Incremental Symbolic Execution of Evolving State Machines

Amal Khalil
School of Computing
Queen's University
Kingston, Ontario
Email: khalil@cs.queensu.ca

Juergen Dingel
School of Computing
Queen's University
Kingston, Ontario
Email: dingel@cs.queensu.ca

***Abstract*—This paper introduces two complementary techniques, memoization-based and dependency-based incremental symbolic execution, that aim to optimize the analysis of state machine models that undergo change. We implement the two proposed techniques on IBM Rhapsody Statecharts and present some evaluation results.**

I. Introduction

Iterative-incremental development and model analysis are central to Model Driven Engineering (MDE) in which artifacts typically 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.

Rhapsody Statecharts (also known as Harel's Statecharts [1]) are a visual state-based formalism implemented in the IBM Rational Rhapsody framework [2] (one of the commercial tools that supports model-driven development) to describe the behavior of reactive systems. They extend Mealy machines with state hierarchy, concurrency and entry/exit actions.

Symbolic execution is a well-known analysis technique that is used to analyze the execution paths of behavioral software artifacts (e.g., programs [3] and state-based models [4], [5]). The output of the execution 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.

Inspired by the research work for optimizing the symbolic execution of evolving programs [6], [7], we propose two different optimization techniques that leverage the similarities/differences between state machine versions to reduce the cost of symbolic execution of the evolved version. The first technique, called *memoization-based symbolic execution (MSE)*, reuses the SET generated from a previous analysis and incrementally updates the relevant parts corresponding to the changes made to the new version. The second technique integrates a change impact analysis (i.e., dependency analysis) technique with the symbolic execution to implement what we call *dependency-based symbolic execution (DSE)*[1]. The dependency analysis allows us to identify the unaffected parts of an evolved artifact; 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.

We have implemented these two techniques and applied them to three industrial-size state machines. The initial results demonstrate the effectiveness of MSE and DSE for reducing the cost of performing symbolic execution on evolving state machines.

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 to 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 to generate 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 II introduces our approach to symbolically executing Rhapsody Statecharts; Section III motivates the ideas of reuse and reduce for optimizing the symbolic execution of evolving Rhapsody Statecharts; Section IV provides an overview of our proposed techniques; Section V presents evaluation results; Section VI covers the related work and Section VII concludes this paper.

II. Symbolic Execution of Rhapsody Statecharts

Our symbolic execution approach to symbolically executing Rhapsody Statecharts is based on that of Zurowska and Dingel [5] with some variations. In contrast to their work, our intermediate machine representation of Rhapsody Statecharts takes the form of Mealy-like Machines (MLMs) instead of

[1]We could also call it regression or partial symbolic execution.

their Functional Finite State Machines (FFSMs). Additionally, we use the off-the-shelf symbolic execution engine KLEE [8] to execute action code encountered in the Statecharts, whereas an in-house symbolic execution engine was used in [5]. 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 standard SE technique are the ones numbered 1 and 2 in Fig. 1, which are: 1) SC2MLM, a transformation that transforms a Statechart model into our MLM representation and 2) MLM2SET, our standard symbolic execution module that traverses an MLM model and symbolically executes the action code encountered in each transition to build the model's symbolic state space.

A. SC2MLM

The basic structure of our MLM formalism 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 MLMs do not have entry/exit actions. Transitions are characterized in the same way as in Rhapsody Statecharts by an event that triggers 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/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 the MLMs formalism. Our current transformation supports the mapping of these 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 by 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 beginning of the action code of its outgoing transitions.

Formally, a *Mealy-like Machine* is a tuple *MLM*=(*S*, *V*, *E*, *EA*, *T*, s_0, val_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* (also called triggers or signals) by which the machine communicates with its environment. Events can have arguments *EA* (also called parameters) and are partitioned into *input events* E^i, *output events* E^o, and *internal events* E^{int};
- *EA* is a finite set of *event arguments* 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's input/internal events in E^i 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 it 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 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*;
- val_0 is the *initial valuation* of the variables in *V*.

B. MLM2SET

The main idea is the same as the symbolic execution of programs with some variations that reflect the different features of MLMs. 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 an 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 an 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 a 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 each location. 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 output 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 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".

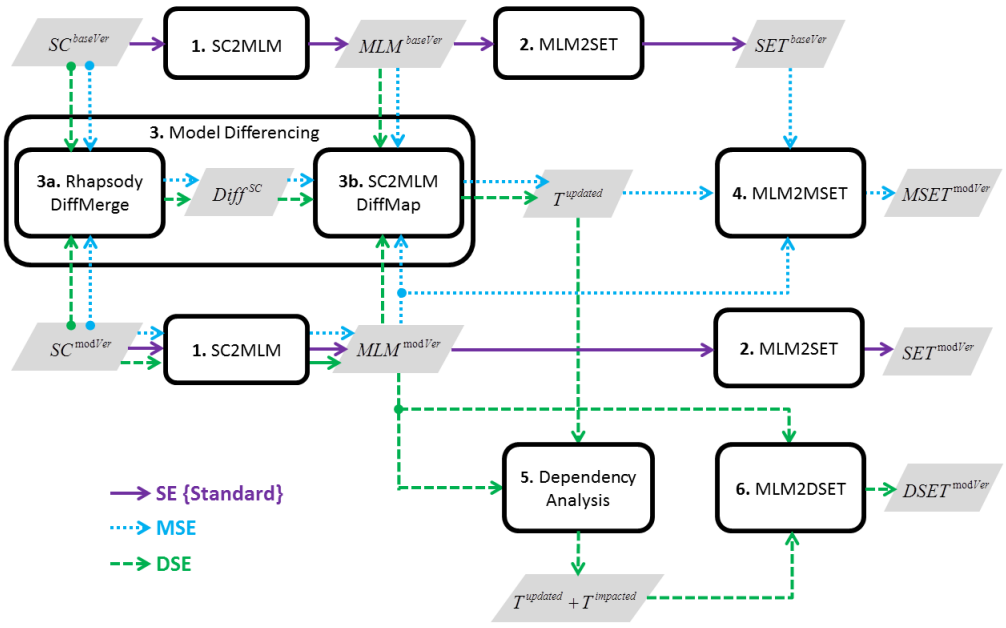


Fig. 1. Symbolic Execution of Rhapsody Statecharts and its Optimizations (SE, MSE and DSE)

Formally, a *symbolic execution tree* of a Mealy-Like-Machine *MLM*=(*S*, *V*, *E*, *EA*, *T*, s_0, val_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 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*, disjoint from *V* and *EA*;
  - σ is a map from the original event argument names $eA \subset EA$ to the new set of variable 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.

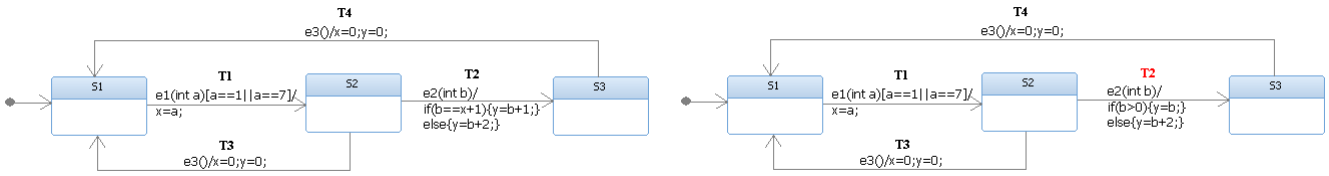
In the remainder of this section, we provide the details of the two interleaved steps to symbolically execute an MLM model. The algorithm implementing these steps can be found in the full version of this paper [9].

The first step 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 two criteria that we consider here are loop bounds and state matching. For the latter, if a symbolic state is *subsumed by* some other previously explored symbolic state, then we do not explore it any further. A symbolic state *ss*=(*s*, *val*, *pc*) is subsumed by another symbolic state *ss'*=(*s'*, *val'*, *pc'*) if *s*=*s'*, *val*=*val'*, and the set of states satisfying *pc* is included in the set of states satisfying *pc'* (i.e., *pc* ⇒*pc'* or *pc* is more constrained than *pc'*).

The second step is the symbolic execution of transitions in the MLM model and the construction of the SET. For this step, we perform the following tasks for each transition: 1) we create and assign a unique set of symbolic variables to replace the event arguments of the transition, if any; 2) we substitute the occurrences of these event arguments in the guard expression and the action code statements by their assigned symbolic values; 3) we check if the guard is satisfiable with respect to the variable updates and the path constraint of the current symbolic state; if so, 4) we use KLEE to symbolically 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 the action code; 5) we use the results from the previous task to create a new set of symbolic states representing the target state of this transition, to label the edges to these new symbolic states and to build the SET.

III. Motivating Example

In this section, we use the model versions presented in Fig. 2 and their SETs in Fig. 3 and 4 to demonstrate how our optimization techniques work. The base version of the example



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

(b) V1 - A modified version of V0

Fig. 2. Two Versions of an Example Statechart

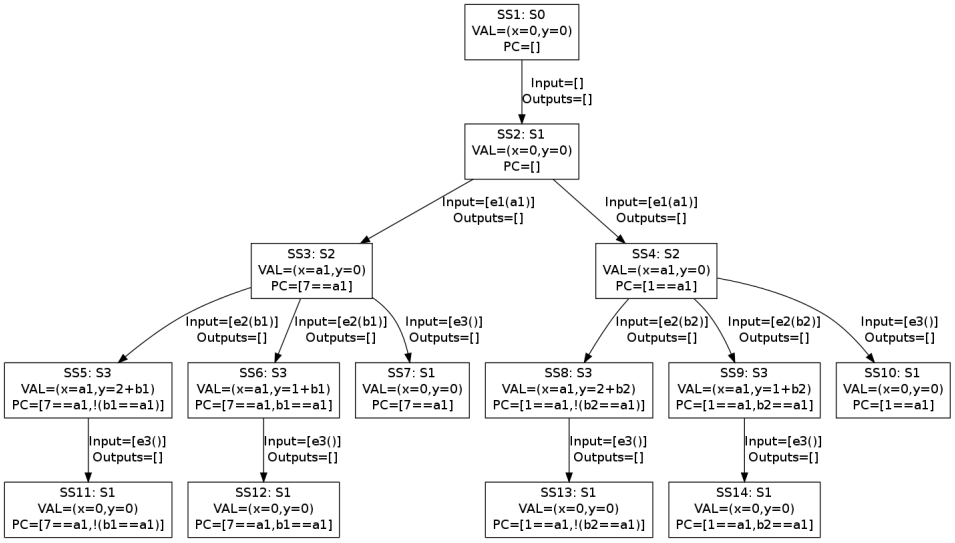


Fig. 3. SET(V0) - SET for the Statechart in Fig. 2a

model is V0 and its modified version is V1 that updates the action code of transition T2.

A. Optimization Via Reuse

By comparing the unhighlighted parts (coloured black) of SET(V1) in Fig. 4a with the SET(V0) in Fig. 3, we notice that these parts also appear in SET(V0). However, by looking at the highlighted parts (coloured red) of SET(V1), we find that: 1) they are slightly different from their corresponding parts in SET(V0); 2) they represent the symbolic execution of transition T2 (the changed transition in V1) and its subsequent transitions; and 3) they reflect the parts of SET(V0) that are impacted by the changes made on T2.

Therefore, if we already have 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 updated and impacted parts to replace the old ones.

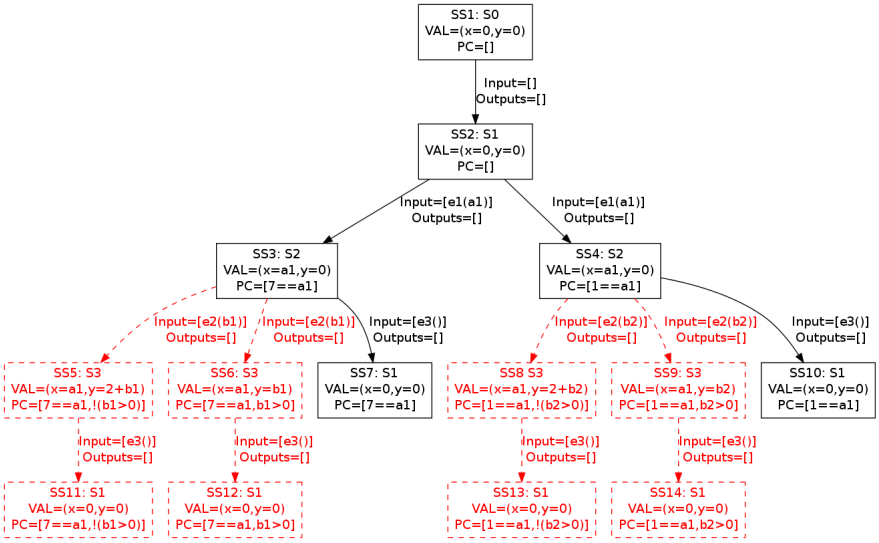
The applicability and utility of running this process depends, first, on the existence of the SET(V0) and, second, on the amount of savings gained from reusing it which can be very large for big complex models that undergo some minor change (i.e., changes with limited impact).

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 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.

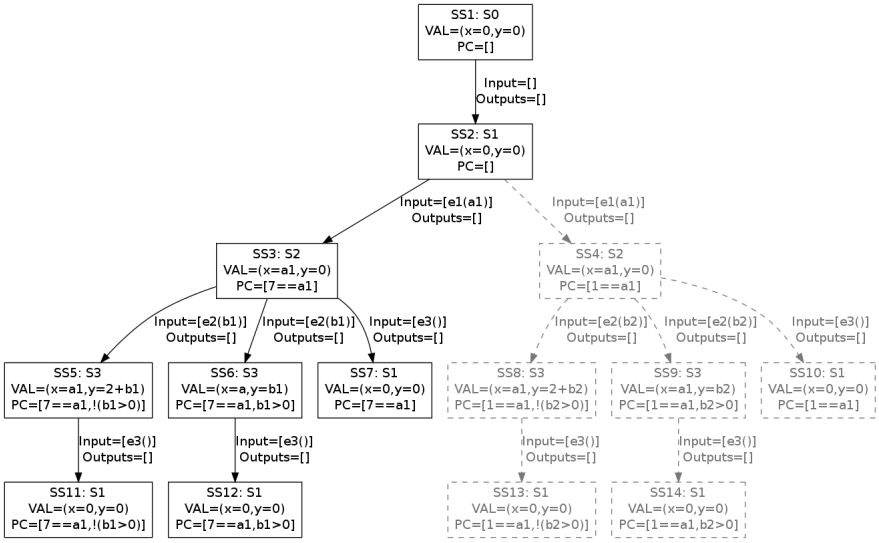
We notice that the location of the change in the model may suggest the effectiveness of reusing the SET(V0). The closer the change is to the initial state of the model, the smaller the number of nodes in the SET(V0) can be reused. Also, the more paths a changed state/transition is involved in, the more updates of the SET(V0) are needed and the smaller the effectiveness of reusing it. 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. Therefore, in our evaluation we try to measure this correlation.

B. Optimization Via Reduction

In order to perform this type of optimization, we assume that the base version of the model (i.e., the version before the change) has already been analyzed successfully (e.g., to



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



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

Fig. 4. SET(V1) - SET for the Statechart in Fig. 2b

determine reachability of states or generate test cases). Based on this assumption, we will be able to reduce the symbolic execution of the unaffected parts of the modified version of the model to a minimum without affecting the analysis quality.

As we discussed in Section II-B, the symbolic execution of a transition in an MLM may result in zero, one or many symbolic transitions in the resulting SET, depending on: 1) the path constraints and the variable valuation of the symbolic state representing the source state of the transition; 2) the satisfiability of the guard condition of the transition; and 3) the number of execution paths of the action code of the transition. For example, the symbolic execution of transition T1 in the two versions of the model in Fig. 2 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). 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 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 one symbolic transition originating from each of the symbolic states SS3

and SS4 and ending at the symbolic states SS7 and SS10, respectively.

Applying a dependency analysis on the modified model version V1 given that transition T2 has been changed with respect to its base version V0 shows that there is no data dependency between T2 and any other transition as none of them defines a variable that transition T2 uses (and vice versa). Based on this information, 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 only one of its symbolic transitions). Following this process to symbolically execute V1 results in the SET shown in Fig. 4b, 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 Fig. 4b 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 that can be pruned from the partial exploration of unimpacted transitions, this technique appears most beneficial if applied to Statecharts that have transitions with multi-path guards or multi-path action code, 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 only partially explored.

IV. Optimizing Symbolic Execution of Evolving State Machines

In this section we provide high level details of the main components implementing our two optimization techniques: memoization-based symbolic execution (MSE) and dependency-based symbolic execution (DSE). Full details can be found in [9]. Our current implementation supports the following types of changes: adding/removing states, adding/removing transitions, updating the entry/exit action of states and updating the action of 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 action of states can be represented by an update to 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.

A. MSE - A Reuse Approach

The main component of our MSE technique is the "MLM2MSET" component as depicted in Fig. 1. The three inputs to this component are: 1) the MLM representation of the modified version of the model MLM^{modVer}, 2) the set of transitions that have been updated in the modified version of the model $T^{updated}$, and 3) the SET to be reused (i.e., $SET^{baseVer}$). Two successive tasks are performed by this component. The first task is to load and explore the input $SET^{baseVer}$ in order to remove all the edges representing any transition belonging to the set of updated transitions $T^{updated}$ (note that removing an edge leads to removing the entire subtree rooted at the target node of the edge). The second task is to re-explore the SET resulting from the previous task to find all symbolic states representing states with outgoing transitions belonging to $T^{updated}$. For each of these symbolic states, we symbolically execute MLM^{modVer} 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 so far to be used for the subsumption checking. The resulting SETs are then merged with the SET resulting from the previous task. The output from the component is a memoization-based SET (MSET) of the modified version of the model $MSET^{modVer}$ that shares all what can be reused from the old tree $SET^{baseVer}$ but also contains the modifications resulting from the SE of the parts of the new version of the model that are found changed.

The computation of the set of updated transitions in the modified version of the model $T^{updated}$ is carried out by the "Model Differencing" component, which also consists of two successive tasks. The first task is to find the differences between the two Rhapsody Statecharts SC^{modVer} and $SC^{baseVer}$ and this is carried out using the Rhapsody DiffMerge tool. The second task is to map these differences into their MLM correspondences and this is performed using our "SC2MLM DiffMap" component. The output from this latter component is a set of all transitions that are found to have been updated $T^{updated}$ in a modified MLM version (with respect to its base version).

B. DSE - A Reduction Approach

The main component of our DSE technique is the "MLM2DSET" component as depicted in Fig. 1. The two inputs to this component are: 1) the MLM representation of the modified version of the model MLM^{modVer} and 2) the set of transitions that are found to have been updated or impacted in the modified version of the model $T^{updated} + T^{impacted}$. The main task performed by this component is similar to the standard SE technique except that the state-space exploration of the input model MLM^{modVer} is guided by the input set of updated and impacted transitions $T^{updated} + T^{impacted}$. The term $T^{impacted}$ defines the set of transitions that have an impact or are impacted by any of the updated transitions in $T^{updated}$ (i.e., the set of transitions that have a dependency with any of the updated transitions in $T^{updated}$).

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. However, a partial exploration mode requires the execution of only one representative path of an explored transition and is applied to transitions that are found neither updated nor impacted. The output from the component is a dependency-based SET (DSET) of the modified version of the model $DSET^{modVer}$ that can be smaller/equal in size to the standard SET of the same model, depending on the amount of savings gained from the partial exploration of the input list of updated/impacted transitions.

The terms $T^{updated}$ and $T^{impacted}$ are computed by components "Model Differencing" and "Dependency Analysis" (Fig. 1), respectively. The two inputs to the "Dependency Analysis" component are: 1) the MLM representation of the modified version of the model MLM^{modVer} and 2) the set of transitions that are found to have been updated in the modified version of the model $T^{updated}$. In this component, we first explore the input MLM model to compute all the dependencies that exist between its transitions, then identify the transitions that have a dependency with any of the input updated transitions. The output is the union of the input updated transitions set and their dependencies. We developed the algorithms implementing the definitions presented in [10], [11] for computing the dependencies in conventional Extended Finite State Machines (EFSMs), which can also be applied to our MLMs. These algorithms compute the maximal paths of the directed graph representing the subject model (EFSM or MLM), and then identify the set of transitions that share the same path and that define/use the same variable(s). A path in an EFSM or in an MLM model is called *maximal* if it terminates in an end state (i.e., state with no outgoing transitions) or it terminates in a cycle.

V. Evaluation

In this section we evaluate the effectiveness of using MSE and DSE for an evolved Rhapsody Statechart model compared with standard SE. To perform our evaluation, we implemented the architecture presented in Fig. 1 as a set of Eclipse plug-ins and performed a case study on three models. Since KLEE supports only C code, we consider only a subset of Java/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 types including characters, Booleans and integers. We also consider the statements used in the action code for sending events.

A. 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 state machine model?
- **RQ2.** Does the SET generated from MSE match the one generated from standard SE?
- **RQ3.** What aspects influence the effectiveness of each technique?

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. Due to space limitations, we only show the results of the first two variables. Complete results can be found in [9].

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.

For RQ3, we consider the following two aspects: 1) *change impact* and 2) *model characteristics*. To measure the impact of the change, 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. For model characteristics, we identify the following the two features 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 and 2) the number of transitions that have a dependency with other transitions in the model. We speculate that the first aspect has more influence on the effectiveness of MSE, whereas the second aspect impacts the effectiveness of DSE.

B. Case Study 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 [12]. 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 I summarizes the characteristics of the three models and their MLM representations, including the two features related to RQ3. The models contain 4, 7, and 3 variables of type integer, respectively.

C. Evaluation Setup

To perform our study, 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.

TABLE I
CHARACTERISTICS OF THE ARTIFACTS USED IN OUR EVALUATION

Numbers of	Air Quality System (AQS)		Lane Guide System (LGS)		Adaptive Cruise Control System (ACCS)		
	Rhapsody Statechart (before flattening)	MLM (after flattening)	Rhapsody Statechart (before flattening)	MLM (after flattening)	Rhapsody Statechart (before flattening)		MLM (after flattening)
Concurrent regions	1	1	1	1	2		1
Hierarchical levels	3	1	3	1	3	1	1
States	3 CS + 14 SS	14 SS	2 CS + 10 SS	10 SS	2 CS + 6 SS	3 SS	19 SS
Transitions	8 GT + 22 ST	55 ST	6 GT + 16 ST	40 ST	9 GT + 5 ST	9 ST	73 ST
Transitions with multi-path guards or action code	5 ST	5 ST	2 GT	10 ST	1 ST	0	1 ST
Transitions with dependencies with each other	3 ST	3 ST	0	0	2 GT + 2 ST	4 ST	16 ST

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

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 these specific types of changes is to keep the number of modified models manageable and to facilitate the manual correctness check of the 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 first recorded the following information: 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 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 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 and the numbers of symbolic states 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.

Table II shows the time taken to perform standard SE on the base versions and the number of symbolic states and execution paths in the resulting SETs.

[2]We use the sample Pearson correlation coefficient formula implemented in the Microsoft Excel function *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.

TABLE II
PERFORMANCE OF STANDARD SE ON BASE VERSIONS

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

Tables III and IV summarize the results of running the three symbolic execution techniques on the modified versions of the AQS and the ACCS models, respectively. The results for the LGS model can be found in [9]; they are omitted due to the space limit and because they were in line with what observed from the results of the AQS model.

In columns 5-6 (resp. 8-9) of Tables III and IV, we show the savings ratios in time and in the number of symbolic states gained from applying MSE (resp. DSE) on the modified versions of the three given models compared to standard SE. We also show, for each such column, the overall average, the standard deviation and the correlation coefficient with the CIM metric data in column 4. The ratios between the total number of symbolic states found in the resulting SETs generated from standard SE and MSE are shown in column 7.

D. Results and Analysis

RQ1. *How effective are our optimizations compared to standard SE?* Based on the values presented in columns 5-6 (resp. 8-9) of Tables III and IV, we can see that MSE achieved an average savings of 76.6% in time (resp. 75.5% in the number of symbolic states) for the AQS model and 46.1% in time (resp. 49.4% in the number of symbolic states) for the ACCS model. Moreover, DSE achieved an average savings of 96.4% in time (resp. 95.6% in the number of symbolic states) for the AQS model and only 0.9% in time (resp. 7.0% in the number of symbolic states) for the ACCS model. We also notice that the standard deviation values of the achieved savings ratios are higher for MSE than they are for DSE, which means that the effectiveness of MSE is more influenced by the changes made in each modified model version than it is for the DSE.

RQ2. *Does the SET generated from MSE match the one generated from standard SE?* As suggested by the ratios

TABLE III
RESULTS OF MSE AND DSE ON THE AQS EXAMPLE

Ver.	Change Information MLM: No. of Changed Transitions	Change Information MLM: No. of Impacted Transitions	Change Information MLM: CIM [0, 100]	SE : MSE: Time	SE : MSE: No. of New Symbolic States	SE : MSE: No. of Total Symbolic States	SE : DSE: Time	SE : DSE: No. of Total Symbolic States
V1	7	0	30	1 : 0.26	1 : 0.08	1 : 1	1 : 0.02	1 : 0.03
V2	1	0	4	1 : 0.43	1 : 0.36	1 : 1	1 : 0.02	1 : 0.03
V3	1	0	100	1 : 1.01	1 : 1	1 : 1	1 : 0.02	1 : 0.03
V4	1	0	20	1 : 0.16	1 : 0.23	1 : 1	1 : 0.03	1 : 0.04
V5	1	2	30	1 : 0.03	1 : 0.01	1 : 1	1 : 0.12	1 : 0.14
V6	1	2	90	1 : 0.53	1 : 0.64	1 : 1	1 : 0.12	1 : 0.14
V7	1	2	0	1 : 0.06	1 : 0.06	1 : 1	1 : 0.12	1 : 0.14
V8	1	0	30	1 : 0.16	1 : 0.23	1 : 1	1 : 0.02	1 : 0.03
V9	1	0	0	1 : 0.02	1 : 0	1 : 1	1 : 0.02	1 : 0.03
V10	1	0	40	1 : 0.17	1 : 0.23	1 : 1	1 : 0.03	1 : 0.04
V11	1	0	30	1 : 0.04	1 : 0.06	1 : 1	1 : 0.02	1 : 0.03
V12	1	0	1	1 : 0.02	1 : 0.02	1 : 1	1 : 0.02	1 : 0.03
V13	5	0	97	1 : 0.66	1 : 0.87	1 : 1	1 : 0.02	1 : 0.03
V14	7	0	30	1 : 0.39	1 : 0.39	1 : 1	1 : 0.02	1 : 0.03
V15	1	0	94	1 : 0.47	1 : 0.61	1 : 1	1 : 0.10	1 : 0.17
V16	1	0	14	1 : 0.01	1 : 0.01	1 : 1	1 : 0.02	1 : 0.03
V17	5	0	14	1 : 0.11	1 : 0.15	1 : 1	1 : 0.02	1 : 0.03
V18	5	0	42	1 : 0.04	1 : 0.06	1 : 1	1 : 0.02	1 : 0.03
V19	1	0	35	1 : 0.01	1 : 0.01	1 : 1	1 : 0.02	1 : 0.03
V20	1	0	48	1 : 0.18	1 : 0.41	1 : 1	1 : 0.02	1 : 0.03
V21	1	0	27	1 : 0.06	1 : 0.01	1 : 1	1 : 0.02	1 : 0.03
V22	1	0	2	1 : 0.06	1 : 0	1 : 1	1 : 0.02	1 : 0.03
V23	1	0	35	1 : 0.29	1 : 0.41	1 : 1	1 : 0.02	1 : 0.03
V24	1	0	14	1 : 0.09	1 : 0.03	1 : 1	1 : 0.02	1 : 0.03
V25	1	0	25	1 : 0.37	1 : 0.40	1 : 1	1 : 0.02	1 : 0.03
V26	5	0	2	1 : 0.43	1 : 0.09	1 : 1	1 : 0.02	1 : 0.03
Average Savings				76.6 %	75.5 %		96.4 %	95.6 %
Standard Deviation				24.6 %	28.0%		3.4 %	4.4 %
CORREL(CIM)				-0.70	-0.83		-0.27	-0.33

TABLE IV
RESULTS OF MSE AND DSE ON ACCS EXAMPLE

Ver.	Change Information MLM: No. of Changed Transitions	Change Information MLM: No. of Impacted Transitions	Change Information MLM: CIM [0, 100]	SE : MSE: Time	SE : MSE: No. of New Symbolic States	SE : MSE: No. of Total Symbolic States	SE : DSE: Time	SE : DSE: No. of Total Symbolic States
V1	1	0	100	1 : 1.02	1 : 1.00	1 : 1	1 : 0.97	1 : 0.92
V2	12	0	30	1 : 0.42	1 : 0.17	1 : 1	1 : 1.02	1 : 0.92
V3	12	0	30	1 : 0.44	1 : 0.15	1 : 1	1 : 1.01	1 : 0.92
V4	2	4	2	1 : 0.78	1 : 0.93	1 : 1.15	1 : 1.00	1 : 0.92
V5	2	4	2	1 : 0.50	1 : 0.62	1 : 1.18	1 : 1.00	1 : 0.92
V6	2	4	60	1 : 0.79	1 : 0.87	1 : 1	1 : 1.01	1 : 0.92
V7	2	4	60	1 : 0.45	1 : 0.57	1 : 1.02	1 : 0.92	1 : 0.92
V8	2	0	20	1 : 1.06	1 : 1.13	1 : 1.38	1 : 0.91	1 : 0.92
V9	2	0	20	1 : 0.26	1 : 0.18	1 : 1	1 : 0.97	1 : 0.92
V10	4	0	36	1 : 0.04	1 : 0.05	1 : 1	1 : 0.99	1 : 0.92
V11	4	0	36	1 : 0.19	1 : 0.21	1 : 1.01	1 : 1.02	1 : 0.92
V12	1	13	99	1 : 1.03	1 : 1.00	1 : 1	1 : 0.96	1 : 1.00
V13	6	0	14	1 : 0.42	1 : 0.22	1 : 1	1 : 1.00	1 : 0.92
V14	1	13	11	1 : 0.59	1 : 0.67	1 : 1	1 : 1.01	1 : 1.00
V15	1	0	11	1 : 0.01	1 : 0.01	1 : 1	1 : 1.00	1 : 0.92
V16	3	2	83	1 : 0.80	1 : 0.87	1 : 1	1 : 1.07	1 : 0.92
V17	3	2	30	1 : 0.37	1 : 0.17	1 : 1	1 : 0.91	1 : 1.00
V18	6	0	31	1 : 0.73	1 : 0.72	1 : 1	1 : 1.04	1 : 0.92
V19	6	0	31	1 : 0.34	1 : 0.09	1 : 1	1 : 1.03	1 : 0.92
Average Savings				46.1 %	49.4 %		0.9 %	7.0 %
Standard Deviation				31.8 %	38.5 %		4.3 %	3.1 %
CORREL(CIM)				-0.49	-0.43		0.01	-0.14

recorded in column 7 of Tables III and IV and our manual inspection for the generated SETs, we found that MSE generated the exact SETs as standard SE for all the versions of the AQS and the LGS models, and 14 /19 versions of the ACCS model. However, for the versions V4, V5, V7, V8 and V11 of the ACCS model, MSE generated larger SETs. This is because MSE updates the affected nodes of the reused SET sequentially, which prevents the detection of existing subsumption relationships between nodes that were updated first and nodes that are updated later. Iterating over the SET resulting from MSE and rerunning the subsumption checking will result in a SET that exactly matches the one generated from standard SE, which means that this limitation only affects the efficiency of the implementation but not its soundness.

RQ3. *What 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 aspect: According to the correlation coefficients computed between the values of columns 5-6 (resp. 8-9) and the values of column 4 of Tables III and IV, 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. 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. Meanwhile, we noticed almost no significant correlation between the savings gained from DSE and the CIM metric.

(2) For the model characteristics aspect: We could not find a clear relationship between the two defined features and the savings gained from MSE since the differences between the savings gained among the three used models are not as significant as they are for the DSE. For example, the differences between the average savings gained from DSE for the AQS model and the ACCS model were 95.5 % in time (resp. 88.6 % in the number of symbolic states). By looking at the model characteristics in Table I, we notice that the number of transitions with multi-path guards or action code is higher in both the AQS and the LGS models than it is in the ACCS model. This increased the opportunity for savings from the partial exploration implemented in DSE for the AQS and the LGS models compared to the ACCS model. We can also notice that the LGS model has the least dependency between its transitions, while ACCS has the largest dependency between its transitions. This also decreased the opportunity for savings from the partial exploration implemented in DSE for the ACCS model because if a change occurs in a transition that depends on or is dependent on some other transitions, then our DSE will have to fully explore the changed transition and all its dependencies.

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.

E. Threats to Validity

A threat to the external validity of our evaluation is given by 1) the relatively small number of models and change types, and by 2) the use of KLEE for the symbolic execution of action code. More experiments involving different models, change types, and supporting tools would mitigate this threat. Internal validity is threatened by bugs in our prototype. 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.

VI. Related Work

Existing approaches to improve efficiency and scalability of symbolic execution when applied to evolving programs are discussed in [7] and [6]. In [7], 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 [13] but in the context of state-space exploration for evolving programs using some model checkers. In [6], 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 [14] for regression model checking.

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

In his statement paper "Evolution, Adaptation, and the Quest for Incrementality", Ghezzi [15] 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.

VII. Conclusion

In this paper, we presented two different techniques for optimizing the symbolic execution of evolving state machines. 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 certain scenarios 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 communicating state machines.

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

[2] I. Rational, "Rational Rhapsody Developer, http://www-03.ibm.com/software/products/en/ratirhap/."

[3] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[4] C. Gaston, P. Le Gall, N. Rapin, and A. Touil, "Symbolic execution techniques for test purpose definition," in *Testing of Communicating Systems*. Springer, 2006, pp. 1–18.

[5] K. Zurowska and J. Dingel, "Symbolic execution of uml-rt state machines," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM, 2012, pp. 1292–1299.

[6] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 504–515.

[7] G. Yang, C. S. Păsăreanu, and S. Khurshid, "Memoized symbolic execution," in *ISSTA'12*. ACM, 2012, pp. 144–154.

[8] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[9] A. Khalil and J. Dingel, "Incremental Symbolic Execution of Evolving State Machines using Memoization and Dependence Analysis," Queen's University, Tech. Rep. 2015-623, pages 1-42, 2015.

[10] K. Androutsopoulos, D. Clark, M. Harman, Z. Li, and L. Tratt, "Control dependence for extended finite state machines," in *Fundamental Approaches to Software Engineering*. Springer, 2009, pp. 216–230.

[11] V. Chimisliu and F. Wotawa, "Improving test case generation from uml statecharts by using control, data and communication dependencies," in *13th International Conference on Quality Software (QSIC)*. IEEE, 2013, pp. 125–134.

[12] A. L. J. Dominguez, "Detection of feature interactions in automotive active safety features," Ph.D. dissertation, University of Waterloo, 2012.

[13] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan, "Incremental state-space exploration for programs with dynamically allocated data," in *ICSE'08*. ACM, 2008, pp. 291–300.

[14] G. Yang, M. B. Dwyer, and G. Rothermel, "Regression model checking," in *ICSM'09*. IEEE, 2009, pp. 115–124.

[15] C. Ghezzi, "Evolution, adaptation, and the quest for incrementality," in *Large-Scale Complex IT Systems. Development, Operation and Management*. Springer, 2012, pp. 369–379.