Change-driven Incremental Symbolic Execution of Evolving State Machines

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

***Abstract*—This paper summarizes our research findings on optimizing the symbolic execution of evolving state machines using incremental analysis.**

I. Problem and Motivation

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

The IBM Rational Rhapsody framework [1] is one of the MDE commercial tools that is heavily used in practice (e.g., in the automotive industry). Rhapsody Statecharts (also known as Harel's Statecharts [2]) 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 entry and exit actions and hierarchical composite states with orthogonal regions.

Symbolic execution is a well-known analysis technique that systematically explores all possible execution paths of behavioral software artifacts (e.g., programs [3] and state-based models [4], [5]) using symbolic inputs such that we can derive precise characterizations of the circumstances in which a specific path is taken. The output of the analysis is a symbolic execution tree (SET) which provides the basis for various types of analysis and verification, including reachability analysis, guard analysis, invariant checking and instant test case generation. One of the key challenges of symbolic execution is scalability, especially when applied to big, complex artifacts where the size of the output SET becomes very large. Repeating the entire analysis even after small changes is not the best solution.

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

The motivation is based on a number of facts. First, symbolic execution has been shown to be a very powerful method for the analysis of programs and there are already several commercial code analysis tools built based on it (e.g., CodeSonar [6]). Similarly, the technique has been adopted and applied in the context of state-based models (e.g., the IAR visualSTATE - Verificator [7]). Second, there is an interest from our industrial partner to improve the model-level analysis capabilities of the IBM Rhapsody tool. Third, research on optimizing the symbolic execution of evolving programs has been recently addressed [8], [9], however to the best of knowledge we are the first to consider such optimizations for the symbolic execution of evolving state machines.

II. Background and Related Work

Although the majority of existing analysis methods for Statecharts-like models (e.g., UML state machines, UML-RT, STATEMATE and Simulink Stateflow) make use of some existing formal verification tools, mainly model checkers such as SPIN [10], [11] or SMV [12], there are also approaches that adopt the symbolic execution analysis technique of programs and use it in the context of state-based models including:

- Modechart specifications (a variation of Harel Statecharts that incorporates timing constraints in the models) for test sequences generation [13];
- Labelled Transition Systems (LTSs) for test case generation [14] and test case selection [15];
- Input Output Symbolic Transition Systems (IOSTSs) [4] for test purpose definition;
- STATEMATE statecharts [16] and UML State Machines [17] for verifying temporal properties of the subject models;
- Simulink/Stateflows [18] for analysis and test case generation of flight software using Java PathFinder and Symbolic PathFinder;
- UML-RT State Machines [5] to support a variety of analyses for this type of models.

In our work, we followed the approach of Zurowska and Dingel [5] with some variations to develop a symbolic execution

module for Rhapsody Statecharts. In contrast to their work [5], 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 [19], to symbolically execute action code encountered in the Statecharts, whereas they use an in-house symbolic execution engine.

In his statement paper "Evolution, Adaptation, and the Quest for Incrementality", Ghezzi [20] 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 this vision needs to be employed by modern analysis and verification methods.

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 [9] and [8]. In [9], 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 [21] but in the context of state-space exploration for evolving programs using model checking. In [8], 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 [22] for regression model checking. In contrast to all the aforementioned approaches, which work for optimizing the analysis of evolving programs, our work tries to realize the same concept for evolving state machine models.

III. Approach and Uniqueness

Inspired by the research work for optimizing the symbolic execution of evolving programs [8], [9], we propose two different optimization techniques that leverage the similarities 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 dependence 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.

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.

Figure 1 shows the main components of the architecture implementing the standard symbolic execution (SE) of Rhapsody Statecharts and the two proposed optimization techniques of an evolved Rhapsody Statechart: the memoization-based symbolic execution (MSE) and the dependency-based symbolic execution (DSE). The components of the architecture are listed below with a brief description of the purpose of each component. Full details can be found in a technical report [23].

SC2MLM - A transformation that transforms a Statechart model into our MLM representation. 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, condition connectors and junction connectors need to be mapped to fit the structure of the MLMs formalism. Our current transformation supports the mapping of these specific features.

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. We developed an interface to the KLEE symbolic execution engine [19] to symbolically execute the action code of each transition. The result from this interface is a set of variable assignments and path constraints that represent different feasible paths in the action code.

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

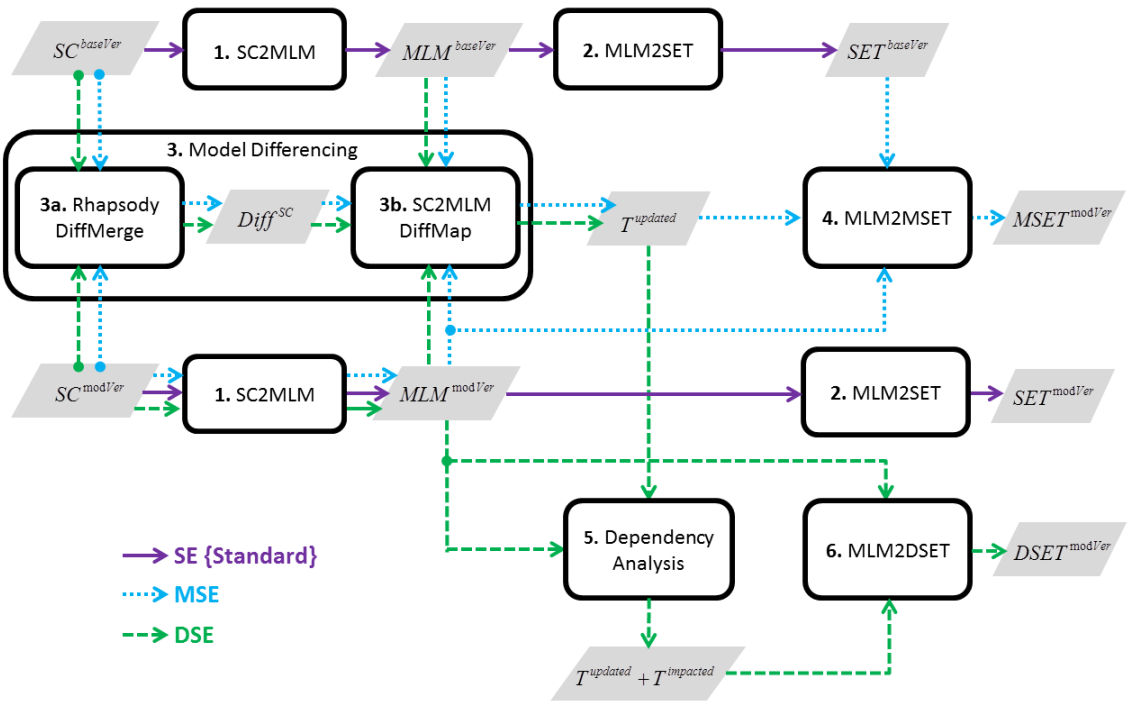


Fig. 1: The architecture of our standard symbolic execution of Rhapsody Statecharts and its optimizations (SE, MSE and DSE).

SC2MLM DiffMap - A mapping of the differences between two Rhapsody Statecharts obtained from the Rhapsody DiffMerge tool into their MLM correspondences. Currently, we consider only the following types of model 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, the output from this component is a set of updated transitions in the modified version of the model $T^{updated}$ including all transitions that have been added to, deleted from or updated in the modified version of the model.

MLM2MSET - The main component of our MSE technique. 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.

Dependency Analysis - A component for computing the set of transitions that have an impact or are impacted by any of the updated transitions found in $T^{updated}$. 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 in the modified version of the model $T^{updated}$. In this component, we first explore the input MLM model to compute all the dependences 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 dependences. We developed the algorithms implementing the definitions presented in [24], [25] for computing the dependencies in conventional Extended

Finite State Machines (EFSMs), which can also be applied to our MLMs.

MLM2DSET - The main component of our DSE technique. 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 $T^{updated} + T^{impacted}$. 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.

IV. Results and Contributions

The architecture presented in Figure 1 is implemented using three Eclipse plug-ins, summarized as follows.

- The first plug-in implements the SC2MLM and the "Dependency Analysis" components and it has been developed in the context of the IBM Rhapsody RulesComposer Eclipse framework.
- The second plug-in implements the CS2MLM DiffMap component.
- The third plug-in realizes the three symbolic execution components: MLM2SET, MLM2MSET and MLM2DSET.

To measure how effective are our optimizations compared to standard SE of a changed state machine model (i.e., how much they reduce the resource requirements of the symbolic execution of a changed state machine model) and to investigate what aspects influence the effectiveness of each technique, we ran our evaluation on 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 [26]. 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 are nested up to the third level. Additionally, the third one has two orthogonal regions.

The performance of the standard symbolic execution of the base versions of the three models took from 11 to 14 min and the numbers of the symbolic states and the symbolic execution paths of the resulting symbolic execution trees ranged from approximately 2000 to 6000 symbolic states and from 1000 to 5000 symbolic execution paths.

To measure the effectiveness and the performance of the proposed optimization techniques compared with the standard one, we first prepared a set of 67 different modified versions for the three models. Second, we ran standard SE, MSE and DSE on each modified version and recorded the time taken to run each technique and the size of their generated symbolic execution trees. Third, we compared the results of standard SE with the results of MSE and DSE.

A. Results

Figures 2a and 2b show a line graph of the average and the standard deviation values of 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 modified versions of the three given models.

Interestingly, in some modified versions, both MSE and DSE achieved savings up to 99%. However, on average, and based on the values presented in Figure 2a, we can see that, on average, MSE achieved an average savings ranged from 48% to 76%, while DSE achieved an average savings ranged from 5.5% to 96%.

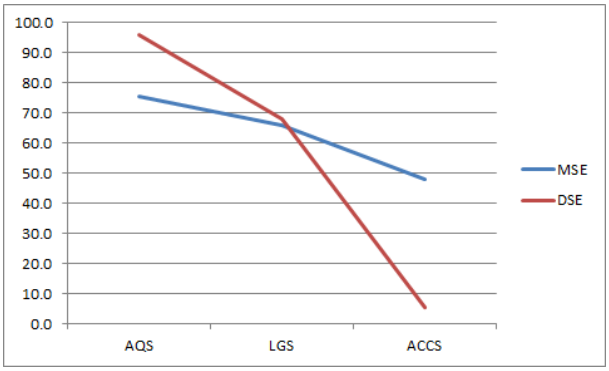
As the gaps between the average savings gained for the three models are more significant for the DSE than they are for MSE, we believe that DSE is more sensitive to the subject models.

We also notice from that the standard deviation values of the achieved savings, shown in Figure 2b, 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 version that it is for DSE.

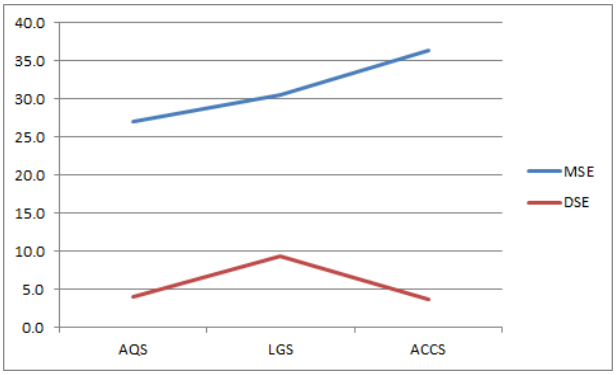
As the results from our experiments on individual evolving state machines look promising and show a significant amount of savings from both optimization techniques, we are currently working on extending our techniques to handle a system of asynchronously communicating state machines.

B. Contributions

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



(a) Average of Savings



(b) Standard Deviation of Savings

Fig. 2: Statistical Measures of the Effectiveness of MSE and DSE for the Three Models

ACKNOWLEGMENT

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

REFERENCES

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

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

[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] G. CodeSonar, "GRAMMATECH CodeSonar, http://www.grammatech.com/codesonar/."

[7] I. visualSTATEl, "IAR visualSTATE, https://www.iar.com/iar-embedded-workbench/add-ons-and-integrations/visualstate/."

[8] 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.

[9] G. Yang, C. S. Păsăreanu, and S. Khurshid, "Memoized symbolic execution," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012, pp. 144–154.

[10] E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann, "Implementing Statecharts in PROMELA/SPIN," in *2nd IEEE Workshop on Industrial Strength Formal Specification Techniques, 1998. Proceedings.* IEEE, 1998, pp. 90–101.

[11] D. Latella, I. Majzik, and M. Massink, "Automatic verification of a behavioural subset of UML statechart diagrams using the spin model-checker," *Formal Aspects of Computing*, vol. 11, no. 6, pp. 637–664, 1999.

[12] E. Clarke and W. Heinle, "Modular translation of Statecharts to SMV," Carnegie Mellon University, Pittsburgh, PA, Tech. Rep., 2000.

[13] N. H. Lee and S. D. Cha, "Generating test sequences using symbolic execution for event-driven real-time systems," *Microprocessors and Microsystems*, vol. 27, no. 10, pp. 523–531, 2003.

[14] L. Frantzen, J. Tretmans, and T. A. Willemse, *Test generation based on symbolic specifications*. Springer, 2005.

[15] T. Jéron, "Symbolic model-based test selection," *Electronic Notes in Theoretical Computer Science*, vol. 240, pp. 167–184, 2009.

[16] A. Thums, G. Schellhorn, F. Ortmeier, and W. Reif, "Interactive verification of statecharts," in *Integration of Software Specification Techniques for Applications in Engineering*. Springer, 2004, pp. 355–373.

[17] M. Balser, S. Bäumler, A. Knapp, W. Reif, and A. Thums, "Interactive verification of uml state machines," in *Formal methods and software engineering*. Springer, 2004, pp. 434–448.

[18] C. S. Pasareanu, J. Schumann, P. Mehlitz, M. Lowry, G. Karsai, H. Nine, and S. Neema, "Model based analysis and test generation for flight software," in *Third IEEE International Conference on Space Mission Challenges for Information Technology, 2009. SMC-IT 2009.* IEEE, 2009, pp. 83–90.

[19] 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.

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

[21] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan, "Incremental state-space exploration for programs with dynamically allocated data," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 291–300.

[22] G. Yang, M. B. Dwyer, and G. Rothermel, "Regression model checking," in *IEEE International Conference on Software Maintenance (ICSM 2009)*. IEEE, 2009, pp. 115–124.

[23] 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.

[24] 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.

[25] 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.

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