Incremental Test Case Generation for UML-RT Models Using Symbolic Execution

Eric James Rapos, Juergen Dingel
School of Computing
Queen's University
Kingston, Canada
{eric,dingel}@cs.queensu.ca

Abstract— **Model driven development (MDD) is on the rise in software engineering and no more so than in the realm of real-time and embedded systems. Being able to leverage the code generation and validation techniques made available through MDD is worth exploring, and is a large area of focus in academic and industrial research. However given the iterative nature of MDD, the evolution of models causes test case generation to occur multiple times throughout a software modeling project. Currently, the existing process of regenerating test cases for a modified model of a system can be costly, inefficient, and even redundant. Thus, it is our goal to achieve an improved understanding of the impact of typical state machine evolution steps on test cases, and how this impact can be mitigated by reusing previously generated test cases. We are also aiming to implement this in a software prototype to automate and evaluate our work.**

Keywords- model-based testing; model-driven development; test case generation; symbolic execution

I. MOTIVATION

Given the inefficiency of a complete regeneration of test cases after every evolution step, we aim to reuse existing test cases to avoid any unnecessary generation. It is our goal to generate test cases incrementally based on an existing test suite and some model evolution performed on a given model. Working specifically with UML-RT state-machines, we aim to solve this problem by incrementally generating the test cases each time a model evolution step occurs. Instead of regenerating an entire test suite we will examine the existing test cases and determine which tests need to be updated, removed or whether or not we need to add more tests.

Another aim of this work is to achieve a better understanding of the effects of model evolution. By determining how evolution steps impact execution and test case generation, we will be better able to understand how models evolve over time, and how this execution can be supported efficiently.

II. BACKGROUND

UML-RT is a real time profile of UML, dealing specifically with modeling real time interactions between a system and its surrounding environment. The main difference between a regular UML model representation of execution and a UML-RT representation is the notion of capsules. A capsule is a singular component of a system that may interact with other capsules via protocols, and each capsule's behaviour is modeled using a state machine. The main difference between UML-RT state machines and standard UML state machines is that UML-RT state-machines do not have any orthogonal regions. The environment being used for UML-RT is IBM's RSA-RTE [5] Version 8.0, which is an Eclipse-based IDE.

There is currently work being done on symbolically executing UML-RT State-Machines [1][2] for the purpose of analysis of models rather than code. As with symbolic execution of code, when symbolically executing a UML-RT state machine, a symbolic execution tree (SET) is produced. A SET contains all possible execution paths of the model, based on symbolic inputs, as well as any constraints on those inputs. Using EMF [6] we have created a meta-model for SETs, which allows us to create instances of SETs that can be used for incremental test case generation. SETs are used for reachability testing, path constraint evaluation, and of course test case generation, our main use for them.

III. MODEL EVOLUTION STEPS

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

IV. PROCESS

A. Differencing Symbolic Execution Trees

Given a SET for a model, and one for another model with one evolution step performed to it, we have two trees with presumably a large amount of similarity and a few select differences. Using a tree differencing algorithm where we compare the two SETs using a breadth-first method, we are able to determine the highest level differences along all paths in the tree. If along a given path the leaves of both

SETs are reached, there is no difference in execution. By the nature of execution trees, when a difference is found it is possible, and probable, that the change will propagate through the rest of that path. This means that the test case for that execution will need to be modified in some way to reflect the new execution behaviour. When a difference is found, it is recorded and the search continues with the rest of the tree until all paths have been fully explored.

B. *Initial Test Case Generation & Examination*

We perform the initial test case generation from a SET by traversing through the tree in a depth-first manner. Beginning at the root, we traverse each possible path of the tree, moving from symbolic state to symbolic state through a series of transitions that are based on the transitions of the UML-RT state machine. As each transition is taken, the required inputs and events, and the constraints on the parameters are recorded in sequence for later use. At the end of each path, when a leaf node is reached, that constitutes a full execution, and the path constraints on any input variables are solved using the Choco Constraint Solver [7]. The solved values are substituted in where the symbolic variables once were and the ordered events are presented as a full test case. This is repeated for each path, and therefore the generated test suite will provide path coverage for all executions. Given a test suite for a model, and the differences observed for that model and an evolution of it, we are able to determine what changes need to occur so that the test suite is sufficient for the modified model. A test case is considered to remain valid if it does not contain a path to one of the differences noted in Section *IV(A)*. Ideally the number of test cases that remain unchanged should be quite high, assuming a small granularity of change.

Note that not only does this examination isolate which tests do no need to be regenerated; it also determines which tests do not need to be rerun on the evolved model. Given that execution will occur in exactly the same manner for the remaining test cases, they do not need to be run again, and this is a significant gain in the efficiency of testing.

C. *Incremental Generation of New Test Cases*

After having removed all test cases that cannot be reused, we must add any newly required tests. This is done by using the state highest in the tree along the path where a difference occurred as the root node of the test case generation algorithm. This will generate tests from that point on, and we then prepend the prefix to each of these tests so they satisfy the full execution path for the new or modified execution of the new model. Adding these new test cases to the remaining original tests gives a full test suite for the new model.

V. CURRENT STATUS

We currently have a prototype that is capable of differencing two trees and determining the highest point of difference along each path as well as generating test suites based on a given SET. Currently we are finishing the implementation of the incremental generation step (Section *IV(B)*). Additionally we have compiled a number of models of different sizes, complexity, etc. and applied the model evolution steps discussed in Section *III* to each of them in order to have a sufficient amount of testing data.

VI. FUTURE WORK

The efficiency/effectiveness of the process outlined above can be further enhanced by reducing the need for symbolically executing the modified model; i.e. by computing the SET of the modified model by reusing as much as possible of the SET of the original model. Using the classifications discussed in the introduction we can reduce the need for symbolic execution almost entirely, only performing it when necessary. We hypothesize that based on the classifications a pattern should emerge, indicating that each time an evolution step is performed on a model, its impact on the SET and test cases for the modified model can be captured by more direct and efficient update operations.

VII. RELATED WORK

There has been work done in incremental test case generation for software product lines [3], which validates the concept that incremental generation is a worthwhile area to pursue. Additionally, the process has been explored in the realm of model based testing [4] and the importance of incremental development using modeling languages. The occurrence of similar research areas is a confirmation of the interest in this type of work.

VIII. CONCLUSION

Using symbolic execution as a means for comparison as well as test case generation, we aim to identify a set of classifications that will illustrate exactly how model evolution steps will affect test cases. Using these classifications we will develop a tool that can incrementally generate tests using existing test suites by avoiding redundant computation whenever possible.

REFERENCES

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

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

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

[4] A. Pretschner, H. Lötzbeyer, J. Philipps, “Model based testing in incremental system development”, Journal of Systems and Software Volume 70 Issue 3, March 2004

[5] IBM Rational Software Architect Real-Time Edition (RSA-RTE) - http://publib.boulder.ibm.com/infocenter/rsarthlp/v8/index.jsp?topic=/com.ibm.xtools.rsarte.legal.doc/helpindex_rsarte.html

[6] Eclipse Modeling Framework (EMF) - http://www.eclipse.org/modeling/emf/

[7] Choco Constraint Solver – http://choco.emn.fr/