Automated State-based Online Testing Real-time Embedded Software with RTEdge

Wafa Hasanain[1], Yvan Labiche[1] and Serban Gheorghe[2]
[1]*Systems and Computer Engineering, Carleton University, 1125 Colonel by Drive, Ottawa, Canada*
[2]*Edgewater Computer Systems Inc., Ottawa, Canada*

Keywords: State-based Testing, Online Testing, Real-time, Embedded, RTEdge.

Abstract: Verifying a real time embedded application is challenging since one has to consider timing requirements in addition to functional ones. During online state-based testing the generation and execution of test cases happen concurrently: test case generation uses information from a state-based test model in combination with observed execution behaviour. This paper describes a practical online testing algorithm that is implemented in the state-based modeling tool RTEdge. Two case studies show that our online testing algorithm produces a test suite that achieves high model coverage, thus facilitating the automated verification of real-time embedded software.

1 INTRODUCTION

Specifying, designing, constructing, and verifying software systems is a challenge and doing so for embedded, real-time software is even harder because one has to account for non-functional (time related) requirements in addition to functional ones. RTEdge is a collection of specification modeling, code generation, simulation and analysis tools that facilitate developing such software systems (http://www.edgewater.ca/software-solutions) as a set of communicating state machines, similarly to IBM's RSA Real-time Edition.

Our objective was to devise an automated test case generation procedure from an RTEdge model. Such test cases can be executed against different simulation settings or on the target (deployment) platform to evaluate impact of design decisions.

We report on a black-box, online, directed random test case generation procedure. It is black-box since we only rely on the specification to derive test inputs, but also since we only use a small portion of the specification: in fact we only use the specification of signals that trigger behaviour in the set of communicating state machines in the RTEdge model. It is online since we rely on RTEdge capabilities to simulate the model: test case creation occurs concurrently to the simulation of their execution. It is directed since we monitor progress of model coverage to decide whether to stop or continue test case construction. Our case studies show that our approach, although random, achieves very high levels of coverage of the RTEdge model. We also show how random test case generation is completed with formal verification to further increase coverage when necessary. This paper contributes to the field of (random) testing is different ways: This paper brings additional data to the on-going debate as to whether random testing is effective or not; We discuss how our approach has been integrated in a commercial CASE tool; We show our solution, even though simple, is effective at covering the model of two representative case studies; Our approach differs in several ways from related work; To the best of our knowledge, this is the first time random testing is applied the way we propose in this paper.

Section 2 discusses related work. Section 3 introduces the RTEdge platform we build upon and section 4 introduces our test framework. Section 5 introduces two case study systems and section 6 discusses results. Section 7 concludes the paper.

2 RELATED WORK

State-based testing is a vibrant field of research, as confirmed for instance by the number of tools that support one or more of the state-based testing activities (Shafique, 2013). It is not feasible in a conference paper to provide a complete picture of this

Hasanain W., Labiche Y. and Gheorghe S..
Automated State-based Online Testing Real-time Embedded Software with RTEdge.
DOI: 10.5220/0005243402940302
In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2015), pages 294-302
ISBN: 978-989-758-083-3

large field of research. We only point the reader to some useful general discussions while focussing on works that are directly related to ours.

Testing from a finite state machine (FSM), or a set of communicating FSMs (C-FSM), heavily depends on the behaviour specification the FSM contains. When the FSM does not have actions (or activities) on transitions or states, or guard conditions, then approaches exist to automatically generate test cases (Khalil, 2010, Mathur, 2008, Mouchawrab, 2011). A C-FSM can under some conditions, so as to avoid a state space explosion, be transformed into a larger extended FSM from which the abovementioned techniques can be used (Luo, 1994). Alternatively, techniques specific to C-FSMs exist, e.g., (Li, 2002). Other techniques involve symbolic execution, e.g., (Conformiq, 2006, Jin, 2011), or a meta-heuristic search, e.g., (Asoudeh, 2014, Guo, 2004). Alternatively, one can consider testing techniques for LTSs, e.g., (Tretmans, 2008). When the FSM has actions (or activities) and/or guards that are all linear, automated test case construction is also feasible: e.g., (Asoudeh, 2014, Duale, 2004, Kalaji, 2011, Larsen, 2005, Schwarzl, 2010, Utting, Vain, 2011). These techniques are typically offline since it is possible to analyse the model and create feasible test cases prior to executing them. There are exceptions, such as UPPAAL-TRON (Larsen, 2005), to online test using a timed automaton specification.

When actions (or activities) and guards are specified with a more complex language, offline testing is typically not possible since it is not possible to statically analyse both the state based behaviour and the complex Java/C/C++ pieces of code to create feasible test cases. Instead, online testing is necessary to simulate the model to identify the resulting state and therefore identify what can be the next event to send to the implementation/simulation of the state based behaviour.

Related work also pertains to random testing (Duran, 1984), that is the automated generation of test inputs from an input domain or an operational profile. Random testing is effective, sometimes surprisingly more so than other (structural) criteria (Duran, 1984, Arcuri, 2010). Adaptive random testing (Chen, 2005) has been proposed to improve random testing, although its real effectiveness has been put to question (Arcuri, 2011).

Other attempts to enhance random (white-box) testing have been proposed. In *directed random testing* (Godefroid, 2005), collecting information on executed paths is used to systematically direct the selection of new random inputs to lead execution to trigger new program paths. Our approach works similarly, though at the model level instead of the code. In *feedback random testing* (Pacheco, 2007), unit test for object-oriented classes are created as legal (according to contracts) sequences of method calls and execution results (feedback) in terms of violated contracts and execution outputs (e.g., returned values) is used to extend (with new method calls and new input data) of test cases. In *coverage rewarded random testing* (Groce, 2011), reinforcement learning is used to obtain interesting new test cases. The first two techniques have been combined to automatically generate tests from stateflow models (Satpathy, 2008), in an attempts to improve upon existing, purely random input selection techniques, e.g., Reactis (Cleaveland, 2008). This required that the state model be flattened and unfolded up to a pre-defined depth. We do not require that.

Our approach differs from these related works in one or more of the following. (1) We do not perform any analysis of the states and transitions to identify new inputs, and we do not perform transformation of the state model. We only use information about the signals that can be sent to the system, i.e., the signals the state-based behaviour can respond to. In other words, the state model (and not only the code) is a black-box. (2) Our test model is a set of communicating state machines, which also include pieces of C/C++ code for the specification of actions, activities and guard conditions: guards and actions are not assumed to be linear. (3) We focus on achieving a complete set of coverage objectives instead of one objective at a time, similarly to some recent white-box testing approach (Fraser, 2013). (4) During random selection, we uniformly sample from a domain without any attempt to improve over this simple random selection.

3 THE RTEdge PLATFORM

RTEdge is a Model Driven Development (MDD) platform, for designing critical real-time embedded applications (Gheorghe, 2011, Sarkar, 2010). RTEdge is built around a modeling subset of AADL (Feiler, 2012) and UML2 (Pender, 2003).

With RTEdge, one **s**pecifi**es** software as a set of state machines communicating through signals and ports (a.k.a., capsules), annotated with assertions (e.g., state invariants), constraints (e.g., guards), activities (C/C++ code), and expected temporal properties. The model responds to Independent System Inputs and Dependent System Inputs. An *Independent System Input* (ISI) is generated by the environment of execution of the software being de-

signed, independently from any behaviour specified in the RTEdge model. A *Dependent System Input* (DSI) is sent by the environment at the request of the software: i.e., the RTEdge model sends a signal to its environment and the environment is expected to respond with a DSI. An ISI is defined by time arrival constraints (typically a period) and may carry data (C/C++ struct). A DSI is also specified by a time arrival constraint (the model/application expects it to arrive within a specific amount of time after it sends the request to the execution environment). Also, RTEdge provides a Periodic Timer through which state machine execution can be triggered. The timer uses a Service to broadcast a timeout Signal to any capsules using the service.

Prior to any other activity (see below), one can verify the structural correctness of the RTEdge model: e.g., two capsules exchange the same set of signals; and one can check temporal correctness and resource utilization (schedulability). During this process, the Worse Computed Response Time (WCRT) is determined for each transaction, i.e., an execution triggered by an ISI that ends when no more internal transitions are to be triggered, when no more DSI is expected. With the RTEdge Formal Link feature one can automatically transform an RTEdge model into an equivalent Promela model (Holzmann, 2003) which can then be automatically run under the SPIN model checker (Holzmann, 2003) to verify safety and liveness properties. RTEdge can then interpret SPIN counter-examples as RTEdge model execution traces.

RTEdge also offers code generation, compiling, deployment, debugging capabilities.

The RTEdge Virtual Time Environment (VT) simulates the execution of an application automatically generated by RTEdge on a host/development machine rather than on the target platform. VT is especially helpful during verification since the user has precise control over the arrival of System Inputs (e.g., injection), has precise control over how time passes and offers the same capabilities as a debugger. VT allows the definition of call-back functions whereby one can register a user-defined function that will be called under a particular circumstance. Particularly interesting to us, call-back functions can be called when a transaction ends, a transaction starts, and an RTEdge periodic timer expires. Such functions can typically be used when creating the oracle, i.e., the piece of code that decides whether a test case passes or not.

4 A TEST FRAMEWORK FOR RTEdge

The objectives of our test framework were initially the following: (i) We wanted to derive test cases from an RTEdge model with a model-driven testing approach/algorithm as simple as possible; (ii) We wanted to rely on the existing features of RTEdge to the maximum extent possible for testing purposes.

Next, we first discuss the rationale for the main decisions that drove the design of our test framework. We then discuss the framework itself and its algorithms. More technical details can be found in the first author's thesis (Hasanain, 2013).

4.1 Selecting an Online Feedback Random Test Case Generation

Since an RTEdge model can become quite complex, with several communicating complex state machines, with complex C/C++ actions on transitions and activities on states, it appeared very quickly that devising a strategy that would analyse the model and allow us to generate adequate test suites made of executable test cases, according to standard selection criteria (Ammann, 2008, Lee, 1996), similarly to what is done in many other pieces of work (section 2) would be too expensive. Specifically we felt it would be too complex or even impossible to devise a test case generation strategy that would identify test inputs (i.e., signals) to ensure that some state model elements are reached. Instead, we decided to use only information about the ISIs and DSIs to create test cases: the details of the model, i.e., its communicating state machines, are not used as (primary) test objectives to drive the test case generation. In other words, we consider the tested software as a black-box, not only because we do not look at its implementation (the source code) to create test objectives and therefore test cases, but also because we do not look at the state model.

We create test objectives from the specification of ISIs and DSIs, that is, their timing information (e.g., period) and the data they carry. At this level of abstraction, there is however no clear relation between such specification and the behaviour actually triggered in communicating state machines: this triggered behaviour is in the state machines, which we do not use. Therefore, instead of using criteria (e.g., based on equivalence classes for the data carried by input signals) to derive test cases, we decided to rely on a random generation (section 4.2).

Since any random test case generation can run

forever unless we identify a stopping criterion, we nevertheless rely on coverage of model elements to decide whether the random generation needs to continue. We monitor model coverage as test cases are generated and executed and we stop when we have reach adequacy or when we do not observe significant coverage improvement (section 4.2).

In summary, we developed an online, directed random test case generation.

4.2 The Test Framework

Our test framework uses the ISI and DSI specifications of an RTEdge model as an input. Because of the time-related aspect of an RTEdge model, our framework is to automatically generate a test script that periodically and randomly sends the input signals to an executable version of the model, and observes the progress in terms of coverage of the model to decide when to stop the creation/execution of test cases. To do so, we rely on RTEdge's VT. (How this is actually realized is shortly discussed in section 4.3.)

Sending Independent System Inputs (ISIs). We send an ISI to the System Under Test (SUT) independently for each ISI, as follows. Since an ISI is periodic, we start by sending an instance of the ISI, randomly generating the data it carries (see below) at time 0 (zero). VT manages time and time 0 corresponds to the SUT being created and ready to respond (steady state). Then, at each time value equal to a multiple of the ISI's period, we send another ISI instance, again randomly generating it data.

Sending Dependent System Inputs (DSIs). A DSI is sent by the environment upon request by the SUT, only once, within a specific amount of time after the environment receives the request. For each request for a DSI, we send a DSI instance by randomly selecting a delay between the request arrival for this DSI and the sending of the DSI instance (randomly selected according to the delay specification of the DSI), randomly selecting data for the DSI (see below).

Generating Random Signal Data. We created a signal data random generator for signals that do carry data. This generator supports a subset of the types that RTEdge supports, specifically the types supported by SPIN plus character. For these primitive types, the generator randomly (uniform distribution) selects a value in the allowed range. RTEdge also supports more complex data types: arrays, enumerations, and structures (similar to C struct). To generate an array, the generator determines the data type of its elements, and generates the required data. For each element in a structure, the generator generates the required data depending on the element type. For an array or a structure, the generation is recursive. For an enumeration, the generator produces at random (uniform distribution) one of the possible values.

RTEdge allows the refinement of existing types by specifying a reduced allowed range of values, thanks to Data Range Constraints. Our generator uses those constraints to randomly (uniformly from the constrained range) generate values.

Stopping Criterion. Since a transaction includes everything the SUT has to do in response to an ISI, the end of a transaction is a good time to decide whether to stop or continue generating test cases. Each time the SUT finishes a transaction, it informs the test framework, providing details about the transaction: start and end times of the transaction, states and transitions covered by the transaction. (Note that VT allows us to stop the simulation of the model to collect that information, thereby avoiding any impact of the collection process on execution times, and therefore deadlines. The simulation thus remains representative of what would actually execute on the deployment platform.) We based our stopping criterion on the coverage achieved by test cases, using two standard criteria (Ammann, 2008): state coverage and transition coverage.

Ideally, the online testing procedure would stop when an adequate test suite has been generated, i.e., when 100% state and transition coverage is reached. This is however not a guarantee since our test case generation is black-box and random. We therefore need a stopping criterion in case the random generation fails to reach adequacy. One possibility could have been to ask the test engineer to decide of a coverage level to reach. But even then, except if one selects a trivial coverage level to reach, there would be no guarantee to actually reach it. One difficulty is that due to the random nature of the test case generation, coverage can increase for some transactions, it may appear to have come to a standstill for another transaction, and may increase again later on. Similarly to what is done in some genetic algorithms (Haupt, 1998), we decided to observe coverage over a user-specified number of transactions. If no additional coverage is observed during this window, then we stop. More formally we defined three flags: (1) `noNewCoverage` is true if no new state/transition coverage is observed during the observation window; (2) `stateTarget` is true if we reach adequacy for states; (3) `transTarget` is true if we reach adequacy for transitions.

Our random generation then stops if the

following condition is true: `(stateTarget and transTarget) or noNewCoverage`; i.e., when we reach adequacy or when no new coverage is observed during the observation window.

Our approach requires some parameters from the user: (1) The number of times a state/transition needs to be visited; (2) The size of the observation window (i.e., number of transactions) during which coverage improvements is monitored; (3) The minimum number of new states or transitions that one expects to be covered during that window.

Notice that we allow the test designer to alter the usual definition of coverage by using the first input parameter: a state (transition) is considered covered if and only if it has been visited a number of times that is at least equal to that input parameter. If the input parameter equals to one, this is the usual meaning of coverage. Requiring that a state (transition) be visited more than once to be considered covered is inspired by the notion of statistical software testing (Thévenod-Fosse, 1991).

4.3 Framework Realization with VT

We relied on RTEdge's VT's capabilities to send signals (SISs and DSIs) so the VT simulation consumes them, to collect coverage information, to collect transaction data.

Before entering into the details, it is important to realize that time in VT progresses in discrete steps: each time a transition in a capsule is triggered, time advances. Each time VT advances time, it is able to perform tasks not related to the simulation proper, such as reporting on various aspects of the simulation: e.g., what is the current state in each capsule, which transitions where last triggered. This is done without any impact on the simulation since that simulation is implicitly paused.

Sending Signals to the VT Simulation. Sending signals (ISI, DSI) to VT is as simple as putting the signals in a queue from where VT fetches the next signal to be consumed by the simulation. Each signal deposited in that queue is specified with an arrival time (according to the time maintained by VT) and data values it carries. This is a priority queue with signal time as a priority. This way, when VT advances time, it looks at the queue for a signal to consume at that time. If there is one, the signal is consumed. As discussed earlier, the sending of a ISI/DSI happens as long as the test case construction does not stop, i.e., the stopping criterion is not met: more signals are put in the queue as needed. Note also that in addition to fetching from the queue, VT also populates the queue: e.g., capsules (state machines) communicate through that queue.

Collecting Coverage Data. Using VT's API, each time VT advances time (see earlier discussion on that), it reports on the current state of each capsule as well as on the last triggered transition to the testing framework, which then maintains a counter for each state/transition to count the number of times each one is visited during the simulation.

Receiving Transaction Data. RTEdge provides callback functions that allow us to extend the functionalities of VT and make sure our testing framework is informed when specific events occur during the simulation, specifically, to get the start and end times of a transaction (i.e., an ISI is being consumed by the SUT, and leads to a completed transaction). The testing framework then calculates the duration of each transaction, and compare this duration with the Worst Computed Response Time (WCRT): if the duration value of a transaction is strictly greater than the WCRT of the ISI that triggered that transaction, as computed during schedulability analysis, then there is a fault in the model and the test case has failed. This may happen because the estimates of execution times which are typically used during schedulability analysis can be optimistic. At the end of the transaction, coverage information is also fetched from VT (see above).

The designer has the possibility to set up timers on capsules in order to detect lack of progress (i.e., lack of change of state, lack of change in behaviour) within the capsule. Timers are similar to periodic events and detect every so often whether progress is being made in a capsule. If this is not the case, a callback function informs the testing infrastructure, which reveals a liveliness problem.

5 CASE STUDIES

We briefly describe the specification of two case studies, and their design using RTEdge. More details can be found online in the first author's thesis (Hasanain, 2013). According to our industry partner, these are not trivial models when compared to models their clients manipulate. They also look representative of other models one can find in publications (e.g., (Kalaji, 2011)).

5.1 The Production Cell System

The Production Cell case study is a realistic industry application in the field of control systems (Lewerentz, 1995). It processes metal plates, which are conveyed to a table by a feed belt. A robot takes

each plate from the belt and places it in a press thanks to a retractable arm equipped with an electromagnet. The press forges the plate. A second robot arm takes the plate from the press and places it on the deposit belt. The production cell implementation contains 14 sensors and 13 actuators. Actuators are used to switch the motors on and off or change their directions, and sensors return value to the control program about the system state.

The specification of the production cell has three kinds of non-functional properties. The safety requirements are the most important: collisions between devices must not occur; plates must be dropped in the safe area, two consecutive plates must be transported at an adequate distance to avoid placing two plates in the press, a movability restriction is implemented to prevent any machine from moving further than what it is allowed. The second important requirement is the liveness of the system: each plate transported by the feed belt should eventually be forged and arrive to the end of the deposit belt. Third, the design of the Production Cell should be flexible and could be easily be modified to similar Production Cell.

We designed the production cell using RTEdge with five major capsules: feed belt, rotary table, robot, press, and deposit belt. (The model has 16 capsules, 138 states and 168 transitions.) We assumed movements of a plate in the Production Cell take time and, similarly to others (Burns, 1998), we assumed specific movements (e.g., travelling on the feed belt to the table) take fixed amounts of time, and different movements require different durations. For example, the forging of a plate in the press needs more time than moving the robot. We simplified the interactions between the controlling software we model and its environment made of sensors and actuators by replacing the sensors in the model with actions in the model, thereby simulating how time elapses during those movements. As a result, when a capsule in some particular state needs a sensor value to proceed with the simulation, then it will wait a specific amount of time in its current state before moving to the next state (where the sensor data is used), thereby simulating that there might be a delayed response by the sensor. For instance, the robot must read a sensor in order to bring its fist arm next to the press.

In the model, the feed belt (capsule) communicates with an external capsule to simulate the receipt of a new plate, and it communicates with the rotary table (capsule) to simulate a plate moving to the table. The press only communicates with the robot, which communicates with the elevating rotary table and the deposit belt. We designed the communicating state machines of the model (i.e., the capsules) such that certain safety requirements are enforced. Specifically, the capsules communications ensure that specific sequences of signals will be ignored. For instance, the table will not accept a signal specifying the arrival of a new plate if it is already holding a plate. To ensure a plate moves from one device to another, the corresponding capsules synchronize through signals.

To simulate how time passes as plates move around, we created transients states with attached activities that make time pass. We specified a transient state for each plate movement: e.g., a transient state simulating the movement of a plate from the beginning of the feed belt to its end.

5.2 The Elevator Control System

The Elevator Control system controls a configurable number (strictly greater than one) of elevators, responding to requests from users at various floors (configurable number) and within the elevators. It also controls the motion of the elevators between floors. In this case study, we assumed four floors and two elevators. The elevator system being a well-known system, often used as a case study, we do not dwelve too much on its specification.

We designed the elevator with 14 communicating capsules: 73 states, 86 transitions. One capsule concurrently controls the cabs movements between the floors of the building, receiving requests from another capsule, calculating each cab direction, then sending requests to make things move, and interacting with the user (e.g., lamps, floor buttons' light).

Our model has two ISIs, which are the floor button request and the hall button request. In real time, these system inputs are aperiodic; however, RTEdge only sends each ISI periodically according to a user-defined period. The period of an ISI must be equal to or greater than the WCRT of the transaction this ISI triggers. Therefore, in order to define a period for an ISI, we have first estimated the required time for our model to complete a transaction for the ISI, then we performed a schedulability analysis of the model, which returned the WCRT. We set the period of each ISI to the computed WCRT, which is six seconds for both ISI.

6 RESULTS

6.1 Production Cell

We used our framework to derive test cases from the Production Cell model. At the same time, since our framework can detect deviations from WCRT or some liveliness problems, the framework participates in the verification of the model. Recall that our framework requires three different inputs. In a first experiment, we set those inputs as follows: The number of times a state/transition needs to be visited to be considered covered is set to one, the observation window is set to three, and the minimum number of new states/transitions that needs to be covered in the observation window is set to one. The automated test case generation created a test case with one transaction, i.e., one ISI (i.e., one plate) that covered each state and each transition at least once. The stated coverage goal was achieved, there was no need for additional transactions, no need to observe coverage progress over an observation window (recall the stopping criterion).

In a second experiment, we kept the values of the last two parameters and required that each state and each transition be visited at least three times to be considered covered. The intent was to study the performance of our approach on a more demanding objective. We generated one test case involving three ISIs (i.e., three plates). All transactions passed, indicating that the duration of each transaction was found to be smaller than the WCRT computed by the schedulability analysis and no timeout was reported.

The first transaction covers five (new) states and no (new) transition: transitions outgoing from initial states in capsules are covered but not counted. Since the state coverage goal is not achieved, the stopping criterion is not true and test case construction proceeds with a second ISI (i.e., plate): eight new states are covered and no (new) transition is covered. Test case construction therefore continued with a third ISI: 125 new states are covered and 168 transitions are covered. Each state/transition was covered at least three times, resulting in the test case construction to stop.

6.2 Elevator

We proceeded similarly to the Production Cell case study. We set inputs as follows: The number of times a state/transition needs to be exercised to be considered covered is set to one, the observation window (i.e., number of transactions) for studying coverage progress is set to two, and the minimum number of new states/transitions that need to be covered in the observation window is set to one.

Our test generation procedure created one test case with five ISI instances with randomly generated buttons and directions: The first ISI comes from the hall buttons with the following randomly generated data (Direction = 1, Floor number = 2); The second ISI comes from an elevator button (CabID = 0, Floor number = 4); The third ISI comes from the hall buttons (Direction = 1, Floor number = 1); The fourth ISI comes from an hall button (CabID = 1, Floor number = 3); The fifth ISI comes from the hall buttons (Direction = 0, Floor number = 3).

All transactions passed. At the end of the test, the coverage objective was not met: 69 (94.5%) states coverage and 78 (90.7%) transitions coverage while the stopping criterion was true (no coverage progress over the observation window with ISI number 4 and 5).

We studied the remaining uncovered states (four) and transitions (eight) and identified that these would be exercised in case of emergency situations with the elevators; such situations were not triggered during the five ISI test case.

One capability we gave our test framework (Hasanain, 2013), and that we did not discuss previously in this paper due to lack of space, is that, thanks to the mapping from an RTEdge model to Promela we use SPIN to give us test cases that will exercise the states/transitions that are missed by the random generation, following already established procedures to benefit from both testing and formal methods (e.g., (Fraser, 2009)). For each uncovered state and transition, our framework automatically defines an LTL property that states that it is never possible to reach this test purpose. Such a property is often called a trap property (Gargantini, 1999). Assuming the test model is correct, i.e., it is indeed possible to reach this test purpose, and SPIN can handle the complexity of this test model, then SPIN will be able to find a counterexample showing how that test purpose can be fulfilled: this counterexample is then a test case achieving that test purpose. Using this automated procedure, all states and transitions uncovered during random testing where exercised, resulting in an overall state and transition adequate test suite.

7 CONCLUSIONS

In the domain of embedded, time critical, real-time systems, assurance of the system meeting its timing requirements as well as functional requirements is

key. To facilitate the verification of such systems, in the context of a Model-Driven Development based on RTEdge (the tool created by our sponsor), we developed a black-box, online, directed random test case generation procedure. We experimented with this procedure on two well-known case studies and showed that we can effectively reach very demanding coverage levels of the model at random.

We conjecture that such impressive results might in part be due to some structural characteristics of our models, which is worth further investigations. In case this conjecture is confirmed and additional experiments show we almost reach coverage goals on other case studies with different structural characteristics, we do not feel overly concerned. Indeed, thanks to the mapping from an RTEdge model to Promela we showed we can use SPIN to give us test cases that will exercise the states/transitions that are missed by the random generation. Using Promela and SPIN only would not be economical to achieve the same level of coverage. However, combining testing and a formal method would be economical, as advocated by others (e.g., (Fraser, 2009)). Future work should also investigate the effectiveness at finding faults of the generated tests.

REFERENCES

P. Ammann and J. Offutt, *Introduction to Software Testing*, Cambridge University Press, 2008.

A. Arcuri and L. Briand, "Adaptive random testing: an illusion of effectiveness?" *Proc. ACM ISSTA*, 2011.

A. Arcuri, M. Z. Iqbal and L. Briand, "Formal analysis of the effectiveness and predictability of random testing," *Proc. ACM ISSTA*, 2010.

N. Asoudeh and Y. Labiche, "Multi-objective construction of an entire adequate test suite for an EFSM," IEEE ISSRE, 2014.

A. Burns, "How to Verify a Safe Real-Time System The Application of Model Checking and a Timed Automata to the Production Cell Case Study," *Real-Time Systems Journal*, 24 (2), 1998.

T. Y. Chen, H. Leung and I. K. Mak., "Adaptive random testing," Proc. *Asian Computing Science Conf.*, 2005.

R. Cleaveland, S. A. Smolka and S. T. Sims, "An Instrumentation-Based Approach to Controller Model Validation," Proc. *Automotive Soft. Workshop*. 2008.

Conformiq, TTCN-3, Qtronic and SIP: The Model-Based Testing of a Protocol Stack, a TTCN-3 Integrated Approach, http://www.verifysoft.com/ttcn-3_qtronic_sip.pdf, [Last checked: May 2014]

A. Y. Duale and M. Ü. Uyar, "A method enabling feasible conformance test sequence generation for EFSM models," *IEEE Trans. on Computers,* 53 (5), 2004.

J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE TSE*, SE-10 (4), 1984.

P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL*, Addison-Wesley, 2012.

G. Fraser and A. Arcuri, "Whole Test Suite Generation," *IEEE TSE*, 39 (2), 2013.

G. Fraser, F. Wotawa and P. E. Ammann, "Testing with model checkers: a survey," *STVR*, 19 (3), 2009.

A. Gargantini and C. Heitmeyer, "Using Model Checking to Generate Tests from Requirements Specifications," *Proc. European Soft. Eng. Conf.*, 1999.

S. Gheorghe, "Integrating Formal Model Checking with the RTEdge™ AADL Microkernel," *SAE Internaional Journal of Aerospace*, 4 (2), 2011.

P. Godefroid, N. Klarlund and K. Sen, "DART: directed automated random testing," *Proc. ACM PLDI*, 2005.

A. Groce, "Coverage rewarded: Test input generation via adaptation-based programming," *Proc. ASE*, 2011.

Q. Guo, R. Hierons, M. Harman, K. Derderian, "Computing Unique Input/Output Sequences Using Genetic Algorithms," *Proc. FATES*, 2004.

W. Hasanain, *Verifying Real-Time Embedded Software by Means of Automated State-based Online Testing and the SPIN Model Checker—Application to RTEdge Models*, Carleton University, 2013. https://curve.carleton.ca/system/files/theses/27490.pdf.

R. L. Haupt and S. E. Haupt, *Practical Genetic Algorithms*, Wiley, 1998.

G. J. Holzmann, *The SPIN Model checker*, Addison-Wesley, 2003.

X. Jin, G. Ciardo, T.-H. Kim and Y. Zhao., "Symbolic verification and test generation for a network of communicating FSMs.," *Proc. ATVA*, 2011.

A. S. Kalaji, R. M. Hierons and S. Swift, "An integrated search-based approach for automatic testing from extended finite state machine (EFSM) models," *IST*, 53 (12), 2011.

M. Khalil and Y. Labiche, "On the Round Trip Path Testing Strategy," IEEE ISSRE, 2010.

K. G. Larsen, M. Mikucionis, B. Nielsen and A. Skou., "Testing real-time embedded software using UPPAAL-TRON: an industrial case study.," *Proc. ACM EMSOFT*, 2005.

D. Y. Lee and M. Yannakakis, "Principles and methods of testing finite state machines-a survey," *Proc. of the IEEE*, 84 (8), 1996.

C. Lewerentz and T. Lindner (Ed.), *Formal Development of Reactive Systems: Case Study Production Cell*, LNCS, 1995.

J. J. Li and W. E. Wong, "Automatic test generation from communicating extended finite state machine (CEFSM)-based models," *Proc. IEEE ISORC,* 2002.

G. Luo, G. V. Bochmann and A. Petrenko, "Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method," IEEE TSE, 20 (2), 1994.

A. P. Mathur, *Foundations of Software Testing*, Pearson, 2008.

S. Mouchawrab, L.C. Briand, Y. Labiche, M. Di Penta, "Assessing, Comparing, and Combining State Machine-Based Testing and Structural Testing: A Series of Experiments," IEEE TSE, 37(2), 2011.

C. Pacheco, S. K. Lahiri, M. D. Ernst and T. Ball., "Feedback-directed random test generation," *Proc. ACM/IEEE ICSE*, 2007.

T. Pender, *UML bible*, Wiley, 2003.

R. Sarkar, "Proof-Based Engineering of Real-Time Applications: An RTEdge™ Case Study," *SAE Int. Journal of Aerospace*, 3 (1), 2010.

M. Satpathy, A. Yeolekar and S. Ramesh, "Randomized directed testing (redirect) for simulink/stateflow models," in *Proc. ACM EMSOFT,* 2008.

C. Schwarzl and B. Peischl, "Test Sequence Generation from Communicating UML State Charts: An Industrial Application of Symbolic Transition Systems," *Proc. IEEE QSIC*, 2010.

M. Shafique and Y. Labiche, "A systematic review of state-based test tools," *STTT*, 2013.

P. Thévenod-Fosse and H. Waeselynck, "An investigation of statistical software testing," *STVR*, 1 (2), 1991.

J. Tretmans, "Model based testing with labelled transition systems," *LNCS*, 2008.

M. Utting and B. Legeard, *Practical Model-based testing*, Morgan Kaufmann.

J. Vain, A. Kull, M. Kääramees, M. Markvardt and K. Raiend, in J. Zander, I. Schiferdecker, and P. Mosterman, Eds., *Model-Based Testing for Embedded Systems,* CRC Press, 2011.