Co-Evolution of Model-Based Tests for Industrial Automotive Software

by

Eric J. Rapos

Supervisory Committee:

James R. Cordy (Supervisor)

Juergen Dingel

Mohammad Zulkernine

A Research Proposal submitted to the School of Computing

in conformity with the requirements for the degree of Doctor of Philosophy

Queen's University

Kingston, Ontario, Canada

May 2014

Abstract

This document presents a research proposal for my doctoral studies. The proposed work is in the approved research area "model-based testing", with a focus on the approved topic "co-evolution of model-based tests". We begin with an overview of the work to be conducted, including a recap of the research area, our motivation, and conclude the introduction with the thesis statement and details of the scope of the work. The main portion of this document is an in-depth literature review, providing background information on the different aspects of the proposed work. Finally, we present the proposed research, including methodology, stopping criterion, validation metrics, possible limitations and risks, proposed contributions, milestones and progress to date.

Contents

SECTION 1:INTRODUCTION

Model-based testing (MBT) is a rapidly expanding field involving the use of software modeling to aid in the creation, automatic generation, execution, and maintenance of software tests. In MBT, executable tests are generated from a model of the system under test (SUT); the SUT must conform to the model, which describes the SUT's desired behaviour, as well as other necessary requirements of the SUT. Test models are intended to be a clear, concise, and abstract representation of the SUT, easily understandable by developers and testers unfamiliar with the application domain. The resulting tests may be executed *online* (directly on the SUT, dynamically), or *offline* (generated for later execution); offline tests may be run automatically or manually [49]. Tests are executed on the SUT in order to verify that it behaves as desired, and conforms to the test model; this is often done by having the test environment automatically compare the results against specified expected outputs, which are included in the test model. MBT is an area with room for improvement; resulting test suites may be of unmanageable size, and maintaining test models over time can be a complex task.

The concept of co-evolution refers to two (or more) objects evolving alongside each other, such that there is a relationship between the two that must be maintained. In the field of co-evolution of model-based tests, this refers to the the test models evolving alongside the source models, such that the test models remain correct for testing the source models.

1.1 Motivation

With model-driven engineering (MDE) becoming more prevalent in software development, a heavier focus on model-based testing (MBT) is needed. Previous work

centered largely on the iterative development aspect of MBT [42, 43], however further attention is needed on the prolonged maintenance of model-based tests after initial release. The iterative regeneration of tests for each change is no longer a viable solution to this issue. Evolution of model-based tests is an area that has been explored in related work [15, 25, 41], however there are few examples of how these approaches apply in an industrial setting.

1.2 Thesis Statement and Scope of Work

Thesis Statement: Model-based test efficiency can be improved by co-evolving test models alongside system models. This can be done through studying software model evolution patterns and their effects on test models in order to apply updates directly to the tests.

The scope of the work is addressed by the methodology presented in the proposed research section. Each subsection details its own individual stopping criterion, and together they form the basis for the scope of this work. We aim to provide an example implementation for our technique for a particular domain (Simulink Automotive Models), however we believe that the concept is applicable across domains. Working with automotive models allows us to look at one particular application of real-time software, and its implementation in safety-critical systems.

SECTION 2:BACKGROUND AND RELATED WORK

To understand how test models evolve alongside system models, it is first necessary to understand several core concepts. In addition, there are areas of interest where these core concepts overlap, which will be referred to as combined concepts. Finally, the overlap of these combined concepts forms the basis for the proposed research topic.

In this section, we provide an explanation, through related work, of the three core concepts for this work: software testing, software modeling, and software evolution. Next, we present work on three combinations of the core concepts, also by way of related work. The combined concepts are model-based testing, test evolution and model evolution. Finally, we present our findings from the limited literature available on the topic of evolution of model-based tests to place our research goals in the context of the state of the art. Figure 2.1 provides a Venn diagram of how all of these concepts combine to form the research topic.

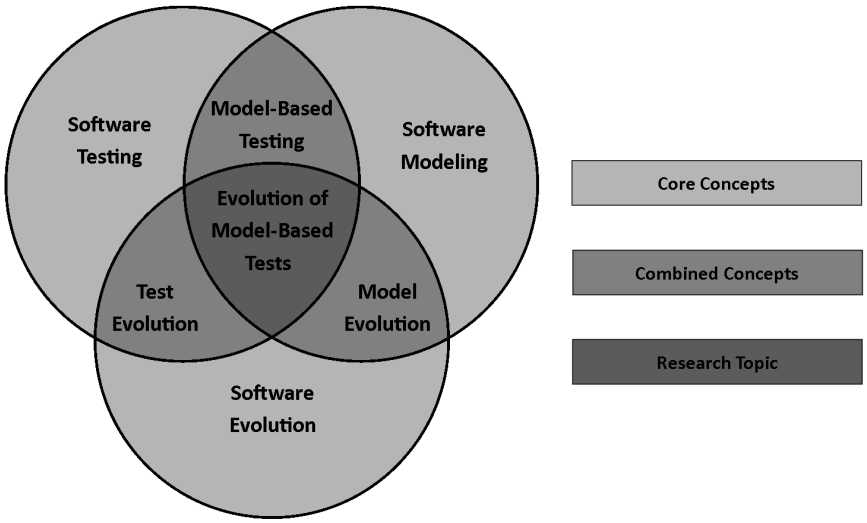

Figure 2.1: Venn diagram of topics

2.1 Core Concepts

As with any research project, a basic understanding of a number of higher level concepts is required to properly frame the work within the field. The proposed work is deeply rooted in three main areas, each of which will be explored at a high level in this section. First, we present the meaning and importance of software testing, as well as common challenges. Next, an introduction to software modeling techniques provides an adequate modeling background for the remainder of this literature review. And finally, we discuss what is meant by software evolution, and some of the current problems in this area.

The three subsections in this section are not meant to be a comprehensive look at the covered topics, but rather a short introduction to the concepts. They also provide the set of relevant related work for this proposal. Section 2.2, in which the core concepts are combined, provides a more in-depth look into how the topics are combined.

2.1.1 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 [36]. 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." [36].

In 1979 Glenford J. Myers presented a comprehensive look at the Art of Software Testing, a text which has since been updated [36], 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 [1] 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 [38] 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

The text goes on to explain a number of other testing practices, which will not be presented here for brevity's sake, however, it is certain that the practices identified are thorough and present a full understanding of the topics contained within them.

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 [47]. 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 [1]), 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 [50].

2.1.2 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 proposal, 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." [7] will be 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 [7]. The reference manual provides a more than

thorough background on software modeling, and the use of UML, so we have chosen not to provide too much detail in this proposal; 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. [32].

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 [24] 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 proposal is the concept of model comparison. The ability to compare two (or more) models, and determine their differences, is extremely useful for our work. A recent comprehensive survey was completed by Stephan et al. [46] which outlines a number of different techniques and technologies for comparing models, as well as an evaluation of these techniques.

2.1.3 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, we draw attention to a text on the subject by Mens and Demeyer [33]. 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 proposal.

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. [13] 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, we present the case study completed by Godfrey and Tu, regarding evolution on open source software [23], specifically the Linix kernel. 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 [39], 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.

2.2 Combining the Concepts

Based on the three core concepts presented in the previous section, we now present three areas of overlap between the core concepts, which we refer to as combined concepts, as each combination of two core concepts is an area of research on its own. We begin by presenting a thorough review of model-based testing, including techniques, applications, and current research problems. Next, we briefly present the concept of test evolution, and how it plays an important role in the continued maintenance of software. Finally, we present another combined concept known as model evolution, dealing with the continued maintenance of models over time, and how this can effect software.

2.2.1 Model-Based Testing

Similar to MDE, where models are the primary testing artifact, Model-Based Testing (MBT) refers to the concept of testing using models. This section of the literature review is the most comprehensive, and in depth, as the field is one that is quite saturated with work, and it is also the most relevant background work for the proposed research.

El-Far and Whittaker [20] 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 SUT
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. [2] 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 within 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 within 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. [11]. 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. [29]. 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. [4] 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. Another approach is presented by Bertolino et al. [5] in which they describe their four step process for MBT. In the following four steps SEQ refers to a sequence diagram, and ST refers to a state diagram.

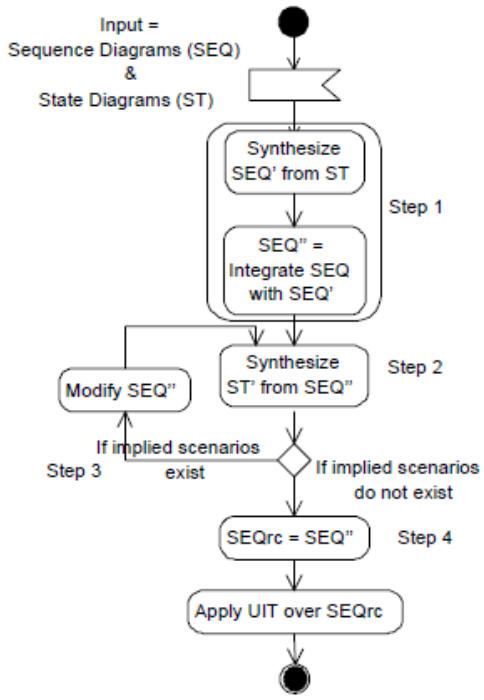

Figure 2.2: Bertolino et al.'s four step approach to MBT [5]

1. Producing a complete scenario specification SEQ'' using SEQ and ST (inputs) in which the inputs are integrated and combined into a sequence diagram.
2. Synthesizing State Diagrams ST' from SEQ'' in which the resulting sequence diagram from step 1 is used to synthesize a modified state diagram.
3. Implied Scenarios detection and resolution process in which ambiguities and implied scenarios (incorrectly identified sequences due to a mismatch between local and global variables) are removed from the model.
4. Generation of the SEQrc model in which the final, reasonably complete (rc) sequence diagram is generated for testing.

Another approach to MBT is the use of the Abstract State Machine Language (AsmL) presented by Barnett et al. [3]. AsmL was developed based on the abstract state machine paradigm, and is fully integrated with the .NET framework. The goal is to use AsmL to semi-automatically generate test sequences at an early stage of development, while taking advantage of the ease of adaptability provided by model-based testing. Their tool uses the following technologies: Parameter generation, FSM generation, sequence generation, and runtime verification.

Bohr presents another interesting take on MBT, by extending it to statistical testing, to create Model-Based Statistical Testing (MBST) [6]. This paper provides an extension/enhancement to the technique of model based statistical testing, previously

presented in other works. The extension allows the treatment of systems with concurrency and realtime constraints, two features characteristic for embedded systems. He provides background on MBST including the concept of a usage model (along with a graphical representation of these models), and how to derive test cases from these usage models. Any path on a usage model that begins at the **source**, and ends at the **sink**, is considered to be a possible test case. From there, the author notes that MBST does not deal with time explicitly, working on a step-like semantics, which is not the best option when dealing with strict real-time embedded systems. To handle both concurrency and time, the authors use Discrete Deterministic and Stochastic Petri Nets (DDSPNs). The purpose of the DDSPNs in this work is the automated model based software testing; generating tests automatically from the DDSPN. They address some of the issues that arise in the process, particularly the issue of two stimuli/transitions, which are related to the same input channel, being enabled at the same time. The solution to this is the use of colored transitions, in which each stimulus/transition related to a specific input channel is given at least one color, and no two stimuli/transitions having a color in common may be enabled at the same time. This introduction is followed by an in depth example of their technique. The author then discusses specifically the notion of test cases, in that they are a sequence of states starting from the initial state of the usage net (UN). Moreover they present specific details of the tool implementation, which allows the graphical creation of a UN, and the generation of the usage model from the UN, and the generation of abstract test cases from the usage models. The tool also allows for the generation, execution, and evaluation of concrete test cases.

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 [19]. Next came work on the automation of deriving tests from UML statecharts, work presented by Briand et al. [10]. This was closely followed by work by Tretmans et al. [48] 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 [41]. 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. [28] 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 proposed domain of automotive software. Bringmann and Kramer present exactly this; a paper on MBT in automotive systems [12]. This work is of particular interest as this is a domain on which we plan to focus, and they make use of MATLAB/Simulink models in their work, which is also of particular interest to us. 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% within 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 within, 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).

The area of model-based testing is an area of interest within the School of Computing; two surveys on the topic have recently been presented, each with a different application or specialization. Zurowska and Dingel present a survey of MBT for reactive systems [53] and Saifan and Dingel present a survey of MBT for distributed systems [44].

Outside of the School of Computing, another comprehensive survey of approaches MBT was published by Dias Neto [18]. The review is of 406 papers found searching five digital libraries (IEEEXplorer, ACM Portal, INSPEC, Compendex IE, and Web of Science), as well as websites and conference proceedings. The authors used a comprehensive search string in an effort to find a complete set of papers to review. Based on their findings, they were able to categorize each of the works into one of the five following categories:

- (A) Model representing information from software requirements (functional testing) and is described using UML diagrams.
- (B) Model representing information from software requirements and is described using any non-UML notation.
- (C) Model representing information from software internal structure (architecture, components, interfaces, units; structural testing) and is described using UML diagrams.
- (D) Model representing information from software internal structure and is described using any non-UML notation.
- (E) Papers collected during the search, but unrelated to MBT and therefore excluded.

A total of 204 papers found were determined to not be relevant, and of the remaining 202 papers, only 78 had been examined at the time of publication; all results which follow are based on these 78 papers. First, the authors looked the level at

which testing was done, and divided the works into the four categories of System, Integration, Unit/Component, and Regression, with system testing being by far the most popular, and regression testing making up only 5% of the papers studied. The second area examined by the authors was the level of automation, which was assessed by the complexity of the manual involvement, and rated Low, Medium or High, with High meaning a high level of manual involvement in the process. The next area they examined, in brief, was the approaches which had tool support, finding that 64% of the approaches have tools to support their execution. Finally, they looked at the types of models used for test case generation for each of the approaches. This is where the division of UML and non-UML came in to play, in that they looked at the models used and categorized them this way. The UML based models were definitely more commonly used, with Statechart diagrams being used in 27 of the approaches, and Class and Sequence Diagrams each being used in 19 of the approaches. Following the above quantitative analysis, a qualitative analysis was conducted based on the following criteria: Testing Coverage Criteria, Behavioural Model Limitations, and Cost and Complexity of Applying MBT approach. The following are the conclusions reached from their survey:

- MBT approaches are usually not integrated with the software development process
- The MBT approach usually cannot represent and test NFRs
- Requirements to use a MBT approach include knowledge about the modeling language, testing coverage criteria, generated output format, supporting tools make the usage difficult/unfeasible; these need to be minimized
- Most MBT approaches are not evaluated empirically and/or not transferred to the industrial environment

2.2.2 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 [26], 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 [30]. 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 [35]. 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. [51] 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. [17], 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.

It is important to note that many of the problems presented by Chapin et al. [13] regarding software evolution directly apply to test evolution as well.

2.2.3 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 [15]. 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 [14]. 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 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 [34]. 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 2.3. Their approach contains two steps at the highest level, the creation of the difference model, and the migration of instance models.

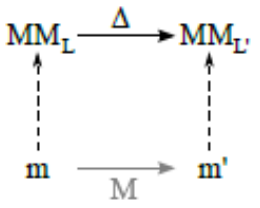

Figure 2.3: Migration of model instances based on meta-model evolution [34]

Yet another take on this is presented by Gray et al. [25], 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. [31] 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 [46] 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. [13] also apply to model evolution.

2.3 Research Topic

Having presented the three core concepts, and the combined concepts, we now present the research topic, which is made up of all three of the core concepts, using methods and implementations from the core concepts. There is a limited amount of existing literature on the research topic, however this section aims at presenting the relevant work, and providing a clear understanding of the problem domain, alternative approaches, and a sense of where the proposed work will fit.

2.3.1 Evolution of Model-Based Tests

We begin with my own MSc work, which was focused on the same topic of evolution of model-based tests, with a different methodology throughout the implementation. Initial work focused on incremental testing of UML-RT models, using symbolic execution [42] while the final thesis focused more on the understanding of model evolution through incremental testing [43]. While both of these works 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. We used existing techniques for symbolic execution and test case generation 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, we were able 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, we presented the results of an initial evaluation of our 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. [52], 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. [21], 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 [40], 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.

In another approach to testing evolving systems, Fourneret et al. present their work on selective test generation for evolving critical systems [22]. The goal of their work is to determine the impact of model evolution on existing test suites, given two versions of a model. Impact analysis is used in many of the presented works on evolution (software, test, and model) with promising results. Beyond the impact analysis, their work also looks at classifying tests into four categories based on their applicability to the new model version:

i. **evolution** - new tests designed to test newly evolved features
ii. **regression** - existing tests to test unchanged features to ensure they still work as designed
iii. **stagnation** - invalid tests that no longer apply as the features they intended to test were removed; these tests are run to ensure they are indeed invalid
iv. **deletion** - these are the stagnation tests from the previous version, and are no longer needed, and can be deleted

Finally, two papers by Briand et al. address the automation of regression test selection using UML designs: their original publication [9] and a follow up which expands upon the original work [8]. 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. [8] 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.

Based on the works 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 will be useful throughout the proposed work.

2.3.2 Comparison with the State-of-the-Art

A number of the works presented in this background section are closely related to the work we are proposing, however they are different enough, and serve as an added indication of the interest in the topic.

The work by Zech et al. [52] for the MoVE frameork, while similar to the proposed research, deals specifically with regression testing and the selection of the test set, however we aim 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. [21] also focuses on

the selection of regression tests for future testing; our work will incorporate this step but also expand to include the adaptation of existing tests.

With regards to the testing work done in Simulink [12], this work looks specifically at the testing of systems, while our work 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 our proposed work, and confirms much of our motivation to work with automotive software.

While similar to these presented works, our work differs in that we plan to focus on more than regression test selection, or how to test Simulink models. These will be a part of our expected results, however we also aim to indicate which parts of existing test suites can be adapted to test the evolved, and how to adapt the tests. Further details of our work can be found in the next section.

SECTION 3:RESEARCH PLAN

The overall goal of the proposed work is to identify methods that will allow model-based tests to remain current and correct throughout a system's lifetime, while also requiring minimal resources such as run time, computational power, and person-hours. As part of the research, we plan to examine existing industrial model and test versions to attempt to identify and compile a catalog of common evolutionary changes. While this may restrict the types of evolutions to a defined set, the ultimate goal will be to accept any arbitrary change, and handle the co-evolution of the tests effectively. Ideally MBT co-evolution will be well-integrated with other MDE tasks such as model development, code generation, and testing, and be implemented in one single integrated development environment (IDE).

3.1 Methodology

The overall goal of this section is to provide a methodology for each of the stages of the research project. For each phase there will be an overview of the goals, methods, and a stopping criteria. The work will be conducted using Matlab Simulink as our modeling language. This is primarily due to the increasing number of automotive companies (General Motors included) that are using this technology in their development. Additionally Simulink provides several added benefits for automotive development which make it an interesting technology to study; the versatility of the Similunk environment provides real-time simulation capabilities which allow for more reliable and accurate testing early in the development process.

There are three phases to the proposed research: (i) Evolution Study, (ii) Algorithm Design, and (iii) Prototype Tool Development. Each phase is meant to be self contained, with results filtering into the next, creating a linear time-line for the

research.

3.1.1 Evolution Study

The first phase of the project consists of an extensive look at a number of consecutive versions of existing models and their corresponding test models (actual models and tests provided by our industrial partners) to determine how evolution occurs within their domain. We will be extending prior work [42, 43], now making use of MATALB Simulink [27] Models, however we require a new set of evolutionary steps specific to this domain, which also cover a wider range of operations.

As part of the evolution study, the first goal will be to determine a concrete set of models that will be used for the remainder of the work. Criteria will include models with multiple versions, and test models included, and they should preferably be industrial models with real world applications.

There are a number of open source models available, along with multiple versions of each model, that will make the basis for initial exploration. Beyond these public models, as part of the research I plan to take part in an internship at General Motors, during which time I plan to expand the exploration to their previous versions of production models. Even without access to the current versions, a set of previous versions should be sufficient to determine the standard evolution steps that commonly occur throughout routine maintenance and updates of their Simulink Models.

Based on the catalog of changes obtained from this examination, the next part of the study will be to analyze the impacts of these model evolution steps on tests to determine how they co-evolve with the models. Again, the models and tests to be used for this phase will be in two parts: the public open source models, and any

models that we will be able to access during an internship at General Motors.

It is first important to understand what exactly tests for Simulink Models are. To test a Simulink Model is to execute (or simulate) the model in such a way that the desired behaviour is observable. This is done by creating test models, which have the purpose of sending the correct signals at the correct times. Additionally, the test models act as an oracle, providing expected outputs which can be compared to the the observed output. Further details as to the actual testing practices will be obtained through integration into the General Motors testing environment.

The methodology for comparing versions of models and tests will be to make use of Simulink's built-in model model comparison, and both the MATLAB and Simulink Report Generators. The tool generates an in-depth interactive report that details changes between two models (in our case two versions of the same model), or even folders of models, based on a comparison of the XML representations of the models. Using this comparison interface, as well as an already implemented MATLAB script which makes use of the same differencing algorithm, we should be able to accurately determine differences between versions in order to create the catalog of differences. Figure 3.1 shows an example comparison using the built-in graphical tool, showing the differences listed in a hierarchical view, and Figure 3.2 shows the textual output of differences generated via the script.

It is believed that these observed differences will fall into three broad categories: additions, modifications, and deletions (similar to those presented by Cicchetti et al. [14]). From there, it is likely that each of these differences will apply to the native elements of Simulink models, such as subsystems, connections, signals, attributes, etc., thus resulting in combinations of the two (added subsystem, deleted subsystem,

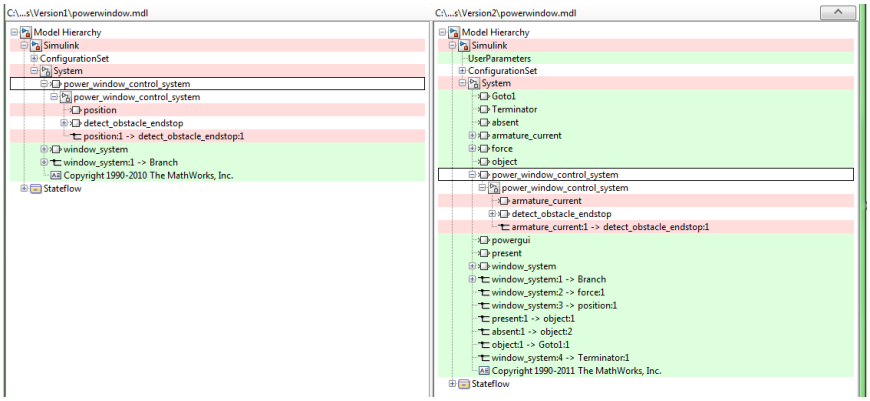

Figure 3.1: Graphical Comparison Tool

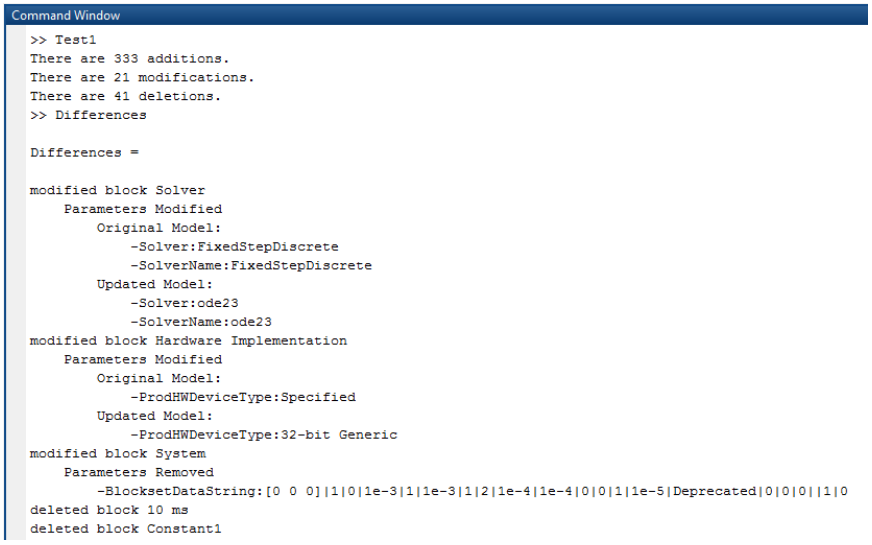

Figure 3.2: Output From Script

modified attribute, deleted connection, etc.). While this will initially lead to finite difference steps, the goal of this step will be to define and observe evolution patterns, which may be groups of these steps that always (or commonly) occur together. These patterns will prove to be more useful than the finite steps in determining effects on tests.

Given the findings observed in this phase, the aim is that the effects of evolution

on test models will be evident, and we will be able to utilize the results in the later stages of the project.

Stopping Criteria: The first portion of this phase will be complete once we have selected the set of models and tests that will be used for the remainder of the work; this set will meet the criteria specified above. The second portion of this phase will be complete when we produce a catalog matching common evolutionary steps to a direct effect on a test model. Complete in this context refers to having a catalog entry for each of the three actions for each of the MATLAB artifacts for all models in our chosen set.

3.1.2 Algorithm Design

Using the information obtained from the Evolution Study, the goal of this phase is to develop a set of algorithms for implementing the co-evolution of model-based tests.

When given an existing model (M), its current test model (TM), and a number of changes applied to the model (Δ) (thus resulting in a new version of the model (M')), we will determine what changes (Δ') need to be made to the test model to ensure that the updated test model (TM') is a correct test for the newly updated model (M'), and how to apply them in the most effective way. Figure 3.3 shows the overview of this process.

The aim is to develop a set of algorithms that take as input M, M' and TM, and are able to apply all necessary updates to TM to generate TM'. Recall the three categories of differences that we propose to be working with: additions, modifications, and deletions. Given an added model component, it is hypothesized that we will likely be required to add functionality to the test model, while a deletion will require the

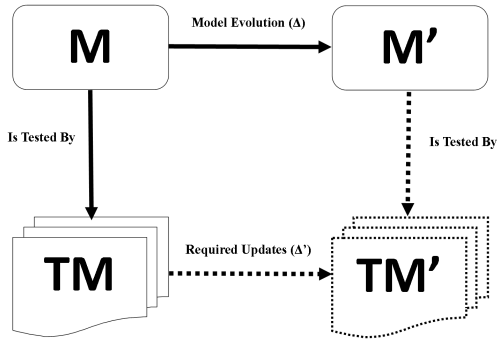

Figure 3.3: Overview of Algorithmic Implementation

removal of functionality. This phase will solidify the rules for these types of updates, and formalize them in a set of algorithms. Furthermore, based on the portions of the test that remain unchanged, we will identify which tests do not need to be rerun during later runs, thus reducing the amount of testing required.

Stopping Criteria: This phase will be complete when we have developed a set of algorithms capable of producing the necessary Δ', to be applied to a test model such that the new test model version meets testing requirements for the new model version (M'). The set of models for this step will be the set chosen in the first stage of research.

3.1.3 Prototype Tool Development

Using the algorithm developed in the previous phase, the third phase will be the development of a prototype tool which automates the co-evolution of model-based testing.

Continuing with the Simulink use-case, the goal will be to implement our co-evolution testing framework within Simulink, writen in MATLAB code, much like

our lab's tool SimNav [16]. The minimum requirement will be an interface that allows users to select two consecutive versions of a model (or set of models), along with the first version's test models; the prototype tool will then perform the updates to the tests. However, it is possible that we can make use of a Simulink version control system to manage the selection of model and test versions, such that the user need only be concerned with updating the models, and the updating of tests is done automatically in the background. Included in the prototype will also be a report to the user of the changes made to the model (reported in easy to understand terms), as well as any necessary updates to the tests, as manual interaction may be required in some cases.

In addition to the co-evolution work from this project, the goal is to create a complete testing workbench within Simulink that is capable of automatically generating test model stubs for a given system, determining differences between versions (model differencing is a well-known application [46]), and automatically co-evolving test models alongside the models (the research contribution of this project).

Stopping Criteria: This phase will be complete when we have produced of a prototype tool that is capable of interfacing with the Simulink Development environment that implements our developed algorithm for co-evolution of Simulink Model Tests. Additional model testing features may be included in the end result.

3.2 Validation

The final phase of the work will be a validation phase in which we test not only the correctness of the implementation, but the performance and usability as well. Validation will be conducted on the chosen set of models used throughout the project.

Correctness

Validating the correctness of our approach will be the easiest result to obtain, as we will simply be comparing the test models generated by our tool against the existing test models. We will be looking for functional equality between the tests, to ensure that our generated tests test *exactly* the same behaviours as the originals.

The minimum accepted standard for correctness will be that the prototype tool produces correct results on all models used. Any less will result in revisiting the implementation, as without a correct implementation, performance results will not be relevant.

Performance

Performance of our prototype tool will be measured on the criteria of time. Time performance will make use of two measurements to identify the amount of time required for the generation of the updated test models. First, we will look at the computation time required by the tool to generate the updated test model, and second we will look at the amount of person-time required to make use of the model and begin executing test models. These times will be compared to baseline results of manually updating existing models and executing the updated versions. It seems very evident that our automated approach will be a significant improvement on the manual updates, however we aim to provide numerical evidence of this claim, along with exact results of how much more efficient our approach is. The end result will be a percentage of improvement over the manual generation.

Usability

With regard to usability, we hope to provide evidence that our interface and methodology will be preferred by test engineers. To show these results, we plan to conduct usability studies, using developers from our industrial partner as subjects, in order to determine which methodology they prefer for the continued updating of test suites for their production models. In addition to simply finding preference, we would also like to elicit feedback on our implementation in order to possibly improve the interface.

3.3 Limitations and Risks

No work is without risk, and this project is no exception. The first possible risk in this proposed work is the availability of a substantial set of models and test models. While we intend on working with models obtained from General Motors, and those available while on internship, these may not be made fully available to us when needed during the research. The open source models will serve as a complete set, however a substantial amount of models will be useful to provide additional results.

Another potential risk of the proposed work is slightly related, and this is the fact that we intend to conduct usability studies of the prototype with actual test engineers. This will require the cooperation of the engineers, and their availability, which is something that cannot be planned for. Given this it is entirely possible that the results will be presented based solely on the correctness and performance of the tool. While this is not the preferred outcome, it is recognized as a potential issue, and will be an accepted end result if necessary.

Finally, there is a risk associated with the aim of a “complete” catalog of model evolutions. Whenever “complete” is used it can be misinterpreted, and it is likely that

we will only be able to include a "reasonably complete" set of evolutions within this catalog. In this case, "reasonably complete" will include all evolution steps within the complete set of open source models used throughout the research, such that our tool performs correctly on all of these models.

3.4 Contributions

Our work will make the following contributions to the fields of model-based testing, and automotive software development:

- provide a methodology for co-evolution of model-based tests
- produce a catalog of evolution patterns for model development in Simulink
- automatically identify areas of test models affected by evolution of models
- increase the efficiency of test evolution process in MBT
- provide a test evolution workbench for industrial automotive model development

These contributions will be made possible largely through involvement in the NSERC CREATE Graduate Specialization in Ultra-Large Scale Software Systems [45] and the NECSIS Research Network [37]. We will be able to work closely with our industrial partners to examine real-world practices; this information will all be used to develop a workbench to aid in the continued maintenance of industrial automotive models. The workbench will be integrated into the existing Simulink model development environment in order to take advantage of the existing tools and practices. Ideally, not only will the findings be utilized in academic development, but by our industrial partners in production systems to improve efficiency in the testing of their systems developed using MDE.

3.5 Milestones and Progress

Thesis progress will be based on the the following milestones:

1. **Model and Test Set Chosen:** The set of models and test models for the remainder of the work is chosen, based on initial examination of evolution patterns, availability of models, and a comprehensive understanding of Simulink testing obtained via GM internship. *Target Date:* October 2014
2. **Evolution Catalog Complete:** Using the chose model and test set, the evolution catalog, which maps changes in models to changes in test suites, is complete. *Target Date:* March 2015
3. **Algorithm Design Complete:** Based on the results of the Evolution Catalog, a set of algorithms is developed to implement the identification of changes, and the adaptation of existing tests. *Target Date:* July 2015
4. **Tool Prototype Complete:** Using the designed algorithms, a tool capable of performing the tasks of difference identification, test updates, and reports to the user will be developed. *Target Date:* December 2015
5. **Validation Experiments Complete:** Using the designed tool prototype, validation of correctness, performance, and usability will be conducted. *Target Date:* March 2016
6. **Finish Writing Dissertation:** Based on all prior work, a dissertation presenting the work will be written. *Target Date:* August 2016
7. **Oral Defense of Dissertation:** Complete the final step of the PhD program and defend the dissertation. *Target Date:* November 2016

Initial exploration has been completed. Examination of the publicly available Automotive models, as well as those obtained from GM, has provided insight into the types of models we will be working with, all of which contributes to Milestone 1. Initial experimentation has been conducted with methods for differencing versions of Simulink models, and a Matlab script has been developed which determines if a change is an addition, modification, or deletion, and identifies the basic Simulink type of the object; this work will contribute to Milestone 3.

BIBLIOGRAPHY

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008.

[2] Paul Baker, Zhen Ru Dai, Jens Grabowski, Oystein Haugen, Ina Schieferdecker, and Clay Williams. Model-based testing. In *Model-Driven Testing*, pages 7–13. Springer Berlin Heidelberg, 2008.

[3] Mike Barnett, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Towards a tool environment for model-based testing with AsmL. In Alexandre Petrenko and Andreas Ulrich, editors, *Formal Approaches to Software Testing*, volume 2931, pages 252–266, Montreal, Canada, 2004. Springer Berlin / Heidelberg.

[4] 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.

[5] A. Bertolino, E. Marchetti, and H. Muccini. Introducing a reasonably complete and coherent approach for model-based testing. *Electronic Notes in Theoretical Computer Science*, 116:85–97, January 2005.

[6] F. Bohr. Model based statistical testing of embedded systems. In *Proceedings of the Fourth International Conference on Software Testing, Verification and Validation (Workshops)*, ICST '12, pages 18 –25, Montreal, Canada, March 2011.

[7] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.

[8] 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.

[9] 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 '02, pages 252 – 261, 2002.

[10] Lionel Briand, Jim Cui, and Yvan Labiche. Towards automated support for deriving test data from UML statecharts. *Lecture Notes in Computer Science*, 2863:249–264, 2003.

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

[12] 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 '08, pages 485 –493, April 2008.

[13] Ned Chapin, Joanne E. Hale, Khaled Md. Khan, Juan F. Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1):3–30, 2001.

[14] 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.

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

[16] James Cordy. Submodel pattern extraction for simulink models. In *Proceedings of the 17th International Software Product Line Conference*, pages 7–10, Tokyo, Japan, August 2013.

[17] Brett Daniel, Tihomir Gvero, and Darko 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.

[18] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A survey on model-based testing approaches: a systematic review. In *Proceedings of the First ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering*, WEASELTech '07/ASE '07, pages 31–36, New York, NY, USA, 2007. ACM.

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

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

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

[22] E. Fourneret, F. Bouquet, F. Dadeau, and S. Debricon. Selective test generation method for evolving critical systems. In *Proceedings of the Fourth International Conference on Software Testing, Verification and Validation (Workshops)*, ICST '11, pages 125 –134, March 2011.

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

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

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

[26] Mary Jean Harrold. Testing evolving software. *Journal of Systems and Software*, 47(23):173 – 181, 1999.

[27] MathWorks Inc. Simulink - sumulation and model based design. `http://www.mathworks.com/products/simulink/`. Last visited on 5/5/2014.

[28] 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.

[29] 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.

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

[31] Florian Mantz, Gabriele Taentzer, and Yngve Lamo. Well-formed model coevolution with customizable model migration. *Electronic Communications of the EASST*, 58:1–13, July 2013.

[32] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling software architectures in the Unified Modeling Language. *ACM Trans. Softw. Eng. Methodol.*, 11(1):2–57, January 2002.

[33] Tom Mens and Serge Demeyer. *Software Evolution*. Springer Berlin Heidelberg, 2008.

[34] Bart Meyers, Manuel Wimmer, Antonio Cicchetti, and Jonathan Sprinkle. A generic in-place transformation-based approach to structured model co-evolution. *ECEASST*, Volume 42, 2011.

[35] 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 '12, pages 231 –240, Montreal, Canada, April 2012.

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

[37] NECSIS. NECSIS (network for the engineering of complex software-intensive systems for automotive systems). `http://www.necsis.ca/`. Last visited on 5/5/2014.

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

[39] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina 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.

[40] 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.

[41] 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 '05, page 392401, New York, NY, USA, 2005. ACM.

[42] 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 '12, pages 962 –963, Montreal, Canada, April 2012.

[43] Eric James 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.

[44] Ahmad Saifan and Juergen Dingel. Model-based testing of distributed systems. Technical Report 2008-548, School of Computing, Queen's University, Kingston, Ontario, Canada, September 2008.

[45] Queen's University School of Computing. NSERC CREATE graduate specialization in ultra-large scale software systems. `http://ulss.cs.queensu.ca/`. Last visited on 5/5/2014.

[46] Matthew Stephan and James 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.

[47] 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.

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

[49] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, July 2010.

[50] 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 '12, pages 915 –920, Montreal, Canada, April 2012.

[51] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge 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.

[52] Philipp Zech, Michael Felderer, Philipp Kalb, and Ruth 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.

[53] Karolina Zurowska and Juergen Dingel. Model-based generation of test cases for reactive systems. Technical Report 2010-573, School of Computing, Queen's University, Kingston, Ontario, Canada, July 2010.