SimEvo: A Toolset for Simulink Test Evolution & Maintenance

Eric J. Rapos
Department of Computer Science & Software Engineering
Miami University, Oxford, OH, USA
Email: rapose@miamioh.edu

James R. Cordy
School of Computing
Queen's University, Kingston, ON, Canada
Email: cordy@cs.queensu.ca

***Abstract*—As Simulink models evolve and change during development, test evolution and maintenance can often be overlooked. SimEvo provides a toolset to assist Simulink developers in co-evolving test harnesses and test cases alongside their source models. Primarily a collection of testing tools, SimEvo combines the impact analysis features of the SimPact impact analysis tool to identify instances of necessary test case changes and potentially affected blocks, with the SimTH test harness generator to automatically determine if changes need to be made to the test harness model, and automatically generate a new one if necessary. This paper examines the implementation of SimTH, its integration with SimPact into the workbench SimEvo, and an overall analysis of the contributions of the toolset.**

I. Introduction

Testing can be an expensive process which takes considerable resources for a software project [2]. Testing includes requirements reviews, test case design, test creation, and test execution. In the automotive domain Simulink modelling has been increasingly gaining traction in recent years. Given the rapid evolution and high quality standards of these systems, consistent test maintenance has become a growing concern. Thus tools to assist in the automation of software test evolution are critical for the future.

In this work we attempt to address that issue with a toolset for Simulink test evolution which includes SimPact, an existing tool for impact analysis [11], as well as SimTH, a new tool specifically designed to automatically generate and update Simulink test harnesses. From these components the SimEvo toolset was created, which combines SimPact, SimTH, and smooth interaction with existing development and test tools in use by our industry partners.

By automating the usually manual process of test harness maintenance, savings in time and effort can be observed throughout the testing process. Previously our industry partners' engineers developed and maintained test harness models using a set of template files and informal conventions that allowed manual creation and updating of tests when changes were made to the source model.

The testing community, as well as our industry partners, see automation of the test harness generation and evolutionary update process as a desirable goal for a number of reasons. From the testing community perspective, automation allows for effort to be spent in other areas of the software process, freeing up valuable resources. From the industry standpoint, the in-house testing tool used by our collaborators has recently been named the recipient of a company wide testing tool sweepstakes, garnering it additional attention by many other groups. Along with this additional attention comes an increased demand for support and functionality. Reducing the test harness maintenance effort would allow them to spend more time on other tasks.

Thus we are motivated by both the strong industrial demand for an automated test harness generator in our industry partners' environment, and by the the opportunity to explore test evolution automation in general in the automotive context.

In this research we make the following contributions:

- A test harness generator for Simulink models.
- Industrial validation of our test harness generator.
- The incorporation of new and existing tools into a test evolution and maintenance toolset.

II. Background and Related Work

A. Testing Simulink Automotive Models

In order to understand the contributions of SimEvo it is important to understand the testing process used by our industrial partners to test their automotive models, and how the test models are conventionally created and the tests run. While this particular process is specific to their environment, it is also typical of Simulink model testing in general.

In the context of our industrial partners, testing of a Simulink model involves execution of the model on a series of timed inputs over a set interval. Timed outputs are compared against oracle values for the expected outputs. Thus, to test a model two artifacts are needed: (i) a test harness capable accepting input values and monitoring output values, and (ii) a set of timed test inputs and expected outputs over a time interval. (This represents one test case. There many be many for the same model.) Specification of the test case values themselves is currently a manual process, and the automation of their maintenance is part of our future work. An example test harness model, like one that would be used by our partners, can be found in Figure 1.

Test cases are specified as spreadsheets in Excel workbooks created manually by domain experts responsible for creating functional tests for the models. Each spreadsheet in a workbook represents exactly one test case, while the workbook as a whole represents the entire test suite. In each tab, there are

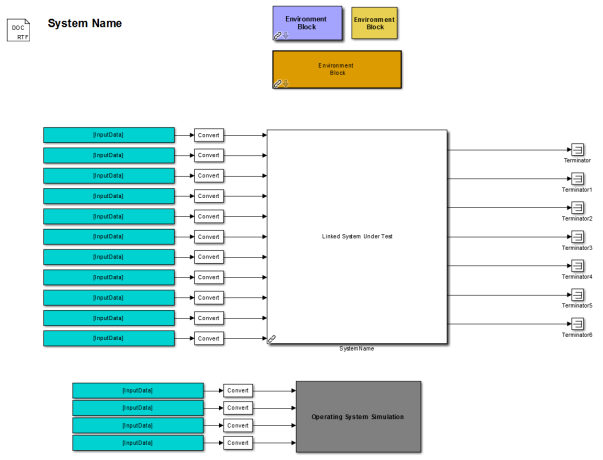

Fig. 1. Sample Test Model, with Environment and Operating System Simulation Blocks

column headers specifying the input signal names, simulation trigger names, output names, with the leftmost column always indicating the time in seconds. Each row represents a single time step and the values at that given time.

With all of the artifacts prepared (test harness model and test suite workbook), the testing tool used by our industry partners is able to run a simulation of the model. The simulation is run once for each test case in the test suite, using the specified time steps and input values, and monitors the outputs to make comparisons against the expected outputs. For each test case the tool produces a results report which contains details about the performance of the tool in the tests, normally as differences between expected and actual outputs.

B. Test Harness Generation

Simulink [4] models are behavioural models based on the flow of inputs through a series of blocks. The blocks perform calculations based on, and make changes to, the input signals provided to the model from external sources.

A test harness is designed to adapt a piece of software such that it can be executed with test values and the results observed for comparison. A test harness supports automatic execution [1]. The next section will detail how this process occurs as a part of our industry partner's software process, and relates it to other approaches to the creation and automatic generation of test harnesses, highlighting how SimTH differs.

Rocha and Martins introduce a model-based method for test harness generation for component testing [12]. Their approach generates a test harness which executes source code rather than a source model. The harness itself is model-based, using UML activity diagrams. It converts the activity diagrams (through a number of intermediate representations) to a test harness as source code in the programming language of the system under test. SimTH differs from this in that it produces a test harness in the native modeling language of the source model.

Okika et. al. present work on the creation of test harnesses for legacy software, using the example of an embedded system [7], which is similar to the goal of SimTH. Their work focuses on the creation of a test harness that will work for the control software for any provided test. This differs from SimTH, which generates a custom test harness for each Simulink model, created automatically given a model as input.

A specific subsection of test case generation work that is worth investigation is the domain specificity of test case generation for Simulink models, which is a fairly recent line of research. For example, Peranandam et. al. present an integrated test generation tool for enhanced coverage of Simulink and Stateflow models [8]. Their work is specific to the application domain, but deals with test case generation as opposed to the generation of harnesses to facilitate testing. The focus of their generated tests is on coverage, while the tests used in our experiments are functional tests based on requirements. Mohalik et. al. present similar work, using model checking to automatically generate test cases for Simulink/Stateflow [6]. These approaches can be thought of as complementary to SimTH, aiming at test case generation rather than automatic generation of test harnesses to execute the tests.

SimTH falls in the intersection of these two areas of related work, addressing specifically the absence of tools designed for model-based test harness generation. While these examples of related work involve model-based test case generation and test harness generation for source code, SimTH aims at model-based test harness generation for Simulink models.

C. Test Co-Evolution

The idea of ensuring tests evolve correctly and consistently alongside their source (whether that be code or models) is known as test co-evolution. Co-evolution is one of the primary goals of SimEvo, to ensure that tests can evolve in response to changes to the model as easily as possible by providing the tool support to do so.

Zaidman et al. [13] present a very comprehensive look at co-evolving tests and production software, from a number of different perspectives. They look at this topic from three views: (i) change history, (ii) growth history, and (iii) test evolution coverage. These views were validated using two open source cases (Checkstyle and ArgoUML) and one industrial case (a project by the Software Improvement Group).

One interesting approach to the concept of test evolution, specifically test case repair, is presented by Daniel et al. [3]. The authors use symbolic execution to repair existing test cases. The authors previously created ReAssert, which was capable of automatically repairing broken unit tests, however they must lack complex control flow and operations on expected values. In this paper they propose **symbolic test repair**, a technique which can overcome some of these limitations through the use of symbolic execution.

The work of Pinto et. al. [9] is aimed at understanding the myths and realities of test-suite evolution. In particular, they investigate why tests change over time. The authors state that test repair (fixing tests that no longer work after a change) is only one of many ways tests can evolve. In fact most changes occur as refactorings or additions and deletions of test cases. One example of the automation of test evolution is the work

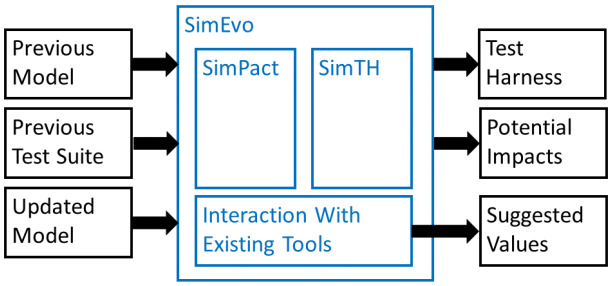

Fig. 2. SimEvo Architecture

of Mirzaaghaei et. al., in which they are able to automatically repair test cases for evolving method declarations [5].

III. SimEvo Architecture

SimEvo serves as a container for other tools and functionality, and as such is best presented by looking at each of its components individually. SimEvo consists of three main contributions: SimTH (test harness generation), SimPact (impact analysis), and its ability to interact with other industry testing tools. Figure 2 provides an overview of the architecture of the SimEvo tool.

A. SimTH

To produce a working test harness for an automotive model, SimTH requires as input only the Simulink behavioural model developed by the industrial engineer. Since the Simulink model to be tested is the only input required, SimTH does not place any new requirements on the developers – an important feature when introducing tools into an existing industrial toolchain.

This section describes the internal logic of SimTH, walking through the steps it uses to create test harness models. There are four main tasks in test harness creation: inclusion of the system under test, input and output management, environment simulation, and standardization. We discuss these four steps in general, and then their specific implementation in SimTH.

After these four steps are done, a final test harness is produced meeting all of the requirements of our industry partner. Each of the four steps maps directly to some component (or set of components) of the completed test harness. Figure 3 overlays the steps on the sample test harness model.

1) Step 1: Inclusion of the System Under Test: In order to test a specific system, a test harness needs an awareness of the system under test (SUT). Thus, the first step in creating a functional test harness is the incorporation of the model's functions into the test harness model by including the SUT components in the test harness.

To implement this in SimTH, we use the Simulink functionality for library linking. This feature allows a model to import the contents of another model, subsystem or block as a library link. Simulink testing is normally done at the subsystem level, and thus the SUT is normally a Simulink subsystem block.

SimTH investigates which library model is linked to the SUT subsystem and creates a link to the same library in the newly created test model. As a result the only element in the test harness is the linked library block. From the linked block,

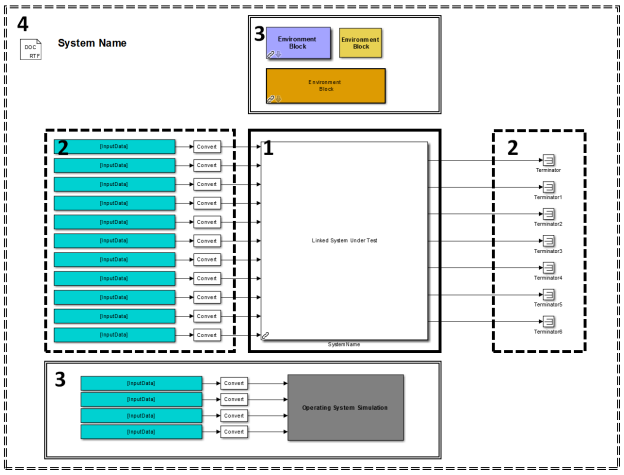

Fig. 3. Implementation Steps for Test Harness Generation, Resulting in the Complete Generated Test Harness

SimTH is able to identify the key parameters of the model, and copy any necessary values into memory for later use, such as simulation time step and solver options.

2) Step 2: Input and Output Management: The next major step is the management of the inputs and outputs to the SUT. As inputs and outputs play a central role in the testing of a system, it is important that the generated harness appropriately handle the inputs from test case files in order to provide them to the corresponding *inports* in the model. It is equally important that the test harness be able to monitor for correct output values and further comparison.

To implement input and output management, SimTH examines the interface of the subsystem block(s) being tested. Each block's interior has a number of *inports* and *outports* (Simulink terminology for inputs and outputs) which translate to labels on the model block itself. Traditionally inputs are on the left edge and outputs on the right, but this is not required.

SimTH begins by iterating over each input one at a time to integrate it into the test harness. Each *inport* has a designated signal name and type; SimTH relies on the signal names conforming to the conventions set by our industry partner to obtain the relevant information. SimTH pulls the relevant information from the SUT library block and creates a special block designed to read values from the workspace, which simulates receiving values from an external input source. This block is known as a *FromWorkspace* block and is parameterized by identifying that it will read a particular input value (using its name) and the timestep value. This process is repeated for all of the inputs to the subsystem being tested.

Some inputs may require additional processing before they can be provided to the model. These include conversions from floating point to fixed point values, as well as conversions for enumerated types. In such cases SimTH is able to infer the required conversions based on types and naming conventions, and generates the necessary conversion blocks between the *FromWorkspace* block and the subsystem being tested.

The outputs of the behavioural model are used to provide

inputs to other blocks outside the scope of the subsystem being tested. For testing purposes, output values are used for comparison to expected values, which is done by the testing tool observing the test simulation. To manage this SimTH links each output of the subsystem being tested to a special block known as a *terminator*, which ends the signal at that point. Examples of sending signals to a terminator block are shown on the right hand side of Figure 1.

3) Step 3: Environment Simulation: One of the main contributions of a test harness is the ability to simulate the environment in which the model will eventually be run without actually having access to that environment. This is done by simulating standard calls from the environment and providing accurate responses to calls to the simulated environment. Ensuring an accurate simulation of the environment by the test harness is of extreme importance in testing the SUT. There are two parts to this implementation in SimTH: the simulation of vehicle operating system (OS) calls, and the inclusion of various other required environment variables and blocks.

The majority of the logic of SimTH is responsible for creating an OS simulation subsystem capable of simulating the necessary parts of the OS for the vehicle. A given subsystem in a vehicle needs to interact with the operating system in various ways during execution, such as receiving signals from predefined flags. Since in the test harness the subsystem is tested in isolation, it needs some representation of the OS capable of simulating all of the required functionality.

The OS simulation is used mainly to simulate triggered events in the OS, which are invoked based on inputs from the test case file. When the flag to trigger an event is set in the test input file, the OS simulation block behaves in a manner consistent with the OS, which allows the rest of the system to react as it would under normal operating conditions. This is imperative in obtaining realistic test results.

An example Operating System simulation block can be seen in the sample test model in Figure 1, below the subsystem being tested. It has a number of simulated inputs similar to the main SUT, but these are to simulate interactions with the OS rather than outside inputs.

The number of different OS calls that can be made from models for the particular portion of automotive software developed by our partners is relatively limited. This means that the functionality for each type of call can be implemented directly in SimTH, such that when a specific type of call is required, SimTH can automatically generate the required blocks in the OS simulation subsystem to handle them. There are three main categories of OS simulations that SimTH must create in the OS simulation subsystem: periodic events (occurring every clock cycle), triggered events (caused by OS events), and initialization events (occurring on startup and after any reinitializations). This step in developing SimTH posed several difficulties when run on systems representing corner cases where the typical clock cycles did not apply, or multiple overlapping clocks were necessary. These were addressed by implementing special OS simulations targeted at these cases.

An internal view of a sample generated OS simulation

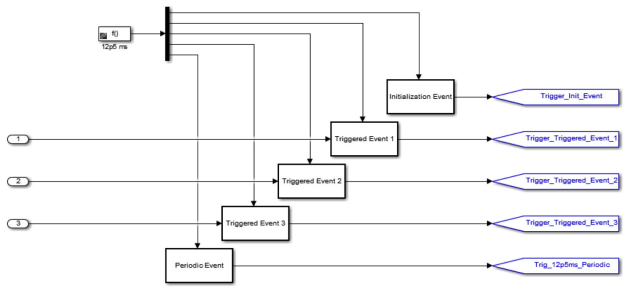

Fig. 4. Sample Operating System Simulation Block (internal view)

subsystem can be seen in Figure 4. This OS simulation block has 3 triggered events in addition to the initialization and periodic events, all for a 12.5ms time step.

In addition to OS simulation, SimTH is also responsible for the inclusion of a number of additional blocks and settings to create the execution environment for the model. There are three types of blocks that are added (2 mandatory, 1 optional) to the top level of the test harness, along with the setting of several parameters. Once these are included we have a fully functional test harness model.

The last part of setting up the environment is setting the model parameters for the test harness based on the values obtained from the SUT's linked library. This includes setting the step time and solver options for the model, such that they are consistent and will not be the source of any discrepancies between actual and expected results. These parameters are set automatically by SimTH based on available information.

4) Step 4: Standardization: The final step in test harness generation focuses on producing a result (in this case the test harness model itself) that conforms to existing standards for testing within the organization. This is to ensure that beyond the functionality covered by the previous sections, the generated test harness is recognizable and familiar to developers.

The main concern is that SimTH-generated test harness models should be visually similar to those developed manually from the template files currently used by our industry partners. This means that the layout needs to be familiar and easy to understand, blocks need to be the same colours and sizes as expected, and labeling and naming conventions must be consistent with current practice. These issues are important from a human interaction standpoint, and while they do not impact functionality, they assist in the smooth and continued maintenance of the software, one of the key goals of SimTH. To address this issue, SimTH is capable of producing test harness models that are visually identical to those manually generated, aside from an additional label at the top level that indicates that it is a generated test model. Since even the manually created files are based on templates, it is rather straightforward to implement SimTH to reproduce the same layout and coloring choices.

B. Including SimPact

The second main component of SimEvo is the integration of an existing tool used for maintaining evolving Simulink

models: SimPact [11]. Since SimEvo's main goal is the support of test maintenance for evolving Simulink models, one of the biggest issues is the impact of changes made to models on test cases and test harnesses. Including a tool aimed at identifying and mitigating the impact of these changes helps guide the use of SimTH and builds a stronger, more comprehensive toolset for our industrial partners.

C. Integration with Existing Test Tools

A major goal of SimEvo is the ability to interact with existing testing tools already developed and used by our industry partner. The use case for this interaction is based on some of the described future work of our previous impact analysis work [11]. When provided with a list of potentially impacted inputs and outputs following changes to a model, developers are left with two options: manual inspection or generation of suggestions. SimEvo's interaction with existing tools handles the second option.

The in-house testing platform previously developed by our industry partners takes a test harness and test input values to simulate and generate expected outputs for the provided inputs. When SimPact identifies the potential need for updates to tests affected by changes to the model, this simulation can provide the developer with candidates for possible updated test values. By making method calls to the test harness (existing or generated using SimTH) along with the existing input test values, the testing tool simulates the test model with these values, observing and recording the new outputs. This entire sequence of events is hosted in SimEvo, providing a single consistent platform for test maintenance as the models evolve.

IV. Validation of SimTH

A tool such as SimTH is used to assist and improve the software process by automating tasks that were previously done manually – as such, it is difficult to measure improvement in terms of performance when the methods are so dissimilar. One metric of success for SimTH is the ability to produce test models that are functionally equivalent to those developed by hand. While there is no benchmark to measure against, we determined it would be interesting to measure the time required for automatic generation of test harness models using SimTH, as a potential benchmark for future work.

To validate SimTH, we perform two experiments: (i) A timed execution of SimTH test harness generation, which provides execution time results, as well as whether or not a test harness could be produced at all for a chosen model set, and (ii) A correctness validation, in which the test models generated by SimTH are run with the testing tool to determine if they produce the same results as those generated manually.

A. Experimental Design

Here we present the two validation experiments in detail.

1) Harness Generation Ability and Efficiency: To test SimTH's ability to generate test harnesses for the complete set of our industrial partners' models, we automated calls to SimTH over the entire model set in succession, documenting the success or failure of SimTH to create an output model. The results were then summarized and presented to include the number of successful and unsuccessful generations. In addition to the ability to generate the test harnesses, we recorded the generation time for each. This is done primarily as a baseline for future improvements, as any automated process will certainly be faster than the current manual process.

2) Harness Correctness in Test Execution: The focus of this experiment is to test whether the automatically generated test harnesses from SimTH produce the same results as the original manually produced test models when run with the same test suites. If the results are the same, we consider that the test harness generated by SimTH is functionally equivalent. This experiment was run on the current final release of all of the models in our partners' model set. Although we have access to several generations of previous versions of the models, we chose this subset because it contains an instance of every model currently still in use, and because it covers all of the types of models in the set.

The result of this experiment is a pass/fail rating for the each automatically generated test harness from SimTH. If the testing tool determined that all of the test cases in the test suite passed, then SimTH received a pass. If any test case failed, SimTH received a fail for that test harness generation.

B. Model Test Set

The models used for this experiment are the same models used in our co-evolution experiment [10], representing several releases of the production models for an entire automotive subsystem. This entire model set, consisting of 457 models in total, is used for validation of test harness generation and performance measures, while subset of models and tests for the current final release, consisting of 45 models, was used to validate the correctness of the generated test harnesses.

C. Results

For the first experiment, testing the ability to generate a test harness for each of the 457 unique model files, SimTH was able to successfully generate a test harness for 423 of the 457 models, or 92.6%. This result is very positive overall. Several reasons were identified for the cases where a test harness was unable to be generated, including multiple time rates and complexity of integration models. Discussions with our industrial partners revealed that these types of harnesses would require special attention, and their current absence from SimTH functionality is not a major issue. In terms of performance, the average execution time for SimTH to generate a test harness was 2.3 seconds, with a maximum of less than 7 seconds.

The second experiment designed to measure the ability to produce the same testing results as the manually produced test harnesses. This experiement was conducted on the 45 test suites corresponding to the 45 test harnesses SimTH was able to generate for the models in the current final release. Of these 45 models, 39 successfully behaved the same (based on testing tool behaviour) as the manually produced test harnesses

provided by our industrial partner, a success rate of 86.7%. These results are not as strong as we would have liked, but are still very promising. SimTH automates a previously manual process, and produces correct test models a high percentage of the time.

We investigated thos instances where SimTH's generated harness did not perform identically to the manual test harnesses, and it appears that the reason for the inconsistency is easily spotted by manual inspection of the top level blocks of the harness (there are potentially extra conversion blocks, missing enumerated type references, etc.). Our industry partner assures us that it is common practice to review test harnesses prior to use, and a notice has been added to SimTH to remind engineers to perform this check on generated harnesses.

Overall, the 92.6% rate of generating test harnesses for the entire set of models, and the 86.7% correctness rate provide us with promising first results for improvements in the testing process using automation.

D. Additional Validation

Following our validation experiments, our industry partners have conducted experiments using SimTH on a much larger set of models, which for confidentiality reasons we are unable to present. However, these internal tests have been beneficial in providing a further measure of the success of SimTH. While we cannot discuss specifics, our industry partners have indicated that SimTH is working for most of the much larger set of models they have run it on.

V. Conclusions

We have presented SimEvo, a toolset for Simulink test evolution and maintenance. SimEvo is a combination of the existing tool SimPact [11], our new tool SimTH, and a set of interactions with other test tools used by our industry partners.

The primary focus of this paper, beyond the compilation of existing tools into a useful toolset, is the creation and validation of SimTH as an effective test harness generation tool for Simulink models in industry. SimTH is capable of taking a source model as input and determining exactly which elements are required for a test harness to test that model, and automatically generating that harness. SimTH was validated on an industrial set of models and was capable of generating test harnesses for 92.6% of the entire set of models, with an 86.7% correctness rate. While a higher rate is desirable, we have identified the cause of most of the issues, which were largely due to changes in naming conventions for timing of events. A standardization has since been introduced to avoid this problem, and in-house validation by our partners has observed a significantly higher success rate.

In the future, we would like to expand on the types of tools included in SimEvo. Currently containing SimPact and SimTH, there is certainly room to expand functionality to include test value generation, and other tools to support evolving test cases. Additionally, we would like to perform additional validation experiments on newer models from our partners in order to verify the success of SimTH on updated models conforming to stricter modeling conventions, which will assist in our automation.

Acknowledgments

The authors would like to thank our industry partners for their continued support in providing valuable feedback, testing our tool in their environment, and assisting in adapting SimTH to work on a larger scale. Specifically we would like to thank Costantino Rotella and Daniel Nowocien for their time and assistance. This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), as part of the NECSIS Automotive Research Partnership with General Motors, IBM Canada and Malina Software Corp., and by the Ontario Ministry of Research, Innovation and Science through an Ontario Research Excellence grant.

References

[1] B. Beckert, R. Hähnle, and P.H. Schmitt. *Verification of object-oriented software: The KeY approach*. Springer-Verlag, 2007.

[2] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *Proceedings of the 2007 Future of Software Engineering*, FOSE '07, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[3] B. Daniel, T. Gvero, and D. Marinov. On test repair using symbolic execution. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, page 207218, New York, NY, USA, 2010. ACM.

[4] Mathworks. MathWorks Simulink Product Page. http://www.mathworks.com/products/simulink/. Accessed: 2015-10-27.

[5] M. Mirzaaghaei, F. Pastore, and M. Pezze. Automatically repairing test cases for evolving method declarations. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM 2010)*, pages 1–5. IEEE, 2010.

[6] S. Mohalik, A.A. Gadkari, A. Yeolekar, K.C. Shashidhar, and S. Ramesh. Automatic test case generation from Simulink/Stateflow models using model checking. *Software Testing, Verification and Reliability*, 24(2):155–180, 2014.

[7] J.C. Okika, A.P. Ravn, Z. Liu, and L. Siddalingaiah. Developing a ttcn-3 test harness for legacy software. In *Proceedings of the 2006 International Workshop on Automation of Software Test*, AST '06, pages 104–110, New York, NY, USA, 2006. ACM.

[8] P. Peranandam, S. Raviram, M. Satpathy, A. Yeolekar, A. Gadkari, and S. Ramesh. An integrated test generation tool for enhanced coverage of Simulink/Stateflow models. In *Proceedings of the 2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 308–311. IEEE, 2012.

[9] L.S. Pinto, S. Sinha, and A. Orso. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 33. ACM, 2012.

[10] E.J. Rapos and J.R. Cordy. Examining the co-evolution relationship between Simulink models and their test cases. In *Proceedings of the 8th International Workshop on Modeling in Software Engineering*, MiSE '16, pages 34–40, New York, NY, USA, 2016. ACM.

[11] E.J. Rapos and J.R. Cordy. SimPact: Impact analysis for simulink models. In *Proceedings of the 33rd International Conference on Software Maintenance and Evolution*, ICSME '17. IEEE Press, 2017.

[12] C.R. Rocha and E. Martins. A method for model based test harness generation for component testing. *Journal of the Brazilian Computer Society*, 14(1):7–23, 2008.

[13] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.