Concolic Testing for Models of State-Based Systems

Reza Ahmadi
Queen's University
Kingston, Ontario, Canada
ahmadi@cs.queensu.ca

Juergen Dingel
Queen's University
Kingston, Ontario, Canada
dingel@cs.queensu.ca

ABSTRACT

Testing models of modern cyber-physical systems is not straightforward due to timing constraints, numerous if not infinite possible behaviors, and complex communications between components. Software testing tools and approaches that can generate test cases to test these systems are therefore important. Many of the existing automatic approaches support testing at the implementation level only. The existing model-level testing tools either treat the model as a black box (e.g., random testing approaches) or have limitations when it comes to generating complex test sequences (e.g., symbolic execution). This paper presents a novel approach and tool support for automatic unit testing of models of real-time embedded systems by conducting *concolic* testing, a hybrid testing technique based on concrete and symbolic execution. Our technique conducts automatic concolic testing in two phases. In the first phase, model is isolated from its environment, is transformed to a testable model and is integrated with a test harness. In the second phase, the harness tests the model concolically and reports the test execution results. We describe an implementation of our approach in the context of Papyrus-RT, an open source Model Driven Engineering (MDE) tool based on the modeling language UML-RT, and report the results of applying our concolic testing approach to a set of standard benchmark models to validate our approach.

CCS CONCEPTS

• **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Concolic Testing, Model-driven Engineering, State Machine

ACM Reference Format:
Reza Ahmadi and Juergen Dingel. 2019. Concolic Testing for Models of State-Based Systems. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338906.3338908

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338908

1 INTRODUCTION

Modern Real-Time Embedded software (RTE) plays a fundamental role in controlling many products and devices found in telecommunication systems, automobiles, aircraft and many other cyber-physical systems. In such complex systems, the behavior of the system depends on real-time constraints as well as on complex communications with the environment using various protocols. Using code-centric-only approaches for developing complex RTE systems is very challenging. MDE techniques tackle this challenge by raising the level of abstraction on which the developers construct software. If a software model contains faults, these faults will propagate to any refinement of that model or the code that is generated from the model. Therefore, finding and resolving faults at the model level is critical for developing high quality software. Although MDE principles including abstraction, automation, and analysis [55] can help deal with the complexity of software, models of modern industrial systems still often are large and can become overwhelming. Thus testing of such models is challenging without automatic tool support.

Symbolic execution [19, 20] is able to automatically generate test cases for a program that achieve high coverage of the program executions. However, symbolic execution typically results in path explosion [20] when executing large systems. The situation can be worse in analysing concurrent systems, where the state space is larger. The scalability issue has been partially addressed, e.g., by distributing the computations on a cloud [15, 26] or by conducting selective symbolic execution [22]. But there are some other issues, e.g., since symbolic execution is typically static, it may not be precise in some situations, e.g., when there are statements in the program that call library functions that are hard to reason about (e.g. calls to operating system libraries) or when the program includes pointer manipulations or pointer aliases [32]. In addition, symbolically executing programs or models with complex constraints or data structures is challenging [19, 60], e.g., symbolic execution will be stuck if it faces non-linear arithmetic path constraints [33]. Besides, symbolic execution typically constructs and maintains a Symbolic Execution Tree (SET) to keep track of path constraints for executions, where for complex systems it is computationally intractable to precisely maintain and solve constraints for test generation due to numerous execution paths [60].

Concolic testing, on the other hand, executes a program both concretely and symbolically. In this technique, a program is executed on some random inputs and symbolic constraints are collected during the course of that execution. Then, the collected constraints are negated and solved again to execute alternative branches in the program. This process is run in a loop until all branches are executed or a user-specified coverage criterion is met. The fact that symbolic constraints are collected during a concrete execution of the program allows the generation of test inputs that will force

the program along the exact same execution. Therefore, every error caught by taking an execution path by the concolic testing is guaranteed to be sound [33]. In addition, a program that includes function calls to external libraries will prevent the symbolic execution from collecting path constraints, since the program includes statements that involve constraints that are outside the scope of the symbolic execution engine. Concolic testing engines, such as DART [33], typically solve this problem by randomly generating inputs and running the function calls to reason about the predicates and to collect constraints [32, 33].

MDE is becoming more prevalent, and software models tend to become large and complex. For instance, Models of RTE systems in automotive and aerospace domains may encompass many interacting state machines that communicate using various protocols with complex action code on the transitions of the state machines. In such models, testing some branches on a state machine can be very challenging since the execution of such branches may need a sequence of events where these events are highly constrained by the input parameters. It becomes more challenging when the execution of an event is guarded by a conditional where that conditional depends on previous event parameters. Observe that in complex RTE systems, event parameters are often of complex data types (rather than primitive types). Concolic testing can be an effective technique for testing these systems. However, as opposed to concolic program testing, model-level concolic testing has not been addressed in the literature. One may propose using the current code-centric concolic testing engines for testing the behavior of models by testing the code generated from the models. However, there are currently concolic testing engines for only a few languages, so it may not be possible to test programs generated (in an arbitrary target language) from models. In addition, the information collected during testing the programs is not easily traceable to the models. Therefore, more work is needed to leverage concolic testing for software models, in particular for models of reactive and real-time systems.

In this paper, we present a novel approach and prototype tool for automatic concolic testing models of RTE systems. To this end, we automatically generate a harness for a model to simulate its environment. The harness is integrated with a model-level concolic engine to dynamically generate test inputs for the model and stimulate it with the generated inputs. We have built our concolic engine on UML-RT [52], a domain specific language, whose constructs come from standard UML constructs [56]. UML-RT is a popular industrial modeling language that is used for modeling industrial systems and is supported by several open-source and commercial tools (Eclipse Papyrus-RT [14], Eclipse eTrice [5], IBM RSA-RTE [3], HCL RTist [7] and IBM RoseRT [6]). Our prototype implementation uses the open source MDE tool Papyrus-RT. To evaluate our approach, we use an industrial case study provided by one of our industrial partners as well as a set of other academic case studies of different sizes and complexities.

In the next section, we briefly introduce UML-RT with an example model and an overview of the current state of the art in the program- and model-level concolic testing. Then, we explain our approach. Next, the implementation of our concolic testing approach is presented. We then briefly describe our prototype tool along with an evaluation of the approach and the tool.

2 BACKGROUND

Developing Executable Models in UML-RT. Constructing complex, often distributed real-time systems needs powerful, well-defined modeling constructs, as well as strong tool support. These constructs and tools can be used to design a well-defined architecture for such complex systems, which eases not only the development of the initial system but also facilitates maintenance and evolution [56]. UML for Real-Time (UML-RT) [56] is a domain-specific language dedicated for modeling real-time systems. UML-RT is a UML profile (similar to, e.g., MARTE [30] and SysML [31]). Since UML-RT focuses on modeling real-time systems, it is, compared to UML, a small language with light notation.

The main concepts of UML-RT are *capsules*, *ports*, and *connectors* [52]. A capsule is an independent active class with its own control flow. Capsules own ports, allowing them to communicate via message passing. State machines model the behaviour of capsules. A UML-RT state machine is an extension of a Mealy state machine [46] augmented with extra features, including state actions, composite states, and deep-history.

UML-RT code generators generate complete executable code for the structural and behavioral aspects of a model. Since UML-RT is both a specification and implementation language, the action code written in the model is integrated as part of the code generated from a model. Fig. 1 presents some behavioral and structural aspects (state machine, sample action code, and capsules) of a Collision Avoidance (CA) system. This system prevents or mitigates collisions by continuously monitoring the road ahead and parts of the side-fronts of the vehicle. Whenever an obstacle is detected, it notifies the driver by audible or visual alerts. In addition, the system automatically brakes, vibrates or steers the wheels in the opposite direction if it detects an imminent collision. Based on the figure, the *CA_Controller* capsule communicates with four other capsules via the ports *lidar*, *panel*, *brake*, and *steer* (observe the action code on transitions in Fig. 1). The first port receives threat measurements from four *lidar* components. Lidar uses the laser to estimate the distance between objects and the direction of the potential collisions with objects. The controller sends error or warning signals to the user panel component via the port *panel* to inform the driver using audible and visual alerts. The controller also communicates with the brake and steering wheel systems by sending commands to the ports *brake* and *steer* to apply the brake, and to vibrate or to steer the steering wheel based on the threats measured. This sample model will be used in the following sections to demonstrate our approach for model-level concolic testing.

Program Symbolic and Concolic Execution. The key idea behind symbolic execution is to use symbolic values instead of concrete data values for program inputs and maintain a collection of symbolic expressions over the symbolic values to represent the program variables throughout the program execution [20, 43]. Symbolic execution maintains a *path constraint* Φ, which consists of quantifier-free first-order formulas over the symbolic expressions, and a *symbolic state* σ, which maps each program variable to a symbolic expression. At execution time, initially Φ is *true* and σ is an empty map. These two variables are updated throughout the program execution. For instance, if at a location in the program input is read using $v = input()$, then $v \mapsto s$ is added to σ, where s

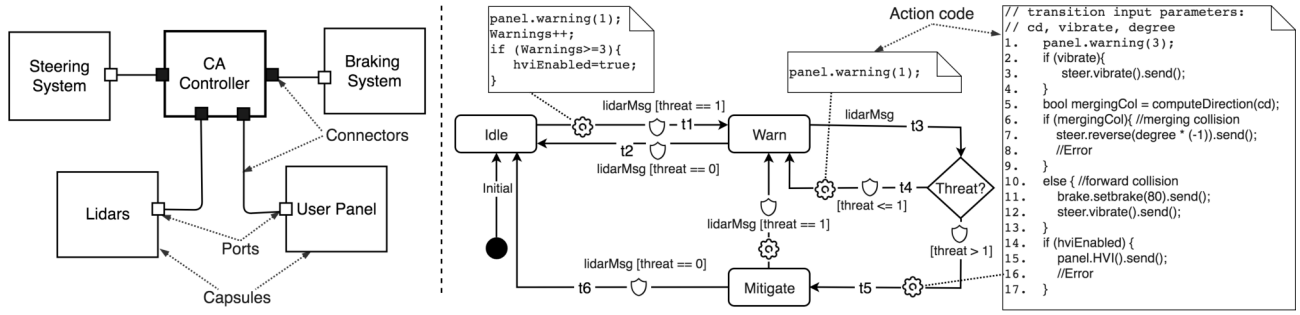

Figure 1: The structure of the CA controller model and its environment: left. The state machine of CA: right

is a fresh symbolic value. The expression means a fresh symbolic variable is allocated for each input variable. During execution, at every assignment statement $v = e$, symbolic execution updates σ by mapping v to $\sigma(e)$, where $\sigma(e)$ is the symbolic expression obtained by evaluating e in σ. At a conditional statement *if (e) then* S_0 *else* S_1, the path constraint Φ is updated to $\Phi \wedge e$ and a new path constraint Φ' is constructed and updated to $\Phi \wedge \neg e$. At this point of execution, if $\Phi \wedge e$ is satisfiable, then a new symbolic execution branch is created and the execution proceeds from the *then* branch. Similarly, if $\Phi \wedge \neg e$ is satisfiable, a new branch is constructed and the execution is continued from the *else* branch. Observe that this kind of execution gives rise to a tree of symbolic executions which is called *Symbolic Execution Tree (SET)*.

At the end of the execution or when an error happens, e.g., at an assertion violation or program crash, Φ (the current path constraint in the SET) is solved using a constraint solver, and test inputs are generated. By executing the program using the generated inputs, the program follows the same path as the symbolic execution. For more details about symbolic execution, please refer to [19, 20, 43].

Concolic testing executes a program both concretely and symbolically [16, 33, 58–60]. In this technique, symbolic execution is conducted dynamically, which means the program under test is executed and during that execution, symbolic expressions and path constraints are collected. Concolic execution maintains two maps of program variables: the symbolic state that maps the variables to symbolic expressions and the concrete state that maps the variables to concrete values. This technique needs some initial program input, which is generated randomly to initiate the execution. At the end of each execution, a constraint in the collected path constraints is negated (the constraint to negate is either selected randomly, systematically or based on some other heuristics [58]), and the program is executed again using the newly generated inputs (by solving the negated constraint) to steer the program along a new execution path. The concolic testing conducts this task either systematically until all feasible distinct execution paths have been visited or the testing budget runs out. Observe that in concolic testing, a SET is not generated, rather a list of path constraints is generated and updated throughout the whole execution and thus there is not the issue of saving and updating the SET.

Program concolic testing has already proved its potential for achieving high code coverage [16, 33–35, 60] for catching bugs in programs that are otherwise very hard to catch using other approaches including static analysis tools and random testing. One example successful application of concolic testing in the industry is SAGE [34] that has been used by Microsoft for catching many vulnerability bugs in large projects including Windows 7 [34]. However, concolic testing of models has not been studied before even though MDE is becoming more prevalent and models tend to become larger and more complex. For instance, the simple model shown in Fig. 1 specifies the behavior of a reactive system that executes by frequent communications with its environment. These types of models have action code on their transitions to conduct various computations including to process incoming messages, update attributes, and produce outgoing messages. These systems execute only when they are in their appropriate environment (e.g., are connected to devices) that sends them the required stimuli. To test these systems during development, one needs to mock the environment for the system, for instance through a test harness, such that the harness sends a sequence of messages in a specific order with appropriate input parameters. It is challenging and labor intensive to manually craft a test harness for each model (this task includes creating a capsule as the test harness, ports, and connectors as well as the behavior of the test harness). Therefore to automate testing, one may need to generate the structure and behavior of the test harness dedicated to each model under test. To archive high coverage, the harness should intensively test the model and monitor and report the test coverage information. In the following sections we explain our approach for concolic testing such models.

3 APPROACH

Given a state machine under test and a test budget (can be the allotted time for testing or the maximum number of consecutive transitions to execute) as input, the objective of our approach is to automatically test the state machine to exhibit as many executions as the test budget permits, to increase the chance for finding bugs in the state machine. In this section we start with a motivating example and an overview of our approach before we elaborate different steps of our technique for model-level concolic testing. Throughout this section, we call the state machine under test *the model*, where *the test harness* is responsible for testing this model.

3.1 Motivating Example

The simple model presented in Fig. 1 has three bugs, two of which can be detected if the lines 8 and 16 on the action code of the transition *t5* (incoming transition to the state *Mitigate*) are executed. For instance, to execute line 8, the state machine must receive the following messages and parameters values must satisfy the constraints in parenthesis: *lidarMsg(threat$_0$==1)* to execute the transition Idle → Warn, and *lidarMsg(threat$_1$>1, computeDirection(cd$_0$)==true)* to execute the transition Warn → Mitigate and execute the second *if* statement. So, in this case, solving the path constraints (PCs) *(threat$_0$==1 ∧ threat$_1$>1 ∧ computeDirection*(cd$_0$)==*true*) allows generating test inputs that will force the state machine to reach and execute line 8. To execute the corresponding *else* branch (line 10), the concolic engine negates the last constraint to end up the PCs *(threat$_0$==1 ∧ threat$_1$>1 ∧ ¬(computeDirection*(cd$_0$)==*true*)), so by solving it the *else* branch is executed. Observe that the third *if statement* (line 14) executes if and only if *hviEnabled* has been enabled in previous transitions, where this predicate gets enabled if the transition *t1* executes at least three times, such that the predicate *Warnings>=3* holds (*Warnings* is a global variable). Therefore, as shown, the execution of a branch on a transition is dependent on the valuation of a global variable in a specific way. This system should not vibrate and steer the wheel at the same time [1, 39]. That is, it vibrates the steering wheel if it detects a forward collision, and steers the car in the opposite direction if it detects a merging collision (the type of the collision is computed by a library using the *cd* input parameter). As transition *t5* shows, if the flag *merging-Col* holds and vibrate is enabled (by a message from the user), this requirement is violated. So this last bug is subtle and can be caught by analyzing the execution traces only, as has been proposed in [8].

3.2 Approach Overview

In our approach, a test harness is responsible for constructing messages and parameters and sending them to the model to trigger transitions on the model and hence execute the action code on the model. A transition might be executed several times to exercise all possible executions on the transition action code. To this end, before conducting testing, the model is transformed by augmenting it by new model elements to enable the harness to control the execution of the model and to steer the model along its different executions. Moreover, as mentioned, in concolic execution, the model is executed both concretely, executing the state machine through the generated input parameters, and symbolically to collect path constraints on values at symbolic variables constructed in terms of input parameters. To enable these side by side executions, during the model transformations, we conduct a set of instrumentations on the action code of the state machine under test by adding new action code beside the existing one, which is responsible for collecting symbolic path constraints. Once the model is transformed, we generate the harness as well as the extra elements required to integrate the harness and the model. Finally, we rely on the standard code generator to generate code from both the harness and the model, as well as the glue code for integrating the two. The code generated is executed, so harness and the model execute concurrently. In the following sections, we present all the steps mentioned above in detail. We first explain the steps of our model transformations and the rationale behind them.

3.3 Model Transformations

Model-Harness Synchronization. The harness and the model under test execute concurrently and asynchronously. The harness should test a model with a sequence of test inputs and after the processing of each input the model should send an acknowledgment to the harness. To this end, during instrumentation, new action code statements are added to the states such that upon complete execution of each transition on the model, the model informs the harness, so the harness can send the next test input. For instance, if a model has transitions that are enabled by timers, the harness is not aware of the exact timeout of a timer on the model, so sending messages from the model to the harness upon execution of a transition helps to synchronize the harness and the model. The message *newState* added on the instrumented model in Fig. 3 informs the harness about a newly visited state, so the harness can prepare and send the next appropriate message to the model under test.

Path Constraint Collection. We instrument the action code on the transitions by inserting extra commands on the action code (Table 1). The instrumented transition executes as the original one, but also invokes the symbolic execution engine for collecting path constraints (PCs). The instrumentations shown in the last row of Table 1 serve as another purpose (than collecting PCs) which are explained later. Given the fact that a transition may be guarded by an expression (a transition is enabled iff it receives the required message trigger and the transition guard expression holds) and a state machine may include *choice points* (states that have multiple outgoing branches where the decision on which branch is executed is based on the guard on that branch), before the instrumentations, we first conduct the following small transformations on the model.

(1) *Transforming choice points*: if the target state of a transition t is a choice point, for each guard expression on the choice point's outgoing transitions we add new corresponding if statement blocks on the action code of t. Therefore, the corresponding path constraints are collected for each outgoing transition, so proper test input is generated to cover all the outgoing transitions. For instance, as shown in Fig. 3, *"Threat?"* is a choice point with two outgoing transitions. The guards on these two outgoings give rise to two new *if statements* on the transformed transition *t3*.

(2) *Transforming guards*: as mentioned, any transition is initially executed by receiving random data, and hence the concolic engine may not be able to execute guarded transitions initially (since it is very likely that random data does not satisfy the guard predicates). Therefore, such transitions need some transformations that by preserving the original behavior, enable the tool to execute them by some random data. To this end, if there is a transition $t_0 : S_0 \rightarrow S_1$ (as shown in Fig. 2), two new transitions t_1 and t_2 and a choice point c are created. The transition t_1 is the incoming transition to c, and t_0 and t_2 are its outgoing transitions. As shown in the figure, t_0 now connects the choice point to the state S_1. In addition, the action code of t_1 now forces the concolic engine to collect two new constraints that can give rise to generating inputs that satisfy the guard of t_0 and cause the execution of the action code of t_0. More examples of this transformation are in Fig. 3.

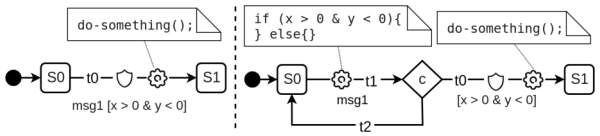

Figure 2: Transforming Guards. Left: Original Model, Right: Transformed One.

Table 1: Action code Instrumentations

Transition Action code		Transition Type
Before Instru.	**After Instru.**	
//reading input $v \leftarrow input();$	$\sigma \leftarrow \sigma + (v \mapsto s);$ $v \leftarrow input();$	$others$
//assignment $v \leftarrow e;$	$m_0 \leftarrow v \mapsto eval\_sym(e, \sigma);$ $\sigma \leftarrow \sigma + [m_0]; v \leftarrow e;$	$others$

//conditional
if (e) then S_0;
else S_1;
	$e \leftarrow eval\_sym(e, \sigma);$
	if (e) then $\{\Phi \leftarrow \Phi \wedge e; S_0;\}$
	else $\{\Phi \leftarrow \Phi \wedge \neg e; S_1;\}$
		$others$

	$reset\_vars(); save\_pcs();$	$iteration$

Action Code Full Coverage. Traditionally the notion of coverage in testing state machines includes transitions/states or predicate coverage [48], so to test a state machine completely one should generate as many as test cases as necessary to trigger all transitions on the state machine or satisfy all guard expressions on all transitions. In our approach, we consider exploring all the branch points on the transition action code as well, which means we should not only consider exploring all the transitions, but also the action code on those transitions. The action code on a transition may have multiple branch points (e.g., due to if statements), and hence the transition should be executed multiple times with different inputs in order to exercise all possible executions. Since after each execution the state of the system changes (either by transiting to a new state or by updating some global variables), the model should be able to restore its original state after each execution. To this end, during the transformations, the model is augmented by new instructions such that after each execution, the test harness can move the model under test back to a starting point, such that the variable values are restored so the execution can be conducted on the original global variable values. To this end, a transition from each state to the initial state is created (to form a loop) and we call the new transition an *iteration*. We call the transitions in this loop a *transition set*. An iteration allows the harness to execute a transition set multiple times. For instance, the transitions *iterate1* and *iterate2* in Fig. 3 represent two iterations. As shown in the last row of the Table 1, the two lines of action code that are executed on an iteration are $reset\_vars()$ and $save\_pcs()$. The former one restores the state machine's variable values and the later one saves the collected path constraints of a transition set into a file. The harness sends new inputs to the model in each iteration, and that happens by restoring the constraints collected by the model, and negating and solving them upon completion of the execution of a transition set. We will elaborate this in the following sections.

3.4 Concolic Testing of the State Machine

In our approach, a test harness is generated for each model under test and is integrated with the model automatically (Fig. 4). Based

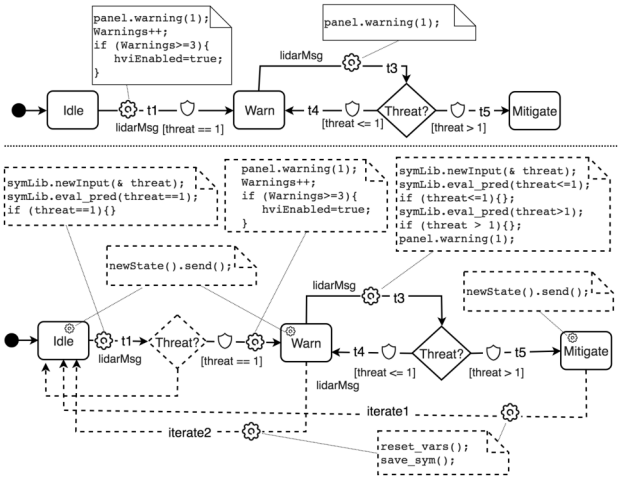

Figure 3: A simplified version of the CA model shown in Fig. 1 before (top) and after (bottom) the model transformations. Model elements dashed are the elements that are newly added/updated during the transformations.

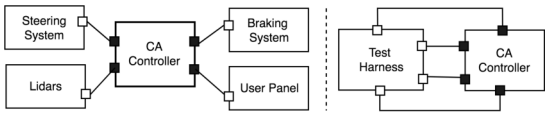

Figure 4: Integrating a model with a test harness

on the figure, the model originally communicates with four other models through their ports (Fig. 4, on the left). The harness mocks the external environment for the model (Fig. 4, on the right), so the model can be tested as a unit in isolation.

Once the state machine is instrumented and is integrated with a test harness, the state machine can be tested. In the remainder of this section we elaborate on how the test harness conducts the concolic testing using the example model shown in Fig. 3.

Fig. 5 presents an abstract behavior of the test harness via a state machine and Fig. 6 presents some of the functions and attributes of the test harness. As shown in Fig. 5, the test harness starts by reading a test configuration file (to specify test budget, such as total iterations), and then enters the state *MsgSending*. When entering this state, the harness calls the function *SendMessage* (shown in Fig. 6) to select the next transition for execution (the *candidate transition*) and to send a message to enable that transition. In the function *SendMessage*, *selectTransition* (line 12) selects the candidate transition from the outgoing transitions of the current state. Since a transition set is executed *iExec* times, the candidate transition is selected based on the following rules: in an iteration *i*, where *i<=iExec*, if none of the outgoing transitions has been executed before, then *selectTransition* chooses one of them randomly, otherwise the transition that has been executed in the iteration *i-1* is selected, where *1<=i-1<=iExec*. This is because the inputs generated from path constraints collected during execution of a transition set execute the same transition set (same transitions and same order of execution), but forces a different execution path in transitions

action code. If a transition set is executed *iExec* times, starting from the initial state, the harness picks an outgoing transition that has not been executed before or a random transition if all the outgoing transitions have been executed *iExec* times. Observe that if in a state machine there is a self-transition *st* (a transition whose source and destination states are identical), *st* can be selected *tsLen* (the transition set length) times so the next transitions never execute. To handle this issue, the harness takes as input a loop bound (*tExec* in Fig. 6), so in an iteration, the harness does not select any transition more than *tExec* times collectively. In line 13, the harness reads inputs either from file if already exists input generated by the symbolic execution engine, otherwise it generates random data for a candidate transition. Then, the harness constructs a message and sends it to the model under test (lines 14-15).

Since the transitions on the model are instrumented (observe Fig. 3), upon reception of the message by the candidate transition, the action code on the transition is executed, which leads to collecting (or updating) path constraints, and the model sends *newState* to the harness, which causes the harness to move to the state *MsgReceived* (Fig. 5), where the harness calls its function *NextStep* (shown in Fig. 6). This function first updates the coverage information (line 2), and decides whether the harness must restart the model under test forcing it to move to its initial state, or to continue executing the next transitions. This decision is based on whether or not *tsLen* (transition set length) number of transitions have been already executed (lines 3-9). If the harness decides to continue, then it sets a timer (line 4) so once the timer times out (after some milliseconds) the harness ends up in the state *MsgSending*, so it can prepare and send the next message. If not, the harness sends the message *iterate* to the model, which causes the model to move to its initial state and send a new *newState* message which causes the harness to move to the choice point *"Iterate?"*. In this state, if the total number of executed iterations equals *totalIter* (Fig. 6) or there is no more branches to execute, then the state machine moves to the state *End*, otherwise it moves back to the state *MsgSending* and the function *Negate_Solve* is executed. Observe that if the testing time budget (*execTime*) runs out, the harness always automatically terminates. We have not shown that functionality here due to space limits.

As shown in Fig. 3, after each iteration, the model stores the Symbolic Execution (SE) object into a file (by calling the function *save_sym*). So, the function *Negate_Solve* in the harness restores this SE (line 18) and based on the selected heuristic (either random or systematic, which we will elaborate later in this section) a constraint in the SE is negated (lines 19-23). Note that the path constraints collected for a transition set is the conjunction of all the constraints collected for each transition, so the test harness always maintains only one path constraint throughout the execution of the whole model (as opposed to symbolic execution that maintains a tree of path constraints for all paths in the system). The harness, then, solves the resulting negated constraints and writes the inputs generated to a file (lines 24-25) so it can later (in the state *MsgSending*) send them to the model.

Example. Assuming transition set length is 3, the sequence diagram shown in Fig. 7 presents the communication between the harness, the instrumented model shown in Fig. 3 and the symbolic execution engine. Based on Fig. 3, the state *Idle* has only one

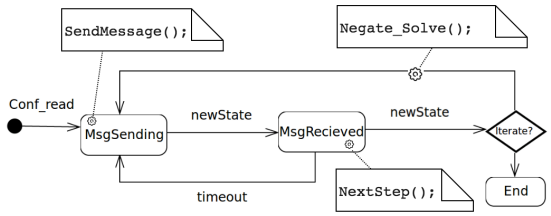

Figure 5: A simplified state machine of the test harness

Input: a state machine *sm*, and a test configuration object testConf with these attributes: heuristic *h*, total iterations *totalIter*, execution time *execTime*, transition set length *tsLen*, iteration executions *iExec*, and transition executions *tExec*.
Harness attributes: *_Executed_Transitions*, *_Sym_Ex_Obj*.
Output: The branch, transition and state coverage results.

```
1:  procedure NextStep()
2:      updateCoverage();
3:      if (_Executed_Transitions < testConf.tsLen) then
4:          timer.set();
5:          let _Executed_Transitions = _Executed_Transitions + 1;
6:      else
7:          iterate.send();
8:          let _Executed_Transitions = 0;
9:      end if
10: end procedure
11: procedure SendMessage()
12:     let candidate = selectTransition(sm.currState.outgoingTransitions);
13:     let inputs = readInputs(candidate);
14:     let message = getMsg(candidate, inputs);
15:     message.send();
16: end procedure
17: procedure Negate_Solve()
18:     let _Sym_Ex_Obj = readSymExObjFromFile();
19:     if (h==0) then //random branch selection
20:         negate_rand(_Sym_Ex_Obj);
21:     else //systematic branch selection
22:         negate_sys(_Sym_Ex_Obj);
23:     end if
24:     let inputs = solveSymExObj(_Sym_Ex_Obj);
25:     writeInputsToFile(inputs);
26: end procedure
```

Figure 6: The behavior of the test harness

outgoing transition *t1*, so the test harness sends the message *lidarMsg(1)* (parameter is randomly generated), so *t1* is executed and executing its action code gives rise to constructing the path constraints ($threat_0 == 1$). Now, the model informs the harness with a *newState* message, so the harness sends the next message (again with some random parameters) *lidarMsg(20)*, which executes *t3* as well as *t5* (since the *t*5 guard holds) and the resulting PCs will be ($threat_0 == 1 \wedge threat_1 > 1$). Now the number of executed transitions is 3, and hence the harness sends the message *iterate*, which results in restarting the model, negating the last part of the constraints (resulting in PCs $threat_0 == 1 \wedge threat_1 <= 1$), and generating new inputs (e.g., $threat_0 = 1$ and $threat_1 = 0$), which this time will execute the transitions *t1* and *t3*, and *t4*, respectively.

DFS vs. BFS Transition Execution. In our approach, we can test a model by choosing a value for the parameter *iteration length* large enough to execute all the transitions starting from the initial state all the way through to some final state (a state with no outgoing transitions), so the execution and testing of transitions will be based on the Depth First Search (DFS) traversal. Conversely, taking

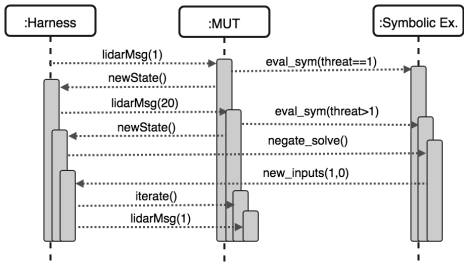

Figure 7: Communication of the harness, the model under test (MUT) and the symbolic execution engine

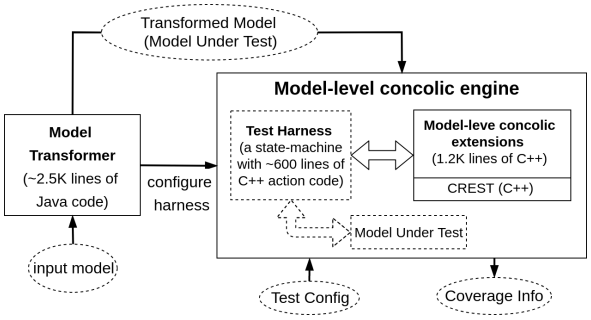

Figure 8: The overview of our prototype tool

smaller numbers as the iteration length and increasing that number iteratively can simulate the Breath First Search (BFS) traversal for testing the transitions.

Heuristics for Branch Selection. We implemented two different techniques for selecting the next branch for execution from the next executable branches: random and systematic. Assuming the number of constraints collected through executing a transition set is n, then, in the former case, a random number x is generated such that $x<=n$ to select the x^{th} branch for execution. In this case, it is possible that a branch on any of the transitions on the current transition set gets executed. In the later case, the harness systematically executes the branches on the transitions and in the order that the transitions have been executed. For instance, if in a transition set the transitions t_1, t_2,.. t_n are executed and the constraints $t_1^{c_1}$, $t_1^{c_2}$, $t_2^{c_1}$,...,$t_n^{c_1}$ are collected initially, then the next path constraints are constructed by negating the constraints in the same order. We note that executing a branch on a transition may give rise to collecting new constraints due to nested conditionals. Therefore, always constraints on the same transition are selected for negation until all the branches on that transition are executed. We refer to these techniques *Random Concolic (RC)* and *Systematic Concolic (SC)*, respectively.

3.5 Behavior Preservation

The instrumentation in our approach preserves the original behavior of the model and only enables collecting path constraints, similar to other tools [16, 33, 60]. Since collecting and solving complex constraints take some time, we needed to consider this factor for models with timers to prevent impact on the behavior of such models. To this end, as mentioned, in our approach, solving the constraints is always carried out during the execution of an iteration transition, rather than at each state where the solving time may be long enough such that some timer on the next outgoing transitions may fire before the constraint solver generates the data. Using this technique, we did not observe any impact (introduced by the overhead of constraint solving) on the timed systems based on our experiments on several case study models (that we will present later). As another observation, solving the constraints is fairly quick (in the order of milliseconds), which is considerably less than the timer values that we observed on the case study models. Hence the tool (even without the technique above) would not interfere with the behavior of the models.

4 EXPERIMENTAL EVALUATION

4.1 Tool Implementation

We have implemented a prototype tool called *mCUTE* (Model-level Concolic Unit Testing Engine) to concolically execute state-based models using the approach that we presented. We have conducted several experiments using our tool on a number of industrial and academic models with different sizes and complexities. mCUTE (which is open source) was implemented partly in Java (static parts related to, e.g., model processing and model transformations) and C++ (the core libraries for concolic testing). We have integrated our tool in Papyrus-RT [14], which is an Eclipse-based open source software modeling IDE with an active user community. Our tool can be integrated to other Eclipse-based IDEs, including RSA-RTE and Rhapsody. The schematic design of our prototype tool is shown in Fig. 8 and various components of our design are explained below.

Model Transformer implements a set of libraries on top of Eclipse Modeling Framework (EMF) [61] to connect to a model and collect information on model elements, such as transitions, states, and action code. This information is collected statically and is stored in some vectors before testing the model. This module then uses this information to configure the test harness by taking the structure of the model into account. This module is also responsible for transforming the model and instrumenting its action code. The transformation is done using EMF libraries, and the action code instrumentation is conducted using CIL [47].

Model-level Concolic Engine module contains a test harness (configured by the model transformer module) integrated with our symbolic execution engine (which was built by extending CREST [16]). The harness implements algorithms for concolic testing, some libraries for measuring the model coverage (including states, transition, and action code coverage) as well as heuristics for transition and branch selections. This module initializes the harness with user-provided test configurations, keeps track of the path constraints collected, and feeds the harness with newly generated test cases. The module stores the coverage information for each transition in an object. The coverage information is updated after each iteration, so the module chooses the candidate branches for negation for a transition from its coverage object. Observe that to initiate testing, we generate code from both the model under test and its harness and we execute the generated code. The code generated from test harnesses varies slightly for each model, but a test harness was on average around 3K lines of C++ code.

Table 2: Our Case Study Models

Model	Model Complexity				
	Total# Transitions	Total# States	Total# Action code	Total# Branches	LOC of the Generated Code
AR	25	20	80	30	2.3K
FDM	55	55	144	34	6.2K
ACC	49	35	105	60	4.5K
CA	50	50	104	84	2.5K
PBX	291	284	1,715	196	11K
RSM	100	100	4,283	1,600	15.5K

4.2 The Benchmark Models

Our criteria for selecting the candidate case study models include: (i) the case studies should have a large number of transitions, action code, and complex branches in their action code, (ii) the case studies should include transitions that have dependencies over global variables. The first criterion above serves to evaluate the scalability of our tool and approach, since in these case studies, the tool may need to collect and solve large and complex path constraints. In addition, the large number of branches may give rise to numerous executions (what in symbolic execution may lead to the path explosion problem). The second criterion evaluates our tool and approach for its effectiveness in exploring branches whose predicate is highly dependent on previous constraints and consequently the effectiveness of the approach in finding bugs that may only be revealed if certain branches are executed. Below we will briefly introduce our case studies for this experiment.

Collision Avoidance (CA) that was explained in previous sections. *Autonomous Rover (AR)* [9] is a system that controls a vehicle equipped with four wheels driven by two engines and sensors to collect information from the environment and to detect obstacles. *Fabric Dyeing Machine (FDM)* [10] is a system to control a fabric dyeing machine. It was designed initially in ROOM [57]. We manually converted it to a behaviorally equivalent system in UML-RT. *Adaptive Cruise Control (ACC)* is a system that adjusts the vehicle speed and distance to that of a target vehicle and was originally designed in AutoFOCUS [2, 37]. *PBX (Private Branch eXchange)* is a system that models the interaction of a user with a private telephone network used within an organization. *Randomly Synthesized Model (RSM)* we also challenged our tool on a large model generated randomly. The model generator generates well-formed executable models, i.e., all the transitions have a trigger, and all states are reachable from the initial state. The tool generates well-formed executable pieces of C code as action code (similar to the program generator tool Csmith [44] that generates C code fragments for testing compilers) with multiple branch points over the transition input parameters. It took just more than 5 minutes for our model generator to generate this model (a fully instrumented and transformed model, which is ready for testing), and around 2 minutes for the code generator to generate code from the model. More information about all of our models can be found in Table 2.

4.3 The Experiment Setup

Coverage Methodology. We used state, transition, and branch coverage as widely understood and uncontroversial metrics to evaluate the effectiveness of mCUTE. The tool measures the coverage during testing and finally reports the coverage. Regarding branch coverage, we only consider feasible reachable branches (i.e., the branches that are indeed executable) and we do not count the number of branches in the library code. Even though mCUTE needs to successfully execute the library code in order to execute the model itself.

Different Types of Bugs on the Models. Models of RTE systems may define a tricky execution space that is built from many pointer operations, including casting, and numerous nested conditionals. This code must often process inputs received from other programs that may include network packets or system call parameters, which may cause run-time errors. We refer to these bugs as *Model Crashing Bugs*. On the other hand, some bugs do not crash the system but cause the system to generate wrong outputs, which are referred to as *Operation Bugs* [25]. These types of bugs are detected by analyzing the model's execution traces only. Prior work has proposed *property monitors* to catch these bugs [8]. Finally, mCUTE can catch code generator bugs, since mCUTE runs the code generated from the models. Some examples of these bugs caught by mCUTE are described in Section 4.4.

The Baseline for Comparison. We compare our approach for testing state machines with the two techniques presented in [8]. The first one (*Random Testing*) is a black box random test generation technique for state machines, that generates a large volume of test cases randomly and merely based on the combinations of various possible messages that a state machine can receive. The second one (*Simple Exploration)* is a white box technique, that generates the test cases by systematically exploring transitions of a state machine. In this technique, starting from the initial state, outgoing transitions in each state are explored, and the trigger for each transition is collected and is added as a test input to a vector. The process continues until the test case length criterion is met, or the current state has no more outgoing transitions. Both of these two techniques generate random data for input parameters bounded by user-defined ranges. The authors have shown that the action code coverage using the above two techniques is very low. In that experiment, these techniques are given a considerable amount of resources (in terms of the test generation time and the number of generated test cases), and yet they fail to discover some bugs in the action code. The main reason is that Random Testing blindly generates test sequences and both techniques generate random data for messages. Therefore, action code that is in nested conditionals or is on transitions that need a long sequence of messages in a specific order to execute, as well as transitions that are guarded by complex predicates, are challenging to execute. We will show how our approach using concolic testing can improve the state machine test coverage considerably. In the current experiment, we define a time frame as the testing budget. That is, all the three techniques (our concolic engine as well as the other two techniques) are executed during the allotted time budget and their performance is measured in terms of coverage and the discovered bugs. We performed our experiments using a computer equipped with a 3.0 GHz CPU and 8GB of RAM.

4.4 Results

4.4.1 Testing Coverage. Fig. 9 presents the performance of each technique in branch coverage. Based on the graph on the left, after around 17 minutes, *Systematic Concolic (SC)* covers nearly all

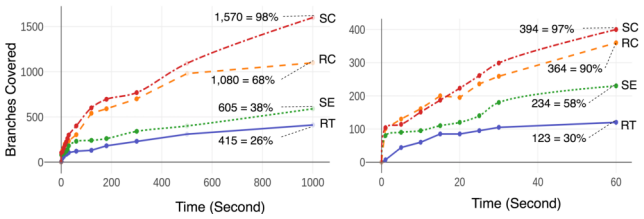

Figure 9: Left: Branch coverage for RSM model, right: coverage for other case studies. SC: Systematic Concolic, RC: Random Concolic, SE: Simple Exploration, RT: Random Testing

the branches in RSM model (1,600 branches). Besides, as the number of branches increases to around 1,000, SC starts to outperform *Random Concolic (RC)* considerably, even though both techniques are concolic. The reason is that in RSM there are a large number of transitions and branches, which gives rise to large transition sets and consequently collecting a large set of constraints. Now, as opposed to SC, RC negates branches randomly, and consequently many branches are executed multiple times. The plots also show that the performance of *Random Testing (RT)* and *Simple Exploration (SE)* is considerably lower than the other techniques, even though the results of SE as a random technique is promising. The graph on the right shows the results of testing other systems except for RSM (performance for the non-RSM systems was similar and aggregated because of the page limit), where in this case the number of branches are around 400, collectively. As the graph shows, SC and RC show almost the same performance. Observe that in this case the techniques do not collect large number of constraints during the execution of individual systems due to relatively less number of branches (as opposed to the case for RSM, where hundreds of constraints are collected during an execution from the initial state to a final state).

There was an interesting observation regarding the states and transitions coverage. The SE and RT techniques were able to achieve the same coverage as the concolic testing, but only in some models. For instance, in some cases, they were not successful in generating the required data to satisfy the transitions guards, and hence the generated test cases could only execute initial states and transitions. That is, final states were not reached in these cases.

Note that, given the fact that a state machine may encompass a large set of states, transitions and consequently action code, in practice, it might be hard to cover all possible executions of the state machine using concolic execution, due to numerous possible executions, which might affect the completeness of our approach or the approach may not be scalable to larger models. However, the action code in state machines is usually relatively small, but, as explained, much of the complexity of concolic testing state machines lies in the dependencies between action code of different transitions and therefore constructing path constraints that can exhibit their executions.

4.4.2 Bugs Found. We found several bugs in our case studies that are shown in Table 3 (the message parameters with curly braces represent non-primitive data, i.e., each value is a field in a *structure* data type). Concolic testing (both SC and RC) could catch these

Table 3: Bugs Found by mCUTE

Bug Description	Sample inputs to reproduce the bug
bug#1 :generated incorrect #define macros (FDM)	setup({1,100}), status(10)
bug#2 :index out of bounds (FDM)	create({2069,1204669}), create({37742,737742})
bug#3 :wrong output generated by state machine (CA)	lidarMsg(1,0), lidarMsg(949366,100)

bugs (after few minutes), SE could only catch the bug #2 (which did not need complex input messages and data) after around 10 minutes. RT could not catch any of these bugs. The bugs were confirmed by running the test cases generated by mCUTE on a raw version of the models (model is neither instrumented nor transformed).

Model Crashing Bugs. We found a bug in the FDM system. This system consists of several concurrent state machines, including *DyeingSystem* that dynamically instantiates new *dyeing units*, where each is dedicated to a dyeing task. The transition action code below is executed in a loop until *totalTasks* number of dyeing units are created (the transition's guard checks this). *DyeingUnits* is an array and its size is equal to *totalTasks*. The attribute *idx* is always incremented, whether the method *createDyeingUnit* creates an object or not (this method fails due to, e.g., type incompatibility or if there is not enough room in the capsule for a new instance [4]). Consequently, even though the *DyeinUnits* array has still room, the system crashes with an index out of bound error at line 7.

```
1 DyeingTask dyeingTask = (*(DyeingTask*)) msg->getParam(0);
2 bool valid = false;
3 if (dyeingTask->temprature>0 && dyeingTask->runTime>0){
4    void* du = createDyeingUnit (valid);
5    idx++;
6    if (valid){
7       DyeingUnits[idx] = du;
8    }
9 }
```

Operation Bugs. An example of this bug was explained in previous sections in the CA system. As explained, the CA must not generate a *vibrate* message and *reverse steering* messages at the same time (since it may distract the driver). As shown in Table 3, mCUTE could generate inputs for the system such that an execution violated this requirement. On this case the DFS strategy helped us to find the bug quicker. To this end, we chose an iteration length large enough (7 in this case) such that mCUTE could always reach transition *t5* and execute it (so no iteration before executing *t5*) and finally mCUTE could hit the bug.

Bugs in the Code Generator. If a state *S* has some incoming transitions, each with some parameters, the code generator should generate a macro using *#define* statement for each parameter (as shown below) such that all the transitions parameters are visible at the state *S*. In the FDM controller, we detected a bug: the code generator generated only one *define* statement (instead of two) at the state *Setup* that has two incoming transitions, one to receive other component's status and one to initiate a dying task using the parameter *RunningData*:

```
231 void Capsule_DyeingRunController::
232 entryaction_____Setup( const UMLRTMessage * msg ){
233    #define data (*(const RunningData *) msg->getParam(0))
234    /* UMLRTGEN-USERREGION-BEGIN */
```

```
235    cout << "temp:" << data.temprature->value << endl;
```

The harness triggers both incoming transitions to the state *Running* (in two iterations), once by sending a status (an *integer* value) to trigger the first transition and once by sending a dying task information (an instance of the struct *RunningData*) to trigger the second one. In both cases, the input parameters were cast to integer, which in the later case caused a run-time error at line 235.

5 RELATED WORK AND DISCUSSION

Program Concolic Testing Approaches. Most of the work in the literature address the development of concolic testing for programs [11, 12, 16, 24, 29, 32–35, 44, 58–60, 66]. Program concolic testing was originally implemented in DART [33] to automate concolic testing C programs with high coverage. Later CUTE [60] was introduced to support pointers as input parameters. CREST [16] was later developed with a set of heuristics to improve path coverage. There are also multiple research works on generating optimal search heuristics for a given program [21]. EXE [18] and KLEE [17] implemented the Execution-Generated Testing (EGT) [20] technique, which again mixes concrete and symbolic execution, but in a slightly different manner. The tool pre-processes the program to check whether there are functions with constant inputs, so the tool runs those functions concretely. This avoids unnecessary overheads that could otherwise be introduced by the symbolic execution [17, 20]. Other program concolic tools include Pex [63], where the tester writes parametrized unit tests (PUT) [64], and Pex generates test inputs for all feasible paths in the PUT. Then, the inputs are used to instantiate the PUTs in order to gain a set of unit tests that exhaustively test the program. In [44], the authors introduce a hybrid concolic testing technique that benefits from both random and concolic testing, where random testing is used to quickly put the system under test into particular states, which are otherwise expensive to perform using concolic testing (due to massive number of execution paths). When random testing saturates, the algorithm automatically switches back to concolic testing to perform exhaustive searching to find a new coverage point.

Concolic Testing of Models. The results of applying the tools mentioned above are promising in terms of increasing branch coverage and the chance of finding bugs in programs. However, the techniques and tools above are not enough to conduct testing on state machines that compared with programs have a different execution semantics and model, and different structure. Concolically testing the code generated from the models might seem like an option. However, for model-level testing and tracing the executions to model elements, the actual model is required. Moreover, to the best our knowledge, no tool for concolic execution of C++ (the language of the generated code) exists. Our approach leverages model information such that only the action code on transitions, which we assume to be in C, needs to be executed concolically. There are some techniques and tools for testing and analysing state machines that use either in-house [27, 53, 67] or off-the-shelf symbolic execution engines (such as Klee) [13, 40]. However, observe that, with symbolic execution, it is difficult to deal with, e.g., heap-based data, pointers, calls to library functions, and timers (in our concolic approach, the engine has access to runtime information such as timer status). Polyglot [13], for instance, translates the structure and behavior of the state machines (modeled in different semantics) into Java and tests them using Java Pathfinder [36, 65]. The authors in [51] propose a semi-automatic approach (since it requires a manual user annotation) for symbolic execution of concurrent process. However, concolic execution and testing models to generate test cases dynamically and automatically has not been studied before, and we think our technique and tool is the first of this kind. For instance, the technique proposed in [38] extracts event handlers from a Java application, takes as input a target line for execution in an event handler, and generates a set of input messages and data to reach that target. This technique is not automatic since one needs to specify a state machine that represents the implementations. The tool presented in [23] also extracts a state machine from programs, but for guiding the concolic execution to gain better coverage. Similarly, the technique presented in [45] extracts a state machine from a web application and test cases are derived from the state machine, but does not use symbolic or concolic execution. The technique presented in [28], proposes analyzing dependencies between transition guards, input parameters, and global variables to assign weights to each transition, such that transitions with greater weights have higher priority for execution, so the technique seems to improve coverage for state machines with high dependency between transitions only. The work in [54] presents a technique for concolic testing Simulink (which is mostly used in automatic control and digital signal processing domain than for modeling real-time applications) models. In both the recently mentioned techniques [28, 54], the authors did not consider action code on transitions and the complex data structures used as input parameters between interacting components. Models of real-time embedded system may have a fair amount of action code to control the behavior of a model and often include complex data structures for message exchanges.

Other Techniques. There are test generation techniques for state machines [25, 41, 42, 49, 50, 62] that work based on different coverage criteria. Compared to ours, these techniques do not use symbolic execution and do not consider the action code on the state machine, which can be the source of bugs in the system.

6 CONCLUSION AND FUTURE WORK

In this paper, we have described and implemented an approach for concolic testing of state machines. We demonstrated our approach and tool for exhaustive testing by conducting an empirical evaluation on a set of UML-RT models. Based on our results, concolic testing could increase the branch coverage and could find more bugs compared to other techniques including random testing. Our approach has some limitations. For instance, in the current implementation, we only support systematic testing of the transitions leaving a state, which means we execute all the outgoing transitions of a state in order. Moreover, we use random and depth-first techniques for action code branch negation and execution. Using a diverse set of heuristics for transition and branch selection might help to gain better coverage in models testing. For instance, the concolic engine might be able to rely, e.g., on the structure of the state machine or the dependencies between the model elements [10] to prioritize the transitions and the branch points and choose the ones with the highest priority. We intend to address these features in future work.

REFERENCES

[1] Model-based systems engineering design of an automobile collision avoidance system. https://www.isr.umd.edu/~austin/enes489p/projects2011a/CollisionAvoidance-FinalReport.pdf, 2011. Accessed: 2018-01-06.

[2] Autofocus. https://af3.fortiss.org/en/, 2016. Accessed: 2017-12-15.

[3] Rational Software Architect Realtime Edition. http://www-01.ibm.com/support/docview.wss?uid=swg24034299, 2016. Accessed: 2016-8-10.

[4] Modeling real-time applications in rsarte. https://www.ibm.com/developerworks/community/blogs/a8b06f94-c701-42e5-a15f-e86cf8a8f62e/resource/Documents/RSARTEConcepts.pdf, 2017. Accessed: 2018-03-06.

[5] eTrice. https://www.eclipse.org/etrice/, 2018. Accessed: 2019-01-10.

[6] RoseRT. https://www.ibm.com/support/knowledgecenter/en/SSSHUF_8.0.1/com.ibm.rational.testrt.studio.doc/topics/troseworking.htm, 2018. Accessed: 2019-01-10.

[7] RTist. https://www.hcltech.com/products-and-platforms/rtist, 2018. Accessed: 2019-01-10.

[8] Reza Ahmadi, Nicolas Hili, and Juergen Dingel. Property-aware unit testing of UML-RT models in the context of MDE. In *European Conference on Modelling Foundations and Applications*, pages 147–163. Springer, 2018.

[9] Reza Ahmadi, Nicolas Hili, Leo Jweda, Nondini Das, Suchita Ganesan, and Juergen Dingel. Run-time monitoring of a rover: MDE research with open source software and low-cost hardware. In *Open Source for Model-Driven Engineering (OSS4MDE'16)*, 2016.

[10] Reza Ahmadi, Ernesto Posse, and Juergen Dingel. Slicing UML-based Models of Real-time Embedded Systems. In *International Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, 2018.

[11] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 59. ACM, 2012.

[12] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Daniel Dig, Amit Paradkar, and Michael D Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Transactions on Software Engineering*, 36(4):474–494, 2010.

[13] Daniel Balasubramanian, Corina S Pasareanu, Michael W Whalen, Gábor Karsai, and Michael Lowry. Polyglot: modeling and analysis for multiple statechart formalisms. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 45–55. ACM, 2011.

[14] Francis Bordeleau and Edgard Fiallos. Model-based engineering: A new era based on Papyrus and open source tooling. In *OSS4MDE@ MoDELS*, pages 2–8, 2014.

[15] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the sixth Conference on Computer Systems*, pages 183–198. ACM, 2011.

[16] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *23rd IEEE/ACM International Conference on Automated Software Engineering, 2008. ASE 2008.*, pages 443–446. IEEE, 2008.

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

[18] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[19] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 1066–1071. IEEE, 2011.

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

[21] Sooyoung Cha, Seongjoon Hong, Junhee Lee, and Hakjoo Oh. Automatically generating search heuristics for concolic testing. In *Proceedings of the 40th International Conference on Software Engineering, ICSE, Gothenburg, Sweden*, 2018.

[22] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective symbolic execution. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, number CONF, 2009.

[23] Chia Yuan Cho, Domagoj Babic, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. MACE: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *USENIX Security Symposium*, volume 139, 2011.

[24] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet? *arXiv preprint arXiv:1503.07217*, 2015.

[25] Tsun S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, (3):178–187, 1978.

[26] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: A software testing service. *ACM SIGOPS Operating Systems Review*, 43(4):5–10, 2010.

[27] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG: A symbolic test generation tool. In *TACAS*, volume 2280, pages 470–475. Springer, 2002.

[28] Giuseppe Di Guglielmo, Masahiro Fujita, Franco Fummi, Graziano Pravadelli, and Stefano Soffia. Efsm-based model-driven approach to concolic testing of system-level design. In *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE), 2011*, pages 201–209. IEEE, 2011.

[29] Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. Con2colic testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 37–47. ACM, 2013.

[30] Madeleine Faugere, Thimothee Bourbeau, Robert De Simone, and Sebastien Gerard. Marte: Also an uml profile for modeling aadl applications. In *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, pages 359–364. IEEE, 2007.

[31] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.

[32] Patrice Godefroid. Compositional dynamic test generation. In *ACM Sigplan Notices*, volume 42, pages 47–54. ACM, 2007.

[33] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[34] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[35] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[36] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[37] Florian Hölzl and Martin Feilkas. 13 AutoFocus 3 - A Scientific Tool Prototype for Model-Based Development of Component-Based, Reactive, Distributed Systems. In *Dagstuhl Workshop on Model-Based Engineering of Embedded Real-Time Systems*, pages 317–322. Springer, 2007.

[38] Casper S Jensen, Mukul R Prasad, and Anders Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 67–77. ACM, 2013.

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

[40] Amal Khalil and Juergen Dingel. Incremental symbolic execution of evolving state machines. In *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on*, pages 14–23. IEEE, 2015.

[41] Young Gon Kim, Hyoung Seok Hong, Doo-Hwan Bae, and Sung Deok Cha. Test cases generation from uml state diagrams. *IEE Proceedings-Software*, 146(4):187–192, 1999.

[42] Young Gon Kim, Hyoung Seok Hong, Doo-Hwan Bae, and Sung Deok Cha. Test cases generation from uml state diagrams. *IEE Proceedings-Software*, 146(4):187–192, 1999.

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

[44] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *Proceedings of the 29th international conference on Software Engineering*, pages 416–426. IEEE Computer Society, 2007.

[45] Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. State-based testing of ajax web applications. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 121–130. IEEE, 2008.

[46] George H Mealy. A method for synthesizing sequential circuits. *Bell Labs Technical Journal*, 34(5):1045–1079, 1955.

[47] George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction*, pages 213–228. Springer, 2002.

[48] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *International Conference on the Unified Modeling Language*, pages 416–429. Springer, 1999.

[49] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Software testing, verification and reliability*, 13(1):25–53, 2003.

[50] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Software testing, verification and reliability*, 13(1):25–53, 2003.

[51] Luis Pedrosa, Ari Fogel, Nupur Kothari, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. Analyzing protocol implementations for interoperability. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 485–498, 2015.

[52] Ernesto Posse and Juergen Dingel. An executable formal semantics for uml-rt. *Software & Systems Modeling*, 15(1):179–217, 2016.

[53] Eric James Rapos and Juergen Dingel. Incremental test case generation for UML-RT models using symbolic execution. In *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST), 2012*, pages 962–963. IEEE, 2012.

[54] Manoranjan Satpathy, Anand Yeolekar, and S Ramesh. Randomized directed testing (REDIRECT) for Simulink/Stateflow models. In *Proceedings of the 8th ACM International Conference on Embedded Software*, pages 217–226. ACM, 2008.
[55] Douglas C Schmidt. Model-driven engineering. *Computer-IEEE Computer Society*, 39(2):25, 2006.
[56] Bran Selic. Using UML for modeling complex real-time systems. In *Languages, compilers, and tools for embedded systems*, pages 250–260. Springer, 1998.
[57] Bran Selic, Garth Gullekson, and Paul T Ward. *Real-time Object-Oriented Modeling*, volume 2. John Wiley & Sons New York, 1994.
[58] Koushik Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572. ACM, 2007.
[59] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *International Conference on Computer Aided Verification*, pages 419–423. Springer, 2006.
[60] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 263–272. ACM, 2005.
[61] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.
[62] Generating tests from UML specifications. Generating tests from uml specifications. *The Unified Modeling Language*, pages 76–76, 1999.
[63] Nikolai Tillmann and Jonathan De Halleux. Pex–white box test generation for .net. In *International Conference on Tests and Proofs*, pages 134–153. Springer, 2008.
[64] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 253–262. ACM, 2005.
[65] Willem Visser, Corina S Pasareanu, and Sarfraz Khurshid. Test input generation with java pathfinder. *ACM SIGSOFT Software Engineering Notes*, 29(4):97–107, 2004.
[66] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 249–260. ACM, 2008.
[67] 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.