Incremental Symbolic Execution of Evolving State Machines using Memoization and Dependence Analysis

Amal Khalil and Juergen Dingel

{khalil, dingel}@cs.queensu.ca

Technical Report 2015-623

School of Computing, Queen's University

Kingston, Ontario, Canada, K7L 2N8

May 7, 2015

Contents

List of Figures

List of Tables

1 Introduction

Iterative-incremental development and model analysis are central to Model Driven Engineering (MDE). Software artifacts can undergo several iterations and refinements. As these artifacts evolve, it is necessary to assess their quality by repeating the analysis and the verification of these artifacts after every iteration or refinement. This process, if not optimized, can be very tedious and time consuming.

Symbolic execution is a well-known analysis technique that is used to analyze the execution paths of behavioral software artifacts (e.g., programs and state-based models). The output from running this technique 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 with big and complex artifacts where the size of the output symbolic execution trees becomes very large. Repeating the entire analysis as an artifact evolves is not the best solution.

Inspired by the research work for optimizing the symbolic execution of evolving programs [20, 27], we propose two different optimization techniques to direct successive runs of the symbolic execution technique to cover only the impacted parts of an evolving state machine. The first technique reuses the symbolic execution tree (SET) generated from a previous analysis and incrementally updates the relevant parts that have been changed in the new version. The second technique integrates a dependency analysis technique with the symbolic execution to implement what we call dependency-based symbolic execution. A dependence analysis allows for the identification of parts of the state machine that are unaffected by the change; complete exploration of these parts is not necessary, and only a single representative path needs to be found during exploration. Consequently, the resulting SET is not complete, but it is sufficient to determine the impact of the change on any SET-based analysis.

The two proposed techniques were run on a total of three industrial-sized state machine models. The initial results showed a 64% average reduction in the size of the symbolic execution trees generated and a 62% average reduction in the time it took to generate them for the memoized-based symbolic execution. It also showed a 57% average reduction in the size of the symbolic execution trees generated and a 59% average reduction in the time it took to generate them for the dependency-based symbolic execution.

A well-known example of applications that can benefit from such optimization techniques is regression testing which is a crucial software evolution task that aims at ensuring that no unintended behavior has been introduced to the system after the change. A naive approach of regression testing usually depends on re-running some existing test suite which has been generated to validate the previous version of the system. This process is expensive and time consuming and it is usually supported

with some optimization mechanisms including test case selection and prioritization. Since symbolic execution can be used as the seed for generating these test suites, optimizing the symbolic execution of the new versions of an artifact can also be added to these optimization mechanisms.

The structure of the paper is as follows: Section 2 provides the background required to understand the context of our work; Section 3 introduces our approach to build the the symbolic execution for Rhapsody Statecharts; Section 4 motivates the ideas of reuse and reduce for optimizing the symbolic execution of evolving Rhapsody Statecharts; Section 5 provides an overview of our proposed techniques; Section 6 gives the details about the implementation and the evaluation of the proposed techniques; Section 7 covers the related work; and Section 8 concludes this report.

2 Background

2.1 Rhapsody Statecharts[1]

The IBM Rational Rhapsody framework [22] is one of the commercial tools that supports 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). Rhapsody Statecharts (also known as Harel's Statecharts [12]) are a visual state-based formalism implemented in the IBM Rational Rhapsody framework to describe the behavior of reactive systems. They extend Mealy Machines - a type of Finite State Machines (FSMs) that perform their action only on firing transitions - with state action, state hierarchy and concurrency. An action is a reactive behavior associated with a state/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 and/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 labeled with an event and optionally with a guard and an action. Events are stimuli delivered to the statechart from the environment 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 a Boolean expression over

[1]The term "Statechart" is used exchangeably with the term "State Machine" in this report.

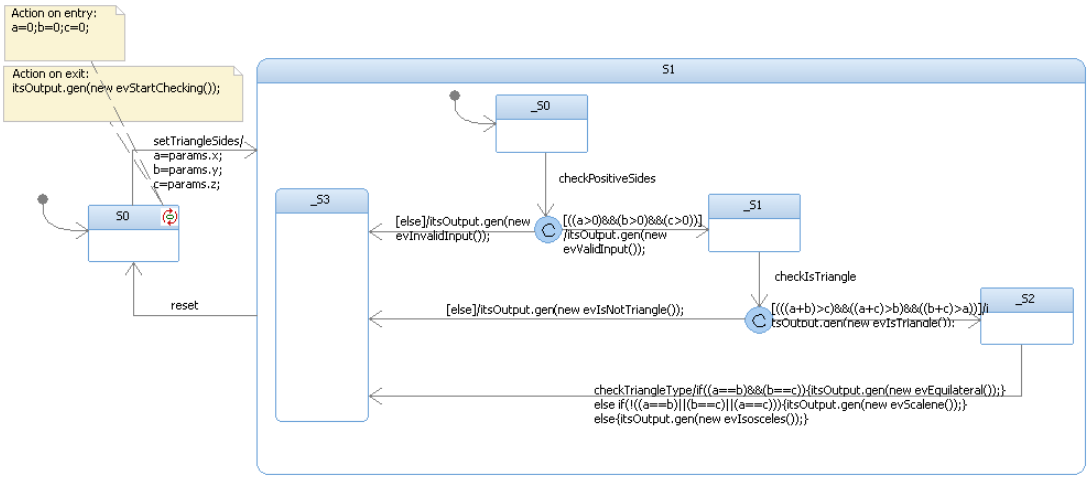


Figure 1: An Example of a Rhapsody Statechart

machine variables and/or events arguments. A transition is fired when its source state is active, when its corresponding event is received, when its guard evaluates to true and when no higher priority transition is enabled. When a transition is fired, its action is executed and, consequently, the system goes to the transition's target state. The detailed semantics of Statecharts is very rich and it is outside the scope of this work to thoroughly discuss it; interested readers are advised to look at [12] for more details. Figure 1 shows an example of Rhapsody Statechart which has been adapted from [13] with the subset of features that we consider in this work.

Although the tool eases the modeling process, it has only limited capabilities for verifying these models. In such contexts, verification tends to be carried out outside the scope of the tool using some advanced verification techniques and usually to verify the generated code rather than the models [18]. The main problem of this practice is that, although the analysis of code is typically more difficult (code contains more details), it does not guarantee the correctness of the design; some design faults cannot be easily detected by code-level verification methods. Another problem is the difficulty of tracing back the errors generated in the code to its source at the model level. Also, the effort to generate executable code can slow the analysis down. The motivation of our research is to highlight the importance of the types of analysis/verifications to be carried out on the model level.

2.2 Symbolic Execution

Symbolic execution is a decades old static analysis technique for systematically exploring potential execution paths of programs [14], [5]. It uses symbolic values instead of concrete data as programs inputs. As the execution proceeds along

program statements, it updates the values of program variables and the path condition along each path. A path condition represents all accumulated constraints that inputs have to satisfy for the particular path to be executed. Both the values of program variables and path conditions are computed as expressions and logical formulas, respectively, over constants and symbolic input values. A path is feasible if its path condition is satisfiable.

A symbolic execution tree (SET) contains the execution paths constructed during the symbolic execution of a program. The nodes of the tree represent the symbolic program states where each state defines a program location, the variables' valuations at this location and the set of constraints collected to reach this location (i.e., the path condition). The edges of the tree represent transitions between states (i.e., program locations). The resulting symbolic execution tree forms the basis for many program analysis, verification and testing activities. 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.

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 [10], Statecharts [25], UML State Machines [3] and recently UML-RT State Machines [28].

2.3 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 [23]. Some of them consider the syntactic similarities between models elements and others try to search for the semantic similarities as well [19]. 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 form 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 which are assigned internally to model elements once created to match between the elements in each model and then to find the differences between the matched ones.

2.4 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 the notion of potential interaction within such artifacts, e.g., Extendend Finite State Machines (EFSMs) [2] where ineractions are defined between transitions since they are the "active" elements of such types of models. In our setting, we transform statecharts to EFSMs-like representation, therefore we can adopt the definitions of such dependencies from conventional EFSMs.

In the sequel, we will refer to a set of definitions for computing control and data dependences that have been adopted from [7], which is based on the work presented in [2, 16, 21].

2.4.1 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 of control dependence in such formalisms. The one that applies to state machines with 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. A cycle is a set of transitions that forms a strongly connected component. Note that the initial state is used as an end state when computing maximal paths. The reason for this is that reaching the initial state in a reactive system is analogous to reaching an end node in a program in the sense that the behavior will start over again.

Definition 2. ***(Non-Termination Sensitive Control Dependence (NTSCD)).*** *$t_i \xrightarrow{NTSCD} t_j$ means that t_j is non-termination sensitive control dependent on a transition t_i iff:*

1. for all paths $\pi \in$ *MaximalPaths(targetState(t_i))*, the *sourceState(t_j)* belongs to π;
2. t_i has at least one sibling t_k and there exists a path $\pi' \in$ *MaximalPaths(sourceState(t_k))* such that *sourceState(t_j)* does not belong to π';

where *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_i if *sourceState(t_k)*=*sourceState(t_i)*.

2.4.2 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 [2].

Definition 3. ***(Data Dependence (DD)).*** *$t_i \xrightarrow{DD} t_k$ means that t_k is data dependent on t_i with respect to variable v if:*

1. $v \in Def(t_i)$, where $Def(t_i)$ is the set of variables defined by the action code of transition t_i;
2. $v \in Use(t_k)$, where $Use(t_k)$ 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 *targetState*(t_k) along which v is not modified.

Example 1. By applying Definitions 1-3 to the state-based model shown in Figure 3, we find that there are six maximal paths in this model composed of the following sequences of transitions: [*T1*, *T4*], [*T1*, *T3*, *T10*], [*T1*, *T2*, *T7*], [*T1*, *T2*, *T6*, *T10*], [*T1*, *T2*, *T5*, *T9*], and [*T1*, *T2*, *T5*, *T8*, *T10*]. Also, there are control dependences between each of the following transitions: *T2* $\xrightarrow{CD}$ *T5*, *T2* $\xrightarrow{CD}$ *T6*, *T2* $\xrightarrow{CD}$ *T7*, *T5* $\xrightarrow{CD}$ *T8* and *T5* $\xrightarrow{CD}$ *T9*, while there are data dependences between each of

Figure 2: Symbolic Execution (SE) of Rhapsody Statecharts

the following transitions: $T1 \xrightarrow{DD}$ T2, $T1 \xrightarrow{DD} T3$, $T1 \xrightarrow{DD} T5$, $T1 \xrightarrow{DD} T6$ and $T1 \xrightarrow{DD} T8$.

3 Symbolic Execution of Rhapsody Statecharts

In this section, we explain how we build the symbolic execution of Rhapsody Statecharts; the base module for our optimization techiques is presented in Section 6. We followed the approach of Zurowska and Dingel [28] with some variations to develop a symbolic execution module for Rhapsody Statecharts. In contrast to their work [28], our intermediate machine representation of Rhapsody Statecharts takes the form of Mealy 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 [4], to symbolically execute action code encountered in the Statecharts, whereas they use an in-house symbolic execution engine. The reason for choosing the Mealy Machine formalism is to have actions associated only with transitions, and we do this in such a way that it preserves the behavior of Statecharts.

The two components of our technique, as shown in Figure 2, are: 1) a translator that translates a Statechart model to a Mealy-like Machine representation (SC2MLM Transformation) and 2) an executor that traverses the Mealy Machine model and symbolically executes the action code encountered in each transition to build its symbolic state space (KLEE-based MLM2SET Generation).

3.1 Statecharts to Mealy-like Machine Transformation: SC2MLM

The basic structure of our 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 those states. Simple states in Mealy-like Machines do not have entry/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,

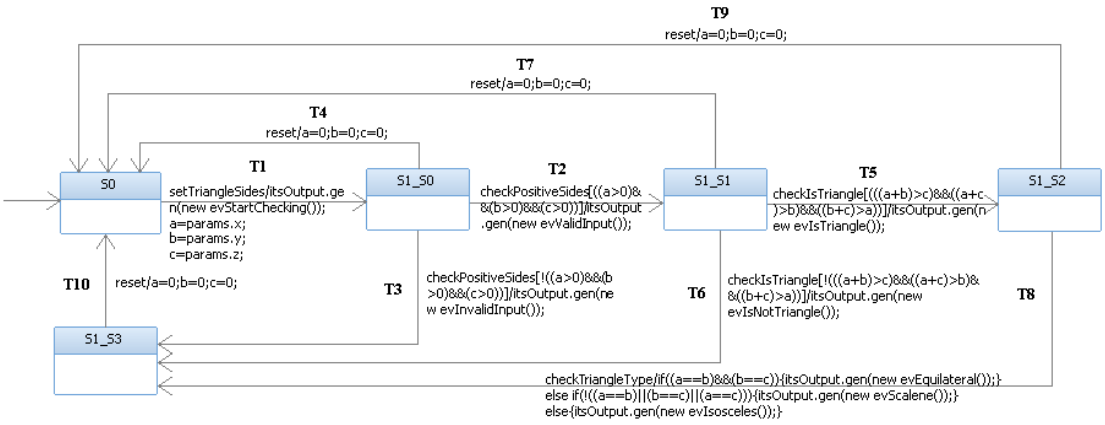


Figure 3: The Mealy-like Machine of the Rhapsody Statechart in Figure 1

states with entry/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 those specific features. For example: 1) Composite and concurrent states and their outgoing group transitions (formally called group transitions or high-level transitions) are flattened into simple ones; 2) Choice points along with their incoming and outgoing branches are replaced with newly created transitions connecting the source state of each choice point with its target states; 3) Entry actions of each state are added at the end of the action code of its incoming transitions; and 4) Exit actions of each state are added at the begining of the action code of its outgoing transitions. Figure 3 shows the Mealy-like Machine that represents the Rhapsody Statechart shown in Figure 1.

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

- S is a nonempty finite set of states;
- V is a set of globally typed 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 (called also triggers or signals) by which the machine communicates with its environment; they are divided into disjoint sets of input events, E^i, output events, E^o, and internal events, E^{int} , and can optionally have arguments (also called parameters), EA;
- EA is a finite set of event arguments that is disjoint with V;
- T is a set of transitions connecting the states in S. A transition is a tuple $t=(s,\ e,\ eA,\ G,\ A,\ s')$, where:

- s and s' are the source state and the target state, respectively;
- e is one of the machine input/internal events in E^i and 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 it 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, constraining some of the variables in V, or the parameters in eA, or it may cause a sequence of output events in E^o to be sent;

- $s_0 \in S$ is the initial state.
- V_0 is the initial valuation of the machine variables in V.

Example 2. Figure 3 presents an example of an *MLM* with:

- S={*S0*, *S1_S0*, *S1_S1*, *S1_S2*, *S1_S3*};
- V={a, b, c};
- E^i={*setTraingleSides*(x, y, z), *checkPositiveSides*(), *checkIsTriangle*(), *checkTriangleType*(), *reset*()};
- E^o={*evStartChecking*(), *evValidInput*(), *evInvalidInput*(), *evIsTriangle*(), *evIsNotTriangle*(), *evEquilateral*(), *evScalene*(), *evIsosceles*()};
- E^{int}={};
- EA={x, y, z};
- s_0=*S0*;
- V_0={a=0, b=0, c=0};
- *T1*=(*S0*, *setTraingleSides*, {x, y, z}, {}, *A1*, *S1_S0*);
- *T2*=(*S1_S0*, *checkPositiveSides*, {}, {(a>0)&&(b>0)&&(c>0)}, *A2*, *S1_S1*)
- ...
- *T10*=(*S1_S3*, *reset*, {}, {}, *A10*, *S0*).

3.2 Symbolic Execution of Mealy-like Machine: MLM2SET

The main idea is the same as 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 those locations and the path constraints collected to reach those locations; therefore, we call them symbolic. The edges in this tree are links between symbolic locations/states, and they reflect the control flow of the execution. In the symbolic execution of MLMs these edges are labeled with the event causing the transition (along 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 symbolic transitions. A symbolic execution path is a sequence of one or more consecutive symbolic transitions in a given symbolic execution tree (SET). 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 is set to "true".

Formally, a symbolic execution tree of a Mealy-Like-Machine, MLM=(S, V, E, EA, T, s_0, V_0), is a tuple SET=(SS, L, ST, ss_0) 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, V^s, σ, Seq) where,
  - e is an input or internal event in E^i or E^{int};
  - V^s is a set of symbolic variables, different from V and EA;

  - σ is a mapping from the original event argument names $eA \subset EA$ to the new set of variables names in V^s;
  - Seq is an optional sequence of output events paired with symbolic valuations of their arguments (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, 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 1.

Algorithm 1 Standard Symbolic Execution (SE) Algorithm

Input: A Mealy-Like Machine $MLM=(S,\ V,\ E,\ EA,\ T,\ s_0,\ V_0)$
Output: A Symbolic Execution Tree $SET=(SS,\ L,\ ST,\ ss_0)$

1: $ss_0 \leftarrow (s_0,\ v_0,\ \text{true})$
2: $SS \leftarrow \{ss_0\}$
3: Create a queue q
4: $q \leftarrow [ss_0]$
5: //Explore the MLM in Breadth-First-Search fashion
6: **while** q is not empty **do**
7: *currentStateToExplore*$\leftarrow$ 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 V^s of new variables in one-to-one relation with the transition's event arguments eA
10: Create a map σ between eA and V^s
11: 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 $A^s \leftarrow \sigma(A)$
12: **if** G^s is satisfiable given val and pc **then**
13: *feasiblePaths*$\leftarrow$ *SymbolicExecutionOfCode*$(A^s,\ val,\ G^s,\ pc)$
14: **for all** $feasiblePath=(val',\ pc',\ out')\in feasiblePaths$ **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: $SS \leftarrow SS \cup \{ss'\}$
18: Create a new label, $l'=(e,\ V^s,\ \sigma,\ out')$
19: $L \leftarrow L \cup \{l'\}$
20: Create a new symbolic transition, $st'=(ss,\ l',\ ss')$
21: $ST \leftarrow ST \cup \{st'\}$
22: $q \leftarrow enqueue(q,\ ss')$
23: **end if**
24: **end for**
25: **end if**
26: **end for**
27: **end while**

The first task in this algorithm concerns the exploration of the MLM, which is done in a breadth-first-search fashion, 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. A symbolic state $ss=(s,\ val,\ pc)$ is subsumed by another symbolic state $ss'=(s',\ val',\ pc')$ if $s=s'$, $val=val'$, and pc is included in pc' (i.e., $pc \Rightarrow pc'$ or

pc is more constrained than *pc'*).

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 those 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 the symbolic execution engin KLEE to execute the updated action code of the transition; the result from KLEE is 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 states/nodes representing the target state of this transition and also to label the edges to these new symbolic states/nodes (Lines 15-19).

The complete SET of the Mealy-like Machine in Figure 3 is shown in the top left corner of Figure 4. As we notice, the SET is of depth 6, and it has 14 symbolic execution paths identified by the number of leaves in the tree. In the following, we show the detailed steps for generating the first three levels of the tree, based on Algorithm 1.

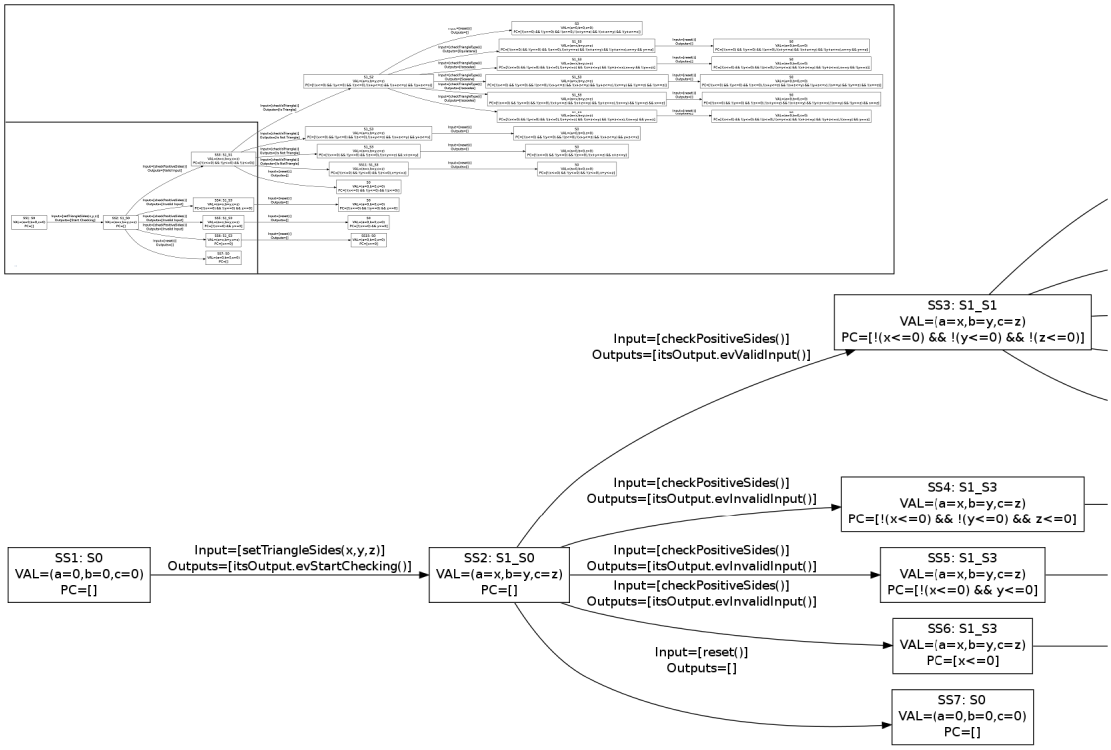
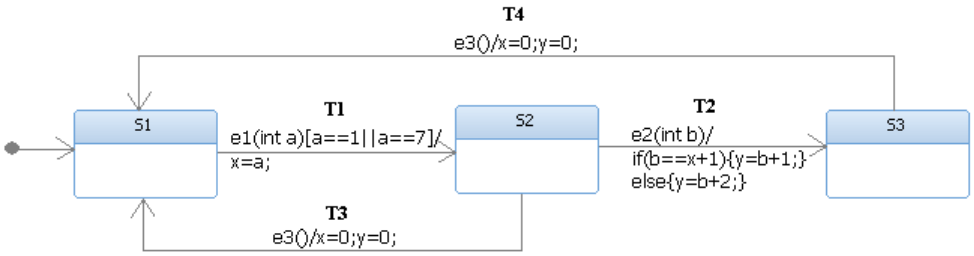


Figure 4: *SET of checkTriangleType* Statechart (highlighted levels: 1, 2 and 3)

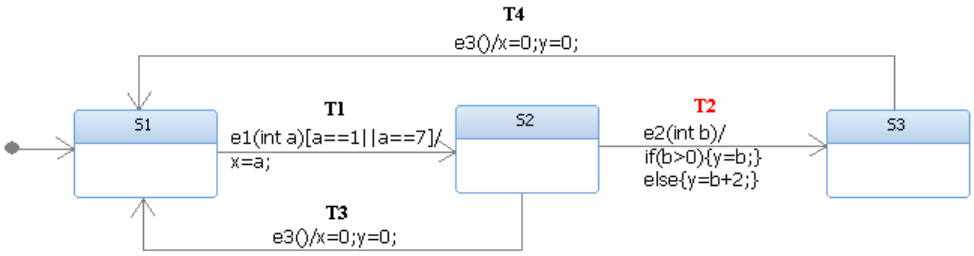
Example 3. As it is shown, the root of the tree is the symbolic state *SS1*=(*S0*, (*a*=0, *b*=0, *c*=0), []). Exploring state *S0* of *SS1* will symbolically execute its outgoing transition *T1*, which creates new symbolic values *x*, *y* and *z*, assigns them to the arguments *X, Y* and *Z* of the event *setTraingleSides*(), updates the machine variables *a*, *b* and *c* to have the new symbolic values of the *setTraingleSides* 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*=[*setTriangleSides*(*x*, *y*, *z*)] and the output *out*=[*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 on 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*), [!(*x*¡=0)ˆ!(*y*¡=0)ˆ!(*z*¡=0)]) and forms the label for the edge between *SS2* and *SS3* with the transition event *e*=[*checkPositiveSides*()] and the output *out*=[*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 variables valuation *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*=[*checkPositiveSide*()] and the output *out*=[*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*=[*reset*()] and the output *out*=[]. The difference between the symbolic state *SS7* and ther other symbolic states *SS4*, *SS5* and *SS6* is that we find that there is another symbolic state, *SS1*, that has been already explored before, and it subsumes *SS7*, therefore no further exploration is needed for *SS7* compared with the others which need to be explored.

4 Motivating Example

Symbolic execution is an expensive approach: scalability (known as the path-explosion problem) is one of the main challenges of this technique especially when applying it to big, complex artifacts (programs or models) where the number of execution paths increases dramatically. Software artifacts can undergo several iterations and refinements and repeating the symbolic execution of those 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 similar/unchanged parts from successive runs of the symbolic



(a) *V0* - The base version of an example Statechart



(b) *V1* - A modified version of the Statechart in (a)

Figure 5: Two Versions of an Example Statechart

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 [27] and [20], we propose two techniques for optimizing the symbolic execution of an evolving state machine model. We use the models in Figure 5 and their symbolic execution trees as shown in Figures 6 and 7 to illustrate how our optimization techniques work.

In Figure 5, we have two versions of an example Rhapsody Statechart model, *V0* (in Figure 5a) is the base version, and V1 (in Figure 5b) is a modified version of the base version where we modified the conditional statement in the action code of transition *T2*. In Figure 6, we show the symbolic execution tree of V0, SET(V0), however in Figures 7a and 7b, we show two different illustrations for the symbolic execution tree of V1, SET(V1).

4.1 Optimization Via Reuse

By comparing the unhighlighted parts (i.e., those coloured black) of the symbolic execution tree of *V1*, $SET(V1)$, in Figure 7a with the symbolic execution tree of *V0*, $SET(V0)$, in Figure 6, we notice that these parts have their equivalents in

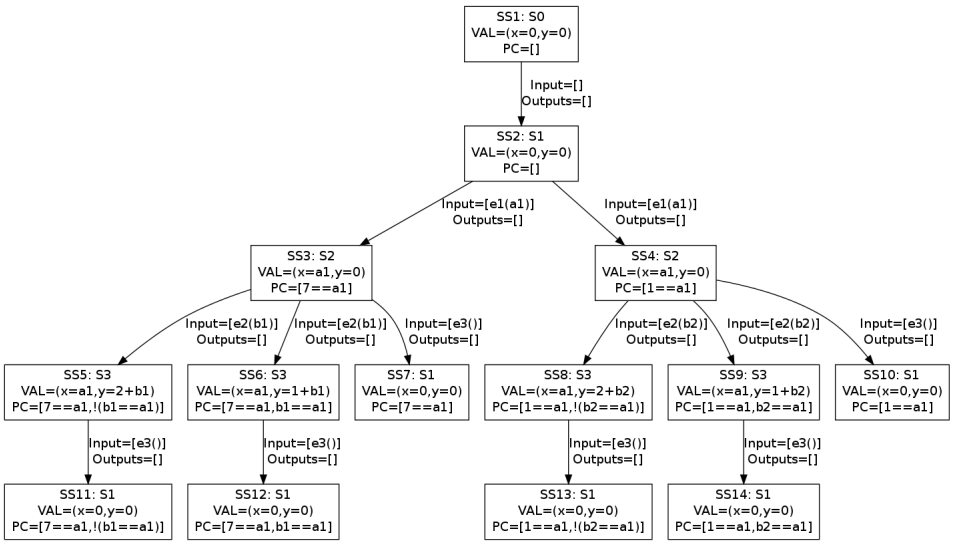


Figure 6: *SET(V0)* - Symbolic Execution Tree for the Statechart in Figure 5a

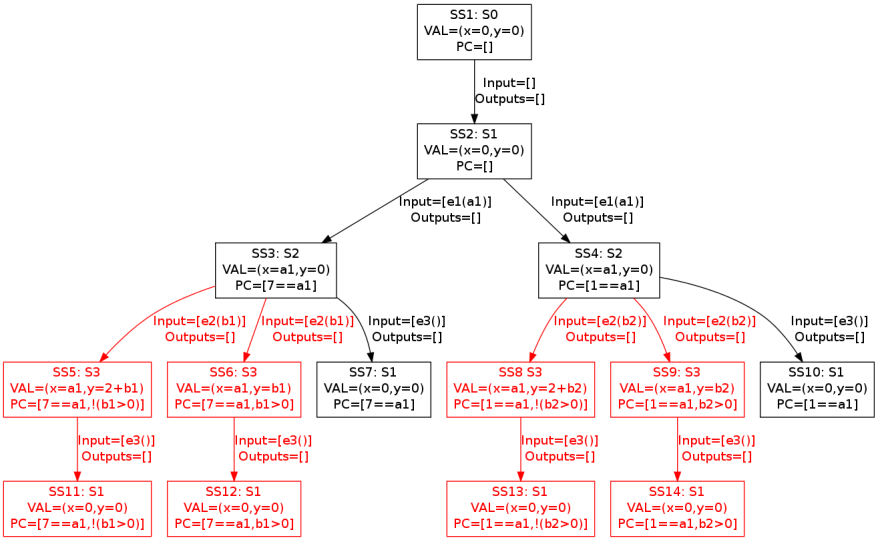
the $SET(V0)$. However, by looking at the highlighted parts (i.e., those which are coloured red) of the $SET(V1)$, we find that: 1) they are slightly different from their corresponding parts in the $SET(V0)$; 2) they represent the symbolic execution of transition *T2* (the changed transition in *V1*) and subsequent transitions; and 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.

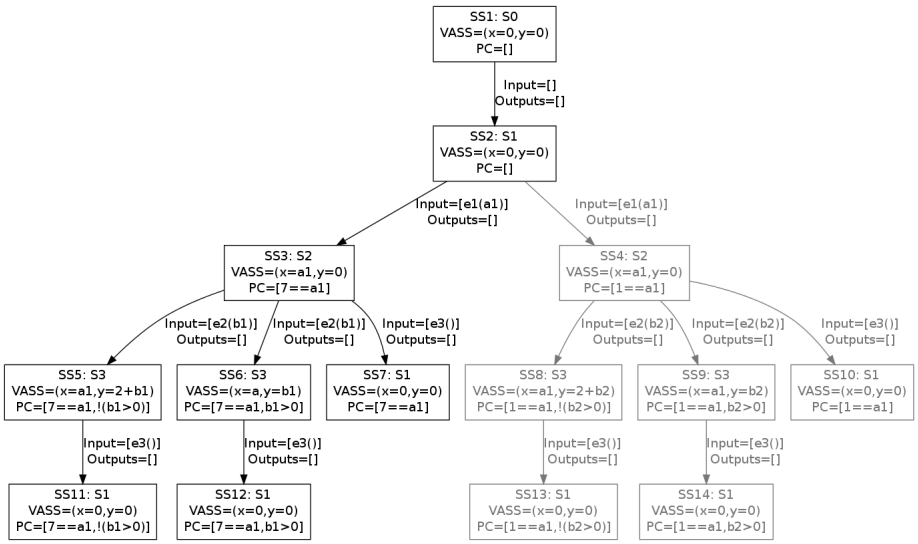
The applicability and utility of running this process depends, first, on the existence of the $SET(V0)$ and, second, on the amount of savings gained from reusing it which can be very large for big complex models that undergo some minor changes (i.e., changes with limited impact).

Possible measures for quantifying the amount of savings are: 1) the time gained from not generating the $SET(V1)$ from scratch; 2) the perecentage of nodes (or symbolic states) in the $SET(V0)$ that can be safely reused; or 3) the perecentage of execution paths in the $SET(V0)$ that can also be safely reused. A very important benefit from this latter measure is that it provides 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 the closer the change is to the initial state of the model, the less the number of nodes in the $SET(S0)$ that can be



(a) Highlighted (in red) the parts that are different from *SET(V0)*



(b) Grayed out the pruned parts resulting from the partial exploration of transitions: *T1*, *T3*, and *T4*

Figure 7: *SET(V1)* - Symbolic Execution Tree for the Statechart in Figure 5b

reused. Additionally, the more model paths a changed state or transition can reach, 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 to update all the descendents of the symbolic state *SS2* in the $SET(V0)$, leaving only two symbolic states to be reused.

4.2 Optimization Via Reduction

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 3.2, the symbolic execution of a transition, t_i, in an MLM model may result in zero, one or many symbolic transitions of t_i 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 t_i; 2) the satisfiability of the guard condition of t_i; and 3) the number of execution paths of the action code of t_i. For example, the symbolic execution of transition *T1* in the two versions of the model in Figure 5 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 *S3* (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*). Similarily, 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 *SS5*, *SS6*, *SS7* and *SS8* and ending at the symbolic states *SS11*, *SS12*, *SS13* and *SS14*, respectively. In the same way, the symbolic execution of transition *T4* results in having one symbolic transition originating from each of the symbolic states *SS3* and *SS4* and ending at the symbolic states *SS7* and *SS10*, respectively.

Third, as we also discuss in Section 2.4, 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 minimum 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 *V1* given that transition *T2* has been changed with respect to its base version, *V0*, shows that there is no dependency between *T2* and any other transitions as none of them defines a variable that transition *T2* uses (and vice versa). Now, with this information in mind, we can direct the symbolic execution of *V1* to "fully" explore the changed transition *T2* (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 any one of its symbolic transitions). Following this process to symbolically execute *V1* results in the SET shown in Figure 7b, which has 6 symbolic states less than the complete $SET(V1)$. Although the new SET is not complete, it is sufficient to run regression types of analysis. For instance, the SET in Figure 7b 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 multi-path guards or action code, which both results 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.

4.3 Reuse Or Reduction?

From our brief discussion of the idea behind each technique, we summarize the following:

- For the first technique, we need to have access to the *SET* of the original model *V0* to be reused and we need to identify the differences between the new version of the model and the original one. The effectiveness of this technique is highly dependent on the change location. The resulting *SET* represents all execution paths of *V1* and can thus be used to, e.g., maintain the test suite for the model.

- For the second technique, 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. Models with complex guards

and action codes will benefit most from this technique. As the resulting *SETs* may not be complete and omit executions that are guaranteed to not have been impacted by the change, they 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 serve the same purpose but with different requirements, and it is the role of the analyst to choose among them. For instance, the applicability of the results from previous analysis is a basic requirement for applying the first technique, 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 the second technique.

5 Optimizing Symbolic Execution of Evolving State Machines

5.1 Memoized-based Symbolic Execution (A Reuse Approach)

In this section we present our memoized-based symbolic execution technique (MSE) as shown in Figure 8.

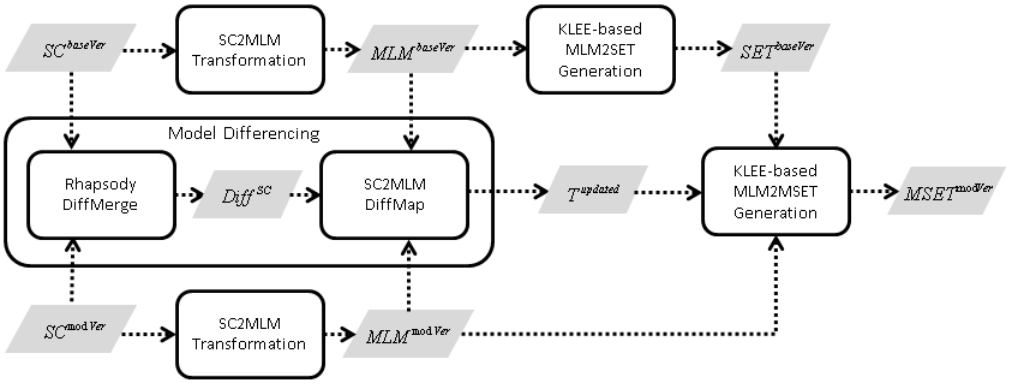


Figure 8: Memoized-based Symbolic Execution (MSE)

The inputs to MSE are: 1) an MLM representation of a modified version of a Statechart model, MLM^{modVer}; 2) the symbolic execution tree of the base version of the model, $SET^{baseVer}$; and 3) the differences between the modified version of a Statechart model and its base version mapped to a set of updated transitions in their corresponding MLM representations. Currently, we consider only the following

types of changes: adding/removing states; adding/removing transitions; updating the actions of states; and updating transitions. All these types of changes can be represented as changes to transitions in our MLM representation. For example, adding/removing states can be represented by an addition/deletion of transitions connecting these states with other states in the model; also updating the entry/exit actions of states can be represented by an update of their incoming/outgoing transitions, respectively. Therefore, we manage to represent these differences in terms of a set of updated transitions $T^{updated}$ including all transitions that have been added to, deleted from or updated in the modified version of the model. To find the differences between two Rhapsody Statecharts models, we use the Rhapsody DiffMerge tool that we mentioned in Section 2.3. These differences are mapped to their MLM correspondences using our "SC2MLM DiffMap" module, which outputs the set of all transitions that are found to have been updated $T^{updated}$ in the modified MLM version.

The output from MSE is a new symbolic execution tree SET^{modVer} that shares all what can be reused from the old tree 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.

The detailed steps of our MSE module are shown in Algorithm 2. Below, we explain each of these steps.

The first step in this algorithm (Lines 1-15) is to load and explore the symbolic execution tree to be reused (i.e., the SET of the base version of the Statechart Model) in order to remove all the edges representing any updated transitions (note that removing an edge leads to removing the entire subtree rooted at the target node of the edge).

The second step is to re-explore the symbolic execution tree, resulting from the previous step, to find all symbolic states representing states with outgoing transitions belonging to $T^{updated}$ (Lines 17-24). For each of these symbolic states, we symbolically execute MLM^{modVer} using an adapted version of Algorithm 1 (Line 26) in which we symbolically execute a given model based on some parameters. These parameters determine where the exploration starts (i.e., the symbolic state to begin the exploration at), which updated transitions to consider 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 purpose of subsumption checking. The resulting symbolic execution trees are then merged with $SET^{baseVer}$ (Line 27).

Algorithm 2 Memoized-based Symbolic Execution (MSE) Algorithm

Input: MLM=(S, V, E, EA, T, s_0, V_0), $SET^{baseVer}$=(SS, L, ST, ss_0), $T^{updated}$
Output: A Symbolic Execution Tree SET^{modVer}=(SS', L', ST', ss_0)

1: //Explore the symbolic execution tree of the based version, $SET^{baseVer}$, in breadth-first-search fashion to remove the parts that need to be updated; symbolic states in the resulting tree are to be used for the purpose of subsumption checking
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: **for all** outgoing symbolic transitions st=(ss, l, ss') of ss **do**
8: Find t in T such that: ($sourceState(t)$=ss)$\wedge$($targetState(t)$=ss')
9: **if** $t \in T^{updated}$ **then**
10: Remove l and ss' with all its children from $SET^{baseVer}$
11: **else**
12: $q \leftarrow enqueue(q, ss')$
13: **end if**
14: **end for**
15: **end while**
16: //Re-explore $SET^{baseVer}$ to merge the results from the symbolic execution of MLM^{modVer} for the parts that need to be updated
17: $q \leftarrow [ss_0]$
18: **while** q is not empty **do**
19: Remove the first element ss=(s, val, pc) from q
20: $T^{toConsider} \leftarrow \{\}$
21: **if** s has one or more outgoing transitions in $T^{updated}$ **then**
22: **for all** ($t' \in OutTrans(s)$) $\wedge$ ($t' \in T^{updated}$) **do**
23: $T^{toConsider} \leftarrow T^{toConsider} \cup \{t'\}$
24: **end for**
25: //Given the set of symbolic states to be reused $SS_{baseVer}$, symbolically execute MLM^{modVer} starting at state s and considering only the outgoing transitions of s that belong to $T^{toConsider}$ when we explore s for the first time
26: $SET_{sub} \leftarrow SymbolicExecution(MLM^{modVer}, ss, T^{toConsider}, SS_{baseVer})$
27: Merge the root of SET_{sub} with the symbolic state ss of $SET^{baseVer}$
28: **else**
29: **for all** outgoing symbolic transitions st'=(ss, l', ss') of ss **do**
30: $q \leftarrow$enqueue(q, ss')
31: **end for**
32: **end if**
33: **end while**
34: $SET^{modVer} \leftarrow SET^{baseVer}$

5.2 Dependency-based Symbolic Execution (A Reduction Approach)

A high level overview of the main tasks of our dependency-based symbolic execution technique (DSE)[2] is depicted in Figure 9. In the following, we describe each task in detail.

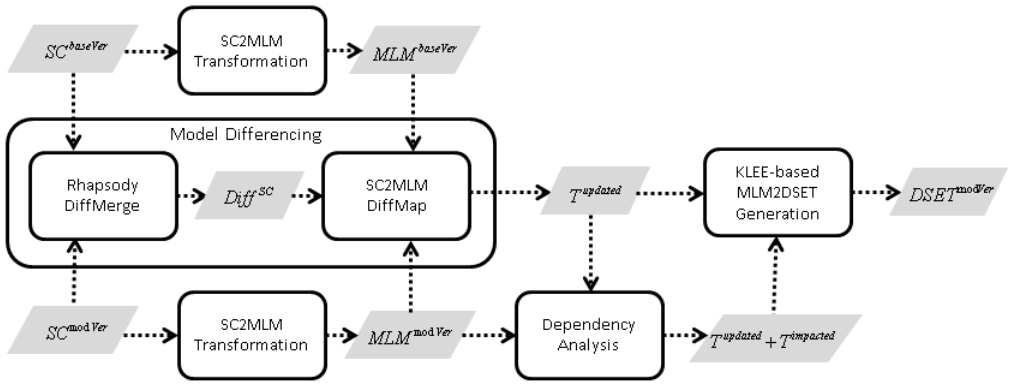


Figure 9: Dependency-based Symbolic Execution (DSE)

The first task is to identify the differences between the two versions of a Statechart model and to map these differences into a set of updated transitions in their corresponding MLM representations. We follow the same procedure as in the previous section to manage this task.

The second task is to perform a dependency analysis on the modified version of the model to compute the list of transitions that have an impact or are impacted by any of the updated transitions set resulting from the previous task. In this module, we implement the algorithms that compute the types of dependences discussed in Section 2.4. The output from this task is a list of all transitions that are found updated or impacted in any way. These transitions are the ones that we need to fully explore when we symbolically execute the modified version of the model.

The third and the last task 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 analysis task. The detailed steps of our DSE module are shown in Algorithm 3. Two modes of exploration are defined: full and partial. A full exploration mode requires a complete exploration of all execution paths of an explored transition and is applied for all transitions that are found to have been updated or impacted (Lines 15-26). However, a partial exploration mode requires

[2]Note that we can also call it regression or partial symbolic execution.

the execution of only one representative path of an exploring transition and is applied to transitions that are found neither updated nor impacted (Lines 27-38).

6 Implementation and Evaluation

6.1 IMPLEMENTATION

We implemented our symbolic execution techniques as two Eclipse plug-ins; one of them runs on Windows and the other runs in an Ubuntu Virtual Machine. Both the translation from Rhapsody Statecharts to Mealy-like Machines and the computation of control and data dependency of these Statecharts are implemented in the context of the Rhapsody RulesComposer Add-On. Identifying the differences between different versions of Rhapsody Statecharts is carried out using Rhapsody DiffMerge, however we needed to map these differences into their corresponding elements in the MLM representation. Identifying the impact of the changes in these versions is done based on the outputs from the two previous modules, which are all implemented under Windows. Because we use the KLEE symbolic execution engine to symbolically execute action code in our Statecharts, we developed our symbolic execution techniques in Ubuntu, where it was easier to run KLEE. We set up a file-sharing mechanism to access the Mealy-like Machines, the differences report, and the change impact report, which are all created and saved under the Windows environment, from the Ubuntu VM (where both KLEE and our symbolic execution implementation work). In order to use KLEE for the symbolic execution of the fragments of action code encountered in Rhapsody Statecharts, we only consider a subset of Java/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.

6.2 EVALUATION

6.2.1 Research Questions and Variables of Interest

We consider the following four research questions:

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

Algorithm 3 Dependency-based Symbolic Execution (DSE) Algorithm

Input: MLM^{modVer}=(S, V, E, EA, T, s_0, V_0), $T^{ofInterest}$=$T^{updated} \cup T^{impacted}$
Output: A Symbolic Execution Tree SET=(SS, L, ST, ss_0)

1: $ss_0 \leftarrow (s_0, V_0, \text{true})$
2: $SS \leftarrow SS \cup \{ss_0\}$
3: Create a queue q
4: $q \leftarrow [ss_0]$
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 V^s of new variables in one-to-one relation with the transition event's arguments eA
10: Create a map σ between eA and V^s
11: 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 $A^s \leftarrow \sigma(A)$
12: **if** G^s is satisfiable given val and pc **then**
13: $feasiblePaths \leftarrow SymbolicExecutionOfCode(A^s, val, G^s, pc)$
14: **if** $t \in T^{ofInterest}$ **then**
15: **for all** $feasiblePath$=(val', pc', out')$\in feasiblePaths$ **do**
16: Create a new symbolic state, ss'=(s', val', pc')
17: **if** ss' is not subsumed by any previously generated state **then**
18: $SS \leftarrow SS \cup \{ss'\}$
19: Create a new label, l'=(e, V^s, σ, out')
20: $L \leftarrow L \cup \{l'\}$
21: Create a new symbolic transition, st'=(ss, l', ss')
22: $ST \leftarrow ST \cup \{st'\}$
23: $q \leftarrow$enqueue(q, ss')
24: **end if**
25: **end for**
26: **else**
27: Select only one $feasiblePath$=(val', pc', out')$\in feasiblePaths$
28: Create a new symbolic state, ss'=(s', val', pc')
29: **if** ss' is not subsumed by any previously generated state **then**
30: $SS \leftarrow SS \cup \{ss'\}$
31: Create a new label, l'=(e, V^s, σ, out')
32: $L \leftarrow L \cup \{l'\}$
33: Create a new symbolic transition, st'=(ss, l', ss')
34: $ST \leftarrow ST \cup \{st'\}$
35: $q \leftarrow$enqueue(q, ss')
36: **end if**
37: **end if**
38: **end if**
39: **end for**
40: **end while**

– **RQ2.** Does the change location have an impact on the effectiveness of the MSE technique?

– **RQ3.** Does the SET generated from MSE match the one generated from standard SE?

– **RQ4.** Can we identify state machine characteristics that gurantee the effectiveness of the DSE technique?

For the study of **RQ1**, we considered the following three correlated variables: 1) the time taken to run each technique, 2) the number of newly computed/created symbolic states and 3) the number of execution paths in the resulting SETs.

For the study of **RQ2**, we defined two independent measures in order to quantify the impact of a change in one or more transitions within an MLM model. The two measures have been computed as a result of performing the dependency analysis, discussed in Section 2.4, on the MLM representations of the models under study (i.e., before running our optimization techniques). The first measure, M1, is a value indicating the percentage of the maximal paths in an MLM model that do not go through (or include) any of the changed transitions; the lower this value is, the higher the number of maximal paths involving the change is and the higher the impact of the change on the given model. The second measure, M2, is a normalized value between zero and one-hundred indicating the average position of a changed transition within its maximal paths; the lower this value, the closer the changed transition to the beginning of these paths is and the higher the impact of this changed transition on all successive transitions sharing the same path.

For the study of **RQ3**, we first calculated/recorded the differences in the number of symbolic states and the number of execution paths generated in the SETs resulting from standard symbolic execution and memoized-based symbolic execution. Then, we performed manual inspection of a subset of the two SETs to ensure their equivalence.

For the study of **RQ4**, we choose the following two factors: 1) the first and most influential factor is the number of transitions in the subject model with multiple symbolic transitions (i.e., those with multi-path guards or action code) - the higher this value is, the more effective the dependency-based symbolic execution technique is, and 2) the second factor is the number of transitions that are found in 1) and have data dependencies with each other or with some other transitions in the model - the higher this value is, the less the opportunity for savings from the partial exploration is and the less effective the dependency-based symbolic execution technique is. This is because if the change occurs in a transition that depends on or is dependent on some other transitions then our dependency-based symbolic execution will have to fully explore the changed transition and all its dependences.

6.2.2 Artifacts

To evaluate the effectiveness of our optimization techniques, 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 [8]. The LGS is an automotive feature used to assist drivers in avoiding 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 therefore we manually converted them to Rhapsody Statecharts. Table 1 summarizes the characteristics of the three models and their equivalent MLM representations.

6.2.3 Evaluation Setup

To perform our evaluation, 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 adding a send event statement to the action code of the subject transition. The main reason for selecting those specific types of changes is to keep the number of modified models manageable in such a way that enables us to ensure the correctness of our implementation and to analyze the evaluation results. 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 recorded the following information related to the change that was made: 1) the number of transitions in the MLM representation of the subject model corresponding to the changed transition in Rhapsody, 2) the number of transitions in the MLM representation of the subject model that have been found to impact or be impacted by a changed transition and 3) the values for the two measures, M1 and M2, computed for the study of RQ2.

Second, we ran our standard symbolic execution (SE) on all the versions of the three selected artifacts, while we ran both the memoized-based symbolic execution (MSE) and the dependency-based symbolic execution (DSE) only on the modified versions, given the results of the SE of their base version in the case of MSE. For each run, we recorded the time to generate the symbolic execution tree, and the number of symbolic states and execution paths in the generated SET. Note that we only considered the symbolic states and execution paths that were newly created when we recorded the results for the MSE runs. To measure the effectiveness of

Table 1: Characteristics of the Artifacts Used in our Evaluation

(a) Air Quality System (AQS)

	Rhapsody Statechart (before flattening)	MLM (after flattening)
# Concurrent regions	1	1
# Hierarchical levels	3	1
# States	3 CS + 14 SS	14 SS
# Transitions	8 GT + 22 ST	55 ST
# Transitions with multi-path guards or action code	5 ST	5 ST
# Transitions with dependencies with each other	3 ST	3 ST

(b) Lane Guide System (LGS)

	Rhapsody Statechart (before flattening)	MLM (after flattening)
# Concurrent regions	1	1
# Hierarchical levels	3	1
# States	2 CS + 10 SS	10 SS
# Transitions	6 GT + 16 ST	40 ST
# Transitions with multi-path guards or action code	2 GT	10
# Transitions with dependencies with each other	0	0

(c) Adaptive Cruise Control System (ACCS)

	Rhapsody Statechart (before flattening)		MLM (after flattening)
# Concurrent regions	2		1
	Region 1	Region 2	1
# Hierarchical levels	3	1	1
# States	2 CS + 6 SS	3 SS	19 SS
# Transitions	9 GT + 5 ST	9 ST	73 ST
# Transitions with multi-path guards or action code	1 ST	0	1 ST
# Transitions with dependencies with each other	2 GT + 2 ST	4 ST	16 ST

SS=Simple State, CS=Composite State, ST=Simple Transition, GT=Group Transition

Table 2: Characteristics of the SETs generated from standard SE on the base versions of the three used examples

SET Characteristics	Air Quality System (AQS)	Lane Guide System (LGS)	Adaptive Cruise Control System (ACCS)
Time (sec)	748	626	796
No. Symbolic States	6039	1769	3492
No. Execution Paths	5019	1325	2855

MSE and DSE compared to standard SE, we computed the ratios between the values of the three aforementioned variables recorded for the standard SE and their correspondences in MSE and DSE. Moreover, we calculated, for each MSE run, the differences between the total numbers of symbolic states and execution paths of the SET generated from MSE and the one generated from standard SE for the study of RQ3. We performed this evaluation 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. Table 2 shows the time taken to perform standard SE on the base versions and the numbers of symbolic states and execution paths in the resulting SETs. Tables 3, 4 and 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.

6.2.4 Results and Analysis

In this section, we discuss the results presented in Tables 3, 4 and 5, according to our four research questions.

RQ1. *How effective are our optimizations compared to a standard SE of the changed model?* In Figures 10a and 10b, we show a line graph of the average and the standard deviation values of the data shown in columns 6-7, 9 (resp. 11-13) in Tables 3, 4 and 5, corresponding to the amount of savings in execution times and in the numbers of symbolic states and execution paths gained from applying MSE (resp. DSE) on all the versions of the three given models. Figure 10a shows that, on average, MSE achieved savings of 76% for the ASQ model, 66% for the LGS model, and 48% for the ACCS model, while DSE achieved savings of 96% for the ASQ model, 68% for the LGS model, and only 5.5% for the ACCS model. Also, Figure 10b shows that the standard deviation values of the achieved savings ratios for the three models are higher for MSE than they are for DSE.

RQ2. *Does the change location have an impact on the effectiveness of the MSE technique?* In Figure 11, we show the Pearson's correlation coefficients[3] that we

[3]Pearson's correlation coefficient is a measure with values between +1 (for total positive correlation) and -1 (for total negative correlation) inclusive of the degree of linear correlation

Table 3: Results of MSE and DSE on the AQS Example

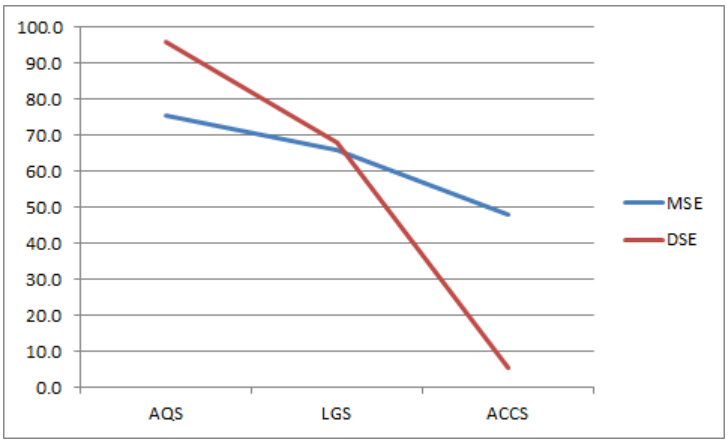
Ver.	Change Information				SE : MSE					SE : DSE		
	MLM											
	No. of Changed Transitions	No. of Impacted Transitions	M1 [0, 100]	M2 [0, 100]	Time	No. of New Symbolic States	No. of Total Symbolic States	No. of New Execution Paths	No. of Total Execution Paths	Time	No. of Total Symbolic States	No. of Total Execution Paths
V1	7	0	70	100	1 : 0.26	1 : 0.08	1 : 1	1 : 0.08	1 : 1	1 : 0.02	1 : 0.03	1 : 0.02
V2	1	0	96	100	1 : 0.43	1 : 0.36	1 : 1	1 : 0.36	1 : 1	1 : 0.02	1 : 0.03	1 : 0.02
V3	1	0	0	22	1 : 1.01	1 : 1	1 : 1	1 : 1	1 : 1	1 : 0.02	1 : 0.03	1 : 0.02
V4	1	0	80	43	1 : 0.16	1 : 0.23	1 : 1	1 : 0.23	1 : 1	1 : 0.03	1 : 0.04	1 : 0.03
V5	1	2	70	41	1 : 0.03	1 : 0.01	1 : 1	1 : 0.01	1 : 1	1 : 0.12	1 : 0.14	1 : 0.13
V6	1	2	10	32	1 : 0.53	1 : 0.64	1 : 1	1 : 0.66	1 : 1	1 : 0.12	1 : 0.14	1 : 0.13
V7	1	2	100	100	1 : 0.06	1 : 0.06	1 : 1	1 : 0.06	1 : 1	1 : 0.12	1 : 0.14	1 : 0.13
V8	1	0	70	41	1 : 0.16	1 : 0.23	1 : 1	1 : 0.23	1 : 1	1 : 0.02	1 : 0.03	1 : 0.02
V9	1	0	100	100	1 : 0.02	1 : 0	1 : 1	1 : 0	1 : 1	1 : 0.02	1 : 0.03	1 : 0.02
V10	1	0	60	48	1 : 0.17	1 : 0.23	1 : 1	1 : 0.23	1 : 1	1 : 0.03	1 : 0.04	1 : 0.03
V11	1	0	70	54	1 : 0.04	1 : 0.06	1 : 1	1 : 0.06	1 : 1	1 : 0.02	1 : 0.03	1 : 0.02
V12	1	0	99	100	1 : 0.02	1 : 0.02	1 : 1	1 : 0.02	1 : 1	1 : 0.02	1 : 0.03	1 : 0.02
V13	5	0	3	56	1 : 0.66	1 : 0.87	1 : 1	1 : 0.91	1 : 1	1 : 0.02	1 : 0.03	1 : 0.02
V14	7	0	70	100	1 : 0.39	1 : 0.39	1 : 1	1 : 0.43	1 : 1	1 : 0.02	1 : 0.03	1 : 0.02
V15	1	0	6	66	1 : 0.47	1 : 0.61	1 : 1	1 : 0.64	1 : 1	1 : 0.1	1 : 0.17	1 : 0.16
V16	1	0	86	100	1 : 0.01	1 : 0.01	1 : 1	1 : 0.02	1 : 1	1 : 0.02	1 : 0.03	1 : 0.02
V17	5	0	86	100	1 : 0.11	1 : 0.15	1 : 1	1 : 0.17	1 : 1	1 : 0.02	1 : 0.03	1 : 0.02
V18	5	0	58	90	1 : 0.04	1 : 0.06	1 : 1	1 : 0.06	1 : 1	1 : 0.02	1 : 0.03	1 : 0.02
V19	1	0	65	75	1 : 0.01	1 : 0.01	1 : 1	1 : 0.01	1 : 1	1 : 0.02	1 : 0.03	1 : 0.02
V20	1	0	52	74	1 : 0.18	1 : 0.41	1 : 1	1 : 0.42	1 : 1	1 : 0.02	1 : 0.03	1 : 0.02
V21	1	0	73	83	1 : 0.06	1 : 0.01	1 : 1	1 : 0.01	1 : 1	1 : 0.02	1 : 0.03	1 : 0.02
V22	1	0	98	100	1 : 0.06	1 : 0	1 : 1	1 : 0	1 : 1	1 : 0.02	1 : 0.03	1 : 0.02
V23	1	0	65	80	1 : 0.29	1 : 0.41	1 : 1	1 : 0.41	1 : 1	1 : 0.02	1 : 0.03	1 : 0.02
V24	1	0	86	88	1 : 0.09	1 : 0.03	1 : 1	1 : 0.03	1 : 1	1 : 0.02	1 : 0.03	1 : 0.02
V25	1	0	75	87	1 : 0.37	1 : 0.4	1 : 1	1 : 0.4	1 : 1	1 : 0.02	1 : 0.03	1 : 0.02
V26	5	0	98	100	1 : 0.43	1 : 0.09	1 : 1	1 : 0.08	1 : 1	1 : 0.02	1 : 0.03	1 : 0.02
Average Savings					76.6 %	75.5 %		74.9 %		96.4 %	95.6 %	96.1 %
Standard Deviation					24.6 %	28.0%		28.8 %		3.4 %	4.4 %	4.2 %

Table 4: Results of MSE and DSE on the LGS Example

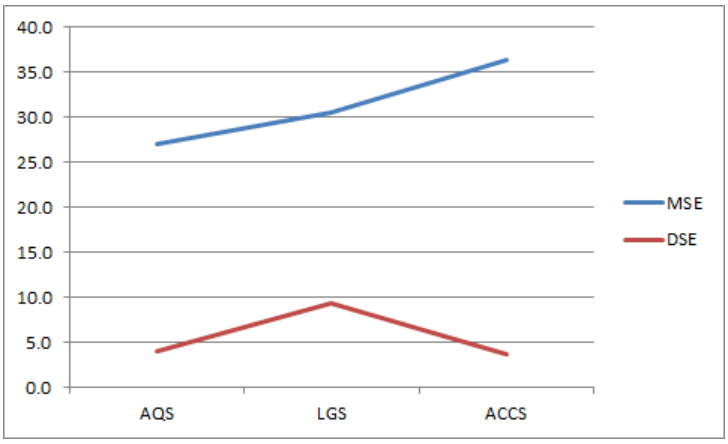
Ver.	Change Information				SE : MSE					SE : DSE		
	MLM											
	No. of Changed Transitions	No. of Impacted Transitions	M1 [0, 100]	M2 [0, 100]	Time	No. of New Symbolic States	No. of Total Symbolic States	No. of New Execution Paths	No. of Total Execution Paths	Time	No. of Total Symbolic States	No. of Total Execution Paths
V1	1	0	0	36	1 : 1.05	1 : 1	1 : 1	1 : 1	1 : 1	1 : 0.31	1 : 0.30	1 : 0.40
V2	7	0	72	100	1 : 0.44	1 : 0.25	1 : 1	1 : 0.33	1 : 1	1 : 0.28	1 : 0.30	1 : 0.40
V3	7	0	72	100	1 : 0.55	1 : 0.25	1 : 1	1 : 0.33	1 : 1	1 : 0.26	1 : 0.30	1 : 0.40
V4	1	0	3	53	1 : 0.97	1 : 1	1 : 1	1 : 1	1 : 1	1 : 0.26	1 : 0.30	1 : 0.40
V5	4	0	87	100	1 : 0.49	1 : 0.5	1 : 1	1 : 0.5	1 : 1	1 : 0.38	1 : 0.47	1 : 0.62
V6	1	0	87	100	1 : 0.64	1 : 0.58	1 : 1	1 : 0.59	1 : 1	1 : 0.26	1 : 0.30	1 : 0.40
V7	5	0	60	84	1 : 0.63	1 : 0.74	1 : 1	1 : 0.74	1 : 1	1 : 0.57	1 : 0.73	1 : 0.98
V8	1	0	78	68	1 : 0.44	1 : 0.27	1 : 1	1 : 0.27	1 : 1	1 : 0.26	1 : 0.30	1 : 0.40
V9	1	0	78	68	1 : 0.43	1 : 0.27	1 : 1	1 : 0.27	1 : 1	1 : 0.26	1 : 0.30	1 : 0.40
V10	1	0	78	68	1 : 0.4	1 : 0.27	1 : 1	1 : 0.27	1 : 1	1 : 0.29	1 : 0.30	1 : 0.40
V11	1	0	78	68	1 : 0.41	1 : 0.27	1 : 1	1 : 0.27	1 : 1	1 : 0.28	1 : 0.30	1 : 0.40
V12	1	0	97	100	1 : 0.04	1 : 0.04	1 : 1	1 : 0.04	1 : 1	1 : 0.31	1 : 0.30	1 : 0.40
V13	1	0	87	81	1 : 0.12	1 : 0.12	1 : 1	1 : 0.12	1 : 1	1 : 0.28	1 : 0.30	1 : 0.40
V14	1	0	97	100	1 : 0.04	1 : 0.04	1 : 1	1 : 0.04	1 : 1	1 : 0.28	1 : 0.30	1 : 0.40
V15	1	0	87	81	1 : 0.11	1 : 0.12	1 : 1	1 : 0.12	1 : 1	1 : 0.28	1 : 0.30	1 : 0.40
V16	1	0	87	81	1 : 0.15	1 : 0.12	1 : 1	1 : 0.12	1 : 1	1 : 0.28	1 : 0.30	1 : 0.40
V17	1	0	97	100	1 : 0.05	1 : 0.04	1 : 1	1 : 0.04	1 : 1	1 : 0.27	1 : 0.30	1 : 0.40
V18	1	0	87	81	1 : 0.14	1 : 0.12	1 : 1	1 : 0.12	1 : 1	1 : 0.27	1 : 0.30	1 : 0.40
V19	1	0	97	100	1 : 0.05	1 : 0.04	1 : 1	1 : 0.04	1 : 1	1 : 0.27	1 : 0.30	1 : 0.40
Average Savings					62.2 %	68.2 %		67.2 %		70.2 %	67.2 %	66.6 %
Standard Deviation					30.6 %	30.6 %		30.5 %		7.3 %	10.5 %	10.4 %

Table 5: Results of MSE and DSE on the ACCS Example

Ver.	Change Information				SE : MSE					SE : DSE		
	MLM											
	No. of Changed Transitions	No. of Impacted Transitions	M1 [0, 100]	M2 [0, 100]	Time	No. of New Symbolic States	No. of Total Symbolic States	No. of New Execution Paths	No. of Total Execution Paths	Time	No. of Total Symbolic States	No. of Total Execution Paths
V1	1	0	0	26	1 : 1.02	1 : 1.00	1 : 1	1 : 1.00	1 : 1	1 : 0.97	1 : 0.92	1 : 0.90
V2	12	0	70	93	1 : 0.42	1 : 0.17	1 : 1	1 : 0.17	1 : 1	1 : 1.02	1 : 0.92	1 : 0.90
V3	12	0	70	93	1 : 0.44	1 : 0.15	1 : 1	1 : 0.17	1 : 1	1 : 1.01	1 : 0.92	1 : 0.90
V4	2	4	98	100	1 : 0.78	1 : 0.93	1 : 1.15	1 : 0.95	1 : 1.15	1 : 1.00	1 : 0.92	1 : 0.90
V5	2	4	98	100	1 : 0.50	1 : 0.62	1 : 1.18	1 : 0.64	1 : 1.18	1 : 1.00	1 : 0.92	1 : 0.90
V6	2	4	40	57	1 : 0.79	1 : 0.87	1 : 1	1 : 0.87	1 : 1	1 : 1.01	1 : 0.92	1 : 0.90
V7	2	4	40	57	1 : 0.45	1 : 0.57	1 : 1.02	1 : 0.58	1 : 1.02	1 : 0.92	1 : 0.92	1 : 0.90
V8	2	0	80	68	1 : 1.06	1 : 1.13	1 : 1.38	1 : 1.17	1 : 1.4	1 : 0.91	1 : 0.92	1 : 0.90
V9	2	0	80	68	1 : 0.26	1 : 0.18	1 : 1	1 : 0.20	1 : 1	1 : 0.97	1 : 0.92	1 : 0.90
V10	4	0	64	95	1 : 0.04	1 : 0.05	1 : 1	1 : 0.04	1 : 1	1 : 0.99	1 : 0.92	1 : 0.90
V11	4	0	64	95	1 : 0.19	1 : 0.21	1 : 1.01	1 : 0.21	1 : 1.01	1 : 1.02	1 : 0.92	1 : 0.90
V12	1	13	1	39	1 : 1.03	1 : 1.00	1 : 1	1 : 1.00	1 : 1	1 : 0.96	1 : 1.00	1 : 1.00
V13	6	0	86	82	1 : 0.42	1 : 0.22	1 : 1	1 : 0.19	1 : 1	1 : 1.00	1 : 0.92	1 : 0.90
V14	1	13	89	100	1 : 0.59	1 : 0.67	1 : 1	1 : 0.68	1 : 1	1 : 1.01	1 : 1.00	1 : 1.00
V15	1	0	89	100	1 : 0.01	1 : 0.01	1 : 1	1 : 0.01	1 : 1	1 : 1.00	1 : 0.92	1 : 0.90
V16	3	2	17	55	1 : 0.80	1 : 0.87	1 : 1	1 : 0.88	1 : 1	1 : 1.07	1 : 0.92	1 : 0.90
V17	3	2	70	75	1 : 0.37	1 : 0.17	1 : 1	1 : 0.20	1 : 1	1 : 0.91	1 : 1.00	1 : 1.00
V18	6	0	69	78	1 : 0.73	1 : 0.72	1 : 1	1 : 0.73	1 : 1	1 : 1.04	1 : 0.92	1 : 0.90
V19	6	0	69	78	1 : 0.34	1 : 0.09	1 : 1	1 : 0.12	1 : 1	1 : 1.03	1 : 0.92	1 : 0.90
Average Savings					46.1 %	49.4 %		48.5 %		0.9 %	7.0 %	8.6 %
Standard Deviation					31.8 %	38.5 %		38.9 %		4.3 %	3.1 %	3.8 %



(a) Average of Savings



(b) Standard Deviation of Savings

Figure 10: Statistical Measures of the Effectiveness of MSE and DSE for the Three Models

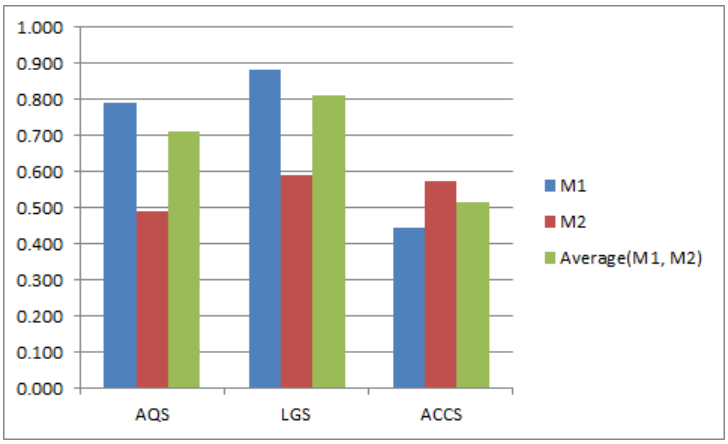


Figure 11: Pearson's Correlation Coefficients between the Average Savings Gained from MSE and the Estimated Change Impact Measures: M1 and M2

have caluclated to measure the correlation between our estimated change impact measures, M1 and M2, and the different savings gained from running MSE on the three given models as recorded in Tables 3, 4 and 5. From this graph, we can see that there is high positive correlation between the different savings gained from MSE and M1 for AQS and LGS, however it is slightly low for ACCS. Moreover, we can see that there is a medium positive correlation between the different savings gained from MSE and M2 for the three models. Our explanation to these observations is due to the discrepancy between the number of maximal paths in a model, which we used to define the two measures, M1 and M2, and the actual number of execution paths in the SET of that model. Another reason is due to the optimization gained from the subsumption checking, which might prune away some of the execution paths related to some transitions. This also lead to some discrepancy between the number of maximal paths in a model and its corresponding ones in the SET of that model.

RQ3. *Does the SET generated from MSE match the one generated from standard SE?* The results show that MSE generated the exact SETs as standard SE for all the versions of the AQS and the LGS model, and 14/19 versions of the ACCS model. However, for the versions V4, V5, V7, V8 and V11 of the ACCS model, we found that MSE generated larger SETs than the ones generated from standard SE. By investagating the reason for this difference, we found that some symbolic states have been explored further although there were some pre-visited symbolic states that can subsume them. This is due to the way the MSE algorithm works where updating the nodes is performed in a sequential manner, starting from the higher-level nodes and proceeding toward the lower-level nodes, which reduces the

between two variables. We use the sample Pearson correlation coefficient formula, r, that has been implemented in the Microsoft Excel function *CORREL*.

chance for the sub-trees of the higher-level nodes to benefit from the subsumption checking savings that could be gained from the sub-trees of the lower-level nodes. This means that, in some cases, the SETs resulting from MSE are less optimized than they could be if they were generated from running standard SE on the same models. In such a case, iterating over the SET resulting from MSE and removing the children of symbolic states that could be subsumed by some other symbolic states that are higher in the tree will result in a SET that exactly matches the one generated from standard SE. We developped this latter procedure and added it to our MSE implementation as an optional step to be performed only if needed.

RQ4. *How are the characteristics of the model related to the effectiveness of the DSE technique?* According to the two factors considered and based on the data shown in Table 1 and Tables 3, 4 and 5, we can see the following: 1) the number of transitions with multiple symbolic transitions (i.e., with multi-path guards or actions) is higher in both the AQS and the LGS models than it is in the ACCS model, 2) the LGS model has the least dependency between its transitions, while ACCS has the largest dependency between its transitions, and 3) there is high savings in execution time, in number of symbolic states, and in number of execution paths gained from applying DSE on the AQS and the LGS models in contrast with the ACCS model. Based on these findings, we can confidently conclude that the two factors considered have direct influence on the effectiveness of DSE.

Therefore, we conclude that our optimization techniques are complementary in the sense that the effectiveness of MSE depends mostly on the impact 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 Related Work

Existing approaches to alleviate the scalability problem and improve the efficiency of symbolic execution when re-applying on an evolving version of a program are discussed in [27] and [20].

In [27], Yang et. al. present memoized symbolic execution of source code; a technique that stores the results of symbolic execution from a previous run and reuses them as a starting point for the next run of the technique to avoid the re-execution of common paths between the new version of a program and its previous one, and to speed up the symbolic execution for the new version. The same idea has been applied earlier by Lauterburg et. al. in [17] but in the context of state-space exploration for evolving programs using some model checkers.

In [20], Person et. al. introduce DiSE (directed incremental symbolic execution); a technique that 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 those impacted locations. A similar approach has been proposed by Yang et. al. in [26] for regression model checking.

In contrast to all the aforementioned approaches, which work for optimizing the analysis of evolving programs, our work targets the same objective but for evolving state machine models.

In his statement paper "Evolution, Adaptation, and the Quest for Incrementality", Ghezzi [11] 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 rerun after the change. The motivation for this is twofold: time efficiency and scalability. Given the iterative development approach suggested by MDD, we believe that our work fits very well with this vision.

8 Conclusion

In this paper, we presented two different techniques for optimizing the symbolic execution of evolving state machine models. 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. The results from our experiments look promising and show a significant amount of savings up to 99 % in some cases with respect to the size of the symbolic execution trees generated from applying either technique and the time taken to generate them. We plan to extend our work to consider the symbolic execution of a collection of state machine models describing the behavior of intercommunicating objects and how to optimize this in the context of undergoing changes.

Acknowlegment

This work was partially funded by NSERC (Canada), as part of the NECSIS Automotive Partnership with General Motors, IBM Canada and Malina Software Corp.

References

[1] Saswat Anand. *Techniques to facilitate symbolic execution of real-world programs*. PhD thesis, Georgia Institute of Technology, 2012.

[2] Kelly Androutsopoulos, David Clark, Mark Harman, Zheng Li, and Laurence Tratt. Control dependence for extended finite state machines. In *Fundamental Approaches to Software Engineering*, pages 216–230. Springer, 2009.

[3] Michael Balser, Simon Bäumler, Alexander Knapp, Wolfgang Reif, and Andreas Thums. Interactive verification of uml state machines. In *Formal methods and software engineering*, pages 434–448. Springer, 2004.

[4] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[5] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[6] Yanping Chen, Robert L Probert, and Hasan Ural. Model-based regression test suite generation using dependence analysis. In *Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 54–62. ACM, 2007.

[7] Valentin Chimisliu and Franz Wotawa. Improving test case generation from uml statecharts by using control, data and communication dependencies. In *13th International Conference on Quality Software (QSIC)*, pages 125–134. IEEE, 2013.

[8] Alma L Juarez Dominguez. *Detection of feature interactions in automotive active safety features*. PhD thesis, University of Waterloo, 2012.

[9] Elizabeta Fourneret and Fabrice Bouquet. Impact analysis for uml/ocl statechart diagrams based on dependence algorithms for evolving critical software. *Laboratoire d'Informatique de Franche-Comté, Besançon, France, Tech. Rep. RT2010-06*, 2010.

[10] Christophe Gaston, Pascale Le Gall, Nicolas Rapin, and Assia Touil. Symbolic execution techniques for test purpose definition. In *Testing of Communicating Systems*, pages 1–18. Springer, 2006.

[11] Carlo Ghezzi. Evolution, adaptation, and the quest for incrementality. In *Large-Scale Complex IT Systems. Development, Operation and Management*, pages 369–379. Springer, 2012.

[12] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[13] E Jobstl, Martin Weiglhofer, Bernhard K Aichernig, and Franz Wotawa. When BDDs fail: Conformance testing with symbolic execution and smt solving. In *Third International Conference on Software Testing, Verification and Validation (ICST)*, pages 479–488. IEEE, 2010.

[14] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[15] Bogdan Korel, Luay Ho Tahat, and Boris Vaysburg. Model based regression test reduction using dependence analysis. In *Proceedings of the International Conference on Software Maintenance*, pages 214–223. IEEE, 2002.

[16] Sébastien Labbé and Jean-Pierre Gallois. Slicing communicating automata specifications: polynomial algorithms for model reduction. *Formal Aspects of Computing*, 20(6):563–595, 2008.

[17] Steven Lauterburg, Ahmed Sobeih, Darko Marinov, and Mahesh Viswanathan. Incremental state-space exploration for programs with dynamically allocated data. In *Proceedings of the 30th international conference on Software engineering*, pages 291–300. ACM, 2008.

[18] Kumar Madhukar, Ravindra Metta, Priyanka Singh, and R Venkatesh. Reachability verification of rhapsody statecharts. In *IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 96–101. IEEE, 2013.

[19] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A manifesto for semantic model differencing. In *Models in Software Engineering*, pages 194–203. Springer, 2011.

[20] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *ACM SIGPLAN Notices*, volume 46, pages 504–515. ACM, 2011.

[21] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, John Hatcliff, and Matthew B Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5):27, 2007.

[22] IBM Rational. Rational Rhapsody Developer, http://www-03.ibm.com/software/products/en/ratirhap/.

[23] Matthew Stephan and James R Cordy. A survey of model comparison approaches and applications. In *Modelsward*, pages 265–277, 2013.

[24] Luay Tahat, Bogdan Korel, Mark Harman, and Hasan Ural. Regression test suite prioritization using system models. *Software Testing, Verification and Reliability*, 22(7):481–506, 2012.

[25] Andreas Thums, Gerhard Schellhorn, Frank Ortmeier, and Wolfgang Reif. Interactive verification of statecharts. In *Integration of Software Specification Techniques for Applications in Engineering*, pages 355–373. Springer, 2004.

[26] Guowei Yang, Matthew B Dwyer, and Gregg Rothermel. Regression model checking. In *IEEE International Conference on Software Maintenance (ICSM 2009)*, pages 115–124. IEEE, 2009.

[27] Guowei Yang, Corina S Păsăreanu, and Sarfraz Khurshid. Memoized symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 144–154. ACM, 2012.

[28] Karolina Zurowska and Juergen Dingel. Symbolic execution of uml-rt state machines. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1292–1299. ACM, 2012.