - \( E_i \) is a nonempty finite set of *events* (also called triggers or signals) by which the machine communicates with its environment. Events can have arguments \( EA_i \) (also called parameters) and are partitioned into input events, \( E_{inp}^i \), output events, \( E_{out}^i \), and internal events, \( E_{int}^i \) and \( E_{inp}^i \cap E_{out}^i \cap ... \cap E_{out}^n = E_{out}^1 \cap E_{out}^2 \cap ... \cap E_{out}^n = \emptyset \);
  - \( EA_i \) is a finite set of event arguments that is disjoint with \( V_i \) and \( EA_1 \cap EA_2 \cap ... \cap EA_n = \emptyset \);
  - \( T_i \) is a set of transitions connecting the states in \( S_i \). A transition is a tuple \( t = (s, e, eA, G, A, s', DD, COMD) \), where:
    - \( s \) and \( s' \) are the source state and the target state, respectively;
    - \( e \) is one of the input or internal events in \( E_{inp}^i \) and \( E_{int}^i \);
    - \( eA \) is a subset of \( EA_i \) representing the arguments of \( e \);
4.2. SYMBOLIC EXECUTION OF A MODEL OF COMMUNICATING RHAPSODY STATECHARTS

- G is a Boolean expression (condition) defined over a subset of $V_i \cup eA$ and is called the *guard* of $t$;

- A is a fragment of action code written in C, C++ or Java and it is called the action of $t$; statements in this fragment can be assignment expressions, conditional statements, iterations or a special type of statements used to generate some output events to be sent to the environment; the execution of this fragment may result in updating the values of a subset of $V_i$, constraining some of the variables in $V_i$, or the parameters in $eA$, or it may cause a sequence of output events in $E_{i}^{out}$ to be sent;

- DD is a subset of the transitions in $T_i$ that have a *data dependency* with $t$ with respect to the variables in $V_i$;

- COMD is a subset of the transitions in $T_j$ of MLM$_j \in M$ where $j = 1...n$ and $j \neq i$ that have a *communication dependency* with $t$.

- $s_{0i} \in S_i$ is the *initial state*.

- $V_{0i}$ is an *initial valuation* (or initialization) of the variables in $V_i$.

- $Q^M = (q_1, q_2, ..., q_n)$ is the set of communication queues in $M$ (these queues allow the exchange of messages between MLMs);

- $s_0^M = (s_{01}, s_{02}, ..., s_{0n})$, where $s_{01}$ is the initial state for MLM$_1$;

- $V^M = V_1 \cup V_2 \cup ... \cup V_n$, where $V_1$ is the set of variables in MLM$_1$ and $V_1 \cap V_2 \cap ... \cap V_n = \emptyset$;

- $V^M_0 = V_{01} \cup V_{02} \cup ... \cup V_{0n}$, where $V_{01}$ is the initial valuation of the variables $V_1$ in MLM$_1$. 
Example 3. Based on this formalism, we can represent the Rhapsody model shown in Section 2, Figure 1 as the following CMLM:

\[ M^{AB} = ((MLM_{itsA}, MLM_{itsB}), (q_{itsA}, q_{itsB}), (aS1, bS1), (x, y), (0, 5)) \] where:

- \( MLM_{itsA} = (\{aS1, aS2, aS3\}, \{x\}, \{e1, e2, e3\}, \{e1Arg1\}, \{itsA_T1, itsA_T2, itsA_T3\}, aS1, \{0\}); \)
  - \( itsA_T1 = (aS1, e1, e1Arg1, 0, (if(params.e1Arg1 > 7){x = 5;} else{x = 1;}), aS2, 0, \emptyset); \)
  - \( itsA_T2 = (aS2, e2, 0, (itsB.gen(newe5(5));), aS3, 0, \emptyset); \)
  - \( itsA_T3 = (aS3, e3, 0, (x = 0;), aS1, 0, \{itsB_T3\}); \)

- \( MLM_{itsB} = (\{bS1, bS2\}, \{y\}, \{e4, e5, e6\}, \{e5Arg1\}, \{itsB_T1, itsB_T2, itsB_T3\}, bS1, \{5\}); \)
  - \( itsB_T1 = (bS1, e4, 0, (y = y + 1;), bS2, 0, \emptyset); \)
  - \( itsB_T2 = (bS2, e5, e5Arg1, (params.e5Arg1 == 5), (itsA.gen(newe3()));), bS1, \{itsB_T1\}, \emptyset); \)
  - \( itsB_T3 = (bS2, e6, 0, (y = y - 1;), bS2, 0, \{itsA_T2\}). \)

Additionally, we can recognize that 1) events \( e1, e2, e4 \) and \( e6 \) are external events, while events \( e3 \) and \( e5 \) are internal events and 2) transition \( itsA_T1 \) is a multi-path transition due to the \texttt{if-else} conditional statement in the transition’s action code.
4.2. SYMBOLIC EXECUTION OF A MODEL OF COMMUNICATING RHAPSODY STATECHARTS

4.2.2 Symbolic Execution of Communicating Mealy-like Machines: CMLM2GSET

The input to this component is a set of Communicating Mealy-like Machines representing a Rhapsody model and the output is a Global Symbolic Execution Tree (GSET). The formal definition of a Global Symbolic Execution Tree (GSET) extends our previous definition in Subsection 1.1 for the symbolic execution tree of an individual MLM. It replaces the definition for the symbolic states in the tree to consider instead symbolic model states and it adds an extra element to represent the queues’ contents in a given model.

Definition 9. (Global Symbolic Execution Tree)

Formally, a global symbolic execution tree of a model of \( n \) asynchronously communicating Mealy-like-Machines, \( M = (MLM^1, Q^M, s_0^M, v^M, v_0^M) \) where \( MLM_i = (S_i, V_i, E_i, EA_i, T_i, s_0, V_0^i) \) is the \( i \)th Mealy-like Machine in \( M \), is a tuple \( GSET^M = (SS^M, L^M, ST, ss^M_0) \) where,

- \( SS^M \) is a finite set of symbolic model states; a symbolic model state is a tuple \( ss^M = (s^M, val, pc, q^M) \) where,
  - \( s^M \) is a model state in \( S^M = (s_1, s_2, ..., s_n) \) where \( s_i \in S_i \) of \( MLM_i \);
  - \( val \) is the symbolic valuation of the model variables \( V^M \);
  - \( pc \) is the path constraint collected to reach the model state \( s^M \);
  - \( q^M \) is the content (or update) of the model queues \( Q^M \);

- \( L \) is a finite set of labels; a label is a tuple \( l = (e, V^e, \sigma, out) \) where,
  - \( e \) is an input or internal event in \( MLM_i \) for some \( i = 1...n \);
4.2. SYMBOLIC EXECUTION OF A MODEL OF COMMUNICATING RHAPSODY STATECHARTS

- \( V^s \) is a set of symbolic variables, different from \( V^M \) and \( EA_1(MLM_1) \) for all \( i = 1...n \);
- \( \sigma \) is a mapping from the original event argument names \( eA_1 \subset EA_1(MLM_1) \) for all \( i = 1...n \) to the new set of variables names in \( V^s \);
- \( \text{out} \) is a sequence of some output events \( e_1^{\text{out}}, e_2^{\text{out}}, ..., e_n^{\text{out}} \) in \( E_0^s \) and the symbolic valuations of their arguments \( e_1^{\text{out}}A, e_2^{\text{out}}A, ..., e_n^{\text{out}}A \) in \( EA_1(MLM_1) \) (if any).

- \( ST \) is a finite set of symbolic transitions defining the transition relation between symbolic states; a symbolic transition is a tuple \( st = (ss^M, l, ss'^M) \) where,
  - \( ss^M \) is the source symbolic model state;
  - \( l \) is the label of the symbolic transition;
  - \( ss'^M \) is the target symbolic model state.
- \( ss^M_0 \) is the initial symbolic model state.

Now we explain the steps to symbolically execute a model of communicating MLMs as listed in Algorithm 6.

The first step in this algorithm is an on-the-fly breadth-first-search exploration for each MLM in the model \( M \) concurrently, starting from the initial model state and proceeding through the outgoing transitions of the initial state in each MLM, and so on, until all model states have been visited or until we reach a certain depth limit in case of unbounded loops. Checking if a symbolic model state has been visited is done using the subsumption criterion definition listed below that extends Definition 7 to consider the queues’ contents of symbolic model states.
4.2. SYMBOLIC EXECUTION OF A MODEL OF COMMUNICATING RHAPSODY STATECHARTS

Definition 10. (Symbolic Model States Subsumption [2])

Formally, a symbolic model state \( ss^M = (s^M, \text{val}, \text{pc}, q^M) \) is subsumed by another symbolic mode state \( ss'^M = (s'^M, \text{val}', \text{pc}', q'^M) \) iff:

1. \( s^M = s'^M \);
2. \( \text{val} = \text{val}' \);
3. \( q^M = q'^M \);
4. \( \text{pc} \) is included in \( \text{pc}' \) (i.e., \( \text{pc} \Rightarrow \text{pc}' \) or \( \text{pc} \) is more constrained than \( \text{pc}' \)).

Symbolic model states that are marked "visited" do not require further exploration and therefore they form the leaves in a GSET. In Figure 4.4, we identified the subsumed nodes for all the leaves of the GSET(\( AB^{V_0} \)). For example, symbolic model state \( SS21 \), which is a leaf node in the GSET(\( AB^{V_0} \)), is subsumed by symbolic model state \( SS11 \), which is a non-leaf node in the GSET(\( AB^{V_0} \)).

A common problem when building the global state space of asynchronous systems are unbounded queues which are continuously filled with messages (sent either from the environment or from the peers in the system) that cannot be consumed at the same rate. In this case the exploration will exhibit an infinite state behavior and will not end. Possible solutions for this problem are addressed in [70] using bounded queues or assuming a “slow” environment in which external events (i.e., events that are sent by the environment to the system) are offered only when the system reaches a stable state (i.e., when the execution of all internal events are completed and all system queues have become empty). In such an environment, a higher priority is given to internal events over external ones.
4.2. SYMBOLIC EXECUTION OF A MODEL OF COMMUNICATING RHAPSODY STATECHARTS

In our approach, we adopt the “slow” environment assumption used in [70] by allowing the execution of transitions with external events only if the system is in a stable state.

The second step is the symbolic execution of transitions which is done using the open-source symbolic execution engine KLEE [16]. For each such transition that has an external event or an available internal event (i.e., the event exists in the head of the queue of the MLM to which the transition belongs to) (Line 11) we do the following:

1. we create and assign a unique set of symbolic variables to replace the event arguments of the transition, if any (Lines 12-13);

2. we substitute the occurrences of these event arguments in the guard expression and the action code statements, by their assigned symbolic values (Line 14);

3. we check if the guard is satisfiable with respect to the variable updates and the path constraint of the current symbolic state (Line 15); if so,

4. we use the symbolic execution engine KLEE [16] to execute the updated action code of the transition; the results are a set of variable assignments and path constraints that trigger different feasible paths in that code (Line 13);

5. we use the results from the previous step to both create a new set of symbolic model states representing the target state of this transition and also to label the edges to these new symbolic model states (Lines 17-28).

Example 4. Applying this algorithm on the example model in Figure 2.1 results in the GSET shown in Figure 4.4 which consists of 25 symbolic model states and 9 symbolic execution paths given by the number of leaves in the tree.
Algorithm 6 Symbolic Execution (SE) of CMLMs

**Input:**
1: A system/model of \( n \) communicating Mealy-Like Machines \( M = (MLM^n, Q^n, s^0, V^n, V^0) \)

**Output:**
1: A Global Symbolic Execution Tree \( \text{GSET}^M = (SS^M, L, ST, ss^0) \)

**Steps:**
1: \( ss^0 \leftarrow (s^0, V^0, \text{true}, q_0^M) \)
2: \( SS^M \leftarrow \{ss^0\} \)
3: Create a queue \( q \)
4: \( q \leftarrow [ss^0] \)
5: //Breadth-First-Search fashion exploration for each MLM in \( M \)
6: while \( q \) is not empty do
7: Remove the first element \( ss^M = (s^M, val, pc, q^M) \) from \( q \)
8: for all MLM \( i \) in MLM\( ^M \) do
9: \( s_i \leftarrow s^M(i) \) where \( s_i \in S_i(MLM_i) \)
10: for all outgoing transitions \( t_i = (s^M, e, eA, G, A, s^M') \) of \( s_i \) do
11: if \( ((q^i = \phi) \land (e \in E^{inp}_i)) \lor ((\text{head}(q^i) = e) \land (e \in E^{out}_i)) \) then
12: Create a unique set \( V^s \) of new variables in one-to-one relation with the transition’s event arguments \( eA \)
13: Create a map \( \sigma \) between \( eA \) and \( V^s \)
14: Substitute every occurrence of a variable \( x \in eA \) in the guard \( G \) and the action code \( A \) by \( \sigma(x) \) to obtain \( G^s \leftarrow \sigma(G) \) and \( A^s \leftarrow \sigma(A) \)
15: if \( G^s \) is satisfiable given \( val \) and \( pc \) then
16: \( \text{feasiblePaths} \leftarrow \text{SymbolicExecutionOfCode}(A^s, val, G^s, pc) \)
17: for all \( \text{feasiblePath} = (val', pc', out') \in \text{feasiblePaths} \) do
18: \( q^M \leftarrow \text{out'} \)
19: Create a new symbolic state, \( ss^M = (s^M, val', pc', q^M) \) where \( s^M = s^M', \text{replace}(s_i, s_i') \)
20: if \( ss^M \) is not subsumed by any previously generated symbolic model state in \( SS^M \) then
21: \( SS^M \leftarrow SS^M \cup \{ss^M\} \)
22: Create a new label, \( l' = (e, V^s, \sigma, \text{out'}) \)
23: \( L \leftarrow L \cup \{l'\} \)
24: Create a new symbolic transition, \( st' = (ss^M, l', ss^M') \)
25: \( ST \leftarrow ST \cup \{st'\} \)
26: \( q \leftarrow \text{enqueue}(q, ss^M) \)
27: end if
28: end for
29: end if
30: end if
31: end for
32: end for
33: end while

4.3. Summary

In this chapter, we have shown examples of individual and communicating Rhapsody Statecharts. We have presented the formalisms we use to represent a semantically equivalent flattened version of Rhapsody Statecharts and a model of communicating Rhapsody Statecharts such that we can symbolically execute them. The proposed
Figure 4.4: GSET(AB^0) – Global Symbolic Execution Tree for the Example Model AB^0 in Figure 2.1
formalisms are called Mealy-like Machines (MLMs) and Communicating Mealy-like Machines (CMLMs). We also have provided the formal definitions for the structures of the symbolic execution tree (SET) and the global symbolic execution tree (GSET) resulting from each symbolic execution technique.
Chapter 5

Optimizing Symbolic Execution of Evolving Rhapsody Statecharts

In this chapter, we first illustrate with a running example the optimization ideas employed in this research for the symbolic execution of evolving Statecharts. Next, we provide the details of our two proposed techniques, namely memoization-based symbolic execution (MSE) and dependency-based symbolic execution (DSE), for both individual and communicating Statecharts.

5.1 Motivating Example

In this section we explain using a running example the optimization ideas employed in the next section. For illustration, we use the models in Figure 5.1 and their symbolic execution trees (SETs) as shown in Figures 5.2 and 5.3 to show how our optimization techniques work for evolving individual Statecharts.

In Figure 5.1, we have two versions of an example of individual Rhapsody Statechart: \( V_0 \) (in Figure 5.1(a)) is the base version and \( V_1 \) (in Figure 5.1(b)) is a modified version of the base version where we modified the conditional statement in the action
5.1. MOTIVATING EXAMPLE

(a) \( V_0 \) - The base version of an example Statechart

(b) \( V_1 \) - A modified version of the Statechart in (a)

Figure 5.1: Two versions of an example Statechart

code of transition \( T_2 \). In Figure 5.2, we show the symbolic execution tree of \( V_0 \), \( \text{SET}(V_0) \), however in Figures 5.3(a) and 5.3(b), we show two different illustrations for the symbolic execution tree of \( V_1 \), \( \text{SET}(V_1) \).

Alternatively, we can use the model version \( AB^{V_1} \) shown in Figure 5.4 and the global symbolic execution trees (GSETs) in Figures 5.5 and 5.6 to illustrate how the same concepts are applied for evolving communicating Statecharts. The model version \( AB^{V_1} \) is the successive version of the model example \( AB^{V_0} \) shown in Figure 2.1 with a change in transition \( \text{itsA}_A^{T_2} \) of the Statechart of object \( \text{itsA} \).
5.1.1 Optimization Via Reuse – An Artifact-oriented Evolution Support Approach

By comparing the unhighlighted parts (colored black) of the symbolic execution tree of V1, SET(V1), in Figure 5.3(a) with the symbolic execution tree of V0, SET(V0), in Figure 5.2, we notice that these parts are identical. However, by looking at the highlighted parts (colored red) of the SET(V1), we find that:

1. they are slightly different from their corresponding parts in the SET(V0);
2. they represent the symbolic execution of transition T2 (the changed transition in V1) and subsequent transitions; and
5.1. MOTIVATING EXAMPLE

(a) The parts that are different from SET($V_0$) are highlighted (in red).

(b) The pruned parts resulting from the partial exploration of transitions: $T_1$, $T_3$, and $T_4$ are grayed out.

Figure 5.3: SET($V_1$) – Symbolic execution tree for the Statechart in Figure 5.1(b)
5.1. MOTIVATING EXAMPLE

(a) $AB^{V1}$ – A successive version of the example model $AB^{V0}$ in Figure 2.1

(b) Dependency graph of the model version $AB^{V1}$

Figure 5.4: A modified model version $AB^{V1}$ and its dependency graph

3. they reflect the parts of the SET($V0$) that are impacted by the changes made on $V1$.

Therefore, if we already have the SET($V0$) and we manage to identify the parts of it that need to be updated in order to account for the changes made on $V1$, then we can direct the symbolic execution of $V1$ to generate only the new updated parts to replace the old ones. The applicability and utility of running this process depends, first, on the existence of the SET($V0$) and, second, on the amount of savings gained from reusing it which can be very large for big complex models that undergo some minor changes (i.e., changes with limited impact). Possible measures for quantifying the amount of savings are: 1) the time gained from not generating the SET($V1$) from scratch; 2) the percentage of nodes (or symbolic states) in the SET($V0$) that can be safely reused; or 3) the percentage of execution paths in the SET($V0$) that can be safely reused. A very important benefit from this latter measure is that it provides
Figure 5.5: MGSET(\(AB^W\)) – The parts that are different from GSET(\(AB^0\)) in Figure 4.4 are highlighted (in red).
Figure 5.6: DGSET(ABV1) – The pruned parts resulting from the partial exploration of transition itsA_T1 are grayed out.
us with a good estimate of how many of the test cases generated from the SET(V0), to test V0, can be reused for testing V1. We notice that the location of the change in the model may suggest the effectiveness of reusing the SET(V0). We believe that, in general, the closer the change is to the initial state of the model, the smaller the number of nodes in the SET(S0) that can be reused. Additionally, the more model paths go through a changed state or transition, the more updates of the SET(V0) are needed and the smaller the effectiveness of reusing SET(V0). For example, having a change in state S1 or transition T1 in V1 will require an update to all the descendants of the symbolic state SS2 in the SET(V0), leaving only two symbolic states to be reused.

5.1.2 Optimization Via Reduction – *An Analysis-oriented Evolution Support Approach*

First, we assume that the base version of a model (i.e., the version before the change) has already been analyzed successfully to, e.g., determine reachability of states or generate test cases. This assumption enables the optimization described below.

Second, as we discussed in Section 4.1, the symbolic execution of a transition, t4, in an MLM model may result in zero, one or many symbolic transitions of t4 in the symbolic execution tree of that model, depending on: 1) the path constraints and the variable valuation of the symbolic state representing the source state of t4; 2) the satisfiability of the guard condition of t4; and 3) the number of execution paths of the action code of t4. For example, the symbolic execution of transition T1 in the two versions of the model in Figure 5.1 results in two symbolic transitions connecting the symbolic state SS2 of state S1 (the source state of T1), with the two
symbolic states SS3 and SS4 of state S2 (the target state of T1). The first symbolic transition is taken if a=7, while the second takes place if a=1 (these are the two cases that satisfy the guard condition of T1). Similarly, the symbolic execution of T2 results in having two symbolic transitions originating from each of the symbolic states SS3 and SS4 and ending at the symbolic states SS5 and SS6 (resp., SS8 and SS9) as a result of having a conditional statement in the action code of T2. On the other hand, the symbolic execution of transition T3, which does not have a guard or any conditional statements in its action code, results in having only one symbolic transition originating from each of the symbolic states SS3 and SS4 and ending at the symbolic states SS7 and SS10, respectively. In the same way, the symbolic execution of transition T4 results in having one symbolic transition originating from each of the symbolic states SS5, SS6, SS8 and SS9 and ending at the symbolic states SS11, SS12, SS13 and SS14, respectively.

Third, as discussed in Section 2.3, applying a dependency analysis on a modified version of a model given the list of the changes made on its previous version allows us to identify all the parts of the model that have an impact on or are impacted by the changes.

Now, with all that said, our question is: Can we direct the symbolic exploration of a modified version of a model to exhaustively explore all changed transitions and all other transitions that either impact or are impacted by one or more changed transitions and to reduce the exploration of the remaining transitions (i.e., the transitions that neither impact nor are impacted by the change) to a minimum (i.e., to consider only one symbolic transition per such transition)? Our goal is twofold: 1) to explore the minimal set of symbolic transitions to perform the analysis on the evolved state
machine model (to, e.g., reveal states that have become unreachable by the change or to generate test cases for the new executions introduced by the change); and 2) to reduce the time of exploration as well as the size of the resulting SETs.

For example, applying a dependency analysis on $V_1$ given that transition $T_2$ has been changed with respect to its base version $V_0$ shows that there is no dependency between $T_2$ and any other transitions as none of them defines a variable that transition $T_2$ uses (and vice versa). Now, with this information in mind, we can direct the symbolic execution of $V_1$ to “fully” explore the changed transition $T_2$ (i.e., to explore all its symbolic transitions) and to only “partially” explore the rest of the transitions in the model (i.e., to explore only one of its symbolic transitions). Following this process to symbolically execute $V_1$ results in the SET shown in Figure 5.3(b), which has 6 symbolic states less than the complete SET($V_1$). Although the new SET is not complete, it is sufficient to run regression types of analysis. For instance, the SET in Figure 5.3(b) can be used to determine if the change introduced any unreachable states or which test cases must be run to test the execution paths introduced by the change.

As the amount of savings to be gained here depends on the number of symbolic transitions to be pruned from the partial exploration of unimpacted transitions, this technique is most beneficial if applied to Statecharts that have transitions with disjunct guards (i.e., “||”) or with action code that has conditional statements, which both result in more than one symbolic transition in the SET. The more symbolic transitions we have for an unimpacted transition, the more savings we gain if it is partially explored.
5.2 Proposed Symbolic Execution Optimizations for Evolving Rhapsody Statecharts

Symbolic execution is an expensive approach especially when applied to big and complex artifacts (programs or models) where the size of the generated SETs can be very large. Software artifacts can undergo several iterations and refinements and repeating the symbolic execution of these artifacts from scratch after every iteration or refinement step can be very tedious and time consuming. The new version of an artifact can be very similar to the previous one, so excluding the unchanged parts from successive runs of the symbolic execution technique reduces the time required for any symbolic execution-based types of analyses. Alternatively, directing the successive runs of the symbolic execution technique away from execution paths that are not impacted and towards execution paths that are impacted also reduces the time required for any symbolic execution-based types of analyses. Inspired by the work in [95] and [80] for optimizing the symbolic execution of evolving programs, we propose two techniques for optimizing the symbolic execution of evolving Rhapsody Statecharts. The two techniques are applied for both individual Statecharts and communicating Statecharts.

5.2.1 Memoization-based Symbolic Execution (MSE)

In this section we present our memoization-based symbolic execution technique (MSE) for both individual and communicating Statecharts. The architectures of implementations of the proposed approaches are shown in Figures 5.7(a) and 5.7(b), respectively.
5.2. PROPOSED SYMBOLIC EXECUTION OPTIMIZATIONS FOR EVOLVING RHAPSODY STATECHARTS

(a) MSE architecture for evolving individual Statecharts

(b) MSE architecture for evolving communicating Statecharts

Figure 5.7: Memoization-based Symbolic Execution (MSE) proposed architecture

Technique Description

The basic idea of this technique is to reuse the results from a previous run of the symbolic execution technique on the model version before the change to optimize the next run of the technique on the model version after the change such that only the parts of the SET (resp., GSET) that are affected by the change will be regenerated. It is obvious that a key requirement for this technique is to have a stored version of the SET (resp., GSET) of the model before the change. Another requirement is to have a list of the changes made to the previous version of the model which will
be used to remove the parts of the previous SET (resp., GSET) that are affected by the changes and to mark their corresponding source nodes (i.e., symbolic states or symbolic model states) as “needing an update”. Rerunning the SE technique on the modified model version will be directed to generate the required updates for these marked nodes.

To cope with the requirements of this technique, we needed to add to the definition of the symbolic state (resp., symbolic model state) presented earlier in Chapter 4 as part of the definition of the symbolic execution tree (resp., global symbolic execution tree) the following two features: 1) a Boolean attribute \texttt{needsUpdate} that indicates whether a given symbolic state needs to be updated or not, and 2) a hashmap \texttt{branchesToUpdate : S \rightarrow \mathcal{P}(T)} of states with their corresponding outgoing transitions that need to be updated.

To be able to identify the nodes and the branches that need to be updated, we need to provide the technique with a list of all the differences between the modified model version and its previous version. Our technique only considers a subset of model changes that can be mapped to changes to transitions in the MLM (resp., CMLMs) representation. This includes: 1) adding states or transitions, 2) removing states or transitions, 3) modifying entry or exit actions of states and 4) modifying events, guards or actions of transitions. For example, adding or removing states can be represented by an addition or a deletion of transitions connecting these states with other states in the model; also, updating the entry actions (resp., exit actions) of states can be represented by an update of their incoming transitions (resp., outgoing transitions). Therefore, we manage to represent these differences in terms of a set of updated transitions \( T_{\text{updated}} \) including all transitions that have been added to, deleted...
5.2. PROPOSED SYMBOLIC EXECUTION OPTIMIZATIONS FOR EVOLVING RHAPSODY STATECHARTS

from or updated in the modified version of the model. Other types of model changes such as the addition, deletion or modification of variables (i.e., class attributes) are not supported since these changes impacts the variable valuation of every state in the model and hence requires a complete regeneration of the entire SET of the model after the change.

To find the differences between two Rhapsody models, we use the Rhapsody DiffMerge tool that we mentioned in Section 2.3. These differences are mapped to their MLM (resp., CMLMs) correspondences using our “SC2MLM DiffMap” (resp., “RhapM2CMLM DiffMap”) component, which outputs the set of all transitions $T_{\text{updated}}$ that are found to have been updated in the modified MLM (resp., CMLMs) version. Each element in the set $T_{\text{updated}}$ indicates the name of a transition that has been affected by the change, the name of the source state of this transition, and the change type applied to this transition (i.e., “Addition”, “Deletion”, “Modification”).

The main component of our MSE technique for individual Statecharts is the “KLEE-based MLM2MSET Generation” component shown in Figure 5.7(a). Similarly, the main component for communicating Statecharts is the “KLEE-based CMLM2MGSET Generation” component shown in Figure 5.7(b). The detailed steps of implementing each component are listed in Algorithm 7 and Algorithm 8, respectively. In the sequel, we describe the details of both components.

The three inputs to both components are: 1) the MLM (resp., CMLMs) representation of the modified model version $\text{MLM}_{\text{modVer}}$ (resp., $\text{CMLM}_{\text{modVer}}$), 2) the set of transitions that have been updated in the modified version of the model $T_{\text{updated}}$, and 3) the SET$_{\text{baseVer}}$ (resp., GSET$_{\text{baseVer}}$) to be reused.

Four successive tasks are performed by both components.
5.2. PROPOSED SYMBOLIC EXECUTION OPTIMIZATIONS FOR EVOLVING RHAPSODY STATECHARTS

Algorithm 7 Memoized-based Symbolic Execution (MSE) of an Evolving MLM

Inputs:
1: A modified MLM model version: $MLM^{\text{modVer}} = (S, V, E, EA, T, s_0, V_0)$
2: A SET of the previous MLM model version $MLM^{\text{baseVer}}$: $SET^{\text{baseVer}} = (SS, L, ST, ss_0)$
3: A set of all changed transitions between $MLM^{\text{modVer}}$ and $MLM^{\text{baseVer}}$: $T^{\text{updated}}$

Output:
1: A Memoized-based SET of $MLM^{\text{modVer}}$: $MSET^{\text{modVer}}$

Steps:
1: //Task 1: Explore $SET^{\text{baseVer}}$ in breadth-first-search fashion to remove the parts (i.e., the sub-trees) that need to be updated and to mark the nodes in $SET^{\text{baseVer}}$ to be re-explored when we re-run the standard SE on $MLM^{\text{modVer}}$
2: $ss_0 \leftarrow (s_0, V_0, \text{true})$
3: Create a queue $q$
4: $q \leftarrow [ss_0]$
5: while $q$ is not empty do
6: Remove the first element $ss = (s, val, pc)$ from $q$
7: if there is $t$ in $T^{\text{updated}}$ such that $s = \text{sourceState}(t)$ and $t$ is marked as “Added” then
8: //Mark $ss$ as “needsUpdate”
9: $ss.needsUpdate \leftarrow \text{true}$
10: //Add state $s$ and its outgoing transition $t$ to the hashmap $branchesTo_Update$ of $ss$
11: $ss.branchesTo_Update \leftarrow ss.branchesTo_Update \cup \{s \rightarrow \{t\}\}$
12: end if
13: for all outgoing symbolic transitions $st = (ss, l, ss')$ of $ss$ do
14: if there is $t \in T^{\text{updated}}$ such that $\text{sourceState}(t) = ss.s$ and $\text{targetState}(t) = ss'.s$ then
15: Assert $t$ is marked as “Modified” or “Deleted”
16: Remove $l$ and $ss'$ with all its children from $SET^{\text{baseVer}}$
17: if $t$ is marked as “Modified” then
18: //Mark $ss$ as “needsUpdate”
19: $ss.needsUpdate \leftarrow \text{true}$
20: //Add state $s$ and its outgoing transition $t$ to the hashmap $branchesTo_Update$ of $ss$
21: $ss.branchesTo_Update \leftarrow ss.branchesTo_Update \cup \{s \rightarrow \{t\}\}$
22: end if
23: else
24: $q \leftarrow \text{enqueue}(q, ss')$
25: end if
26: end for
27: end while
28: //Task 2: Iterate over $SET^{\text{baseVer}}$ and set the attributes $needsUpdate$ and $branchesTo_Update$ of all leaf symbolic states that are no longer subsumed by any of pre-visited nodes to “true” and “All”, respectively
29: $SET^{\text{baseVer}} = \text{Task2_helperMethod}(SET^{\text{baseVer}})$
30: //Task 3: Re-explore $SET^{\text{baseVer}}$ to merge the results from the standard symbolic execution of $MLM^{\text{modVer}}$ for the parts that need to be updated
31: $q \leftarrow [ss_0]$
32: //Create a set of pre-visited symbolic states for the purpose of subsumption checking
33: $SS^{\text{prevVisisted}} \leftarrow \{\}$
Algorithm 7 Memoization-based Symbolic Execution (MSE) of an Evolving MLM (continued)

34: while q is not empty do
35:   Remove the first element ss = (s, val, pc) from q
36: if ss has one or more outgoing symbolic transitions then
37:   //Record ss in the set of pre-visited symbolic states
38:   SSpreVisited ← SSpreVisited ∪ {ss}
39:   //Add all the target symbolic states of the outgoing symbolic transitions of ss to the queue
40:   for all outgoing symbolic transitions st = (ss, l, ss') of ss do
41:     q ← enqueue(q, ss')
42:   end for
43: end if
44: end for
45: end if
46: if ss.needsUpdate then
47:   //Re-run the SE on the model version MLMmodVer taking into consideration only transitions in ss.branchesToUpdate
48:   for all outgoing transitions t = (s, e, eA, G, A, s') of s ∧ t ∈ ss.branchesToUpdate do
49:     Create a unique set V* of new variables in one-to-one relation with the transition’s event arguments eA
50:     Create a map σ between eA and V*
51:     Substitute every occurrence of a variable x ∈ eA in the guard G and the action code A by σ(x) to obtain G* ← σ(G) and A* ← σ(A)
52:     if G* is satisfiable given val and pc then
53:       feasiblePaths ← SymbolicExecutionOfCode(A*, val, G*, pc)
54:       for all feasiblePath = (val', pc', out') ∈ feasiblePaths do
55:         Create a new symbolic state, ss' = (s', val', pc')
56:         if ss' is not subsumed by any previously generated symbolic state in SSpreVisited then
57:           ss'.needsUpdate ← true
58:           ss'.branchesToUpdate ← "All"
59:           SS ← SS ∪ {ss'}
60:           Create a new label, l' = (e, V*, σ, out')
61:           L ← L ∪ {l'}
62:           Create a new symbolic transition, st' = (ss, l', ss')
63:           ST ← ST ∪ {st'}
64:           q ← enqueue(q, ss')
65:         end if
66:       end for
67:     end if
68:   end for
69: end if
70: end while
71: //Task 4 (Optional): Iterate over SETbaseVer and remove any node that can be subsumed by any other nodes (either at the same hierarchical level or higher)
72: SETmodVer = Task4(helperMethod(SETbaseVer))
73: MSETmodVer ← SETbaseVer

Task 1: The first task is to load and explore the input SETbaseVer (resp., GSE–TbaseVer) in order to: 1) remove all the edges representing any transition belonging to the set of updated transitions Tupdated (note that removing an edge leads to removing the entire sub-tree rooted at the target node of the edge) and 2) mark all symbolic states (resp., symbolic model states) representing states with outgoing transitions belonging to Tupdated as "needing an update" with respect to these outgoing transitions.
5.2. PROPOSED SYMBOLIC EXECUTION OPTIMIZATIONS FOR EVOLVING RHAPSODY STATECHARTS

Algorithm 8 Memoization-based Symbolic Execution (MSE) of an Evolving CMLM

Inputs:
1: A modified CMLM model version of \( n \) communicating MLMs: \( M_2 = (\text{MLM}^{M_2}, Q^{M_2}, s_0^{M_2}, V^{M_2}, V_0^{M_2}) \)
2: A GSET of the previous CMLM model version \( M_1 \): \( \text{GSET}^{M_1} = (\text{SS}^{M_1}, \text{L}, \text{ST}, s_0^{M_1}) \)
3: A set of all changed transitions between \( M_2 \) and \( M_1 \): \( T^{\text{updated}} \)

Output:
1: A memoization-based GSET of \( M_2 \): \( \text{MGSET}^{M_2} \)

Steps:
1: //Task 1: Explore \( \text{GSET}^{M_1} \) in breadth-first-search fashion to remove the parts (i.e., the sub-trees) that need to be updated and to mark the nodes in \( \text{GSET}^{M_1} \) to be re-explored when we re-run the standard SE on \( M_2 \)
2: \( s_0^{M_1} \leftarrow (s_0^{M_1}, V_0^{M_1}, \text{true}, q_0^{M_1}) \)
3: Create a queue \( q \)
4: \( q \leftarrow [s_0^{M_1}] \)
5: while \( q \) is not empty do
6: Remove the first element \( s s^{M_1}=(s^{M_1}, \text{val}, pc, q^{M_1}) \) from \( q \)
7: if there is \( s1 \) in \( s^{M_1} \) and there \( t \) in \( T^{\text{updated}} \) such that \( s1 = \text{sourceState}(t) \) and \( t \) is marked as “Added” then
8: //Mark \( ss^{M_1} \) as “needsUpdate”
9: \( ss^{M_1}.\text{needsUpdate} \leftarrow \text{true} \)
10: //Add state \( s1 \) and its outgoing transition \( t \) to the hashmap \( \text{branchesToUpdate} \) of \( ss^{M_1} \)
11: \( ss^{M_1}.\text{branchesToUpdate} \leftarrow ss^{M_1}.\text{branchesToUpdate} \cup \{s \rightarrow \{t\}\} \)
12: end if
13: for all outgoing symbolic transitions \( st = (ss^{M_1}, l, ss^{M_1}) \) of \( ss^{M_1} \) do
14: if there is \( t \in T^{\text{updated}} \) such that sourceState(\( t \)) \( \in ss^{M_1}, st \) and targetState(\( t \)) \( \in ss^{M_1}, st \) then
15: Assert \( t \) is marked as “Modified” or “Deleted”
16: Remove \( l \) and \( ss^{M_1} \) with all its children from \( \text{SET}^{M_1} \)
17: if \( t \) is marked as “Modified” then
18: //Mark \( ss^{M_1} \) as “needsUpdate”
19: \( ss^{M_1}.\text{needsUpdate} \leftarrow \text{true} \)
20: //Add state \( s1 \) and its outgoing transition \( t \) to the hashmap \( \text{branchesToUpdate} \) of \( ss^{M_1} \)
21: \( ss^{M_1}.\text{branchesToUpdate} \leftarrow ss^{M_1}.\text{branchesToUpdate} \cup \{s \rightarrow \{t\}\} \)
22: end if
23: else
24: \( q \leftarrow \text{enqueue}(q, ss^{M_1}) \)
25: end if
26: end for
27: end while
28: //Task 2: Iterate over \( \text{GSET}^{M_1} \) and set the attributes \( \text{needsUpdate} \) and \( \text{branchesToUpdate} \) of all leaf symbolic states that are no longer subsumed by any of pre-visited nodes to “true” and “All”, respectively
29: \( \text{GSET}^{M_1} = \text{Task2\_helperMethod} (\text{GSET}^{M_1}) \)
30: //Task 3: Re-explore \( \text{GSET}^{M_1} \) to merge the results from the symbolic execution of \( M_2 \) for the parts that need to be updated
31: \( q \leftarrow [s_0^{M_1}] \)
32: //Create a set of pre-visited symbolic states for the purpose of subsumption checking
33: \( \text{SS}^{\text{preVisited}} \leftarrow \{\} \)
Algorithm 8 Memoization-based Symbolic Execution (MSE) of an Evolving CMLM (continued)

34: while \( q \) is not empty do
35: \hspace{1em} Remove the first element \( ss^{M1} = (s^{M1}, val, pc, q^{M1}) \) from \( q \)
36: \hspace{1em} if \( ss^{M1} \) has one or more outgoing symbolic transitions then
37: \hspace{2em} //Record \( ss^{M1} \) in the set of pre-visited symbolic model states
38: \hspace{2em} \( S_{\text{preVisited}}^{\text{visited}} \leftarrow S_{\text{preVisited}}^{\text{visited}} \cup \{ss^{M1}\} \)
39: \hspace{2em} //Add all the target symbolic states of the outgoing symbolic transitions of \( ss^{M1} \) to the queue
40: \hspace{2em} for all outgoing symbolic transitions \( st = (ss^{M1}, l, ss^{M1}) \) of \( ss^{M1} \) do
41: \hspace{3em} \( q \leftarrow \text{enqueue}(q, ss^{M1}) \)
42: \hspace{2em} end for
43: \hspace{2em} end if
44: \hspace{1em} if \( ss^{M1}.\text{needsUpdate} \) then
45: \hspace{2em} //Re-run the SE on the model version \( M2 \) taking into consideration only the states and the transitions in the hashmap \( \text{branchesTo}Update \) of \( ss^{M1} \)
46: \hspace{3em} for all MLMj in MLM\( ^{M2} \) do
47: \hspace{4em} \( s_i \leftarrow s^{M2}(i) \) where \( s_i \in S_i(MLMj) \)
48: \hspace{4em} if \( s_i \) is found in the \( \text{branchesTo}Update \) map of \( ss^{M1} \) then
49: \hspace{5em} for all outgoing transitions \( t_i = (s_i, e, A, G, A, s_i') \) of \( s_i \) do
50: \hspace{6em} if \( t_i \) is found in the \( \text{branchesTo}Update \) map of \( ss^{M1} \) then
51: \hspace{7em} if \( ((q^s = \phi) \land (e \in E_{\text{inp}}^M)) \lor ((\text{head}(q^s)) = e) \land (e \in E_{\text{out}}^M) \) then
52: \hspace{8em} Create a unique set \( V^A \) of new variables in one-to-one relation with the transition’s event arguments \( eA \)
53: \hspace{8em} Create a map \( \sigma \) between \( eA \) and \( V^A \)
54: \hspace{8em} Substitute every occurrence of a variable \( x \in eA \) in the guard \( G \) and the action code \( A \) by \( \sigma(x) \)
55: \hspace{8em} to obtain \( G^s \leftarrow \sigma(G) \) and \( A^s \leftarrow \sigma(A) \)
56: \hspace{8em} if \( G^s \) is satisfiable given \( val \) and \( pc \) then
57: \hspace{9em} feasiblePaths \leftarrow \text{Symbolic} ExecutionOfCode\( (A^s, val, G^s, pc) \)
58: \hspace{9em} for all \( \text{feasiblePath} = (val', pc', out') \) \in \text{feasiblePaths} do
59: \hspace{10em} \( q^{M2} \leftarrow \text{out}' \)
60: \hspace{10em} Create a new symbolic model state \( ss^{M2} = (s^{M2}, val', pc', q^{M2}) \) where \( s^{M2} = \phi \)
61: \hspace{10em} if \( ss^{M2} \) is not subsumed by any previously generated symbolic model state in \( S_{\text{preVisited}} \) then
62: \hspace{11em} Mark \( ss^{M2} \) as “needsUpdate” with respect to “All” states in \( s^{M2} \) and “All” their outgoing transitions
63: \hspace{11em} \( S_{\text{M1}} \leftarrow S_{\text{M1}} \cup \{ss^{M2}\} \)
64: \hspace{11em} Create a new label \( l' = (e, V^A, \sigma, out') \)
65: \hspace{11em} \( L \leftarrow L \cup \{l'\} \)
66: \hspace{11em} Create a new symbolic transition \( st' = (ss^{M1}, l', ss^{M2}) \)
67: \hspace{11em} \( ST \leftarrow ST \cup \{st'\} \)
68: \hspace{11em} //Record \( ss^{M2} \) in the set of pre-visited symbolic model states
69: \hspace{11em} \( S_{\text{preVisited}}^{\text{visited}} \leftarrow S_{\text{preVisited}}^{\text{visited}} \cup \{ss^{M2}\} \)
70: \hspace{9em} \( q \leftarrow \text{enqueue}(q, ss^{M2}) \)
71: \hspace{10em} end if
72: \hspace{10em} end if
73: \hspace{10em} end if
74: \hspace{10em} end if
75: \hspace{9em} end for
76: \hspace{9em} end if
77: \hspace{9em} end for
78: \hspace{9em} end if
79: \hspace{9em} end while
80: //Task 4 (Optional): Iterate over \( GSET^{M1} \) and remove any node that can be subsumed by any other nodes (either at the same hierarchical level or higher)
81: \( GSET^{M1} = \text{Task4}_{\text{helperMethod}}(GSET^{M1}) \)
82: \( MGSET^{M2} \leftarrow GSET^{M1} \)
This is done by updating the attributes \texttt{needsUpdate} and \texttt{branchesToUpdate} of each such symbolic state (resp., symbolic model state).

\textbf{Task 2:} As a result of the previous task, we may find some leaves in the resulting \texttt{SET_{baseVer}} (resp., \texttt{GSET_{baseVer}}) that lost their subsuming nodes. In this case, we need also to mark these leaves as “needing an update” with respect to all outgoing transitions of their corresponding model states. Therefore, the second task of both components is to iterate over the nodes in the \texttt{SET_{baseVer}} (resp., \texttt{GSET_{baseVer}}) resulting from the previous task and to set the attributes \texttt{needsUpdate} and \texttt{branchesToUpdate} of all leaf symbolic states (resp., symbolic model states) that are no longer subsumed to “true” and “All”, respectively. A detailed description of this task is omitted for brevity.

\textbf{Task 3:} The third task of both components is to re-explore the \texttt{SET_{baseVer}} (resp., \texttt{GSET_{baseVer}}) resulting from the previous task to find all symbolic states (resp., symbolic model states) that need an update. For each such symbolic state (resp., symbolic model state), we rerun the SE technique on the modified model version \texttt{MLM_{modVer}} (resp., \texttt{CMLMs_{modVer}}) targeting only the parts of the model identified by the hashmap \texttt{branchesToUpdate} of the symbolic state (resp., symbolic model state) and taking into consideration the set of symbolic states (resp., symbolic model states) that have been previously explored to be used for the subsumption checking. Our strategy to implement this task differs from the one that we initially used to implement the “\textit{KLEE-based MLM2MSET Generation}” component in an earlier version of this work [53]. This modification is done based on the intuition that we got from applying the MSE technique for individual Statecharts where we found that our \textit{Initial Nodes Update Strategy} may generate, in some cases, extra symbolic states. The process
of detecting and removing these extra symbolic states may reduce the efficiency of
the technique especially for large SETs. Additionally, when we followed the same
strategy in the implementation of the “KLEE-based MLM2MSET Generation” com-
ponent, we found, based on an initial evaluation of the technique, that it generates
larger MGSETs with many extra symbolic model states. Therefore, we have adapted
this initial nodes update strategy in the implementation of the two components:
“KLEE-based MLM2MSET Generation” and “KLEE-based MLM2MGSET Generation”
presented in this thesis such that we overcome, to some extent, this problem. The
difference between both strategies lies in the way we generate the sub-trees required
to update the symbolic states (resp., symbolic model states) that need an update. In
the following, we explain this in details.

1. **Initial Nodes Update Strategy (INUS):** Our initial nodes update strategy
depends on generating the sub-trees required to update symbolic states (resp.,
symbolic model states) that are marked as “needing an update” independently
(i.e., one at a time and in a breadth-first manner) by running an adapted version
of the SE on the modified model version based on some additional parameters.
These additional parameters determine where the exploration starts (i.e., the
symbolic state to begin the exploration at), which updated transitions to con-
sider when exploring the state representing the start symbolic state for the first
time, and the set of symbolic states that have been previously explored to be
used for the subsumption checking. The resulting sub-SETs for running this
adapted version of the SE are to be merged with their corresponding symbolic
states in the SET$^\text{baseVer}$. One limitation of this strategy is that it prevents the
detection of existing subsumption relationships between symbolic states in the
sub-SETs that were generated first and symbolic states in the sub-SETs that are generated later.

2. **Adapted Nodes Update Strategy (ANUS):** One possible solution to overcome the limitation presented in the initial nodes update strategy is to run the SE on the modified model version for symbolic states (resp., symbolic model states) that need an update on-the-fly as they are identified. In this case, the construction of the sub-SETs for the nodes that need an update is carried out jointly (concurrently) at the same time rather than successively as in the initial strategy. This procedure is implemented in Algorithm 7, Lines 30-68 and in Algorithm 8, Lines 30-79. This strategy can overcome part (but not all) of the limitation raised by the initial strategy (i.e., we may also find some extra symbolic model states in the generated SETs following this strategy as well).

A step-by-step illustration of both update strategies is shown in Figures 5.9 and 5.10, respectively, for updating nodes 2 and 5 in the example SET shown in Figure 5.8. Please note that: 1) nodes are labelled based on their creation order and 2) leading zeros in a node label indicate the number of reruns of the SE technique that created the node. A comparison between the SETs generated after applying each update strategy is shown in Figure 5.11. A demonstration of a scenario of an undetected subsumption relation between existing and newly created nodes following any of the two presented nodes update strategies is shown in Figure 5.12. In this scenario, the root node “0” is marked as “needing an update” with respect to a newly added branch which is colored red. Following either update strategies will generate the sub-tree consisting of nodes “001”, “002” and “003”. Assume the case where a subsumption relation now exists between the already existing node “5” and the newly created node “001”.
5.2. PROPOSED SYMBOLIC EXECUTION OPTIMIZATIONS FOR EVOLVING RHAPSODY STATECHARTS

Table 5.1: Components’ tasks and their corresponding algorithms’ steps

<table>
<thead>
<tr>
<th>Task</th>
<th>“KLEE-based CMLM2MSET Generation”</th>
<th>“KLEE-based CMLM2MGSET Generation”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>Algorithm 7, Lines 1-27</td>
<td>Algorithm 8, Lines 1-27</td>
</tr>
<tr>
<td>Task 2</td>
<td>Algorithm 7, Lines 28-29</td>
<td>Algorithm 8, Lines 28-29</td>
</tr>
<tr>
<td>Task 3</td>
<td>Algorithm 7, Lines 30-68</td>
<td>Algorithm 8, Lines 30-79</td>
</tr>
<tr>
<td>Task 4 (Optional)</td>
<td>Algorithm 7, Lines 69-70</td>
<td>Algorithm 8, Lines 80-81</td>
</tr>
</tbody>
</table>

In this case, nodes “02”, “03”, “06” and “07” are considered extra nodes. This is because the newly detected subsumption relation between nodes “5” and “001” does not exist before the creation of node “001” and also when the generation of nodes “02”, “03”, “06” and “07” took place.

**Task 4:** Since the resulting SETs (resp., GSETs) of the previous task may contain some extra symbolic states (resp., symbolic model states), the last task to be performed in both components is to detect and remove such extra symbolic states (resp., symbolic model states). A detailed description of this task is straightforward and omitted for brevity.

In Table 5.1, we highlight the steps of the algorithms implementing each one of the aforementioned four tasks of both components.

The output from both components is a memoization-based SET (resp., GSET) of the modified model version, \( \text{MSET}^{\text{modVer}} \) (resp., \( \text{MGSET}^{\text{modVer}} \)), that shares all what can be reused from the old tree \( \text{SET}^{\text{baseVer}} \) (resp., \( \text{GSET}^{\text{baseVer}} \)) but also contains the modifications resulting from the SE of the parts of the new version of the model that are found changed. We use a tree-based data structure for representing, storing, retrieving, and manipulating these symbolic execution trees.

**Example 5.** Applying MSE on the modified model of individual Statechart V1 shown
5.2. PROPOSED SYMBOLIC EXECUTION OPTIMIZATIONS FOR EVOLVING RHAPSODY STATECHARTS

Figure 5.8: An example SET with nodes 2 and 5 are marked as “needing an update” with respect to some of their outgoing branches. Nodes are labeled based on their creation order.

in Figure 5.1(b) and on the modified model of communicating Statecharts $AB^{v1}$ shown in Figure 5.4 generates the memoization-based SET (resp., GSET) that are shown in Figures 5.3(a) and 5.5, respectively. The amount of savings gained from applying MSE on the modified model $V1$ (or $AB^{v1}$) compared to applying standard SE on the same models is measured by the percentage of reused nodes (which are colored black) to the total number of nodes in the resulting MSET (or MGSET). We can see that the savings obtained in the MSET($V1$) in Figure 5.3(a) is equal to 6/14 (i.e., approximately 43 %), while the savings gained in the MGSET($AB^{v1}$) in Figure 5.5 is equal to 11/21 (i.e., approximately 52 %).

It is quite clear that in some cases the savings gained from MSE can be very minimal depending on how many tree nodes are reused and the location of tree nodes that need to be updated. In this case, it is the analyst decision to choose between using MSE or standard SE.
5.2. PROPOSED SYMBOLIC EXECUTION OPTIMIZATIONS FOR EVOLVING RHAPSODY STATECHARTS

(a) Step 1: Re-exploring node 2 given pre-visited nodes = \{0, 1, 2, 3, 4, 5, 6\}

(b) Step 2: Merging the sub-tree updating node 2 with the original SET

(c) Step 3: Re-exploring node 5 given pre-visited nodes = \{0, 1, 2, 3, 4, 5, 6, 01, 02, 03, 04, 05\}

(d) Step 4: Merging the sub-tree updating node 5 with the original SET

Figure 5.9: A demonstration of the Initial Nodes Update Strategy used in the implementation of an earlier version of MSE [53, 54]. Nodes are labelled based on their creation order and leading zeros in a node label indicate the number of reruns of the SE technique that created the node.
5.2. PROPOSED SYMBOLIC EXECUTION OPTIMIZATIONS FOR EVOLVING RHAPSODY STATECHARTS

(a) Step 1: One-level depth exploration of node 2 given pre-visited nodes = \{0, 1, 2, 3, 4, 5, 6\}. The resulting node 01 is marked (with double circles) as “needing an update”.

(b) Step 2: One-level depth exploration of node 5 given pre-visited nodes = \{0, 1, 2, 3, 4, 5, 6, 01\}. The resulting node 02 is marked (with double circles) as “needing an update”, while the resulting node 03 is found to be subsumed by one of the given pre-visited nodes.

(c) Step 3: One-level depth exploration of node 01 given pre-visited nodes = \{0, 1, 2, 3, 4, 5, 6, 01, 02, 03\}. The resulting nodes 04 and 05 are both found to be subsumed by one of the given pre-visited nodes. Note that subsumption relation between nodes 04 and 02 is detected.

(d) Step 4: One-level depth exploration of node 02 given pre-visited nodes = \{0, 1, 2, 3, 4, 5, 6, 01, 02, 03, 04, 05\}. The resulting nodes 06 and 07 are both found to be subsumed by one of the given pre-visited nodes.

Figure 5.10: A demonstration of the Adapted Nodes Update Strategy used in the implementation of the current version of MSE as presented in Algorithms 7 and 8 – One-level depth exploration with marking generated nodes (if not subsumed by a pre-visited node) as “needing an update”. Nodes are labelled based on their creation order and leading zeros in a node label indicate the number of reruns of the SE technique that created the node.
5.2. PROPOSED SYMBOLIC EXECUTION OPTIMIZATIONS FOR EVOLVING RHAPSODY STATECHARTS

(a) Less optimized MSET resulting from our Initial Nodes Update Strategy

(b) A more optimized MSET resulting from our Adapted Nodes Update Strategy

Figure 5.11: Comparison between our two nodes update strategies showing (on the left) a demonstration of undetected subsumption Relation (SR) between nodes 02 and 001 resulting in having extra nodes 04 and 05

Figure 5.12: A demonstration of a possible case of undetected subsumption relation (SR) between nodes 5 and 001 resulting in having extra nodes 02, 03, 06 and 07
5.2. PROPOSED SYMBOLIC EXECUTION OPTIMIZATIONS FOR EVOLVING RHAPSODY STATECHARTS

Discussion

We consider MSE an example of an artifact-oriented optimization technique because it can be used to optimize the generation of one type of intermediate artifact (in our case they are in the form of symbolic execution trees) as the system evolves. At any point in time, these intermediate artifacts are maintained (i.e., up to date) and complete (i.e., they represent all execution paths of their corresponding models) and thus they can be used to run any SET-based analyses.

Nevertheless, by marking the new parts of the generated MSETs (resp., MGSETs) resulting from MSE from the reused parts, we can use the new parts to run regression-based analyses. In that sense, MSE can also be considered as an example of an analysis-oriented optimization technique meaning that if a given analysis has been performed on a given model before the change, then using only the new parts of the generated MSETs (resp., MGSETs) of the model after the change will allow us to rerun the same analysis and to conclude the overall correctness of the changed model.

5.2.2 Dependency-based Symbolic Execution (DSE)

In this section we present our dependency-based symbolic execution technique (DSE)\(^1\) for both individual and communicating Statecharts. The architectures of the implementation of the proposed approach are shown in Figures 5.13(a) and 5.13(b), respectively.

\(^1\)Note that we can also call it regression or partial symbolic execution.
5.2. PROPOSED SYMBOLIC EXECUTION OPTIMIZATIONS FOR EVOLVING RHAPSODY STATECHARTS

(a) DSE architecture for evolving individual Statecharts

(b) DSE architecture for evolving communicating Statecharts

Figure 5.13: Dependency-based Symbolic Execution (DSE) proposed architecture
5.2. PROPOSED SYMBOLIC EXECUTION OPTIMIZATIONS FOR EVOLVING RHAPSODY STATECHARTS

Technique Description

The main idea of this technique is to generate a sub-set (or a partial view) of the complete SET of a modified model version that characterizes all model paths that are impacted by the change which is sufficient to perform regression-based analyses. To achieve this, we integrate a dependency-based change impact analysis with our standard SE technique such that we can direct the successive runs of the SE technique to fully explore model transitions that are impacted by the change and to partially explore model transitions that are not impacted by the change. A full exploration of a model transition means to consider all the symbolic execution paths resulting from symbolically executing the transition guard and action code, while a partial exploration means to consider only one of these symbolic paths, namely the first one. Note that only models with multi-path transitions (i.e., transitions with disjoint guard conditions and/or conditional statements in their action code) can benefit from this technique. We found that this model feature is extensively used in many of existing Simulink/Stateflow models. In our model examples in Section 5.1, transitions $T_1$ and $T_2$ in Figure 5.1 (resp., transition $itsA.T_1$ in Figure 5.4(a)) are all examples of multi-path transitions. The SE of each one of these transitions results in two symbolic paths originating at the same symbolic state (resp., symbolic model state) representing the source state of the transition and ending at two different symbolic states (resp., symbolic model states) representing the target state of the transition, each of which is characterized by the valuations of variables, the path constraints and the updates of queues resulting from the symbolic evaluation of each branch in the guard condition and in the conditional and iterative statements found in the action code of the transition.
5.2. PROPOSED SYMBOLIC EXECUTION OPTIMIZATIONS FOR EVOLVING RHAPSODY STATECHARTS

Based on this idea we can conclude that the key requirements of this technique are first to identify the differences between two model versions and second to perform a dependency analysis on the modified model version to compute the impact of these differences. These two requirements are achieved by the two components “Model Differencing” and “Dependency-based Change Impact Analysis” shown in Figures 5.13(a) and 5.13(b).

To implement the “Model Differencing” component, we follow the same procedure as in the previous section. First we use the Rhapsody DiffMerge to find the differences between two Rhapsody models, then we use our “SC2MLM DiffMap” (resp., “RhapM2CMLM DiffMap”) component to map these differences to their MLMs (resp. CMLMs) correspondences. The output in this case, as we explained earlier, represents the set $T_{\text{updated}}$ of all transitions that are found to have been updated in the modified MLM (resp., CMLMs) version. Each element in the set $T_{\text{updated}}$ indicates the name of a transition that has been affected by the change, the name of the source state of this transition, and the change type applied to this transition (i.e., “Addition”, “Deletion”, “Modification”).

To implement the “Dependency-based Change Impact Analysis” component, we considered the two types of dependences discussed in Section 2.4: data dependence (DD) and communication dependence (COMD). The formal definitions of both types of dependences are provided in Section 2.4 and the algorithms to compute them are listed in Appendix A (Algorithms 3 and 4). The two inputs to this component are: 1) the MLM (resp., CMLMs) representation of the modified model version MLM$^{\text{modVer}}$ (resp., CMLM$^{\text{modVer}}$), and 2) the set of transitions $T_{\text{updated}}$ that have been updated in the modified model version. The first task to be performed by this component is
5.2. PROPOSED SYMBOLIC EXECUTION OPTIMIZATIONS FOR EVOLVING RHAPSODY STATECHARTS

...to compute the data dependences and the communication dependences between the transitions of the input model version. The second task to be performed is to identify the set of transitions $T^{\text{impacted}}$ that have a dependency relation (DD or COMD) with any of the transitions in $T^{\text{updated}}$. The output of this component is the union set of $T^{\text{updated}}$ and $T^{\text{impacted}}$; these transitions are the ones that we need to fully explore when we run our DSE technique on the modified model version MLM$^{\text{modVer}}$ (resp., CMLM$^{\text{modVer}}$).

The last task to be performed is to run the symbolic execution of the modified version of the model guided by the list of updated/impacted transitions resulting from the “Dependency-based Change Impact Analysis” component. Two modes of exploration are defined: “full” and “partial”. The “full” exploration mode requires a complete exploration of all execution paths (i.e., symbolic transitions) of a given transition and is applied to all transitions that are found to have been updated or impacted. However, the “partial” exploration mode requires the execution of only one path (i.e., symbolic transition) of an explored transition and is applied to transitions that are found neither updated nor impacted. The components performing this task are the “KLEE-based MLM2DSET Generation” component shown in Figure 5.13(a) for individual Statecharts and the “KLEE-based CMLM2DGSET Generation” component shown in Figure 5.13(b) for communicating Statecharts. The algorithms implementing this task for each component are listed in Algorithm 9 and Algorithm 10, respectively.

**Example 6.** Applying our dependency-based change impact analysis on the model version $V1$ will not report transition $T1$ among the list of transitions that are data dependent on the modified transition $T2$. As a result, only a partial exploration of
5.2. PROPOSED SYMBOLIC EXECUTION OPTIMIZATIONS FOR EVOLVING RHAPSODY STATECHARTS

Algorithm 9 Dependency-based Symbolic Execution (DSE) of an Evolving MLM

Input:
1: A modified MLM model version: $MLM^{modVer} = (S, V, E, EA, T, s_0, V_0)$
2: A set of all changed transitions and their impacted ones: $T^{ofInterest} = T^{updated} \cup T^{impacted}$

Output:
1: A Dependency-based Symbolic Execution Tree of $MLM^{modVer}$: $DSET^{modVer} = (SS, L, ST, ss_0)$

Steps:
1: $ss_0 \leftarrow (s_0, V_0, \text{true})$
2: $SS \leftarrow \{ss_0\}$
3: $L \leftarrow \{}$
4: $ST \leftarrow \{}$
5: Create a queue $q$
6: $q \leftarrow [ss_0]$
7: //Explore the MLM in Breadth-First-Search fashion
8: while $q$ is not empty do
9: Remove the first element $ss = (s, val, pc)$ from $q$
10: for all outgoing transitions $t = (s, e, eA, G, A, s')$ of $s$ do
11: Create a unique set $V_s$ of new variables in one-to-one relation with the transition event’s arguments $eA$
12: Create a map $\sigma$ between $eA$ and $V_s$
13: Substitute every occurrence of a variable $x \in eA$ in the guard $G$ and the action code $A$ by $\sigma(x)$ to obtain $G^s \leftarrow \sigma(G)$ and $A^s \leftarrow \sigma(A)$
14: if $G^s$ is satisfiable given $val$ and $pc$ then
15: $feasiblePaths \leftarrow \text{SymbolicExecutionOfCode}(A^s, val, G^s, pc)$
16: if $t \in T^{ofInterest}$ then
17: //Full-exploration mode
18: for all $feasiblePath = (val', pc', out') \in feasiblePaths$ do
19: Create a new symbolic state, $ss' = (s', val', pc')$
20: if $ss'$ is not subsumed by any previously generated state then
21: $SS \leftarrow SS \cup \{ss'\}$
22: Create a new label, $l' = (e, V^s, \sigma, out')$
23: $L \leftarrow L \cup \{l'\}$
24: Create a new symbolic transition, $st' = (ss, l', ss')$
25: $ST \leftarrow ST \cup \{st'\}$
26: $q \leftarrow \text{enqueue}(q, ss')$
27: end if
28: end for
29: else
30: //Partial-exploration mode
31: Select only one $feasiblePath = (val', pc', out') \in feasiblePaths$
32: Create a new symbolic state, $ss' = (s', val', pc')$
33: if $ss'$ is not subsumed by any previously generated state then
34: $SS \leftarrow SS \cup \{ss'\}$
35: Create a new label, $l' = (e, V^s, \sigma, out')$
36: $L \leftarrow L \cup \{l'\}$
37: Create a new symbolic transition, $st' = (ss, l', ss')$
38: $ST \leftarrow ST \cup \{st'\}$
39: $q \leftarrow \text{enqueue}(q, ss')$
40: end if
41: end if
42: end if
43: end for
44: end while
5.2. PROPOSED SYMBOLIC EXECUTION OPTIMIZATIONS FOR EVOLVING RHAPSODY STATECHARTS

Algorithm 10 Dependency-based Symbolic Execution (DSE) of an Evolving CMLM

Input:
1. A modified CMLM model version of n communicating MLMs: \( M = (CMLM^M, Q^M, s_0^M, V^M, V_0^M) \)
2. A set of all changed transitions and their impacted ones: \( T_{of\, interest} = T_{updated} \cup T_{impacted} \)

Output:
1. A Dependency-based Global Symbolic Execution Tree of \( M \): \( DGSET^M = (SS^M, L, ST, ss_0^M) \)

Steps:
1. \( ss_0^M \leftarrow (s_0^M, V_0^M, true, q_0^M) \)
2. \( SS^M \leftarrow \{ ss_0^M \} \)
3. \( L \leftarrow \{ \} \)
4. \( ST \leftarrow \{ \} \)
5. Create a queue \( q \)
6. \( q \leftarrow \{ ss_0^M \} \)
7. //Breadth-First-Search fashion exploration for each MLM in \( M \)
8. while \( q \) is not empty do
9. for all MLMs in \( MLMs \) in \( M \) do
10. \( s_i \leftarrow s_i^M(i) \) where \( s_i \in S_i(MLM_i) \)
11. for all outgoing transitions \( t_i = (s_i, e, eA, G, A, s_i') \) of \( s_i \) do
12. if \( (q_i^M = \phi) \cap (e \in E_{inp}^i) \lor ((head(q_i^M) = e) \land (e \in E_{out}^i)) \) then
13. Create a map \( \sigma \) between \( eA \) and \( V^i \)
14. Substitute every occurrence of a variable \( x \in eA \) in the guard \( G \) and the action code \( A \) by \( \sigma(x) \) to obtain \( G^i \leftarrow \sigma(G) \) and \( A^i \leftarrow \sigma(A) \)
15. if \( G^i \) is satisfiable given \( val \) and \( pc \) then
16. \( feasiblePaths \leftarrow SymbolicExecutionOfCode(A^i, val, G^i, pc) \)
17. for all \( feasiblePath = (val', pc', out') \) in \( feasiblePaths \) do
18. Create a new symbolic state, \( ss^M = (s^M, val', pc', q^M) \) where \( s^M = s^M.replace(s_i, s_i') \)
19. if \( ss^M \) is not subsumed by any previously generated symbolic model state in \( SS^M \) then
20. \( SS^M \leftarrow SS^M \cup \{ ss^M \} \)
21. Create a new label, \( l' = (e, V^i, \sigma, out') \)
22. \( L \leftarrow L \cup \{ l' \} \)
23. Create a new symbolic transition, \( st' = (ss^M, l', ss^M) \)
24. \( ST \leftarrow ST \cup \{ st' \} \)
25. \( q \leftarrow enqueue(q, ss^M) \)
26. end if
27. end for
28. else
29. //Partial-exploration mode
30. Select only one \( feasiblePath = (val', pc', out') \) in \( feasiblePaths \)
31. \( q^M \leftarrow \{ \} \)
32. Create a new symbolic state, \( ss^M = (s^M, val', pc', q^M) \) where \( s^M = s^M.replace(s_i, s_i') \)
33. if \( ss^M \) is not subsumed by any previously generated symbolic model state in \( SS^M \) then
34. \( SS^M \leftarrow SS^M \cup \{ ss^M \} \)
35. Create a new label, \( l' = (e, V^i, \sigma, out') \)
36. \( L \leftarrow L \cup \{ l' \} \)
37. Create a new symbolic transition, \( st' = (ss^M, l', ss^M) \)
38. \( ST \leftarrow ST \cup \{ st' \} \)
39. \( q \leftarrow enqueue(q, ss^M) \)
40. end if
41. end if
42. end if
43. end for
44. end for
45. end if
46. end if
47. end for
48. end if
49. end if
50. end for
51. end while
5.2. Proposed Symbolic Execution Optimizations for Evolving Rhapsody Statecharts

Transition $T_1$ is performed and since it is a multi-path transition, only one feasible path (i.e., one symbolic transition) is considered and the rest are pruned. Similarly, applying our dependency-based change impact analysis on the model version $AB^{V1}$ will not report transition $itsA_T1$ among the list of transitions that are data or communication dependent on the modified transition $itsA_T2$. As a result, only a partial exploration of transition $itsA_T1$ is performed and since it is a multi-path transition, only one feasible path (i.e., one symbolic transition) is considered and the rest are pruned. The amount of savings gained from applying DSE is measured by the percentage of nodes that are pruned during the partial exploration mode. We can see that the savings gained in the DSET($V1$) in Figure 5.3(b) is approximately equal to 43% which is the percentage of the 6 pruned nodes (these which are colored gray) to the total number of nodes of the complete SET which is 14 nodes. Similarly, the savings gained in the DGSET($AB^{V1}$) in Figure 5.6 is approximately equal to 43% which is the percentage of the 9 pruned nodes (these which are colored gray) to the total number of nodes of the complete GSET which is 21 nodes.

Discussion

Assume that the SET-based type of analysis required here is to check that the system has no dead ends. In a given SET, a tree leaf that is not subsumed by any other node in the tree may indicate: 1) reaching a pre-visited symbolic state, 2) reaching a depth bound of some loop or 3) reaching a dead end symbolic state. In our example, we neither have termination states nor loops in any of the action code of transitions. Therefore, in this case, a SET leaf node that is not subsumed by any other node will indicate a dead end symbolic state. Also, assume that an analyst has performed

---

$^{2}$Dead ends are states that have no feasible outgoing transitions and they are not end states.
5.3. MSE OR DSE?

Based on our discussion of the idea behind each technique, we conclude:

- For MSE, we need to have access to the SET (resp., GSET) of the original model (i.e., the model before the change) to be reused and we need to identify the differences between the new version of the model and the original one. We speculate that the effectiveness of this technique may be highly dependent on the change location. The resulting MSET (resp., MGSET) represents all execution paths of the modified model (i.e., the model after the change) and can thus be used to perform all SET-based analyses. However, if reestablishing a previous analysis result as the goal, a regression analysis can also be run on just the changed parts of the SET.

This type of analysis on the GSET(AB\textsuperscript{V0}) and has concluded that AB\textsuperscript{V0} has no dead ends. In this case, applying the same type of analysis on the DGSET(\textit{AB}\textsuperscript{V1}), which is a partial view of the GSET(AB\textsuperscript{V1}), will be sufficient to check whether AB\textsuperscript{V1} has a dead end or not. By checking the leaves of the DGSET(\textit{AB}\textsuperscript{V1}), we can see that the symbolic model state SS15 is not subsumed by any other node in the tree and thus we can conclude that \textit{AB}\textsuperscript{V1} has a dead end. By inspecting 1) the valuations of the variables and the contents of the queues of this symbolic model state (i.e., SS15) and 2) the outgoing transition \textit{itsB_T3} of state \textit{bS2} that is fired by the event \textit{e5}, we can see that the guard of this transition is unsatisfiable with respect to the given value of the event parameter. Since the DGSET(\textit{AB}\textsuperscript{V1}) represents only a subset of the complete GSET(\textit{AB}\textsuperscript{V1}), we cannot use it to perform a different type of analysis. In that sense, DSE is considered as an example of an analysis-oriented optimization approach.
For DSE, we need to identify the differences between the new version of the model and the original one as well as to integrate a dependence analysis to identify the impact of the change. We speculate that models with complex guards and action code may benefit most from this technique. As the resulting DSET (resp., DGSET) may not be complete and omit executions that are guaranteed to not have been impacted by the change, it can be used only to run regression-types of analysis.

Having said that, we think that both optimization techniques complement each other, in the sense that they both can serve the same purpose but are most useful under different circumstances, and it is the role of the analyst to choose among them. For instance, the availability of the SET from a previous analysis is a basic requirement for applying MSE, while the complexity of the guards or action code of the transitions in the model under study is a key factor for the effectiveness of DSE.

5.4 Summary

In this chapter we have presented two symbolic execution optimization techniques, MSE and DSE, for both evolving individual and communicating Statecharts. We first explained with examples the idea behind each technique. Second, we introduced our proposed architectures for both techniques and provided the required description of the different components in each architecture.
Chapter 6

Implementation

In this chapter we provide a high-level description of the implementation decisions that we have made to build the architectures presented in Chapter 4 for Standard SE and in Chapter 5 for MSE and DSE. Figure 6.1 shows the operating system required and the technology used to implement each of the components presented in these architectures. We first highlight the implementation decisions that we made based on the initial requirements of this research work. Second, we provide the details of three core tasks. These are: 1) the interaction with the symbolic execution engine KLEE, 2) building the model transformation and text generation in the context of IBM Rational Rhapsody RulesComposer, and 3) the interaction with the IBM Rational Rhapsody DiffMerge tool.

6.1 Implementation Decisions

Most of the implementation decisions that we made are based on the following two main requirements.

1. The context and the type of state-based models that we are going to use to
develop our research. The decision of this aspect was made based on an interest from our industrial partner to provide such analysis capability for the state-based models developed in the context of the IBM Rational Rhapsody MDE tool, which are known as Rhapsody Statecharts. We found that although working with proprietary and commercial tools has its advantage in terms of quality and documentation, they limit the choices one may have to develop some external functionality or process and manipulate the artifacts they create to some extent. Therefore, we decided to implement our required components
6.1. IMPLEMENTATION DECISIONS

using IBM related tools (e.g., IBM Rational Rhapsody Developer RulesComposer Add-On for model transformation and custom code generation, and IBM Rational Rhapsody DiffMerge for model differencing).

2. The symbolic execution engine to be reused for the action code encountered in the state-based models to be analyzed. Our decision was to use the KLEE symbolic execution engine – an open-source symbolic execution engine for C code. We made this decision after we examined the feasibility of building an API that enables the interaction with KLEE. In Section 6.2, we provide a more detailed description of this process. To be able to use KLEE for the symbolic execution of the fragments of action code encountered in Rhapsody Statecharts, we only consider a subset of Java and C++ code that has the same syntax as C code, including the basic assignment statements, conditional statements and iterative statements. The data types supported are the basic types including characters, Booleans and integer numbers. We also consider the type of statements that are used in the action code for sending events to other objects.

Unfortunately, KLEE does not run under Windows. It runs only on certain types of Linux distributions including Ubuntu but not Red Hat. On the other hand, the IBM Rational Rhapsody tools can be installed on Windows and only on the Red Hat Linux distribution. Therefore, we decided to add the Ubuntu VM to our Windows machine. The IBM Rational Rhapsody tools are to be installed on the Windows machine and the KLEE symbolic execution engine is to be installed on the Ubuntu VM. The decision of which component is to be implemented on which VM was made based on the functionality of each component and the development environment it
requires. Figure 6.1 highlights this categorization. For example, components that require the IBM infrastructure to operate are developed and run on Windows. Examples of these components are the “SC2MLM Transformation” and “RhapM2CMLM Transformation”, which are implemented in the context of the Rhapsody RulesComposer Add-On to perform the following two tasks: 1) to translate from Rhapsody Statecharts to Mealy-like Machines and 2) to compute the data and communication dependencies in and between the generated Mealy-like Machines. Similarly, components “SC2MLM DiffMap”, “RhapM2CMLM DiffMap” and “Dependency-based Change Impact Analysis” require the manipulation of the files representing the Mealy-like Machines generated from the transformation components and this requires access to the Mealy-like Machine meta-model and its related schema definitions which are tool-dependent generated artifacts. Thus, we have implemented these components as a Java project in the context of the Rhapsody RulesComposer Add-On. All other components which are the KLEE-based ones require the interaction with the KLEE symbolic execution engine and therefore they are developed and run on the Ubuntu VM.

In order to facilitate the communication between the part of the implementation that runs on the Ubuntu VM (where KLEE works) and the part that runs on the Windows VM (where the IBM Rational Rhapsody tool works), we use the Java Remote Method Invocation (RMI) system. We developed the part of the implementation that runs on the Ubuntu VM side as an RMI server API and the part of the implementation that runs on Window VM as an RMI client API which has been integrated with the IBM Rational Rhapsody tool as an external helper.
6.2 Interaction With KLEE

We referred to this task in the algorithms provided in Chapters 4 and 5 using the pseudo-code function: \texttt{SymbolicExecutionOfCode(A^s, val, G^s, pc)}. It identifies the information necessary to symbolically execute the action code and the guard of a transition given the variables’ valuation and the path constraints collected to reach a symbolic state corresponding to the source state of a transition. The output from executing this statement is a set of all feasible paths resulting from parsing the output from the KLEE symbolic execution engine which we use to create a set of symbolic transitions and their target symbolic states. In the sequel we explain in details how we implement this task using the illustrations in Figure 6.2.

In Figure 6.2(a), we show: 1) on the left, part of an example Statechart model that consists of two states: \( S2 \) and \( S3 \) which are connected by a transition \( T2 \) and 2) on the right, an example symbolic state representation \( SS4 \) of the transition’s source state \( S2 \). We also highlight the necessary information about the attribute(s) defined for the given model, the event \( e2 \) of transition \( T2 \) and its arguments (if any) and the symbolic variable(s) used in the given symbolic state representation \( SS4 \). In Figure 6.2(b), we show the flow diagram of how we interact with the KLEE symbolic execution engine to symbolically execute the action code of transition \( T2 \) given the symbolic state \( SS4 \). In this flow diagram, we identify two components: “KLEE Input File Preparation” and “Feasible Paths Data Extraction”.

The “KLEE Input File Preparation” component takes as inputs a transition to be symbolically explored and a symbolic state representing the source state of the
transition and generates as an output a C file named “TAC.c” which has at the beginning the declarations of all required variables, including those which are symbolic, and the action code statements of the input transition as well as the path constraints definition of the input symbolic state. Following this, we add, for every declared variable, a conditional statement to check whether its last assigned value is concrete or symbolic and to record in a user-defined output file named “TAC-output.txt” the variable name and its value if it is concrete or the keyword “SYMBOLIC” if it is symbolic. In order to get the symbolic expression corresponding to a symbolic variable, we use KLEE’s special function `klee_print_expr` which prints its output to the STDOUT stream of the running process. In order to manipulate this output, we redirect the STDOUT stream into a file named “klee-stdout.log”. The output C file “TAC.c” is to be compiled and executed by the KLEE symbolic execution engine. The output from the execution is recorded in an output directory named “klee-out-0” which is created by KLEE. We also use this directory to record the aforementioned user-created files: “TAC-output.txt” and “klee-stdout.log”.

Having the output files generated from KLEE ready, the next step is to parse these files to extract the required information about each feasible path reported by KLEE. This step is performed by the “Feasible Paths Data Extraction” component which consists of a series of shell script calls and a parser generated by JavaCC. Among the list of global files that KLEE generates for each execution, there is a set of per-path files to record information about the test case generated by KLEE for each path and the constraints associated with each path. The extensions for these files are .ktest and .pc, respectively. The test case files are binary files which require a
specific tool called ktest-tool to read their contents, however the path constraint files are textual files written in the KQuery language – the input language of the Kleaver constraint solver used by KLEE. In order to convert these constraints into Java Boolean expressions, we have developed a JavaCC parser for a relevant subset of the KQuery language (the token definition of the lexical analyzer of our JavaCC parser is shown in Listing 6.9). We use this parser to parse the path constraints files as well as the symbolic expressions found in the user-created file “klee-stdout.log”.

**Example 7.** In Listing 6.1 we show an excerpt of the file “TAC.c” that is generated by the component “KLEE Input File Preparation” for transition T2 and its symbolic source state SS4 in Figure 6.2(a). In Listings 6.2 and 6.3, we show the two path constraints files generated by KLEE as a result of executing the input file “TAC.c” shown in Listing 6.1. The Java Boolean expression representation corresponding to each path constraint is shown in Listings 6.4 and 6.5 respectively. Similarly, the files “TAC-output.txt” and “klee-stdout.log” resulting from executing KLEE on the input file “TAC.c” are shown in Listings 6.6 and 6.7. The Java expression representation corresponding to the symbolic variables found in the file “klee-stdout.log” in Listing 6.7 is shown in Listing 6.8. In Figure 6.2(c) we show the symbolic transitions ST7 and ST8 and their symbolic target states SS8 and SS9 resulting from the parsing and the extraction step implemented by the component “Feasible Paths Data Extraction” for the files in Listings 6.2, 6.3, 6.6 and 6.7.
6.2. INTERACTION WITH KLEE

(a) On the left is part of a Statechart showing a transition $T_2$ and its source and target States $S_2$ and $S_3$. On the right is a symbolic state $SS_4$ of the source state $S_2$.

(b) Illustration of our interface components with KLEE

(c) Highlighted (in red) are the generated symbolic transitions $ST_7$ and $ST_8$ and their target symbolic states $SS_8$ and $SS_9$ resulting from the SE of transition $T_2$ given a symbolic state $SS_4$ of its source state $S_2$.

Figure 6.2: An example illustrating an execution of the pseudo-code function $\text{SymbolicExecutionOfCode}(A^a, \text{val}, G^a, \text{pc})$
6.2. INTERACTION WITH KLEE

Listing 6.1 Excerpt from TAC.c

```c
#include <stdio.h>
#include <klee/klee.h>

int main()
{
    FILE *file;
    // variable declaration and initialization
    int T1_1_a;
    klee_make_symbolic(&T1_1_a, sizeof(T1_1_a), "T1_1_a");
    int T2_2_b;
    klee_make_symbolic(&T2_2_b, sizeof(T2_2_b), "T2_2_b");
    int x = T1_1_a;
    int y = 0;
    // user-defined variables to represent output events to be generated
    char *output1_eventName = "";
    char *output1_eventArg_1 = "";
    char *output1_eventArg_2 = "";
    // pre-defined path constraint (PC) declaration
    klee_assume(1==T1_1_a);
    // action code symbolic representation
    if(T2_2_b==x+1) {
        y=T2_2_b+1;
    } else {
        y=T2_2_b+2;
    }
    output1_eventName = "itsEnv.evReportVarsVal";
    output1_eventArg_1 = x;
    output1_eventArg_2 = y;
    // logging all variables’ valuation
    file = fopen("./klee-out-0/TAC-output.txt","a+");
    fprintf(file,"%s
","Output = ");
    printf("%s
","Output = ");
    if(klee_is_symbolic(T1_1_a)) {
        fprintf(file,"%s
","T1_1_a = SYMBOLIC");
        klee_print_expr("T1_1_a = ", T1_1_a);
    } else if(output1_eventName="") {
        fprintf(file,"%s\n","T1_1_a = ", T1_1_a);
    }
    ...}
    if(!(output1_eventName=="")) {
        fprintf(file,"%s\n","output1_eventName = ", output1_eventName);
        if(klee_is_symbolic(output1_eventArg_1)) {
            fprintf(file,"%s\n","output1_eventArg_1 = SYMBOLIC");
            klee_print_expr("output1_eventArg_1 = ", output1_eventArg_1);
        }
        else if(output1_eventArg_1=="")
            fprintf(file,"\n","output1_eventArg_1 = ");
        else
            fprintf(file,"%s\n","output1_eventArg_1 = ", output1_eventArg_1);
        if(klee_is_symbolic(output1_eventArg_2)) {
            fprintf(file,"%s\n","output1_eventArg_2 = SYMBOLIC");
            klee_print_expr("output1_eventArg_2 = ", output1_eventArg_2);
        }
        else if(output1_eventArg_2=="")
            fprintf(file,"\n","output1_eventArg_2 = ");
        else
            fprintf(file,"%s\n","output1_eventArg_2 = ", output1_eventArg_2);
    } else {
        fprintf(file,"%s\n","output1_eventArg_2 = ");
    }
    fclose(file);
}
```
Listing 6.2 KLEE generated KQuery constraint file associated with the first feasible path of the code in Listing 6.1:

test000001.pc

array T1_1_a[4] : w32 -> w8 = symbolic
array T2_2_b[4] : w32 -> w8 = symbolic
(query [(Eq 1
    (ReadLSB w32 0 T1_1_a))
    (Eq false
      (Eq 2
        (ReadLSB w32 0 T2_2_b)))]
false)

Listing 6.3 KLEE generated KQuery constraint file Associated with the second feasible path of the code in Listing 6.1:

test000002.pc

array T1_1_a[4] : w32 -> w8 = symbolic
array T2_2_b[4] : w32 -> w8 = symbolic
(query [(Eq 1
    (ReadLSB w32 0 T1_1_a))
    (Eq false
      (Eq 2
        (ReadLSB w32 0 T2_2_b)))]
false)

Listing 6.4 The Boolean Expressions of the KQuery Path Constraint in Listing 6.2

(1==T1_1_a) && !(2==T2_2_b)

Listing 6.5 The Boolean Expressions of the KQuery Path Constraint in Listing 6.3

(1==T1_1_a) && (2==T2_2_b)
6.2. INTERACTION WITH KLEE

Listing 6.6 User-created File from running KLEE on the LLVM bitcode representation of file TAC.c: TAC-output.txt

Output =
T1_1_a = SYMBOLIC
T2_2_b = SYMBOLIC
x = SYMBOLIC
y = SYMBOLIC
output1_eventName = itsEnv.evReportVarsVal
output1_eventArg_1 = SYMBOLIC
output1_eventArg_2 = SYMBOLIC
Output =
T1_1_a = SYMBOLIC
T2_2_b = SYMBOLIC
x = SYMBOLIC
y = SYMBOLIC
output1_eventName = itsEnv.evReportVarsVal
output1_eventArg_1 = SYMBOLIC
output1_eventArg_2 = SYMBOLIC

Listing 6.7 User-created file of the resulting stdout from running KLEE on the LLVM bitcode representation of file TAC.c: klee-stdout.log

KLEE: output directory = "klee_out-0"

Output =
T1_1_a = (ReadLSB w32 0 T1_1_a)
T2_2_b = (ReadLSB w32 0 T2_2_b)
x = (Add w32 2
  (ReadLSB w32 0 T2_2_b))
y = (Add w32 2
  (ReadLSB w32 0 T2_2_b))
output1_eventArg_1 = (Add w32 2
  (ReadLSB w32 0 T1_1_a))
output1_eventArg_2 = (Add w32 2
  (ReadLSB w32 0 T2_2_b))
Output =
T1_1_a = (ReadLSB w32 0 T1_1_a)
T2_2_b = (ReadLSB w32 0 T2_2_b)
x = (Add w32 1
  (ReadLSB w32 0 T1_1_a))
y = (Add w32 1
  (ReadLSB w32 0 T2_2_b))
output1_eventArg_1 = (Add w32 1
  (ReadLSB w32 0 T1_1_a))
output1_eventArg_2 = (Add w32 1
  (ReadLSB w32 0 T2_2_b))

KLEE: done: total instructions = 194
KLEE: done: completed paths = 2
KLEE: done: generated tests = 2

Listing 6.8 The Boolean expressions of the KQuery path constraint in Listing 6.7

T1_1_a = T1_1_a
T2_2_b = T2_2_b
x = T1_1_a
y = 2 + T2_2_b
output1_eventArg_1 = T1_1_a
output1_eventArg_2 = 2 + T2_2_b

T1_1_a = T1_1_a
T2_2_b = T2_2_b
x = T1_1_a
y = 1 + T2_2_b
output1_eventArg_1 = T1_1_a
output1_eventArg_2 = 1 + T2_2_b
6.3 Implementation of the MQL & TGL-based Transformation Components

Listing 6.9 The token definition of the lexical analyzer of our JavaCC parser for a subset of the KQuery language

```
TOKEN:
{
    < LBR : "(" >
    | < RBR : ")" >
    | < QUERY : "query" >
    | < LSBR : "[" >
    | < RSBR : "]" >
    | < FALSE : "false" >
    | < TRUE : "true" >
    | < VarDeclStart : ("ReadLSB") >
    | < AEK : ("Add" | "Sub" | "Mul" | "UDiv" | "URem" | "SDiv" | "SRem") >
    | < CEK : ("Eq" | "Ne" | "Ult" | "Ule" | "Ugt" | "Uge" | "Slt" | "Sle" | "Sgt" |
               "Sge") >
    | < type : 
               (["w"])(["0"-"9"])+ >
    | < identifier : ("[a"-"z", "A"-"Z", "]")>("[a"-"z", "A"-"Z", "0"-"9", ", "]")*>>
    | < colon : ":" >
    | < number : (<decConstant>| <signedDecConstant>) >
    | < signedDecConstant : ("+")|("-")<decConstant>>
    | < decConstant : (<digit>)+ >
    | < digit : ["0"-"9"] >
}
```

6.3 Implementation of the MQL & TGL-based Transformation Components

Our transformation components are implemented in the context of the IBM Rational Rhapsody Developer RulesComposer Add-On which is an Eclipse-based development environment for model transformation and text generation. In such an environment, a model transformation is defined using an imperative transformation language called Model Query Language (MQL). The basic element of a model transformation is a ruleset which is a group of logically interdependent rules, each of which specifies a set of procedural expressions that query source and target model elements to create new elements, update the attributes and the references of existing elements, or launch text generation templates. A model transformation may be implemented using one or more rulesets. Text generation is defined using templates written in a language
called the Text Generation Language (TGL) which uses MQL expressions (i.e., placeholders) for dynamic text generation. Scripts are methods written in MQL, TGL or Java to extend and customize a meta-type of a source or a target meta-model for a particular model transformation or text generation. They are called from MQL and TGL code on model elements in the same way as regular Java methods.

To implement our transformation components, we first defined the meta-model of the CMLMs representation defined in Chapter 4 as an Ecore model which we show in Appendix A. We also imported this Ecore model as a meta-model project that is added to the default set of meta-models defined within the RulesComposer Add-On. Then, we created a RulesComposer project with references to the Rhapsody meta-model which is already defined and integrated within the tool and the CMLMs meta-model that we have defined and imported into the tool. Our implementation of this model transformation and text generation RulesComposer project consists of:

- 19 rules grouped into 2 rulesets. The first ruleset groups a set of 17 rules that specify how to flatten and map a Rhapsody source model into a CMLMs target model. The second ruleset groups a set of 2 rules that computes the data and the communication dependencies in a given CMLMs model.

- 48 MQL scripts distributed among the following meta-types:
  - 9 scripts for “rhapsody_Statechart”,
  - 21 scripts for “rhapsody_State”;
  - one script for “rhapsody_Transition”,
  - 5 scripts for “cmlms_CMLM”,

– 5 scripts for “cmlms_MLM”, and
– 7 scripts for “cmlms_Transition”.
– 6 Java scripts equally distributed between the “cmlms_CMLM” and “cmlms_MLM” meta-types. We defined these scripts to be able to create specialized ArrayList, HashMap and HashSet objects to be used within our rules.
– 2 TGL templates. These templates are used to generate a Java representation for our CMLMs representations; one for the individual MLMs and one for the communicating MLMs.
– 4948 lines of code (LOC) distributed as follows:
  – 2761 LOC for the rulesets,
  – 1516 LOC for the MQL scripts,
  – 102 LOC for the Java scripts, and
  – 568 LOC for the TGL templates.

6.4 Interaction with the IBM Rational Rhapsody DiffMerge and its Related Components

As we decided to use the IBM Rational Rhapsody DiffMerge tool to find the differences between evolving versions of Rhapsody models, we needed to find a way to process and parse the output generated by the tool. Two types of reports can be exported from the tool that summarize the differences found between compared models. The first report is a Rich Text Format (RTF) report and it is designed to contain a greater level of detail but it is not suitable for further processing to extract difference
6.4. INTERACTION WITH THE IBM RATIONAL RHAPSODY
DIFFMERGE AND ITS RELATED COMPONENTS

data. On the other hand, the second report is a Comma Separated Values (CSV)
format report which is easier to process but it is designed to contain a lower level of
detail. In order to extend the information included in the CSV format reports about
the difference data, we had to extend the underlying CSV report template of the tool.
The default template file we found is called “DiffReport_csv.dpl” and it is written
in a language that is embedded in the IBM Rational Rhapsody ReporterPLUS for
generating CSV format reports about the difference data found by the Rhapsody
DiffMerge tool.

The original version of this template file contains 156 LOC and it reports only
on six specified attributes for a given difference. This includes information about
the meta-class of the difference elements, their names, their locations, the difference
type indicating whether the difference element is an added element, a deleted ele-
ment or a modified element, the differences count (i.e., the number of attributes in
the difference elements with different values) and whether a difference is a trivial (i.e.,
non-conflicting) difference or not. This latter feature is mainly used when the three-
way comparison with the base unit option is performed. In this case if a difference
is found in the two compared units with respect to their base unit, then this differ-
ence is a non-trivial (i.e., conflicting) difference which needs manual resolution; the
automatic merging operation of the tool cannot resolve non-trivial differences. The
merging operation is outside the scope of our work and we do not need the three-
way comparison option. Therefore, all the differences reported were trivial differences.

Our adapted version of the same template file contains 205 LOC and it adds
three extra attributes to be reported to the original list. These are the names of the attributes in the difference elements with different values and their values in both compared models, namely the “left value” and the “right value”. Recording the values of these attributes enabled us to uniquely identify specific Rhapsody model elements which may have the same value assigned to their “name” attribute. Transitions are an example of such elements where two or more transitions can have the same name. Users are not allowed to set the value of the “name” attribute of a transition in the IBM Rhapsody tool. Instead, the tool uses the values assigned to the “trigger”, the “guard” and the “action” attributes of a transition to generate and assign a compound value to its “name” attribute. For the unique identification of transitions in a model, the tool uses an internal global unique identifier number to be assigned to each newly created transition. The attribute that is used for this purpose is a private variable which cannot be accessed by users. We also noticed that the value of the “name” attribute of a difference transition recorded by the Rhapsody DiffMerge tool does not exactly match the value recorded for the attribute “name” of the same transition in the corresponding Rhapsody Statechart model. Only the “trigger” and the “guard” parts of the name of a transition are used by the Rhapsody DiffMerge tool to report the name of a difference transition. This information alone is not enough to uniquely identify a transition in a model especially in the case of poorly designed Statecharts that may have non-deterministic transitions. Therefore, we believe that the addition we have made to the default CSV report template of the Rhapsody DiffMerge tool was mandatory and it enabled us to uniquely identify the difference elements reported including states and transitions.

Given our adapted CSV format difference reports, the next step for us was to develop
6.5. SUMMARY

In this chapter we have provided an overview of the implementation of the architectures presented in the previous chapters, showing the operating systems required and the technologies used and how we realized the interaction between the different components to fulfill the required tasks.

a parser such that all the difference information recorded in a given CSV difference report are mapped to our user-defined data structure objects. We use this parsed information and the MLM or CMLMs representations of compared Rhapsody models as the inputs to components “SC2MLM DiffMap” and “RhapM2CMLMs” in order to map the identified difference elements in the compared Rhapsody models into their MLM or CMLMs correspondences. The generated difference report from either of the two aforementioned components is used as an input to component “Dependency-based Change Impact Analysis” to identify the parts of an input MLM or CMLMs representation that have a data or communication dependence with a difference element in the input difference report.

The implementation of the CSV parser, the two DiffMap components and the “Dependency-based Change Impact Analysis” component are all built as a Java project in the context of the IBM Rational Rhapsody RulesComposer Add-On. The project consists of 10 Java classes and approximately 1581 lines of code (LOC).
Chapter 7

Evaluation

In this chapter we present the evaluation of the effectiveness of the two proposed optimization techniques: MSE and DSE, on both individual Statecharts and communicating Statecharts. We first state the research questions considered in our evaluation. Second, we list the set of artifacts used as case studies for individual and communicating Statecharts. Third, we describe the evaluation setup. Finally we discuss the results and draw conclusions.

It is worth mentioning that we performed our evaluation in two stages. The first stage considered the evaluation of individual Statecharts, while the second stage considered the evaluation of communicating Statecharts. In the first stage we consider only modification changes, while in the second stage we consider also addition and deletion changes.

7.1 Research Questions and Variables of Interest

We consider the following three research questions:

- **RQ1.** How effective are our optimizations (i.e., how much do they reduce the
resource requirements of the symbolic execution of a changed state machine model) compared to standard SE of a changed Statechart model?

For RQ1, we consider the following three correlated variables: 1) the time taken to run each technique, 2) the number of symbolic states and 3) the number of execution paths in the resulting SETs.

– RQ2. Does the SET generated from MSE match the one generated from standard SE?

For RQ2, we compare the total number of symbolic states in the SETs resulting from standard SE and MSE. We also perform manual inspection of a subset of the two SETs to ensure their equivalence.

– RQ3. Which aspects influence the effectiveness of each technique?

For RQ3, we consider the following two aspects: 1) change impact and 2) model characteristics.

To measure the impact of the change, we consider the following two criteria:

1. we define a change impact metric (CIM) as the percentage of the maximal paths of the MLM model involving the change. The lower the value of this metric is, the lower the impact of the change on the model and the higher the opportunity to benefit from a previous analysis results, and vice versa. This metric is used only for individual Statecharts (i.e., it is applied only on the AQS, LGS and ACCS models) where the notion of maximal path is defined and, thus, can be computed.

2. we identify the type of the change applied to create each modified version
of the subject models. The change types we consider are modification (M), addition (A) and deletion (D).

For model characteristics, we identify the following two metrics that appear likely to have an impact on the effectiveness of DSE:

1. the number of transitions in the subject model with multi-path guards or action code.
2. the number of transitions that have a dependency with other transitions in the model.

We speculate that the change impact is a better predictor for the effectiveness of MSE, whereas the model characteristics are more suitable as indicators of the effectiveness of DSE.

7.2 Case Study Artifacts

To evaluate the effectiveness of our optimization techniques on both individual and communicating Statecharts, the following artifacts are selected:

1. For individual Statecharts: We chose three industrial-sized models from the automotive domain. The first model, the Air Quality System (AQS), is a proprietary model that we obtained from our industrial partner that is responsible for air purification in the vehicle’s cabin. The second and the third models, the Lane Guide System (LGS) and the Adaptive Cruise Control System (ACCS), are non-proprietary models designed at the University of Waterloo [30]. The LGS is an automotive feature used to avoid unintentional lane departure by providing alerts when certain events occur. The ACCS is an automotive feature
used to automatically maintain the speed of a vehicle set by the driver through the automatic operation of the vehicle. The three models were developed as Simulink/Stateflow models and we manually converted them to behaviorally equivalent Rhapsody Statecharts. Table 7.1a summarizes the characteristics of the three models and their MLM representations, including the two model characteristics metrics used to answer RQ3. The models contain 4, 7, and 3 variables of type integer, respectively.

2. *For communicating Statecharts:* We used an adapted version of a system model implementing a well-known communication protocol called Full-duplex Alternating Protocol (FABP) found in [65]. The model consists of three communicating objects: Transmitter, Receiver and Buffer. The behavior of each object is modeled as a Statechart. A summary of the number of states and transitions in each state machine is shown in Table 7.1b. We also listed the values of the model characteristics metrics, namely the number of transitions with multi-path action in each Statechart and the number of transition with data or communication dependencies. There are 4 integer variables in this model.

### 7.3 Evaluation Setup

To perform our study, we followed the following procedures:

1. *For individual Statecharts:* We first prepared a set of different versions for each artifact: the base version and a number of modified versions. Each modified version introduces a single change to one simple or group transition in the base version. Each change is made in the form of an alteration to the event name or
7.3. EVALUATION SETUP

Table 7.1: Characteristics of the artifacts used in our evaluation

(a) Three individual Statecharts models

<table>
<thead>
<tr>
<th></th>
<th>Air Quality System (AQS)</th>
<th>Lane Guide System (LGS)</th>
<th>Adaptive Cruise Control System (ACCS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrent regions</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Hierarchical levels</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>States</td>
<td>3 CS + 14 SS</td>
<td>14 SS</td>
<td>2 CS + 6 SS</td>
</tr>
<tr>
<td></td>
<td>10 SS</td>
<td>3 SS</td>
<td>19 SS</td>
</tr>
<tr>
<td>Transitions</td>
<td>8 GT + 22 ST</td>
<td>55 ST</td>
<td>9 GT + 5 ST</td>
</tr>
<tr>
<td></td>
<td>6 GT + 16 ST</td>
<td>40 ST</td>
<td>9 ST</td>
</tr>
<tr>
<td></td>
<td>40 ST</td>
<td>3 SS</td>
<td>73 ST</td>
</tr>
<tr>
<td>Transitions with</td>
<td>5 ST</td>
<td>5 ST</td>
<td>2 ST</td>
</tr>
<tr>
<td>multi-path guards</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or action code</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transitions with</td>
<td>3 ST</td>
<td>3 ST</td>
<td>2 ST</td>
</tr>
<tr>
<td>data dependencies</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| SS=Simple State, CS=Composite State, ST=Simple Transition

(b) One communicating Statecharts model

<table>
<thead>
<tr>
<th></th>
<th>Full-duplex Alternating Bit Protocol (FABP)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Transmitter (Trans)</td>
</tr>
<tr>
<td>Concurrent regions</td>
<td>1</td>
</tr>
<tr>
<td>Hierarchical levels</td>
<td>1</td>
</tr>
<tr>
<td>States</td>
<td>4 SS</td>
</tr>
<tr>
<td></td>
<td>2 SS</td>
</tr>
<tr>
<td>Transitions</td>
<td>14 ST</td>
</tr>
<tr>
<td></td>
<td>14 ST</td>
</tr>
<tr>
<td>Transitions with</td>
<td>0</td>
</tr>
<tr>
<td>multi-path guards</td>
<td></td>
</tr>
<tr>
<td>or action code</td>
<td></td>
</tr>
<tr>
<td>Transitions with</td>
<td>8 ST</td>
</tr>
<tr>
<td>data dependencies</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Transitions with</td>
<td>10 ST</td>
</tr>
<tr>
<td>communication</td>
<td></td>
</tr>
<tr>
<td>dependencies</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 ST</td>
</tr>
</tbody>
</table>
| SS=Simple State, ST=Simple Transition

Table 7.2: Performance of standard SE on the base version of the four models: AQS, LGS, ACCS and FABP

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (min)</td>
<td>≈ 10</td>
<td>≈ 5</td>
<td>≈ 9</td>
<td>≈ 14</td>
</tr>
<tr>
<td>No. Symbolic States</td>
<td>6039</td>
<td>1769</td>
<td>3492</td>
<td>2805</td>
</tr>
<tr>
<td>No. Execution Paths</td>
<td>5019</td>
<td>1325</td>
<td>2855</td>
<td>1342</td>
</tr>
</tbody>
</table>
the addition of a send event statement to the action code of the subject transition. The main reason for selecting these specific types of changes is to keep the number of modified models manageable and to facilitate the manual correctness check of the results. Another important reason is that some of the tools used in the implementation (e.g., the IBM Rhapsody DiffMerge) do not have command line access which made it impossible to fully automate the collection of the performance data and evaluate the techniques more comprehensively using, e.g., a large number of automatically modified models. The total numbers of the modified versions created for each model are: 26 for the AQS model, 19 of the LGS and 19 for the ACCS model.

For each modified version, we first recorded the following information: 1) the change ID\(^1\) and the number of transitions in the MLM representation of the subject model corresponding to the changed transition in Rhapsody, 2) the number of multi-path transitions in the MLM representation of the subject model that have been found to impact or be impacted by the changed transition and 3) the value of the CIM metric computed for the study of RQ3. Second, we ran our standard SE on all the versions of the three selected artifacts, while we ran both the MSE and the DSE only on the modified versions, given the results of the SE of their base versions in the case of MSE.

For each run, we recorded the time to generate the SET and the total number of symbolic states in the generated SET. Additionally, in case of an MSE run, we recorded the number of symbolic states that are newly created. This is to

---

\(^1\)A compound value of the model name and the ID of the transition in Rhapsody where the change is made prefixed with the change type (M for a modification change, A for an addition change or D for a deletion change).
differentiate between the symbolic states generated by the MSE technique and those that have been reused from the SET of the base version. To measure the effectiveness of MSE and DSE compared to standard SE, we computed the ratios between the execution times, the number of symbolic states and the number of the execution paths recorded for the standard SE and their correspondences in the MSE and DSE, respectively. From these ratios, we computed the average savings and the standard deviation. We also computed the correlation coefficients\(^2\) between the savings gained from each technique and the computed CIM metric values.

2. For communicating Statecharts: We followed the same procedure described above for individual Statecharts, however here we prepared three sets of different versions of the base version of the FABP model. Each set has 32 modified versions with either an added transition, a deleted transition or a modified transition. All changes in the modified versions are atomic. However, they cover all the transitions in the FABP model. For each model version, we recorded the same information listed earlier except the information about the CIM metric which is not applicable for communicating Statecharts. We ran our standard SE and the MSE on all versions, while we ran the DSE only on the versions where the change is found to not impact all of the multi-path transitions in the model. Performing the DSE on a model version with a change that has an impact on all multi-path transitions will result in no savings compared to performing the standard SE on the same model. For each run, we recorded\(^2\)

\(^2\)We use the sample Pearson correlation coefficient formula implemented in the Microsoft Excel function \texttt{CORREL} to measure the degree of linear correlation between two variables with a range between +1 and -1 inclusive. A value of +1 indicates a perfect positive correlation, while a value of -1 indicates a perfect negative correlation. A value of 0 indicates no correlation.
the same information listed earlier about the characteristics of the generated GSETs.

The characteristics of the SETs generated from running the standard SE on the base versions of the AQS, LGS, ACCS and FABP models are shown in Table 7.2. They include the time taken to run the standard SE technique and the number of symbolic states and execution paths in the resulting SETs.

Tables 7.3, 7.4 and 7.5 summarize the results of running the three symbolic execution techniques on the modified versions of the AQS, the LGS and the ACCS models, respectively. Additionally, a summary of the results for the versions of each change type of the FABP model is depicted in Tables 7.6, 7.7 and 7.8 respectively.

In columns 5-6, 8 (resp., 10-12) of Tables 7.3–7.5 and columns 4-5, 7 (resp., 9-11) of Tables 7.6–7.8, we show the savings ratios in time, in the number of symbolic states and in the number of execution paths gained from applying MSE (resp., DSE) on the modified versions of the four given models compared to standard SE. We also show, for each such column, the overall average and the standard deviation. Additionally, we show the correlation coefficient with the CIM metric data in column 4 of Tables 7.3–7.5 for the individual Statecharts models.

The ratios between the total number of symbolic states (resp., execution paths) found in the resulting SETs generated from standard SE and MSE for the four given models are shown in column 7 (resp., 9) of Tables 7.3–7.5 and column 6 (resp., 8) of Tables 7.6–7.8. The values recorded in these four columns are computed with “Task 4” of the MSE Algorithms 7 and 8 being enabled and they are used for answering RQ2 with respect to MSE.
7.3. EVALUATION SETUP

Table 7.3: Results of MSE and DSE on the AQS example

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MAQS-06</td>
<td>/7</td>
<td>0</td>
<td>30</td>
<td>1</td>
<td>0.3</td>
<td>1</td>
<td>0.08</td>
<td>1</td>
<td>1</td>
<td>0.08</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-27</td>
<td>/1</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>0.3</td>
<td>1</td>
<td>0.36</td>
<td>1</td>
<td>1</td>
<td>0.36</td>
<td>1</td>
<td>1</td>
<td>0.01</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-07</td>
<td>/1</td>
<td>0</td>
<td>100</td>
<td>1</td>
<td>0.51</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-00</td>
<td>/1</td>
<td>0</td>
<td>20</td>
<td>1</td>
<td>0.26</td>
<td>1</td>
<td>0.23</td>
<td>1</td>
<td>1</td>
<td>0.23</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
<td>0.04</td>
<td>1</td>
<td>0.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-10</td>
<td>/1</td>
<td>1</td>
<td>30</td>
<td>1</td>
<td>0.21</td>
<td>1</td>
<td>0.01</td>
<td>1</td>
<td>1</td>
<td>0.01</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
<td>0.14</td>
<td>1</td>
<td>0.13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-08</td>
<td>/1</td>
<td>1</td>
<td>90</td>
<td>1</td>
<td>0.44</td>
<td>1</td>
<td>0.64</td>
<td>1</td>
<td>1</td>
<td>0.66</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
<td>0.14</td>
<td>1</td>
<td>0.13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-28</td>
<td>/1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.23</td>
<td>1</td>
<td>0.06</td>
<td>1</td>
<td>1</td>
<td>0.06</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
<td>0.14</td>
<td>1</td>
<td>0.13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-29</td>
<td>/1</td>
<td>0</td>
<td>30</td>
<td>1</td>
<td>0.27</td>
<td>1</td>
<td>0.23</td>
<td>1</td>
<td>1</td>
<td>0.23</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-11</td>
<td>/1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.19</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-12</td>
<td>/1</td>
<td>0</td>
<td>40</td>
<td>1</td>
<td>0.24</td>
<td>1</td>
<td>0.23</td>
<td>1</td>
<td>1</td>
<td>0.23</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
<td>0.04</td>
<td>1</td>
<td>0.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-13</td>
<td>/1</td>
<td>0</td>
<td>30</td>
<td>1</td>
<td>0.2</td>
<td>1</td>
<td>0.06</td>
<td>1</td>
<td>1</td>
<td>0.06</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-14</td>
<td>/1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0.22</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-04</td>
<td>/5</td>
<td>0</td>
<td>97</td>
<td>1</td>
<td>0.64</td>
<td>1</td>
<td>0.87</td>
<td>1</td>
<td>1</td>
<td>0.91</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-05</td>
<td>/7</td>
<td>0</td>
<td>30</td>
<td>1</td>
<td>0.38</td>
<td>1</td>
<td>0.39</td>
<td>1</td>
<td>1</td>
<td>0.43</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-15</td>
<td>/1</td>
<td>0</td>
<td>94</td>
<td>1</td>
<td>0.41</td>
<td>1</td>
<td>0.61</td>
<td>1</td>
<td>1</td>
<td>0.64</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
<td>0.17</td>
<td>1</td>
<td>0.16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-16</td>
<td>/1</td>
<td>0</td>
<td>14</td>
<td>1</td>
<td>0.22</td>
<td>1</td>
<td>0.01</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-17</td>
<td>/5</td>
<td>0</td>
<td>14</td>
<td>1</td>
<td>0.29</td>
<td>1</td>
<td>0.15</td>
<td>1</td>
<td>1</td>
<td>0.17</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-18</td>
<td>/5</td>
<td>0</td>
<td>42</td>
<td>1</td>
<td>0.22</td>
<td>1</td>
<td>0.06</td>
<td>1</td>
<td>1</td>
<td>0.06</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-20</td>
<td>/1</td>
<td>0</td>
<td>35</td>
<td>1</td>
<td>0.21</td>
<td>1</td>
<td>0.01</td>
<td>1</td>
<td>1</td>
<td>0.01</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-21</td>
<td>/1</td>
<td>0</td>
<td>48</td>
<td>1</td>
<td>0.32</td>
<td>1</td>
<td>0.41</td>
<td>1</td>
<td>1</td>
<td>0.42</td>
<td>1</td>
<td>1</td>
<td>0.01</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-22</td>
<td>/1</td>
<td>0</td>
<td>27</td>
<td>1</td>
<td>0.22</td>
<td>1</td>
<td>0.01</td>
<td>1</td>
<td>1</td>
<td>0.01</td>
<td>1</td>
<td>1</td>
<td>0.01</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-23</td>
<td>/1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0.21</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0.01</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-24</td>
<td>/1</td>
<td>0</td>
<td>35</td>
<td>1</td>
<td>0.29</td>
<td>1</td>
<td>0.41</td>
<td>1</td>
<td>1</td>
<td>0.41</td>
<td>1</td>
<td>1</td>
<td>0.01</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-25</td>
<td>/1</td>
<td>0</td>
<td>14</td>
<td>1</td>
<td>0.2</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>1</td>
<td>0.01</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-26</td>
<td>/1</td>
<td>0</td>
<td>25</td>
<td>1</td>
<td>0.3</td>
<td>1</td>
<td>0.4</td>
<td>1</td>
<td>1</td>
<td>0.4</td>
<td>1</td>
<td>1</td>
<td>0.01</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAQS-03</td>
<td>/5</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0.28</td>
<td>1</td>
<td>0.09</td>
<td>1</td>
<td>1</td>
<td>0.08</td>
<td>1</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
<td>0.03</td>
<td>1</td>
<td>0.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|      | Average Savings | 71.0 % | 75.5 % | 74.9 % | 98.4 % | 96.5 % | 96.1 % |
|      | Standard Deviation | 10.9 % | 28.0 % | 28.8 % | 0.2 % | 4.4 % | 4.2 % |
|      | CORREL(CIM) | -0.81 | -0.83 | -0.84 | -0.21 | -0.33 | -0.33 |

Our evaluation is performed on a standard PC with Intel Core i7 CPU 3.4 GHz and 8 GB of RAM and running Windows 7 as a host and Ubuntu 12.04 as VM.
### 7.4. RESULTS AND ANALYSIS

In this section, we discuss the results presented in Tables 7.3–7.8 according to our three research questions.

**RQ1. How effective are our optimizations compared to standard SE?**

In Table 7.9, we show a summary of the average and the standard deviation values of the data shown in columns 5-6, 8 (resp., 10-12) of Tables 7.3–7.5 and columns 4-5, 7

<table>
<thead>
<tr>
<th>Ver.</th>
<th>MLM Change ID / No. of Changed Transitions</th>
<th>Change ID / No. of Multi-path Impacted Transitions</th>
<th>CIM [0, 100]</th>
<th>SE : MSE</th>
<th>SE : DSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>M[LGS-01] / 1</td>
<td>0</td>
<td>100</td>
<td>1 : 0.81</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V2</td>
<td>M[LGS-02] / 7</td>
<td>0</td>
<td>28</td>
<td>1 : 0.19</td>
<td>1 : 0.25</td>
</tr>
<tr>
<td>V3</td>
<td>M[LGS-05] / 7</td>
<td>0</td>
<td>28</td>
<td>1 : 0.19</td>
<td>1 : 0.25</td>
</tr>
<tr>
<td>V4</td>
<td>M[LGS-09] / 1</td>
<td>0</td>
<td>97</td>
<td>1 : 0.74</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V5</td>
<td>M[LGS-08] / 4</td>
<td>0</td>
<td>13</td>
<td>1 : 0.38</td>
<td>1 : 0.5</td>
</tr>
<tr>
<td>V6</td>
<td>M[LGS-06] / 7</td>
<td>0</td>
<td>13</td>
<td>1 : 0.47</td>
<td>1 : 0.58</td>
</tr>
<tr>
<td>V7</td>
<td>M[LGS-07] / 5</td>
<td>0</td>
<td>40</td>
<td>1 : 0.55</td>
<td>1 : 0.74</td>
</tr>
<tr>
<td>V8</td>
<td>M[LGS-11] / 1</td>
<td>0</td>
<td>22</td>
<td>1 : 0.24</td>
<td>1 : 0.27</td>
</tr>
<tr>
<td>V9</td>
<td>M[LGS-13] / 1</td>
<td>0</td>
<td>22</td>
<td>1 : 0.25</td>
<td>1 : 0.27</td>
</tr>
<tr>
<td>V10</td>
<td>M[LGS-19] / 1</td>
<td>0</td>
<td>22</td>
<td>1 : 0.3</td>
<td>1 : 0.27</td>
</tr>
<tr>
<td>V11</td>
<td>M[LGS-20] / 1</td>
<td>0</td>
<td>22</td>
<td>1 : 0.28</td>
<td>1 : 0.27</td>
</tr>
<tr>
<td>V12</td>
<td>M[LGS-12] / 1</td>
<td>0</td>
<td>3</td>
<td>1 : 0.09</td>
<td>1 : 0.04</td>
</tr>
<tr>
<td>V13</td>
<td>M[LGS-15] / 1</td>
<td>0</td>
<td>13</td>
<td>1 : 0.14</td>
<td>1 : 0.12</td>
</tr>
<tr>
<td>V14</td>
<td>M[LGS-10] / 1</td>
<td>0</td>
<td>3</td>
<td>1 : 0.09</td>
<td>1 : 0.04</td>
</tr>
<tr>
<td>V15</td>
<td>M[LGS-14] / 1</td>
<td>0</td>
<td>13</td>
<td>1 : 0.15</td>
<td>1 : 0.12</td>
</tr>
<tr>
<td>V16</td>
<td>M[LGS-16] / 1</td>
<td>0</td>
<td>13</td>
<td>1 : 0.18</td>
<td>1 : 0.12</td>
</tr>
<tr>
<td>V17</td>
<td>M[LGS-21] / 1</td>
<td>0</td>
<td>3</td>
<td>1 : 0.09</td>
<td>1 : 0.04</td>
</tr>
<tr>
<td>V18</td>
<td>M[LGS-17] / 1</td>
<td>0</td>
<td>13</td>
<td>1 : 0.16</td>
<td>1 : 0.12</td>
</tr>
<tr>
<td>V19</td>
<td>M[LGS-18] / 1</td>
<td>0</td>
<td>3</td>
<td>1 : 0.09</td>
<td>1 : 0.04</td>
</tr>
</tbody>
</table>

**Average Savings:** 71.7 % 68.2 % 67.2 % 76.7 % 67.0 % 66.6 %

**Standard Deviation:** 21.6 % 30.6 % 30.5 % 2.2 % 10.5 % 10.4 %

**CORREL(M):** -0.89 -0.88 -0.89 -0.48 -0.09 -0.08
Table 7.5: Results of MSE and DSE on the ACCS example

<table>
<thead>
<tr>
<th>Ver.</th>
<th>M[ACCS-04]</th>
<th>1</th>
<th>0</th>
<th>100</th>
<th>1 : 0.82</th>
<th>1 : 1</th>
<th>1 : 1</th>
<th>1 : 1</th>
<th>1 : 1</th>
<th>1 : 1</th>
<th>1 : 1</th>
<th>1 : 1.05</th>
<th>1 : 0.92</th>
<th>1 : 0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>V2</td>
<td>M[ACCS-05]</td>
<td>12</td>
<td>30</td>
<td></td>
<td>1 : 0.21</td>
<td>1 : 0.17</td>
<td>1 : 1</td>
<td>1 : 0.17</td>
<td>1 : 1</td>
<td>1 : 0.78</td>
<td>1 : 0.92</td>
<td>1 : 0.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V3</td>
<td>M[ACCS-06]</td>
<td>12</td>
<td>30</td>
<td></td>
<td>1 : 0.19</td>
<td>1 : 0.15</td>
<td>1 : 1</td>
<td>1 : 0.17</td>
<td>1 : 1</td>
<td>1 : 0.78</td>
<td>1 : 0.92</td>
<td>1 : 0.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V4</td>
<td>M[ACCS-19]</td>
<td>2</td>
<td>2</td>
<td></td>
<td>1 : 0.61</td>
<td>1 : 0.79</td>
<td>1 : 1</td>
<td>1 : 0.8</td>
<td>1 : 1</td>
<td>1 : 1.02</td>
<td>1 : 0.92</td>
<td>1 : 0.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V5</td>
<td>M[ACCS-20]</td>
<td>2</td>
<td>2</td>
<td></td>
<td>1 : 0.43</td>
<td>1 : 0.44</td>
<td>1 : 1</td>
<td>1 : 0.46</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 0.92</td>
<td>1 : 0.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V6</td>
<td>M[ACCS-13]</td>
<td>2</td>
<td>60</td>
<td></td>
<td>1 : 0.73</td>
<td>1 : 0.87</td>
<td>1 : 1</td>
<td>1 : 0.87</td>
<td>1 : 1</td>
<td>1 : 1.02</td>
<td>1 : 0.92</td>
<td>1 : 0.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V7</td>
<td>M[ACCS-14]</td>
<td>2</td>
<td>60</td>
<td></td>
<td>1 : 0.45</td>
<td>1 : 0.56</td>
<td>1 : 1</td>
<td>1 : 0.56</td>
<td>1 : 1</td>
<td>1 : 1.02</td>
<td>1 : 0.92</td>
<td>1 : 0.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V8</td>
<td>M[ACCS-15]</td>
<td>2</td>
<td>20</td>
<td></td>
<td>1 : 0.56</td>
<td>1 : 0.76</td>
<td>1 : 1</td>
<td>1 : 0.77</td>
<td>1 : 1</td>
<td>1 : 1.03</td>
<td>1 : 0.92</td>
<td>1 : 0.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V9</td>
<td>M[ACCS-16]</td>
<td>2</td>
<td>20</td>
<td></td>
<td>1 : 0.24</td>
<td>1 : 0.18</td>
<td>1 : 1</td>
<td>1 : 0.2</td>
<td>1 : 1</td>
<td>1 : 1.03</td>
<td>1 : 0.92</td>
<td>1 : 0.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V10</td>
<td>M[ACCS-17]</td>
<td>4</td>
<td>36</td>
<td></td>
<td>1 : 0.14</td>
<td>1 : 0.05</td>
<td>1 : 1</td>
<td>1 : 0.04</td>
<td>1 : 1</td>
<td>1 : 1.04</td>
<td>1 : 0.92</td>
<td>1 : 0.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V11</td>
<td>M[ACCS-18]</td>
<td>4</td>
<td>36</td>
<td></td>
<td>1 : 0.26</td>
<td>1 : 0.2</td>
<td>1 : 1</td>
<td>1 : 0.18</td>
<td>1 : 1</td>
<td>1 : 1.04</td>
<td>1 : 0.92</td>
<td>1 : 0.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V12</td>
<td>M[ACCS-07]</td>
<td>1</td>
<td>99</td>
<td></td>
<td>1 : 0.81</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1.02</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V13</td>
<td>M[ACCS-08]</td>
<td>6</td>
<td>14</td>
<td></td>
<td>1 : 0.27</td>
<td>1 : 0.22</td>
<td>1 : 1</td>
<td>1 : 0.19</td>
<td>1 : 1</td>
<td>1 : 1.02</td>
<td>1 : 0.92</td>
<td>1 : 0.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V14</td>
<td>M[ACCS-09]</td>
<td>1</td>
<td>11</td>
<td></td>
<td>1 : 0.59</td>
<td>1 : 0.67</td>
<td>1 : 1</td>
<td>1 : 0.68</td>
<td>1 : 1</td>
<td>1 : 0.87</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V15</td>
<td>M[ACCS-21]</td>
<td>1</td>
<td>11</td>
<td></td>
<td>1 : 0.11</td>
<td>1 : 0.01</td>
<td>1 : 1</td>
<td>1 : 0.01</td>
<td>1 : 1</td>
<td>1 : 0.99</td>
<td>1 : 0.92</td>
<td>1 : 0.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V16</td>
<td>M[ACCS-11]</td>
<td>3</td>
<td>83</td>
<td></td>
<td>1 : 0.73</td>
<td>1 : 0.87</td>
<td>1 : 1</td>
<td>1 : 0.88</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 0.92</td>
<td>1 : 0.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V17</td>
<td>M[ACCS-12]</td>
<td>3</td>
<td>30</td>
<td></td>
<td>1 : 0.23</td>
<td>1 : 0.17</td>
<td>1 : 1</td>
<td>1 : 0.2</td>
<td>1 : 1</td>
<td>1 : 0.99</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V18</td>
<td>M[ACCS-10]</td>
<td>6</td>
<td>31</td>
<td></td>
<td>1 : 0.63</td>
<td>1 : 0.72</td>
<td>1 : 1</td>
<td>1 : 0.73</td>
<td>1 : 1</td>
<td>1 : 1.02</td>
<td>1 : 0.92</td>
<td>1 : 0.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V19</td>
<td>M[ACCS-22]</td>
<td>6</td>
<td>31</td>
<td></td>
<td>1 : 0.17</td>
<td>1 : 0.09</td>
<td>1 : 1</td>
<td>1 : 0.12</td>
<td>1 : 1</td>
<td>1 : 1.02</td>
<td>1 : 0.92</td>
<td>1 : 0.9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Average Savings: 56.9 %, 53.2 %, 52.6 %, 1.4 %, 7.0 %, 8.6 %
Standard Deviation: 24.6 %, 35.0 %, 35.1 %, 8.0 %, 3.1 %, 3.8 %
CORREL(CIM): -0.59, -0.56, -0.56, -0.22, -0.14, -0.14

1. MSE achieved an average of savings in execution time that ranges from 56.9 % (of 9 min) for the ACCS model to 71.7 % (of 5 min) for the LGS model, while DSE achieved an average of savings in execution time that approximately ranges...
Table 7.6: Results of MSE and DSE on the versions of the FABP example with Modification Changes

<table>
<thead>
<tr>
<th>Ver.</th>
<th>Change ID / No. of Changed Transitions</th>
<th>No. of Multi-path Impacted Transitions</th>
<th>Time</th>
<th>No. of New Symbolic States</th>
<th>No. of Total Symbolic States</th>
<th>No. of New Execution Paths</th>
<th>No. of Total Execution Paths</th>
<th>Time</th>
<th>No. of Total Symbolic States</th>
<th>No. of Total Execution Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>M[Buff-01] / 1</td>
<td>0</td>
<td>1 : 0.71</td>
<td>1 : 0.77</td>
<td>1 : 1</td>
<td>1 : 0.78</td>
<td>1 : 1</td>
<td>1 : 0.03</td>
<td>1 : 0.03</td>
<td>1 : 0.04</td>
</tr>
<tr>
<td>V2</td>
<td>M[Buff-02] / 1</td>
<td>6</td>
<td>1 : 0.69</td>
<td>1 : 0.76</td>
<td>1 : 0.98</td>
<td>1 : 0.77</td>
<td>1 : 0.99</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V3</td>
<td>M[Buff-03] / 1</td>
<td>0</td>
<td>1 : 0.13</td>
<td>1 : 0.05</td>
<td>1 : 1</td>
<td>1 : 0.05</td>
<td>1 : 1</td>
<td>1 : 0.03</td>
<td>1 : 0.03</td>
<td>1 : 0.04</td>
</tr>
<tr>
<td>V4</td>
<td>M[Buff-04] / 1</td>
<td>6</td>
<td>1 : 0.57</td>
<td>1 : 0.56</td>
<td>1 : 1</td>
<td>1 : 0.62</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V5</td>
<td>M[Rcv-01] / 1</td>
<td>6</td>
<td>1 : 0.19</td>
<td>1 : 0.17</td>
<td>1 : 1</td>
<td>1 : 0.18</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V6</td>
<td>M[Rcv-02] / 1</td>
<td>6</td>
<td>1 : 0.44</td>
<td>1 : 0.6</td>
<td>1 : 1</td>
<td>1 : 0.49</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V7</td>
<td>M[Rcv-03] / 1</td>
<td>6</td>
<td>1 : 0.37</td>
<td>1 : 0.38</td>
<td>1 : 1</td>
<td>1 : 0.39</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V8</td>
<td>M[Rcv-04] / 1</td>
<td>6</td>
<td>1 : 0.27</td>
<td>1 : 0.3</td>
<td>1 : 1</td>
<td>1 : 0.31</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V9</td>
<td>M[Rcv-05] / 1</td>
<td>6</td>
<td>1 : 0.2</td>
<td>1 : 0.17</td>
<td>1 : 1</td>
<td>1 : 0.18</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V10</td>
<td>M[Rcv-06] / 1</td>
<td>6</td>
<td>1 : 0.41</td>
<td>1 : 0.38</td>
<td>1 : 1</td>
<td>1 : 0.38</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V11</td>
<td>M[Rcv-07] / 1</td>
<td>6</td>
<td>1 : 0.42</td>
<td>1 : 0.45</td>
<td>1 : 1</td>
<td>1 : 0.44</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V12</td>
<td>M[Rcv-08] / 1</td>
<td>6</td>
<td>1 : 0.68</td>
<td>1 : 0.73</td>
<td>1 : 1</td>
<td>1 : 0.72</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V13</td>
<td>M[Rcv-09] / 1</td>
<td>6</td>
<td>1 : 0.64</td>
<td>1 : 0.76</td>
<td>1 : 1</td>
<td>1 : 0.75</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V14</td>
<td>M[Rcv-10] / 1</td>
<td>0</td>
<td>1 : 0.16</td>
<td>1 : 0.08</td>
<td>1 : 1</td>
<td>1 : 0.11</td>
<td>1 : 1</td>
<td>1 : 0.03</td>
<td>1 : 0.03</td>
<td>1 : 0.04</td>
</tr>
<tr>
<td>V15</td>
<td>M[Rcv-11] / 1</td>
<td>0</td>
<td>1 : 0.34</td>
<td>1 : 0.28</td>
<td>1 : 1</td>
<td>1 : 0.38</td>
<td>1 : 1</td>
<td>1 : 0.03</td>
<td>1 : 0.03</td>
<td>1 : 0.04</td>
</tr>
<tr>
<td>V16</td>
<td>M[Rcv-12] / 1</td>
<td>0</td>
<td>1 : 0.29</td>
<td>1 : 0.23</td>
<td>1 : 1</td>
<td>1 : 0.23</td>
<td>1 : 1</td>
<td>1 : 0.03</td>
<td>1 : 0.03</td>
<td>1 : 0.04</td>
</tr>
<tr>
<td>V17</td>
<td>M[Rcv-13] / 1</td>
<td>0</td>
<td>1 : 0.47</td>
<td>1 : 0.5</td>
<td>1 : 1</td>
<td>1 : 0.51</td>
<td>1 : 1</td>
<td>1 : 0.03</td>
<td>1 : 0.03</td>
<td>1 : 0.04</td>
</tr>
<tr>
<td>V18</td>
<td>M[Rcv-14] / 1</td>
<td>6</td>
<td>1 : 0.63</td>
<td>1 : 0.7</td>
<td>1 : 1</td>
<td>1 : 0.71</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V19</td>
<td>M[Trans-01] / 1</td>
<td>6</td>
<td>1 : 0.55</td>
<td>1 : 0.55</td>
<td>1 : 1</td>
<td>1 : 0.71</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V20</td>
<td>M[Trans-02] / 1</td>
<td>6</td>
<td>1 : 0.74</td>
<td>1 : 0.88</td>
<td>1 : 1</td>
<td>1 : 0.89</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V21</td>
<td>M[Trans-03] / 1</td>
<td>6</td>
<td>1 : 0.12</td>
<td>1 : 0.01</td>
<td>1 : 1</td>
<td>1 : 0.02</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V22</td>
<td>M[Trans-04] / 1</td>
<td>6</td>
<td>1 : 0.43</td>
<td>1 : 0.44</td>
<td>1 : 1</td>
<td>1 : 0.41</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V23</td>
<td>M[Trans-05] / 1</td>
<td>6</td>
<td>1 : 0.58</td>
<td>1 : 0.64</td>
<td>1 : 1</td>
<td>1 : 0.63</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V24</td>
<td>M[Trans-06] / 1</td>
<td>6</td>
<td>1 : 0.12</td>
<td>1 : 0.01</td>
<td>1 : 1</td>
<td>1 : 0.01</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V25</td>
<td>M[Trans-07] / 1</td>
<td>6</td>
<td>1 : 0.88</td>
<td>1 : 0.99</td>
<td>1 : 1</td>
<td>1 : 0.99</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V26</td>
<td>M[Trans-08] / 1</td>
<td>6</td>
<td>1 : 0.77</td>
<td>1 : 0.83</td>
<td>1 : 1</td>
<td>1 : 0.85</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V27</td>
<td>M[Trans-09] / 1</td>
<td>6</td>
<td>1 : 0.29</td>
<td>1 : 0.27</td>
<td>1 : 1</td>
<td>1 : 0.27</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V28</td>
<td>M[Trans-10] / 1</td>
<td>6</td>
<td>1 : 0.34</td>
<td>1 : 0.32</td>
<td>1 : 1</td>
<td>1 : 0.33</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V29</td>
<td>M[Trans-11] / 1</td>
<td>6</td>
<td>1 : 0.14</td>
<td>1 : 0.04</td>
<td>1 : 1</td>
<td>1 : 0.04</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V30</td>
<td>M[Trans-12] / 1</td>
<td>6</td>
<td>1 : 0.25</td>
<td>1 : 0.19</td>
<td>1 : 1</td>
<td>1 : 0.16</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V31</td>
<td>M[Trans-13] / 1</td>
<td>6</td>
<td>1 : 0.14</td>
<td>1 : 0.01</td>
<td>1 : 1</td>
<td>1 : 0.01</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V32</td>
<td>M[Trans-14] / 1</td>
<td>6</td>
<td>1 : 0.13</td>
<td>1 : 0.02</td>
<td>1 : 1</td>
<td>1 : 0.04</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
<td>1 : 1</td>
</tr>
</tbody>
</table>

Average Savings: 59.2 % 58.9 % 58.6 % 18.2 % 18.1 % 18.0 %
Standard Deviation: 22.7 % 29.8 % 29.8 % 38.5 % 38.3 % 38.2 %
Table 7.7: Results of MSE and DSE on the versions of the FABP example with Addition Changes

<table>
<thead>
<tr>
<th>Ver.</th>
<th>Change ID</th>
<th>No. of Multi-path Impacted Transitions</th>
<th>No. of New Symbolic States</th>
<th>SE : MSE</th>
<th>SE : DSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>A[Buff-01] / 1</td>
<td>0</td>
<td>1 : 0.72</td>
<td>1 : 0.78</td>
<td>1 : 0.78</td>
</tr>
<tr>
<td>V2</td>
<td>A[Buff-02] / 1</td>
<td>6</td>
<td>1 : 0.72</td>
<td>1 : 0.8</td>
<td>1 : 0.98</td>
</tr>
<tr>
<td>V3</td>
<td>A[Buff-03] / 1</td>
<td>0</td>
<td>1 : 0.13</td>
<td>1 : 0.05</td>
<td>1 : 0.05</td>
</tr>
<tr>
<td>V4</td>
<td>A[Buff-04] / 1</td>
<td>6</td>
<td>1 : 0.53</td>
<td>1 : 0.57</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V5</td>
<td>A[Rev-01] / 1</td>
<td>6</td>
<td>1 : 0.21</td>
<td>1 : 0.17</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V6</td>
<td>A[Rev-02] / 1</td>
<td>6</td>
<td>1 : 0.5</td>
<td>1 : 0.5</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V7</td>
<td>A[Rev-03] / 1</td>
<td>6</td>
<td>1 : 0.41</td>
<td>1 : 0.38</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V8</td>
<td>A[Rev-04] / 1</td>
<td>6</td>
<td>1 : 0.35</td>
<td>1 : 0.3</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V9</td>
<td>A[Rev-05] / 1</td>
<td>6</td>
<td>1 : 0.23</td>
<td>1 : 0.17</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V10</td>
<td>A[Rev-06] / 1</td>
<td>6</td>
<td>1 : 0.4</td>
<td>1 : 0.38</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V11</td>
<td>A[Rev-07] / 1</td>
<td>6</td>
<td>1 : 0.44</td>
<td>1 : 0.45</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V12</td>
<td>A[Rev-08] / 1</td>
<td>6</td>
<td>1 : 0.72</td>
<td>1 : 0.74</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V13</td>
<td>A[Rev-09] / 1</td>
<td>6</td>
<td>1 : 0.71</td>
<td>1 : 0.76</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V14</td>
<td>A[Rev-10] / 1</td>
<td>0</td>
<td>1 : 0.17</td>
<td>1 : 0.09</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V15</td>
<td>A[Rev-11] / 1</td>
<td>0</td>
<td>1 : 0.4</td>
<td>1 : 0.28</td>
<td>1 : 0.98</td>
</tr>
<tr>
<td>V16</td>
<td>A[Rev-12] / 1</td>
<td>0</td>
<td>1 : 0.26</td>
<td>1 : 0.18</td>
<td>1 : 0.99</td>
</tr>
<tr>
<td>V17</td>
<td>A[Rev-13] / 1</td>
<td>0</td>
<td>1 : 0.48</td>
<td>1 : 0.5</td>
<td>1 : 0.7</td>
</tr>
<tr>
<td>V18</td>
<td>A[Rev-14] / 1</td>
<td>6</td>
<td>1 : 0.71</td>
<td>1 : 0.7</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V19</td>
<td>A[Trans-01] / 1</td>
<td>6</td>
<td>1 : 0.58</td>
<td>1 : 0.7</td>
<td>1 : 0.7</td>
</tr>
<tr>
<td>V20</td>
<td>A[Trans-02] / 1</td>
<td>6</td>
<td>1 : 0.83</td>
<td>1 : 0.88</td>
<td>1 : 0.98</td>
</tr>
<tr>
<td>V21</td>
<td>A[Trans-03] / 1</td>
<td>6</td>
<td>1 : 0.13</td>
<td>1 : 0.01</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V22</td>
<td>A[Trans-04] / 1</td>
<td>6</td>
<td>1 : 0.47</td>
<td>1 : 0.44</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V23</td>
<td>A[Trans-05] / 1</td>
<td>6</td>
<td>1 : 0.62</td>
<td>1 : 0.64</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V24</td>
<td>A[Trans-06] / 1</td>
<td>6</td>
<td>1 : 0.12</td>
<td>1 : 0.01</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V25</td>
<td>A[Trans-07] / 1</td>
<td>6</td>
<td>1 : 0.9</td>
<td>1 : 0.99</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V26</td>
<td>A[Trans-08] / 1</td>
<td>6</td>
<td>1 : 0.78</td>
<td>1 : 0.83</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V27</td>
<td>A[Trans-09] / 1</td>
<td>6</td>
<td>1 : 0.32</td>
<td>1 : 0.27</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V28</td>
<td>A[Trans-10] / 1</td>
<td>6</td>
<td>1 : 0.33</td>
<td>1 : 0.33</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V29</td>
<td>A[Trans-11] / 1</td>
<td>6</td>
<td>1 : 0.15</td>
<td>1 : 0.04</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V30</td>
<td>A[Trans-12] / 1</td>
<td>6</td>
<td>1 : 0.3</td>
<td>1 : 0.19</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V31</td>
<td>A[Trans-13] / 1</td>
<td>6</td>
<td>1 : 0.14</td>
<td>1 : 0.01</td>
<td>1 : 1</td>
</tr>
<tr>
<td>V32</td>
<td>A[Trans-14] / 1</td>
<td>6</td>
<td>1 : 0.13</td>
<td>1 : 0.02</td>
<td>1 : 1</td>
</tr>
</tbody>
</table>

Average Savings 56.6 % 59.0 % 66.1 % 18.2 % 18.1 % 18.0 %

Standard Deviation 23.9 % 30.1 % 29.1 % 38.5 % 38.3 % 38.2 %
Table 7.8: Results of MSE and DSE on the versions of the FABP example with Deletion Changes

<table>
<thead>
<tr>
<th>Change Information</th>
<th>SE : MSE</th>
<th>SE : DSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMLM</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ver.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Average Savings 82.3 %  91.0 %  91.1 %  18.0 %  17.9 %  17.8 %
Standard Deviation 13.9 %  16.5 %  16.6 %  38.0 %  37.9 %  37.7 %
Table 7.9: Summary of statistical measures of the effectiveness of MSE and DSE for the four models: AQS, LGS, ACCS and FABP with respect to the savings in execution time and in the number of symbolic states and execution paths

(a) Average and Standard Deviation of Savings in Time

<table>
<thead>
<tr>
<th></th>
<th>AVG</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MSE</td>
<td>71.0 %</td>
<td>71.7 %</td>
<td>56.9 %</td>
<td>66.0 %</td>
<td>10.9 %</td>
<td>21.6 %</td>
<td>24.6 %</td>
<td>23.5 %</td>
<td></td>
</tr>
<tr>
<td>DSE</td>
<td>98.4 %</td>
<td>76.7 %</td>
<td>1.4 %</td>
<td>18.1 %</td>
<td>0.2 %</td>
<td>2.2 %</td>
<td>8.0 %</td>
<td>37.9 %</td>
<td></td>
</tr>
</tbody>
</table>

(b) Average and Standard Deviation of Savings in Number of Symbolic States

<table>
<thead>
<tr>
<th></th>
<th>AVG</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MSE</td>
<td>75.5 %</td>
<td>68.2 %</td>
<td>53.2 %</td>
<td>69.6 %</td>
<td>28.0 %</td>
<td>30.6 %</td>
<td>35.0 %</td>
<td>30.1 %</td>
<td></td>
</tr>
<tr>
<td>DSE</td>
<td>95.6 %</td>
<td>67.0 %</td>
<td>7.0 %</td>
<td>18.0 %</td>
<td>4.4 %</td>
<td>10.5 %</td>
<td>3.1 %</td>
<td>37.7 %</td>
<td></td>
</tr>
</tbody>
</table>

(c) Average and Standard Deviation of Savings in Number of Execution Paths

<table>
<thead>
<tr>
<th></th>
<th>AVG</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MSE</td>
<td>74.9 %</td>
<td>67.2 %</td>
<td>52.6 %</td>
<td>69.3 %</td>
<td>28.8 %</td>
<td>30.5 %</td>
<td>35.1 %</td>
<td>30.3 %</td>
<td></td>
</tr>
<tr>
<td>DSE</td>
<td>96.1 %</td>
<td>66.6 %</td>
<td>8.6 %</td>
<td>18.0 %</td>
<td>4.2 %</td>
<td>10.4 %</td>
<td>3.8 %</td>
<td>37.6 %</td>
<td></td>
</tr>
</tbody>
</table>

from 1.4 % (of 9 min) for the ACCS model to 98.4 % (of 10 min) for the AQS model. Similarly, MSE achieved an average of savings in the number of symbolic states (resp., execution paths) that ranges from 53.2 % (resp., 52.6 %) for the ACCS model to 75.5 % (resp., 74.9 %) for the AQS model, while DSE achieved an average of savings in the number of symbolic states (resp., execution paths) that ranges from 7.0 % (resp., 8.6 %) for the ACCS model to 95.6 % (resp., 96.1 %) for the AQS model.

2. The range of the average values of the achieved savings ratios for DSE is larger than it is for MSE which means that DSE can achieve much higher savings for
Table 7.10: Summary of statistical measures of the effectiveness of MSE and DSE for the FABP model per change type with respect to the savings in time, in the numbers of symbolic states and in the numbers of execution paths

(a) Average and Standard Deviation of Savings in Time

<table>
<thead>
<tr>
<th></th>
<th>AVG</th>
<th>STDEV</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FABP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Modification</td>
<td>Addition</td>
</tr>
<tr>
<td>MSE</td>
<td>59.2 %</td>
<td>56.6 %</td>
</tr>
<tr>
<td>DSE</td>
<td>18.2 %</td>
<td>18.2 %</td>
</tr>
</tbody>
</table>

(b) Average and Standard Deviation of Savings in Numbers of Symbolic States

<table>
<thead>
<tr>
<th></th>
<th>AVG</th>
<th>STDEV</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FABP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Modification</td>
<td>Addition</td>
</tr>
<tr>
<td>MSE</td>
<td>58.9 %</td>
<td>59.0 %</td>
</tr>
<tr>
<td>DSE</td>
<td>18.1 %</td>
<td>18.1 %</td>
</tr>
</tbody>
</table>

(c) Average and Standard Deviation of Savings in Numbers of Execution Paths

<table>
<thead>
<tr>
<th></th>
<th>AVG</th>
<th>STDEV</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FABP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Modification</td>
<td>Addition</td>
</tr>
<tr>
<td>MSE</td>
<td>58.6 %</td>
<td>58.3 %</td>
</tr>
<tr>
<td>DSE</td>
<td>18.0 %</td>
<td>18.0 %</td>
</tr>
</tbody>
</table>

some models than the others. This shows a higher sensitivity of DSE to the subject models than it is for MSE.

3. The standard deviation values of the achieved savings ratios for the AQS, LGS and ACCS models are much higher for MSE than they are for DSE, which means that the effectiveness of MSE for these three models is more influenced by the changes made in each modified model version than it is for the DSE. Interestingly, this observation is reversed for the FABP model.
RQ2. Does the SET generated from MSE match the one generated from standard SE?

The main purpose of this research question is twofold: 1) to ensure the correctness of our MSE algorithms and their implementation and 2) to draw a precise conclusion about the relationship between the SET generated from MSE and the one generated from standard SE. Our initial expectation was that both SETs should be identical, however the results show that, in some cases, this assumption does not hold.

1. For individual Statecharts: By looking at the ratios recorded in columns 7 and 9 of Table 7.3–7.5 and our manual inspection, we notice that MSE generated the exact same SETs as standard SE for all versions of the three models: AQS, LGS and ACCS.

2. For communicating Statecharts: By looking at the ratios recorded in columns 6 and 8 of Tables 7.6–7.8 and our manual inspection, we notice that MSE generated the exact same GSETs as standard SE for 26 of the 32 FABP model versions with “modification” and “addition” changes and all the FABP model versions with “deletion” changes. However, for all other versions, MSE generated slightly different GSETs. Possible reasons for this are that: 1) the subsumption relation that may exist between the nodes of a SET (resp., GSET) is non-symmetric and 2) modifying an existing SET (resp., GSET) may change the order of the nodes of the tree as well as the subsumption relation between the nodes. As a result, we may have two semantically equivalent SETs with different numbers of nodes for the same model if the order of generating the nodes of the tree is changed. In Figure 7.1 we show an example of two differently sized SETs of the same model resulting from changing the order in which the
7.4. RESULTS AND ANALYSIS

(a) An example SET where the branch leading to node “c” is created before the branch leading to node “d”.

(b) The SET resulting from modifying the SET in (a) such that the branch leading to node “c” is now created after the branch leading to node “d”.

Figure 7.1: An example of two differently sized SETs of the same model, each of which has a different branch order. Note the reduction in the number of nodes in the SET in (b) compared to the one in (a) as a result of the subsumption relation (SR) that is detected in the SET in (b) between node “j” and the previously visited node “l” which cannot exist if the creation order of both nodes is reversed as it is the case in the SET in (a).

different branches of the tree have been created. It presents a typical scenario that occurs when running the MSE algorithms where manipulating an already existing SET (resp., GSET) takes place to modify some parts of the tree or to add new ones.

Based on the results of this research question, we conclude that it is possible in some cases for MSE to generate a SET (resp., GSET) that does not exactly match
the one generated from standard SE, yet they are both semantically equivalent.

**RQ3. Which aspects influence the effectiveness of each technique?**

We summarize the key findings related to the two aforementioned aspects as follows.

1. For the change impact metric:

   - Change impact measure criterion (applicable only for individual State-charts models AQS, LGS and ACCS): According to the correlation coefficients computed between the values of columns 5-6, 8, 10-12 and the values of column 4 of Tables 7.3–7.5, we notice that there is high negative correlation between the savings gained from MSE and the CIM metric for the AQS and the LGS models, meaning that in these cases the percentage of maximal paths affected by the change is a good predictor for the effectiveness of MSE. However, it is much lower for the ACCS model and this is due to the discrepancy between the notions of maximal paths and execution paths. A maximal path in a model may correspond to zero, one or multiple feasible execution paths in the SET of the model depending on the guard condition and the action code of the transitions in the path. This discrepancy grows with the numbers of parallel transitions in the model (i.e., transitions with the same source and target states and guards but different triggers). The ACCS model contains more of these parallel transitions than the AQS and LGS models. We also notice that there is almost no significant correlation between the savings gained from DSE and the CIM metric.
7.4. RESULTS AND ANALYSIS

- Type of change criterion (applicable only for communicating Statecharts model FABP): In Table 7.10, we show a summary of the average and the standard deviation values of the data shown in columns 4-5, 7 (resp., 9-11) of Tables 7.6–7.8 corresponding to the amount of savings in the execution times and in the numbers of symbolic states and execution paths gained from applying MSE (resp., DSE) on the different versions of the FABP model per change type. Based on the values presented in this table, we notice that MSE achieved higher average savings for the versions of the FABP model with deletion changes than for the versions the FABP model with modification and addition changes, while DSE achieved similar average savings for the versions of each change type. This indicates that DSE is less influenced by the type of the changes made to the subject model.

2. For the model characteristics metrics: By looking at the characteristics of the four models in Table 7.1 and the summary of the average and the standard deviation values of the amount of savings in the numbers of symbolic states gained for the four models in Table 7.9, we notice the following:

- The differences between the average of savings gained among the four used models are not as significant for MSE as they are for DSE. As a result we could not find a clear relationship between the two defined metrics and the savings gained from MSE.

- DSE achieved the least average savings for the ACCS model. This is because ACCS has the least number of multi-path transitions which decreases the opportunity for savings from the partial exploration implemented in
DSE.

- DSE achieved much higher average savings for both the AQS and LGS models compared with the FABP model. This is because both AQS and LGS have less dependency between their multi-path transitions compared with the FABP model which has the highest number of multi-path dependent transitions. As we mentioned earlier, a high dependency between multi-path transitions in a model decreases the opportunity for savings from the partial exploration implemented in DSE.

- The standard deviation of the savings gained from DSE for the FABP model is the highest among the other models and it is even higher than the average value of the savings. This is because: 1) 78 of the 96 of the FABP versions have changes in transitions that have dependencies with all the multi-path transitions in the model and therefore result in 0% savings with DSE and 2) only 18 of the 96 of the FABP versions have changes in transitions that have no dependencies with any of the multi-path transitions in the model and therefore result in 96% savings with DSE.

Based on the key findings of this research question, we conclude that our optimization techniques are complementary in the sense that the effectiveness of MSE depends mostly on the impact and the type of the change, while the effectiveness of DSE depends more on the numbers of transitions with multi-path guards or action code as well as the numbers of transitions with dependencies with each other.
7.5  Threats to Validity

Threats to external validity for our evaluation may include: 1) the use of KLEE in the implementation of our proposed techniques, 2) the selection of artifacts used in the evaluation, and 3) the types of changes applied to create the modified versions of the used models. Replacing KLEE with any other symbolic execution engine could produce slightly different SETs, depending on the searching strategy employed to explore the different paths of the code been executed, but this should have no impact on the effectiveness of our proposed techniques and rerunning the same evaluation steps with the other symbolic execution engines would mitigate this threat. The artifacts selected for our study are typical behavioral models that can be systematically explored using symbolic execution. The types of changes we used to create the modified versions were selective and may not reflect actual changes; however, rerunning the evaluation on actual model versions would address this threat.

Threats to internal validity may include: 1) the originality of the proposed techniques and 2) the correctness of the implementation of our transformation components and proposed algorithms. Although the work presented here is inspired by an already existing work for evolving programs, we are the first to apply it in the context of evolving state-based models. Regarding the correctness of our implementation, we have already tried to address this threat through extensive testing and inspection; more experimentation might allow us to reduce the risk of bugs further.

7.6  Summary

In this chapter we have provided the evaluation of the proposed optimization techniques. We have identified the research questions and the criteria that the evaluation
should investigate and the artifacts that we use to perform the evaluation. Three models of hierarchical individual Statecharts as well as one model of three communicating Statecharts are used to evaluate the optimization techniques for each type. An analysis of the results shows a considerable amount of savings resulting from both optimization techniques. In certain versions the savings have reached up to 90+. The results of MSE have shown its sensitivity to the location and the type of the change made in the models versions, while the results of DSE have shown its sensitivity to the characteristics of the subject models that are related to the existence of multi-path transitions in these models that have few dependencies with other transitions.
Chapter 8

Conclusions and Future Work

In this thesis, we have presented two different techniques for optimizing the symbolic execution of evolving Rhapsody Statecharts. The first technique reuses the symbolic execution tree of a previous version of a model to improve the symbolic execution of the current version such that it avoids redundant exploration of common execution paths between the two versions, whereas the second technique uses a change impact analysis to reduce the scope of the exploration to mainly exercise the parts impacted by the change. A key contribution in this work is a proof-of-concept of the feasibility and the effectiveness of the two proposed optimization techniques such that they can be used to speed up the analysis process of such behavioral models as they evolve.

The two techniques proposed in this work are applied on both individual and communicating Statecharts and are implemented and integrated in the IBM Rhapsody tool as a set of three external helpers. The first helper represents the transformation components which are built in the context of IBM Rhapsody RulesComposer Add-On and are responsible for 1) flattening a Rhapsody model into our Communicating Mealy-like Machines (CMLM) internal representation, 2) compute the data and the
communication dependencies in a CMLM and its MLMs and 3) generate a Java representation of the CMLM representation. The second helper represents the component that is responsible for: 1) mapping a difference report that is generated from the Rhapsody DiffMerge tool to compare between two successive versions of a Rhapsody model into a difference report between the CMLM representation of each version and 2) identifying the impacted elements of each change found in the difference report. The last helper represents our KLEE-based symbolic execution components including standard SE and the two proposed optimizations: MSE and DSE. We performed an extensive evaluation of the two proposed techniques on a set of three individual hierarchical Statecharts and one model of three communicating Statecharts with a set of 160 different model versions and different types of atomic changes including modification, addition and deletion of the transitions in these models. The results from our experiments show a significant amount of savings up to 90+ % in certain scenarios with respect to the size of the symbolic execution trees generated from applying either technique and the time taken to generate them. The results also highlight the key factors that mostly influence the effectiveness of each technique. These are the change type and location for the MSE technique and the model characteristics concerning the number of multi-path transitions in a model and the dependencies between its transitions for the DSE technique.

Some directions for future work include but not limited to the following:

1. The first direction is to extend the support for more advanced features of action code including more complex data types and user-defined method calls. Both features can be easily added to the current implementation by extending our interface component with the KLEE symbolic execution engine.
2. Another important direction is to extend the support for some user-defined or domain-specific semantics of specific Statechart features (e.g., priority schema for conflicting transitions and execution order of concurrent states) that is used in practice to override the default one of the tool. To the best of our knowledge, only practitioners who build these models can determine such refinements. In such cases, a modification to our current transformation components should be made.

3. The third direction is to extend the symbolic execution tree (SET) representation to include the test case values of the symbolic variables used to reach a feasible symbolic state such that a complete set of traces can be extracted for a generated SET to be used as a driver for model-based testing and runtime monitoring. This extension is feasible and is supported by the KLEE’s “ktest-tool” utility that enables the reading of the contents of the “.ktest” binary files of the test cases generated by the tool.

4. The fourth direction is to extend the evaluation of the proposed work to consider more models with actual versions. This extension is limited by the accessibility to existing model repositories.

5. The fifth direction is to build a SET-based query engine to facilitate user queries and visualization of sub-parts of the tree. This feature is very important to provide us with a more precise and automated way to inspect some features in a given SET or to compare two SETs.

6. Finally, the proposed work in this thesis can be adapted to consider other state-based models used in other MDE tools, especially the open-source ones where
the opportunity for extensibility is higher and is not limited by the use of some proprietary tools.
Bibliography


Appendix A

Ecore meta-model of CMLMs

In this appendix, we present the Ecore meta-model of our CMLMs. For clarity, we show it in two parts. The first part highlights the elements used to represent individual Mealy-like machines. The second part highlights the elements used to represent communicating Mealy-like machines.
Figure A.1: CMLMs Ecore Meta-model – Part 1/2
Figure A.2: CMLMs Ecore Meta-model – Part 2/2