Supporting Simulink Model Management

by

Eric J. Rapos

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
February 2017

Abstract

With the increasing use of Simulink modeling in embedded system development, there comes a need for effective techniques and tools to support managing these models and their related artifacts. Because maintenance of models, like source code, makes up such a large portion of the cost and effort of the system as a whole, it is increasingly important to ensure that the process of managing models is as simple, intuitive and efficient as possible.

By examining the co-evolution patterns of Simulink models and their respective test cases (a useful modeling artifact), it is possible to gain an understanding of how these systems evolve over time, and what the impact of changes to a model are on the relevant test cases. This analysis uncovered opportunities to present useful findings to developers in order to effectively manage model changes. By tracing the impact of a change to a Simulink model block on both the surrounding blocks and the tests associated with the model, developers can ensure that changes are accurately propagated, and can avoid changes that would lead to inconsistencies.

To support the model management process, three tools have been produced, each addressing a different aspect of the maintenance process: SimPact is used to identify and highlight the impact of changes to model blocks on tests and the rest of the model, SimTH automatically generates test harnesses for Simulink models, and

SimEvo combines these tools into a comprehensive evolution support package, with the ability to interface with existing industry tools. Each of these tools has been evaluated against a large industrial model set, and some are already in current use in industry, demonstrating their effectiveness and applicability to real world problems.

Acknowledgments

Where to start? The five years that have went into this thesis have both flown by and seemed like an eternity, but it is an experience I wouldn't trade for anything. When it comes to providing the thanks for the support, there is no way two pages in my thesis can cover it, but it's a start. Those I would like to thank fall into three categories, those who have provided the academic support to make it through, those who have provided the personal support, and those who made the research possible through funding and collaboration.

From the academic perspective, I would have been at a loss without the guidance of my supervisor, Jim Cordy. Jim has truly been an amazing supervisor, and I aspire to become even half the mentor he has been to me. Thank you for all of your feedback, support, and friendship throughout this process. The lessons learned that fall outside of this thesis will last a life time. Beyond this, I also owe thanks to a number of others who have helped out along the way: Juergen Dingel and Mohammed Zulkernine - members of my supervisory committee who have given me valuable feedback, and were sounding boards for initial proposals and ideas along the way, Manar Alalfi and Tom Dean for their guidance and feedback for NECSIS related projects that were worked on with others from the lab, Scott Grant who provided guidance as my CREATE mentor, but also a friend and welcomes distraction, and lastly, the rest of the School of Computing faculty and staff - while I cannot name them all, it is important to note that this department has been a home for me for so long, and the support network here has been nothing short of amazing.

Then comes the personal support; a PhD is a long process and I needed to rely on those around me, often more than I would have liked. To start off, Jenn, you have been there for me during my journey through the ups and downs, and I cannot thank you enough; you provide me with the inspiration to dream big. Beyond relationships, friends also played a vital role in my successes, so thank you Benjamin Cecchetto, Bahram Kouhestani, Jesse Burstyn, Doug Martin, and so many others for making the ride a little easier. I also want to thank my parents for their ongoing support; it can't be easy answering your friends' questions about ”When is he going to finish school and get a real job?”, so for that, I am sorry. You two have always been there for me, and this PhD is no different.

Lastly, I would like to acknowledge the support and contributions that made this research possible. First and foremost is the support provided by General Motors, and in particular Costantino Rotella, my supervisor during my internship at GM. While GM provided funding through NECSIS, he played a vital role in identifying areas of work and the ongoing relationship that made it possible. My work was funded through a number of sources: the NSERC CREATE Fellowship, the NECSIS Automotive Research Network (a partnership with General Motors, IBM, and Malina Software), the Ontario Graduate Scholarship, and NSERC grants.

This is all a long winded way of saying two simple words; so for those of you looking for the TL;DR version, I give you:

Thank you!

Statement of Originality

I hereby certify that all of the work described in this thesis is the original work of the author and doctoral supervisor. Any published (or unpublished) ideas and/or techniques from the work of others are fully acknowledged in accordance with the standard referencing practices.

Portions of Chapters 3 & 4 have previously been published as a conference paper[57], with some work from Chapter 2 taken from my research proposal.

Eric James Rapos

April, 2017

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

With the continued increase in model-driven engineering (MDE) in industry, there comes an increase in the various types of software models and related artifacts (model versions, testing harnesses, test cases, etc.). With a larger set of artifacts to maintain and ensure consistency, the field of model management has become prevalent. Model management refers to the practices associated with managing the complexities that come from software modeling. Since MDE relies heavily on models as a primary artifact, and the fact that models bring with them a large number of new issues such as meta-models, model-based tests, model transformations, and multiple types of related models, it becomes a field of research to keep track of these artifacts, and ensure that they remain consistent over time.

This thesis aims to improve the ability of industrial engineers to manage Simulink models and related artifacts, specifically testing artifacts, in order to ensure a more effective and streamlined software development process.

Based on initial conversations with, along with a summer internship at, our industry partner, it became evident there was a need for tools to support processes that had been manually performed. There was specific interest in tools to determine necessary changes in test case values, as well as to automate some of the more labour intensive model and test generation tasks. In addition to being desired in industry, these posed relevant and interesting research problems, leading to work in supporting Simulink model management, specifically through impact analysis to determine changes in test cases, and the automation of test harness generation.

The ability to work with real world problems, validate with real data, and provide support in a growing industry which is adopting and adapting to new technologies played a crucial role in the conduct of this research.

1.2 Thesis Statement

Through understanding the common co-evolution patterns between Simulink models and tests, it is possible to identify instances of interest in model management. By applying impact analysis to evolving models, it is possible to identify necessary changes in existing test cases in order to maintain consistency across artifacts. These observations, combined with additional tool support, can contribute to the ongoing effort to support industrial Simulink model management.

1.3 Contributions

This thesis makes the following contributions:

- An understanding of how models traditionally evolve in the application domain (Chapter 4: Co-Evolution of Simulink Models and Tests)
- A method for determining potential impact of changes in models on test cases (Chapter 5: Simulink Impact Analysis for Evolving Models)
- An implementation of impact analysis in a static and dynamic analysis tool, capable of identifying potential changes directly in test case files (Section 6.1: SimPact: Simulink Impact Analysis)
- A Test Harness Generator for Matlab Simulink Automotive models (Section 6.2: SimTH: Simulink Test Harness Generation)
- An evolution support tool which combines other tools and interfaces with existing industry tools (Chapter 6: Simulink Evolution Tool Support)
- Industrial verification/validation of the above using real-world automotive models provided by our industry partner (Chapter 7: Validation)

1.4 Outline

Chapter 2 presents relevant background information and related works, to situate this thesis in the literature. Following that, Chapter 3 presents a high level overview of the research conducted. The remainder of the thesis outlines the contributions in detail, broken into five chapters: Chapter 4 addresses the co-evolution relationship between industrial Simulink models and their tests, Chapter 5 addresses the identification of changes over model versions, and isolation of their impact on test values, Chapter 6 addresses the industrial tool which implements the impact analysis as well as other evolution support features, Chapter 7 presents validation experiments and results, and Chapter 8 presents the conclusions of our work and some potential expansions into future work.

Chapter 2

Background and Related Work

This chapter serves two purposes: to provide relevant background information sufficient to understand the work completed, and to compare with recent related work to provide justification for the uniqueness and usefulness of the work completed.

2.1 Background

This section on background knowledge covers work in the following areas:

- model-based testing
- model-based test evolution
- model-management

2.1.1 Model-Based Testing

In the field of model-driven engineering (MDE), the process of developing executable tests from source models is known as model-based testing (MBT). The concept relies on the models containing enough relevant information to be a reliable source to either manually create or automatically generate test cases for the system under test. These tests can be run on the models themselves as a simulation, or on generated code as a test for model/code conformance.

This subsection describes the two higher level concepts that contribute to MBT (software modeling & and software testing), and then present in detail the overlap of the two and its importance to the research.

Software Modeling

Whether is it a state machine, a domain-specific model, or a class diagram, software modeling is seen as a method of abstract representation of a software system. For the purpose of this background research, the general definition provided in the UML reference manual of "A model is a representation in a certain medium of something in the same or another medium." [10] is used as a baseline.

While modeling of software has existed in some form for some time, the real emergence of software modeling came with the development and and release of the Unified Modeling Language, or UML [10]. The reference manual provides a more than thorough background on software modeling, and the use of UML, thus only a small amount of detail is presented here; more effort is devoted to exploring model-based testing (one of the combined concepts) which demonstrates many of the properties of general software modeling. One of the major uses of UML is the modeling of software architecture, through the use of class diagrams, class dependencies, and other artifacts. An in-depth look at using UML for this purpose was presented by Medvidovic et al. [41].

One of the important advances that has emerged out of the use of software modeling is an entirely new design and development process. A text by Hassan Gomaa [26] presents this process in a clear manner, using UML and use cases to further explain software modeling.

The one field in software modeling of most relevance to this research is the concept of model comparison. The ability to compare two (or more) models, and determine their differences, is extremely useful for this work. A recent comprehensive survey was completed by Stephan et al. [63] which outlines a number of different techniques and technologies for comparing models, as well as an evaluation of these techniques.

Software Testing

Software testing is an extremely important part of the software development process, taking up approximately 50% of the time and more than 50% of the costs of development [47]. Software testing can be considered "a process, or a series of processes, designed to make sure computer code does what it was designed to do and that is does not do anything unintended..", which can be simplified to the definition: "Testing is the process of executing a program with the intent of finding errors." [47].

In 1979 Glenford J. Myers presented a comprehensive look at the Art of Software Testing, a text which has since been updated [47], in which he presents a high level look at what software testing really is, among a number of more detailed topics: inspections, walkthroughs, design of test cases, unit testing, testing methodologies, higher-order testing, debugging, and several others. This text has stood the test of time and presents a very clear overview of the topic.

A more recent, and more in-depth text was published by Ammann and Offutt [3] which presents the topic in more detail, with a wider range of topics. The text discusses the concept of coverage criteria (graph, logic, input, and syntax based), as well as applications of coverage criteria. A section devoted to testing tools explains how to develop tools for testing specific pieces of software. And finally, the text concludes with a chapter on the challenges of software testing. The only mention of model-based testing in the entire text is a reference in the last paragraph, citing it as a 'more recent approach', which shows that MBT is a new, and not widely explored field, especially in regards to software testing as a practice.

A third text by William Perry [51] rounds out the collection of texts to help understand software testing. Differing from the other two, Perry aims specifically at providing methodology for effective testing, through the whole process, including the capabilities of the testing teams. The text looks heavily at the building of a software testing environment, both physical and technological, before actually developing a testing process. The text then deals with selecting tools and processes appropriate for the task at hand. The text can be considered a step-by-step guide to testing, especially through its presentation of the seven step testing process:

1. Organizing for Testing
2. Develop the Testing Plan
3. Verification Testing
4. Validation Testing
5. Analyzing and Reporting Test Results
6. Acceptance and Operational Testing
7. Post-Implementation Analysis

With a basic understanding of software testing, it would not be a far stretch to assume testing is a code specific process, and one that analyzes source code specifically. However, this is something that is obviously untrue, due to the emergence of model-based testing. In addition to this is the realization that tests can be generated from other artifacts, such as software requirements [64]. Furthermore, another common misconception is that testing is based solely on some sort of coverage method (which are thoroughly covered in Ammann and Offutt's text [3]), however this too would be untrue; another common type of testing deals with constraint solving to generate tests. For example Vorobyov and Krishnan present their work on combining constraint solving with static analysis to generate tests [67].

Model-Based Testing

The combination of software modeling and software testing lies in the field of model-based testing (MBT), a field with a large amount of background that is helpful in understanding the MBT applications of the research presented in this thesis.

El-Far and Whittaker [22] present a book chapter which serves as an early (2001) introduction to the area of model-based testing, providing background and motivation for the early work, implementation details, benefits and drawbacks, and a number of other interesting insights. It begins the topic of MBT by looking specifically at models, and what they are. As stated, "Simply put, a model of software is a description of its behavior." They then proceed to discuss what models aim to do, as well as common types of models. From there they discuss how models are used in software testing. The authors discuss a number of different types of models used in software testing, and provide examples of usefulness and implementations. The models discussed are: finite state machines, statecharts, grammars, and Markov chains. From here, the authors present a list of main tasks for any MBT project:

1. Understand the System
2. Choose the Model
3. Build the Model

4. Generate the Tests
5. Run the Tests
6. Collect the Results
7. Make use of Test Results

Another book chapter on MBT by Baker et al. [6] presents an excellent overview of the subject. They ascertain that abstractions of systems are indeed models, and in the simplest terms, were the beginnings of model-based testing. The chapter then goes on to introduce the UML Testing Profile (UTP) and its role in the software development process. When discussing traditional testing, the authors present two models of testing that are accepted in the testing community, the V-model and the W-model, both of which stress the need for early feedback in the development process. The authors cite the ease of communication between the customer and the project team as one of the major benefits to model-based testing, as it allows for use cases and tests to be designed at a level of abstraction familiar to the customer, but usable by the project team. Furthermore, the authors provide a deeper background of relevant testing techniques, such as black-box and white-box testing, and provide details as to their use in MBT. As part of this, they introduce the problems faced in coverage criteria for model based testing, as it is not as clear to determine as source code coverage. The last area of software testing that is explored is the area of automatic test generation and how it is handled in MBT. The authors look specifically at sequences of transitions between states in FSMs as well as labeled transitions systems, abstract state machines, and Petri nets. The authors conclude with the statement that there is no universal approach for the automatic test generation from UML models, highlighting the need for further exploration of this area.

There are a number of general papers on MBT, one of which deals with the use of UML for system testing by Briand et al. [12]. An approach to derive system test cases directly from the UML models is presented. More specifically, they aim to derive tests from use case diagrams, use case descriptions, interaction diagrams, and class diagrams; the main focus being on the non-functional requirements. The work they present is part of a larger project called TOTEM (Testing Object-orienTed systEms with the unified Modeling language). Another paper about using UML state diagrams to generate tests was published by Kim et al. [31]. They identify control flow by converting UML state diagrams into extended finite state machines (EFSMs), then converting the EFSMs into flow graphs to obtain data flow, which can be used to generate tests.

Beyond these general approaches used in MBT, there has been significant work on the different approaches to MBT. One work in particular by Benjamin et al. [8] relates to coverage driven test generation. This paper looks at bridging the gap between formal verification and simulation, through the use of a hybrid technique. There were two overall goals of the work: to develop a method of verification that bridges this gap, and to perform a quantitative comparison of this methodology with existing simulation based verification techniques. The concept of a coverage driven test generator is introduced as a program that finds paths through a finite state machine model of the design, with the goal of satisfying each goal in the model; each path found is then considered to be an abstract test, which is then used to generate concrete tests. An intermediate representation, known as a test specification is used. The tool developed by the authors, GOTCHA (Generator Of Test Cases for Hardware Architectures), is presented as a prototype coverage driven test generator, which although it is still a functional model checker, is extended to support the generation of abstract test specifications based on a state or transition coverage model.

To automate MBT a large number of approaches have been presented over the years. Early work was done by Dick and Faivre on automating generation and sequencing tests from model-based specifications [21]. Next came work on the automation of deriving tests from UML statecharts, work presented by Briand et al. [11]. This was closely followed by work by Tretmans et al. [65] in which they present their tool TorX, which is used for automating MBT, specifically for specification based testing using formal methods. Based on a number of approaches, Pretschner et al. present an evaluation of MBT and its automation [55]. The overall goal of the study was to address four questions of the automation of model-based testing:

1. How does the quality of model-based tests compare to traditional hand-crafted tests?
2. How does the quality of hand-crafted tests compare to automatically generated tests?
3. How do model and implementation coverages relate?
4. What is the relationship between condition/decision (C/D) coverage and failure detection?

One slight variation on MBT is the concept of model-driven testing, which Javed et al. [30] define as "a form of model-based testing that uses model transformation technology using models, their meta-models and a set of transformation rules." Their approach makes use of several well-known tools/techniques/technologies, to achieve their goal of generating tests using the model transformation technology of model-driven architecture (MDA), based on platform independent models of the system. The authors make use of the Eclipse Modeling Framework (EMF) to model the elements, Tefkat to aid in the model-to-model transformations, MOFScript to aid in the model-to-text transformations, and the use of the xUnit family (including JUnit and SUnit) as testing frameworks. Using these technologies, the authors present their methodology to implement generation of tests suites based on platform independent models.

With a good understanding of MBT, it is also important to provide literature showing its use in the application domain of automotive software. Bringmann and Kramer present exactly this; a paper on MBT in automotive systems [16]. This work is of particular interest as they make use of MATLAB/Simulink models in their work. The authors present their test tool TPT (abbreviation of Time Partition Testing), which "masters the complexity of model-based testing in the automotive domain". The authors frame their work by describing the growing trend of automotive systems containing more and more software (estimated over 90% in the next decade). From there they go on to motivate the use of MBT for its ease of understandability in such an interdisciplinary field, stating that it improves communication in, and between, different levels of design. The paper looks deep into the requirements for automotive MBT, and summarizes them as: test automation, portability between integration levels, systematic test case design, readability, reactive testing/closed loop testing, real-time issues and continuous signals, and testing with continuous signals. Each of these aspects is inspected in closer detail in the paper. The following are the goals of TPT:

1. To support a test modeling technique that allows the systematic selection of test cases,
2. To facilitate a precise, formal, portable, but simple representation of test cases for model-based automotive developments, and thereby
3. To provide an infrastructure for automated test execution and automated test assessments even for real-time environments. This is important for hardware-in-the-loop tests, for example.

The paper then presents a case study to better illustrate how TPT works, dealing with an Exterior Headlight Controller (EHLC). TPT is used by Daimler for their interior production-vehicle products (which are all model-based).

With regards to the testing work done in Simulink [16], this work looks specifically at the testing of systems, while this thesis aims to explore how these types of tests evolve. However the use of Simulink models for their work serves as an indication of applicability of the work done for this thesis, and confirms much of the motivation to work with automotive software.

2.1.2 Model-Based Test Evolution

With an understanding of model-based testing, the next important area of background is the study of how model-based tests evolve. Since this thesis aims to support model management for Simulink models, which includes managing test models and test cases as they evolve, an understanding of model-based test evolution is extremely useful.

It is first important to discuss the evolution of software in a general sense. As such, the following four areas are explored: software evolution, test evolution, model evolution, and evolution of model-based tests.

Software Evolution

As with any product, software will change over time. This process is known as software evolution. Research in the area of software evolution looks at supporting the evolution process, as well as analyzing data to improve evolution processes.

As with the previous subsections, a text on the subject, this one by Mens and Demeyer [42], is most useful in providing sufficient background information. While the book is not an introductory text, it provides, by way of examples and applications, a breadth of knowledge in software evolution to adequately prepare the informed reader. The first chapter in particular presents the history and challenges of software evolution, which are of particular interest for this research.

The term software maintenance is often tightly coupled with software evolution, due to the fact that as software evolves, it must be maintained. A paper on the types of software evolution and maintenance was published by Chapin et al. [17] describing just this. The authors propose that software evolution and maintenance can be defined in 12 types, split into 4 clusters: **business rules** (enhancive, corrective, reductive), **software properties** (adaptive, performance, preventative, groomative), **documentation** (updative, reformative), and **support interface** (evaluative, consultive, training).

For a detailed understanding of software evolution, the case study completed by Godfrey and Tu, regarding evolution on open source software [25], specifically the Linix kernel, is quite useful. The results of their findings show that the Linux kernel, over its first six years of existence, shows growth that was super-linear, which was a surprising result as many large systems tends to slow as size increases. This result is explained as an artifact of the open source development process, as much research in evolution prior to this study was conducted on single company traditional systems. The take away from this is that regardless of development style, software is evolving at an increasing rate.

The last piece of the puzzle in terms of software evolution, specifically in applications, is the ability to track evolution in a meaningful way. The simplest form of this would be differencing versions of software to determine how one differs from another.

However the results here may not be extremely important, and further investigation is required. Another approach is to look at the differences in a different way; Person et al. provide their approach to differential symbolic execution [52], which makes use of symbolic execution to determine symbolic meaning of a program, and then compare the two symbolic meanings to understand exactly how a system has changed.

Test Evolution

As presented in the section on Software Evolution, it is known that software is not a static object, and it will change and evolve over time; this is also true of software tests, out of necessity to ensure they are reflective of the new software functionality. Tests must change alongside the source code, to ensure that no new bugs have been introduced to previously tested code. Mary Jean Harrold presents work on testing evolving software [28], which fits this concept into the field of software development, and discusses areas of research in test evolution. Her work deals with the idea of selective retesting, which refers to determining which (if any) of the existing tests can (should) be reused for the next iteration of testing. Two other common areas of work in test evolution that are presented by Harrold are coverage identification (determining what type of coverage is suitable for successive evolutions of test suites) and test-suite minimization (determining the least amount of testing required to meet a certain criteria).

Tests for evolving software have become more commonly referred to as **regression testing**, however the concept and motivation remains the same. Insights into regression testing are presented by Leung and White [35]. The authors present the concept that regression testing can be split into two groups: **progressive regression testing**, and **corrective regression testing**. These groupings are based on whether or not the specification has changed or not - a change in specification would be progressive, where other changes are corrective. The authors also introduce the concept of a piece of software being **regression testable** - “a program is regression testable if most single statement modifications to the program entail rerunning a small proportion of test cases in the current test plan”.

Another term that applies to test evolution and maintenance of tests over time is **test case adaptation**. Mirzaaghaei et al. present their work on using test case adaptation to support test suite evolution [45]. Test case adaptation deals with automating the process of repairing and generating test cases during software evolution. Their approach uses heuristics to take data from existing test cases, and repair invalidated test cases, and generate new test cases as needed, such that the test suite is reflective of the new software. The first step in their process is one that has become a common step in test evolution, and this is calculating the difference between the two versions of the system. The second step of actually adapting the existing tests is by far the more difficult step, with a number of different approaches. The authors present five different algorithms used in their process: (i) repair signature changes, (ii) test class hierarchies, (iii) test interface implementations, (iv) test new overloaded methods, and (v) test new overridden methods. They performed quite successful experiments using the first two algorithms, and have plans to extend to the others in the future.

Another term that applies to this concept is **test co-evolution**, referring the fact that tests evolve alongside the software. Zaidman et al. [69] 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 demonstrated and validated using two open source cases (Checkstyle and ArgoUML) and one industrial case (a project by the Software Improvement Group (SIG)).

One interesting approach to the concept of test evolution, specifically test case repair, is presented by Daniel et al. [20], in which they 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. is aimed at understanding the myths and realities of test-suite evolution [53]. 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 of Mirzaaghaei et. al., in which they are able to automatically repair test cases for evolving method declarations [44].

It is important to note that many of the problems presented by Chapin et al. [17] regarding software evolution directly apply to test evolution as well.

Model Evolution

Model evolution is a process very similar to software or test evolution, but it is mainly centered on the model of the software system. In MDE, the model is the primary artifact of a system, and code can be generated from the model, so the evolution of the source code is no longer a primary concern. However, in a large number of cases, the model of the system is often an instance of a meta-model, and ensuring that models stay in sync with the meta-model they are based on is a key area of research in MDE. When dealing with concurrent versions of models and meta-models, a number of issues can occur; Cicchetti et al. present their proposed solution to this issue in their paper [18]. This process is often known as co-evolution of models, as opposed to simply focusing on one artifact and referring to it as model evolution; however the more general term can be applied. In their approach, Cicchetti et al. make use of model differencing, model merging, and model transformations to ensure that no inconsistencies arise between meta-model and model versions.

Earlier work by Cicchetti et al. presents a slightly different, but easier to understand methodology on automating co-evolution in MDE [19]. Their work deals with meta-model evolution and the co-evolution of conforming models; specifically looking at how meta-models may evolve over time. The authors present three categories of changes that may occur, and how they affect instances of the changes meta-model: **non-breaking changes** which have no effect on the conformance of the instance models, **breaking and resolvable changes** which have an effect on the model but are easily resolvable and can be automatically co-evolved, and finally **breaking and unresolvable changes** which break the conformance, and cannot be automatically

$$\begin{tikzcd}
MM_L \arrow[r, "\Delta"] & MM_{L'} \\
m \arrow[u, dashed] \arrow[r, "M"'] & m' \arrow[u, dashed]
\end{tikzcd}$$

Figure 2.1: Migration of model instances based on meta-model evolution [43]

repaired, requiring user intervention. They discuss how they produce a delta between the versions of the meta-models; changes are generalized into three groups: **additions**, **deletions**, and **changes**. These differences are then analyzed to ensure adaptability of instances (and in cases where this is not possible, adapted such that it is possible), in order to proceed. The differences are refined in such a way that a list of transformations is produced, which if applied to any model conforming to the original meta-model, will yield a model which conforms to the new meta-model.

Meyers et al. have their own take on the topic of co-evolution of models and meta-models [43]. Existing practices for updating instance models were seen to be time consuming and error prone, so a new approach is presented. Their approach is to make migration changes in a step-wise manner, ensuring that conformance is carried throughout the transformation. The goal is that after each change to a meta-model from MM_L to $MM_{L'}$, it is possible to automatically update all instance models m (which conform to MM_L) to instances models m' (which conform to $MM_{L'}$) by creating a single suitable migration M (the goal). This process is visualized in Figure 2.1. Their approach contains two steps at the highest level, the creation of the difference model, and the migration of instance models.

Yet another take on this is presented by Gray et al. [27], in which their main goal was to support model evolution in two categories of changes: changes that crosscut the model representations hierarchy, and scaling up parts of a model. The manual execution of these changes can not only reduce performance, but can also affect the correctness of the resulting representation. The solution proposed by the authors is C-SAW (Constraint-Specification Aspect Weaver), a generalized transformation engine for manipulating models. The authors claim that "the combination of model transformation and aspect weaving provides a powerful technology for rapidly transforming legacy systems from the high-level properties that the models describe."

An alternate approach to the model transformation method is presented by Mantz et al. [37] in which they propose the use of graph transformations to ensure accurate co-evolution.

One of the common themes that comes out of most (if not all) work on model evolution is that one of the most important steps in any process is determining how a model or meta-model has changed, and more often than not, this is done with some sort of model-comparison [63] tool. This is further evidence that while these research areas are presented as disjoint, they are certainly closely related. The other common theme that arises is that many of the issues that face software evolution presented by Chapin et al. [17] also apply to model evolution.

With the expansion of the software field to include modeling, the field of software evolution has expanded to include the concept of model evolution as well. Paige et. al. present a comprehensive survey of model evolution work over the years in their recent paper [49]. Particularly, the authors discuss relevant work in the topics of metamodel and model co-evolution, and model versioning - two concepts that closely relate to the work performed in this thesis.

Evolution of Model-Based Tests

The overlap of the three previous areas is the idea of evolving model-based tests. This section of work was the basis of my own MSc work, which was focused on the topic of evolution of model-based tests. Initial work focused on incremental testing of UML-RT models, using symbolic execution [58] while the final thesis focused more on the understanding of model evolution through incremental testing [56]. While both of these projects talk about the ideas of how models, and model-based tests evolve, the focus was never on the co-evolution of the tests, but more so on the models, and then incrementally generating new tests, as opposed to evolving the tests. The focus of the research was to achieve an improved understanding of the impact of typical model evolution steps on both the execution of the model and its test cases, and how this impact can be mitigated by reusing previously generated test cases. Existing techniques for symbolic execution and test case generation were used to perform an analysis on example models and determine how evolution affects model artifacts; these findings were then used to classify evolution steps based on their impact. From these classifications, it was possible to determine exactly how to perform updates to existing symbolic execution trees and test suites in order to obtain the resulting test suites using minimal computational resources whenever possible. The approach was implemented in a software plugin, IncreTesCaGen, that is capable of incrementally generating test cases for a subset of UML-RT models by leveraging the existing testing artifacts (symbolic execution trees and test suites), as well as presenting additional analysis results to the user. Finally, results were presented of an initial evaluation of the prototype tool, which provides insight into the tools performance, the effects of model evolution on execution and test case generation, as well as design tips to produce optimal models for evolution.

In work by Zech et al. [70], they present a platform for model-based regression testing, the product of their work is the MoVE (Model Versioning and Evolution) Framework, which is their generic platform to handle model-based regression testing. MoVE is a model repository that supports the versioning of models. The framework is able to work over a number of different model types, any arbitrary XMI based model format. This generality provides significant power to the framework. The process consists of three steps (delta calculation, delta expansion, and test set generation). All three of the steps make use of OCL queries to accomplish their goals. The delta calculation (or model differencing) is done by a modified version of EMF Compare to produce a delta model, which is used in the process, by further expanding this delta, to determine the exact changes, which are then used to generate the new test set. As a proof of concept of their work, a case study was conducted, comparing the generated regression tests with the tests produced through two existing model-based testing approaches: UTP (UML Testing Profile) and TTS (Telling TestStories).

Another approach to model-based regression testing for evolving software is presented by Farooq et al. [24], which focuses on selecting the appropriate tests to test software after system evolutions. Their work takes a state-based approach to regression testing, using the following tasks in order: change identification, change impact analysis, and regression test selection. These steps are combined into their tool START (STAte-based Regression Testing), which is an Eclipse-based plugin compliant with UML 2.1. The authors then present a case study using START on a Student Enrolment System, demonstrating its effectiveness in determining which tests from the original set are reusable (future use), re-testable (next round of testing), and obsolete (can be thrown away).

Pretschner et al. present their work on model-based testing in evolutionary software development [54], which presents executable graphical system models as both a representation of the system, as well as a method for model-based test sequence generation. As is the case with most of the work presented in this section, along with the proposed research, motivation is derived from the cyclical, incremental, and iterative development processes that have become more common in software development, and the resulting need for many sequential versions of tests, specifically in the MDE environment. Their approach uses the AutoFocus CASE tool as the basis for testing, which utilizes both propositional logic, and constraint logic programming (CLP) approaches to automated test case generation. The research focuses on the CLP approach, as it overcomes a number of limitations of the propositional logic approach (namely a state-space explosion problem); the AutoFocus model is transformed into Prolog rules and constraints, which when successively applied, symbolically execute the model, leading to one or more system run. While the approach does not directly deal with evolving model-based tests, it presents an alternative approach for model-based testing of evolving software.

Two papers by Briand et al. address the automation of regression test selection using UML designs: their original publication [15] and a follow up which expands upon the original work [13]. Much like the work presented by Fourneret, Briand et al. also aim to categorize tests, however only into three, somewhat similar, categories:

i. **reusable** - a test case that is still valid, but does not need to be rerun
ii. **retestable** - a test case that is still valid, but needs to be rerun in order to consider the regression safe
iii. **obsolete** - a test case that cannot be run on the new version (this may mean it requires modification, or complete removal)

In their extended report Briand et al. [13] apply their approach on three separate case studies: a IP Router system developed by a Telecom company, an ATM system, and a cruise control and monitoring system; the last two systems were developed by students which allowed the authors to define a variety of changes to make things more interesting. From these case studies, the authors discovered that the changes required between versions of tests can vary widely, and although their tool was able to reuse up to 100% of tests in some cases, the variability could become problematic. However they hypothesize that the approach would become more useful in larger systems due to its automation.

Another piece of relevant background is the specific application of this work, testing of Simulink Models. The recent work of Matinnejad et. al. is aimed at the automated testing of Simulink models [40], focusing on the creation of test suites for existing models. While this work deals with the creation of tests, it seemingly would require regeneration as models evolve, and doesn't explicitly provide a framework for updates to the models. Regeneration of tests can become expensive with many changes.

Based on the related work presented, it is evident there are a number of key ideas that connect research projects in model-based test evolution. Namely, the impact analysis of model version changes on tests is something that occurs in many cases, and there is often an effort to classify the existing test cases to determine the applicability/usability of these tests for the updated model/system versions. These concepts are useful throughout the thesis work.

2.1.3 Model Management

Model management refers to the practices associated with managing the complexities that come from software modeling. Since model-driven engineering (MDE) relies heavily on models as a primary artifact, and the fact that models bring with them a large number of nuances such as meta-models, model-based tests, model transformations, and multiple types of related models, it becomes a field of research to keep track of these artifacts, and ensure that they remain consistent over time.

Related to the fact that testing takes a large part of the cost and time in a software project, the same is true about continued maintenance over time. This is increasingly important when it comes to models, because there are more artifacts to deal with.

Work in model management takes place in numerous forms, and focuses on different combinations of model artifacts, such as managing models and meta-models, or models and their model-based tests.

Early exploration of the Software Model Management (SMM) problem was presented by Salay et. al[61]. The original work focuses on the management of large groups of models, and the requirement to merge these models. Particularly, the authors present their Eclipse-based tool framework for SMM.

The same research group has expanded their tool support for model management, citing the need to keep models up to date as requirements will change and the models need to respond to these changes. It is relatively simple to make a single change, but because there may be overlap in models, changes can have impacts in other related models, and this is something that is more difficult to ensure by hand - thus the need for tool support. After some work, MMINT was produced as their graphical tool support for interactive model management[62].

Beyond tool support, these champions of SMM have applied their techniques to several use cases, two of which are presented here. First they presented a position paper on model management for regulatory compliance[33]. This work looked the requirements and regulations set out by governing bodies, and how difficult it can be to conform to them, specifically with overlapping, and possibly even conflicting standards - they then present their solution by means of model management to ensure consistency between artifacts as changes occur. The second application for model management comes in the form of assurance case reuse[32]. Essentially the authors realize that evolving systems needn't require complete regeneration or recreation of assurance cases, and that the ability to reuse existing artifacts would be beneficial in the long run. This work is very similar to the work in test case impact analysis in this thesis, in that both aim to reuse existing model-based artifacts, and provide tool support to do so.

The term model management in the context presented here is confined to the research group presented for the most part, and the work presented in this thesis aims to expand its usage to include the test evolution and change impact analysis work it presents. Expansion of the type of management, as well as the type of models, both of which are done by the work in this thesis, help expand the research in this important area.

2.2 Related Work

With a solid background knowledge, it is now important to compare the work presented in this thesis to some of the more closely related work, and set this work apart from the rest by showing its uniqueness and importance.

The main contributions of this work fall into three categories, each of which are explored here in detail for similar work.

2.2.1 The Co-Evolution Relationship

A study similar to the project examining the co-evolution relationship was conducted by Zaidman et. al. [68] in which they examine the relationship between source code and tests, as opposed to models and tests. The authors explore research questions regarding the synchronicity of co-evolution, testing efforts surrounding major events, detection of test strategies from repository data, and the relationship between test writing and test coverage. Following this initial work, a follow-up study was conducted on the use of association rules to study the co-evolution relation between tests and source code [36]. Their work was then expanded to include, in addition to open source, industrial software, again using repository mining [69]. In that work they show that the observations of their earlier work on open source systems also hold in the industrial case study. Most recently they have applied their technique at a more fine-grained level [39].

Another similar study, conducted on 6 open source software systems, was conducted by Marinescu et. al. [38] in which they present Covrig, their framework for the analysis of code, test, and coverage evolution. In their framework they pose 10 research questions, several of which relate to the co-evolution relationship between source and tests. This study however finds that testing does not increase as often as source changes, and they conclude that testing is a phased activity (with the exception of one of the 6 systems, Git).

Ens et. al. present a different approach to the analysis of test and source co-evolution: they propose the use of visualization to understand the relationship, using their tool ChronoTwigger [23]. ChronoTwigger uses co-change as it's basis, examining the correlations between source and tests, to visualize their relationship, based on the premise that visualization helps with comprehension. From a small user study (3 users) the authors conclude that ChronoTwigger can lead to inferences about the relation between source and testing that are not easily made using existing techniques.

While these co-evolution studies are focused on source code and associated tests, the motivation is similar to ours, and the insights from their findings led us to want to attempt a similar study on models and tests.

In addition to looking at other studies regarding the co-evolution relationship, it is also important to look at work on the applications of co-evolution. One of the first applications of co-evolution as a tool for test updating was presented by Arcuri and Yao [4] in which they present their methodology for using co-evolution to automate bug fixing. The authors demonstrate their application by automatically fixing bugs found in an implementation of a sorting algorithm.

The work by Zech et al. [70] for the MoVE framework, while similar to the research conducted in this thesis, deals specifically with regression testing and the selection of the test set, however this thesis aims to focus on the co-evolution aspect, and how exactly tests change and evolve alongside the source models. Similarly the approach to model-based regression testing for evolving software presented by Farooq et al. [24] also focuses on the selection of regression tests for future testing; the evolution study in this thesis incorporates this step but also expand to include the adaptation of existing tests.

2.2.2 Test Harness Generation

One of the contributions of this thesis is a reliable test harness generator for Simulink models in the automotive domain, as such it is important to situate this work in the state of the art and differentiate from others.

Rocha and Martins introduce a similarly model-based method for test harness generation for component testing [59], however this approach generates a test harness which executes on code, rather than a source model. The concept is the harness is model-based, using UML activity diagrams as the source, and it converts the activity diagrams (through a number of intermediate representations) to a programming language test harness. SimTH differs from this in that it produces another model in the native modeling language which is responsible for testing the source model.

Okika et. al. present work in the creation of a test harness for legacy software, specifically using the example of and embedded system[48], 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, which differs from SimTH, which is able to generate a unique test harness for each Simulink model, which is created automatically given the model as input. The work of Okika et. at. provides a more general solution, however it is not applicable to the type of testing required by our industry partners.

When it comes to model-based test generation, the related work expands greatly. Outside of test harness generation, there are numerous researchers working on model-based test generation, which generally refers to the test data values. A 2002 survey by Hartman introduces a variety of tools designed to generate test cases based on models [29], and a more recent (2012) taxonomy of model-based testing approaches is provided by Utting et. al.[66]. To discuss the tools provided in these surveys in detail would be outside of the scope of this paper, but it should be abundantly clear that work on model-based test generation, specifically for test data, is plentiful.

One specific subsection of test case generation work that is worth further 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 their approach for an integrated test generation tool for enhanced coverage of Simulink and Stateflow models[50]. Their work, while specific to the application domain also only deals with test case generation as opposed to the generation of harnesses to facilitate testing; additionally the focus of the generated tests is on coverage, while the tests used in the co-evolution experiments are functional tests based on requirements. Mohalik et. al. present similar work, using model checking to automatically generate test cases for Simunlink/Stateflow[46]. Although this work is closely related in aims to SimTH, it is another approach for test case generation rather than automatic generation of test harnesses.

As noted, SimTH falls in the intersection of these two areas of related work, which presents a major gap, coming specifically from the absence of tools designed for model-based test harness generation. While these examples of work involve model-based test case generation and test harness generation for source code, even with the specific application of Simulink models, there is still a gap that can be filled by SimTH.

2.2.3 Impact Analysis

Impact analysis refers to the science and art of determining what software parts affect each other[5]. Essentially, the goal is to determine what the impact of a particular change is on the artifact and related artifacts.

Similar work in impact analysis of source code was presented by Rungta et. al.[60], in which they present an analysis of impact to show evolving program behaviours. This work differs from the work in this thesis as it focuses on source code change impact instead of at the model level, however the concepts are similar and the desire to understand the behaviour of evolving systems provides a similar motivation to the co-evolution study conducted in this thesis.

In terms of work in the modeling domain, Briand et. al. present work in impact analysis and change management of UML models[14]. Additionally, they present an automation of this impact analysis based on UML designs[15]. These are very closely related, however the implementations differ greatly, as do the applications. The work in this thesis applies to Matlab Simulink models, for which these types of impact analysis techniques have not previously been effectively applied.

The closest related work in terms of impact analysis of Simulink models is a tool called DiffPlug[1]. DiffPlug is capable of performing visual differencing of Simulink models, as well as model merging and an implementation of impact analysis, which they call signal tracing. However, this tool falls short in a number of areas, where the implementation produced for this thesis does not. First, DiffPlug is not implemented directly in Simulink, but rather a stand alone program with the ability to view Simulink models; the developers cite this as an advantage, however this means that those using Simulink in industry would require two tools (Simulink and DiffPlug) at the same time to view the results and interact with the models. Since the SimPact plugin is integrated in Simulink, users do not need to switch between tools, which is a huge benefit to industrial users. The second area where DiffPlug falls short is in the implementation of its signal tracing. It provides the ability to select a block and trace its signals, which can be used in the same use case as SimPact (what-if scenarios of change), but it only does this in the current model level and does not trace through the whole model hierarchy, which SimPact does. Essentially, DiffPlug requires users to take multiple hops to get to the target inputs and/or outputs whereas SimPact is capable of traversing the model hierarchy and showing all impact from a given block. These two major deficits of DiffPlug are substantial enough to differentiate SimPact, and also provide further validation of the effectiveness of SimPact for the desired applications.

2.3 Summary

This chapter served to familiarize the reader with important background knowledge as well as related work in the applications of the thesis.

The background information fell into three categories (model-based testing, model-based test evolution, and model management), each having a number of important sub-topics, and presented some of the key discussions for the respective fields.

In terms of the related work, the research and technical contributions of this thesis fall into three areas (co-evolution, test harness generation, and impact analysis), each of which contain some closely related work. In this chapter the similarities of, and more importantly the differences between, the state of the art and the work in this thesis were presented. From these comparisons, it can be concluded that the work presented in this thesis is sufficiently unique from existing work while also being directly impactful in the application domain.

Chapter 3

Overview

This chapter introduces the contributions of this thesis and their relation to one another. Following chapters detail each of the individual steps.

The overarching theme of this work is an exploration of the evolution of Simulink models in an industrial context. The exploration proceeds in five steps: an examination of Simulink models and tests, a study of the co-evolution relationship for these models and tests, the development of a method for impact analysis for evolving Simulink models, the creation of a tool suite to support Simulink model management, and validation on real-world models.

3.1 Simulink Models and Tests

The first step in the research was understanding the problem, and to do this an understanding was required of the artifacts that are part of the case study. This section discusses the Simulink Models and Tests received from our industry partner.

To begin, Simulink[2] is a graphical modeling technology used to create models, which are collections of blocks, subsystems, lines and embedded code, that form block diagrams, used to simulate real world systems. Simulink is used to model systems for early testing and analysis, and ultimately is capable of generating the actual source code that will be run on these embedded control unit (ECUs). Simulink is commonly used in the automotive and aerospace industries as it provides increased confidence in the final developed products.

The models provided to us are Matlab Simulink Models from the automotive domain. The models are structured into nine main components (referred to as *rings*), each of which is made up of a number of sub-components. For example, ring 1 is made up of 4 sub-components, so there is a model for each of the 4 sub-components, and one integration model for the ring, which links the sub-components together. In total there are 55 sub-component models, and 9 ring integration models, for a total of 64 different models in the system (not all of them exist in all versions).

In total there are 15 releases of the entire system from a version control system (VCS). Table 3.1 shows the breakdown of models per release per ring; each count includes the sub-component models and the ring integration model. There are actually several releases of the system during which there were no changes and no versions added to the VCS (releases 4,5,9,12,14). Since those involved no evolution, for this research they were removed entirely, leaving 10 releases to examine. In some of these remaining releases, there were no updates to some of the models, meaning there are not 10 versions of each model. For the purpose of this research, duplicates were created of the previous versions where a version does not exist for a release, for the purpose of comparing versions.

The term "release" in the context of this work does not actually refer to a release version of the system, as all of the versions examined are pre-release. A release in this context refers to a milestone in development; every incremental time block, the

Table 3.1: Number of models provided for each ring at each release

	Rings								
Rel.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	4	4	8	11	4	4	10	7	8
2				11			10		
3	4	4	8	10	4	4	10	7	8
4									
5									
6	5	5	8	10			10	7	8
7	5	4		10				7	
8	6	4	8	10	3	3	10	7	8
9									
10		4	4				10	7	8
11	6	4	8	10	3	3	10	7	8
12									
13	6				3	3	10		
14									
15	6	4	8	10	3	3	10	7	8

release number is increased. The releases where all models were originally provided (as seen in Table 3.1) can be considered major releases in the life-cycle of the system; thus releases 1, 3, 8, 11, and 15 are the major releases for this software.

For each of the models, there is an associated test suite which was provided along with the models. For the purposes of this application, a test suite is contained in an Excel workbook comprised of a number of Excel spreadsheets (using the tabs in Excel), each containing test inputs and expected outputs for the given model, over a number of time steps. Each row represents a time step, and each column is an input value or an expected output value. The tests are run by simulation in Simulink, and the results compared with the expected outputs; this process is described further in Section 6.2.1.

3.2 Co-Evolution of Simulink Models

With an understanding of the models and tests, the next step was to understand how they evolve during a typical software life cycle. Given the availability of models and tests over versions, it was determined that by examining the differences between models and tests over versions it could be possible to uncover details about the relationship of co-evolution.

To this end, experiments were conducted to highlight when co-evolution occurred, as well as times when it did not (revealing types of changes where tests were not impacted by change). This work is described further in Chapter 4.

3.3 Impact Analysis for Evolving Models

From the results of the co-evolution study, along with discussions with our industry partner, areas where tool development could greatly improve the support for Simulink model evolution were discovered; the first of which was the fact that determining how a change impacts the rest of the model and the test cases was desirable. Other areas included automation of developer heavy processes, but these are discussed in a later section.

The first issue relating to impact analysis led us to develop a set of algorithms to eventually be included in a support tool.

As part of impact analysis, there were two major tasks undertaken, each of which resulted in an algorithm developed specifically for this application, but also capable of generalizing to a other uses. The two components are:

- determining relevant changes between model versions

- determining the impact of any given changed block on the surrounding model blocks and test values

Each of these algorithms was later incorporated into a tool or piece of a tool, as described in the next section. This work of developing these these algorithms is described in detail in Chapter 5.

3.4 Simulink Evolution Tool Support

With a number of analysis techniques developed, it was important to bring these to our industry partner in a tool they would be comfortable using, and something that fit into their already extensive workflow. The idea was to integrate with existing tools, and design interfaces that are visually similar to their other custom tools. This took shape in one industrial tool, consisting of two major functionalities and the ability to interact with existing tools.

SimEvo is the Simulink Evolution Support Tool designed to take as input the previous version of a model library and its test suite, along with an updated model library, and interact with them in a number of ways to produce information or testing artifacts useful for model developers and testers. The two main functions which were developed as part of this thesis are the ability to determine which test vectors have potentially been impacted by the change (thus identifying candidates for change), and the capability of generating a new test harness model for the updated library; both of these are described below. The secondary functionality of SimEvo is its ability to interact with existing tools used by our industry partner. Given a set of candidate values requiring updates, SimEvo can use the simulation power of our industry partner's testing tool to produce suggested outputs for the developer to use as possible output values, following a manual inspection. The ability to interact with existing tools, along with two unique contributions make SimEvo a valuable tool for our industry partner.

The development of SimEvo, along with the two subtools, SimPact and SimTH, is described in detail in Chapter 6.

3.4.1 SimPact

SimPact is the subtool used to determine the impact of changes to models over versions to the test values, both inputs and outputs. It works by determining the differences (which have an impact on tests) between model versions and then tracing through the model to the inputs and outputs impacted by the change, producing lists of test input/output vectors as candidates for change. These results can then be used for manual inspection, or by suggesting outputs using existing tools as described above.

In addition to the static analysis on a new model version, SimPact was implemented as a Simulink plugin to do dynamic analysis of change impact to models. An addition to the Simulink block context menu allows developers to perform a what-if analysis on a model block (including inputs and outputs) to determine potential impacts of changing that block. The impacted paths are highlighted directly in the model and transcend the model hierarchy to all levels.

The combination of static and dynamic analysis of change impact on test values provided by SimPact is an important aspect of the SimEvo tool.

3.4.2 SimTH

SimTH is the subtool used to automatically generate a new test harness model for the updated library model. Most often, changes to a model will not require a new test harness to be generated, but in the cases where the changes are significant enough (changes to the number of inputs or outputs), a new harness is required. SimTH is able to augment the library block (representing the system under test) with the required blocks and signal generators to simulate the test environment, and is a fully automated test harness generation tool.

SimTH also works for new models where a test harness has never existed, thus saving a lot of development time upfront, in addition to simply supporting the evolution of test harnesses.

3.5 Validation

With a number of methods introduced and new tools implementing them, validation of the applications was required prior to introducing them into an existing toolchain. Using the model sets described above, the effectiveness of both SimPact and SimTH was demonstrated using validation experiments.

Full descriptions of the experiments and the results and findings are presented in Chapter 7.

3.6 Summary

This chapter introduced the five steps of the research in Simulink model evolution. In the following chapters, the details of each step are outlined in detail.

Chapter 4

Co-Evolution of Simulink Models and Tests

Note: The work in this chapter has been previously published in a similar form[57], and is included here as a part of the larger thesis.

The first step in this project to support the management of Simulink models, was to examine the model set provided and determine how it, along with the associated tests, evolve over the course of the time leading up to release. In order to understand what types of issues can arise, and what types of changes occur frequently versus less often, it became an important first step to study the co-evolution relationship for the model set.

The link between development and testing is a strong one, and one that strengthens even more over the life-cycle of a software system. When this relationship is coupled with the fact that software (be it source code, models, etc.) is not a static object, and will change over time [34], it becomes increasingly important to ensure the corresponding tests remain up to date.

A change in a software system is very likely to require a corresponding change in its tests. This overall relationship is known as test co-evolution, and when applied to model-based tests is it is known as model-based test co-evolution. This work aims to improve the ability of our industrial partner to easily and effectively maintain the model-based tests that correspond to source models as they evolve over time, and to infer associated practices to aid in this process. Our industrial partners currently maintain this relationship manually, which can be unnecessarily time consuming. The long-term goal is to provide semi-automated guidance on the required updates to test cases, and ultimately to automate much of the test co-evolution process, in order to ensure the continued production of high quality software with reduced effort.

By first completing the case study presented in this chapter, it is possible to identify the frequency of changes in models that require updates in tests, and other interesting basic relationships. Based on this, an analysis was performed of a system of 64 production Matlab Simulink Models and their corresponding test cases, provided by an industrial partner, to answer the following research questions:

RQ1: Does test co-evolution happen synchronously or is there a delay?

RQ2: Is there a noticeable increase in development and test activity surrounding major releases or significant events? Is there a noticeable stabilization nearing the final release?

RQ3: Are there instances of changes to models that do not necessitate changes in the tests? If so, how common are they?

RQ1 and **RQ2** are very similar to the first two research questions investigated in the work of Zaidman et. al. [68] on the co-evolution of production and test code. This study aims at answering these questions in relation to models, rather than source code.

These questions are being asked in order to better understand the relationship between Matlab Simulink Models and the tests for these models. Obtaining an understanding of the relationship between them helps determine how a change at the model level can propagate through to the tests. In addition, an understanding of the relationship between modeling and testing is of interest to our industrial partner; being able to determine where greater testing effort is required can help to increase software quality.

4.1 Experiment

4.1.1 Preprocessing

The process began by removing the ring integration models from the analysis, due to the fact that their contents are simply the merge of the sub-components along with connecting lines between them. Additionally, the test cases for the ring integration models were not created until the final release (work completed during my internship), thus there is no valuable information provided by including them in the study. Furthermore, there are two models which simply produce output, and do not reply on input. These were excluded from the analysis since there are no associated tests for these models, and thus no comparison results to be obtained. Release 2 was removed from the analysis due to some anomalies in one of the model files in that version. This problem was dealt with by performing the comparison between release 1 and release 3. There is no concern about the validity of removing this release since there were only two systems with changes, and only one model in each was changed. These changes are adequately captured in the comparison between releases 1 and 3.

After all of the pruning of extraneous models, the set contains 53 component models over 9 releases, for a total of 477 models and their associated tests, to perform this experiment on.

4.1.2 Process

The experiment was conducted by performing a pairwise comparisons of two consecutive versions of the same model, and recording the results. This was repeated for each model, and then for every pair of releases, resulting in 8 release comparisons for the 53 models.

The result of a model comparison is one of the following:

1. No Change
2. Model Does Not Exist Anymore (but existed previously)
3. Model Does Not Exist Yet (but will exist in a later release)
4. Model Newly Created This Release
5. Model Deleted This Release
6. Model Was Modified (listed as # of additions, modifications, and deletions)

The same process was repeated for the test cases, again storing the results for analysis. The result of a test case comparison is one of the following:

1. No Change
2. Test Does Not Exist Anymore (but existed previously)
3. Test Does Not Exist Yet (but will exist in a later release)
4. Test Newly Created This Release
5. Test Deleted This Release
6. Test Was Modified (listed as # of test cases added or removed OR # of test which are the same and # of tests that have changed)

With results for both models and tests, matching types of results, a comparison between each result for a given model and test evolution was performed. This is to say, for sub-component 1, between release 1 and release 3, a result is obtained based on how the model changed (if at all) and how the test changed (if at all) and compare the two looking for a relationship. The result of this comparison is one of the following:

1. No Change In Both
2. Model and Test Do Not Exist in Both Releases
3. Model and Test Newly Added This Release
4. Model and Test Deleted This Release
5. Change in Model But Not In Test
6. No Change in Model But Change In Test
7. Change in Model and Test.

The above outcomes can be summarized as no change (1,2), matching changes (3, 4, 7) or not matching (5, 6). These results form the basis of the analysis.

4.1.3 Implementation

This section presents the implementation of the experiment. As the focus of the analysis was on Matlab Simulink Models, the comparisons were performed directly in the Matlab environment. Matlab scripts were written to collect all of the appropriate model and test files, and to perform the pairwise comparisons.

Model Comparison

The Matlab script took as input the paths to two versions of the same Matlab Simulink model, and performed a comparison between the two versions. To do this, the built-in

Simulink differencing tool was leveraged, which is an XML comparison method that returns a single object, containing all of the differences between any two Simulink models (or two versions in this case).

Given this result, the script then removes any non-semantic differences such as placement, colours, version number labels, etc. and then iterates over each difference one at a time. The script then determines whether the difference is the addition of a new element to the model, a modification to an existing element, or the deletion of an element from the previous version. The result is reported as a string of the form:

A: # M: # D: #

where the numbers of additions, modifications, and deletions are reported respectively.

This process is repeated for all of the models, performing a total of 424 pairwise comparisons of the models, and storing the results in a Matlab cell-array. Upon completion the results are written to an Excel spreadsheet for further investigation.

Test Suite Comparison

Similarly to the model comparison, a script was created which took as parameters the paths to the two Excel workbooks containing the test suites for two versions of the same test suite.

The first step in comparison was to examine the number of test cases in the test suite, which is done by counting the spreadsheets in the workbook, and determining if there are the same number of test cases. If there are an equal number of test cases investigation continues, otherwise the result is reported as the addition or removal of some number of test cases.

If there are the same number of test cases, a comparison of each test case from each test suite to the corresponding test case in the other test suite is conducted.

This is done by performing an element by element comparison of the spreadsheet; if a single cell is found to be different, that individual test case is marked as changed, and exploration continues with the remaining test cases. Upon completion of the comparison, the result is reported as the number of test cases that have changed and the number that remained the same. If all of the tests change, or none of them change, this is the result reported, otherwise, the resulted is reported as a string of the form:

Same: # Diff: #

where the number of tests that are the same and the number that have changed are reported respectively.

This process is repeated for all of the test suites, performing a total of 424 pairwise comparisons of the test suites, and storing the results in a Matlab cell-array. Upon completion the results are written to an Excel spreadsheet for further investigation.

Co-Evolution Relationship

With the results of both comparisons available, a final Matlab script was created to perform a comparison between the model results and the test results, in order to come up with the relationship between the two comparisons.

This was done by simply iterating over the 424 cells in each table and comparing them to come up with a result; the result being a number from 1 to 7, corresponding to the seven outcomes presented in Section 4.1.2. The results were written to an Excel spreadsheet for further investigation and final analysis.

Table 4.1: Co-Evolution Relationships Between Models and Tests

	Model Release Pairs							
mdl	**1—3**	**3—6**	**6—7**	**7—8**	**8—10**	**10—11**	**11—13**	**13—15**
1	**1**	**7**	**5**	**5**	**1**	**5**	**1**	**1**
2	**7**	**7**	**1**	**5**	**1**	**6**	**1**	**1**
3	**2**	**2**	**2**	**3**	**1**	**7**	**1**	**1**
4	**2**	**3**	**7**	**5**	**1**	**7**	**7**	**1**
5	**7**	**1**	**4**	**2**	**2**	**2**	**2**	**2**
6	**2**	**2**	**3**	**7**	**1**	**5**	**1**	**1**
7	**1**	**7**	**1**	**5**	**1**	**5**	**1**	**5**
8	**7**	**7**	**1**	**1**	**1**	**1**	**1**	**1**
9	**7**	**1**	**4**	**2**	**2**	**2**	**2**	**2**
10	**2**	**3**	**1**	**7**	**5**	**7**	**1**	**1**
11	**1**	**7**	**1**	**1**	**1**	**1**	**1**	**1**
12	**1**	**5**	**1**	**1**	**6**	**5**	**1**	**1**
13	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
14	**1**	**1**	**1**	**5**	**1**	**1**	**1**	**1**
15	**5**	**1**	**1**	**1**	**5**	**1**	**1**	**1**
16	**1**	**7**	**1**	**1**	**1**	**5**	**1**	**1**
17	**7**	**1**	**1**	**5**	**1**	**1**	**1**	**1**
18	**5**	**1**	**1**	**1**	**1**	**1**	**1**	**7**
19	**7**	**7**	**1**	**1**	**1**	**5**	**1**	**1**
20	**7**	**1**	**1**	**5**	**1**	**1**	**1**	**1**
21	**7**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
22	**7**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
23	**7**	**5**	**7**	**7**	**1**	**1**	**1**	**7**
24	**4**	**2**	**2**	**2**	**2**	**2**	**2**	**2**
25	**7**	**7**	**1**	**1**	**1**	**5**	**1**	**1**
26	**1**	**1**	**1**	**7**	**1**	**1**	**1**	**1**
27	**1**	**1**	**1**	**7**	**1**	**1**	**7**	**1**
28	**7**	**1**	**1**	**4**	**2**	**2**	**2**	**2**
29	**1**	**1**	**1**	**7**	**1**	**1**	**6**	**1**
30	**1**	**1**	**1**	**7**	**1**	**1**	**5**	**1**
31	**7**	**1**	**1**	**4**	**2**	**2**	**2**	**2**
32	**5**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
33	**7**	**1**	**1**	**7**	**1**	**1**	**1**	**1**
34	**5**	**5**	**1**	**5**	**1**	**1**	**1**	**1**
35	**5**	**7**	**1**	**1**	**1**	**1**	**1**	**1**
36	**7**	**1**	**1**	**7**	**1**	**1**	**7**	**1**
37	**5**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
38	**5**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
39	**5**	**1**	**1**	**5**	**1**	**1**	**1**	**1**
40	**5**	**1**	**1**	**5**	**7**	**1**	**1**	**1**
41	**1**	**7**	**1**	**1**	**1**	**1**	**1**	**1**
42	**5**	**5**	**1**	**5**	**6**	**5**	**1**	**1**
43	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
44	**1**	**1**	**1**	**5**	**1**	**1**	**1**	**1**
45	**1**	**1**	**7**	**1**	**1**	**1**	**1**	**1**
46	**5**	**1**	**1**	**1**	**5**	**1**	**1**	**1**
47	**7**	**1**	**1**	**5**	**1**	**1**	**1**	**1**
48	**7**	**5**	**1**	**5**	**1**	**1**	**1**	**1**
49	**6**	**1**	**1**	**5**	**1**	**1**	**1**	**1**
50	**6**	**1**	**1**	**5**	**1**	**1**	**1**	**1**
51	**7**	**1**	**1**	**5**	**5**	**1**	**1**	**1**
52	**6**	**7**	**1**	**5**	**5**	**1**	**1**	**1**
53	**6**	**1**	**1**	**7**	**1**	**1**	**1**	**1**

Numbers refer to definitions in Section 4.1.2

White – No Change in Both (1,2)
Light Gray – Co-Evolution Occurrence (3,4,7)
Medium Gray – Change in Test, None in Model (6)
Dark Gray – Change in Model, None in Test (5)

4.2 Analysis

4.2.1 Results

The results of the co-evolution relationship analysis between the models and tests are shown in Table 4.1. Each column represents the evolution from one release to the next, while each row is a specific model. The values in the cells correspond to the comparison results described in Section 4.1.2. However, since several of the outcomes can be grouped together, the cells have been shaded to show the groupings of similar outcomes. The most common outcome (72.17%), was when there was no change in the model and no change in the test or when neither artifact existed (results 1 or 2), which is shown by all of the white cells. The next most common outcome (14.15%) was the fact that a co-evolution relationship existed between the model and the test suite; this can mean that either both artifacts were added (result 3), both artifacts were deleted (result 4), or both were changed (result 7); all of these are shown by the cells shaded light gray. The final two results, which occurred the least frequently (13.68%) were the results where there was a change in one artifact but not the other: a change in the test but not in the model (result 6) is shown in medium gray, and a change in the model but not the test (result 5) is shown in dark gray. It is these two categories that provide relationships outside of the expected, and thus the ones that require further investigation; this analysis can be found in the answers to **RQ1** and **RQ2**.

From this data, it was desirable to determine how often throughout a particular model's life-cycle did it occur that there was a change in one artifact but not the other. This data is shown in Figure 4.1. What can be seen here is that 38 of the 53 models (71.1%) only have this occur 0 or 1 times throughout their life cycle, with a small portion of the models displaying this property more frequently. Of note, model 42 demonstrated the property of having a change in one artifact and not the other in 5 of the 8 evolutions. Upon further investigation, it was found that model 42 was a model that changed frequently, but the changes did not necessitate changes to the test cases as they were internal changes (more on this type of change is discussed in **RQ3**).

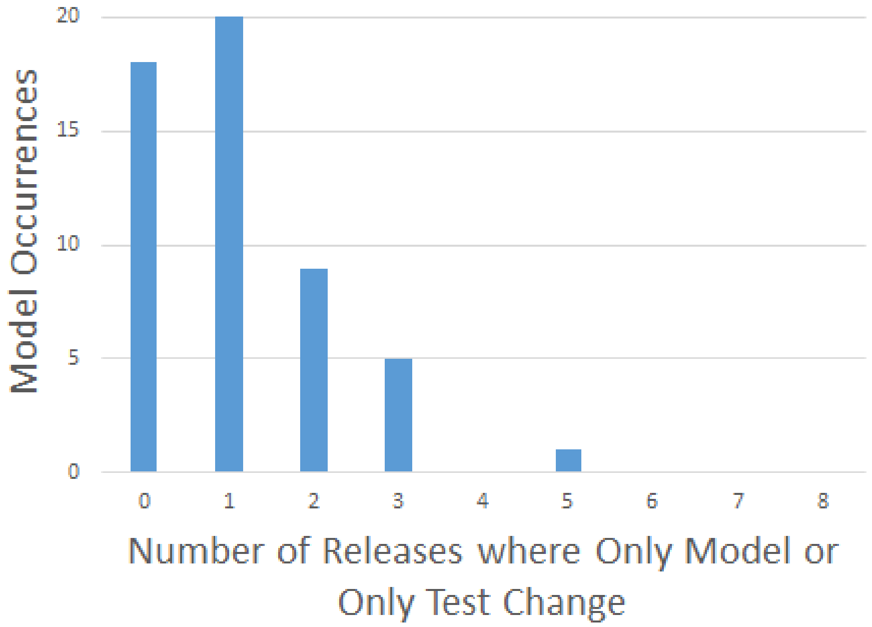


Figure 4.1: Frequency Graph of Evolution Step Disparity

While there was an initial belief there should be a relationship between changes in the models and changes in the tests, it was important to examine this relationship in detail. Figure 4.2 shows us the relationship between the percentage of models that change between each release and the percentage of tests that change in each release. From the data it is possible to conclude there is a strong positive correlation between models changed and tests changed, $r(6) = 0.9, p < 0.01$.

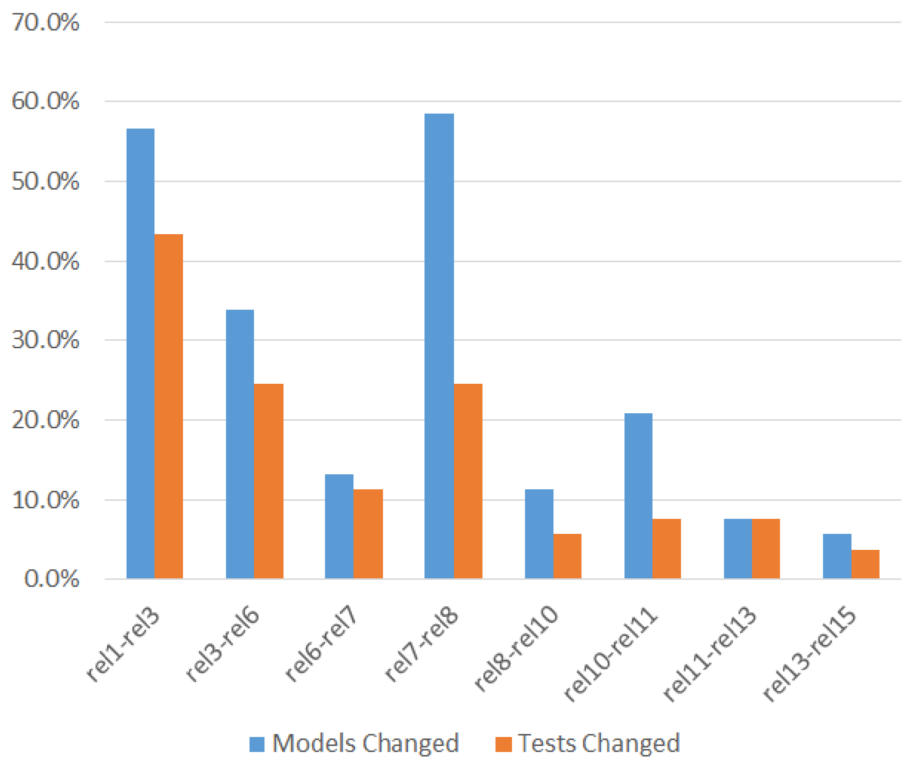


Figure 4.2: Percentage of Models & Tests Changed at Each Release

The next area of analysis was an examination of the type of change (or lack thereof) over time. For this it is necessary to return to the groupings from Table 4.1, but group the two types where changes occur in only one artifact into one group. This left us with the fact that in a given evolution, models and tests could either both not change, both change, or only one change. The results of this can be found in Figure 4.3.

4.2.2 Discussion

RQ1: *Does co-evolution happen synchronously or is there a delay?*

Since it is the case that at each release the tests are run on the models prior to

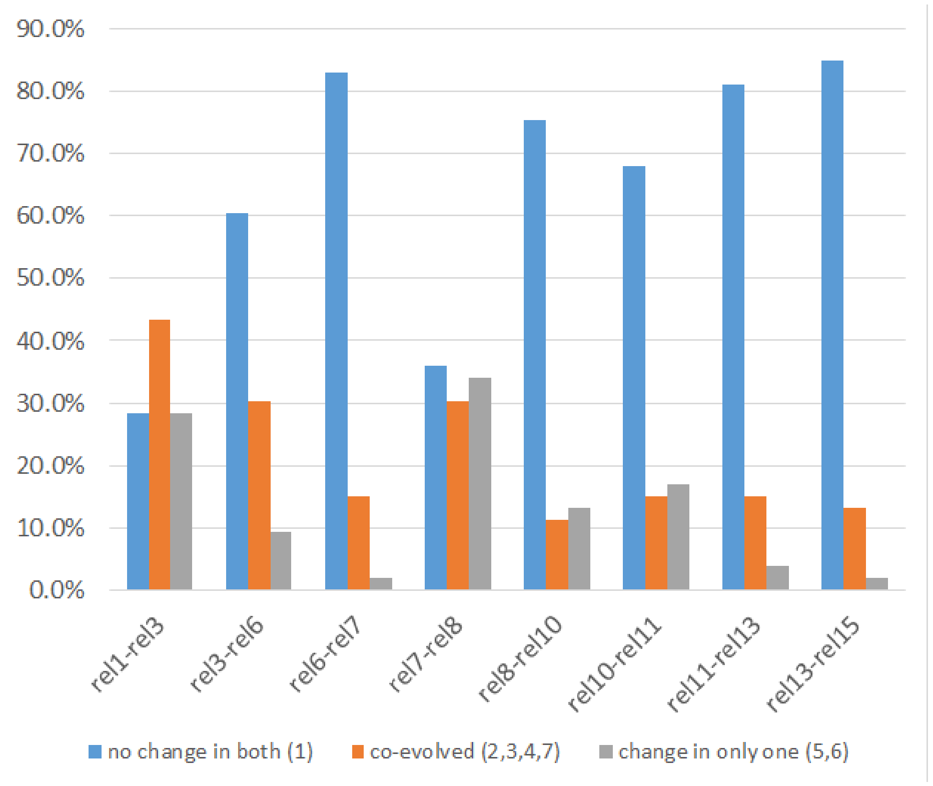


Figure 4.3: Types of Change over Time

checking into the VCS, and the only model information is the version check-ins for the releases, it is not possible to provide any further granularity beyond releases to know exactly when changes in tests occur in relation to changes in the models. Given the information available, it can be concluded that it does happen consistently when required, with only a few minor exceptions.

The exceptions are those instances where there is a change in one artifact and not the other: a change in model and not test (result 5), and a change in test but not model (result 6). **RQ3** addresses those instances of result 5, however when a change is made to a test but not the model, this can be seen as an instance of a delayed update to the test, which warrants further investigation.

There are 8 instances where the tests changed and not the model (result 6), and these occurred only once in the life time of models 2, 12, 29, 42, 49, 50, 52, and 53. Each of these are discussed here in order to understand what the lone occurrence of test changes means for the system.

To begin, a discussion Models 49-53 is helpful; the changes made to these tests was that they were newly added tests. These 5 models are all sub-components of the same ring, and the absence entirely of tests in the first release of the system means that there was in fact, not a delay of changes to the models, they were just not included with the check-in of this version for some reason. Thus, this negates the concept of a delay in test updates to these tests.

The occurrences of changes in tests but not models to models 2, 29, and 42 all share similar circumstances; there were no changes made to the existing tests, and additional tests were created. This again negates the concept of a delay in testing, and simply represents an increased effort in testing when changes were not required to the models during this release period.

The final occurrence of changes to a test suite but not the source model is in model 12, where all of the test cases were modified. This marks the first result requiring investigation into the actual test cases to determine the changes; however the findings were that the changes to each test case were changes to the frequency of changes in values (the time step of the values), and not the inputs or expected outputs themselves.

From this analysis, it can be concluded that in this model set, there is no concept of delayed updates to test suites after model updates, and that when a change to test suites is required, it is completed prior to the end of the current release cycle.

RQ2: *Is there a noticeable increase in development and test activity surrounding major releases or significant events? Is there a noticeable stabilization nearing the final release?*

The answer to this question can be found by examining the percentage of models and/or test that change surrounding these events, as compared to the releases immediately prior. A summary of this information can be seen in Figure 4.2.

Recall that releases 1, 3, 8, 11, and 15 are the major releases for this software. By eliminating the first release, since there is no prior information, which can also be said for the change from release 1 to release 3, it is possible to highlight the changes made between releases 7 and 8, 10 and 11, and 13 and 15 as the points of interest for this research question, as they are the updates made prior to the major events in the product life cycle.

Upon a visual inspection of Figure 4.2, it becomes clear that there is indeed an increase in development and testing prior to release 8 and 11, but not release 15.

Between releases 6 and 7 13.2% of the models changed, and between releases 7 and 8 58.5% of the models changed, an increase of 343.2%; similarly changes in tests at these two evolution steps increased from 11.3% to 24.5%, an increase of 116.8%. Examining these same differences for the major event surrounding release 11, it is important to note that changes in models increased from 11.3% to 20.8%, an increase of 84.1%, and changes in tests increased, only slightly, from 5.7% to 7.5%, an increase of 31.6%.

While it is shown that changes in models and changes in tests actually decrease prior to the final release (decreasing 24% and 49.3% respectively), it is still reasonable to conclude that there are substantial increases in development and testing efforts surrounding major events of a software system prior to release, with the exclusion of the final release. An understanding of this exception is that the final release changes would only be small minor 'polishing' changes.

Regarding the question of stabilization, referring to the reduction of changes made at each evolution step, as the system approaches the final release, examination of Figure 4.3, showing the amount of change over time, does indeed indicate a stabilization. The obvious conclusion here is that there exists a noticeable point of stability as the system approaches the final release, with the peak of no change being the final evolution step, at 94.3% of the models and tests not changing at all. There are however 3.8% of the models and tests that have a consistent co-evolution at this co-evolution step; this can again attributed to the same types of 'polishing' changes discussed above.

RQ3: *Are there instances of changes to models that do not necessitate changes in the tests? If so, how common are they?*

The answer to this question is of importance for the following chapters, in that identifying the types and frequency of changes in models that do not have a direct impact on the test suites can help reduce the testing efforts by identifying when a change needs (or need not) to be made to the test suite.

Since the test suite for each model is run and must pass at every release prior to being checked into the VCS, any time where there is a change in the model but not the test (a result of 5 in Table 4.1) provides us with an instance of changes that do not necessitate changes in the test suites. Based on this, it seems possible that there are indeed changes that do not require direct updates to the test suite.

To the question of frequency, there are 50 occurrences of this over the life of the system, whereas there are 110 instances of model changes occurring over the life of the system; this equates to 45.5% of the model changes not requiring a change in the associated test suite.

From this it stands to reason that there are a substantial number of changes that occur to source models that do not impact the overall behaviour of the models, nor the results of running the existing functional tests.

4.3 Findings

This experiment provided evidence of co-evolution between models and tests, as well as a strong indication that there exist a substantial amount of model changes that do not require changes in test cases. It is these findings that demonstrate the need for an accurate impact analysis tool, that can determine precisely the impact of changes in a model on the related tests.

For a model developer, if they are able to determine, in real time as they work on a model, what the impact of a particular change may be, they are able to make informed decisions about changes, and also to ensure that the related artifacts (test models, test cases, connected models, etc.) are updated promptly to reduce the potential for errors.

This impact analysis is the focus of the next chapter, which focuses on the identification and reporting of potential impact of Simulink model changes.

4.4 Summary

This chapter presents a case study of the co-evolution relationship between Simulink models and test cases provided from the automotive domain; there were, after pre-processing, 53 models over 9 releases, and the associated test cases for each model, used for this experiment.

Three research questions were asked and answered, with the following being the general conclusions:

- With minimal exceptions, co-evolution happens synchronously, meaning that the tests and models are evolving consistently, in this given model set.
- Including the instances where both the model and test set do not change, there is a strong positive correlation between models and tests as they evolve in this model set: $r(6) = 0.9, p < 0.01$.
- It is reasonable to conclude that this model set adheres to existing notions of how traditional software systems evolve, in that development increases prior to major releases, and stabilizes approaching the final delivery.
- There are a significant number of changes that can be made to a model that do not have an impact on the test cases - these are the purpose for further investigation.

Based on these findings, it becomes important to move into specific impact analysis for model changes, which is the focus of the next chapter of this thesis.

Chapter 5

Simulink Impact Analysis for Evolving Models

Given the findings of the co-evolution study it is evident that the ability to determine, on a change by change basis, the impact of a given model change on the test cases is desirable as a means of identifying changes of both high and low impact, so that appropriate action can be taken to maintain consistency between model and test artifacts.

Since the concept of model management aims to keep artifacts consistent, the ability to determine which test values and corresponding models need to be changed based on a given model change would fall directly into this aim.

This chapter explores the design of algorithms used to produce impact analysis reports for a given pair of Simulink models representing two consecutive versions. The goal of this work was that given two consecutive models, a list of inputs and outputs (test values) that were potentially impacted by changes from the first version to the next could be obtained.

There are two steps to this process which are described in detail: (i) change identification and isolation, which is the process of determining the exact changes between one version and the next, and (ii) impact identification, done through forward and back propagation of changes through the model.

5.1 Change Isolation

The first step in determining the impact of changes at the model level on test values is the identification and isolation of changes made to the model. In order to find the impact of change, the changes themselves must first be discovered.

This process is essentially an exercise in model differencing, which is a process that has been studied previously by other researchers, and was also part of the previous co-evolution experiment, where differences were calculated between model versions for comparison with test differences.

For the implementation of the change isolation for impact analysis, even less detail is required than available from the previous implementation, which was able to provide the nature of changes, and the full details of what changed; for example it could identify that a block parameter changed, and list the previous value and the updated value. The extent of what is required for the impact analysis is the identification of which model artifacts (blocks or lines) have changed, and whether or not it was an addition, modification, or deletion.

As such, the model differencing script described in the previous chapter was pared down to produce a list of handles to Simulink objects in the updated model version that had been modified or added, and a list of those in the previous model version that had been deleted - both lists are then combined and produced as output of the first phase of the impact analysis.

As a reminder, the differencing script works by using the built in Matlab comparison tool and stripping away any unnecessary change information, such as layout

```
diffs =
    [1.0495e+04] 'm'
```

Figure 5.1: Example output for model difference phase, showing one change

```
diffs =
    [1.0494e+04] 'm'
    [1.0520e+04] 'm'
    [1.0383e+04] 'a'
    [1.0402e+04] 'm'
    [1.0538e+04] 'm'
    [1.0648e+04] 'm'
    [ 78.0632] 'd'
    [ 87.1476] 'm'
    [ 224.0161] 'm'
    [ 225.0155] 'm'
```

Figure 5.2: Example output for model difference phase, showing multiple changes

information and file metadata, leaving only the semantic changes to the model.

The list produced by the tool is a set of tuples, which pair the object handle along with a single character identifier, showing whether the change was an addition ('a'), modification ('m'), or a deletion ('d').

Figure 5.1 shows sample output that would be produced when a single block has its parameter changed from one version to the next.

From this a developer could navigate to the identified block and see how it changed from one version to the next. Figure 5.2 shows an example from a more complex set of changes between two versions involving addition modification, and deletion:

This first step of the analysis can serve as input the next step, which are discussed in the next section, or can be used as an intermediate output useful to the developer - a list of changes from one version to the next. This list itself can be useful in helping developers identify what has been done since the previous version of the model, which can be used to retrace actions taken, or if they need to make similar changes in related models. Thus, while the change isolation is a first phase of the impact analysis, it can be used in several other use cases as well.

5.2 Determining Impact of Changes on Test Values

Given the list of changes made produced by the previous step, the impact analysis must then determine the impact on test values for these changes. This is done by iterating over each change identified, finding it in the model, and propagating the changes forward and/or backward as chosen by the user. This propagation is done using a recursive algorithm for each direction, that essentially finds the next blocks in the selected direction, and recursively finds the impacted blocks from that block, with the base case being a top level signal that corresponds to an input or output, at which point that identifier is added to the list of values that are potentially impacted.

Figure 5.3 is a pseudocode representation of the algorithms used to determine the impact. The *impactAnalysis* method is the main method for this phase, and takes as input the changes found in the previous phase. For each change, it generates a list of forward and backward changes by making calls to $forward$ and *backward*. Each of these recursive methods work the same way but in opposing directions. If the current change candidate is a top level input or output signal, then the traversal is complete and the candidate is added to the final list of impacted values, otherwise, for each signal moving in the chosen direction (backwards or forwards), if that target has not yet been visited, add it to the visited list and make a recursive call to continue exploration. The reason for the list of visited blocks and lines is to avoid infinite recursion due to cycles in the model; if a model block has been visited, the downstream impacts have already been accounted for.

The end result of this analysis is both an *outputList* and an *inputList*, which are the candidate values for further examination.

Figure 5.4 shows example output of the second phase of this analysis, showing the list of outputs and inputs potentially impacted by the changes.

5.2.1 Understanding the Results

Given the results of this impact analysis, it is important to understand how to interpret and handle these findings. The first and most important point is that the values identified here are **candidates** for impact, and do not guarantee that those values will require change. The goal of this work is to determine the potential impact of a given set of changes and that is what is produced.

Further processing of the results is required in order to use them effectively, and this can be done in one of two ways: either the developer can manually inspect the test values, or they can use simulation to generate potential outputs given existing inputs, which also requires manual inspection following the generation. Both of these options are discussed further in the next chapter, as they deal specifically with the tool implementation of this algorithm.

The key takeaway is that the impact analysis performed is capable of generating a list of potentially impacted inputs and outputs given two consecutive model versions.

```
impactAnalysis (changes)
  for each item in changes
    currentChange = item
    forward(currentChange)
    backward(currentChange)
  end
end impactAnalysis

forward(currentChange)
  if currentChange is TopLevelOutput //if there is nowhere left to go forward
    listOfOutputs.add(currentChange);
  else
    for each item in currentChange.outputs
      if ~(visited.contains(item))
        visited.add(item)
        forward(item)
      end
    end
  end
end forward

backward(currentChange)
  if currentChange is TopLevelInput //if there is nowhere left to go backward
    listOfInputs.add(currentChange);
  else
    for each item in currentChange.inputs
      if ~(visited.contains(item))
        visited.add(item)
        backward(item)
      end
    end
  end
end forward
```

Figure 5.3: Pseudodode listing for impact analysis

```
outputs =
    'SignalX'
    'SignalY'
    'SignalZ'


inputs =
    'SignalA'
    'SignalB'
```

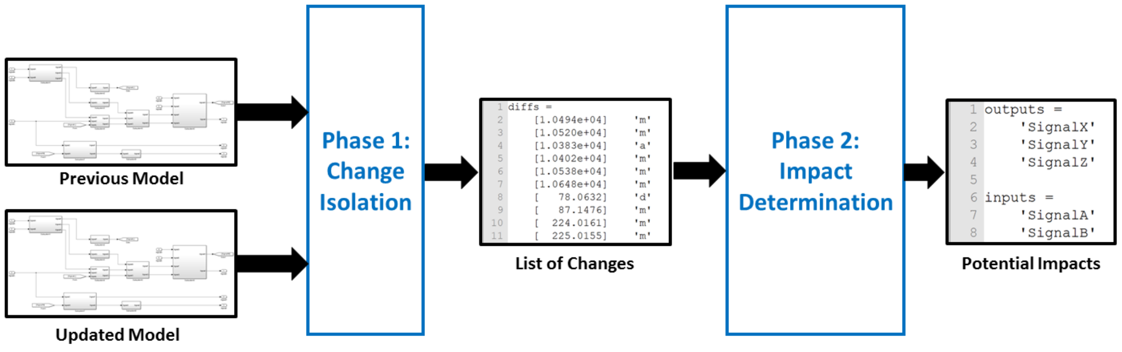
Figure 5.4: Example output of the impact analysis phase

5.3 Summary

In this chapter, the two phase process for performing Simulink model change impact analysis was presented. The first phase involves model differencing to generate a list of changes between model versions (change isolation), and the second determines the potential impact of those changes on inputs and outputs via recursive propagation of changes throughout the model to the top level inputs and outputs (determining impact of changes on test values).

Each of these processes was presented in detail, with pseudocode descriptions and example outputs provided. The whole process can be summarized as a two step computation with two models as inputs and a list of potential impacts as output, with the intermediate representation useful in some use cases (such as manual analysis of changes). This can been seen in Figure 5.5.

With an implementation of impact analysis developed, the next goal of this thesis is to package up the implementation of these algorithms, along with other support for model management, into a collection of tools for industrial use. The next chapter describes the development of a toolset to support simulink evolution, and more



```
diffs =
    [1.0494e+04]    'm'
    [1.0520e+04]    'm'
    [1.0383e+04]    'a'
    [1.0402e+04]    'm'
    [1.0538e+04]    'm'
    [1.0648e+04]    'm'
    [   78.0632]    'd'
    [   87.1476]    'm'
    [  224.0161]    'm'
    [  225.0155]    'm'
```

```
outputs =
    'SignalX'
    'SignalY'
    'SignalZ'

inputs =
    'SignalA'
    'SignalB'
```

Figure 5.5: Overview of Impact Analysis by Change Isolation and Impact Determination (artifacts in black, processes in blue)

generally model management.

Chapter 6

Simulink Evolution Tool Support

With one of the goals of this thesis being tool support for model management in an industry setting, it was important that a tool be designed that could be provided to our industry partners that implemented the impact analysis work that was completed, but also offered other support for model management of evolving models. To this end SimEvo, the Simulink Evolution Tool, was developed as a single interface to several tools aimed at supporting model management.

The particular set of tools that were developed for this purpose were SimPact, a tool for Simulink impact analysis, and SimTH, a Simulink Test Harness Generator. Each of these tools can be standalone tools, but are also incorporated in SimEvo, and are discussed in detail in this chapter. The third component of SimEvo is the support for interaction with existing tools used by our industry partner, which takes two different forms: calls to some of their processes useful for model management, and a visual integration via similar user interface design, the first of these is described in a Section 6.3.

As inputs, SimEvo accepts two versions of a model (previous and updated) and the previous version of the test suite, and is able to produce as outputs a test harness, potential impact lists, or suggested test output values via calls to existing tools. Not all inputs and outputs are required for each run, and are optional based on the use case, several of which are presented later.

A diagram of the SimEvo architecture can be seen in Figure 6.1.

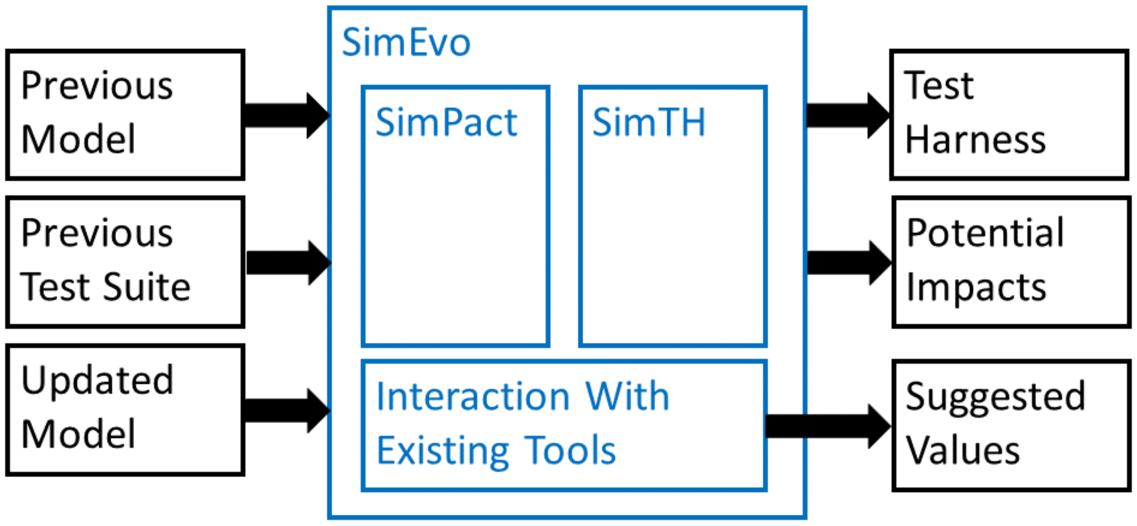
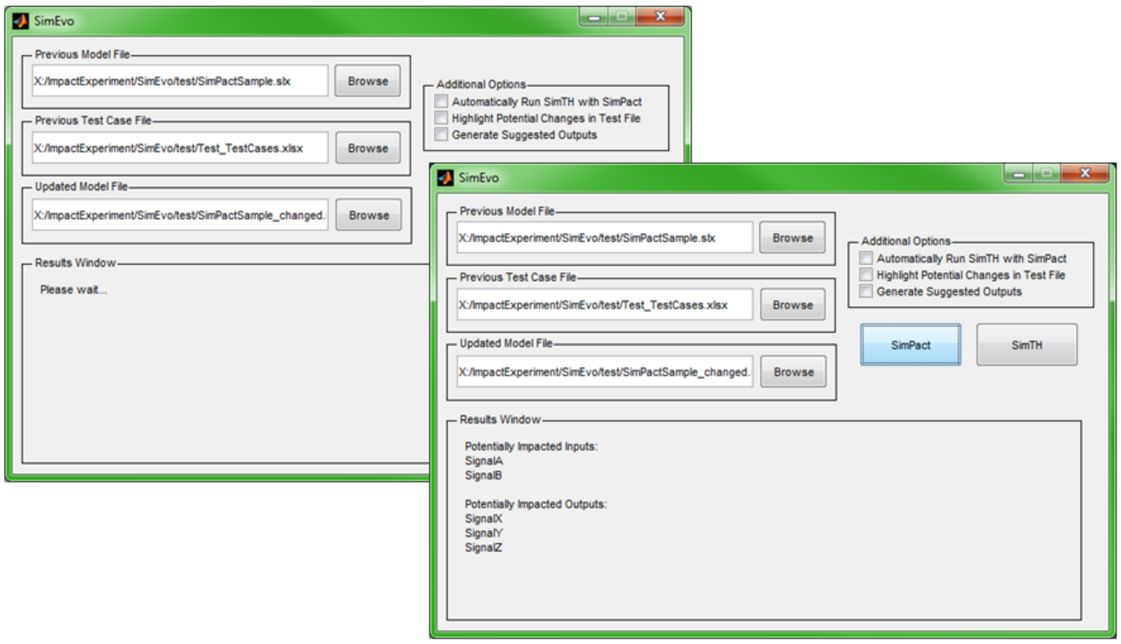


Figure 6.1: SimEvo Architecture (artifacts in black, processes in blue)

An important part of tool design for an industry setting is the production of a tool that will fit into an existing workflow without too much disturbance of the status quo. Thus one of the goals in designing SimEvo was to create a graphical user interface (GUI) that was visually and functionally similar to the GUI of existing tools used, to avoid issues of context switching between applications. By using similar layouts, coloring, and option selection styles as existing tools, SimEvo appears to be from the same suite, has a familiar control structure, and could easily be incorporated into the existing industry tool. Unfortunately it is not possible to display the existing GUI of the industry tool, however the SimEvo GUI can be seen in Figure 6.2.



X:/ImpactExperiment/SimEvo/test/SimPactSample.slx

X:/ImpactExperiment/SimEvo/test/Test_TestCases.xlsx

X:/ImpactExperiment/SimEvo/test/SimPactSample_changed.

Please wait...

X:/ImpactExperiment/SimEvo/test/SimPactSample.slx

X:/ImpactExperiment/SimEvo/test/Test_TestCases.xlsx

X:/ImpactExperiment/SimEvo/test/SimPactSample_changed.

```
Potentially Impacted Inputs:
SignalA
SignalB

Potentially Impacted Outputs:
SignalX
SignalY
SignalZ
```

Figure 6.2: SimEvo Graphical User Interface

6.1 SimPact: Simulink Impact Analysis

The first component of SimEvo is SimPact, the tool used to perform Simulink impact analysis. Chapter 5 explained in detail the algorithms used to determine the potential impact of changes between two versions of a model; SimPact is essentially the tool implementation of that process.

Initially, SimPact was only a Matlab command line script, which took as input two consecutive model versions, and produced the list of potential impacts as textual output, however this needed refinement which came in the form of two use cases: a static analysis tool and a dynamic analysis tool. The static analysis implementation of SimPact works on two versions and produces the list of total impact between those two versions, where the dynamic analysis implementation is a plugin to do real time examination of potential impacts for selected Simulink blocks. Each of these implementations is discussed below.

6.1.1 Static Analysis

The static analysis implementation of SimPact works on two static model files, consecutive versions, and does an analysis of change similar to that described in Chapter 5, providing the list of potentially impacted inputs and outputs.

This type of analysis is useful after a number of updates or changes have been applied to a model and the developer wants to obtain the potential impacts of those changes - consider this as batch processing of changes made since the last update.

Originally a command line only program, since its integration into SimEvo, this process is done by using file browsers in the GUI to select both the previous and updated model versions, and clicking on the SimPact button; the previous test case file is not used in the basic option set for SimPact. The analysis is performed and the results are shown directly in the GUI, as shown in the example in Figure 6.2.

There are several options that apply SimPact specifically that can be chosen for the dynamic analysis: Highlight Potential Changes in Test File, and Generate Suggested Outputs. For both of these options, the previous test case file is required. If the first option is checked, SimPact, in addition to identifying the potentially impacted values in the GUI, highlights the columns that contain the potentially impacted values directly in the test case file, and save a new version of it for manual inspection; this saves the developer from having to find all of the values identified by hand. The second option takes this one step further, and makes use of existing tools to generate suggested output values for the previous input values. This is done by making calls to other tools, and is described further in Section 6.3.

6.1.2 Dynamic Analysis: A Simulink Plugin

The dynamic analysis implementation of SimPact came out of discussions with our industry partner regarding use cases for an impact analysis tool, particularly the desire to select a model block, and see, directly in Simulink, what the potential impact of changing that block would be.

The need for this type of analysis comes from the inherent complexity of the multi-level hierarchical models developed in Simulink; it can become far too difficult to track these changes and impacts by hand. For example, Figure 6.3 shows a system in the top left corner, and if a developer wants to know the impact of change the block in the center (a sum block) to perform some other calculation, they can easily see the impact at that level, but as you move up the model hierarchy, it becomes more and more difficult to determine where in the model change occurred, and almost impossible to enumerate all of the impact.

It was this realization from discussions that led to a tool capable of highlighting changes at all levels, as well as the impact of those changes to surrounding blocks, and ultimately the top level inputs and outputs. Based on this it was determined that a plugin, capable of interacting with Simulink models directly, would be the most effective implementation.

The actual calculation of impact is identical to the method for the static analysis, however there is no determination of changes; the block that is selected is set as the only change, and propagation of change occurs in either or both directions, based on which option is chosen. Essentially the dynamic analysis uses only the second half of the static analysis.

Since there are multiple use cases for this type of tool it was determined that there

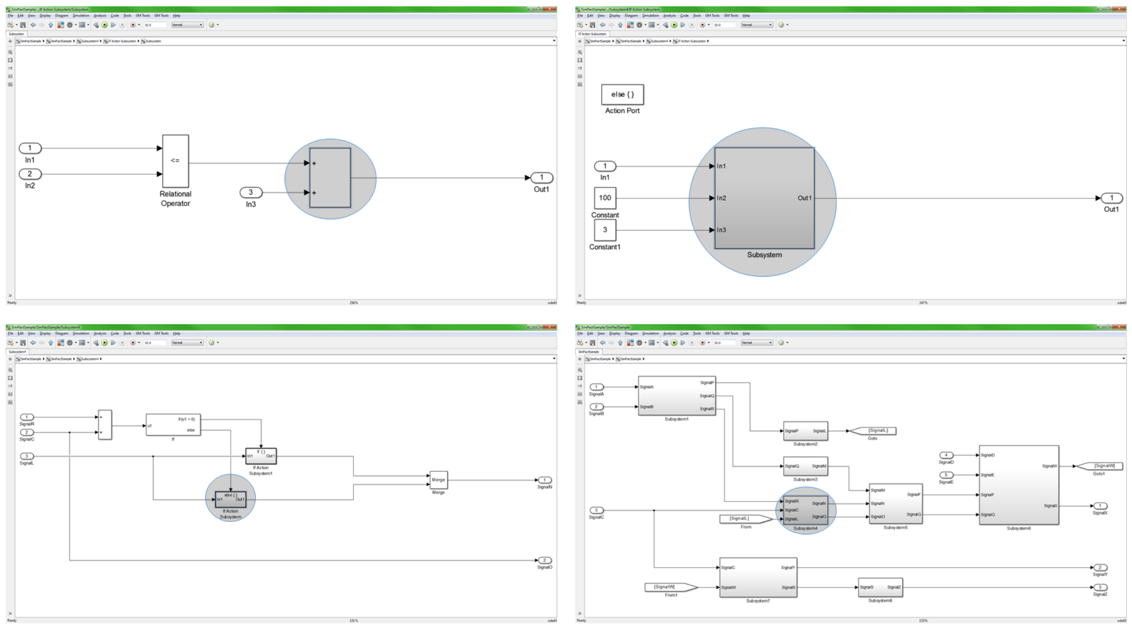


Figure 6.3: Trying to Track Changes Manually

should be three options for highlighting the impact in a model: forward only, backward only, and both directions. Each of these options is added to the Simulink context menu for blocks by creating a custom *sl_customization.m* file with the required code; this special file is Matlab's method of plugin development. Figure 6.4 shows the three additions to the Simulink context menu, when right clicking on a Simulink block.

As a use case, if a user were to select ”Highlight All Impact” on the sum block used in the example showing the difficulty of manual tracing, it would highlight the block that was changed in yellow, and any impacted blocks are outlined in red. This can be traced up through the model, with the containing system highlighted in yellow, and any impacted blocks again outlined in red. Figure 6.5 shows this highlighting for that same change at all model levels.

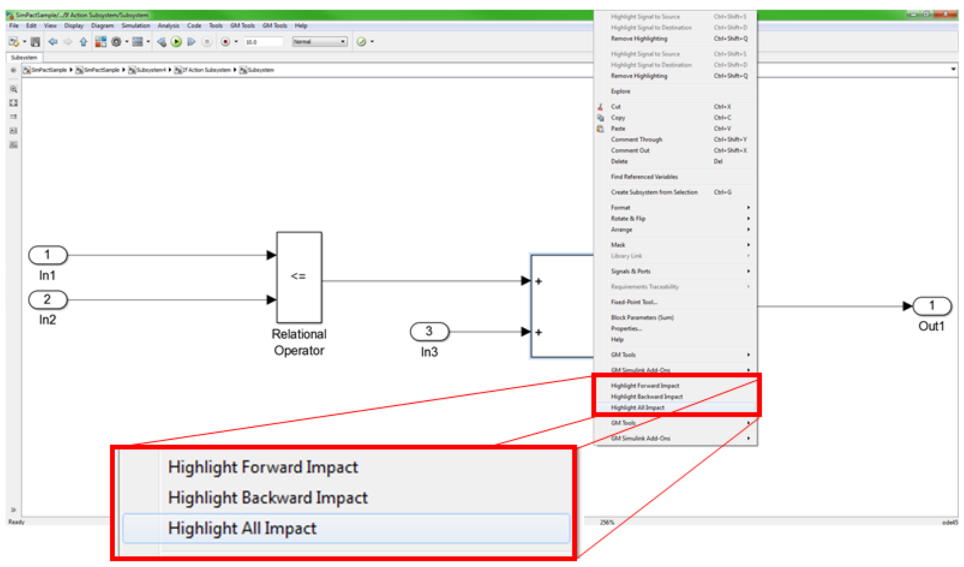


Figure 6.4: SimPact added to Simulink Block Context Menu

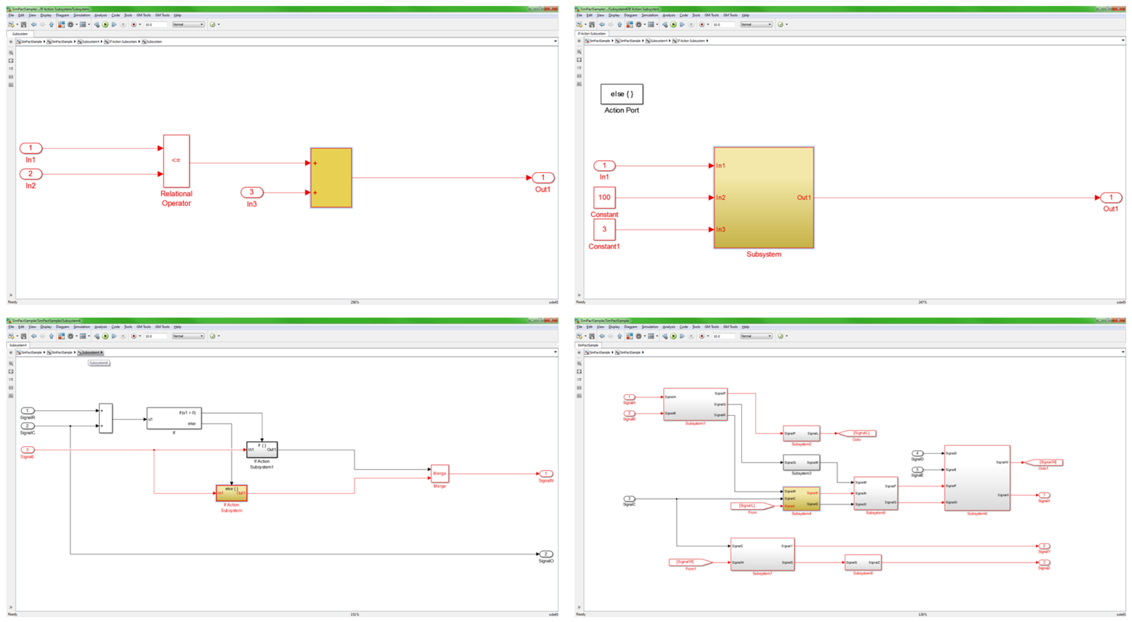


Figure 6.5: SimPact Dynamic Analysis Plugin highlights impact at all levels

6.2 SimTH: Simulink Test Harness Generation

Software testing plays an important role in the lifecycle of a software system, and opportunities for enhancing or streamlining testing make for important enhancements in the software process. Our industrial partners work in model-based development of software for automobiles, particularly Simulink[2] behavioural models, from which code is automatically generated. As new models are added or existing models updated to enhance functionality, new tests must be added and existing tests updated to correspond. This adaptation was initially a manual process in the toolchain, and ripe for automation.

To address this need, SimTH was developed, a test harness generator for Simulink automotive models, that is capable of taking as input the Simulink models produced by engineers, and automatically generating the required test harness to test the model using their spreadsheet-based testing tools.

Testing can be an expensive process which takes considerable resources for a software project[9]; testing includes requirements reviews, test case design, test creation, and test execution. The overall goal is to assist in automating these processes, specifically test creation in the case of SimTH via test harness generation. By automating a manual process, sufficient savings in time and effort can be observed throughout the testing process. Previously, test harness models were manually developed based on template files and a set of conventions which allowed engineers to create test harnesses for new models, or make updates to them when changes are made to the source model. Manual creation of testing artifacts is a difficult task on its own, and the responsibility of updating tests and test harnesses to reflect evolutionary changes to the models can be even more difficult and time consuming.

The testing community, as well as our industry partners, see automation of the test harness generation and update process as a desirable goal for a number of reasons. From the testing community perspective, automation allows for efforts to be spent in other areas of the software process, freeing up valuable resources. Manual creation and maintenance of test models, as is the case with most software artifacts, is more prone to errors being inadvertently introduced, leading to more time spent correcting them. Automation reduces the opportunity for these types of errors to be introduced. 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. With this more widespread use of the tool, many engineers expressed a desire for support in developing test harness models that conform to the tool's requirements; support which would be costly to provide. The second reason a tool for this purpose was requested by our partners is that with an expansion to the set of models being maintained by our collaborators' group, the amount of work going into maintaining these models has greatly increased. Reducing the test harness maintenance effort would allow them to spend more time on other tasks.

Due to both the contributions to testing process and the strong industrial demand for an automated test harness generator for these reasons, it was evident that a tool could be developed for this purpose, as it fit directly in line with goal of model management.

A test harness is designed to adapt a piece of software in such a way that it can be executed with test values and the results observed for comparison; essentially a test harness supports automatic execution[7]. The next section details how this process occurs as a part of our industry partner's software process.

6.2.1 Testing Simulink Automotive Models

In order to understand the contributions of SimTH, 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.

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 may be many for the same model.) SimTH takes care of producing the test harness model, which augments the existing model with Simulink blocks for reading values from test case files, simulating calls to the operating system, and other features which are presented in the next section. Initial specification of the test case values themselves is currently a manual process, however SimPact is useful for supporting the ongoing updates to these test values.

The test harness models required by the testing tool appear similar to the source models in appearance, and at all levels beyond the top level, are identical in functionality. This is due to the fact that the harness simply imports the system block as a library link, and provides all of the necessary workspace context and environment simulation to the testing tool so that it can perform the simulations. An example test harness model, like one that would be used by our partners, can be found in

Figure 6.6.

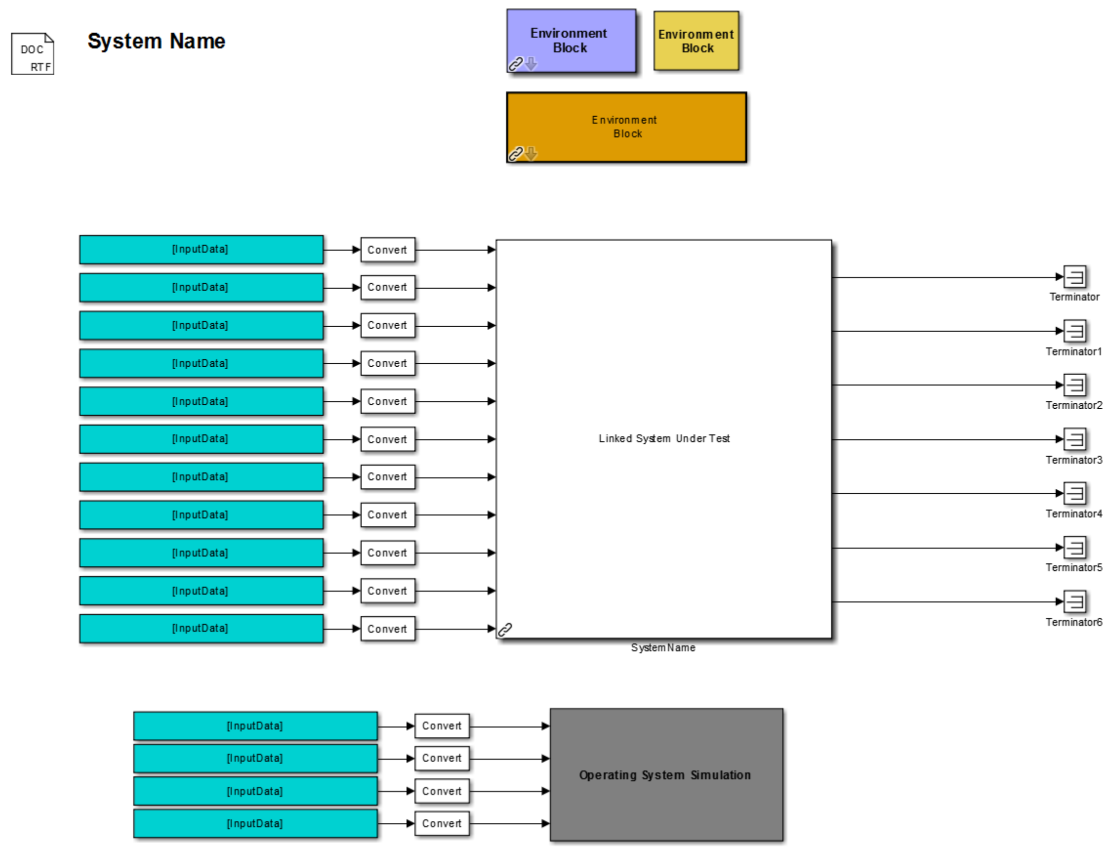
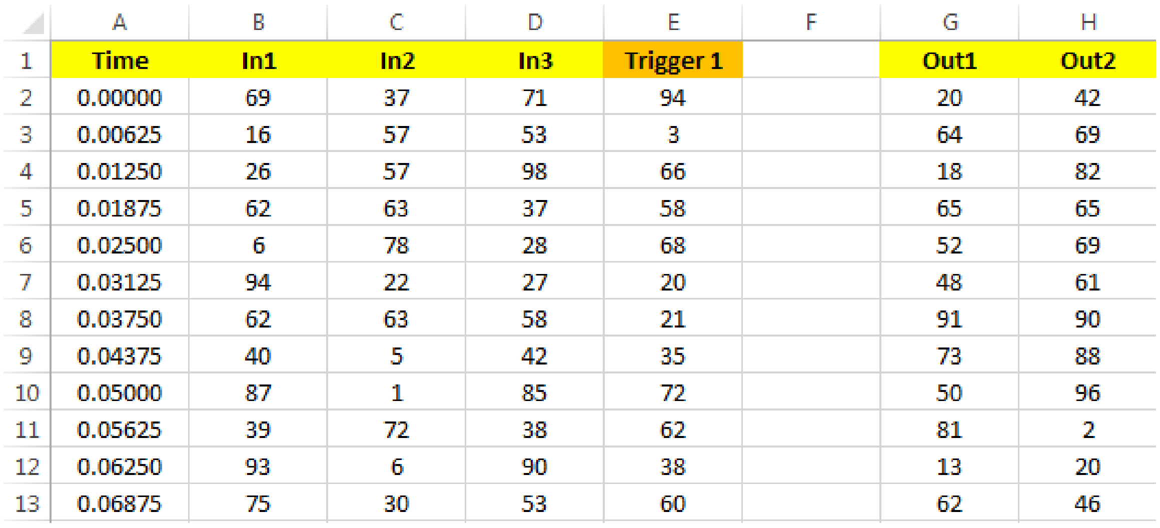


Figure 6.6: Sample Test Model with FromWorkspace blocks, and operating system simulation block

The test cases themselves 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, with the workbook as a whole representing the entire test suite. In each tab, there are column headers specifying the input signal names, simulation trigger names, output names, and the left most column is always be the time in seconds. Each row represents a single time step and the values at that given time. An example test case tab can be seen in Figure 6.7.



	A	B	C	D	E	F	G	H
1	Time	In1	In2	In3	Trigger 1		Out1	Out2
2	0.00000	69	37	71	94		20	42
3	0.00625	16	57	53	3		64	69
4	0.01250	26	57	98	66		18	82
5	0.01875	62	63	37	58		65	65
6	0.02500	6	78	28	68		52	69
7	0.03125	94	22	27	20		48	61
8	0.03750	62	63	58	21		91	90
9	0.04375	40	5	42	35		73	88
10	0.05000	87	1	85	72		50	96
11	0.05625	39	72	38	62		81	2
12	0.06250	93	6	90	38		13	20
13	0.06875	75	30	53	60		62	46

Figure 6.7: Sample Test Case tab from an Excel Spreadsheet, using random values to demonstrate format

With all of the artifacts prepared (test harness model and test suite workbook), the testing tool used by our industry partners is able to simulate the model, one run for each test case in the test suite, using the specified time steps and input values, and monitor 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 based on the tests, normally reported as differences between expected and actual outputs.

6.2.2 Test Harness Generation: SimTH

In order to produce a working test harness for an automotive model, SimTH requires as input only the behavioural model developed by the industrial engineer. These behavioural models are the main artifacts maintained by the engineering team, and contain all of the functionality and logic for the models; they are also the models from which code is eventually generated. Because this is the only input required by

SimTH, it means that no additional effort is required, thus the tool does not introduce any new requirements on the developers themselves, which is an important feature when introducing tools into an industry toolchain.

This section describes the internal logic of SimTH, walking through the steps it uses to create test harness models. There are 4 main tasks as part of harness creation: inclusion of the system under test, input and output management, environment simulation, and standardization. These four steps are introduced generally, and then the specific implementation for SimTH is presented.

After each of these 4 steps are completed, a final test harness is produced meeting all of the requirements of our industry partner. Each of the 4 steps map directly to some component (or set of components) in the completed test harness. This can be seen in Figure 6.8, which overlays the steps on the sample test harness model.

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. Because of this, the first step in creating a functional test harness is the incorporation of the model's functions into the test harness model by including required information into the test harness.

To implement this in SimTH, the Simulink functionality for library linking is used, which allows a model to import the contents of another model (or block) as a library link. Simulink library linking allows both the possibility that changes made to the linked model are propagated back to the original, or that an independent instance of the library is created.

SimTH investigates which library model is linked to by the behavioural model,

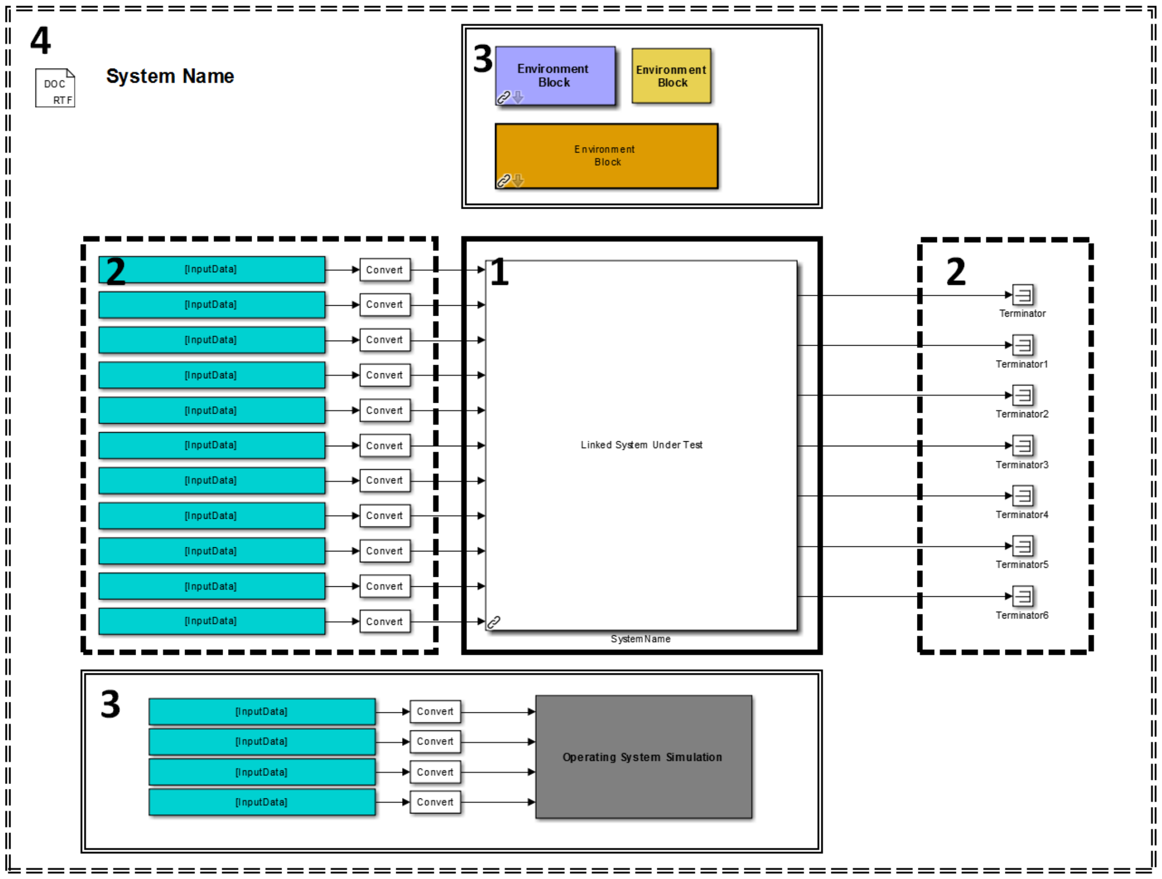


Figure 6.8: Conceptual Diagram of Implementation Steps for Test Harness Generation

and creates a link to the same library in the newly created test model. After the first step, the only element in the test harness is the linked library block. From the linked block, 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.

With the library linked, the system has been included in the test harness, and further processing can occur.

Step 2: Input and Output Management

The next major step is the management of the inputs and outputs for the system being tested. 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 correct inports in the model, as well as monitoring the correct output values for further comparison.

To implement input and output management for SimTH, the interface of the subsystem blocks (contained in the linked library) being tested are examined. The block's interior has some number of inports and outports (Simulink terminology for inputs and outputs) which translate to the labels on the model block itself - traditionally inputs are on the left edge and outputs on the right.

Beginning with the inputs, SimTH iterates over them one at a time to handle creating the requirements for the test harness. Each inport has a designated signal name (which matches the names in the test case files) as well as a signal type; SimTH pulls this information from the 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 it is parameterized by identifying that it reads 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, and the blocks are aligned with the input ports, as shown on the left hand side of Figure 6.6. A number of input types 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 cases such as these, SimTH is able to infer the required conversions based on types and naming conventions, and generate the necessary conversion blocks between the FromWorkspace block and the subsystem being tested. Once this is complete, the input portion of the test harness generation is complete.

The outputs of the behavioural model are used to provide inputs to other blocks outside the scope of the subsystem being tested. For the purpose of testing, output values are only useful for comparison to the expected values, which is done by their testing tool observing the simulation - thus the outputs of the test harness model are not required beyond its scope. To manage this, SimTH simply sends each output of the subsystem being tested to a special block known as a terminator; this ends the signal at this point. Examples of sending signals to a terminator block are shown on the right hand side of Figure 6.6.

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 the environment, which could cause additional errors for untested systems. This is done by simulating calls from the environment, and providing accurate responses to calls to the environment. Ensuring an accurate simulation of an environment by the test harness is of extreme importance for testing the system.

There are two parts to this implementation in SimTH, the simulation of the OS calls themselves, and the inclusion of various other environment variables and blocks. This section addresses both of these areas in detail.

The main logic of SimTH consists of creating a subsystem block capable of simulating the necessary parts of the operating system for the vehicle. A given subsystem in a vehicle will need to interact with the operating system in various ways during execution, such as receiving signals from predefined flags. Since during testing, the subsystem is tested in isolation, it needs some representation of the OS capable of providing all of the required functionality, but without actually running a version of the OS, as that would be far too complex for what is required to test on the small scale.

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 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, which is imperative in obtaining realistic test results.

In the sample test model in Figure 6.6, below the subsystem being tested, there is a dark grey block with a number of simulated inputs; this is the Operating System Simulation block that is required by the testing tool, which must be generated by SimTH in order to produce a valid test harness model.

The number of different OS calls that can be made from models for the particular portion of automotive software developed by our partners fairly 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 create the required blocks in the OS simulation block to handle them. There are three main categories of OS simulations that SimTH must create in the simulation block: periodic events, triggered events, and initialization events.

There is one type of event that the OS sends to the model on a consistent basis, and this is a periodic event. The periodic event behaves exactly like a ticking clock.

Every time step (determined when examining the linked library model) sends a signal to the model that the period has occurred. The periodic event is the simplest event to simulate, as it is implemented as a timed signal generator, which is set to generate a signal at a determined interval, that then triggers all required periodic events in the subsystem being tested. All models have at least one periodic event, but it is possible that more than one exists if there are events that need to occur on offset intervals (e.g. something occurs every 100ms and something every 6.25ms).

The second type of event that needs to be simulated for the OS is initialization events. Initialization events are special events that are used when the system starts up, to simulate the initialization of the controller on which the software (or in this case model) is running. These are implemented similarly to the periodic events.

The final main type of event that needs to be simulated for the OS is triggered events. Triggered events are events that only occur when a specific trigger in the OS causes them to occur. These triggers may come from other systems, or as a result of another stimulus. However, in testing the model, these events are simply triggered at desired times by setting a flag in the test case file. These flag values are read from the workspace in the same way as input values are, and constantly updated at every clock cycle. The values are then passed to a special block that when the trigger is received, then in turn triggers the specific event in the operating system. These triggered blocks also contain the same time based signal generator as the periodic blocks, but the signal is only allowed to leave the block and trigger the event in time steps when the flag has been received from the test case. Essentially, the triggered event blocks behave like a periodic block, but with the flag as a guard against sending the event signal at the wrong time. It is entirely possible that small simple models may not have any triggered events, but the majority have several of them. For example, the model in Figure 6.6 has four triggered OS events, as shown by the two inputs to the OS simulation block.

The internal view of a sample operating system simulation block can be seen in Figure 6.9. This OS simulation block has 3 triggered events in addition to the initialization and periodic events, all for a 12.5ms time step. This figure shows that the periodic timer (12p5ms) is used as a trigger to all of the event blocks, but internally they all handle it in they specific way their design requires. Notice the three triggered blocks in the center all have inputs coming into them from outside of the OS simulation block, these are the flags set by the test case. Along the right are the goto blocks that actually make the calls to events in the model itself.

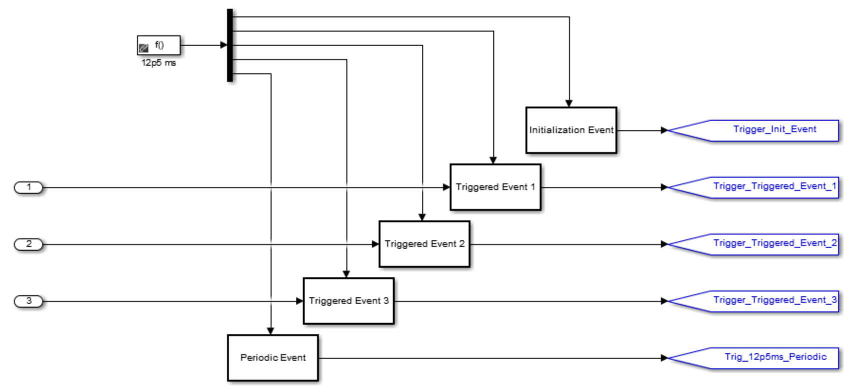


Figure 6.9: Sample Operating System Simulation Block (internal view)

SimTH fully automates this process by examining the required calls in the model, and reverse engineering the correct OS simulation block for the model.

Beyond the 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, that once done, signal the creation of a fully functional test harness model.

The first required block is the Controller Scheduling block. This is another dynamically created block that is used to assist in scheduling events. It acts as an interface between the OS simulation block (discussed previously) to the places these events are called from in the model. Functionally the Controller Scheduling block is very simple, as it is a one-to-one mapping of calls from the OS simulation block to tags that correspond to calls in the model. Figure 6.10 shows the internal view of a sample controller scheduling block, corresponding to the OS simulation block shown in Figure 6.9.

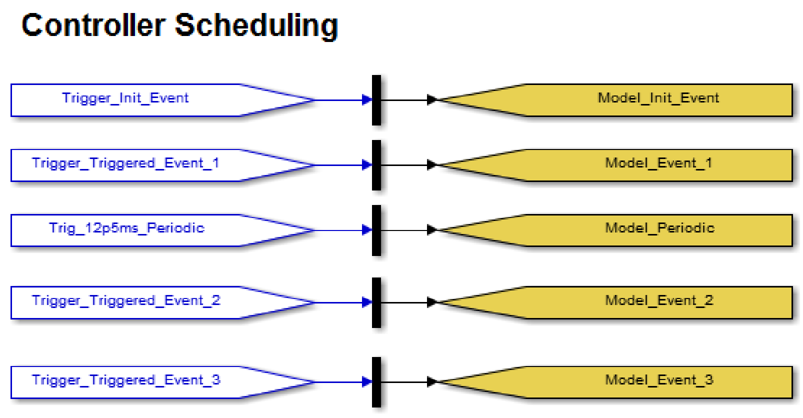


Figure 6.10: Sample Controller Scheduling Block (internal view)

The second type of required block that is created as part of the environment is a block containing definitions of custom data types specific to our industry partners. This block is simply added directly from a library, and no additional work is done by SimTH, but it is necessary for the test harness to function.

The final type of block that is part of the environment setup is responsible for implementing additional diagnostic functionality. This block is only necessary in models that interface with this portion of the automobile system, and thus is only added when calls are made through this interface. SimTH can discern the necessity of this block by examining the linked library, and if needed can add it from the same library as the previous block.

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

Once the environment is setup with all of these parts, the model test harness created by SimTH is fully functional and can be run using the testing tool to validate models. However, since the model was constructed entirely automatically, if a developer ever needed to inspect it due to errors, it would be near impossible to investigate as it doesn't have any layout properties, and is so far from what the developer is used to in manually created test harness models; the next section addresses this issue.

Step 4: Standardization

The last part of test harness generation focuses on producing a result (in this case the test harness model itself) that conforms to existing standards for testing in the organization. This is merely to ensure that beyond the functionality covered by the previous sections, the generated test harness would appear to developers as something they are familiar with.

The main concern is that SimTH should produce test harness models that are visually similar to those developed manually from the template files used by our industry partners. This means that the layout needs to be simple enough to understand, blocks need to be the same colours and sizes as expected, and labeling and naming conventions must be consistent. These visual standards for models need to be adhered to from a human interaction standpoint, and while they do not impact functionality, proper layout and other standards are helpful in the continued maintenance of the software, which is one of the key goals of SimTH as a tool.

SimTH is capable of producing test harness models that are visually identical to those manually generated, aside from one additional label at the top level that indicates that it is a generated test model.

6.3 Integration with Existing Tools

The last major contribution of SimEvo is its ability to interact with an exiting testing tool developed and used by our industry partner. The use case for this interaction is one of the described follow ups to impact analysis - when provided with a list of potentially impacted inputs and outputs, developers are left with two options: manual inspection or generation of suggestions; this interaction handles the second option.

The in house testing tool developed by our industry partner, which was discussed previously, is capable of taking a test harness and test input values to simulate and generate the expected outputs for the provided inputs. Since this provides a developer doing impact analysis with candidates for possible updated test values, this is a useful option to have. SimEvo provides, via method calls, the appropriate test harness, which it generates using SimTH, along with the previous input test values, and the testing tool simulates the test harness with these values, observing and recording the outputs for later use.

This option is accessed by selecting the third option in SimEvo, “Generate Suggested Outputs”, and using SimPact with all three inputs provided. The first step is it performs static impact analysis between the two model versions using SimPact, then it generates a test harness for the updated model using SimTH, and finally it passes the previous test case file and newly generated test harness to the testing tool to simulate and output candidate test case output values. If the “Highlight Potential Changes in Test File” option is selected, the columns with potential impact is highlighted, while those that have not been impacted appear as usual.

6.4 Summary

In this chapter, the combined tool support for evolving Simulink models was presented in the form of SimEvo, and its contained tools SimPact and SimTH.

SimPact is capable of performing impact analysis, both statically on two model versions and dynamically on an interactive model, providing as output the list of impacted input and output values, which are of importance in maintaining the model-based tests, a large part of model management. The addition of the dynamic analysis feature to the Simulink context menu is one of the features that sets SimPact apart from other similar tools, along with its direct implementation in Simulink rather than as a standalone tool.

SimTH is a test harness generation tool capable of automatically generating test harnesses for industrial Simulink models. The harnesses generated are used by an existing tool to evaluate a model against test cases. The automation of this process contributes significant progress both in terms of increased speed, but also in increased consistency.

SimEvo is a tool that combines both SimPact and SimTH into a single GUI, which fits into existing tool chains used by our industry partner, and is capable of effectively interacting with the existing tool to provide valuable feedback and information in addition to the impact analysis performed by SimPact.

Some of these tools have been used by our industry partner with positive feedback, and interest in further collaboration to continue support for model management for evolving Simulink models.

As is the case with newly developed tools, appropriate validation of their effectiveness are required; this validation is conducted in the next chapter.

Chapter 7

Validation

When new tools are developed and introduced into an existing workflow, it is important to ensure that they are first performing as desired. This chapter outlines the validation of both SimPact and SimTH, the two new implementations introduced in this thesis, by means of experiments demonstrating their capability and comparing against benchmark values.

7.1 SimPact

SimPact is essentially a prediction tool; its aim is to predict what inputs and outputs have been impacted by the changes made to a model. As such, the validation of its effectiveness was conducted by calculating the precision and recall of its predictions against the ground truth of observed test case changes. This section details the experimental design, observed results, and present discussion on the findings.

7.1.1 Experimental Design

In order to observe the effectiveness of SimPact's prediction abilities, a comparison against a ground truth is necessary. To determine the precision and recall, experiments were run on the entire model set described in Chapter 3.

For each of the 8 of the evolution steps (i.e. the move from one version to the next e.g. release 13 to release 15), a four step process was taken:

1. SimPact was run on every pair of models in the evolution step to predict input and output changes
2. a differencing of the corresponding pairs test case files was done to extract the actual changes in inputs and outputs
3. precision and recall were calculated for the inputs and outputs or every pair
4. the average precision and recall for inputs and outputs was calculated over the entire evolution step

With four results for every evolution step (input precision, input recall, output precision, output recall), a final averaging of these results was calculated to obtain a precision and recall value over the entire model set for both inputs and outputs. As a final step, the harmonic mean was calculated for both inputs and outputs.

Each of the above steps is described in further detail below.

SimPact for Change Prediction

For every pair of consecutive model versions, the static analysis implementation of SimPact was run to determine what the potential impact on the test cases would be. The result of this step for each pair was a list of impacted inputs and outputs.

Observing Actual Test Changes

An extension of the test differencing script described in Chapter 4 was developed for this experiment. The extended script takes as input two test case files, and iterates over each of the inputs and outputs to compare them against the corresponding inputs and outputs in the other test case file. For each difference observed (including the addition or deletion of input or output signals) that change is noted, and the final output is a list of which input and output vectors have changed between the two versions.

Since these changes are observed from the actual test cases which were successfully run on the updated models, they form the ground truth for the required changes to the test cases, and were used for comparison in this experiment.

Calculation of Precision and Recall

For each evolution step, a list of predicted and actual changes were obtained, and using the formulas below, precision and recall were calculated. Since developers may have different use cases, and due to the bidirectionality of the prediction ability of SimPact, the precision and recall scores were calculated separately for inputs and outputs.

The following definitions and calculations were used for this step of validation:

- **True Positive (TP)** - Test Values that were predicted by SimPact that actually changed
- **False Positive (FP)** - Test Values that were predicted by SimPact that did not actually change
- **False Negative (FN)** - Test Values that were not predicted by SimPact that actually changed
- **True Negative (TN)** - Test Values that were not predicted by SimPact that did not actually change

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

Combining Precision and Recall Metrics

With precision and recall for input and output calculated over every pair of models over the 8 evolution steps, the 424 4-tuples were not very useful in determining how effective SimPact is at predicting change. The average precision and recall were calculated for every model in each of the evolution steps, leaving 8 4-tuples of results, which in addition to reducing the values provides meaningful results, as it shows the effectiveness of SimPact at different stages of development (early, nearing major releases, approaching final release, etc.).

In order to provide further combined results, the average of these 8 averages was calculated to provide precision and recall for input and output over the entire model set. Finally, the precision and recall values were combined into an F-measure to provide their harmonic mean using the formula below. Thus the final, highest abstraction, result is an F-measure for input and an F-measure for output.

$$F - measure = \frac{2 * (Precision * Recall)}{Precision + Recall}$$

7.1.2 Results

The results of the validation experiment to find the precision and recall of SimPact are shown in Table 7.1.

Table 7.1: Precision & Recall of SimPact

	Input Precision	Input Recall	Output Precision	Output Recall
release 1-3	0.66	0.67	0.58	0.65
release 3-6	0.91	0.91	0.85	0.85
release 6-7	0.99	0.98	0.99	0.97
release 7-8	0.76	0.83	0.61	0.62
release 8-10	0.89	0.90	0.85	0.85
release 10-11	0.80	0.81	0.84	0.85
release 11-13	0.98	1.00	0.98	1.00
release 13-15	1.00	1.00	0.96	0.95
Average	**0.87**	**0.89**	**0.83**	**0.84**

Input F-Measure	**0.88**
Output F-Measure	**0.84**

7.1.3 Discussion

Overall the results for SimPact are very promising, with a few areas for improvement, or rather discussion of the results.

To begin, the overall performance of the tool is impacted by three particular evolution steps: release 1-3, release 7-8, and release 10-11. These three results are the worst performing, however it is important to realize that these three are the same evolution steps identified in Chapter 4 as those prior to a major release in the project - thus there is an increased number of changes occurring in these evolution steps. If these comparisons are removed from the results, the F-measures for input and output increase to 0.93 and 0.95 respectively. From this, it is possible to determine that SimPact works more effectively on smaller amounts of change, and this is not necessarily a drawback of the tool, but rather the intended use case. Since this experiment was run using the predefined releases as the steps for change, the number of changes is much more than one would typically use a tool like SimPact for. The more common usage would likely be on a day to day basis to determine the impacts of that days changes, rather than at the end of the release, capturing seven weeks of changes. It is this realization that boosts the confidence in SimPact's ability to predict the impact of changes to a model on test input and output values.

When performing a prediction experiment, the typical areas of concern are the false positives and false negatives, both of which come with an explanation. While the numbers of false positives and negatives is still rather low, it is worth investigating.

When it comes to false positives, these are simply inputs and outputs predicted to change by SimPact which did not actually change. Recall the experiment from Chapter 4, which concluded there were a number of model changes that simply did not require changes in the test; these form part of the collection of false positives. Further to this, from the perspective of software quality, the false positives are of relatively little concern, as it simply means that additional scrutiny was used in examining the changes.

The largest area of concern is the presence of false negatives, which are the changes made to the test values that SimPact was not able to predict, since these mean changes could possible be left out as they have not been identified. The overall rate of false negatives is fairly low over the entire set, with the recall values being consistently higher than precision, which is a positive result, however some investigation into the cause of the false negatives was needed. While it hasn't been confirmed, from an examination of the cases of poor performance and the existence of false negatives, it appears that the missed changes in output values are caused by changes in input values, which would have also been missed; essentially the change in the test values was not due to the change in the model, but a change in the test inputs, something

SimPact does not have access to during it's prediction.

7.2 SimTH

A tool such as SimTH is used to assist and improve the software process by automating a process that was previously done manually - as such it is difficult to measure improvement in terms of performance when the methods are so dissimilar. However, one metric of success for SimTH is the ability to produce test models that are functionally equivalent to those developed by hand. In addition to the functional correctness, while there is no benchmark to measure against, it was nonetheless interesting to measure the time required for automatic generation of test harness models using SimTH, as a potential benchmark for future work, and as a baseline for other experiments.

To validate SimTH, two experiments were conducted: (i) a timed execution of SimTH, 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 models produced by SimTH are run with the testing tool to determine if they produce the same results as those generated manually.

The specific experiments and test model sets are described in the following sections.

7.2.1 Experimental Design

There are two parts to the validation experiment: determining if SimTH is capable of generating test harnesses for the set of models (and how efficiently), and determining whether the test harnesses generated by SimTH produce the same results as those manually created when run using the testing tool. These two experiments are described below.

Harness Generation Ability and Efficiency

The first experiment focuses on whether or not SimTH is capable of generating a test harness for the entire set of models. This does not focus on the correctness of the generation, simply the ability to take as input a model created by developers, and produce as output a Simulink model for testing the system.

To validate this metric, calls were made to SimTH over the entire model set in succession, documenting the success or failure of SimTH to create an output model. Following the execution of this script the results are summarized and presented to include the number of successful and unsuccessful generations. The goal of this work is a 100% success rate on the subset of our industrial partners' model set which are directly accessible for this experiment.

In addition to the ability to generate test harnesses, timing results were generated for the generation of these test harnesses, primarily as a baseline for future improvements; as any automated process will certainly be faster than the current manual process. This is done by adding a timing element to the automated script to get an execution time for each run of SimTH on the model set, and is reported for each individual.

Harness Correctness in Test Execution

The second experiment focuses on whether the test harnesses generated by SimTH are functionally equivalent to those provided to us that were created by the developers manually. The main concern here is whether or not the automatically generated harnesses from SimTH can produce the same results after being run with the same test suite files as the original test harness models. If the results are the same, this would be considered a correct generation by SimTH, and evidence that the tool is reliable for its desired purpose.

This experiment was carried out by running the testing tool on a subset of all of the models from the previous experiment, specifically the final release version of the models. This particular subset was chosen because it contains a version of every model in existence at the final release in the set, and is an accurate representation of all of the types of models that are contained in the whole set. The reason for reduction of the overall set is that the time taken to execute the test suites is substantially larger than the time required to generate the test harnesses, and the representative sampling of models still presents valid and interesting results.

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. The goal in this experiment is also a 100% success rate, as less than this would indicate that there are models for which SimTH cannot generate the required test harness.

Recall that the total number of unique models contained in the experiment set is 457. For each of the models, there is an associated test suite which was provided along with the models. In order to test the correctness of SimTH, the provided test suites for the final release (release 15) were used to verify that the test harnesses generated by SimTH still passed the tests provided. There are a total of 54 models in the final release for which test suites were provided; these are the tests which were run in the testing tool with generated harnesses to verify correctness of SimTH.

7.2.2 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 industry partner 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. A graph showing all of the execution times for the 423 models can be seen in Figure 7.1.

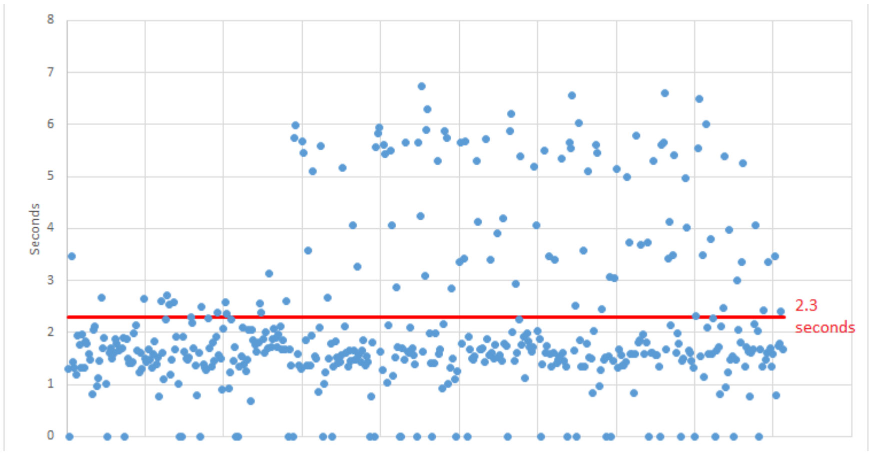


Figure 7.1: Time Required for Generation of Test Harnesses using SimTH

The second experiment, designed to measure the ability to produce the same testing results as the manually produced test harnesses was conducted on the 45 test suites corresponding to the 45 harnesses SimTH was able to generate for the final release. Of these 45 models, 39 successfully behaved the same (based on testing tool behaviour) as the test harnesses provided by our industrial partner, which is a success rate of 86.7%. These results are not as strong as initially desired, but are still very promising. SimTH automates a previously manual process, and produces correct models a high percentage of the time.

In the instances where SimTH's generated harness did not match the performance of the manual test harnesses, it appears that the reason for the inconsistency is something that is easily spotted based on manual inspection at the top level blocks of the harness. Our industry partner assured us this is common practice anyway, and a notice has been added to SimTH to perform this inspection of 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.

7.2.3 Additional Validation

Beyond the validation experiments conducted, our industry partners have also been conducting experiments on a much larger set of models, which due to confidentiality reasons are currently inaccessible. However, the running of their own tests has been beneficial in providing further results in terms of the success of SimTH.

While specifics cannot be discussed, our industry partners have communicated that SimTH is working for most of the models they have run it on. The reasons for failures on some of the models have been due to intricacies related to those specific models, and they can easily adjust the implementation to meet these additional requirements. This feedback from our collaborators is very reassuring and provides evidence that SimTH may soon begin to be used on a wider scale.

7.3 Summary

This chapter provides validation experiments for the two tools produced in this thesis, SimPact and SimTH.

SimPact, as a prediction tool, was validated by calculating the precision and recall for its ability to predict changes to input and output test values against the actual changes observed in the model and test set. Over the entire model set, SimPact had an F-measure of 0.87 for input and 0.84 for output. If instances of non-typical use are removed, these results improve to 0.95 and 0.93 respectively. Given a more typical use case, the performance of SimPact is promising.

SimTH, as a task automation tool, was validated for both computation time (as a means of consistency) and correctness. In terms of computation time, SimTH was able to generate test harnesses for the model set in an average of 2.3 seconds, which is a huge improvement over the manual creation of the harnesses. In addition to the speed up, the automation of the process increases the consistency in creation, both in terms of function, but also in visual standardization. With respect to correctness, SimTH was able to correctly generate 86.7% of test harnesses for the final release of the models, which is less than desired, but is explained by the older models used in the experiment; an issue already corrected by our industry partner. SimTH is being used by our partners with positive feedback and the issues faced in the experiments largely resolved due to internal changes to naming conventions.

Overall, the two tools produced in support of Simulink Model Management (SimPact and SimTH) have performed well in experimentation, and even better in practice in the industry setting. Combine the performance with the satisfaction of the industry requests, and it is clear these tools serve the desired purpose.

Chapter 8

Summary and Conclusions

8.1 Summary

This thesis makes the following contributions:

- An understanding of how models traditionally evolve in the application domain (Chapter 4: Co-Evolution of Simulink Models and Tests)
- A method for determining potential impact of changes in models on test cases (Chapter 5: Simulink Impact Analysis for Evolving Models)
- An implementation of impact analysis in a static and dynamic analysis tool, capable of identifying potential changes directly in test case files (Section 6.1: SimPact: Simulink Impact Analysis)
- A Test Harness Generator for Matlab Simulink Automotive models (Section 6.2: SimTH: Simulink Test Harness Generation)
- An evolution support tool which combines other tools and interfaces with existing industry tools (Chapter 6: Simulink Evolution Tool Support)
- Industrial verification/validation of the above using real-world automotive models provided by our industry partner (Chapter 7: Validation)

8.2 Future Work

There are several areas where the work presented in this thesis could be naturally extended to improve upon current results or to offer additional functionality of the tools developed. These are discussed for each of SimPact, SimTH, and SimEvo.

One of the shortcomings of SimPact is still that it provides *potential* impact, and *suggested* test values via simulation. One possible area of future work is to increase the confidence in its ability by specifically determining exact impact, and providing test case values based on something other than simulation. The exact implementation requires significant work, but is certainly a logical follow up to the current contributions of SimPact.

As discussed in the findings for SimTH, its performance was limited due to the models used for validation being an out of date set of models while SimTH was developed for current conventions. It would be beneficial to perform the validation experiment on a newer set of models using the updated naming conventions and organizational information, in order to confirm what is currently only known anecdotally from our industry partner.

Since SimEvo is meant to be a complete workbench for Simulink model management of evolving systems, there is certainly the opportunity to implement additional features and grow the workbench into an even more useful tool. Two such ideas are the integration with the version control system in order to not have to have both versions of the model on the workstation (rather compare against a checked-in version as the previous version), and the provision of an interactive review of potential test case changes - currently these are either marked in the test file or generated via simulation, however it may be useful to have the developer see the changes as they are suggested and either approve or reject them interactively. Beyond these two features, there are likely other tool support ideas that will emerge over time, both from the research perspective and from industry, and it would benefit SimEvo to adopt more functionality to support Simulink model management.

Each of the proposed future work ideas above are contingent on one key factor: the continued relation with our industry partner. The aim is to continue this partnership, which has served beneficial to all parties, into the future to continue to provide them support for Simulink model management, and for them to provide real world problems that require solutions.

8.3 Conclusions

The work conducted in this thesis falls into three main categories: the initial exploration of co-evolution and its findings, the impact analysis methods and implementations (SimTH) and the ability to generate test harnesses automatically (SimTH). Each of these areas contribute their own conclusions to the thesis as a whole, and support Simulink model management in their own way.

8.3.1 Co-Evolution Relationship

In the evolution study, the relationship between evolution of models and the evolution of their associated tests in a production model-driven automotive industrial system were examined in detail, with the goal of determining whether or not a co-evolution relationship exists. This was done by comparing the differences between versions of both the models and the tests, at every version, then creating a link between these comparisons. The results of this can be found in Table 4.1. Of note, it was shown that there is a strong positive correlation between models changed and tests as they evolve, $r(6) = 0.9, p < 0.01$, thus confirming that there is indeed a co-evolution relationship between the source models and their tests.

In addition to simply examining the existence of the relationship, 3 specific research questions were posed regarding the existence of a delay in updates, increases in efforts surrounding major events in a life-cycle and approaching final release, and the existence of changes to models that do not require changes to tests. Each of these questions was discussed in detail, with a definitive answer provided.

This initial understanding allowed research to progress in a meaningful direction by providing indication of importance of impact analysis. Additionally, the understanding of how models and tests evolve is useful in ongoing support of Simulink model management.

8.3.2 SimPact

SimPact is able to accurately trace impact of changes to any given model block throughout the rest of a model, traversing through the hierarchy, to the top level inputs and outputs, thus identifying potential impact on test case values. This was implemented in both a dynamic and static analysis tool. Static analysis works on batch changes between two versions of the same model, and dynamic analysis works directly in an open Simulink model, accessible from the block context menu.

SimPact was validated by calculating precision and recall, and ultimately an F-measure for its ability to predict changes in inputs and outputs of the test cases. The F-measures for input and output were calculated to be 0.87 and 0.84 respectively (and with the removal of non-standard use cases, these improve to 0.95 and 0.93

respectively). As a prediction tool SimPact is quite successful and provides promising results contributing to the goal of supporting Simulink model management.

8.3.3 SimTH

SimTH was requested by our industry partners due to high demand for such functionality, and is already in early stages of use in their testing process. Initial feedback is positive from our collaborators, and it seems that the tool is working correctly on a larger set of models than those used in the local experiment, providing a higher level of confidence in SimTH.

During validation experiments, it was concluded that SimTH was able to generate 423/457 test harnesses, with an average execution time of 2.3 seconds, and of the 45 tests run on the testing tool, 39 of them produced correct results. These are extremely promising first results for an industry inspired test automation tool, and this sentiment is echoed by our industrial partner. Given access to updated models conforming to the stronger naming conventions used by our partner, SimTH will likely demonstrate improved performance.

With one of the main goals of Simulink model management being the consistency between model artifacts, the ability to automatically generate test harnesses from the original library model removes one step in this process by removing the need to manually apply updates. SimTH greatly improves developers' ability to effectively manage Simulink models.

8.3.4 General Conclusions

This thesis presents model exploration and implementation of tools, all with the aim of supporting Simulink model management. The analyses presented significant evidence of the existence of a co-evolution relationship between Simulink models and tests in the given set, and helped identify types of change meriting further investigation. This additional investigation took the form of an impact analysis, capable of identifying the potential impact of model changes on the test values (and by extension the connected model files). This analysis was then implemented as an evolution support tool, SimEvo, capable of performing impact analysis (SimPact), as well as other evolution support tasks, such as test harness generation (SimTH), and interfacing with existing industry tools to integrate with their processes. These tool implementations were validated using real world models, and the results were presented along with discussion.

Bibliography

[1] Diffplug unleashes Simulink's potential. `https://www.diffplug.com/features/simulink`. Accessed: 2017-02-06.

[2] MathWorks Simulink product page. `http://www.mathworks.com/products/simulink/`. Accessed: 2015-10-27.

[3] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008.

[4] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Proceedings of the IEEE World Congress on Computational Intelligence, Evolutionary Computation, 2008 (CEC 2008)*, pages 162–168. IEEE, 2008.

[5] R.S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[6] P. Baker, Z.R. Dai, J. Grabowski, O. Haugen, I. Schieferdecker, and C. Williams. Model-based testing. In *Model-Driven Testing*, pages 7–13. Springer Berlin Heidelberg, 2008.

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

[8] M. Benjamin, D. Geist, A. Hartman, Y. Wolfsthal, G. Mas, and R. Smeets. A study in coverage-driven test generation. In *Proceedings of the 36th Design Automation Conference*, DAC '99, pages 970 –975, 1999.

[9] 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.

[10] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.

[11] L. Briand, J. Cui, and Y. Labiche. Towards automated support for deriving test data from UML statecharts. *Lecture Notes in Computer Science*, 2863:249–264, 2003.

[12] L. Briand and Y. Labiche. A UML-based approach to system testing. *Lecture Notes in Computer Science*, 2185:194–208, 2001.

[13] L.C Briand, Y. Labiche, and S. He. Automating regression test selection based on UML designs. *Information and Software Technology*, 51(1):16–30, January 2009.

[14] L.C. Briand, Y. Labiche, and L. O'Sullivan. Impact analysis and change management of uml models. In *Proceedings of the International Conference on Software Maintenance, 2003 (ICSM 2003)*, pages 256–265, Sept 2003.

[15] L.C. Briand, Y. Labiche, and G. Soccar. Automating impact analysis and regression test selection based on UML designs. In *Proceedings of the International Conference on Software Maintenance (ICSM 2002)*, ICSM '02, pages 252 – 261, 2002.

[16] E. Bringmann and A. Kramer. Model-based testing of automotive systems. In *Proceedings of the First International Conference on Software Testing, Verification, and Validation (ICST 2008)*, ICST '08, pages 485 –493, April 2008.

[17] N. Chapin, J.E. Hale, K.M. Khan, J.F. Ramil, and W. Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1):3–30, 2001.

[18] A. Cicchetti, F. Ciccozzi, and T. Leveque. A solution for concurrent versioning of metamodels and models. *The Journal of Object Technology*, 11(3):1:1, 2012.

[19] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. In *Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference*, EDOC '08, pages 222 –231, September 2008.

[20] 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.

[21] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. *Lecture Notes in Computer Science*, 670:268–284, 1993.

[22] I.K. El-Far and J.A. Whittaker. Model-based software testing. In *Encyclopedia of Software Engineering*. John Wiley & Sons, Inc., 2002.

[23] B. Ens, D. Rea, R. Shpaner, H. Hemmati, J.E. Young, and P. Irani. Chronotwigger: A visual analytics tool for understanding source and test co-evolution. In *Proceedings of the Second IEEE Working Conference on Software Visualization (VISSOFT 2014)*, pages 117–126. IEEE, 2014.

[24] Q. Farooq, M. Iqbal, Z.I. Malik, and M. Riebisch. A model-based regression testing approach for evolving software systems with flexible tool support. In *Proceedings of the 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS)*, pages 41–49, 2010.

[25] M.W. Godfrey and Q. Tu. Evolution in open source software: a case study. In *Proceedings of the International Conference on Software Maintenance (ICSM 2000)*, pages 131–142, 2000.

[26] H. Gomaa. *Software Modeling and Design*. Cambridge University Press, New York, NY, USA, 2011.

[27] J. Gray, Y. Lin, and J. Zhang. Automating change evolution in model-driven engineering. *Computer*, 39(2):51 – 58, February 2006.

[28] M.J. Harrold. Testing evolving software. *Journal of Systems and Software*, 47(23):173 – 181, 1999.

[29] A. Hartman. Model based test generation tools. *Agedis Consortium, URL: http://www. agedis. de/documents/ModelBasedTestGenerationTools_cs. pdf*, 2002.

[30] A.Z. Javed, P.A. Strooper, and G.N. Watson. Automated generation of test cases using model-driven architecture. In *Proceedings of the Second International Workshop on Automation of Software Test*, AST '07, pages 3–3, 2007.

[31] Y.G. Kim, H.S. Hong, D.H. Bae, and S.D. Cha. Test cases generation from UML state diagrams. *Proceedings, IEE Software*, 146(4):187 –192, August 1999.

[32] S. Kokaly, R. Salay, V. Cassano, T. Maibaum, and M. Chechik. A model management approach for assurance case reuse due to system evolution. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS '16, pages 196–206, New York, NY, USA, 2016. ACM.

[33] S. Kokaly, R. Salay, M. Sabetzadeh, M. Chechik, and T. Maibaum. Model management for regulatory compliance: A position paper. In *Proceedings of the 8th International Workshop on Modeling in Software Engineering*, MiSE '16, pages 74–80, New York, NY, USA, 2016. ACM.

[34] M.M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[35] H.K.N. Leung and L. White. Insights into regression testing [software testing]. In *Proccedings of the Conference on Software Maintenance, 1989*, pages 60–69, 1989.

[36] Z. Lubsen, A. Zaidman, and M. Pinzger. Using association rules to study the co-evolution of production & test code. In *Proceedings of the 6th IEEE International*

Working Conference on Mining Software Repositories, 2009. (MSR'09), pages 151–154. IEEE, 2009.

[37] F. Mantz, G. Taentzer, and Y. Lamo. Well-formed model co-evolution with customizable model migration. *Electronic Communications of the EASST*, 58:1–13, July 2013.

[38] P. Marinescu, P. Hosek, and C. Cadar. Covrig: A framework for the analysis of code, test, and coverage evolution in real software. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 93–104. ACM, 2014.

[39] C. Marsavina, D. Romano, and A. Zaidman. Studying fine-grained co-evolution patterns of production and test code. In *Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM 2014)*, pages 195–204. IEEE, 2014.

[40] Reza Matinnejad, Shiva Nejati, and Lionel Briand. Automated test suite generation for time-continuous Simulink models. Technical report, 2015.

[41] N. Medvidovic, D.S. Rosenblum, D.F. Redmiles, and J.E. Robbins. Modeling software architectures in the Unified Modeling Language. *ACM Transactions of Software Engineering Methodology*, 11(1):2–57, January 2002.

[42] T. Mens and S. Demeyer. *Software Evolution.* Springer Berlin Heidelberg, 2008.

[43] B. Meyers, M. Wimmer, A. Cicchetti, and J. Sprinkle. A generic in-place transformation-based approach to structured model co-evolution. *Electronic Communications of the EASST*, Volume 42, 2011.

[44] 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.

[45] M. Mirzaaghaei, F. Pastore, and M. Pezze. Supporting test suite evolution through test case adaptation. In *Proceedings of the Fifth International Conference on Software Testing, Verification and Validation (ICST 2012)*, ICST '12, pages 231 –240, Montreal, Canada, April 2012.

[46] 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.

[47] G. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., second edition, 2004.

[48] 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.

[49] R.F. Paige, N. Matragkas, and L.M. Rose. Evolving models in model-driven engineering: State-of-the-art and future challenges. *Journal of Systems and Software*, 2015.

[50] 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.

[51] W.E. Perry. *Effective Methods for Software Testing.* John Wiley & Sons, Inc., Indianapolis, IN, USA, 3rd edition, 2006.

[52] S. Person, M.B. Dwyer, S. Elbaum, and C.S. Pasareanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, page 226237, New York, NY, USA, 2008. ACM.

[53] 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.

[54] A. Pretschner, H. Lotzbeyer, and J. Philipps. Model based testing in evolutionary software development. In *Proccedings of the 12th International Workshop on Rapid System Prototyping*, RSP '01, pages 155 –160, 2001.

[55] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zlch, and T. Stauner. One evaluation of model-based testing and its automation. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2013)*, ICSE '05, page 392401, New York, NY, USA, 2005. ACM.

[56] E.J. Rapos. Understanding the effects of model evolution through incremental test case generation for UML-RT models. Masters thesis, Queen's University, Kingston, ON, September 2012.

[57] 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.

[58] E.J. Rapos and J. Dingel. Incremental test case generation for UML-RT models using symbolic execution. In *Proceedings of the Fifth International Conference on Software Testing, Verification and Validation (ICST 2012)*, ICST '12, pages 962 –963, Montreal, Canada, April 2012.

[59] 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.

[60] N. Rungta, S. Person, and J. Branchaud. A change impact analysis to characterize evolving program behaviors. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM 2012)*, pages 109–118, Sept 2012.

[61] R. Salay, M. Chechik, S. Easterbrook, Z. Diskin, P. McCormick, S. Nejati, M. Sabetzadeh, and P. Viriyakattiyaporn. An eclipse-based tool framework for software model management. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '07, pages 55–59, New York, NY, USA, 2007. ACM.

[62] A. Di Sandro, R. Salay, M. Famelis, S. Kokaly, and M. Chechik. Mmint: A graphical tool for interactive model management. In *Proceedings of the ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015), Demo and Poster Session*, pages 16–19, 2015.

[63] M. Stephan and J.R. Cordy. A survey of methods and applications of model comparison. Technical Report 2011-582, School of Computing, Queen's University, Kingston, Ontario, Canada, 2011.

[64] L.H. Tahat, B. Vaysburg, B. Korel, and A.J. Bader. Requirement-based automated black-box test generation. In *Proceedings of the 25th Annual International Computer Software and Applications Conference*, COMPSAC '01, pages 489 – 495, 2001.

[65] J. Tretmans and E. Brinksma. TorX: automated model based testing. In *Proceedings of the First European Conference on Model-Driven Software Engineering*, pages 31–43, December 2003.

[66] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Proceedings of the International Conference on Software Testing, Verification and Reliability (ICST 2012)*, 22(5):297–312, 2012.

[67] K. Vorobyov and P. Krishnan. Combining static analysis and constraint solving for automatic test case generation. In *Proceedings of the Fifth International Conference on Software Testing, Verification and Validation (ICST 2012)*, ICST '12, pages 915 –920, Montreal, Canada, April 2012.

[68] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen. Mining software repositories to study co-evolution of production & test code. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST 2008)*, pages 220–229. IEEE, 2008.

[69] 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.

[70] P. Zech, M. Felderer, P. Kalb, and R. Breu. A generic platform for model-based regression testing. *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, 7609:112–126, 2012.