CHANGE ANALYSIS ACROSS VERSION HISTORIES OF SYSTEMS MODELS

by

SAHEED POPOOLA

JEFF GRAY, COMMITTEE CHAIR
JEFFREY CARVER
CHRIS CRAWFORD
RICHARD PAIGE
RANDY SMITH

A DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2021

ABSTRACT

Model-Based Systems Engineering (MBSE) elevates models as first-class artifacts throughout the development process of a system's lifecycle. This makes it easier to develop standard tools for automated analysis and overall management of a system process; thereby, saving cost and minimizing errors. Like all systems artifacts, models are subject to continuous change and the execution of changes may significantly affect model maintenance. Existing work has already investigated processes and techniques to support, analyze and mitigate the impact of changes to models. However, most of these works often focus on the analysis of changes between two sets of models and do not take a holistic approach to the entire version history of models. To support change analysis across the entire version history, we developed a Change Analyzer that can be used to query and extract change information across successive versions of a model. We then used the Change Analyzer to mine several versions of Simulink models, computed the differences across the versions, and classified the computed differences into appropriate maintenance categories in order to generate information related to understanding the rationale of the design decisions that necessitated the observed changes. To study the impact of changes on the models, we used the Change Analyzer to analyze the evolution of seven bad smells in 81 LabVIEW models across 10 open-source repositories, and four bad smells in 575 Simulink models across 31 open-source repositories. The evaluation of the Change Analyzer indicates that it can be used to construct concise queries that execute faster than a generic model-based query engine. The results of the change analysis process also show a high similarity of the recovered design decisions with the manually identified decisions, even though the manual

identification process takes much more time and often does not provide additional information about the changes executed to implement the design decisions. Furthermore, we discovered that adaptive maintenance tasks often lead to an increase in the number of smells in systems models, but corrective maintenance tasks often correlate with a decrease in the number of smells.

DEDICATION

To my mother and to my wife, both of whom have always stood by me.

LIST OF ABBREVIATIONS

EMF	Eclipse Modelling Framework
EOL	Epsilon Object Language
LabVIEW	Laboratory Virtual Instrument Engineering Workbench
LabVIEW NXG	LabVIEW Next Generation
LOC	Lines of Code
MDE	Model-Driven Engineering
MBSE	Model-Based Systems Engineering
OOP	Object-Oriented Programming
VCS	Version Control Systems
VI	Virtual Instruments
WEKA	Waikato Environment for Knowledge Analysis

ACKNOWLEDGEMENTS

First, I would like to thank my wife, Barakat, for her patience and support throughout my PhD years. Without her support and motivation, none of these would have happened.

I thank my advisor, Dr. Jeff Gray for the wonderful guidance, support, and feedback that I received throughout the years. It has really been a remarkable experience with him, and he has certainly made me a better person and a better academician. He has also provided me with an opportunity to explore outreach activities that has made it easy for me to contribute to society.

I would also like to thank my committee members Dr. Jeffrey Carver, Dr. Randy Smith, Dr. Crawford, and Dr. Richard Paige for their valuable feedback and agreeing to be on my committee. I really appreciate the guidance that you have given me over the years.

I am grateful to Dr. Dimitris Kolovos who introduced me to research in software engineering. Without his initial guidance, I may not have been able accomplish this success. I am also grateful to Dr. Antonio García -Dominguez and Dr. Taylor Riché for their help with the Hawk and LabVIEW tools, respectively.

Finally, I would like to thank my colleagues in the Computer Science department, especially the software engineering group, for the wonderful time we spent together. We always shared ideas, and they have provided useful feedback to most of my work throughout my PhD journey.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

CHAPTER 1

INTRODUCTION

Systems Engineering is a discipline that deals with the creation, execution and overall management of processes related to the development of a system in a way that ensures timely and cost-effective satisfaction of the system requirements [Friedenthat 2007, Haskins 2006]. Technically, Systems Engineering is the synthesis and development of an entire system by defining abstractions to capture the system requirements and support the efficient management of the relationships among these abstractions, external design constraints, components, and other user-defined measures [Denmo 2008].

Traditionally, Systems Engineering uses a document-based approach that involves several manual tasks. Firstly, the system requirements are saved in documents. Then, a design of the system is manually produced via drawing tools such as PowerPoint or Visio. These design drawings have little intelligence about the component elements such as requirements, behaviour or structure of the system even though the drawings are meant to capture these elements. The analysis and optimization is also manually performed on the drawings [Bajaj 2012].

The traditional document-centric approach has many limitations. Firstly, there is an obvious disconnect across the different phases of the system's life cycle. This makes it difficult to modify the system based on any change in the requirements [Bajaj 2012]. Furthermore, the analysis and optimization phase often requires high-level expertise and a steep-learning curve [Bajaj 2012]. This increases cost and sometimes there are differences in understanding between

the person who created the design drawings and the analyst who is working on the analysis and optimization phase; thereby introducing additional errors. The lack of traceability also makes it challenging to propagate changes across different phases of a system's life cycle [Yadav 2016].

Model-Based Systems Engineering (MBSE) aims to address the limitations in the traditional approach to Systems Engineering via the use of formalized models [Friedenthat 2007]. MBSE is the use of models as first-class artifacts throughout the development lifecycle of systems. The models are formal abstractions that capture the system requirements and conform to a set of standard notations. These models make it easy to develop standard tools that can automate the analysis and optimization phase via transformations of the models to different formats; thereby saving time, cost and minimizing errors [Gau 2012, Yadav 2016].

MBSE provides a platform for managing the development activities from the conceptual phase such as requirements planning and analysis, up to the final implementation and testing phase [Bajaj 2011]. The International Council on Systems Engineering (INCOSE) has long recognized the importance of MBSE with a dedicated working group towards ensuring that future approaches to systems engineering is model-centric and there is a robust framework of tools, languages and expertise to address emerging challenges [Friedenthat 2007].

There has been a growing interest in MBSE in industry and academia. For example, NASA utilizes MBSE technique for many of their research projects [Gough 2018] while the US Navy also developed an MBSE system to facilitate collaboration among engineers in diverse locations [Crisp 2002, Jenkins 2021]. Aerospace companies such as Boeing and Airbus also use MBSE approaches for many of their development scenarios [Gau 2012, Malone 2016]. Many

systems engineering programs now offer specialized courses in MBSE[1] and a number of industries have been established that primarily offer specialised MBSE services[2].

MBSE represents a paradigm shift from the traditional document-based method of implementing system engineering designs and ensures that models play a central role in the requirements analysis, design, integration, testing, and maintenance activities of a system [Estevan 2007]. This implies that changes are continuously executed on the models as the system evolves.

1.1 Modelling Environments

A number of tools have been developed to manipulate models and support the overall management of modelling activities in MBSE. A key feature of MBSE projects is the interaction between hardware and software components of a system; hence, most MBSE-based tools often provide support for effective integration between hardware and software. Common examples of such tools include Papyrus[3], MagicDraw[4], Simulink[5], and LabVIEW [Johnson 1997]. The research presented in this dissertation focuses on models developed via LabVIEW and Simulink modelling environments.

The Laboratory Virtual Instrument Engineering Workbench (LabVIEW) by National Instruments is an extensible systems modeling platform that is currently used by hundreds of thousands of users in more than 17,000 companies all over the world (Services n.d.; Falcon 2017). LabVIEW provides a graphical programming environment for developing test

[1] https://sysengonline.mit.edu/
[2] http://intercax.com/
[3] http://www.eclipse.org/papyrus/
[4] https://www.nomagic.com/products/magicdraw
[5] https://www.mathworks.com/products/simulink.html

instruments and software systems that require fast access to hardware data. However, despite the extensive industrial and academic usage of this platform in developing complex and sometimes critical systems, LabVIEW systems models rarely gain attention in software engineering research. Furthermore, the platform is mostly used by traditional engineers (e.g., systems engineers or mechanical engineers) who may not be familiar with basic software engineering concepts; thereby increasing the possibility of bad software designs and high maintenance effort. A previous study has shown that LabVIEW developers often tend to prioritize correctness over other properties that may affect the maintainability of the software over a period of time [Chambers 2013].

Simulink [Simulink] is a popular graphical tool for designing, modelling, and simulating dynamic systems. Simulink is widely used in both industry and academia for developing and analysing complex dynamic systems in multiple domains (e.g., avionics and automotive), with over four million users[6]. Similar to LabVIEW, Simulink is also mostly used by traditional engineers. However, there has been considerable software engineering research that targets Simulink-based systems [Gerlitz 2015, Hooman 2004, Rapos 2017]. Unfortunately, most of these research efforts do not provide sufficient analysis of changes in Simulink-based systems.

1.2 Software Repositories

Software repositories such as GitHub provide capabilities to track changes and maintenance activities executed across the lifecycle of a system. This provides a rich set of data for analysing changes executed on a system, monitoring the evolution of the system, and understanding the various maintenance activities that are executed to ensure that the system continues to perform

[6] https://www.mathworks.com/content/dam/mathworks/handout/2020-company-factsheet-8-5x11-8282v20.pdf

optimally. There exists a number of research works that have targeted the extraction and analysis of data from software repositories [Chatzigeorgiou 2014, Palomba 2013, Tahmid 2016]. However, the core research often targets repositories of software developed with text-based programming languages, with little focus on graphical or visual languages used in a design or implementation.

1.3 Change Analysis

System artifacts like models are subject to continuous change due to addition of new requirements, quality control, or correcting errors and anomalies. Unfortunately, these changes may sometimes introduce new errors, or the cost of implementing such change outweighs the potential benefits. Therefore, it is not surprising that a number of techniques and approaches have been developed to detect, extract, and manage changes to modelling artifacts [Briand 2003, Paige 2016, Toulme 2006]. However, the majority of these approaches often focus on two sets of artifacts at a time and do not take a holistic view of the changes across the entire version history of the models. This holistic view is important to understand the impact of these changes on the structure of the models, and also extract the patterns of changes that affect the optimal functioning of the system. Furthermore, there has been little research to understand the reasons behind these changes. The lack of understanding the reason behind a change can lead to a number of challenges such as a lack of understanding of adopted technical solutions, repeating past mistakes, use of incompatible technologies, and increase in maintenance costs [Borrego 2019, Van Heesch 2009].

1.3.1 Change Impact and Evolution of Bad Smells in Models

Bad smells are design defects in software that tend to affect the optimal functioning and maintenance of a system [Fowler 2018]. Bad smells are not necessarily bugs because they do not affect the correct functioning of the system; however, they are design flaws that may negatively affect the quality of the software and generally increase the cost of maintaining the system.

Bad smells may be introduced at the initial version of a system, and propagated through subsequent versions, while new smells may also be introduced at any version of the system [Olbrich 2009]. Therefore, as a system ages and new changes are continuously executed on the system, the instances of bad smells embedded in the system continue to evolve and may lead to larger and more complex architectural problems. Therefore, the evolution of bad smells represents a way to capture the impact of changes across the system lifecycle.

A number of research efforts have sought to understand the impact of changes to models. However, the majority of these works have focused on how changes may necessitate other changes or introduce inconsistencies [Briand 2003, Lin 2015, Wang 2018]. There has been limited research to understand the impact of changes to the evolution of bad smells in models even though this research is important for determining the set of best practices for building models with minimal smells and the set of anti-patterns that should be avoided due to the likelihood of introducing new smells to models.

1.3.2 Understanding the Reason behind a Change

Software repositories provide capabilities to capture and track changes across the lifecycle of a system. Existing research provides support for extracting changes across two versions of models that have been captured in a software repository. However, the reason behind a change is not often documented and this situation may lead to *knowledge vaporization* and its associated

problems, such as the use of incompatible technologies, violation of design issues, and overall increase in maintenance cost [Borrego 2019, Capilla 2007].

Design decisions are the sets of solutions that have been proposed to address a specific requirement. These sets of solutions often include the considered alternatives, tradeoffs, and rationale behind the chosen solution. Hence, design decisions help us to understand the reason why changes are executed on models. Unfortunately, the design decisions are rarely documented, but there has been a number of studies to recover the design decisions that necessitate the evolution of systems and software artifacts [Jansen 2008, Shahbazian 2018, Shahin 2014]. However, most of these studies target systems developed with text-based programming languages rather than models.

Change classification techniques can provide useful information related to the rationale behind a change [Mockus 2000]. For example, knowing that a change is associated with the addition of a new feature helps us to understand why the change was implemented. Unfortunately, many of the change classification techniques often focus only on identifying bug-fixing code and do not consider other types of change. Furthermore, these studies often deals with text-based programming languages instead of models.

1.4 Research Goal and Objectives

A similar paradigm to MBSE is Model-Driven Engineering (MDE) [Schmidt 2006, Brambilla 2012]. MDE is a software engineering paradigm that uses models as first-class artifacts in the software development process rather than the traditional code-centric approach. In order to bridge the gap between the user's requirements and the final software product, MDE uses models to describe complex systems at multiple levels of abstraction and provides support for automated transformation and analysis of models [Völter 2013]. This is similar to the approach used by

MBSE for managing a system's life cycle. *The goal of the research described in this dissertation is to facilitate the adaptation of techniques from MDE to address existing software engineering challenges in MBSE.*

To address the software engineering challenges related to change analysis, bad smells, and understanding the rationale behind a change within the context of MBSE, the research presented in this dissertation provides an approach to analyse changes across the entire version history of system models. Furthermore, the research demonstrates how the change analysis approach can be used to better understand the evolution of bad smells, classify changes into appropriate categories, and extract the design decisions behind the changes executed on system models as the system evolves. Figure 1.1 provides an integrated view of the research presented in this dissertation.

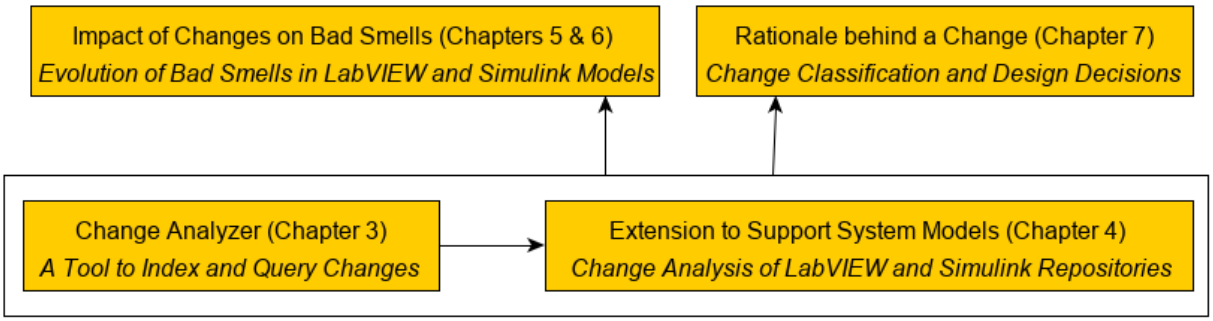

Figure 1.1. Research Overview

The objectives of this dissertation include the following.

1. To develop an approach and tool for analyzing the changes across the entire version history of models.
2. To extend the approach to support LabVIEW and Simulink models, thereby bringing software engineering research closer to systems modelling.

3. To use the approach to analyse the relationship between changes to LabVIEW and Simulink models, and the evolution of bad smells in these models
4. To use the approach to classify changes into their appropriate categories in order to better understand the rationale behind the design decisions that necessitate the evolution history of systems models.

The change analysis approaches described in this dissertation are focused on LabVIEW and Simulink graphical models. However, these approaches should be generalizable to other graphical and textual models.

1.5 Dissertation Roadmap

This chapter has introduced the overall theme of this dissertation and we have highlighted the current gaps in research related to change analysis, bad smells, change classification and change rationale within the context of MBSE. Chapter 2 will discuss relevant concepts and tools in MBSE while Chapter 3 discusses how we implemented a Change Analyzer for querying change information across the version history of models. Chapter 4 deals with how we extend our Change Analyzer to support LabVIEW and Simulink models, including the development of an open-source metamodel for LabVIEW. Chapters 5 to 7 discuss how we used the Change Analyzer to support the analysis of bad smells, change classification, and design decisions of LabVIEW and Simulink in GitHub repositories. Chapter 5 presents the evolution of bad smells in LabVIEW models while Chapter 6 examines the evolution of bad smells and maintenance activities in Simulink models. Chapter 7 considers how we classify changes associated with design decisions across the version history of Simulink models. Chapter 8 concludes the work presented in this dissertation and discusses the future research plans.

CHAPTER 2

BACKGROUND

This chapter aims to present the basic information about models within the context of MBSE. The chapter also discusses the synergy between MDE and MBSE, as well as the relevant modelling standards and tools that have been adopted in this dissertation. Finally, the chapter provides an overview of the model comparison process for extracting changes between two or more sets of models.

2.1 Models and Metamodels

A model is an abstraction for a phenomena of interest [Ludewig 2003]. Models help to simplify the complexity of a system by enabling the developer to focus on the interesting parts while hiding away other phenomena that are not related to the object of focus. Models have also been used for documenting the design of software and communicating technical aspects of the software development process to non-technical users [Brambilla 2017]. A typical model is composed of model elements, attributes and references. An element is often a component of the model that captures a specific feature of the system that is represented by the model. Attributes represent unique properties of the model or a model element, while references capture the relationships between the model elements.

In MBSE, models are created via a modelling language that is defined by a metamodel. The structure of models are defined by a set of well-defined rules and semantics that are embedded in their metamodel. A metamodel is a model that specifies the types of elements that

exist in a language for a specific domain, the relationships that should exist among these elements, and any constraints that should be enforced in the relationships. A model is said to be valid when the type of each of its elements are specified in the metamodel, and the relationship among the elements corresponds to the relationships that are allowed among the appropriate types in the metamodel. Thus, a metamodel is similar to a grammar for a programming language.

A conformance relationship exists between a metamodel and its model. A model is said to conform to a metamodel, or it is a valid instance of such a metamodel, when each of the model element's type is defined in the metamodel and the relationship among the elements corresponds with the relationship specified among the corresponding types in the metamodel.

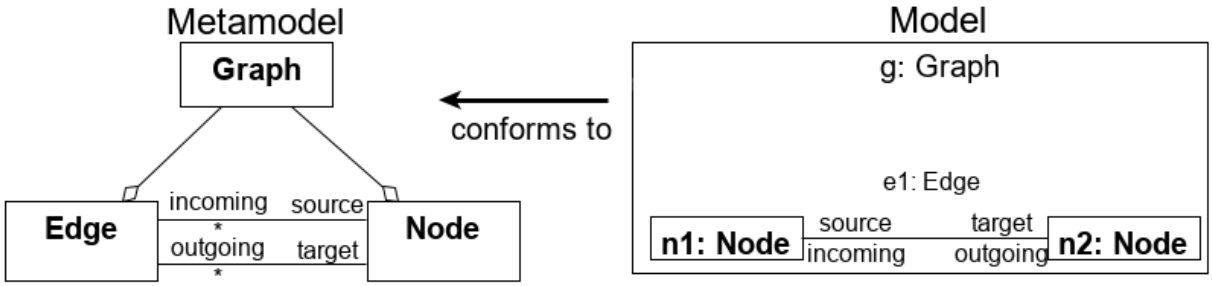

Figure 2.1. A Graph Metamodel and its Conforming Model

Figure 2.1 gives an example of a metamodel that defines a simple graph structure and a graph model that conforms to the metamodel. The metamodel describes a graph that is composed of nodes and edges. Each node can be connected to multiple edges, but each edge is connected to exactly two nodes - a source node and a target node. The source node regards the edge that connects to it as an incoming edge, while the target node treats its connecting edge as an outgoing edge. The Graph g, which contains two Nodes named n1 and n2, connected by an Edge named el, conforms to these requirements.

2.2 Synergy between MDE and MBSE

MDE is a software engineering paradigm that uses models as principal artifacts of the software development process. This allows the developers to focus on the design and abstract important concepts of the problem space into these models without worrying about the underlying computing environment or programming language [Brambilla 2012, Selic 2003]. In order to realize the benefits of MDE, model management programs and tools are needed to manage the inherent complexity of models. Model management programs are frameworks that help to construct, execute and manipulate models in order to generate software artifacts. Model management also preserves the validity of a model by ensuring that the models conform to their respective metamodels. There exist several model management frameworks that help to manage the lifecycle of models, such as Epsilon [Kolovos 2008], ATL [Jouault 2008], VIATRA [Csertán 2002], GEMOC Studio [GEMOC 2021] and Hawk [Barmpis 2013].

MBSE is a systems engineering paradigm that elevates models to first-class artifacts throughout a systems' lifecycle. MBSE uses formal models to support requirements specification, system design and analysis, testing and maintenance activities [Friedenthal 2007]. This makes it easy to have a common artifact that connects all the different phases of a system's lifecycle, thereby enhancing communication among team members, reducing cost, increasing productivity, enhancing transfer of knowledge, and making it much easier to manage the overall complexity of systems [Denmo 2008][Friedenthal 2007].

Three main types of models are often used in MBSE: 1) structural models to capture the system's architecture, 2) behavioural models to represent the dynamic behavior of a system and its response to external events, and 3) performance models for analysing the system's performance, a key objective of systems engineering [Friedenthal 2007] [Yadav et al 2016].

Furthermore, most models in MBSE are required to support multiple viewpoints and capture the relationships among all the viewpoints [Denmo 2008].

Many tools have also been developed to support the overall management of modelling activities in MBSE. The majority of MBSE projects involve seamless communication between hardware and software components of a system; therefore, most MBSE-based tools often provide support for effective integration between hardware and software. Common examples of such tools include Simulink and LabVIEW.

Several similarities exist between MDE and MBSE. Firstly, both paradigms aim to use models as principal artifacts in the development of a system. Both paradigms also require tools and techniques for managing and manipulating models. In fact, some tools are interoperable for both paradigms and there is a growing effort to support models created by MBSE tools in MDE tools, and vice-versa [Xue 2015]. Furthermore, tools in both paradigms offer increasing support for automated analysis.

However, some subtle differences also exist between MDE and MBSE. Firstly, MDE targets software engineers/developers while MBSE is a paradigm that primarily targets system/traditional engineers. Most MBSE tools offer efficient integration of the software components with hardware components while most MDE tools often focus on the software parts. Many of the widely used MBSE tools are proprietary and they often manipulate models stored in custom formats. Table 2.1 summarizes the comparison between MBSE and MDE.

MBSE	MDE
A systems engineering paradigm	A software engineering paradigm
A paradigm with tools that do not consider some software engineering concepts	A paradigm with more tools that implement standard software engineering practices
Models as first-class artifacts	Models as first-class artifacts
Has many proprietary tools to manipulate models	Has many open-source tools to manipulate models
Mostly used by traditional (non-software) engineers	Mostly adopted by software engineers
Often requires software and hardware integration	Mostly focused on software components

Table 2.1. MBSE and MDE Comparison

2.3 Modelling Technologies

Modelling technologies are important for the construction, execution, and querying of models. Modelling technologies provide mechanisms for the overall management of models and the execution of model-specific tasks, such as automated transformation of a model to another format, translating models into executable artifacts, and validating if a model satisfies a set of user-defined constraints. The following subsections discuss the five modeling technologies that have been used in this dissertation to support change analysis.

2.3.1 Eclipse Modelling Framework

The Eclipse Modelling Framework (EMF) [Steinberg 2008] is a standard modelling infrastructure that is supported by diverse modelling tools, thereby facilitating the heterogeneous integration of these tools. EMF is a modelling extension to the Eclipse IDE, and provides a set of plugins for building structured models and appropriate Java classes to support command-based

execution of the models. EMF is composed of three main parts: an Ecore metamodel that can be used to describe other models and metamodels, EMF.Edit for building editors for the models, and EMF.Codegen for appropriate code generation activities.

2.3.2 LabVIEW Graphical Environment

LabVIEW is a systems modelling tool for managing model-based systems engineering processes. The tool provides a graphical language named G that can be used to model and develop applications that require fast access to hardware and test data [Johnson 1997, Kodosky 2020]. The tool also provides support for many third-party hardware and software vendors, as well as the ability to extend the tool to develop custom user interfaces and commands.

LabVIEW models are composed of Virtual Instruments (VIs) that are often stored in separate files similar to how classes are handled in Object-Oriented Programming (OOP). A VI can run on its own or it can be grouped together with other VIs to provide a common functionality. A VI is composed of two main components: a front-panel that captures the front-end or user interface of the application, and a block-diagram that stores the main program logic of the application. A third component named icon is used to store external images used in the model.

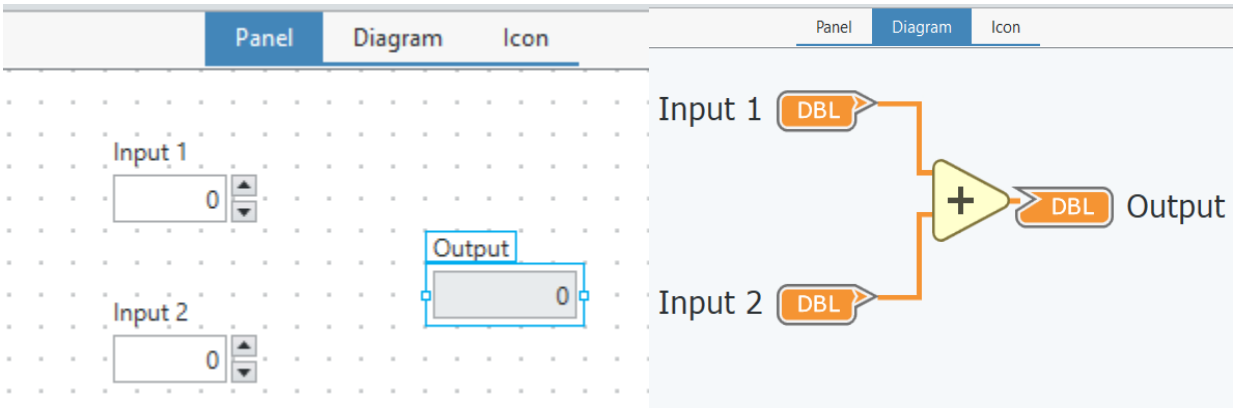

Figure 2.2. A Front Panel and Block Diagram to Add Two Numbers

Figure 2.2 shows the front panel and block-diagram of a VI that adds two inputs and returns a result as output to the user. LabVIEW models were originally stored in binary format, which makes it hard to analyse using automated tools. However, the recent LabVIEW Next Generation (LabVIEW NXG) version stores models in XML, making it more amenable to read and analyse by external tools. The LABVIEW-related research presented in this dissertation focuses on LabVIEW programs developed in the new NXG version.

```
1 < BlockDiagram Id="12">
2      < DataAccessor Id="16" Label ="19">
3         <Terminal DataType =" Double " Direction = " Output " />
4      </ DataAccessor >
5      <NodeLabel AttachedTo ="16" Id="19">
6        <p. Text >Input 1</p. Text >
7      </ NodeLabel >
8      < DataAccessor Id="21" Label ="24" >
9         <Terminal DataType =" Double " Direction = " Output "/>
10     </ DataAccessor >
11     <NodeLabel AttachedTo ="21" Id="24">
12        <p. Text >Input 2</p. Text >
13     </ NodeLabel >
14     < DataAccessor Id="26" Label ="29" >
15        <Terminal DataType =" Double " Direction = " Input "/>
16     </ DataAccessor >
17     <NodeLabel AttachedTo ="26" Id="29">
18       <p. Text >Output </p. Text >
19     </ NodeLabel >
20     <Add Id="30" Terminals ="o=33 , c0t0v =31 , c1t0v =32 " />
21     <Wire Id="31" Joints ="N (16 :Value )| N (30 :  c0t0v )" />
22     <Wire Id="32" Joints ="N (21 :Value )| N (30 : c1t0v )" />
23     <Wire Id="33" Joints ="N (30 :o)|N (26 :Value )" />
24 </ BlockDiagram >
```

Listing 2.1. Simplified Block Diagram Part of a VI file

Listing 2.1 shows how the VI in Figure 2.2 is persisted as files. Lines 2 to 7 contain information about “input 1” terminal; lines 2 to 4 are used to capture the input value while lines 5 to 7 are used to capture the name of the terminal (i.e., input 1). Lines 8 to 13 contain information about “input 2” terminal, while lines 14 to 19 represent the output terminal. Line 20 represents the

addition operation, while lines 21, 22, and 23 represent the different wires that links input 1, input 2, and output terminals with the “addition” node.

2.3.3 Simulink Tool for Model-Based Design

Simulink [Simulink] by Mathworks[7] is an extensible, graphical programming environment that supports the analysis, modeling and simulation of dynamic systems. Simulink is part of the MATLAB tool[8] and it provides support for diverse model management tasks such as automated code generation, testing and model verification. Simulink also provides a graphical editor and extensible libraries that can be used to design models of systems and run different simulations on the models.

A typical model in Simulink is composed of three main types of elements. These elements are *blocks* that capture different subsystems, *lines* for connecting the blocks, and *ports* for input/output of data from the blocks. Figure 2.3 shows a Simulink model chosen from one of the sample models in the Simulink package.

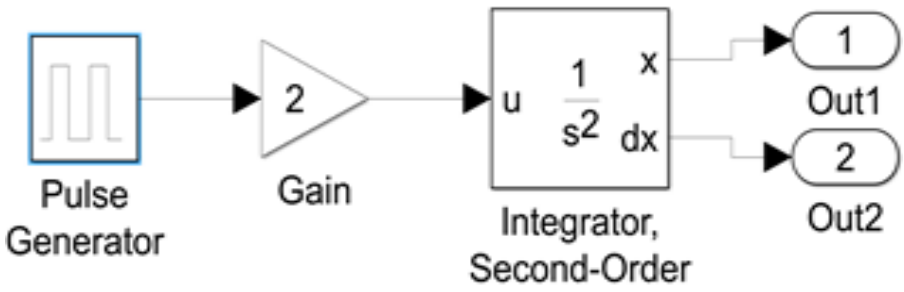

Figure 2.3. A Simulink Model of a Car Acceleration System[9]

[7] https://www.mathworks.com/
[8] https://www.mathworks.com/products/matlab.html
[9] https://www.mathworks.com/help/simulink/gs/create-a-simple-model.html

Figure 2.3 is a graphical representation of a Simulink model that captures the motion of a car after the speed pedal is pressed and the resulting position of the car relative to its starting point. The model contains five blocks connected by lines. The *pulse generator* generates the input signal to indicate the pressing of the speed pedal, while the *gain* block multiplies the input signal by a defined factor to indicate the resultant effect of pressing the speed pedal on the car's acceleration. The *second-order integrator* performs the integration of the input signal twice to calculate the relative position of the car based on the acceleration. The two *output* blocks designate the position of the car relative to its starting point.

2.3.4 Massif Tool for Transforming Simulink to EMF

Massif [Hovart 2015] integrates Eclipse-based EMF tools and the MATLAB/Simulink framework via Java commands in the MATLAB API. The tool supports bi-directional transformation from Simulink models to EMF-based models, as well as user-based configurations to guide the transformation. This transformation is achieved in Massif by first connecting to the MATLAB Engine, then sending a series of commands to the MATLAB API to either get useful information about the Simulink model in order to construct an appropriate EMF representation of the model, or to construct a new Simulink model based on the properties of an EMF model. The EMF models that are supported by the Massif tool must conform to an Ecore-based metamodel that captures most of the essential elements in a Simulink model. Figure 2.4 is the graphical representation of Massif's EMF representation of the Simulink model in Figure 2.3.

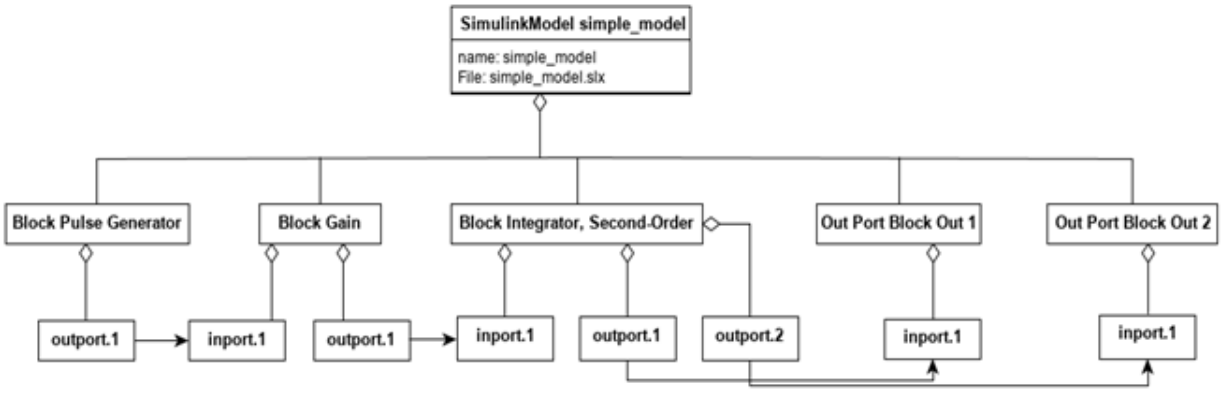

Figure 2.4. Massif's EMF Representation of a Simulink Model

2.3.5 Hawk Tool for Indexing Models in Repositories

Hawk [Barmpis 2013] is a scalable model indexing framework that can be used to query models that are distributed over a large number of files, such as those that are stored in file-based repositories (e.g., Version Control Systems - VCS). A VCS such as Git[10] can track changes that are made to files, support a large number of users, and handle conflicts in the changes made to files by different users. However, a VCS does not provide support for managing the relationships between the model elements stored in the files, leaving that complexity to the user [Bartelt 2008]. Although a number of model-centric repositories have been developed, they have not been widely adopted by practitioners: some lack features such as branching and tagging, while others re-implement similar functionality (e.g., user management, locking, checking out a model), or only work seamlessly with their own modelling tool. These repositories tend to lack the widespread tool support (e.g., continuous integration systems) of traditional file-based VCS.

[10] https://git-scm.com/

Hawk provides a model-centric layer over file-based repositories, thereby combining the model-centric querying and navigation capabilities with the maturity and widespread use of file-based repositories. This model-centric layer is implemented via a common interface for querying models where various parts of the model are stored in different files. This interface provided by Hawk allows the users to manage the model-specific complexities not handled by a traditional VCS while still providing full access to the typical VCS capabilities. Hawk can answer arbitrary queries in a dialect of the Epsilon Object Language (EOL) [Paige 2009], report changes across files in the repository, or present views of the indexed models as read-only EMF models.

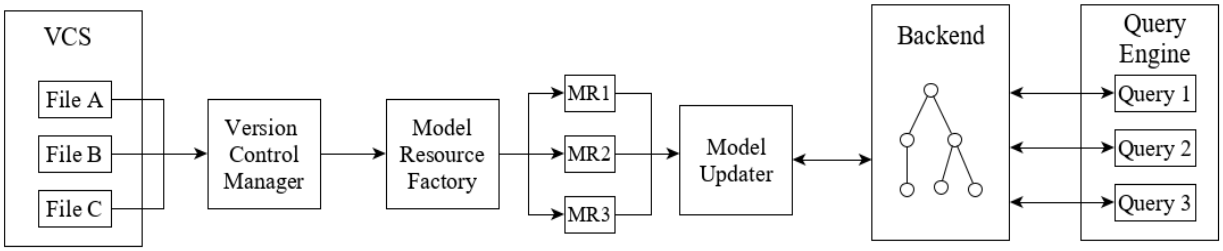

Figure 2.5. Overview of the Hawk Framework [Barmpis 2020]

Figure 2.5 provides a graphical overview of Hawk. The Hawk framework follows a component-based architecture that can be extended in various ways. A *VCS manager* component in Hawk retrieves file-based changes in a VCS repository, and a *model resource factory* component parses the files into an in-memory model abstraction inspired by EMF (but independent from it). A *model updater* component compares the in-memory models against the current state of the *backend* component (currently, one of several graph databases), and performs an incremental update on the backend to reflect the latest state of the models. For large models fragmented across files, Hawk will index each fragment separately, and then

reconnect the fragments, to save memory. After the backend is updated, it is ready to answer questions via *query engine* components.

In the original versions of Hawk, and in its current default configuration, Hawk does not track the history of the changes, and it only contains information about the most recent version of the models in the repository. Hawk has been recently extended with time-aware indexing and querying capabilities [García-Domínguez 2018, García-Domínguez 2019], with the ability to record the full history of the indexed models into a temporal graph [Hartmann 2017]. Each version of the model is associated with a time point that corresponds to when the change was made. A time-aware dialect of EOL is provided to query historic information about the models, such as when the models started exhibiting some behaviour, or when a model element was first added to the repository. These time-aware extensions allow for efficient investigation and analysis of the evolution of models in file-based repositories.

2.4 Overview of the Model Comparison Process

Model comparison is the process of extracting the changes between two or more sets of models. Model comparison is often a complex and time-consuming process that involves two main tasks: *model matching* for identifying corresponding elements among the different sets of models and *model differencing* for computing the differences among the corresponding elements that have been identified during the matching phase. Model matching is an NP-hard problem and existing approaches to address this challenge often targets a specific modeling language [Lin 2007]. Kolovos et al. [Kolovos 2009] identified three main approaches to model matching and they are static-identity, signature, and similarity matching.

1. **Static Identity matching**. This approach involves the assignment of a static, persistent and unique identifier to each model element when the element is created. Matching

elements are identified based on the unique identifier. Although this approach is fast, it cannot be used when the candidate models are developed independently or when the underlying modelling infrastructure does not support the maintenance of the unique identifiers.

2. **Signature matching**. This approach adopts a dynamic identifier for each element based on some of the properties (e.g., name and type) of the elements. This means that the identifier does not need to be persisted by the underlying modelling infrastructure and the approach can be used to match elements belonging to models that are constructed independently of each other. Unfortunately, this approach often involves significant user effort to configure the function that will be used to generate the dynamic identifier.
3. **Similarity matching**. This approach identifies matching elements based on the aggregate similarity of their features. This approach is more time-consuming, but it tends to be more accurate than the other two approaches [Kolovos 2009].

The model differencing phase uses difference algorithms [Alanen 2003] to extract the differences between the corresponding model elements. A typical model differencing algorithm is able to detect three kinds of differences: the addition, deletion and modification of model elements.

2.5 Chapter Summary

This chapter has provided basic information about models and metamodels within the context of MBSE. The chapter has also provided an overview of the relationship between MDE and MBSE paradigms. Five modelling technologies related to the research presented in this dissertation have also been discussed. The five modelling technologies are Hawk, LabVIEW, Simulink, EMF and Massif. The core of this dissertation's research extends the Hawk tool to support the analysis of models developed with the LabVIEW and Simulink platforms. Hawk primarily supports EMF-

based models; hence, the Massif tool was used for the bi-directional transformation of Simulink models to EMF models, while a custom parser was developed to transform LabVIEW models to EMF models. Finally, this chapter has provided a summary of the model comparison process, including the existing approaches for identifying matching elements across models to be compared.

CHAPTER 3

CHANGE ANALYZER FOR QUERYING CHANGE INFORMATION ACROSS VERSION HISTORIES OF MODELS

This chapter presents an overview of the tool developed to analyse changes across the version history of models. The chapter discusses the motivation behind the tool, how the changes are extracted and stored, and a set of language constructs to facilitate the efficient querying of change information across the version history of a model. The chapter concludes with an evaluation of the developed tool by comparing the Change Analyzer with a generic tool for querying models in repositories.

3.1 Chapter Introduction

Models are software artifacts that are subject to continuous change. The extraction and analysis of these changes is important for the maintenance of models. For example, when fixing a bug, we may want to know the changes that have been executed when fixing similar problems in the past. An important step in the analysis of changes executed on models is the extraction of such changes via model comparison techniques.

Model comparison approaches and techniques are used to extract the changes that have been implemented between successive versions of a model [Kolovos 2009]. Model comparison techniques help to compare versions, understand the type of change that was executed, and merge two variants of a model into a single consistent model. Unfortunately, existing model comparison tools such as EMFCompare [Toulme 2006] can be computationally expensive and time consuming. Yet, most of these comparison tools do not provide a persistence mechanism for

storing the results of the comparison process. This means that the results of the comparison process are often discarded when the program is terminated and the comparison operation needs to be recomputed if the changes are needed again.

Similarly, most model comparison tools do not provide support for querying the changes in a model. For example, queries such as “Find all the elements that were renamed in a particular version” has to be done manually. Furthermore, model comparison tools often focus on two or three sets of models at once; hence, it may be difficult to extract information that spans across the entire version history of the models such as the number of times a model element has been changed (to determine change proneness) throughout its version history.

This chapter presents our approach for developing a Change Analyzer that provides capabilities for reasoning about changes that have been executed across a model’s life cycle. The Change Analyzer extends the Hawk tool to support efficient querying of the history of changes executed on the models. The evaluation of the Change Analyzer shows that it can be used to support more concise change-based queries that execute faster than change-based queries developed via the Hawk tool alone.

3.2 Overview of the Change Analyzer

A Change Analyzer has been developed to facilitate the querying of change information across the version history of models. The Change Analyzer extends the Hawk tool for indexing models in software repositories and consists of two main components: a persistence mechanism to store changes and a set of language constructs to query the stored changes. The persistence mechanism is implemented by extending the model updater component of Hawk, thereby making it possible to store the changes computed by Hawk when the model is being updated. The set of language constructs was implemented by extending the query engine in Hawk to support queries that

enhance the extraction of change information across version histories of models. Figure 3.1 provides a graphical overview of the Change Analyzer. The figure also captures how we extended the Hawk tool via a change-based model updater and a change-based query engine.

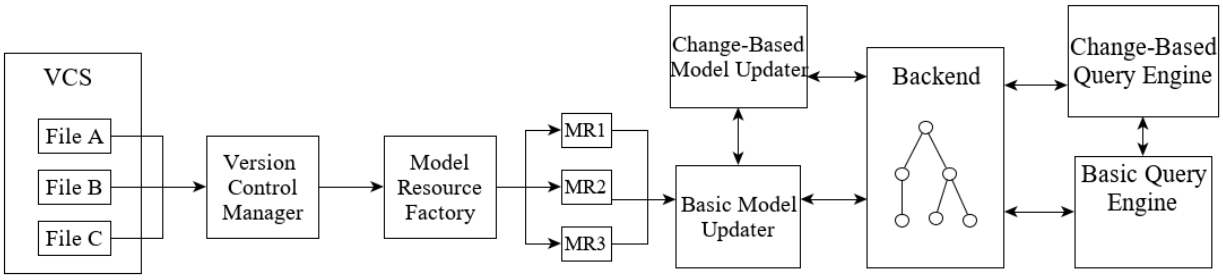

Figure 3.1. Overview of the Change Analyzer

3.3 Indexing and Storage of Changes across Successive Versions

Whenever the VCS manager in Hawk detects changes in a model repository, the model updater component computes the changes between the current state of the model that has been stored in the backend database and the new version of the model. The computed changes are used to update the model in the backend database. These changes are then discarded when the update process is completed. We extend the model updater to capture and store these changes before they are discarded by Hawk.

Two categories of changes are captured by the model updater: the type of change and the changed elements. We extract and store these two categories of change to facilitate efficient queries via the type of change or the changed element. Hawk adopts a Signature approach to match identical elements in models to be compared during the update. This signature approach is implemented by assigning unique dynamic identifiers to model elements and creating indexes to link corresponding entities between model elements in the repository and the elements that are stored in the Hawk backend database. We used the dynamic identifier used by Hawk to establish

correspondences between the change types and the changed elements, as well as between the changed elements captured by the Change Analyzer and the corresponding element that is indexed in Hawk. This ensures that the default indexing and querying mechanisms in Hawk will still function optimally in the Change Analyzer and the results of the queries generated by the Change Analyzer can still be used for further analysis in the default Hawk configuration.

The indexing of the addition and deletion of elements is straightforward because it involves only the association of the change type with the change elements. However, the modification of elements is more complex because it is possible that multiple attributes or references can be modified between any two successive versions. Furthermore, the indexing of modified elements often involves the preservation of the old value and new value, which is in contrast to the addition or deleting of elements where only one of the values is required because the other will be null. Listing 3.1 is the algorithm we used to index and store changes across two successive versions.

```
foreach change to elements do
   create a changenode in the graph and assign change type and changed element
   if (change type = modify)
         set changeattribute with attribute name, previous value, new value
         foreach change in the references do
            create a relationship between this changenode and the changenode of
            the referenced element
         end
   end
end
```

Listing 3.1. Algorithm to Index and Store Changes in the Backend Database

3.4 Language Constructs to Query Changes

To facilitate the efficient query of change information that has been indexed by the Change Analyzer, we provide a set of language constructs to support the extraction of change

information. We introduced three main sets of constructs and adapted two existing constructs in Hawk to support the query of change information. The main sets of language constructs are queries related to change elements, queries related to change types, and queries to extract a summary of changes executed on the models.

1. **Queries related to changed elements**. This provides support to extract information about the change history of model elements. The user can extract information about when the model element was created, all the updates executed on the element during its lifecycle, and when the element was deleted. For example, Listing 3.2 is a code snippet that returns the latest changes that have been executed on all elements of type “Block”.

```
for (i in Block.allInstances){
    i.getChanges().println;
}
```

   Listing 3.2. Code Snippet for Latest Changes to Block Elements

2. **Queries related to change type**. This supports the extraction of information according to a particular change type. Currently, the Change Analyzer supports three types of change: *add* for addition of new elements, *delete* for deletion of old elements, and *modify* for elements whose attributes or references have been updated from one version to the next version. For example, the function “Model.addedChanges()” returns all the elements that were added to the repository.
3. **Summary of changes**: This set of language constructs provides support for extracting the summary of changes executed on all the models that have been indexed in the backend database. This feature returns all the models that have been added, deleted or modified. It also returns the total number of model elements that have been added, deleted or modified.

The Change Analyzer has also adapted two sets of language constructs that are used in the current configuration of the time-aware Hawk. The language constructs are the version scoping and timeline execution. The following paragraphs explain these constructs.

1. **Basic Timepoint / Version scoping**. Every model that is indexed in the time-aware Hawk is associated with a timepoint that shows when that specific state of the model was committed to the repository. *Timepoint scoping* provides capabilities for the user to specify a specific time range such that only changes that occur within that time range are returned. We also provided support for a similar construct called *version scoping* where a version is defined as any timepoint where any change was detected on any of the models. The versions are then incrementally labelled where version 1 corresponds to the initial state of the models while the last version number corresponds to the total number of versions in the database. Version scoping allows users to limit the execution of queries to a specific version or a range of versions. Both version and timepoint scoping gives the users more flexibility because it is possible that there will be many versions within a short time frame or there may be a very long time between two successive versions.
2. **Timeline**. Timeline capabilities provide support to find occurrences of a particular query across all the history of the model. Therefore, each query is evaluated as a predicate and the result is automatically generated for all the timepoints where the predicate is true. This approach is much faster than writing queries that iterate through the entire history of a model because the search for elements that satisfies a specific predicate can be executed by an efficient look-up rather than the more computationally expensive iteration of elements [Barmpis 2015].

3.5 EMFCompare Extension to the Change Analyzer

EMFCompare [Toulme 2006] is a model comparison framework for extracting differences between two or three sets of models. EMFCompare supports four types of changes between two sets of models: *add* to indicate addition of new elements, *delete* to show removal of existing elements, *change* to indicate the change of attribute values, and *move* to indicate reordering of model elements.

We provide an EMFCompare extension to the Change Analyzer to offer flexibility in the model comparison process. EMFCompare adopts the similarity-based approach for matching candidate elements during a model comparison process. The similarity-based approach is more efficient but slower than the signature approach provided by the basic Change Analyzer. Furthermore, EMFCompare also provides a new type of change called “MOVE” to indicate elements that have been reordered. This change type is not provided in the default configuration of the Change Analyzer. Finally, we mapped the *change* type returned by EMFCompare to the change type *modify* in the default Change Analyzer for consistency in the indexing and querying of change information.

To facilitate the extension of the Change Analyzer to support EMFCompare, each version of the models indexed in Hawk was extracted, converted to an XMI format, and then compared with its succeeding version using EMFCompare for efficient extraction of changes between the versions. These sets of changes extracted were then indexed and stored based on the type of change and the changed elements, similar to the process described in Section 3.2. It should be noted that while the EMFCompare extension offers more flexibility in terms of the type of changes that are available, it does not offer tight-coupling of elements with the default Hawk configuration. For example, the extension will be unable to provide the change history of specific

elements that has been generated by a Hawk query. However, the EMFCompare extension can still answer generalized queries such as extracting the summary of changes or all the changes associated with a model element type (e.g., all the nodes in a graph).

3.6 Evaluation of the Change Analyzer

The default time-aware configuration of Hawk is able to track the history of model elements. However, the Change Analyzer discussed in this chapter offers a faster approach to managing change history. For example, to search for "rename" operations in Hawk, we need to iterate through all the elements and compare the value of its attribute "name" with the value in the preceding version. However, in the Change Analyzer, we need to iterate through only elements that are associated with the "modify" change type and whose changed elements involve a modification of the "name" attribute. Hence, in the context of the scenario presented above, this evaluation process aims to validate the following research hypothesis.

"A dedicated Change Analyzer can be used to construct queries that are more concise and perform better than queries developed via a generic query language."

The evaluation process compares the Change Analyzer with the Hawk tool. The evaluation process will focus on two main properties: conciseness of the queries and execution time.

3.6.1 Case Study

Refactoring activities address changes to the structure of a program without altering the program's external behaviour [Fowler 2018, Opdyke 1992]. Refactoring is important for correcting errors, implementing new features and correcting bad smells in programs [Silva 2016]. Therefore, refactoring operations are important to enhance the maintenance and readability of software programs [Tsantali 2018].

For example, the **Extract Superclass** operation is a refactoring mechanism for extracting the common elements in two or more classes [Fowler 2018]. This is often achieved by creating a new superclass that abstracts the common features in these classes. This refactoring operation helps to reduce duplicate features that are spread across these classes [Zhang 2005]. Figure 3.2 provides an example of an “extract superclass” refactoring task.

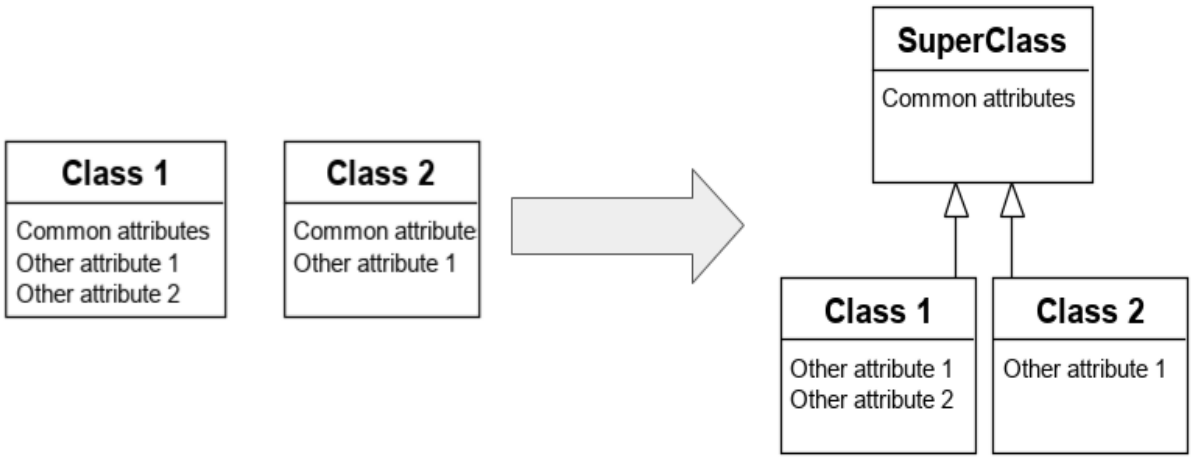

Figure 3.2. An Extract Superclass Refactoring Task

The detection of refactoring operations is crucial for understanding changes to a program’s structure, adapting configurations of API libraries, merging versions of code, automated code completion, and code review [Alves 2014, Balaban 2005, Foster 2012, Ge 2014]. Hence, a number of techniques and tools have been developed to support the detection of refactoring operations [Dig 2006, Prete 2010, Tsantalis 2020]. However, the majority of these refactoring detecting tools and approaches are focused on text-based programs.

This evaluation process demonstrates the capability of our Change Analyzer to detect Extract Superclass refactoring operations in models and extract changes across successive versions of models. We detect the extract superclass operation by iterating through the newly added models to search for a model that is a superclass for two more models that already exist in

the previous version. Listing 3.3 is a query for the Change Analyzer that returns true if the Extract Superclass refactoring task was executed in a particular version. We compared the results with the results produced by the query generated with the Hawk configuration. Listing 3.4 is a sample query for Hawk that detects if the Extract Superclass refactoring was executed in a specific version. We also evaluated the capability of the Change Analyzer to extract changes across successive versions of models. The query "Model.getChanges()" returns all the changes in the current version compared to the previous version. Listing 3.5 is a sample query for Hawk that extracts similar sets of changes. The Hawk query also requires the specification of the timepoint when the previous version was indexed in Hawk in order to extract deleted elements. All the queries were constructed for models that conform to the Simulink Ecore-based metamodel[11] provided by the Massif tool. A SimulinkModel element represents the root element of a Simulink model while a ModelReference element is used to represent a superclass model that is linked to the original model.

```
var addedModels= SimulinkModel.addedChanges();
var referenceModels=new Collection;
for i in ModelReference.modifyChanges(){
    if( addedModels.includes(i.referenceModel)){
        If (referenceModels.includes(i){
            return true;
        }
        else{
            referenceModels.add(i);
        }
    }
}
return false;
```

Listing 3.3. A Change Analyzer Query to Detect Extract Superclass Refactoring

[11] https://viatra.github.io/massif/user/simulink_ecore_metamodel.html

```
var addedModels= SimulinkModel.allInstances().select(i| i.previous.isUndefined());
var referenceModels=new Collection;
for (i in ModelReference.allInstances){
    if (i.previous ==null){
        if (referenceModels.includes(i)){
            if (addedModels.includes(i.referenceModel)){
                return true;
            }
            else{
                referenceModels.add(i);
            }
        }
    }
}
return false;
```

Listing 3.4. A Hawk Query to Detect Extract Superclass Refactoring

```
var addedElements= Model.allInstances().select(i| i.previous.isUndefined());
var modifiedElements= Model.allInstances().select(i| i <> i.previous);
var deletedElements= Model.allInstancesAt(1519538494000L).select(i|
i.next.isUndefined());
var changedElements = addedElements.includingAll
        (modifiedElements.includingAll(deletedElements));
return changedElements;
```

Listing 3.5. A Hawk Query to Extract Changes between Two Versions

3.6.2 Results of the Query Execution

We executed the queries in Listing 3.3 and the query “Model.getChanges()” via the Change Analyzer. We also executed the queries in Listing 3.4 and Listing 3.5 via the default Hawk configuration. The dataset used for the experiment was extracted from a Simulink repository[12] and converted to Hawk via the Massif tool described in Section 2.3.4. The selected Simulink

[12] https://github.com/petercorke/robotics-toolbox-matlab

repository contains 23 models with 194,239 model elements. The developed queries were executed 5 times in each of the tools under evaluation. On the average, the indexing phase in Hawk took 2389ms while the indexing phase in the Change Analyzer took 2408ms. This shows that the indexing phase in the Change Analyzer was 19ms or 0.79% slower than the indexing phase in Hawk. However, this is a one-time cost since the indexing was done once. This cost is due to the extra need to index the changes along with the models, since the Hawk tool only indexes the models.

The execution of the queries in Listing 3.4 via Hawk took 5ms while the execution time for the queries in Listing 3.3 via Change Analyzer took 1.4ms. This is because Hawk needs to iterate through all the instances of ModelReference and SimulinkModel elements while the Change Analyzer only needs to iterate through the SimulinkModel elements that were recently added and the ModelReference elements that were recently modified. The execution time for the query "Model.getChanges()" in the Change Analyzer was 86ms, while the execution time for the queries in Listing 3.5 via Hawk could not be determined. This is because queries such as those in Listing 3.5, which involve iterating through all the elements (e.g., getting all the newly added elements and not just elements of a specific type), often causes the program to crash on an 8GB RAM Windows computer because all the nodes in the graph database must be loaded into the memory. Hence, the evaluation experiment has to be repeated on a smaller dataset[13] containing a single model with 21,932 elements. At the end of the second experiment, the execution time for Model.getChanges() in the change Analyzer was 38ms while the execution time for Listing 3.5 in Hawk was 887ms. Finally, the slower execution of the queries in Hawk is a recurring cost that is incurred every time a query is executed.

[13] https://github.com/ktalke12/MATLAB_MiP

3.6.3 Analysis of the Query Results

The queries developed with the Change Analyzer are more concise than the queries developed for Hawk. The query in the Change Analyzer involves a single line of code and 9 lines of code with two conditional statements, while the query in Hawk involves 5 lines of code with 3 conditional statements and 10 lines of code with four conditional statements. This shows that the Change Analyzer can be used to write more concise queries thus validating the first part of the hypothesis.

The results of the queries reported in Section 3.5.2 shows that the queries generated with the Change Analyzer are able to execute faster than the queries generated with Hawk. This comes at a one-time cost of slightly higher time to index the models because the Change Analyzer needs to index both the models and the changes. The results of the query execution shows that the Change Analyzer can be used to construct queries that executes faster, thus validating the second part of our hypothesis.

3.7 Related Work

Change is inevitable in a software development process; hence, a number of techniques and approaches have been developed to manage changes to modelling artifacts in MDE. Briand et al. [Briand 2003] proposed the use of defined rules in analysing the impact of changes across versions of UML class diagrams. Lin et al [Lin 2015] and Wang et al. [Wang 2018] introduced a workflow approach for analysing the impact of changes in SysML models. Rajabi et al. [Rajabi 2019] used colored Petri nets to handle changes in UML models and ensure consistency across design artifacts. Muller et al. [Muller 2014] and Keller et al. [Keller 2012] leveraged impact rule specifications to define the impact of changes in UML diagrams. Kchao et al. [Kchao 2012] developed a metamodel to capture the impact of changes in models while Rapos et al. [Rapos

2017] adopted change propagation techniques to extract the potential impact of a change in Simulink models. Yohannis et al. [Yohannis 2017] developed a change-based model persistence format that preserves the change history of models. This change-based format can be used to facilitate faster processing of models during a collaborative session.

Furthermore, many works have focused on identifying common elements in a pair of models in order to extract the differences between the models. Addazi et al. [Addazi 2016] developed an approach to support the matching of models based on semantic interpretation of the models via a lexical database. Lin et al. [Lin 2007] developed an approach for extracting differences between domain-specific models via algorithms that are agnostic to multiple metamodels. Nejati et al. [Nejati 2007] exploits the structural and behavioral properties of statechart models to develop a similarity-based approach for matching and merging statecharts. Yohannis et al. [Yohannis 2019] developed a model comparison approach that compares only the changed elements in candidate models instead of comparing all the elements in the models. Treude et al. [Treude 2007] developed a multidimensional tree-based technique to match model elements and extract differences between the two sets of models.

Similarly, Xing et al. [Xing 2005] developed UMLDiff to extract the differences between two successive versions of UML models generated from the reverse engineering of object-oriented code. Toulme et al. [Toulme 2006] developed a customizable tool named EMFCompare for extracting differences between models. The tool matches model elements based on the similarity of their features and provides capabilities of extending all components of the differentiation engines to meet user-specific requirements. The approaches and tools discussed so far have focused on extracting the differences between two sets of models and do not capture the changes across the entire version history of the models.

Work has also been done on change analysis in software engineering. Fluri et al. [Fluri 2006] developed a taxonomy of change types by classifying changes based on their potential impact on the source code. Fluri et al. [Fluri 2008] implemented a clustering algorithm to discover patterns of change types and when they occur. Their results show that specific development activities such as code cleanup often require similar types of changes. Hussain et al. [Hussain 2019] exploited the history of change history and the relationship among the classes in source code to identify potential unstable classes. The effectiveness of the approach has been evaluated on 10 open-source projects, and the results show up to 16% increase over traditional approaches to predicting class instability. Stevens et al. [Stevens 2013] developed the QwalKeko tool for querying the evolution history of a program. QwalKeko has been used to detect potential refactoring activities in source code. The approaches discussed in this paragraph have focused on the source code developed with text-based languages and do not support the complex relationships among visual model elements.

Finally, García-Dominguez et al. [García -Dominguez 2019] provided capabilities for annotating and querying the version history of models. However, they do not explicitly capture the changes that led to evolution of the models. Therefore, while it is quite easy to extract information about the state of a model at a specific time point in history, simple change-based queries such as detecting all the elements that are renamed, will be computationally expensive for models with a large number of elements. This chapter extends their work to provide first-class support for querying changes across the version history of the models.

3.8 Chapter Summary

This chapter introduced a Change Analyzer that can be used to query changes across the version history of models in a repository. The Change Analyzer was developed by extending the Hawk

tool to provide first-class support for indexing, storing and querying changes in a repository. Information related to the type of change and the changed elements were extracted during the model updating process in Hawk and stored in a backend database. A set of language constructs has been provided to facilitate the efficient querying of changes. The Change Analyzer has been evaluated by comparing it with the default Hawk configuration. The results of the evaluation process shows that the Change Analyzer can be used to construct queries that are more concise and execute faster than queries developed with the default Hawk configuration.

CHAPTER 4

TOOL INTEGRATION TO SUPPORT CHANGE ANALYSIS OF LABVIEW AND SIMULINK MODELS

This chapter discusses how we extend the Change Analyzer presented in Chapter 3 to support LabVIEW and Simulink models. The chapter also discusses the development of an open-source metamodel for LabVIEW to facilitate the adaptation of MDE approaches to the LabVIEW ecosystem.

4.1 Chapter Introduction

Systems Engineering projects are inherently complex, traverse through different phases and often involve various stakeholders working together on a project [Estefan 2007, Hergiz 2014]. Unfortunately, for most large-scale projects, different tools are used across different phases of the system's life cycle or even within the same phase [Malone 2016]. Furthermore, the system often needs to interact with other systems with different architecture, data format and complex interfaces; thereby making it challenging to effectively integrate all the required components and subsystems [Malone 2016]. Hence, tool integration is a fundamental challenge that needs to be addressed in most systems engineering projects.

This chapter presents the tool integration techniques adopted in this dissertation to facilitate the adaptation of MDE techniques to LabVIEW and Simulink. LabVIEW and Simulink tools are currently used by millions of users to model, develop and analyze complex systems. However, existing MDE tools and techniques often adopt the EMF standard and do not directly support LabVIEW and Simulink models. Although a number of MDE research efforts exist that

are related to Simulink models, MDE research with LabVIEW models is very limited even though there are a number of opportunities where LabVIEW developers will benefit greatly from MDE research. For example, large LabVIEW models tend to be difficult to analyze manually in order to reason over the design characteristics, identify elements that can be affected due to a change, and locate potential design problems such as redundancies or unused code. MDE can be used to analyse these models and identify potential design problems [Costa 2016]. However, there is no open-source LabVIEW metamodel that is easily accessible to facilitate the adaptation of MDE tools and approaches to LabVIEW models. This makes it challenging to carry out automated analysis of LabVIEW models.

The research presented in this chapter also provides another opportunity to address the need for open-source modelling artifacts in MDE. Open-source datasets are essential to conduct experiments for model analysis and empirical research. Simulink has over 4 million users, while LabVIEW has over 300,000 users. These tools can provide the much needed data for empirical research in MDE. Furthermore, the sets of developers who often use Simulink and LabVIEW tools are likely to be different from the set of developers who used traditional MDE tools; therefore, the datasets of LabVIEW and Simulink models are likely to enhance the diversity of the available datasets for empirical research in MDE and provide new perspectives to modelling.

4.2 Tool Integration to Support Change Analysis of LabVIEW Models

The Hawk tool was designed to work directly with EMF-based models; hence, the Change Analyzer that extends the Hawk tool also works directly with EMF-based models. However, Hawk facilitates the adaptation of Hawk with other modelling formats via a model parser. The model parser is expected to parse any file object and return an EMF-resource. Furthermore,

Hawk also requires an Ecore-based metamodel for the models generated via the model parser in order to validate such models and also facilitate the querying of the indexed models.

We extended the Hawk tool to support LabVIEW models via a model parser and a LabVIEW metamodel that captures the properties and relationships across elements in existing LabVIEW models. The model parser extracts model elements from LabVIEW model files and converts the elements to a format supported by Hawk. The Hawk tool then validates the resulting Hawk-format model against the LabVIEW metamodel, thereby ensuring the consistency of the models processed by Hawk. Our adaptation of Hawk allows efficient querying of LabVIEW models in file-based repositories. Figure 4.1 is a graphical overview of our extended Hawk tool. The Change Manager component in the figure captures the change-based model updater and change-based query engine discussed in Section 3.2. The next section discusses the LabVIEW metamodel in detail.

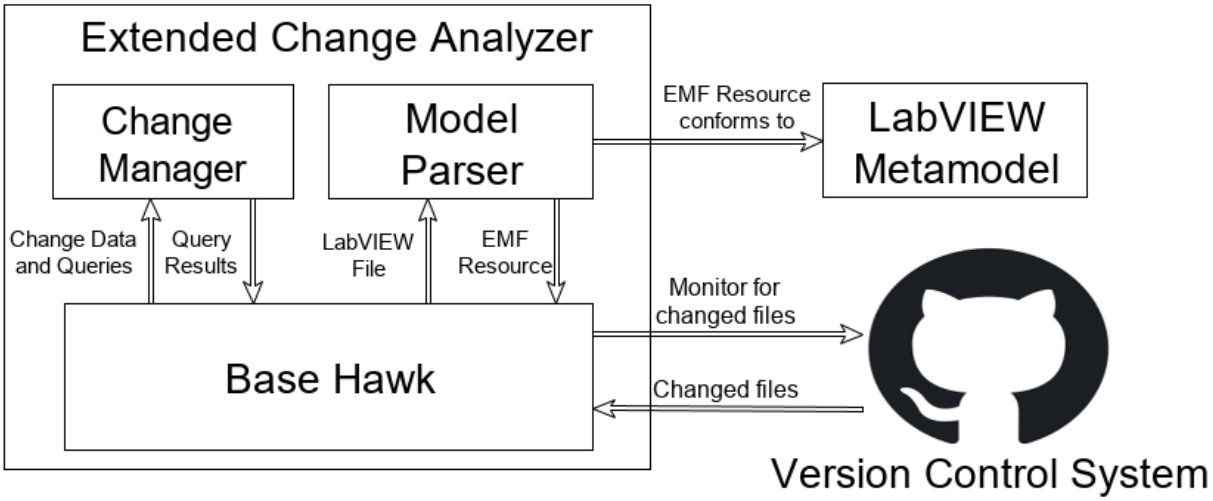

Figure 4.1. Extended Change Analyzer to Support LabVIEW Models

4.3 A LabVIEW Metamodel to Support Automated Analysis

Models are created via a modelling language that is defined by a metamodel. Similar to grammar for a programming language, metamodels can also be used to formalise a modelling language by capturing its essential concepts and the relationships that exist among the concepts [Paige 2014]. This helps to ensure a consistent use of the language and also validates the syntactic correctness of a model written via the modelling language [Ledeczi et al. 2001]. An essential foundation for any MDE process is a precise and robust metamodel that sufficiently captures the complexities of artifacts used during the development process.

A precise and semantically rich modelling language is important for developing models that can capture and correlate different aspects of the problem space that is related to the domain of interest. A modelling language often contains three main components: a concrete syntax that captures concepts in the domain, an abstract syntax that specifies the relationships among the concepts, and the semantics that specifies the structural and behavioural properties of the relationships in the abstract syntax, thereby increasing the precision and reducing the possibility of incorrect assumption. A metamodel captures these three essential elements that are required in a modelling language [Cho 2011].

Metamodelling provides a capability to define new modelling languages in a precise manner, thereby making it easier to manage, understand, and manipulate the models created with the modelling language [Clark 2008]. For example, metamodelling makes it possible to transform models to another language via mappings across the two language metamodels, generate instance models for testing purposes, and create layers of abstraction to hide implementation-specific details, thereby improving the developer’s productivity.

Metamodels also assist in adapting software tools to different modelling languages and technologies [López-Fernández 2015]. This increases the interoperability of such tools and makes it possible to adapt the tools to new environments. For example, by loading a metamodel, a tool can automatically understand the constructs in a model, user interactions with such a model, and how to automate important software engineering tasks such as design analysis and testing. Finally, metamodels provide a set of standards for developers to interact with the modelling language, thereby simplifying the automation and formalization of processes involved in construction of models via the language [Atkinson 2003][Selic 2003].

LabVIEW is currently used by hundreds of thousands of users but there is no open-source LabVIEW metamodel that is easily accessible. This makes it challenging to carry out automated analysis of LabVIEW models. The following subsections discuss how we constructed an EMF-based open-source metamodel that sufficiently captures the essential properties of a LabVIEW model. The metamodel has been developed by examining over 100 LabVIEW models and following standard object-oriented concepts such as abstraction and inheritance. The metamodel captures several types of LabVIEW dependencies and the relationships that exist across these dependencies. We believe that this is the first open effort to describe a metamodel for LabVIEW [Popoola 2019].

4.3.1 Construction of LabVIEW Metamodel

Section 2.3.2 shows that each model in LabVIEW is divided into separate files called VI, and the new generation of LabVIEW software (called LabVIEW NXG) divides each VI model into three main components: 1) the Front Panel that serves as the front end-user interface of the application to be developed, 2) the Block Diagram that serves as the backend and contains the control code, and 3) the Icons that handle images. Figure 4.2 shows the Front Panel and Block Diagram of a

sample LabVIEW model that calculates the length of a string. The model takes an “input string” and computes its length via the “string length” function that is shown in the Block Diagram. The “length” object displays the output of the computation. This research work focuses on the Block Diagram, which is the main component that stores the control logic of the execution model.

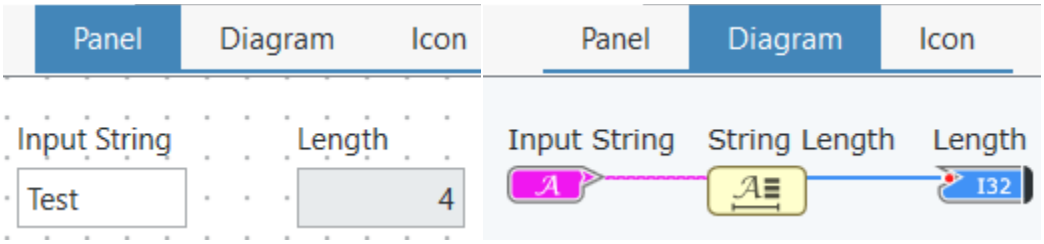

Figure 4.2. A Front Panel and Block Diagram to Compute the Length of a String

We constructed an EMF-based metamodel that captures all the major elements in the BlockDiagram. The metamodel was developed by curating over a hundred LabVIEW models from the LabVIEW examples repository that is present in the LabVIEW IDE. The metamodel was developed via the following recurring steps: 1) identify class elements in the LabVIEW models, 2) detect common patterns across elements, 3) extract relationships among the elements based on detected patterns, and 4) update the metamodel to accommodate new elements.

A. Identification of Class Elements

The first step is to generate the concrete syntax of the modelling language by extracting the class elements, attributes and methods from sample models. A typical LabVIEW NXG model is serialized in an XML-compatible text format, hence an XML parser was developed to extract BlockDiagram elements from a LabVIEW model, and then the needed class structure was extracted from the generated XML. Two Base elements were initially identified in the development process, the Object class which would be the root of all the class elements, and a

BlockDiagram class that serves as the root instance and has a containment reference to all the other elements in a BlockDiagram. Figure 4.3 is a graphical representation of the initial elements and the relationship between them.

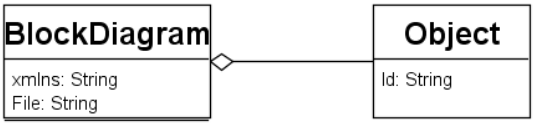

Figure 4.3. Initial structure of LabVIEW Metamodel

Subsequent base elements were identified by the direct containment references in the BlockDiagram, while other elements were identified recursively based on the containment reference of its parent element. For example, in Listing 4.1 that has been extracted from the LabVIEW model in Figure 4.2, the following class elements could be identified: DataAccessor, NodeLabel, StringLength, and Wire. The Data Accessor class also contains a reference to the Terminal class, while the NodeLabel class contains reference to p.Text.

```
<BlockDiagram Id="12">
        <DataAccessor Id="16" Label="20" >
            <Terminal DataType="String" Direction="Output" Id="Value"/>
        </DataAccessor>
        <DataAccessor Id="17" Label="21" >
            <Terminal DataType="String" Direction="Input" Id="Value" />
        </DataAccessor>
        <StringLength Id="18" Label="22" Template="Icon" />
        <Wire Id="23" Joints="N(16:Value)|(130,-5) N(18:string)|(170,-5)" />
        <Wire Id="24" Joints="N(18:length)|(210,-5) N(17:Value)|(263,-5)" />
        <NodeLabel AttachedTo="16" Id="20" >
            <p.Text>Input String</p.Text>
        </NodeLabel>
        <NodeLabel AttachedTo="17" Id="21">
            <p.Text>Length</p.Text>
        </NodeLabel>
        <NodeLabel AttachedTo="18" Id="22" IsReadOnly="True">
            <p.Text>String Length</p.Text>
        </NodeLabel>
</BlockDiagram>
```

Listing 4.1. A Sample LabVIEW Model Serialized in XML

B. Detecting Common Patterns

The next step is to identify class elements that have a similar set of attributes and methods. This makes it possible to identify potential abstractions and inheritance relationships across the model elements. Many elements were discovered to have similar attributes and containment references. For example, both the While loops and For loops should have a containment reference to almost every class element. This similarity led to the creation of a new class named Loop that has a containment reference to almost every other known class. Figure 4.4 shows the two classes and the common elements identified.

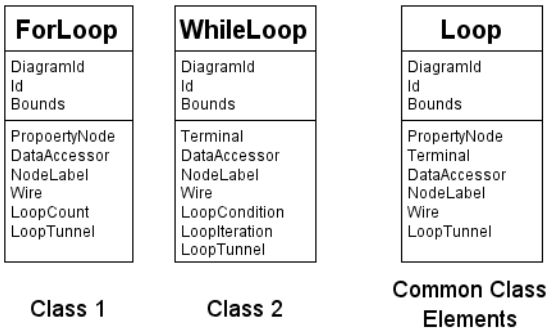

Figure 4.4. Two Class Elements and a Third Class with the Common Features

C. Extract Relationship across Model Elements

In order to create the abstract syntax and semantics for LabVIEW, relationships across model elements were extracted. Two forms of relationships were identified: relationships based on containment references to other elements and relationships based on the set of similar patterns. Containment references help to identify new class elements and build part of the abstract syntax representation, while the pattern-based relationship completes the abstract syntax representation and facilitates the description of the semantic representation. Figure 4.5 shows a pattern-based relationship between the ForLoop and WhileLoop through the same parent class.

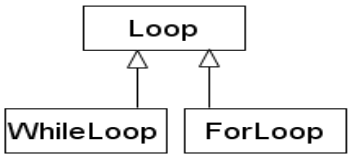

Figure 4.5. A Loop Class Based on Common Patterns in WhileLoop and ForLoop

D. Update of Metamodel

After the construction of the initial metamodel, we validated the metamodels against many sample models and updated the metamodel to accommodate new class elements and relationships. For example, when another model contains the “if” block and it appeared to have many containment references similar to the Loop class, the Loop class was renamed to GroupObject class and the If class element was created as a subclass of the GroupObject class. Figure 4.6 shows an update to the metamodel to accommodate the new If class.

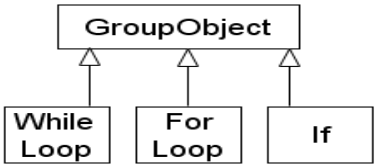

Figure 4.6. Updated Metamodel to Accommodate the If Class

This four-step process was repeated for each unidentified element that was discovered in a new LabVIEW model.

4.3.2 Description of the Current Metamodel

The metamodel currently contains the following classes: BlockDiagram, Object, GroupObject, DataType, NodeObject, NodeLabel and Wires. Every Element extends the Object Element and the BlockDiagram serves as the root instance of the metamodel. Figure 4.7 gives a simplified view of the metamodel and the relationship across the elements. The complete metamodel is publicly available (please see http://bit.ly/LMeta). A description of each major class is given below.

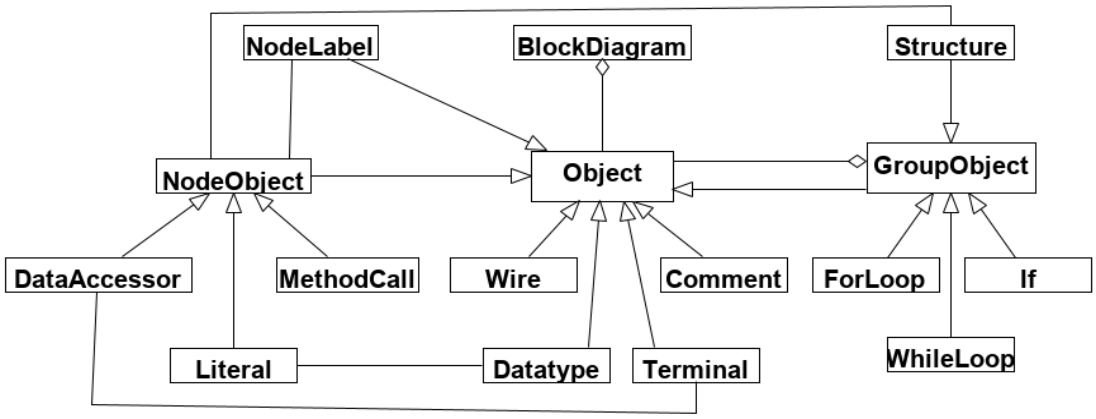

Figure 4.7. A Simplified View of the Metamodel

1. **Object**: This is the Parent class element for all the elements in the metamodel. It contains three main attributes: 1) Id, which is the class identifier, 2) Bound, which specifies the location of the Object on the Block Diagram, and 3) Visibility, which shows whether the Object is visible or not.
2. **BlockDiagram**: This is the root instance of the metamodel and it serves as a container reference for all the objects that are captured in the metamodel. The BlockDiagram is also an instance of an object.
3. **GroupObject**: The GroupObject captures class elements that generally have a containment relationship with any other elements. Examples include Loops, If statements, and Events.
4. **NodeObject and NodeLabel**: Node Objects are used as place holders for other elements. They can store data and retrieve or programmatically control properties of the referenced elements. The Node Labels store information about the Node object. The NodeObject and NodeLabel often occur in pairs in a LabVIEW model and also contain opposite references that link them together.
5. **Structure**: A Structure class refers to a set of elements that is composed of other elements and divides the model into a form of hierarchical sub-models. They are mainly used for

controlling the execution of elements embedded in the structure. An example is the event case structure that allows elements embedded within it to be executed only when a particular event is triggered.

6. **Terminals**: Terminals are elements that can be used as variables to store a particular type of data or reference other objects. They are often associated with a data type. Terminal are connected via wires.
7. **Wires**: Wires connect exactly two elements together and make it possible to transfer data from one element (known as source) to the other element (called sink). Wires also ensure that only elements with compatible data types can be connected together.

4.3.3 Evaluation of the Metamodel Completeness

The constructed metamodel has been validated against models in 10 LabVIEW NXG repositories on GitHub. The projects contained a total of 81 models and a model validator was used to check that all the models were valid instances of the metamodel. Table 4.1 provides an overview of the 10 LabVIEW repositories.

4.3.4 Limitations of the Metamodel

The metamodel discussed in this section has been validated against a limited set of models and there is no guarantee that the metamodel captures all of the existing concepts in LabVIEW. For example, class elements and relationships associated with third-party libraries are not captured in the current version of the metamodel. Furthermore, new concepts that are incompatible with the existing metamodel may also be introduced in the future. However, because the metamodel presented in this paper is extensible and open source, we believe that the metamodel can be modified to capture previously unidentified concepts and relationships.

Repo/ Properties	URL for Repository	# of Models	# of Elements
Repo 1	https://github.com/ni/webviexamples	1	680
Repo 2	https://github.com/JKISoftware/ JKI-State-Machine-NXG	10	5337
Repo 3	https://github.com/ni/labviewnxg-jenkins-build	14	270
Repo 4	https://github.com/rajsite/ webvihack	11	654
Repo 5	https://github.com/prestwick/ customizing-webvis	5	1650
Repo 6	https://github.com/therinoy/ LabVIEW-NXG-BL1.1W-Web-App-Project	3	1312
Repo 7	https://github.com/navinsubramani/ develop-and-deploy-WebVI	15	487
Repo 8	https://github.com/wimtormans/ LabVIEWRaspberryPI_ IndoorMonitoring	8	1947
Repo 9	https://github.com/eyesonvis/ niweek2019-webVI-hands-on	3	538
Repo 10	https://github.com/doczhivago/ DownloadUploadAFileWebVI	6	171

Table 4.1. Overview of 10 LabVIEW Repositories.

4.4 Tool Integration to Support Change Analysis of Simulink Models

The Change Analyzer has been extended to support the indexing of Simulink models by adding a model parser to parse Simulink files. The model parser transforms Simulink models into an appropriate EMF resource via the Massif tool. When changes to a Simulink file are detected by the Change Analyzer, the affected Simulink file is sent to the model parser. The model parser connects to the Massif tool, and the Massif tool then connects to the MATLAB/Simulink tool to

request information about the Simulink file. The MATLAB/Simulink tool sends appropriate characteristics about the model to the Massif tool, which then constructs an appropriate EMF-based Simulink model based on the information sent by the MATLAB/Simulink tool. The EMF-based Simulink model is then sent to the model parser and processed by the Change Analyzer. The Massif tool also provides an EMF-based metamodel that captures essential properties of the Simulink model. This metamodel is used by the Change Analyzer to validate input of EMF-based Simulink models and facilitate efficient query of the Simulink models that have been indexed by the Change Analyzer. Figure 4.8 provides an overview of how the extended Change Analyzer supports Simulink models. The Change Manager component in the figure captures the change-based model updater and change-based query engine discussed in Section 3.2.

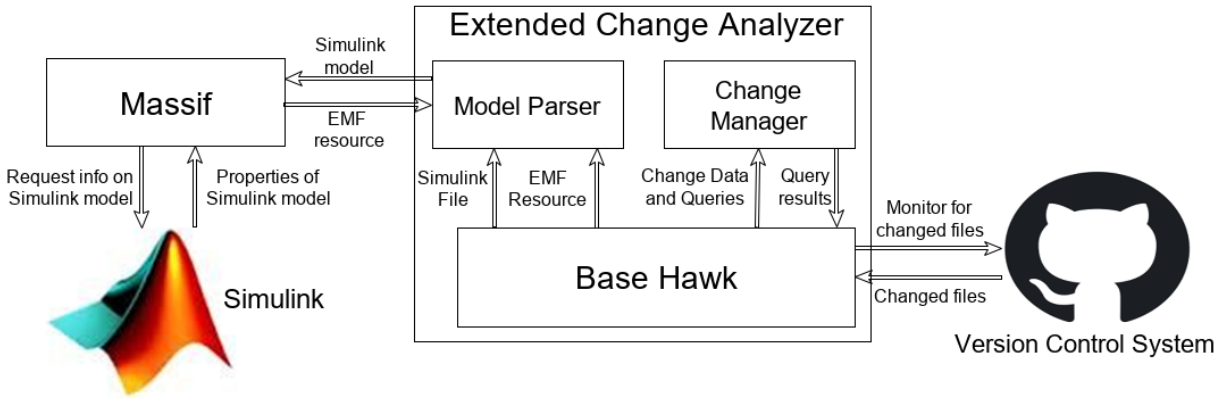

Figure 4.8. Extended Change Analyzer to Support Simulink Models

This approach to support Simulink models via the transformation of Simulink models to EMF-based models has some limitations. For example, there is no guarantee that an EMF-based model generated by Massif will be a faithful representation of the original Simulink model. Furthermore, there may also be some synchronization issues between the generated EMF models and the original Simulink models, such as when a Simulink model is updated to a newer version of MATLAB/Simulink IDE. To minimize the impact of these limitations, we have manually

verified the results of the analysis on the original Simulink models. In the future, we plan to provide native support for Simulink models in the Change Analyzer instead of transforming the models to EMF-based representations.

4.5 Related Work

This section discusses the research work related to tool integration and metamodels in MBSE.

4.4.1 Tool Integration in MBSE

MBSE inherently involves interaction with different heterogeneous components, subsystems and programs. Furthermore, most phases in the systems engineering process often use distinct tools; hence, there is a need to ensure integration of tools and provide support for interactions with multiple programs [Bajaj 2011]. However, tool integration is a challenging task due to the mutual dependencies that exists across components, the need to ensure that the functional flow of the system is preserved, the complex interfaces of associated systems, the heterogeneous nature of the components with custom architecture, and the different formats of data that need to be exported or imported [Gough 2018, Karsai 2000, Montgomery 2013].

Much work has been done to address the challenges associated with tool integration in MBSE. Qamar et al. [Qamar et al. 2009] developed an approach for designing mechatronic systems via the integration of SysML and Simulink models. Mechatronic systems integrate mechanics, electronics and software to provide a common functionality; hence, the design of such systems is quite challenging due to the overlapping objectives of each of the three domains. A mapping of elements in SysML and Simulink models was created, and a transformation language was used to convert from one model format to the other.

Shi et al. [Shi 2007] developed a system that maps the structural and behavioural components of a UML model to the corresponding components in a Simulink model. Hooman et

al. [Hooman 2004] developed a coupling tool that acts as a bridge between a UML-based tool and Simulink by converting the continuous time representations in Simulink to discrete event timings in the UML-based tool via the execution time of the transitions that occur in the UML-based tool. Reichmann et al. [Reichmann 2004] also created a UML-based system that contains plugins for connecting to Simulink. Both the initial UML model and the Simulink model are converted to the same executable format.

Finally, Hovarth and Starr [Hovarth 2015] developed Massif, a tool that integrates Eclipse-based EMF tools and Simulink via the MATLAB API. The tool supports bi-directional transformation from EMF-based model to Simulink models, as well as user-based configurations to guide the transformation. The research presented in this dissertation integrates the Massif tool with the Change Analyzer discussed in Chapter 3, to support the change analysis of Simulink models in open-source repositories.

Most of the related works introduced have focused on tool integration across MBSE tools, while others focused on how to convert from UML, which is widely used by MDE-based tools and other model formats supported by MBSE tools. Only Massif provides support for integration between an MDE-based tool and MBSE tool. This chapter discusses the integration between an MDE-based tool (Change Analyzer) and two MBSE-based tools (LabVIEW and Simulink).

4.4.2 Metamodels in MBSE

Javed et al. [Javed 2008] developed an approach for recovering the metamodel of a domain-specific language from a set of domain models via grammar inference, which specifies techniques for recovering a grammar from a set of programs. Hence, Javed et al. developed a mapping that translates domain models into textual representations, generates the grammar, and

then maps the output back to a metamodel format. De la Vara et al. [De la vara 2013] developed a metamodel to facilitate compliance with diverse safety standards. The metamodel abstracts notations and concepts that can be used to describe data that shows compliance to a specific safety standard. Beydoun et al. [Beydoun 2009] also developed a metamodel to support modelling languages that are used to develop agent-based software. The metamodel was constructed by extracting frequently occurring notations in the modelling of agent-based software and mapping out the relationship among these notations. The Massif Project also developed a metamodel for Simulink models [Horvath 2015]. The Simulink metamodel contains a root model element named SimulinkModel that maintains a reference to the original Simulink file. The Simulink metamodel makes it possible to transform Simulink models into EMF-compatible resources and provide native support for Simulink models in existing EMF-compatible tools. However, we could not find any similar open-source metamodel for LabVIEW despite the extensive use of LabVIEW in developing complex systems in industrial practice. Hence, it is not surprising that there has been limited research on the analysis of developed LabVIEW models.

4.5 Chapter Summary

This chapter has discussed how we extended the Change Analyzer developed in the previous chapter, to support LabVIEW and Simulink models. The chapter also discussed the construction of a LabVIEW metamodel to support the automated analysis of LabVIEW models. The next three chapters discuss how we used the work described in this chapter to support change analysis across 81 LabVIEW models in 10 GitHub repositories and 575 Simulink models in 31 GitHub repositories.

CHAPTER 5

EVOLUTION OF BAD SMELLS IN LABVIEW GRAPHICAL MODELS

This chapter discusses the impact of changes on the evolution of bad smells in LabVIEW models. Seven queries were constructed with the Change Analyzer to detect instances of seven bad smells in LabVIEW. The results from the execution of these queries were analyzed to understand the evolution of bad smells in 81 LabVIEW models across 10 repositories. The analysis of the bad smells was aimed at understanding the prevalence and introduction of smells as well as the relationship between bad smells and the structural changes made to the models. This chapter summarizes the need for better analysis of design smells in systems models and suggests an approach that may assist in improving the structure and quality of systems models developed in LabVIEW.

5.1 Chapter Introduction

The design of software systems may contain flaws that negatively affect quality and maintainability. These flaws (known as “Bad Smells”) are generally considered undesirable in the practice of software engineering because they tend to reduce the overall quality of the software, increase the complexity of future refactoring activities, decrease the understanding of the software code, and reduce the reusability features of system components [Fowler 1999]. Bad smells are not bugs (i.e., they do not prevent the program from functioning correctly), but suggest potential weaknesses in design that may increase the risk of bugs in the future. Studies

have shown that code with many bad smells takes 32% longer to debug and causes more frustration to programmers [Zaman 2012]. Therefore, bad smells represent a major challenge to the quality and maintenance of large and complex software systems.

Bad smells may be introduced at the initial version of a system, and propagated through subsequent versions, while new smells may also be introduced at any version of the system [Olbrich 2009]. Therefore, as a system ages, the instances of bad smells embedded in the system continue to evolve and may lead to larger and more complex architectural problems. Past research resolved bad smells via data extracted from public software repositories in textual programming languages [Chatzigeorgiou 2014, Tahmid 2016]. This body of research is very important in understanding and analysing the evolution of bad smells in software. However, the majority of data stored in software repositories and the approaches developed to manipulate them have targeted the source code of text-based languages. There is a considerably less amount of software engineering research on graphical languages, in general, related to bad smells in system evolution.

In this chapter, we used the Change Analyzer discussed in Chapters 3 and 4 to mine LabVIEW models in software repositories, and developed user-defined queries to detect and analyse the presence of seven of the design smells identified by end users in [Chambers 2013] and [Zhao 2019]. The queries were used to extract and analyse the evolution of bad smells that were present in the version history of 81 models across 10 GitHub repositories. The analysis of the query results show that all repositories contain at least one type of smell and most smells are often introduced in the initial version of the repositories. Furthermore, the number of smell instances in these models tends to increase steadily for a period and then decrease even though

the size of the models continues to increase throughout the system's life cycle. The contributions of this chapter include:

1. We propose a semi-automated approach to detect bad smells in graphical languages via user-defined queries.
2. We investigate the presence of seven bad smells in 10 LabVIEW GitHub projects.
3. We also present the evolutionary analysis of the persistence of these bad smells across the life cycle of the LabVIEW models in these projects.

5.2 Overview of Bad Smells in Systems Models and Text- Based Programs

The concept of bad smells evolved from the research on design patterns. Gamma et al. [Gamma 1995] categorically presented a catalog of succinct solutions to commonly occurring design problems. Their 23 design patterns assist software developers in creating more flexible, elegant, and reusable designs without having to rediscover the design solutions. Opdyke formalized refactoring to support the design, evolution and reuse of object-oriented application frameworks in his Ph.D. dissertation [Opdyke 1992]. The refactorings are defined to be behavior preserving and remove bad smells in Object-Oriented Programming (OOP), provided that their preconditions are met. Although bad smells have been studied thoroughly in the context of text-based programming environments over the past two decades, the examination of bad smells in graphical models is particularly limited.

Bad smells in systems models (also known as "model smells") indicate bad designs that usually correspond to a deeper problem in a systems model. Bad smells in systems models are related to bad smells in OOP, but there also exist significant differences between bad smells in OOP and bad smells in systems models. OOP has a correspondence with textual programming

languages (such as Java, C++, and Python), but systems models largely adopt graphical representations (such as Simulink and LabVIEW models). In textual languages, source code often defines the executable order of a computation; in graphical languages, a communication medium (such as wires in LabVIEW and Simulink systems models) for passing data between different blocks is always required. This distinction leads to bad smell summarization differences. For example, *Unorganized Wires* is a bad smell reported by LabVIEW end-users [Zhao 2019]. It refers to the use of unorganized wires to connect different parts of models, thus making the model hard to read and understand. However, no similar bad smells are discussed in existing literature in the context of OOP. Bad smells in OOP and systems models also share some commonalities. For example, *Long Parameter List* is a bad smell mentioned in existing literature that occurs when a method includes too many formal parameters. A similar bad smell is found in LabVIEW systems models [Chambers 2013]. A deeper analysis of bad smells in systems models has the potential to provide engineers with more insight on how to improve the maintainability and reliability of systems models.

5.3 Research Questions to Understand the Evolution of Bad Smells

We analysed the evolution of bad smells in LabVIEW models. Our aim was to discover the prevalence of smells in LabVIEW, at what point in time the smells are introduced, and the code changes that introduce, increase, or decrease the smells in the models. Specifically, our study answers the following three research questions.

RQ1. *How prevalent are bad smells in LabVIEW models?* Chambers and Scaffidi [Chambers 2013] conducted a study to show that LabVIEW developers tend to prioritize the correctness of the software to be developed over other properties that may affect the maintainability of the software over a period of time. This suggests that LabVIEW

models may be a ripe source for bad smells. However, we do not know of any study that validates this assumption. Our study aims to discover the prevalence of several selected bad smells in LabVIEW models stored in open repositories.

RQ2 *When are bad smells introduced in LabVIEW models?* Fowler [Fowler 1999] proposes that bad smells in software programs are often introduced due to the evolution and maintenance activities that are performed throughout the program's life cycle. This may indicate that bad smells are often introduced at later stages of the software development cycle. However, Tufano et al. [Tufano 2015] conducted a study of 5 types of smells in over 200 open source Android, Apache and Eclipse projects. Their results show that most smells are present in the first version of a program. Therefore, the results from both Fowler [Fowler 1999] and Tufano et al. [Tufano 2015] suggest that bad smells may creep into software at various stages of development. We do not know of a similar study that has been conducted to investigate when smells are introduced in LabVIEW models, or graphical models, in general. Our study aims to discover the common trends of how bad smells are first introduced in LabVIEW repositories.

RQ3 *What is the relationship between the bad smells and structural changes made to a model?* This phase of the research work aims to unravel the set of changes to a model that led to the introduction of new smells in the model and the set of changes that led to an increase, reduction or removal of bad smells in the model. This is a first step to generate a set of best practices for writing programs with minimal smells, as well as the anti-patterns that should be avoided due to the likelihood of introducing new smells to a model.

To answer the research questions mentioned above, we implemented the following approaches.

1. We selected 10 repositories from GitHub. These repositories were selected by searching for the keyword "labview nxg" in GitHub and the repositories were checked to contain at least one LabVIEW NXG file. We were able to gather 10 repositories.
2. We constructed seven queries in the Change Analyzer to detect instances of seven smells in LabVIEW. The execution of the queries in the Change Analyzer will produce the evolution history of smells in selected models.
3. We manually analysed the results of the queries to answer the research questions discussed above.

5.4 Smell Selection and Queries to Detect Smell Instances

Many smells in LabVIEW have been identified in previous works. Chambers and Scaffidi [Chambers 2013] identified smells based on an interview conducted with experienced LabVIEW developers while Carcao et al. [Carçao 2014a, Carçao 2014b] identified more smells based on the rate of energy consumed when LabVIEW programs are executed. However, these smells are more related to performance issues and do not cover a wider range of developer's experience (e.g., a new developer may find some tasks challenging and it may be trivial to an expert). Due to the number of smells that can be detected, it is practically infeasible for us to analyse all possible smells that can be detected in LabVIEW models. In this paper, we analysed selected smells that have been extracted from posts in LabVIEW discussion forums [Zhao 2019]. Due to the open-nature of these online forums, various issues are discussed by developers with a wide-range of expertise; hence, we believe that the smells captured in this way are representative both in terms of the needs of LabVIEW developers and the category of programs involved. Moreover, statistical results show that end-users are deeply interested in reviewing and replying to posts

related to LabVIEW model smells. From 2000 to 2019, the average review and reply per post are 1197.39 and 5.94, respectively; while the average review and reply related to bad smell posts are 1356.02 and 15.55, respectively.

Zhao and Gray [Zhao 2019] identified 15 bad smells, while Chambers and Scaffidi [Chambers 2013] identified 13 smells. Six of these smells were reported in both papers; thereby, resulting in a net total of 22 smells. We selected the seven most prominent smells for this study. The seven smells studied in this chapter are: 1) *Large Variables*, 2) *No Wait in a Loop*, 3) *Build Array in a Loop*, 4) *Excessive Property Nodes*, 5) *String Concatenation in Loop* 6) *Multiple Nested Loops*, and 7) *Deeply Nested Subsystem Hierarchy*. These smells are selected for the following reasons:

1. **Ease of validation**. The selected smells have well-defined criteria for affirming the presence of the specific smell in the set of models. For example, the smell *No Wait in a Loop* means that the absence of a *Wait* element in any loop is considered a smell. Hence, we only need to check whether any of the loops in a set of models does not contain the *Wait* element in order to affirm the presence of the smell in the models.
2. **Scope of inclusion**. The selected smells cover two major perspectives of a VI: performance and structure. Performance relates to the model execution, such as the time a model needs to run, and the memory a model consumes while executing. *No Wait in a Loop* is likely to cause synchronization issues [Chambers 2013] while the *Build Array in a Loop* smell and the *String Concatenation in a Loop* smell slows the program's performance and causes memory issues [Chambers 2013]. *Multiple Nested Loops* occurs when loop structures are embedded in another loop structure. This increases the model's structural complexity, reduces readability, and may lead to other control and performance

issues. *Large Variables* refers to the use of many variables in a model, *Excessive Property Nodes* refers to the usage of *Property Nodes* element in a model, and *Deeply Nested Subsystem Hierarchy* refers to when a VI contains too many levels of subVIs. These three smells increase the structural complexity of a model, thus affecting the model understandability.

3. **Level of granularity**. The selected smells represent different levels of granularity within a VI. *Large Variables* examines the usage of the most basic data unit that a VI uses. Four smells: *No Wait in a Loop, Build Array in a Loop*, *Excessive Property Nodes*, and *String Concatenation in Loop*, focus on the investigation of smells related to a single node/structure in a systems model. *Multiple Nested Loops* explores multiple nodes/structures within a VI that may affect system models and *Deeply Nested Subsystem Hierarchy* analyses bad smells from a structural aspect. The inspection of these smells at different granularity levels helps us to better understand bad smells in systems models and observe how different smells evolve over time.

The following paragraphs provide a detailed description of each of the seven smells in our study.

1. **Large Variables**. This refers to the adoption of too many local variables in a model. In LabVIEW, a local variable is used to communicate between structures within one module. It is similar to formal parameters of a method in OOP. Overusing local variables, such as using them to avoid long wires across a block diagram or using them instead of data flow, can lead to several maintenance issues. Local variables make copies of data buffers. If users adopt too many local variables to transfer large amounts of data from one place on the block diagram to another, more memory is consumed and may result in slower execution.

2. **No Wait in a Loop**. This bad smell often occurs during a data acquisition program. In a general object-oriented program, when a *For Loop* or a *While Loop* finishes executing one iteration, it may immediately begin running the next. The frequency of one iteration is usually not considered. In LabVIEW modelling, however, it is often beneficial to control how often a loop executes. For example, in industrial practice, it is important to mark the data acquisition rate. If users want to acquire data in a loop, they would need a method to control the frequency of the data acquisition. In this situation, a pause in a loop is necessary. Timing a loop also allows the processor time to complete other tasks such as updating and responding to the user interface.

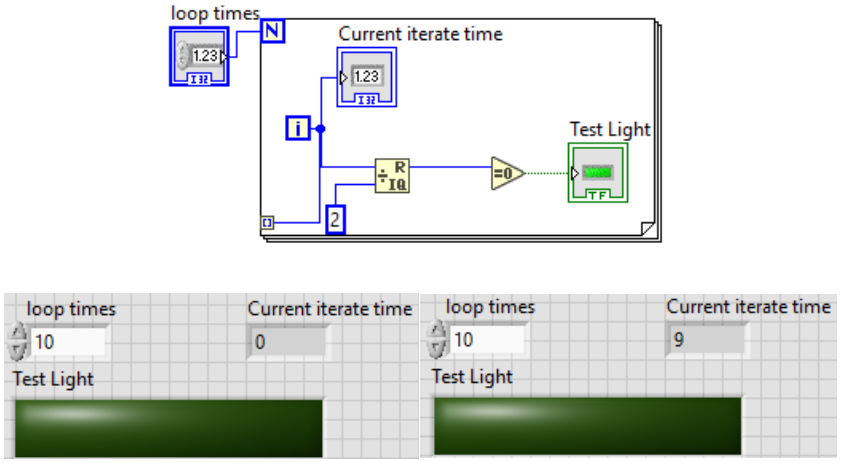

Figure 5.1. A Sample Model with No Wait in a Loop Smell. Top Figure Shows the Model. Bottom Figures Show the Initial State and Ending State.

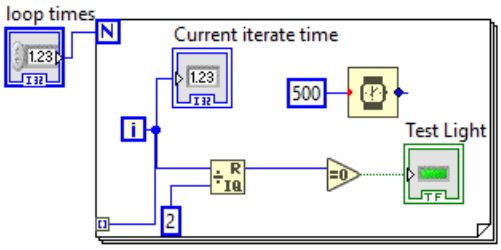

Figure 5.2. A Sample Model with the Wait node. Integer 500 means the Loop Pauses 500ms every iteration

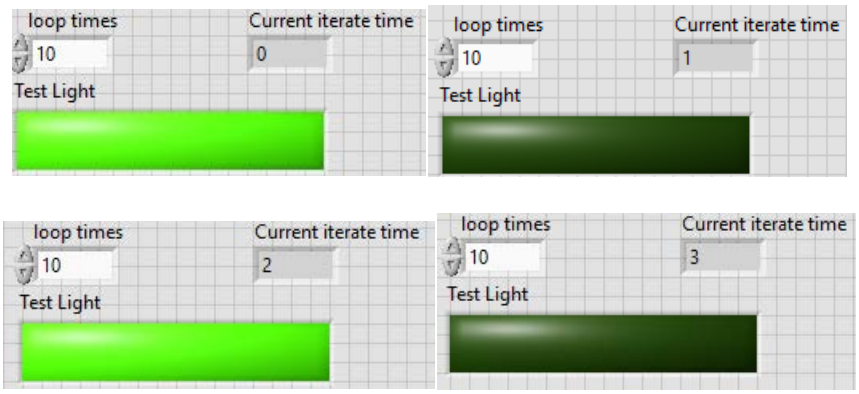

Figure 5.3. LED Light Changes Become Visible to Users after Adding a Wait Node.

Figure 5.1 is a sample LabVIEW model that illustrates how "No Wait in a Loop" could cause problems. The model specifies that an LED light will turn green whenever the loop counter is even. However, the LED light does not change its status during the model's execution, because without a Wait node, the model runs the Loop in such a short time that the users cannot observe any changes in the Front Panel. Figure 5.2 shows a Wait node that is added to the model. After adding this Wait node, users can observe the LED light changes during every iteration, as shown in Figure 5.3.

3. **Build Array in a Loop.** This bad smell refers to the construction of an array inside a loop structure. When an array node is inside a loop, every time the loop starts a new iteration, a new copy of the array is constructed. This process leads to increased memory consumption and may cause severe performance issues.
4. **Excessive Property Nodes.** The purpose of *Property Nodes* in a LabVIEW model is to programmatically control the properties of a front-end or user interface object, such as color, visibility, position, numeric and display format. For example, users could change the color of a dial to go through blue, green, and red as its numerical value increases. However, the excessive usage of *Property Nodes* can introduce several issues in LabVIEW models. One of the issues is that there is a lot of overhead with *Property*

Nodes. Each *Property Node* access will result in a context switch to the UI thread, which will slow model execution.

5. **String Concatenation in a Loop.** This bad smell, as the name suggests, refers to the adoption of *String Concatenation* structure inside the body of a loop. Chambers and Scaffidi [Chambers 2013] first identified this bad smell from their interview with LabVIEW experts. They affirm that the implementation of *String Concatenation* structure in the body of a loop causes slow performance and memory issues.
6. **Multiple Nested Loops.** This bad smell refers to placing loop structures inside the body of another loop structure. *Multiple Nested Loops* are considered as a negative programming practice because it's execution is time consuming and it also increases the program's complexity. Many works have been conducted to solve deeply nested problems [Karunaratne 2018, Quilleré 2000]. Similar to other programming languages, the use of nested loops in LabVIEW models reduces readability [Chambers 2013] and sometimes can be problematic when introducing other structures in the outer loop. Due to the nature that LabVIEW is a dataflow-driven programming paradigm, the introduction of other structures inside of a loop may contribute to control and design issues. Many discussions related to these issues are found in the LabVIEW discussion forum (such as nested loop control[14], design issue[15]).
7. **Deeply Nested Subsystem Hierarchy.** Modularity is essential for software systems and is supported by decoupling software into reusable units [Exman 2014]. In LabVIEW, modularity is achieved by the adoption of a SubVI. A SubVI is the same as a VI and it

[14] https://forums.ni.com/t5/LabVIEW/Nested-while-loops/tdp/2167378?profile.language=en
[15] https://forums.ni.com/t5/LabVIEW/Nested-FOR-loops-and-flat-sequence-Agood-design/td-p/2454042?profile.language=en

also contains a front panel and a block diagram, but a SubVI is called within a VI. The relationship between a VI calling a SubVI is similar to a public method in a class calling another method in a separate class. In LabVIEW, users can define their SubVIs and customize an icon for each SubVI. The icon is equivalent to the SubVI in the block diagram. *Deeply Nested Subsystem Hierarchy* suggests that a VI may contain too many levels of SubVIs. For example, myModel.vi includes a myModelSub.vi; myModelSub.vi includes a myModelSubSub.vi. A model with too many hierarchies may increase the difficulty in understanding the model because sub-levels hide the logic and implementation details from top levels. This bad smell is also identified and analysed in Simulink model smells [Gerlitz 2015].

We simplify the amount of effort required to translate a bad smell into an efficient query method via the carefully designed metamodel discussed in Section 4.3. The metamodel supports OOP concepts such as inheritance, thereby making it possible to group related elements. The selected bad smells focus on structures and certain characteristics that have been known to suggest a bad design. The constructed metamodel captures these structures and characteristics, thereby simplifying the query needed to identify these properties in a LabVIEW model. It should be noted that the metamodel can be used to simplify the query for detecting any arbitrary bad smells that focus on the structural properties of a LabVIEW model. The query to identify smells has to be manually constructed, but the detection and historical evolution of smells based on the queries has been automated. A sample query was developed for each of the bad smells to detect the presence of bad smells in the repositories.

To detect instances of *Large Variables*, we extracted the total number of variables in each model. The *No Wait in a Loop* bad smell was extracted by examining all of the loops (Both

While Loop and *For Loop*) and recursively checking for the presence of a *Wait* element in any of the loops. The *Build Array in a Loop* and the *String Concatenation in a Loop* bad smells were identified in a similar way to the *No Wait in a Loop* smell, except that the *Build Array* and *String Concatenation* elements were respectively the object of interest in the loops. The *Excessive Property Nodes* bad smell was extracted via the *Property Node* type defined in the metamodel. *Multiple Nested Loops* was extracted by searching for all the loop structures that have another loop structure embedded within them. Finally, instances of the *Deeply Nested Subsystem Hierarchy* smell was detected by first extracting all the subVI's in a Model, and then searching through the extracted subVIs for any one that has at least one subVI. In LabVIEW, a subVI is usually a complete VI in another file that is called by its parent VI. Listing 5.1 is a sample query to detect instances of the No Wait in a Loop smell.

The correctness of the queries has been manually verified by randomly selecting four models that return a positive result for a smell and four models where the execution of a query did not detect instances of the smell. This process was repeated for all of the seven queries. The only exception was the *Build Array in a Loop* smell where the execution of the query detected instances of the smell in only two models. Therefore, only two models were used to verify the positive result. The results of the verification process shows that the queries were 100% correct. Appendix A contains all of the seven queries developed to detect instances of the bad smells.

```
var i=0;
var j=0;
for (t in WhileLoop.allInstances){
    if (t.Wait.size()>0){
        continue;
    }
     else{
         i= i+getRecursiveWait(t);
     }
}
return i;

// recursive function to check for Wait node
operation getRecursiveWait(element:Any): Integer{
    // check if the element as any children (containment references)
    if (Model.getChildren(self).size()<1){
        return 1;
    }
    //check if the element has a Wait
    else if (element.Wait.size()>0){
        return 0;
     }
     else{
      // recursively go through element's children and see if it has a Wait
        for (j in Model.getChildren(self)){
            if(getRecursiveWait(j)==0){
                return 0;
            }
        }

         //return 1 if none of the children has a Wait
        return  1;
    }
}
```

Listing 5.1. Query to Detect Instances of No Wait in a Loop Smell

5.5 Research Results and Analysis

We developed three research questions to study the evolution of LabVIEW models. The research questions deal with the introduction and prevalence of bad smells in LabVIEW repositories, as well as the structural changes to models that are likely to introduce bad smells. The research questions, which were discussed in more detail in Section 5.3, are as follows:

RQ1 *How prevalent are bad smells in a LabVIEW model?*

RQ2 *When are bad smells introduced in a LabVIEW model?*

RQ3 *What is the relationship between the bad smells and structural changes made to the models?*

To answer these research questions, a preliminary investigation and analysis of the selected seven bad smells was performed on 81 LabVIEW models stored in 10 GitHub repositories of varying size and complexity. Section 4.3.3 provides an overview of the 10 GitHub repositories. The following subsections address each of the research questions and provides an in-depth study of the evolution of bad smells in LabVIEW models.

5.5.1. Prevalence of Bad Smells in LabVIEW Models

To answer the question related to the prevalence of bad smells in LabVIEW models, we queried the repository to check for the presence of each of the seven selected bad smells in the identified repositories. Table 5.1 shows that most of the repositories contain instances of two or three kinds of smells across the life cycle of the models embedded in the repositories. Furthermore, the size and complexity of the models embedded in the repositories does not seem to affect the possibility of the models containing a bad smell. This may also validate the hypothesis that the engineers are not very familiar with software engineering design principles because bad smells are often associated with bad software engineering practices [Van Emden 2002]. Table 5.1 gives a breakdown of the presence of each of the bad smells in the life cycle of the models stored in the repositories.

Repo \ Smells	Large Variables	No Wait	Build Array	Property Node	String Concatenation	Nested Loop	Nested Subsystem
Repo 1	yes	yes	no	no	no	yes	yes
Repo 2	yes	yes	no	yes	no	yes	yes
Repo 3	no	no	no	yes	no	no	no
Repo 4	yes	yes	no	yes	no	no	no
Repo 5	yes	yes	no	yes	no	yes	no
Repo 6	no	yes	yes	no	yes	no	no
Repo 7	yes	yes	no	yes	no	yes	no
Repo 8	yes	yes	no	yes	no	no	no
Repo 9	yes	yes	no	no	no	no	no
Repo 10	yes	yes	no	yes	no	no	no

Table 5.1. Detection of each Smell across 10 Repositories.

5.5.2. When are Bad Smells Introduced?

The introduction and consequent increase/decrease of instances of a bad smell is very important to understand the impact of maintenance activities in the life cycle of LabVIEW models. To answer this research question, we first checked for when instances of each smell were first detected in each repository. The results show that 55% of the smells were introduced in the first version of the repositories, 17% of the smells were introduced in the second version, and the remaining 27% were introduced in other versions. This result shows that bad smells are introduced at various stages of the software development process, but most of the smells were introduced at the initial version. This suggests that software maintenance activities are not solely responsible for the introduction of bad smells. Figure 5.4 summarizes the results corresponding to when a bad smell is introduced.

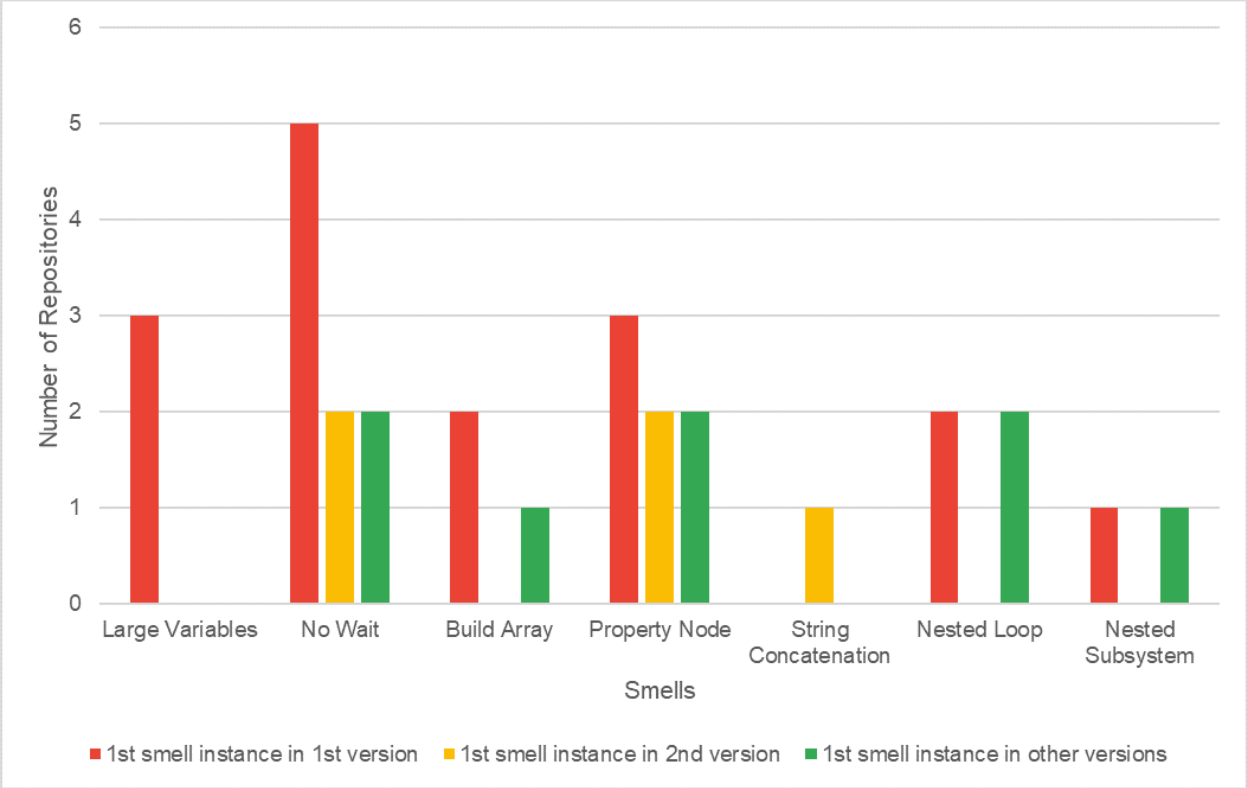

Figure 5.4. Smell Introduction across Repositories

The number of bad smell instances also tends to increase steadily over time and then decline after a peak period. Figures 5.5, 5.6, 5.7, 5.8 show the evolution of four types of smells across the version history of the repositories. It should be noted that the smells *String Concatenation in a Loop*, *Deeply Nested Subsystem Hierarchy*, and *Build Array in a Loop* have been omitted from the figures. This is because only trivial changes were observed in the number of smell instances across the version history of models in all the repositories. Furthermore, some of the repositories have less than seven versions; hence, there may be a constant value from the last version number to the "current version" specified in the figures.

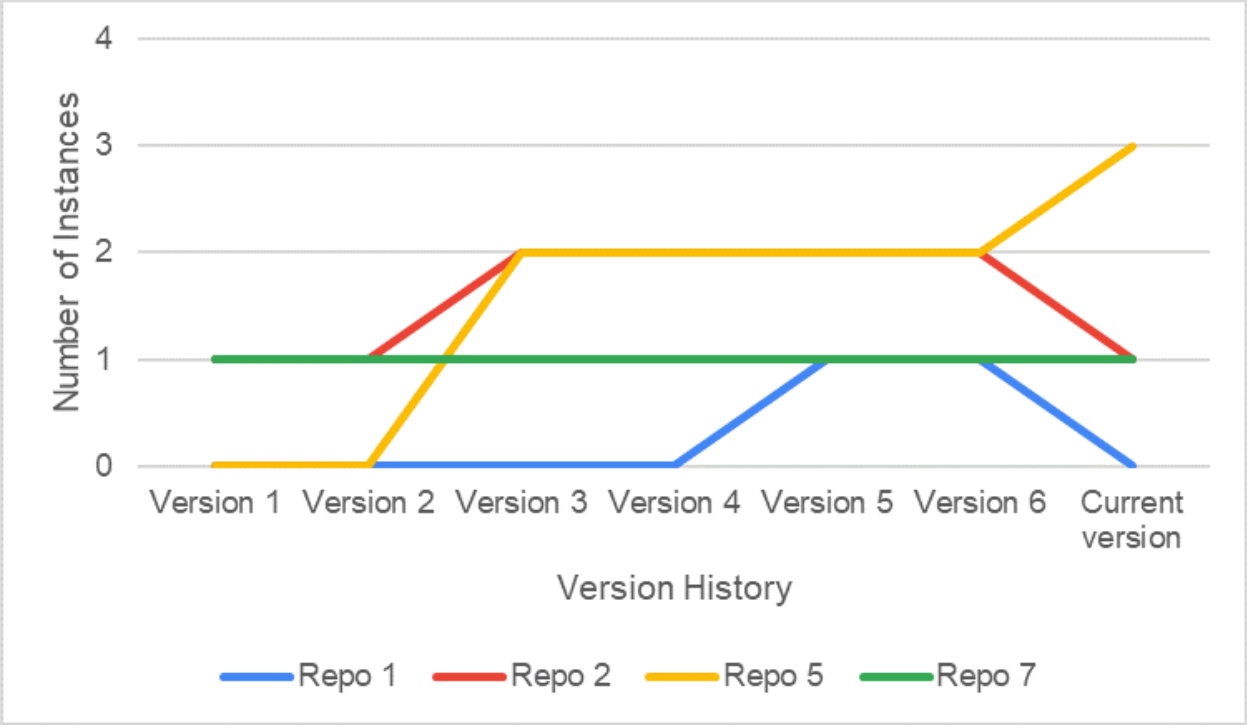

Figure 5.5. Evolution of Multiple Nested Loop across Version History of LabVIEW Models

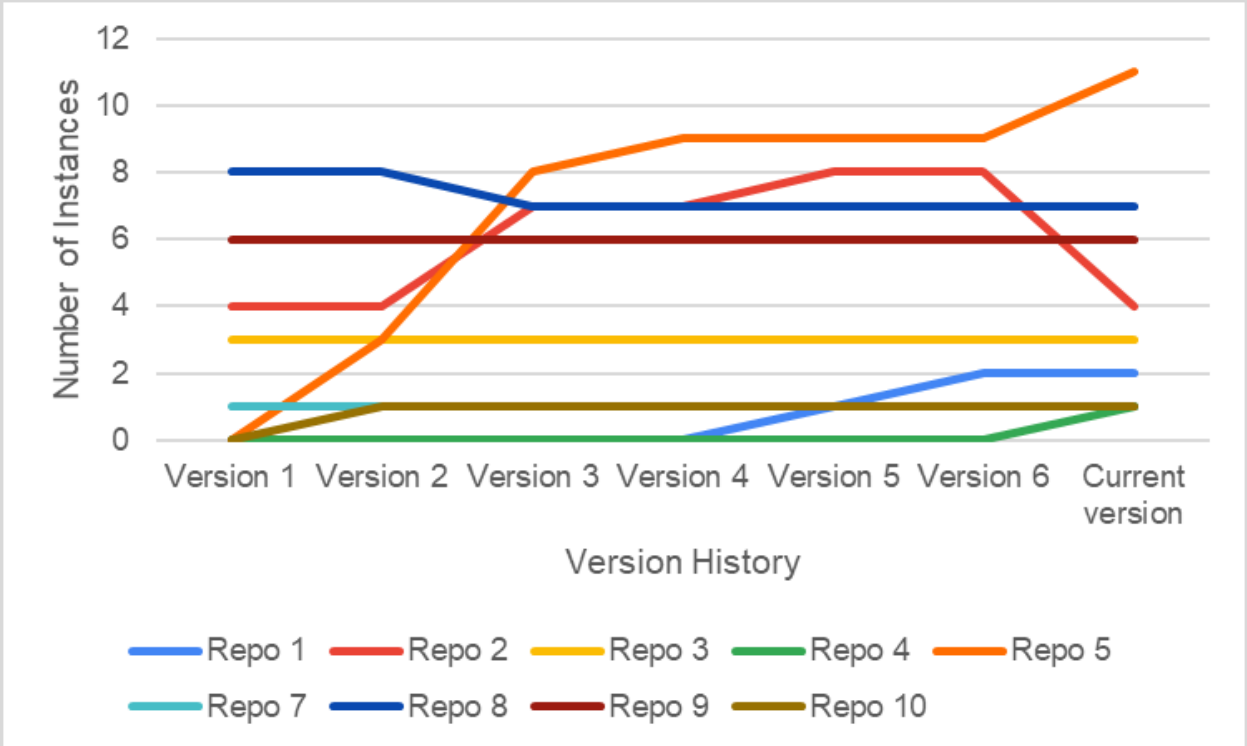

Figure 5.6. Evolution of No Wait in a Loop across Version History of LabVIEW Models

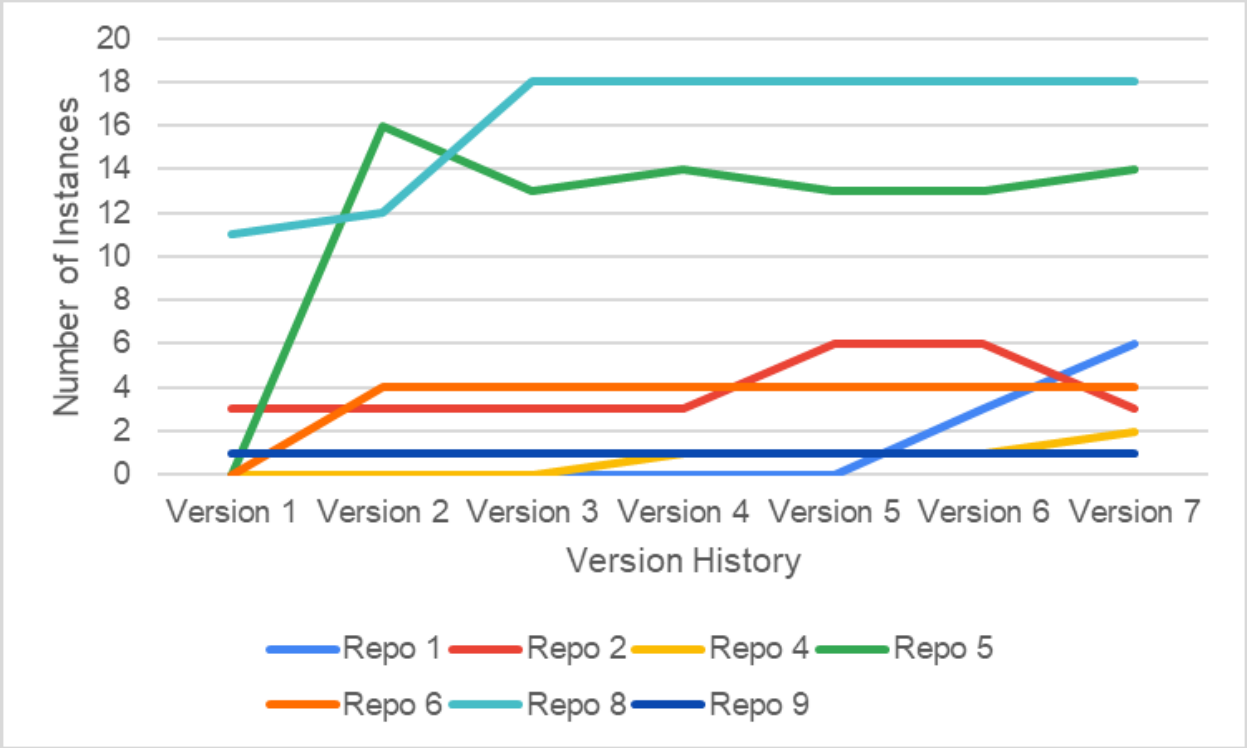

Figure 5.7. Evolution of Excessive Property Nodes across Version History of LabVIEW Models

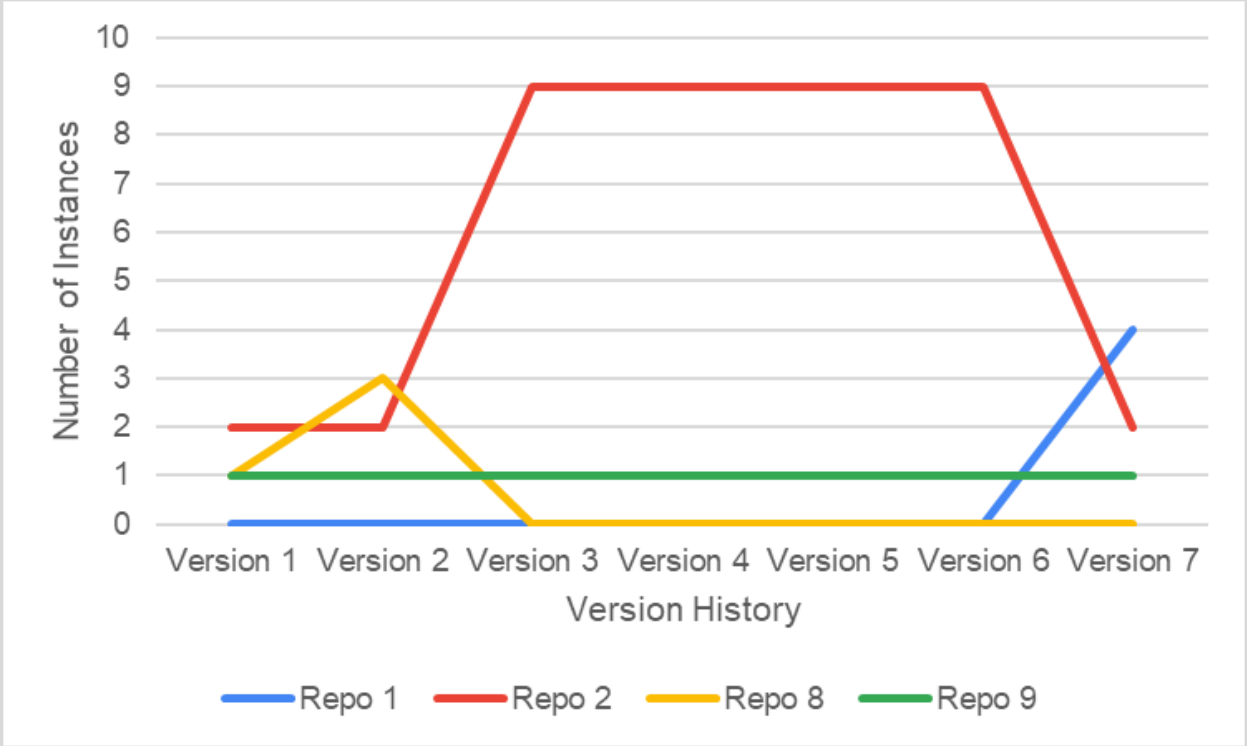

Figure 5.8. Evolution of Large Variables across Version History of LabVIEW Models

We also analysed the evolution of all the smells in each repository as the size of the models increased. While the full data is provided in the GitHub repository, we provide a sample analysis of Repository 2. Figure 5.9 gives an overview of the evolution of smells in Repository 2, while Figure 5.10 shows the increasing size of the models in the repository within the same period. However, it should be noted that while the overall size of the models increased throughout the life cycle of Repository 2, the number of model elements actually decreased in versions after the peak period.

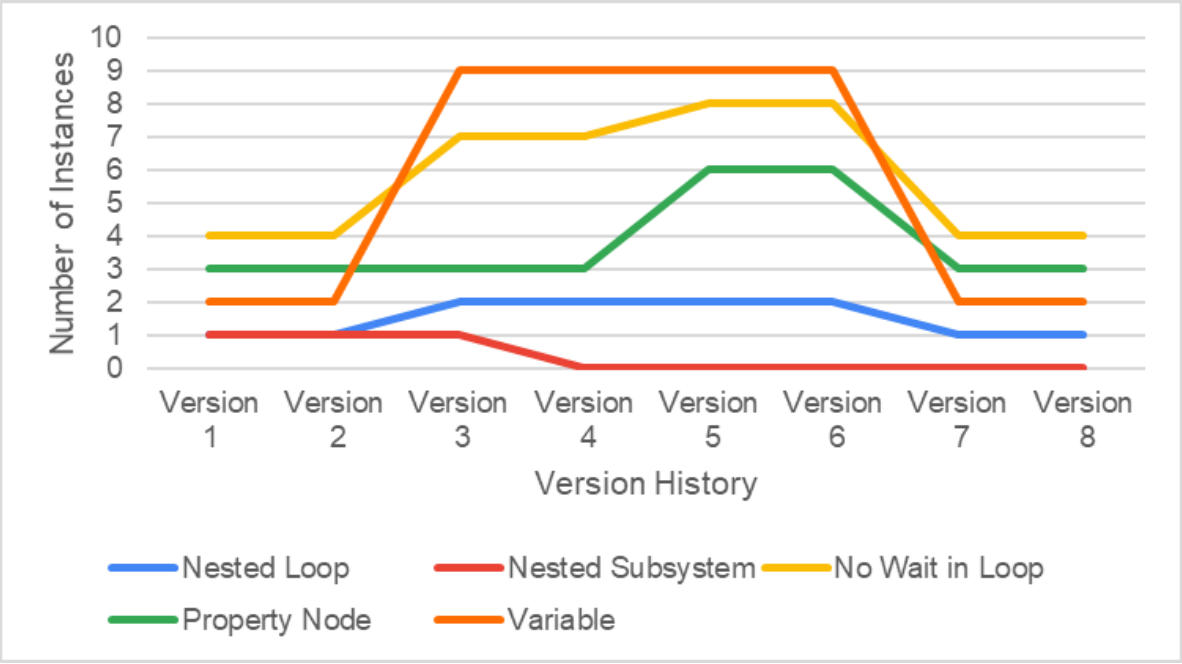

Figure 5.9. Evolution of Bad Smells in Repository 2

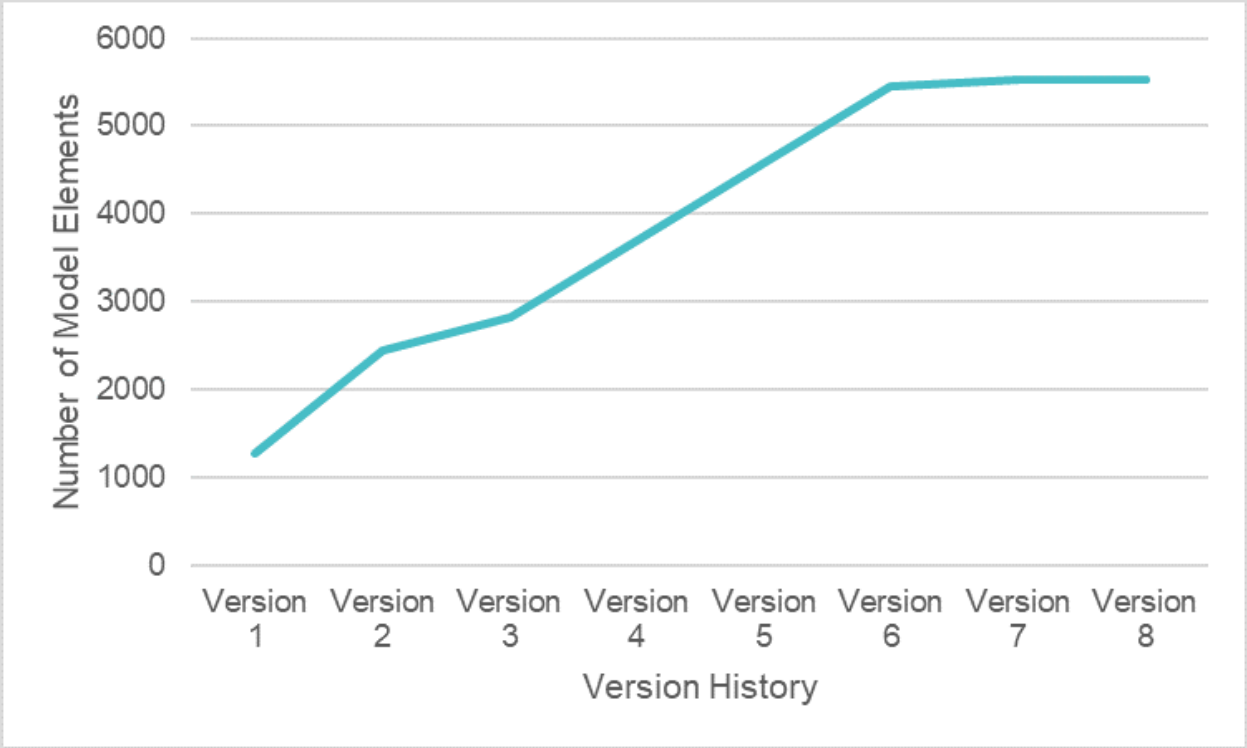

Figure 5.10. Evolution of the Model Size in Repository 2

5.5.3. Structural Changes Related to Bad Smells

The evolution of a model necessitates the continuous introduction of new changes to the model. These changes are needed for various reasons such as adapting the model to new requirements, fixing a bug, or making the model easier to understand. However, these changes may introduce new unintended smells within the model. This phase of the research identifies the set of structural changes that led to the introduction, increase or reduction of bad smells in the models.

The EMFCompare extension of the Change Analyzer has been used to analyse the differences in models across various versions. This analysis makes it easier to understand the changes that necessitate the evolution of LabVIEW programs. The EMFCompare extension supports four types of changes between two sets of models: *add* to indicate addition of new elements, *delete* to show removal of existing elements, *modify* to indicate the change of attribute

or reference values, and *move* to indicate elements that have been reordered. These structural changes to the models were then mapped to the changes in the bad smells exhibited by such models.

The results of the analysis of changes across the version history of LabVIEW models in the 10 repositories show that the majority of the changes involved addition of new elements. Table 5.2 shows the average number of modifications that are performed across each version in the repositories.

Repo \ Change Type	Avg ADD	Avg DELETE	Avg MODIFY	Avg MOVE
Repo 1	322	0	0	0
Repo 2	998	0	0	0
Repo 3	270	0	0	0
Repo 4	228.8	1.5	12.2	10.6
Repo 5	453.2	52.1	82.4	7.9
Repo 6	658	1	5	0
Repo 7	487	0	0	0
Repo 8	894	25	105	15
Repo 9	538	0	0	0
Repo 10	159	0	0	0

Table 5.2. Average Number of Change Types across Version History.

The addition of new elements to succeeding versions also corresponds to increasing instances of smells in the repository. Furthermore, versions with a lower number of smell instances are also associated with a higher number of deletions and attribute changes than number of additions. Figure 5.11 shows the trend of different kinds of changes in Repository 2. It should be noted that there was a sharp increase in the number of deletions and re-orderings (move) of elements. There

was also a decrease in the number of additions. This corresponds to the decrease in bad smells shown in Figures 5.5 to 5.9.

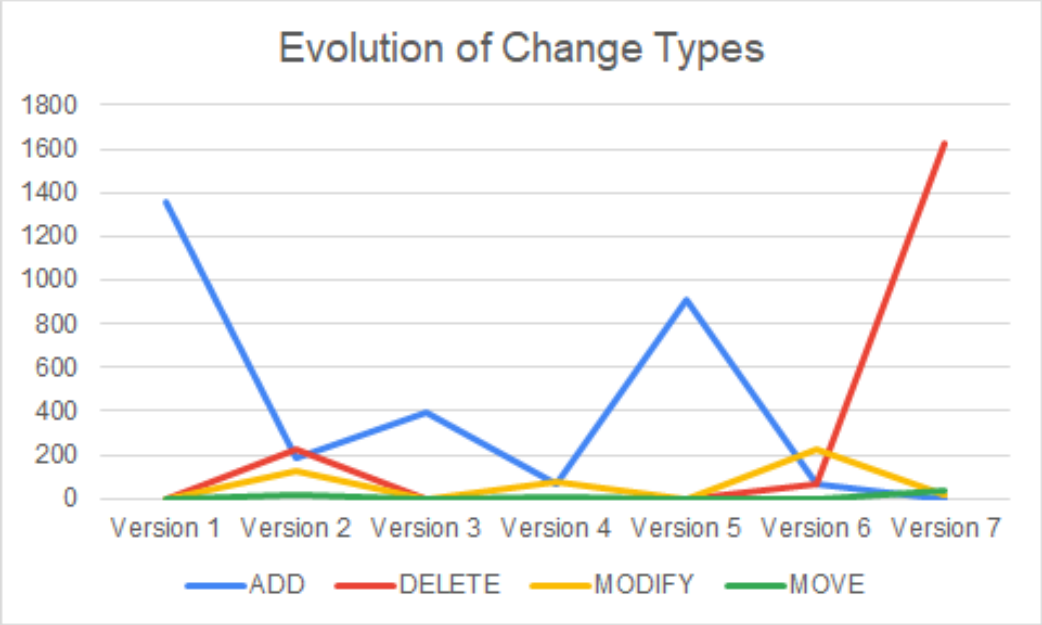

Figure 5.11. Evolution of Change Types in Repository 2

5.6 Threats to Validity

This chapter presents an analysis of bad smells in LabVIEW models. Although, a number of steps have been taken to ensure that the results presented are valid and generalizable, we have identified at least four main threats to the validity of the results presented in this chapter:

1. **Small sample size**. The sample size of the models used in this evaluation is rather small and thus the results may not be generalizable to all LabVIEW models. The small sample size is due to the relative young age of the LabVIEW NXG platform and the limited number of open-source repositories that contain LabVIEW NXG models.
2. **Author demographics**. We do not know the experience of the engineers who published the selected projects and how their experience affected the quality of the selected models. It is possible that more experienced engineers are able to produce models with higher

quality and fewer instances of bad smells. Unfortunately, information about the author demographics could not be extracted from the GitHub repositories used in this study.

3. **LabVIEW-specific smells and models**. The paper only considers LabVIEW models for the evaluation and some of the smells are specific to the LabVIEW environment. Furthermore, even though some of the smells are applicable to other systems modelling or data acquisition platforms, this paper only considers the effect of the smells on LabVIEW models. Hence, the results may not be generalizable for all systems modeling or data acquisition environments.
4. **Correctness of the queries**. The results presented in this paper depend on the correctness of the queries used to identify instances of bad smells in the LabVIEW models. Although we have ensured that the queries in this paper are correct via the manual verification process described at the end of Section 5.4, it may be possible that a new query generates some false negatives or false positives. We plan to address each of these concerns as part of future work. However, we believe that the results reported in this chapter provide significant insight into the general evolution of bad smells in systems models.

5.7 Related Work

Much work exists on bad smell identification in text-based languages. Chatzigeorgiou et al. [Chatzigeorgiou 2014] investigated the presence and persistence of bad smells in two open source projects. Their results show that bad smells are introduced at the initial version and often persist up to the current version of a project. Furthermore, the few eliminated smells often occur from a side effect of routine maintenance activities instead of a refactoring activity that specifically targets the removal of the smell. Olbrich et al. [Olbrich 2009] also conducted a similar study and concluded that code with smells are more prone to frequent changes than

projects that do not contain bad smells. Tahmid et al. [Tahmid 2016] also presented an approach for analysing bad smells in code repositories based on the relationships between different bad smells and the architecture of the source code. The approach simplifies the development of clusters of bad smells embedded in the code for easy refactoring. However, all of these approaches focus on text-based languages and none of them considered systems models developed via graphical languages.

Some work has been done to identify bad smells in graphical modelling languages. Simulink provides an interactive graphical environment for modeling, simulating, and analysing dynamic systems. It includes a comprehensive library of predefined blocks to be used to construct graphical models of systems using drag-and-drop mouse operations. Bad smells in Simulink systems models are identified by Gerlitz et al. [Gerlitz 2015]. In this work, the authors summarized 21 bad smells into 5 categories: name, partition, interface, signal flow and signal structure. Two tools, Artshopr [Kowalewski 2016] and SLRefactor [Tran 2013], were implemented to address these bad smells. Stephan and Cordy adopted near-miss cross-clone detection to find instances of antipatterns derived from the literature in public Simulink projects [Stephan 2015].

Bouhours et al. [Bouhours 2009] developed an approach for detecting bad smells in design patterns and design models. The collected a list of defective design patterns and provide a tool that can detect these defective patterns during a software design process. Sidhu [Sidhu 2018] provided a catalogue of bad smells for UML class diagrams, while Ghannem et al. [Ghannem 2011] used heuristic search techniques to detect instances of three bad smells in software models. Bonet et al. [Bonet 2018] conducted a study to understand how coding styles affects bad smells in model transformations. Their results show that declarative programming style tends to make

the model transformation less prone to bad smells. Bettini et al. [Bettini 2019] also developed an approach for detecting bad smells in metamodels based on a set of quality attributes that is specified by a modeler.

Chambers and Scaffidi [Chambers 2013] developed an approach for identifying bad smells in systems models via interviews with modeling experts. In our own earlier work, we developed an approach for identifying bad smells in LabVIEW models from an end-users' perspective by mining online forum posts [Zhao 2019]. However, both [Chambers 2013] and [Zhao 2019] do not investigate the presence of the identified smells in code repositories nor do they study the evolution of such smells. This chapter extends the literature by proposing techniques for identifying bad smells in graphical languages and analysing the evolution of such smells across the version history of the software systems developed.

5.8. Chapter Summary

This chapter has introduced a semi-automated approach for detecting and analysing bad smells in LabVIEW models via user-defined queries. The support for inheritance relationships in the LabVIEW metamodel was exploited to simplify the development of user queries to detect known bad smells from systems models. The approach has been evaluated on 81 models in 10 GitHub repositories. The evaluation process includes the analysis of seven selected bad smells reported by end users. The preliminary results suggest that most smells are introduced in the early versions of a model, and these smells often persist throughout the life cycle of the modeled system. Furthermore, most of the projects contain three or four types of smells. This may be an indication of the need for more automated analysis and refactoring support for systems engineers who may not be familiar with bad smell identification and refactoring.

CHAPTER 6

EVOLUTION OF BAD SMELLS AND MAINTENANCE TASKS IN SIMULINK REPOSITORIES

This chapter discusses how we used the Change Analyzer to analyze the evolution of four bad smells in 575 Simulink models across 31 open-source repositories. The first step was to extract the evolution history of Simulink models in GitHub. Next, we manually classified each version to a maintenance category (i.e., adaptive, preventive, corrective, or perfective). Then, we developed queries to detect instances of four selected bad smells. Finally, we analysed the evolution of each of the smells across the version history of the repositories, the relationships between the smells and the size of the models, and the impact of maintenance activities on the evolution of the identified bad smells.

6.1 Chapter Introduction

A software system often needs to be updated frequently in order to ensure that the system continues to function optimally while addressing evolving needs. These updates to the system often involve the execution of different maintenance activities (e.g., adaptive, corrective, perfective or preventive tasks) on the software. Telea and Lucian [Telea 2011] suggest that up to 80% of software cost is associated with maintenance activities and associated changes (e.g., adding new classes). Although these changes are necessary to keep the system up to date, the changes may have an unwanted side effect of introducing new smells to the software. The introduction of new smells may also make the system more prone to changes [Khomh 2012],

thereby generating an unending vicious cycle of new changes and smells that results in a poorly maintained system that is challenged when new requirements emerge. Therefore, there is a need to understand the relationship between the maintenance activities executed on a system and the evolution of bad smells in that system.

Simulink is a popular graphical tool that is widely used in both industry and academia for developing and analysing complex dynamic systems in multiple domains (e.g., avionics and automotive), with over four million users [Mathworks 2021]. Due to the popularity of the Simulink tool, there exists many complex industrial projects in open-source repositories such as GitHub. These repositories provide the necessary data to analyse Simulink models, extract the changes executed across the lifecycle of the models, and study the evolution of bad smells in these repositories.

This chapter presents our work on the analysis of bad smells evolution in Simulink models and its relationship with the maintenance tasks executed on these models. We analyse four smells in 31 GitHub repositories with 575 Simulink models. The results of the analysis suggests that larger models tend to contain more instances of smells and most smells are introduced at the first version in the repositories. This is similar to the results discussed in Chapter 5 about the evolution of bad smells in LabVIEW repositories. Furthermore, adaptive maintenance tasks tend to increase the number of smells in the repositories, while corrective maintenance tasks tend to reduce the number of smells.

6.2 Research Questions to Understand Evolution of Bad Smells

This section presents our analysis of the evolution of bad smells in Simulink models and the relationship with the maintenance activities executed on the models. Specifically, we aim to answer the following research questions.

RQ1. *What is the relationship between bad smells and the size of the models?* Khomh et al. [Khomh 2012] showed that programs with large classes (measured in terms of Lines of Code, LOC) tend to contain more bad smells than smaller classes. Although a few works have identified bad smells in Simulink projects, these works did not discuss the relationship between the bad smells and size of models. This research question aims to discover if the size of models (measured by number of model elements) has any relationship with the presence of bad smells in such models.

RQ2. *When are bad smells introduced?* The results produced by Tufano et al. [Tufano 2015] suggests that bad smells are usually introduced in a system when the program is first created. However, some categories of smells usually appear as a result of specific software maintenance tasks, and not during the initial program creation. This research question targets the time when smells are initially introduced to Simulink models.

RQ3. *What is the relationship between bad smells and the types of maintenance activity?* The study conducted by Tufano et al. [Tufano 2015] revealed that bad smells are often introduced to the software as a result of adaptive maintenance activities, while Chatzigeorgiou et al. [Chatzigeorgiou 2014] showed that trivial bad smells can be removed as a side effect of adaptive maintenance activities. This research question seeks to understand the relationship between bad smells and maintenance activities in Simulink projects.

To answer the research questions mentioned above, we implemented the following approaches.

1. We selected 31 repositories from GitHub. These repositories were selected by searching for the keyword “simulink” in GitHub and the repositories were checked to contain at least one Simulink file.

2. We constructed four queries in the Change Analyzer to detect instances of four smells in Simulink. The execution of the queries in the Change Analyzer will produce the evolution history of smells in selected models.
3. We manually classified each version of the selected GitHub repositories to one or more maintenance tasks based on the set of changes added to the version and the versions's commit message.
4. We manually analysed the results of the queries to answer the research questions discussed above.

6.3 Smell Selection and Queries to Detect Smell Instance

A number of smells in Simulink models have been identified in just a few previous works. We select four smells that were described in Gerlitz et al. [Gerlitz 2015] and Stephan et al. [Stephan 2015]. Although any arbitrary smell might have been chosen for the purpose of this analysis, the four smells were chosen due to the following reasons:

1. **Clear Metrics**. There are clear metrics for evaluating whether the smell is present or not. This means that the presence of a smell can be observed objectively and it is not subjective to user interpretation.
2. **Good Spread**. The four smells cover a wide range of smell categories. Furthermore, two of the smells have been reported by both Gerlitz et al. [Gerlitz 2015] and Stephan et al. [Stephan 2015].

The following subsection provides a detailed description for each of the smells.

6.3.1. Description of the Smells

This section introduces the four smells that have been used in this study. The selected smells are *Superfluous Subsystem* (or sometimes called *Primitive Obsession*), *Long Port List*, *Deeply Nested Subsystem*, and *Superfluous Bus Signal*. The following provides a detailed description of each of the smells.

1. **Superfluous Subsystem**. This occurs when a model contains very simple subsystems with minimal functionalities, such as when a subsystem contains only virtual blocks or a single non-virtual block. This smell increases the complexity of the model and increases the execution time. This smell is also called *Primitive Obsession* [Stephan 2015]. Figure 6.1 is a sample model containing the Superfluous Subsystem smell. The model computes the area of a circle (pi*r^2). However, the model calls a subsystem Square Subsystem that only computes the square of two numbers. This subsystem is superfluous because it contains only a very simple function and also increases the complexity of the system. Figure 6.2 is a standard model for calculating the area of a circle.

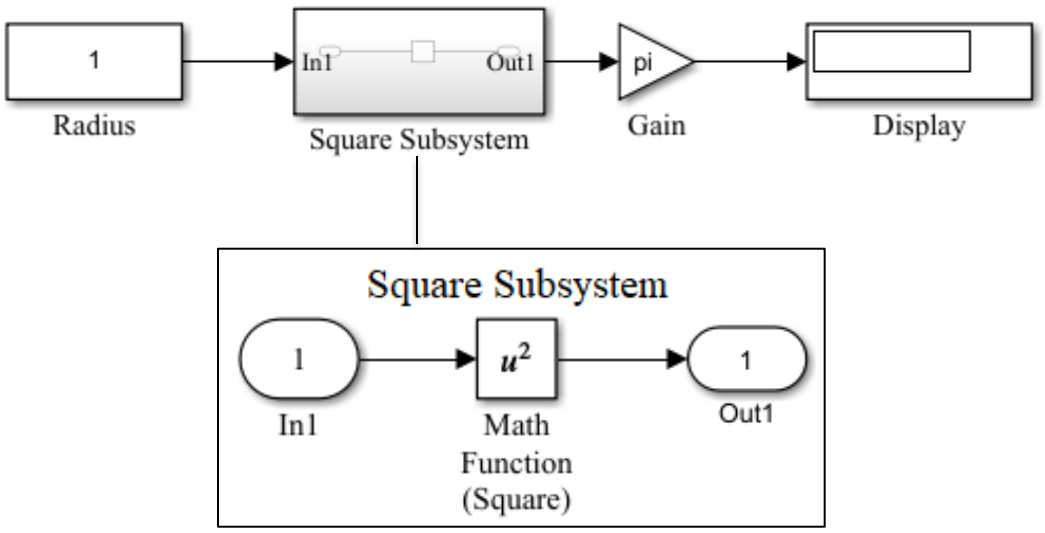

Figure 6.1. A Simulink Model with the Superfluous Subsystem Smell

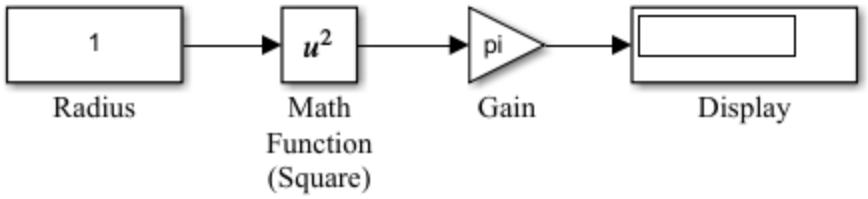

Figure 6.2. A Simulink Model to Compute the Area of a Circle

2. **Long Port List**. This smell occurs when a Simulink subsystem has too many inports or outports. An inport is the interface for receiving inputs, while the outport is used as the output interface. When a Block has too many inports or outports, it reduces the ability to visually link the connecting components together and this makes the whole model difficult to understand. This smell is similar to the smell *Long Parameter List* [Fowler 2018] that is found in text-based programming languages.

   Figure 6.3 is an example of an abstracted Simulink model with too many inports. The model has been abstracted to improve its readability. The model calculates the discount on an item, where an additional 10% discount is provided to anyone older than 60 years. The model includes a subsystem named Discount that requires four inputs for price, quantity, discount, and age of the customer. The high number of inports makes it hard to understand which input variable is required at which port and what each variable actually does. In situations similar to this example, the developer is forced to actually go through the model in the subsystem in order to understand what each input variable represents.

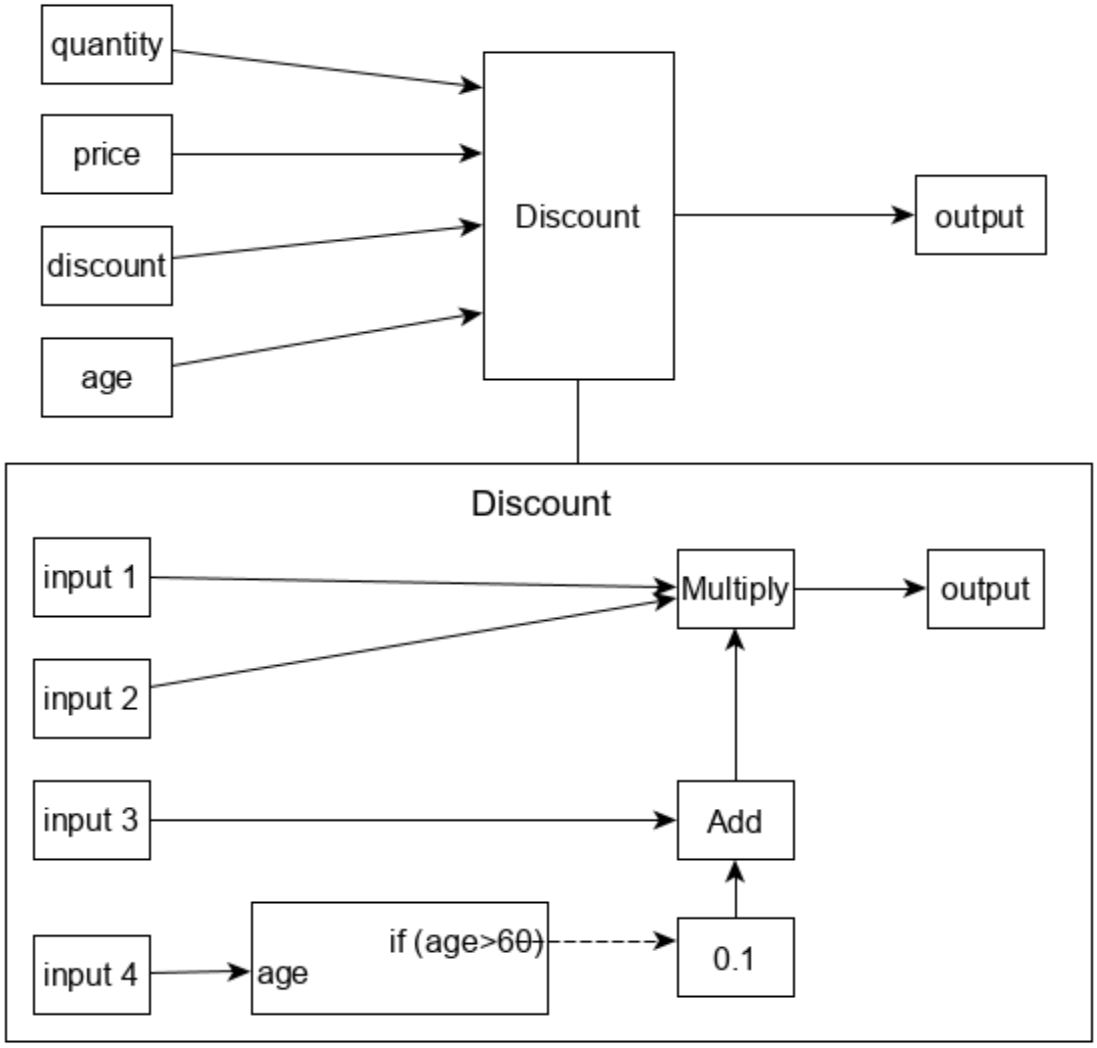

Figure 6.3. A Simulink Model with the Long Port Smell

3. **Deeply Nested Subsystem**. This occurs when a model has a deep hierarchical level of subsystems (i.e., three or more hierarchical levels). This smell makes the model very hard to understand because the execution semantics are often hidden in deeply nested layers. This smell is also known as *Middle Man* [Stephan 2015]. For example, to calculate the surface area of a cone, we add the area of the base circle and the area of the curved surface. The area of the curved surface also requires the height of the slant.

   Figure 6.4 is a sample model that calculates the surface area of a cone with radius r and height h. The model calls a subsystem Curved Surface that calculates the area of the

curved surface. The subsystem Curved Surface calls another subsystem Slant Height that calculates the slant height. A straightforward solution would be to use the formula pi*r (r + sqrt(r*r + h*h)) as shown in Figure 6.5.

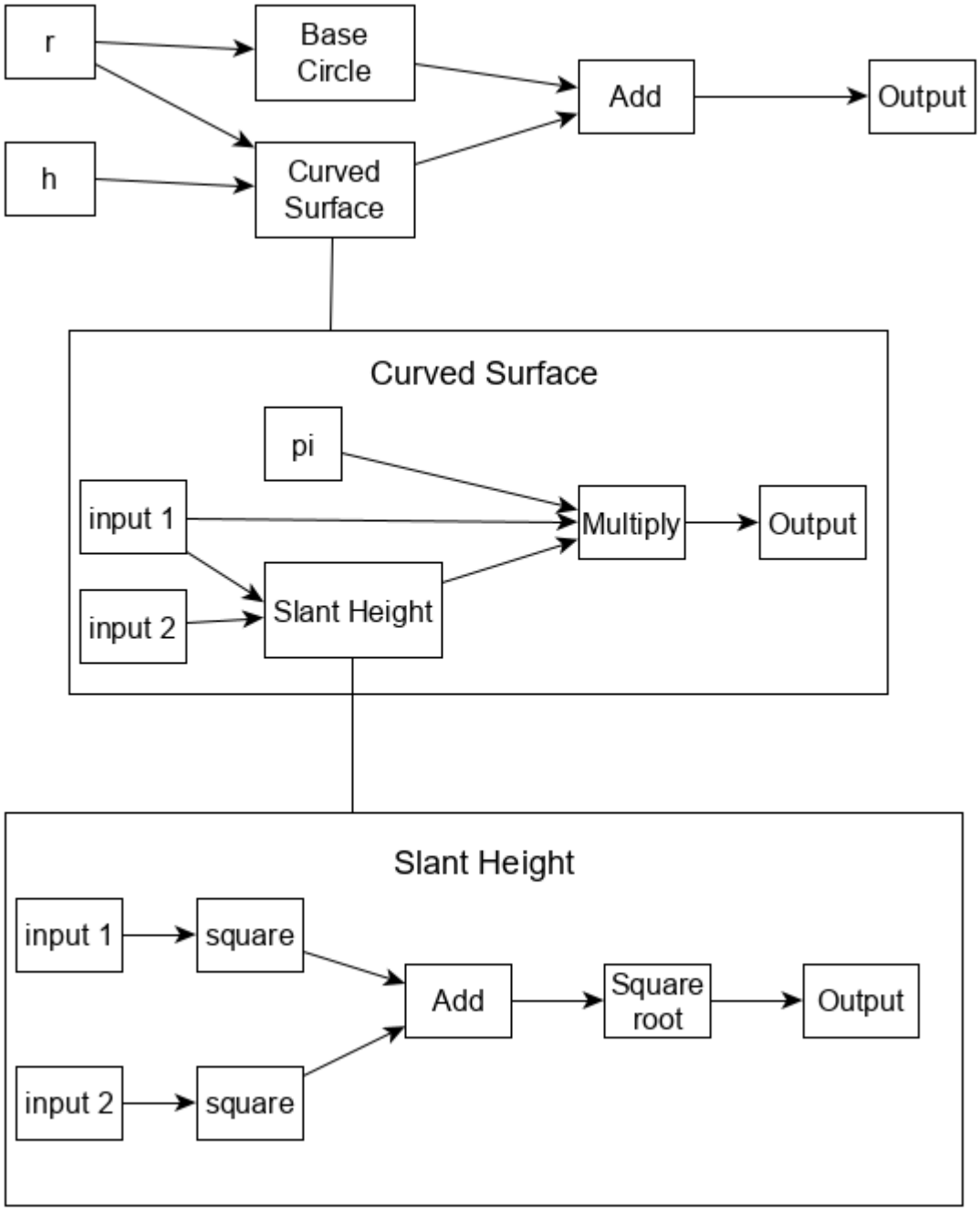

Figure 6.4. A Simulink Model with Deeply Nested Subsystem Smell

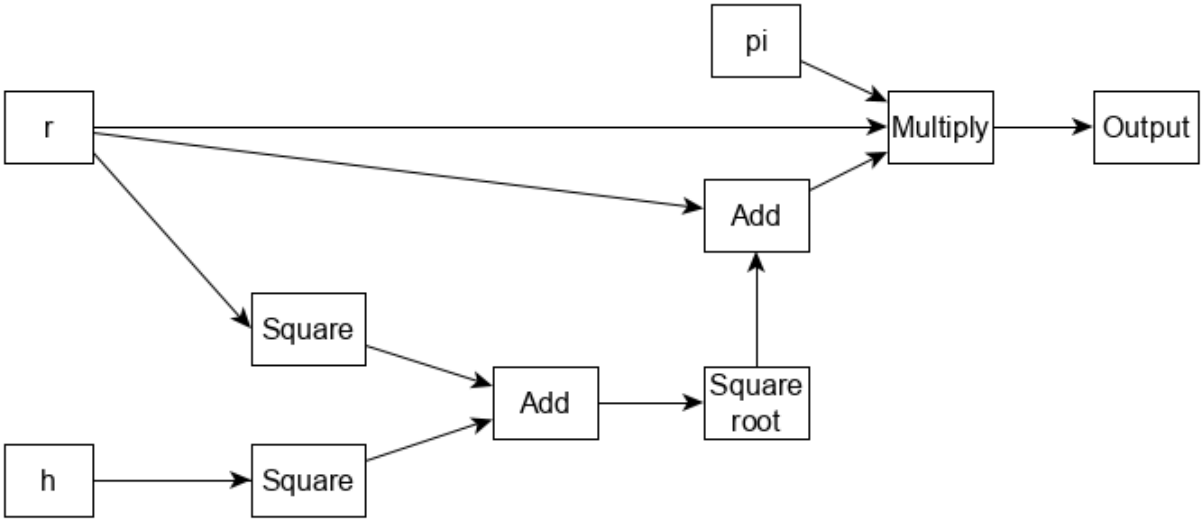

Figure 6.5. A Simulink Model to Calculate Area of a Cone

4. **Superfluous Bus Signal**. A bus is used to group related signals or connections between the entities. This is necessary to prevent too many lines from crossing each other, thereby making it hard to understand the execution flow of the model. Figure 6.6 gives an example of a Simulink Block function that requires four input constants A, B, C ,D. A direct connection between the constants and the Block might lead to clustered wires, especially if the positions of the input constants and Block are far apart in the Simulink diagram. A bus signal can be used to collapse the four wires into a single wire via the *Bus Creator*, while the *Bus Selector* can be used to decompose the wires when needed.

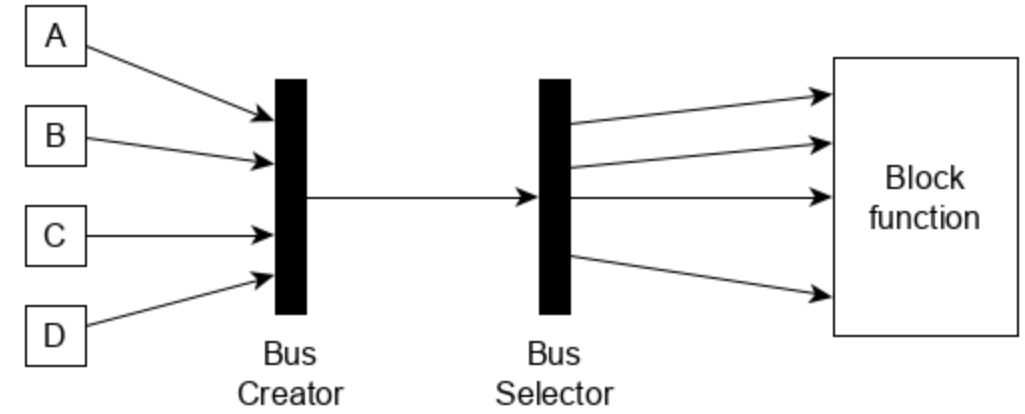

Figure 6.6. A Simulink Model with Bus Signal

A *Superfluous Bus Signal* occurs when a bus contains very few signals (one or two), which will increase the model execution time without improving the readability of the model. Figure 6.7 provides a sample of the smell where a bus signal contains only one signal, thereby adding unnecessary overhead to the model.

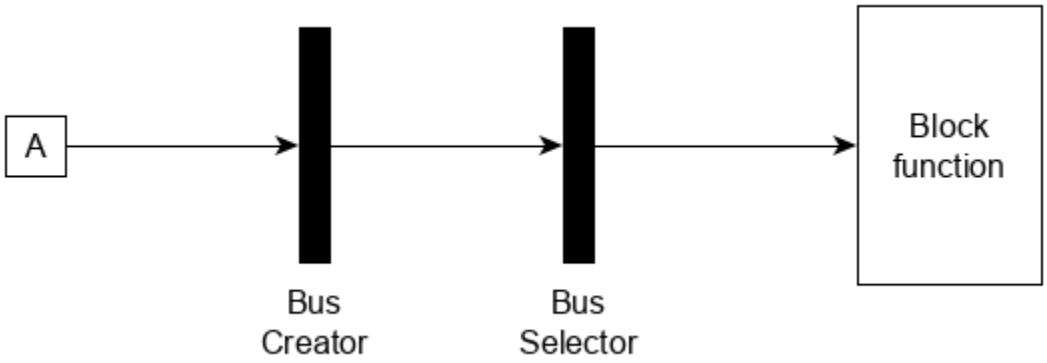

Figure 6.7. A Simulink Model Containing the Superfluous Bus Signal Smell

6.3.2. Detection of Smells via User-Defined Queries

We developed a set of queries in the Change Analyzer to detect the presence of each of the four selected smells in Simulink models. These queries return a result of all instances of each smell in a specific version. Therefore, the execution of the queries for the entire version history in a

particular repository, we were able to see how the number of instances of each smell changes across the lifecycle of models in the repository. For example, to detect instances of the *Deeply Nested Subsystem*, we iterate through the Subsystems (defined by checking if a sub block is a subsystem) at the root of a model. For each Subsystem, we also extract all of the Subsystems at the second hierarchical level. Finally, we iterate through the Subsystems at the second hierarchical level and detect the number of subsystems at the third hierarchical level. The query will return the number of Subsystems at the third hierarchical level for each model. Listing 6.1 is a query for the Change Analyzer to detect the number of instances of *Deeply Nested Subsystem* smell in a model.

```
var x=0;
for (y in SubSystem . allInstances ){
      for (a in y. subBlocks ){
          if (a. isKindOf ( SubSystem )){
                for (b in a. subBlocks ){
                    if (b. isKindOf ( SubSystem )){
                          for (c in b. subBlocks ){
                              if (c. isKindOf ( SubSystem )){
                                    x=x +1;
                                    continue ;
                              }
                          }
                    }
                }
          }
      }
}
return x;
```

Listing 6.1. Query to Detect Instances of Deeply Nested Subsystem Smell

To detect instances of the *Superfluous Subsystem*, we iterate through all the instances of the Subsystem to search for Subsystems with only virtual blocks or a single non-virtual block. Instances of the *Long Port List* were detected by iterating through elements of type *Block* and searching for elements with more than three inports or outports. Finally, we detect instances of

Superfluous Bus Signal by iterating through the *Bus* elements to extract elements with less than two *Signals*. The queries for detecting instances of the four smells is provided in Appendix B.

6.4 Overview of Selected Repositories

The work presented in this chapter includes the analysis of more than 500 models in 31 GitHub repositories. To select the repositories used in this study, we searched for the keyword "simulink" in GitHub and the initial results produced more than 700 repositories. The repositories were manually examined and we filtered out repositories with less than 10 commits. The threshold was chosen in order to ensure the repositories have enough version history to study its evolution. The filtered results produced about 100 repositories. Finally, we used the Change Analyzer to query the filtered repositories to extract repositories with more than five versions (i.e., more than five commits with changes to at least one Simulink model). The final set of repositories used for this study consists of 31 repositories across many industrial and academic domains. The complete list of the 31 repositories is provided in Appendix C. Table 6.1 provide a statistical overview of the selected repositories. The table shows the total number of commits, the average number of commits, the minimum number of commits in any of the repositories, the maximum number of commits in a single repository, the average number of commits across all the repositories, and the median number of commits. The same set of statistical data was also extracted for the number of versions, models, model elements, branches, and contributors. It can be seen from the table that this study analyzes 575 models containing an aggregate of over four million model elements.

Figure 6.8 provides a graphical summary of the domains of the repositories. The figure shows that the robotics and avionics domains have the highest number of repositories. Furthermore, we could not assign two repositories to a particular domain; therefore, we labelled

them as others. In total, the 31 repositories span across 13 different domains, thereby showing the heterogeneity of the repositories used in this study, which offers a more diverse understanding of the evolution of bad smells and maintenance tasks.

Properties\ Variables	Commits	Version	Models	Model elements	Branches	Contributors
Total	6518	564	575	4254037	148	74
Average	210.3	19.2	18.5	137227	4.8	2.5
Minimum	20	5	1	911	1	1
Maximum	1168	90	174	830193	48	8
Median	87	11	8	68449	1	1.5

Table 6.1. Overview of Selected Simulink Repositories

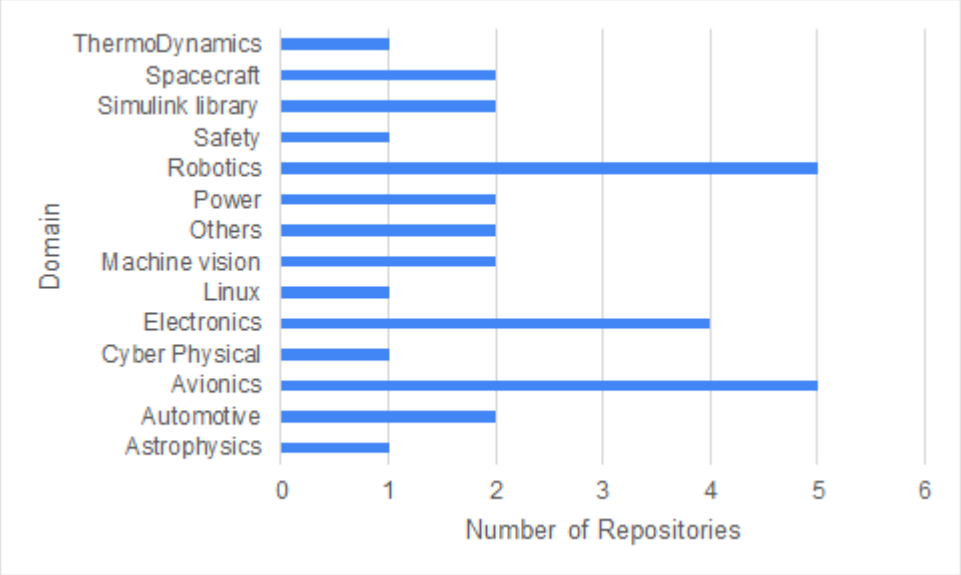

Figure 6.8. Domain Distribution in the Repositories

Figure 6.9 provides a summary of the prevalence of the four selected smells across the 31 repositories. The data in the figure shows that instances of the *Superfluous Subsystem* smell is the most prevalent across the 31 repositories, and the least prevalent smell is the *Superfluous Bus*

Signal. Figure 6.10 gives a more detailed distribution of the smells across each of the repositories, and shows that all of the repositories contain instances of at least one smell, while only two repositories contain instances of all four smells used in this study.

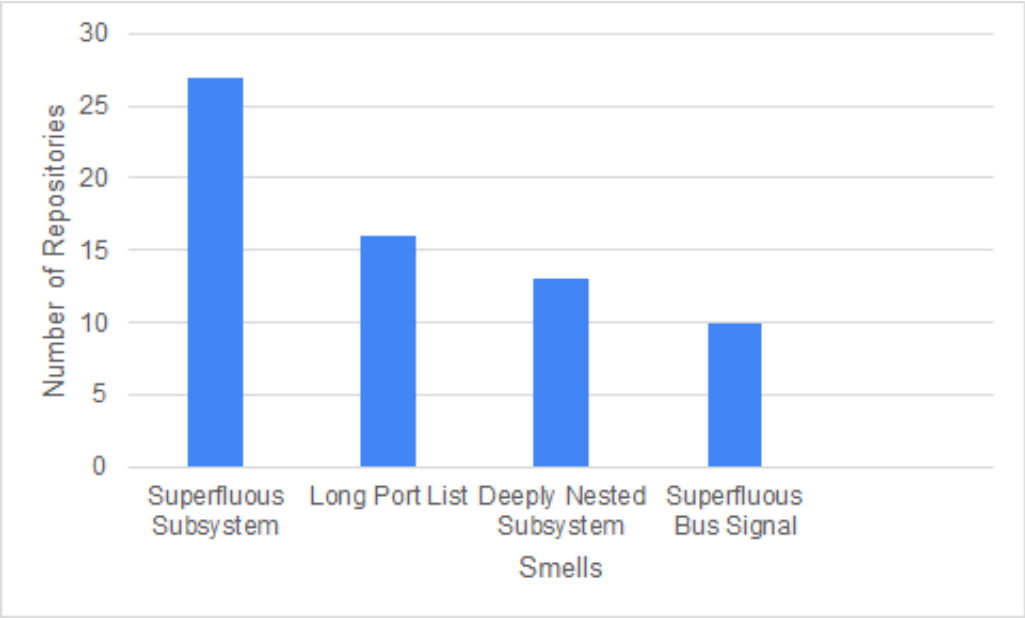

Figure 6.9. Number of Repositories with Instances of Each Smell

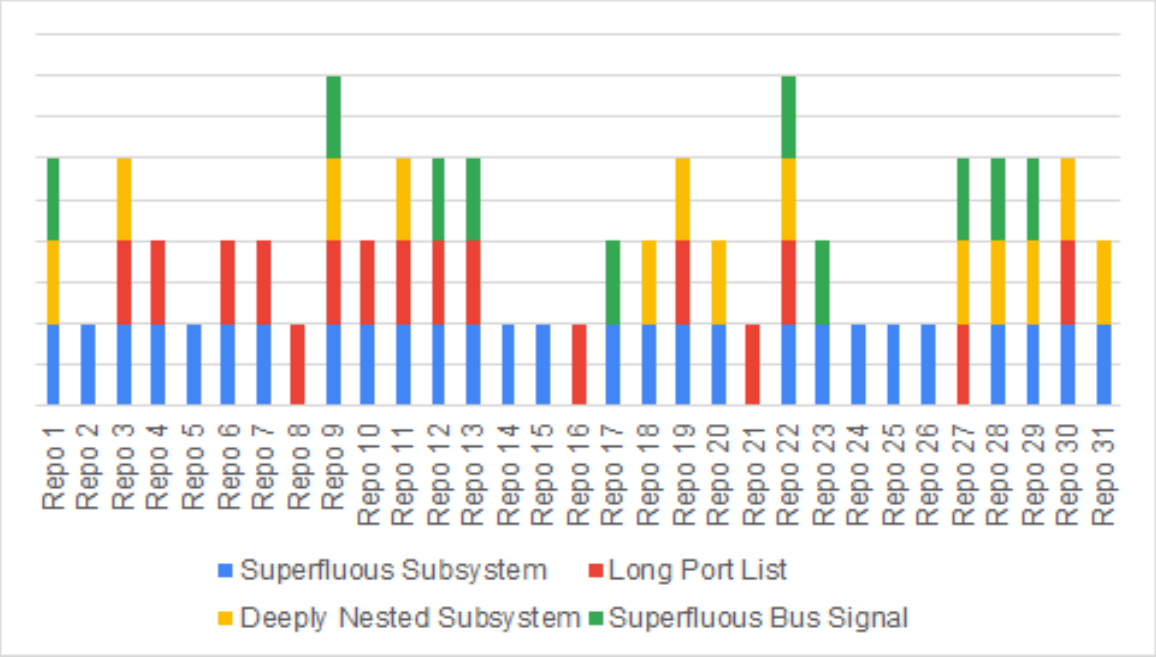

Figure 6.10. Smell Distribution across the Repositories

6.5 Research Results and Analysis

We developed three research questions to study the evolution of Simulink models. The research questions deal with the relationship between bad smells and model size in Simulink repositories, the introduction of bad smells, as well as the maintenance activities that are likely to affect the evolution of bad smells. The research questions, which were discussed in more detail in Section 6.2, are as follows:

RQ1 *What is the relationship between bad smells and the size of the models?*

RQ2 *When are Bad Smells Introduced?*

RQ3 *What is the relationship between bad smells and the types of maintenance activity?*

The following subsections discuss how we analysed the evolution of bad smells to answer the research questions listed above.

6.5.1. RQ1: Bad Smells and Model Size

To answer this research question, we analysed the relationship between bad smells and model size from two perspectives. First, we considered the relationship between the presence of smells across all of the selected repositories and the size of such models. Second, we analysed the evolution of bad smells and the corresponding increase or decrease in the number of model elements within the evolution history of models in the same repository.

For the first analysis, the results show that for most of the repositories with model size less than 10,000 model elements, the repository contains only one or two types of smells. Furthermore, as the size of the models increases, the models tend to contain more variety of smells. This may not be surprising because instances of some smells such as *Deeply Nested Subsystem* are likely to be found in only large models. Figure 6.11 summarizes the results of the first analysis.

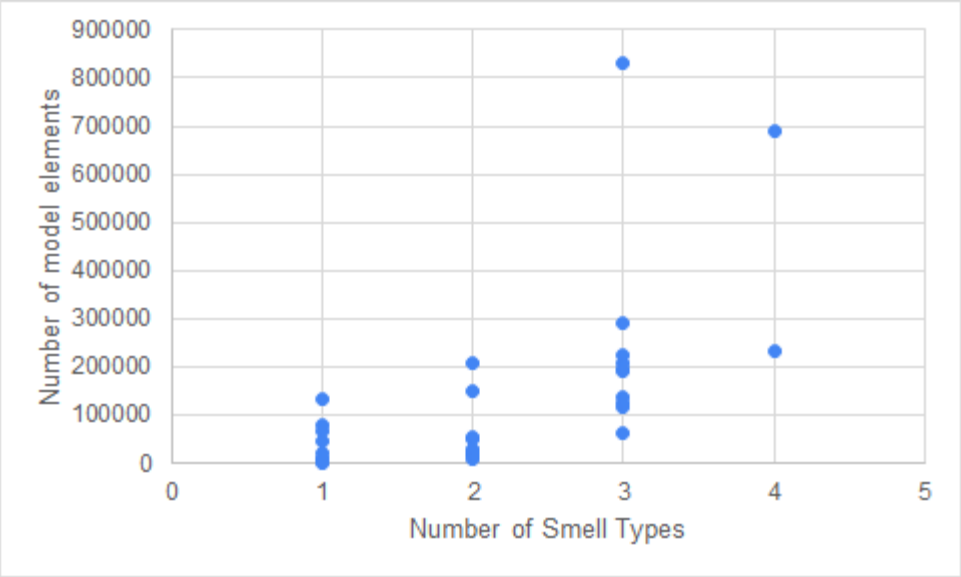

Figure 6.11. Relationship between Smell Types and Model Size

The second analysis examined relationships between the smells and model size for each repository. The analysis shows that in most cases, the size of each model and the number of smell instances continuously increase across each version. Hence, we observed over 250 cases where the size of models in a repository increased from one version to another. This is compared to only 20 cases where the size of the models actually decreased across successive versions. There were also 124 cases where the instances of smells increased, while there were only 25 cases of smells decreasing across the versions. Furthermore, the size of the models increased in 117 cases (out of 124) when the number of smells present in the models increased. This indicated that there are about 169 cases (out of 284) where the size of the models increased, but the number of smells either remained the same or decreased. There were also five cases where the size of the models increased, but the number of smell instances decreased. However, it should be noted that in two of these cases, there were increases in other types of smells (e.g., an increase in model size corresponding to a decrease in instances of *Superfluous Subsystem* smell, but increase in instances of *Superfluous Bus Signal* smell within the same version). There were also 17 cases

where a decrease in the size of the models corresponded to a decrease in the smell instances. This suggests that there were only three cases where the decrease in size of a model does not have an effect on the number of smell instances. Finally, there were no cases where the number of smells increased while the size of the models decreased. Figure 6.12 summarizes the result of this analysis on bad smells and model size.

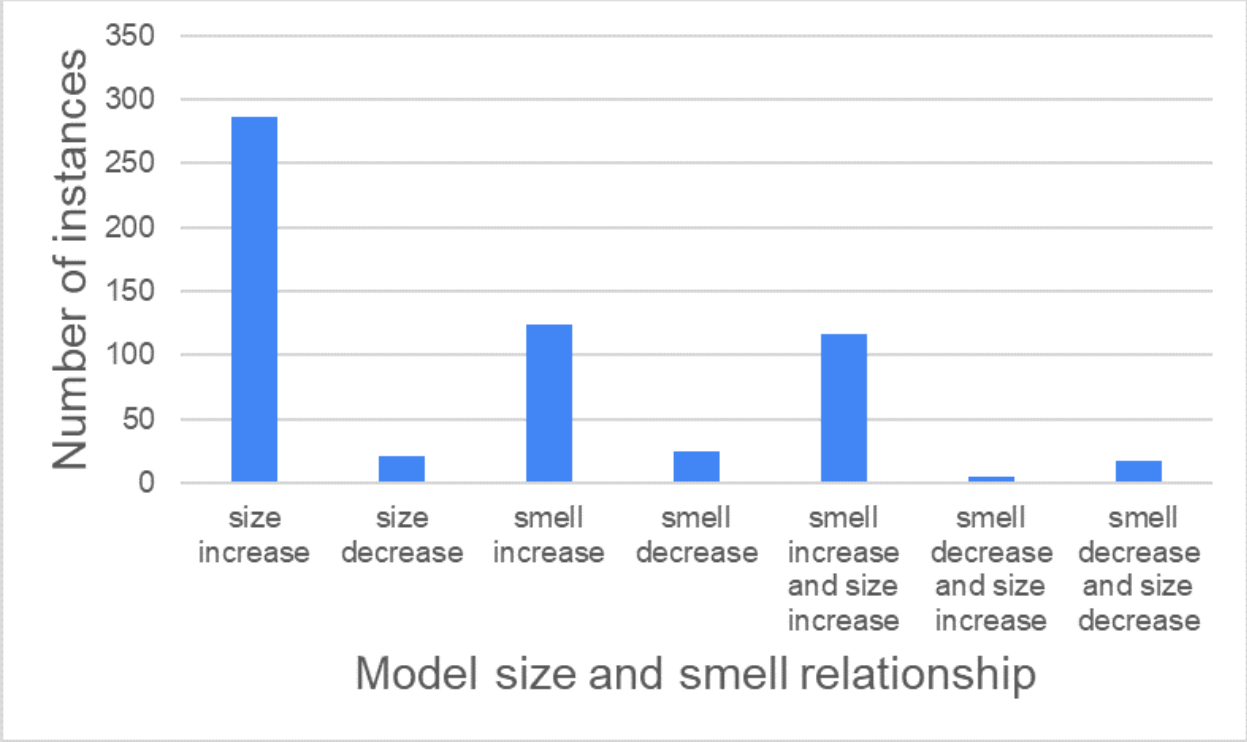

Figure 6.12. Relationship between Model Size and Smell Instances

Figures 6.13 and 6.14 provide a graphical overview of the evolution of bad smells and the model size in repositories 3[16] and 27[17], respectively. These two repositories provide a sample representation of the evolution of bad smells in the repositories, because it is not feasible to provide such graphs for all 31 repositories. The two repositories were chosen because they cover all of the bad smells and have sufficient version history with cases of both increases and

[16] https://github.com/analogdevicesinc/MathWorks_tools
[17] https://github.com/voldemoriarty/mrac

decreases to the size of the models and the smell instances. It should be noted that Repo 3 does not contain any instances of the *Superfluous Bus Signal* smell.

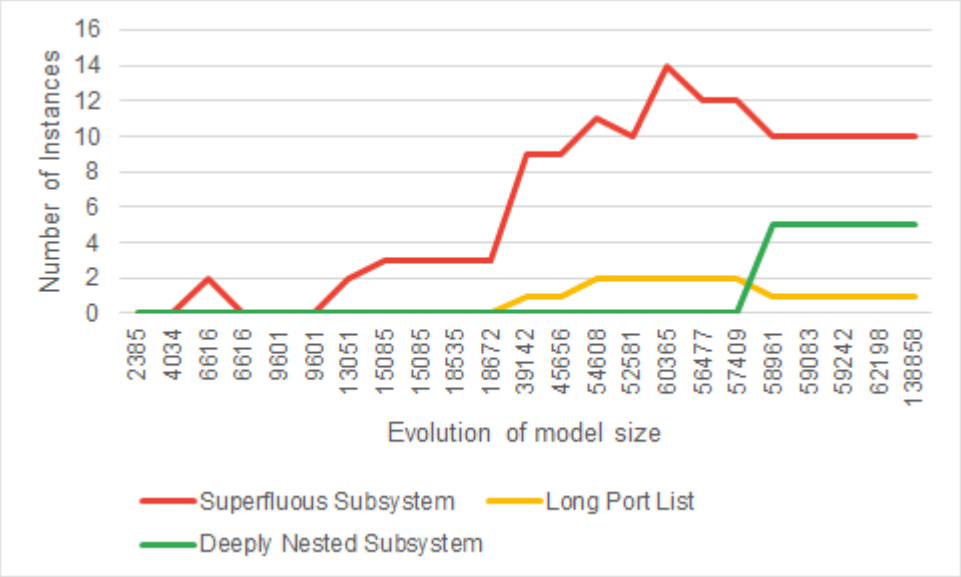

Figure 6.13. Evolution of Bad Smells in Repository 3

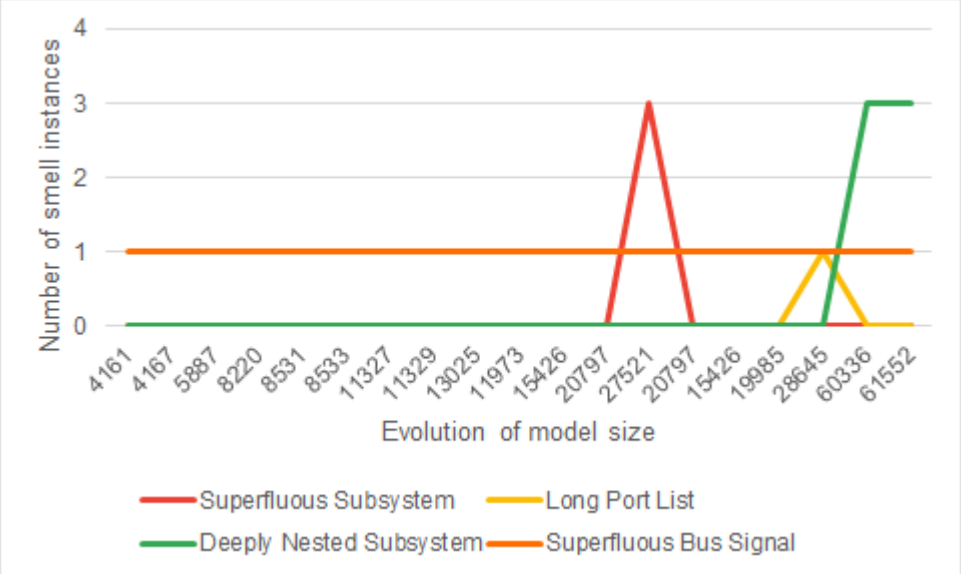

Figure 6.14. Evolution of Bad Smells in Repository 27

Summary of RQ1

Larger models tend to contain a higher number of smell types and smell instances. Hence, an increase in the number of smell instances is very often associated with an increase to the model size, and a decrease in the smell instances also correlates with a decrease in the model size. However, changes to the size of the models does not necessarily correspond to a resultant change to the number of smell instances.

6.5.2. RQ2: Introduction of Bad Smells

This research question aims to detect when bad smells are introduced into a repository. The results captured in Figure 6.11 show that all the repositories have at least one instance of a smell across each version history. In order to detect when bad smells are introduced to the repositories, we searched for the version that contains the first instance of any smell and the first instance of each of the four smells across the repositories. We discovered that 26 out of the 31 repositories (i.e., 84%) contain at least one instance of a smell in its first version, while two repositories have the first instance of any smell in their second version. The remaining three repositories recorded their first smell in the 3rd, 4th and 6th version, respectively. All of the four smells selected for this study were often introduced to the repositories in the first version. 66% and 62% of the *Superfluous Subsystem* and *Long Port List* smells were introduced in the first version, while 66% and 57% of the *Deeply Nested Subsystem* and *Superfluous Bus Signal* smells were introduced in the first version of the repositories. In the second version, 11% of the *Superfluous Subsystem* smell, 13% of the *Long Port List* smell, 8% of *Deeply Nested Subsystem* smell, and 14% of the *Superfluous Bus Signal* smell were introduced to the repositories. Figure 6.15 summarizes the results. It should be noted that the smell “Any” refers to instances of any of the four smells used in this study.

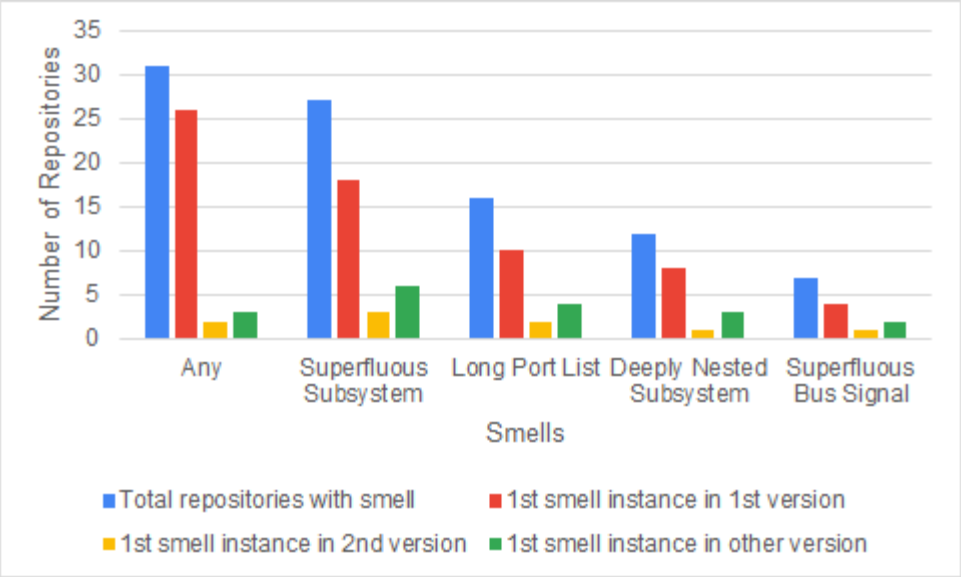

Figure 6.15. Smell Introduction across the Repositories

Summary of RQ2

About 60% of the smells were introduced in the first version of the repositories, while about 40% of the bad smells were introduced because of the maintenance tasks during the model evolution. This observation aligns with the results produced by Tufano et al. [Tufano 2015] in their study of bad smells in three open-source Java-based systems.

6.5.3. RQ3: Bad Smell and Maintenance Tasks

The maintenance of software systems is important to ensure that a software system continues to function optimally. Therefore, a software system is subject to continuous changes due to maintenance tasks executed on the system. These maintenance activities are usually embedded in the version history of the models stored in a VCS. The phase of this study associates each version of a model history with a maintenance activity and compares the maintenance activity with the evolution of bad smells in the repositories. Four main types of maintenance activities

have been identified based on the ISO/IEC 17464 standard [ISO/IEC 2006]. The four maintenance categories include the following.

1. **Adaptive**: This activity focuses on enhancing the functionalities of the software to satisfy new requirements or specifications. Examples of common adaptive maintenance tasks include adding new features, satisfying new requirements or adapting to new external factors.
2. **Corrective**: This maintenance activity is associated with fixing an issue that negatively affects the optimal functioning of the software. This maintenance activity is usually executed when the system is already in use. The set of tasks associated with this maintenance category includes correcting bugs or mistakes in the software.
3. **Perfective**: This activity is associated with optimizing the functionalities or features of a software. The set of tasks associated with this maintenance category include reorganizing the code base, improving the software documentation, and refactoring the software to improve maintainability.
4. **Preventive**: This includes the set of maintenance tasks that is done in order to address a possible future need. Examples of preventive maintenance tasks include testing components of the software.

We classify each version of the selected GitHub repositories to one or more maintenance tasks based on the set of changes added to the version and the versions's commit message. The set of changes were extracted via the Change Analyzer. The set of labels and descriptions recommended by Murgia et al. [Murgia 2014] was used as a guideline for classifying each version into its respective maintenance categories. This manual classification process was independently done by two PhD candidates. The few versions (about 10 versions) that could not

be resolved to common maintenance types by the two students were added to all the categories under consideration for that specific version. For example, if the first student thinks a version belongs only to the adaptive category, while the second thinks it belongs only to the perfective category, that version is assigned to both the adaptive and perfective categories. It should be noted that more than one maintenance task is usually carried out in a single version.

A total of 421 versions across the 31 repositories were annotated to a maintenance category. 334 of the 421 versions were classified to a single maintenance type, 60 versions were classified to multiple categories, and 27 versions could not be assigned to a maintenance category because the commit messages were written in non-English languages or because the commit messages did not contain enough descriptive detail. At the end of the classification process, 54% of the versions were classified as perfective, 37% were classified as adaptive, 14% were classified as corrective, and 1% were classified as preventive. Figure 6.16 summarizes the distribution of the maintenance types across the 31 repositories. The Figure shows the number of versions for each maintenance category. The single classification index indicates the number of versions that were classified as only the maintenance type under consideration, while multiple categories indicate the number of versions that included the maintenance type under consideration along with one or more other maintenance types. Figure 6.17 provides a graphical overview of the distribution of the maintenance types across the 31 repositories.

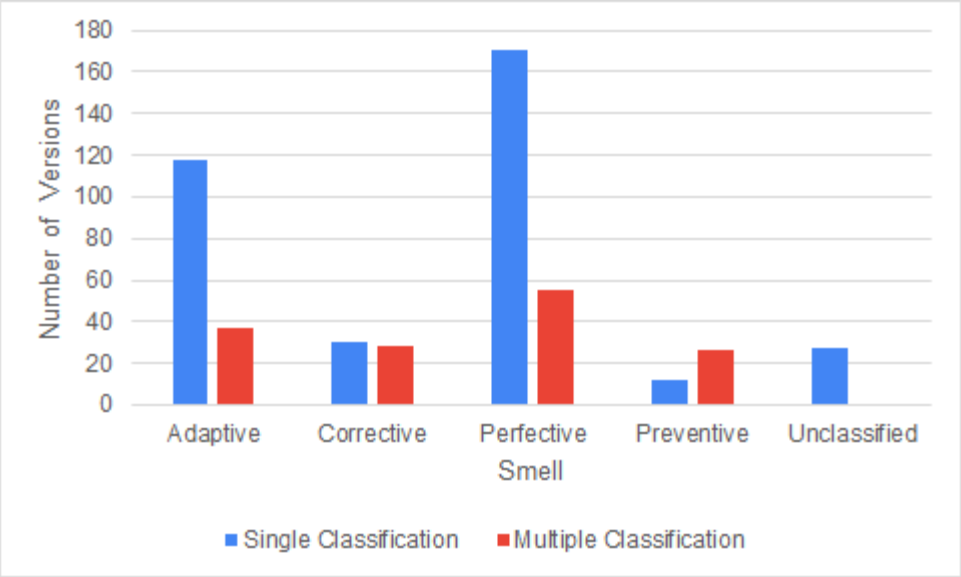

Figure 6.16. Number of Versions and Maintenance Types

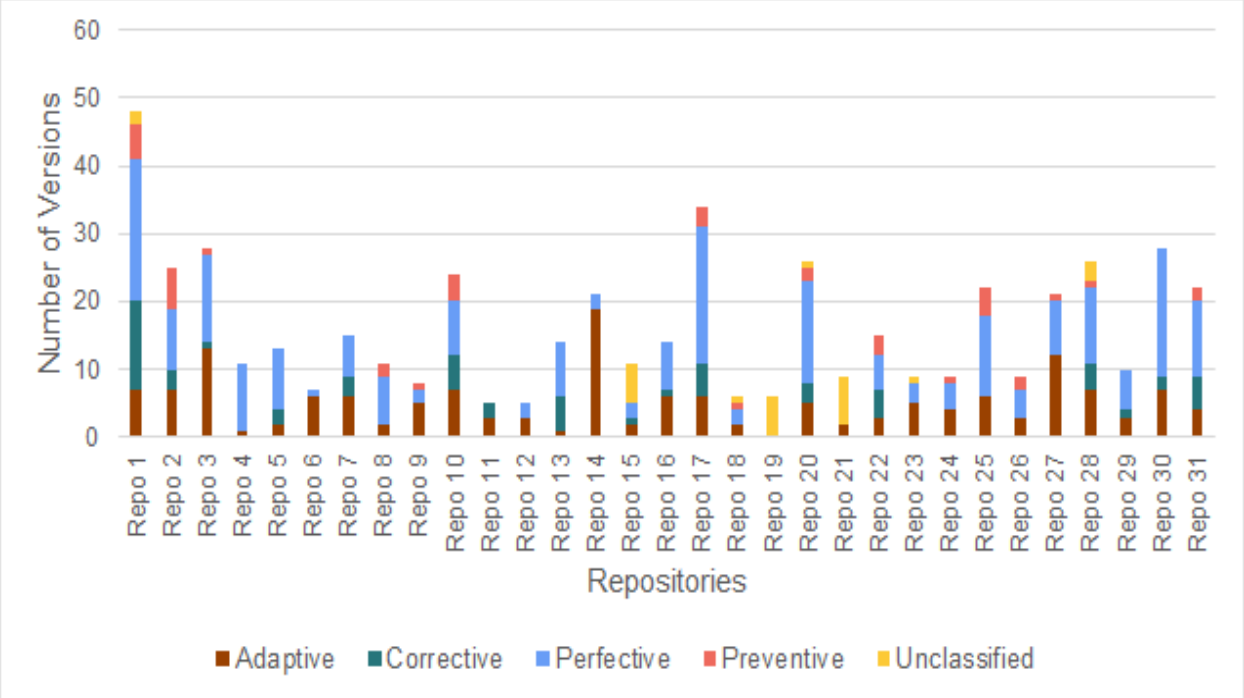

Figure 6.17. Distribution of Maintenance Types across Repositories

We analysed the maintenance types associated with each version and the changes in the number of smell instances in each version. We then extracted the type of maintenance types for each version where the number of smell instances increased or decreased. This makes it easy to

visualize the type of maintenance activities that influenced the increase or decrease of smells in Simulink models. At the end of the analysis, adaptive maintenance was observed to have the highest contribution to increase in smells in the repositories. The adaptive maintenance tasks correlate with smell increase in 64 versions out of the 155 versions (41%) associated with adaptive maintenance tasks, and a smell decrease in four versions. The corrective maintenance tasks contributed to smell increase in 17 versions out of 58 versions (29%) and smell decrease in seven versions. The perfective maintenance tasks correlated with smell increase in 58 versions out of 225 versions (26%), but it also contributed to the highest decrease in smell instances with a smell decrease in 22 versions. Finally, the preventive maintenance tasks contributed to smell increase in 12 versions out of 38 (32%), and smell decrease in three versions. It should be noted that these numbers included versions that involved changes of a single maintenance type and versions that involve changes of multiple maintenance tasks. For example, the preventive maintenance tasks contributed to 12 versions, but only two versions were exclusively preventive maintenance tasks. This means that other types of maintenance tasks were also executed in the remaining 10 versions.

Figures 6.18 and 6.19 provide a graphical overview of the results of the analysis. Figure 6.18 shows the relationship between the maintenance types and increases in smell instances while Figure 6.19 provides an overview of the relationship between the maintenance types and decrease in smell instances.

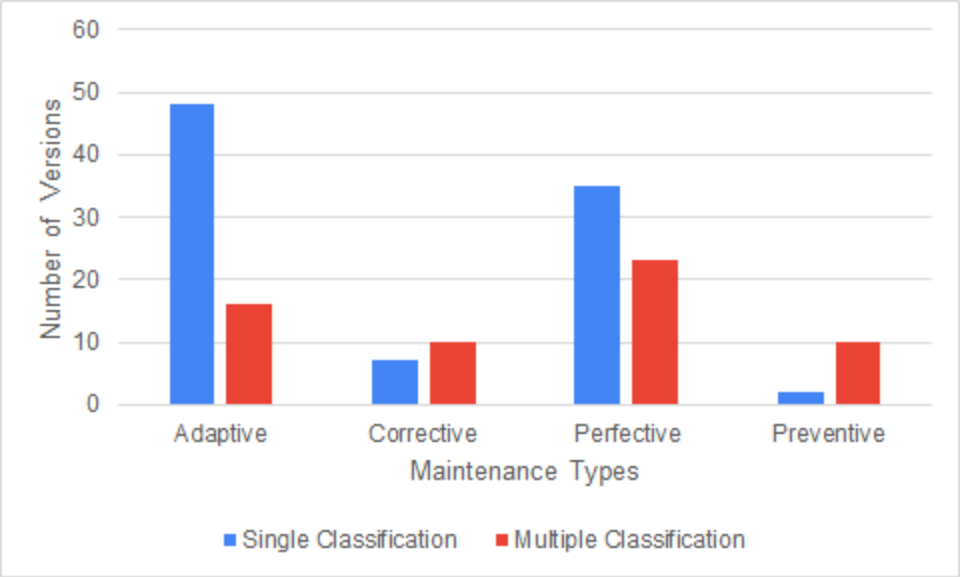

Figure 6.18. Maintenance Types and Smell Increase

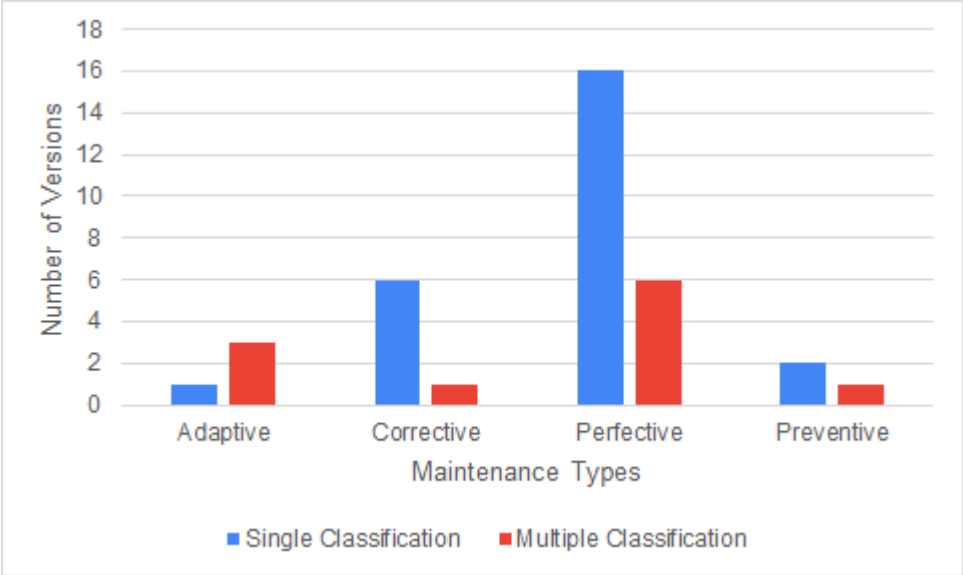

Figure 6.19. Maintenance Types and Smell Decrease

Summary of RQ3

Adaptive maintenance activities usually lead to an increase in the number of smells, while a decrease in the number of smells is usually related with perfective maintenance activities.

6.6 Threats to Validity

There exists a number of threats that can affect the validity of the results we present in this chapter. Although a number of steps have been taken to limit the effects of these threats, we highlight some of the important limitations of this work.

1. **Tool Correctness**. The analysis presented in this paper relies on the correctness of a number of tools, especially the faithful conversion of Simulink models to EMF. For example, we may not be able to guarantee that a smell that has been detected in the EMF-equivalent of a Simulink model actually exists in such a Simulink model. Although we have taken a number of measures (such as manual verification of smells in the Simulink models) to reduce the possibility of this threat, we cannot guarantee that the EMF-equivalent model produced by the Massif tool will always be a true representative of the original Simulink model.
2. **Limited Number of Smells**. The analysis presented in this chapter was based on four smells in Simulink models even though existing works have identified up to 15 smells in Simulink models. Although the selected smells have been carefully chosen to be representative for most Simulink smells, the analysis presented in this paper might not be generalizable for all smells.
3. **Non-Simulink Artifacts in the Repositories**. Simulink repositories often contain other artifacts such as MATLAB code. Therefore, it is possible that the maintenance task extracted from a commit may not be exclusively due to changes to the Simulink models. This may lead to incorrect classification of maintenance type. We have tried to reduce the potential of this threat by considering both the commit message and the changes to only

the Simulink models in order to obtain a more accurate classification of maintenance activities that specifically target the Simulink models.

6.7 Related Work

Tahmid et al. [Tahmid 2016] investigated the relationship among bad smells in different components throughout the software lifecycle. They observed that the number of interconnected code smells increases continuously as the system evolves until a mega cluster forms. This makes it easy to group smelly parts for refactoring. Chatzigeorgiou et al. [Chatzigeorgiou 2014] conducted a study involving three smells in two open source projects in order to understand how bad smells evolve and the frequency of refactoring activities that target the removal of bad smells. Their study showed that many bad smells are often introduced at the first version in the repositories and the smells usually remain throughout the system lifecycle. Furthermore, they also noted that there were few refactoring activities that targeted the removal of these smells.

Khomh et al. [Khomh 2009] conducted a study to analyse the impact of bad smells on change-proneness of software. Their results show that classes with higher smells are more prone to changes. Palomba et al. [Palomba 2013] developed an approach to detect bad smells in code based on the history of changes executed on the code. Their results show that the use of the software's change history makes it possible to detect bad smells that cannot be detected with the use of only code analysis techniques. Tufano et al. [Tufano 2015] analysed the change histories of 200 open-source projects in order to understand when and why code smells are introduced into software. Their results confirm the intuitive observation that code smells are usually introduced when adding new features to the system. Aversano et al. [Aversano 2020], Olbrich et al. [Olbrich 2009], and Palomba et al. [Palomba 2018b] conducted a study of the impact of code smells on changes made to software during the evolution of the software. Their studies show that

the smelly parts of software changes more frequently and these smelly parts also exhibit a different change behaviour.

Palomba et al. [Palomba 2018a] also conducted a large-scale study to understand the evolution of smells that often occur together. They observed that more than 50% of the smelly parts of software have more than one type of smell and the authors were able to identify six pairs of smell types that frequently occur together. Furthermore, the authors also reported that these co-occurring smells are often removed together and the smell removal process usually occurs due to the deletion of the smelly code. Peters and Zaidman [Peters 2012] introduced a tool for mining software repositories in order to determine the lifespan of smells during the evolution history of seven software systems. Their results show that there are minimal refactoring activities that specifically target the removal of smells even though the developers are often aware of the presence of these smells.

Most of the approaches discussed in the literature have targeted text-based programming languages. Zhao and Gray [Zhao 2019] categorized different types of bad smells in LabVIEW systems models from discussions in forum posts, while Chambers and Scaffidi [Chambers 2013] used interview techniques with LabVIEW experts in order to identify potential bad smells. The work most similar to our work was conducted by Stephan et al. [Stephan 2015] and Gerlitz et al. [Gerlitz 2015]. Stephan et al. adopted clone detection techniques to identify bad smells in four Simulink projects. Gerlitz et al. developed a catalog of bad smells for Simulink models based on their own experience. The identified smells were grouped into five main categories and techniques for detecting and refactoring the smells were presented. To our knowledge, none of these approaches studied the evolution of smells in graphical models, nor has any work analysed the relationship between bad smells in systems models and various maintenance activities.

6.8 Chapter Summary

In this chapter, we discussed the evolution of bad smells in Simulink models and the relationships among different maintenance tasks executed throughout the model lifecycle. We identified 31 repositories containing 575 models with at least five versions of changes executed on Simulink models. The repositories span across 13 domains thereby ensuring that the results of our analysis should be generalizable to diverse domains. Then, we developed queries for our Change Analyzer to detect instances of four bad smells in the models. These bad smells were selected because they cover more smell categories and they have clear metrics for detecting instances of the smells. The final step before the analysis phase involved the annotation of each version in the repositories to indicate the maintenance activities executed in that version. The type of maintenance activities was manually identified based on the changes executed in that version and the version's commit message.

The first analysis conducted in this study targeted the impact of model size on the evolution of bad smells. Then, we studied the first instances of smells in the models to understand when bad smells are often introduced to the repositories. Finally, we examined the correlation between the maintenance tasks executed on the models and the resultant increase or decrease in bad smells.

The results of the analysis show that larger models tend to contain more smells and an increase in the size of models is likely to introduce new instances of bad smells to the models. Furthermore, our analysis shows that most smells are introduced in the first version of the repositories. Finally, we were able to show that adaptive maintenance tasks are likely to introduce new smells to the models and decrease in smells are often associated with perfective maintenance tasks.

CHAPTER 7

CLASSIFYING CHANGES ASSOCIATED WITH DESIGN DECISIONS IN SIMULINK MODELS

This chapter discusses how we used the Change Analyzer to classify changes associated with design decisions, which helped us to understand the evolution of Simulink models in GitHub repositories. In this chapter, we mined several versions of Simulink models, extracted changes across the versions, and classified the extracted changes that are associated with design decisions. To aid in the change classification process, we used a set of classifiers to predict the categories of change instances. The approach has been evaluated on more than 300 versions of Simulink models across 28 repositories with a weighted average F-1 measure of 79%. The classified changes automated by our work have also been compared with a set of labels associated with design decisions that were manually extracted from the issues reported within a similar timeframe. The comparison results indicate a high similarity of the classified changes automated by our tool with the manually classified labels, even though the manual classification process takes much more time and often does not provide additional information about the changes executed to implement the design decisions.

7.1 Chapter Introduction

The design of software is often a complex process that involves a series of stakeholder discussions, tradeoffs among alternating design decisions, technological or cost limitations, and other considerations [Van der Ven 2006]. At the end of a software design process, two products are usually generated: the final design blueprint and the set of design decisions that led to the

development of the design [Van der Ven 2006]. Much emphasis is often placed on the final design blueprint as the outcome of the process, but the set of decisions that include considered alternatives, tradeoffs, and rationale often remain only in the mind of the designers [Van der Ven 2006]. Yet, the set of decisions that captures the intent of the original designers for adopting a specific design template is important toward understanding the final software design and implementation. Furthermore, existing research shows that developers who are aware of the impact of their changes are likely to produce higher quality software [Paixao 2017], and that properly documented design decisions aid in understanding the system [Shahin 2014].

Unfortunately, while the final blueprint of the software design may be captured by a model and documented for future use, the set of decisions that led to the development of the design are often not documented and the tacit rationale behind the design may be lost forever, especially if the original developers are no longer available [Van Heech 2009]. This leads to a situation termed *knowledge vaporization* of software systems and its associated problems (e.g., repetition of past mistakes and increase in maintenance cost) lead to a loss of information behind a solution to a set of system requirements [Tyree 2005][Van der Ven 2006]. Knowledge vaporization contributes to challenges such as a lack of understanding of adopted technical solutions, repeating past mistakes, use of incompatible technologies, reduced knowledge transfer across teams, and inadequate reusability of important concepts. Other negative effects of knowledge vaporization are an increase in bad smells and technical debt, the violation of design rules and constraints (compromising a system's integrity), bad design decisions, and an increase in maintenance costs across the system life cycle [Borrego 2019][Capilla 2007][Van Heech 2009].

Several studies have been conducted to monitor, track and capture the set of design decisions in software systems. However, these studies often focus on the proper documentation of the

design decisions and they rarely discuss the recovery of such design decisions from existing software, nor do they attempt to find out why a decision was made [Shahbazian 2018]. Furthermore, these studies often target software systems developed with general-purpose programming languages, and not systems models.

The classification of changes into its maintenance types helps to understand the reason behind a change, supports diverse analysis regarding a changeset, and helps with future decision-making tasks [Hindle 2009, Mockus 2000, Yan 2016]. For example, knowing that a changeset deals with corrective maintenance activities such as fixing a bug helps in understanding why the change set was implemented in the first place. However, existing research in change classification is rarely associated with design decisions in systems models, but focus more on text-based general-purpose programming languages.

This chapter introduces the concept of design decisions for models and contributes towards understanding the rationale behind the design decisions based on the premise that the type of changes made to a system is a good indicator of the reason behind such changes [Mockus 2000]. Our research described in this chapter makes the following contributions:

1. We introduce the notion of design decisions for systems models defined in Simulink.
2. We provide a change classification approach that targets the understanding of the rationale behind a design decision. This approach uses a combination of model differencing and the type of change executed on the models.

7.2 Recovery of Design Decisions

A design decision is the description of a set of solutions that have been proposed to address a given software requirement [Van der Ven 2006]. For this chapter, we add the notion of time to

differentiate design decisions made at different stages of the software life cycle based on the changes detected in the version-based repositories. Hence, we define a design decision to be the set of solutions that have been proposed to address a software requirement at a particular point in time. A typical design decision often includes the design consequences and the design rationale, because they are crucial in understanding the system design and preventing knowledge vaporization in software systems [Shahbazian 2018]. The design consequences are the effect of the execution of a design decision on the system and it refers to the set of additions, deletions, and modifications that are executed on the system. It is often captured by the changes that are made to the system. The design rationale is the reason behind the adoption of a particular design choice and it reflects why the design consequences were executed.

This phase of the research discusses how we used the Change Analyzer to recover the design consequences from the evolution history of a set of Simulink models in 28 GitHub repositories. This phase also discusses how we classified the changes associated with the design decisions to help in understanding the rationale behind the design decisions. The selected repositories were chosen from the 31 repositories discussed in Section 6.4. One of the original 31 repositories was excluded because it contains only trivial changes, and two other repositories were also excluded because they were used in the evaluation discussed in Section 7.5. The remaining part of this section discusses the extraction of the consequences of the design decisions via the Change Analyzer, and the classification of the changes associated with the design decision.

7.2.1 Extracting Consequences of Design Decisions

We derive the consequences of design decisions via the extraction of changes across the version history of models. Change has always been recognized as an important factor in software evolution and many sets of metrics have been developed to quantify and analyze changes to

software systems. We adapted existing change metrics defined by Wen et al. [Wen 2004] to fit changes to systems models. In particular, we considered six operations that characterize the evolution of a set of models from version A to version B. These operations are grouped into three categories: changes to the models (adding/removing models), changes to model elements (adding/removing/reordering elements), and changes to the attributes of the model elements.

Shahbazian et al. [Shabazian 2018] showed that the consequences of the design decisions are the changes made to the software because of a design decision. Hence, we define the consequences of design decisions as the set of changes executed on the models. These changes were extracted in two phases. The first phase involves checking for the addition or deletion of models, while the second phase involves extraction of the changes in identical models via the EMFCompare extension of the Change Analyzer. In the first phase, we extracted the high-level changes to models in the repository by comparing the models via the file name property of the models that is captured by the Simulink metamodel provided by Massif. The sets of added and deleted models between each pair of adjacent versions are computed as shown in Listing 7.1.

```
Let A= set of models in prior version
Let B= set of models in following version
If A B is empty
  Added models= All B
  Deleted Models = All A
Else
  ChangedModel = (A  B) - (A   B)
  Added models = ChangedModel  B
  Deleted models =  ChangedModel  A[1]
```

Listing 7.1. Differencing for Model-Level Changes

The second phase involves extracting changes at the level of model elements and attributes via the EMFCompare extension of the Change Analyzer. Four kinds of changes (i.e., ADD, DELETE, MODIFY, or MOVE) and the affected model element/attributes were extracted from

the results produced by the Change Analyzer. The model that contained the changed elements is also added to a list of modified models.

7.2.2 Classifying Changes associated with a Design Decision

This phase of our work describes how we classify the change sets extracted from the consequences generated in the previous section. The aim of the change classification process is to support recovery of the rationale behind a design decision. Shahbazian et al. [Shahbazian 2018] attempted to recover the rationale of a design decision by mapping the changes to the issue repository based on the premise that issues contain the rationale behind a design decision if there is an associated consequence embedded in the code repository. However, most of the Simulink repositories used in our work have limited or no issue history. Figure 7.1 is a graphical overview of the number of issues found in Simulink repositories. The repositories with the highest number of issues had 24 issues while more than 60% of the repositories had no issues recorded.

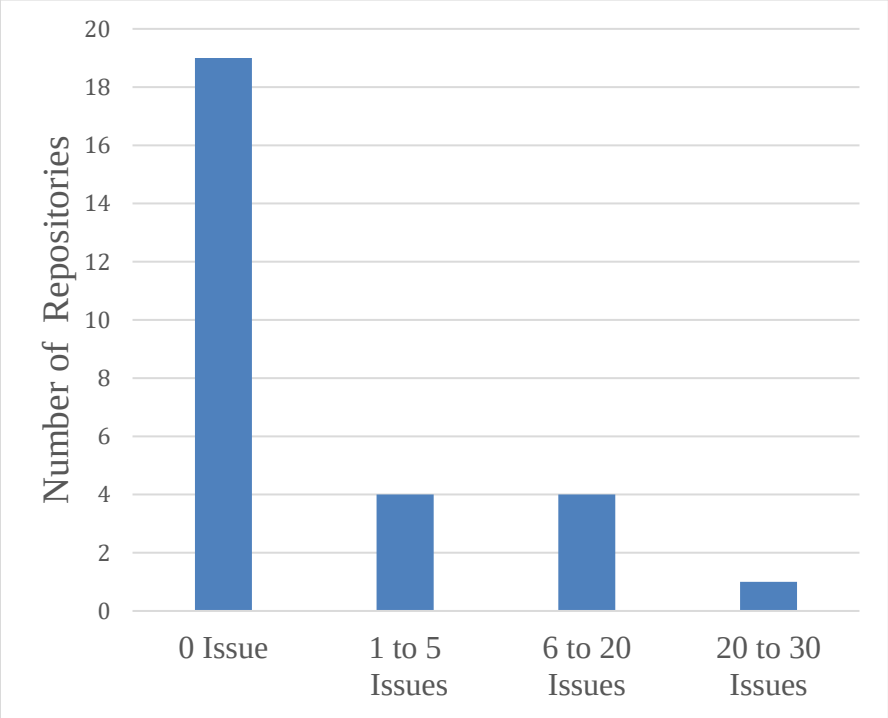

Figure 7.1. Distribution of Issue Logs in Repositories

We provide a change classification process that can help to understand the rationale behind a design decision based on the premise that the type of change executed on the software is a good indicator of the reason why the change was made [Mockus 2000]. We also provide an approach to extract the associated commit messages. Even though a commit message is user-defined and may contain trivial or unmeaningful information, a well-written commit message can offer helpful insight into why a set of changes was made [Peters 2012]. To classify a changeset, we used a set of classifiers from the Waikato Environment for Knowledge Analysis (WEKA) toolset [Frank 2009]. The WEKA toolset is open-source software that provides access to state-of-the-art machine learning and classification algorithms. These algorithms can be easily accessed by users to execute many machine-learning tasks on user-defined data. The WEKA toolset was used to classify the changes recovered from the previous section into four categories of changes associated with software maintenance. The set of classifiers used for the classification process includes *OneR* [Holte 1993], *Random Forest* [Breiman 2001], *Multi-class Classifier* [Liu 2006], C*lassification via Regression* [Frank 2009], *KStar* [Cleary 1995], *SMO* [Platt 1999], *LibSVM* [Chang 2011], *Bayes Net* [Pearl 1998], *Naive Bayes* [Langley 1992], and *J48* [Quinlan 2014]. The categories of changes considered are the corrective, adaptive, perfective and corrective type of changes defined in Section 6.5.3.

Dataset Collection and Analysis. The training data for the supervised learning algorithm was developed by manually annotating over 300 changes from 28 repositories via the process described in Section 6.5.3. For each changeset, we extracted seven metrics and matched them to the manually annotated label for that changeset. The seven metrics are the total number of elements added, total number of elements deleted, total number of elements with at least one attribute changed, total number of elements that have been reordered, total number of newly

added models, total number of models deleted, and the total number of existing models that have at least one element or attribute changed. The class labels used for the classification are the types of changes manually annotated for each changeset and they are adaptive, predictive, corrective, and preventive. It is possible that a particular commit belongs to more than one classification. Therefore, we adopt a binary classification where each commit is either classified as one of the class labels or not. This means that we have four identical files for each category: the first file classifies the changesets into adaptive or non-adaptive, and the second file divides the changesets into corrective or non-corrective. The third and fourth files also follow a similar pattern for the perfective and preventive categories.

In summary, we have four identical files for each of the change categories with over 300 changesets. This also means that the classification process was executed four times with each of the classifiers mentioned previously. Finally, each entry to the dataset used for the classification process in Weka contains seven numeric attributes, representing each of the seven extracted metrics and a class label of either positive or negative for the change category that is under consideration.

Methodology for Validating the Classifiers. A ten-fold cross-validation technique is a popular mechanism for assessing the performance of machine learning models and algorithms [Arlot 2010]. In a ten-fold cross validation process, the dataset is divided into ten equal parts (called *folds*) and ten rounds of validation are conducted. For each validation round, one part or fold of the data set is used as test data and the remaining 9 folds are used as a training set. For the next round of validation, a new fold is used as test data and the other 9 folds (including the one used as test data in the prior round) are used as the training set. The procedure is repeated ten times, with each fold used as test data once.

The ten-fold cross validation technique has been used to assess the performance of the classification algorithm used in our work. Every set of changes in the dataset has been assigned exactly one label, whose value is either a positive or negative for the change category under consideration. The dataset is divided into ten parts as per the requirement of the ten-fold cross validation technique. The training data that consists of 9 parts of the data set is used to train the classification model to identify the best set of patterns and anti-patterns of changes that best match a given classification label. The testing fold is used to validate the classification model by initially excluding the manually annotated label from the training data, then allowing the classification model to predict the appropriate change classification, and comparing the results with the initial label. In this way, the validation process mimics realistic scenarios where the training set corresponds to available data that have been manually classified by humans, and the test data correspond to the set of changes that are classified by our classification algorithm.

The results of the classification process show that some of the classifiers perform better in one category while others excel in other categories. No classifier actually performed better than other classifiers in all the change categories being considered. The average F1-score for all the classifiers across the four categories was 79%. Overall, the Random Forest classifier offers the best aggregate performance across the four categories with an average F1-score of 83.9%. This is similar to the results obtained by Pandey et al. [Pandey 2017] in their empirical study involving the classification of issue logs into corrective or non-corrective maintenance categories. The LibSVM has the worst performance with an average F-1 score of 70.9%. Furthermore, the best performance from all the classifiers was in the preventive category with an average F1-score of 94.8% while the perfective category has the worst score of 65%. Figure 7.2 summarizes the results of the classification process.

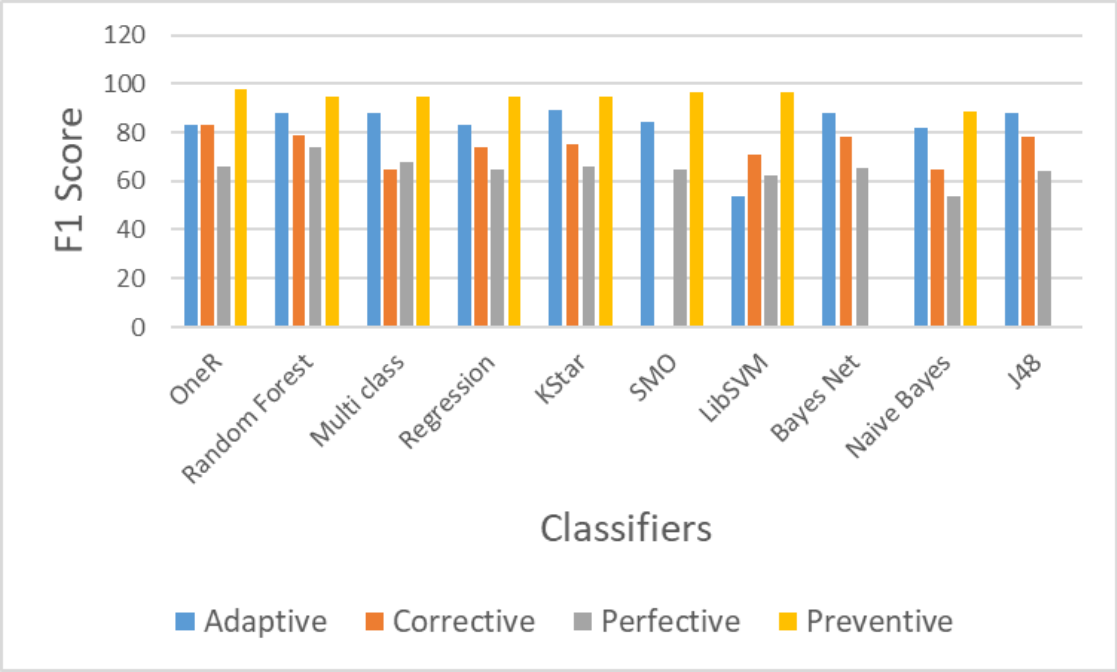

Figure 7.2. Results of the Change Classification Process

7.2.3 Results of the Design Decisions Recovery Process

Subsection 7.2.1 presented our approach to recover the design consequences, while Subsection 7.2.2 discussed how we supported reasoning of the rationale behind a design decision, in terms of the change categories generated by the learning algorithm and the commit message. This section presents the combination of the results generated from subsections 7.2.1 and 7.2.2 in order to produce the design decisions from each commit history. We identified a design decision as any version that can be classified to at least one of the change categories. While a design decision often involves substantial modifications to the models, a design decision with deep impact might require only a minor set of modifications. Hence, for this study we did not set any threshold to a set of changes in order to qualify it as a design decision, although we did provide support in our tool to set a threshold based on the length of a commit message or size of model modifications. For illustration, Table 7.1 captures the consequences and design

decisions generated for the first four versions of a sample repository. The Simulink *Model rev* column indicates the version histories that involve changes to Simulink models, and the *Repo rev.* column indicates changes to any file in the repository (i.e., commits). For example, version 1 corresponds to index 1 and version 2 corresponds to index 4. This shows that no changes were made to any Simulink file during the second and third commits. The Date column captures the time when the changes were made, while the Consequences column shows the high-level consequences of the design decision that has been extracted from the repository. The final column shows the combined change class and commit message.

Model rev.	Repo rev.	Date	Consequences	Class of Change and Commit Message
1	1	Sat Jun 09 10:52 2012	2 models added (launchpad_R2012a.mdl, launchpad.slx)	This is an adaptive change. Initial commit
2	4	Wed Jun 13 12:10 2012	1 models added (launchpad_lib.slx), 1 model modified (launchpad.slx)	This is an adaptive change. Initial implementation of serial read/write blocks
3	6	Thu Jun 14 10:06 2012	1 model modified (lauchpad.slx)	This is a perfective change. Update launchpad Simulink block library
4	10	Thu Jun 14 10:10 2012	1 model modified (launchpad_lib.slx)	This is a preventive change. Update example/test models

Table 7.1. Excerpt of the Recovered Sets of Design Decisions in a Sample Repository

7.3 Evaluation Methodology and Results

This section describes the procedure we followed to evaluate the effectiveness of our approach in classifying changes associated with design decisions. Specifically, we compared our results with the design decisions that have been extracted from the discussions in the issues reported in two large repositories. The logged issues that are reported in the issues section of the repository often contain some discussions on why a design decision is necessary, and the required change that has been executed to implement the decision. Therefore, a manual analysis of the discussion around an issue might be able to reveal the rationale behind a design decision. This makes the issues a helpful way to evaluate the performance and accuracy of our tool.

We evaluated the accuracy of our approach on two repositories: NASA T-MATS[18] (a thermodynamic modeling package), and CJT[19] (a personal repository for modeling robotic joints). These repositories were chosen because they are open source, and they have a high number of logged issues and commits in GitHub. These repositories have also been excluded from the original 28 repositories used for the training set, in order to prevent bias when evaluating the generalizability of our approach. Table 7.2 gives an overview of the repositories, with number of issues, number of issues related to Simulink models, number of commits, and application domains. Table 7.2 shows that only a small fraction of the total number of issues typically affects Simulink models. This suggests that even for the small percentage of repositories with logged issues, the manual extraction of design decisions would require the analysis of over 300% more commits and issues in order to recover design decisions that specifically involved Simulink models.

[18] https://github.com/nasa/T-MATS
[19] https://github.com/geez0x1/CompliantJointToolbox

Properties\Repositories	T-MATS	CJT
Number of issues	89	75
Number of resolved issues	86	58
Number of issues related to Simulink models	26	10
Number of commits	221	668
Domain	Thermodynamics	Robotics

Table 7.2. Overview of T-MATS and CJT Repositories

To evaluate the applicability of the change classification process, we compared our results with the issues in the repository within the same period. For example, the first version of the T-MATS repository was on Jan 31, 2014, which our framework predicted to be a corrective type of change with its associated consequences; on the same day, two issues were closed and they were labelled *correction* and *bugs*, respectively. In particular, we aim to answer the following research questions:

1. *What percentage of design decisions extracted from the issue logs were recovered by our tool?* The issue logs often provide lots of useful information about changes made during the lifecycle of a software, and many of the design decisions are usually documented in the issue logs [Shahbazian 2018]. This research question aims to evaluate the performance of our tool at detecting design decisions embedded in the version history by comparing the design decisions discovered by our tool with the design decisions that have been manually extracted from the issue logs.
2. *What is the performance of the Random Forest classifier in terms of precision, recall, and F1-score, over the labels associated with some of the design decisions in the issues*

reported in the repositories? The Random Forest classifier offers the best performance among the set of the classifiers chosen from the WEKA toolset. This research question targets the accuracy of the classified changes.

7.3.1 Recovered Design Decisions

We analysed the discussions related to Simulink models from the issue logs in order to extract the design decisions in the issue repositories. The extracted design decisions involve issues where any aspect of a Simulink model was discussed or issues that require modification to Simulink models. We were able to manually identify 36 such design decisions (26 from T-MATS and 10 from CJT) in the issue logs.

We were able to detect 52 design decisions in the version history using our tool even though only 36 were identified in the issue logs. Our tool captured 33 out of the 36 design decisions in the issue logs. The remaining design decisions not detected by our tool actually involve changes to the non-Simulink files, such as the HTML documentation, and one of the issues involves the reimplementation of a Simulink block as MATLAB code. Eighteen of the design decisions detected by our tool could not be found in the issue logs. This may mean that these 18 design decisions were either trivial or part of a larger design decision not explicitly discussed in the issue logs. Furthermore, two of the design decisions present in the issue logs span multiple commits and were represented as separate design decisions in our tool. In addition, there were eight instances where a commit contains more than one design decision extracted from the issue logs. This means that such multiple design decisions in a single commit were captured as one design decision by our tool. Furthermore, the consequences of some design decisions were part of a commit (i.e., multiple design decisions were captured in one commit).

In summary, 33 out of the 36 (92%) design decisions manually extracted from the issue logs were detected by our tool, while 18 (35%) of the design decisions detected by our tool could not be found in the issue logs. Figure 7.3 shows a brief overview of the results.

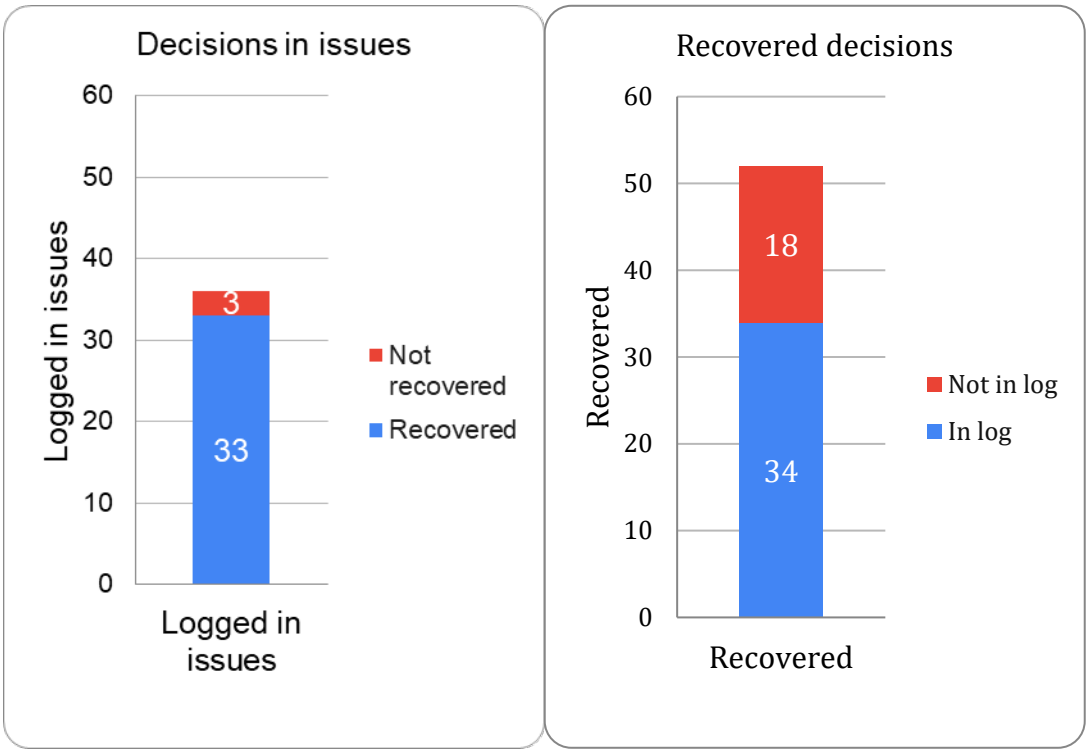

Figure 7.3. Results of the Design Decisions Recovery Process

7.3.2 Performance of the Random Forest Classifier with Respect to the Labels in the Issue Logs

The Random Forest classifier offers the best aggregate performance from the list of classifiers discussed in Section 7.2. Therefore, we used the Random Forest classifier to evaluate the accuracy of predicting the type of change. To investigate the performance of the classifier, we extracted the seven attributes discussed in Section 7.2 for each of the commits in the two repositories used for the evaluation. Then we consider the performance of our change classification algorithm with respect to the labels attached to some of the design decisions in the

issue logs. All the affected decisions have only four kinds of labels, and we matched these labels to appropriate change types. The four labels are the *correction* and *bug* labels that were matched to the corrective change type, the *clean up* label that was matched to the perfective change type and the *enhancement* label that can be either adaptive or perfective change type. It should be noted that none of the labels were associated with the preventive change category. Finally, we calculated the precision, recall and F1-score for each of the change types.

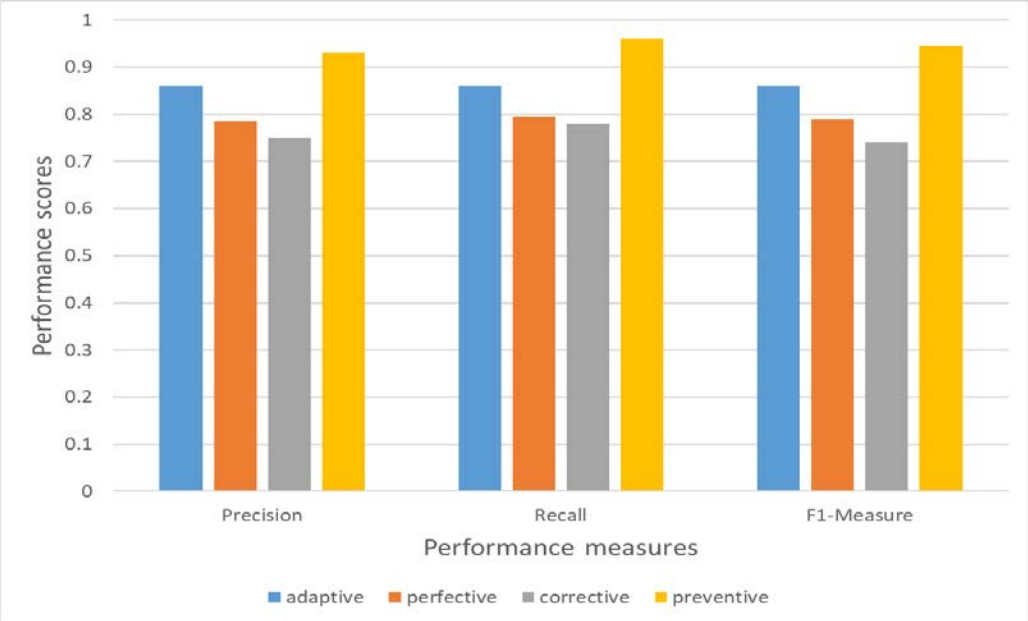

Figure 7.4. Evaluation Results for the Random Forest Classifier

Figure 7.4 gives an overview of the correctness of the prediction classification with respect to the labels in the issue logs. The figure shows that the classification algorithm performed very well for all the classes with an aggregate F-1 score of 83%. The preventive class offers the highest performance of 94.6%. This means that the random forest classifier could efficiently predict that none of the labels were associated with a preventive class. Furthermore, the corrective class offers the least performance of 74%. A manual investigation revealed that the corrective cases

that have been labelled incorrectly actually involved the addition of new features. However, the new features were added to fix a bug and not to satisfy new requirements.

7.4 Threats to Validity

There are some limitations that may affect the generalization of the results presented in this chapter. We analyse some of the threats to the validity of our results in the following paragraphs.

1. *Simulink repositories do not only contain Simulink models*. Other file formats (especially MATLAB files) are often present in the same repository. Furthermore, we only consider the commits where changes are made to any Simulink model.
2. *This work assumes that models have the same names across versions*. Hence, if a model name was changed across two versions, the tool presented in this chapter detects such scenarios as the addition of a new model in the succeeding version and the removal of another model in the previous version. Furthermore, our work primarily targets Simulink models, and the process of matching identical models uses the *file* attribute captured in the Simulink metamodel provided by the Massif tool. However, we believe that most aspects of our work should be generalizable to any EMF-based framework. We plan to address these limitations in the future by using additional VCS capabilities and metadata: Git can be asked to report renames, for instance.
3. *Although every change is a consequence of a design decision [37], there may not be a strict one-to-one mapping between a changeset and a design decision*. Design decisions may be complex, crosscutting and intertwined. Hence, it is possible that a design decision may span over many versions and a version may incorporate more than one design decision.

Despite the above limitations, we believe that our work lays a good foundation for classifying changes and recovering design decisions from the evolution history of models. Several of these limitations form the pursuits of future work.

7.5 Related Work

Many studies have been published on design decisions within the software engineering community. However, most of these studies often focus on how to capture and properly document design decisions as soon as those decisions are made. Che et al. [Che 2014] and Stevanetic et al. [Stevanetic 2015] focus on how to capture design decisions during the development phase. Che et al. proposed the use of views to represent different aspects of a design decision, while Stevanetic et al. advocated for an integration of different tools to manage the various aspects of the design phase and capture the design decisions made in those phases. Lee et al. [Lee 2008] provided a visualization tool to aid in the documentation and analysis of design decisions. Jansen et al. [Jansen 2008] introduced a conceptual framework for analysing current systems to aid in the recovery of design decisions. Their framework is able to automatically detect the changes across the software, but the analysis and rationale are not automated. The manual nature of the analysis phase makes the approach error prone, especially if the original developer is not available. Manoj et al. [Bhat 2017] developed machine learning techniques to automatically detect and categorize design decisions by manually annotating over 1500 issues from two open-source repositories. Zimmermann et al. [Zimmermann 2010] described solutions to model design decisions that can be reusable across the system life cycle.

Some studies also use MDE to model design decisions and reason about design decision processes. Malavota et al. [Malavolta 2011] introduced support for evolution of design decisions,

by capturing the impact of a new design decision on existing ones as the system evolves. Lytra et al. [Lytra 2013] used model-based transformations to ensure consistency between a design decision and the corresponding design model as the system evolves over time. Capilla et al. [Capilla 2007] also developed a modelling framework to capture the different aspects of developing and analysing design decisions with an integrated view of ensuring consistency as the system evolves. Zhu et al. [Zhu 2007] leveraged UML profiles for capturing design decisions and maintaining consistency between the design decisions and the system implementation.

The research works discussed above mostly focused on capturing and documenting design decisions as soon as they are made. They do not consider how to detect design decisions that were not documented in existing systems. The research most similar to our work was conducted by Shahbazian et al. [Shahbazian 2018] where they developed an approach to recover the consequences and rationale of a design decision via the architectural changes on the system. These changes are extracted from the code repositories and then mapped to issues obtained in the issue repository within the same period, in order to detect changes that are associated with a design decision and to recover the rationale behind the design decisions. Our work extends this approach by improving the rationale via the commit messages and classification of change types.

Some work has also been done in classifying changes to software. Ferzund et al. [Ferzund 2009], Herzig et al. [Herzig 2013], Kim et al. [Kim 2008], and Xia et al. [Xia 2016] introduced various techniques and algorithm to classify software changes as bug-free or buggy code. Ferzund et al. [Ferzund 2009] presented a metric-based approach to classify changes while Herzig et al. [Herzig 2013] classified changesets using a graph of the structural dependencies among the changesets. Kim et al. [Kim 2008] adopted a machine learning approach that uses data from the version history of a previous project, and Xia et al. [Xia 2016] used genetic

algorithms to identify defective changes. Pandey et al. [Pandey 2017] conducted an empirical study where they used various machine-learning algorithms to classify issue logs into bugs or non-buggy categories. Their results show that the Random Forest classifier offers the best performance, similar to the results reported in this chapter. These research works mainly focus on corrective maintenance activities and did not consider other types of maintenance activities.

Hassan [Hassan 2008] developed an algorithm to classify commit messages into their respective maintenance categories. Mockus et al. [Mockus 2000] also classified changes to their appropriate maintenance types via the manual classification of a change's commit message. Li et al. [Li 2019] conducted an empirical study to identify characteristics of influential changes and then developed a classifier to predict changes that are influential based on the identified characteristics. The evaluation results shows that the classifier is able to achieve an F-1 score of 80%. Hindle et al. [Hindle 2009] developed a classifier to categorize changes in a large commit using only the commit metadata such as commit message, commit authors and modified files. Their study reported that data related to the commit message and commit author is sufficient for classifying changes in a commit into their respective categories. Gharbi et al. [Gharbi 2019] and Yan et al. [Yan 2016] also uses active-learning and topic-modelling techniques respectively to classify changes belonging to multiple maintenance categories via its commit message.

The research related to software change classification have targeted text-based programming languages. Furthermore, most of these research works have focused on classifying changes to their respective maintenance types via its commit message. The research presented in this chapter extends the literature by classifying changes to models using the changeset metadata such as number of elements added, deleted, or modified. Furthermore, the research was

implemented within the context of a design decision to support the understanding of the rationale behind a design decision.

7.6 Chapter Summary

Our work introduces the notion of design decisions in model-based systems and proposes techniques to extract the consequences of design decisions behind the evolution of Simulink models in GitHub repositories. We used the Change Analyzer to mine over 300 version histories of Simulink models across 28 repositories. We recovered the consequences by extracting the changes across the version histories of the models at three hierarchical levels: model, model elements and attributes.

The chapter also introduced a change classification approach to help understand the rationale behind the extracted design decisions. The change classification process introduces supervised learning models to predict the type of change embedded in change instances between two version histories. Four types of changes were identified and a *Random Forest* classifier has been trained to predict the type of change. These change classifications and the commit messages have been combined to help understand why a design decision was implemented.

CHAPTER 8

CONCLUSION AND FUTURE WORK

This chapter summarizes the research presented in this dissertation and outlines the contributions of this work to MBSE. Finally, this chapter highlights the proposed future plan for extending the research presented in this dissertation.

8.1 Research Objectives

Chapter 1 introduced the need to address challenges related to change analysis, bad smells, and design decisions within the context of MBSE. Hence, the research objectives of this dissertation as stated in Section 1.4 were the following.

1. To develop an approach and tool for analyzing the changes across the entire version history of systems models.
2. To extend the approach to support LabVIEW and Simulink models thereby bringing software engineering research closer to systems modelling.
3. To use the approach to analyse the relationship between changes to LabVIEW and Simulink models, and the evolution of bad smells in these models.
4. To use the approach to classify changes in order to understand the rationale behind the design decisions that necessitated the evolution history of systems models.

Chapter 2 provided background information on the necessary concepts and tools that are associated with this research. The next section summarizes the contributions of this dissertation in order to satisfy the research objectives stated above.

8.2 Summary of Research Contributions

The research presented in this dissertation has helped to facilitate the analysis of changes across the entire version history of systems models. Specifically, this research has developed an MDE-based tool to provide first-class support for querying change information in systems models. The tool was then extended to support MBSE platforms such as Simulink and LabVIEW. The extended tool has been used to analyse changes in systems models in order to understand the impact of changes to the evolution of bad smells, classify changes into appropriate maintenance categories, and recover the design decisions that necessitated the evolution history of systems models. The research presented in this dissertation targets models developed via LabVIEW and Simulink modelling platforms. However, we believe that the tool and techniques discussed in this dissertation can be applicable to other textual and graphical models. The following subsections summarize how the work has satisfied each of the research objectives.

8.2.1 A Change Analyzer to Extract and Query Change Information

Chapter 3 presented a Change Analyzer that can be used to query change information across the entire version history of models. The Change Analyzer was built on top of the Hawk tool to provide first-class support for indexing, storing, and querying changes across successive versions of a model. The Change Analyzer extracts the changes generated by Hawk during a model update process, then it indexes and stores the extracted changes in a backend database. Two main components of change information were captured by the Change Analyzer; the change type and the changed elements. The Change Analyzer supports three types of changes: *add* to indicate the addition of new elements, *delete* to show the deletion of existing elements, and *modify* to indicate an update to the model's attribute or references. An EMFCompare

extension to the Change Analyzer also adds a fourth type of change called *move* to capture re-ordered elements.

A set of language constructs have also been provided to facilitate efficient querying of change information that are stored in the backend database. The language constructs allow us to create queries based on the type of change, changed elements, and summary of changes. We have also adapted two existing constructs in Hawk to support change information. The adapted constructs are *timepoint scoping* to provide support for limiting the results to user-defined timepoints or versions and *timeline constructs* to support faster execution of queries across all the version history of a model.

The Change Analyzer has been evaluated for conciseness and performance by comparing it with the Hawk tool. The evaluation results have shown that the Change Analyzer can be used to construct a more concise query that can execute faster than a query developed via the Hawk tool.

8.2.2 Extending the Change Analyzer to Support Systems Models

Chapter 4 discusses how we extended the Change Analyzer to support systems models developed via the LabVIEW and Simulink tools. Hawk directly supports models that conforms to the EMF standard. Hence, the Change Analyzer, which is an extension of the Hawk tool, only provides direct support for EMF-based models. To support LabVIEW and Simulink models, we extended the model parser component of the Change Analyzer to convert the appropriate systems models to an EMF resource.

To support LabVIEW models, we extracted the Block diagram component of the model and mapped it directly to EMF. We also developed an open-source metamodel for LabVIEW in order to validate the extracted models and facilitate querying of LabVIEW models in the

Change Analyzer. The metamodel allows the Change Analyzer to understand the structure of a LabVIEW model, provide constructs to develop queries related to element types and attributes, and support the overall management of LabVIEW models.

To provide support for Simulink models, we integrated the Change Analyzer with the Massif tool for transforming Simulink models to EMF models. We also reused the Simulink Ecore-based metamodel provided by the Massif tool to facilitate the management of Simulink models in the Change Analyzer.

8.2.3 Evolution of Bad Smells in Systems Models

Chapters 5 and 6 presented the analysis of the evolution of bad smells in LabVIEW and Simulink models, respectively. Chapter 5 discussed how we used the Change Analyzer to understand the prevalence and introduction of bad smells in LabVIEW repositories. The chapter also analyzed the relationship between the structural changes executed on models and evolution of bad smells in such models. To support the analysis mentioned earlier, seven queries were developed to detect instances of seven bad smells in 81 LabVIEW models across 10 GitHub repositories. The results showed most smells were introduced in the earlier versions of the models and these smells often persist throughout the lifecycle of the models.

Chapter 6 discussed how the Change Analyzer has been used to support the evolution of bad smells and software maintenance tasks in Simulink models. Four queries were constructed to detect instances of four bad smells in 575 Simulink models across 31 GitHub repositories. The Change Analyzer has also been used to extract changes and commit messages across all the version histories of the Simulink models. These extracted changes and commit messages have been used to manually classify changes in a version to a software maintenance task. Four types of maintenance tasks have been identified based on the ISO/IEC standard. These maintenance

tasks are the *adaptive tasks* to indicate addition of new features, *corrective tasks* to capture the repair of anomalies, *perfective tasks* that capture routine activities to ensure the optimal functioning of the systems, and *preventive tasks* that targets the prevention of future errors (e.g., testing activities). The results of the analysis shows that adaptive maintenance tasks are likely to introduce new smells to the models and decrease in smells are often associated with perfective maintenance tasks.

8.2.4 Change Classification and Design Decisions in Systems Models

Chapter 7 discussed how the Change Analyzer has been used to classify changes and recover the consequences of design decisions in Simulink models. The consequences of the design decisions are the changes that have been executed due to a design decision. The consequences generated by the Change Analyzer includes changes related to models and model elements. These changes have been classified into an appropriate maintenance type. The classified changes was combined with the commit message to provide information related to the rationale of the design decision. The kind of change (mapped to a type of maintenance task) was predicted via supervised learning models in the Weka tool environment. The design consequences and the classified changes have been combined to help understand the design decisions that necessitate the evolution history of Simulink models.

The approach discussed above has been evaluated by comparing the recovered design decisions via the Change Analyzer with a set of design decisions that were manually extracted from the issues reported within a similar timeframe. The comparison results indicate a high similarity of the recovered design decisions with the manually identified decisions, even though the manual identification process takes much more time and often does not provide additional information about the changes executed to implement the design decisions.

8.3 Perspectives on Future Work

The Change Analyzer has been used to support the analysis of bad smells, change classification and design decisions in LabVIEW and Simulink models. This section discusses how we can improve the Change Analyzer to support heterogeneous artifacts. This section also provides insights to other types of analysis that can be automated by the Change Analyzer.

8.3.1 Support for Heterogeneous Artifacts

Most software repositories contain other types of artifacts that also affect the evolution of models. For example, a typical Simulink repository would also contain MATLAB code, images, and other files. It is possible that the fix to a bug in the Simulink model will be implemented with MATLAB code. However, the current configuration of the Change Analyzer is only able to support software models. In the future, we plan to explore how the Change Analyzer can be extended to further support other systems artifacts for a more robust analysis.

8.3.2 Refactoring Analysis

Refactoring analysis is important for correcting errors, implementing new features and correcting bad smells in programs [Silva 2016, Zhang 2005]. Section 3.6.1 discussed how the Change Analyzer can be used to detect instances of the “Extract Superclass” refactoring task. In the future, we plan to support the detection of more refactoring operations and also facilitate other kinds of refactoring analysis, such as detecting potential opportunities for applying refactoring operations, exploring the relationship between refactoring and bad smells, and understanding the impact of refactoring operations.

8.3.3 Change Impact Analysis

Much work (see Section 3.7) has been done to understand the impact of changes on models. However, most of the approaches for understanding the impact of changes do not focus on the entire version history of the models. In the future, we plan to support change impact analysis from a historical perspective such as understanding how patterns of past changes can be used as a potential indicator of future change.

8.3.4 Smell Prediction

In this dissertation, we discussed how we detect instances of bad smells in systems models. Specifically, we studied the relationship between structural changes executed on models and the evolution of bad smells in models. In the future, we plan to extend this work to automatically predict if a change is likely to introduce new smells or increase the instances of smells in a model. Furthermore, we hope to develop a set of best practices on how to execute changes to models in order to limit the prevalence of bad smells.

REFERENCES

Addazi, Lorenzo, Antonio Cicchetti, Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. "Semantic-based Model Matching with EMFCompare." In *Proceedings of Models and Evolution Workshop at the 19th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pp. 40-49. 2016.

Alves, Everton LG, Myoungkyu Song, and Miryung Kim. "RefDistiller: A Refactoring Aware Code Review Tool for Inspecting Manual Refactoring Edits." In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 751-754. 2014.

Atkinson, Colin, and Thomas Kuhne. "Model-driven Development: A Metamodeling Foundation." *IEEE Software*, vol. 20, no. 5, pp. 36-41. 2003.

Arlot, Sylvain, and Alain Celisse. "A Survey of Cross-validation Procedures for Model Selection." *Statistics Surveys*, vol. 4, pp. 40-79. 2010.

Aversano, Lerina, Umberto Carpenito, and Martina Iammarino. "An Empirical Study on the Evolution of Design Smells." *Information*, vol. 11, no. 7, pp. 348-368. 2020.

Bajaj, Manas, Dirk Zwemer, Russell Peak, Alex Phung, Andrew Scott, and Miyako Wilson. "4.3. 1 Satellites to Supply Chains, Energy to Finance—SLIM for Model-Based Systems Engineering: Part 1: Motivation and Concept of SLIM." *INCOSE International Symposium*, vol. 21, no. 1, pp. 368-394. 2011.

Bajaj, Manas, Andrew Scott, Douglas Deming, Gregory Wickstrom, Mark De Spain, Dirk Zwemer, and Russell Peak. "Maestro–A Model-based Systems Engineering Environment for Complex Electronic Systems." *INCOSE International Symposium*, vol. 22, no. 1, pp. 1999-2015. 2012.

Balaban, Ittai, Frank Tip, and Robert Fuhrer. "Refactoring Support for Class Library Migration." *ACM SIGPLAN Notices*, vol. 40, no. 10, pp. 265-279. 2005

Barmpis, Konstantinos, and Dimitris Kolovos. "Hawk: Towards a Scalable Model Indexing Architecture." In *Proceedings of the 1st Workshop on Scalability in Model Driven Engineering*, pp. 1-9. 2013

Barmpis, Konstantinos, Seyyed Shah, and Dimitrios S. Kolovos. "Towards Incremental Updates in Large-Scale Model Indexes." In *Proceedings of the 11th European Conference on Modelling Foundations and Applications*, pp. 137-153. 2015.

Barmpis, Konstantinos, Antonio García-Domínguez, Alessandra Bagnato, and Antonin Abherve. "Monitoring Model Analytics over Large Repositories with Hawk and MEASURE." In *Model Management and Analytics for Large Scale Systems*, Academic Press, pp. 87-123. 2020.

Bartelt, Christian. "Consistence Preserving Model Merge in Collaborative Development Processes." In *Proceedings of the 1st International Workshop on Comparison and Versioning of Software Models*, pp. 13-18. 2008.

Bettini, Lorenzo, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. "Quality-Driven Detection and Resolution of Metamodel Smells." *IEEE Access*, vol. 7, pp. 16364-16376. 2019.

Beydoun, Ghassan, Graham Low, Brian Henderson-Sellers, Haralambos Mouratidis, Jorge J. Gomez-Sanz, Juan Pavon, and Cesar Gonzalez-Perez. "FAML: A Generic Metamodel for MAS Development." *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 841-863. 2009

Bhat, Manoj, Klym Shumaiev, Andreas Biesdorf, Uwe Hohenstein, and Florian Matthes. "Automatic Extraction of Design Decisions from Issue Management Systems: A Machine Learning Based Approach." In *Proceedings of the 11th European Conference on Software Architecture*, pp. 138-154. 2017.

Bonet, Nicolás, Kelly Garcés, Rubby Casallas, María Elsa Correal, and Ran Wei. "Influence of Programming Style in Transformation Bad Smells: Mining of ETL Repositories." *Computer Science Education*, vol. 28, no. 1, pp. 87-108. 2018.

Borrego, Gilberto, Alberto L. Morán, Ramón R. Palacio, Aurora Vizcaíno, and Félix O. García. "Towards a Reduction in Architectural Knowledge Vaporization during Agile Global Software Development." *Information and Software Technology*, vol. 112, pp. 68-82. 2019.

Bouhours, Cédric, Hervé Leblanc, and Christian Percebois. "Bad Smells in Design and Design Patterns." *The Journal of Object Technology*, vol. 8, no. 3, pp. 43-63. 2009.

Brambilla, Marco, Jordi Cabot, and Manuel Wimmer. "Model-Driven Software Engineering in Practice." *Synthesis Lectures on Software Engineering*, vol. 3, no. 1, pp. 1-207. 2017.

Breiman, Leo. "Random Forests." *Machine Learning*, vol. 45, no. 1, pp. 5-32. 2001.

Briand, Lionel., Yvan Labiche, and Leeshawn O'Sullivan. "Impact Analysis and Change Management of UML Models." In *Proceedings of the 19th International Conference on Software Maintenance*, pp. 256-265. 2003.

Capilla, Rafael, Francisco Nava, and Juan C. Duenas. "Modeling and Documenting the Evolution of Architectural Design Decisions." In *Proceedings of the 2nd Workshop on Sharing and Reusing Architectural Knowledge-Architecture, Rationale, and Design Intent*, pp. 9-9. 2007.

Carçao, Tiago (2014). "Measuring and Visualizing Energy Consumption within Software Code." In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 181–182. 2014.

Carçao, Tiago, Jácome Cunha, Joao Paulo Fernandes, Rui Pereira, and Joao Saraiva. "Energy Consumption Detection in LabVIEW." In *IEEE Symposium on Visual Languages and Human-Centric Computing Travel Support Competition.* http://www4.di.uminho.pt/jas/Research/Papers/nationalInstruments.pdf.

Chambers, Christopher, and Christopher Scaffidi. "Smell-driven Performance Analysis for End-user Programmers." *Proceedings of IEEE Symposium on Visual Languages and Human Centric Computing*, pp. 159-166. 2013.

Chang, Chih-Chung, and Chih-Jen Lin. "LIBSVM: A Library for Support Vector Machines." *ACM Transactions on Intelligent Systems and Technology*, vol. 2, no. 3, pp. 1-27. 2011.

Chatzigeorgiou, Alexander, and Anastasios Manakos. "Investigating the Evolution of Code Smells in Object-Oriented Systems." *Innovations in Systems and Software Engineering*, vol. 10, no. 1, pp. 3-18. 2014.

Che, Meiru, and Dewayne Perry, "Architectural Design Decisions in Open Software Development: A Transition to Software Ecosystems." In *Proceedings of the 23rd Australian Software Engineering Conference*, pp. 58-61. 2014.

Cho, Hyun, and Jeff Gray. "Design Patterns for Metamodels." In *Proceedings of the 11th Workshop on Domain-Specific Modelling at the ACM Conference on Systems, Programming, Languages, and Applications (SPLASH)*, pp. 25-32. 2011.

Clark, Tony, Paul Sammut, and James Willans. *Applied Metamodelling: A Foundation for Language Driven Development*. Middlesex University Research Repository. 2008.

Cleary, John, and Leonard E. Trigg. "K*: An Instance-Based Learner using an Entropic Distance Measure." In *Proceedings of the 12th International Conference on Machine Learning*, pp. 108-114. 1995.

Crisp, H. "A Family of Systems Collaborative Engineering Environment." *SIMULATION Series*, vol. 34, no. 1, pp. 159-167. 2002.

Csertán, György, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel Varró. "VIATRA: Visual Automated Transformations for Formal Verification and Validation of UML Models." In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, pp. 267-270, 2002.

De la Vara, Jose Luis, and Rajwinder Kaur Panesar-Walawege. "Safetymet: A Metamodel for Safety Standards." In *Proceedings of the 16th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pp. 69-86. 2013.

Denno, Peter O., Thomas Thurman, John Mettenburg, and Dwayne Hardy. "On Enabling a Model-based Systems Engineering Discipline." *INCOSE international Symposium*, pp.2748-2766, 2008.

Dig, Danny, Can Comertoglu, Darko Marinov, and Ralph Johnson. "Automated Detection of Refactorings in Evolving Components." In *Proceedings of the 20th European Conference on Object-Oriented Programming*, pp. 404-428. 2006.

Estefan, Jeff. "Survey of Model-based Systems Engineering (MBSE) Methodologies." *International Council on Systems Engineering - MBSE Focus Group*, vol. 25, no. 8, pp. 1-12. 2007.

Exman, Iaakov. "Linear Software Models: Standard Modularity Highlights Residual Coupling." *International Journal of Software Engineering and Knowledge Engineering*, vol. 24, no. 2, pp. 183-210. 2014.

Falcon, Jeannie. "Facilitating Modeling and Simulation of Complex Systems through Interoperable Software." *Keynote Address at the 20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. 2017.

Ferzund, Javed, Syed Nadeem Ahsan, and Franz Wotawa. "Software Change Classification using Hunk Metrics." In *Proceedings of 25th IEEE International Conference on Software Maintenance*, pp. 471-474. 2009.

Fluri, Beat, and Harald C. Gall. "Classifying Change Types for Qualifying Change Couplings." In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pp. 35-45. 2006.

Fluri, Beat, Emanuel Giger, and Harald C. Gall. "Discovering Patterns of Change Types." In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 463-466. 2008.

Foster, Stephen R., William G. Griswold, and Sorin Lerner. "WitchDoctor: IDE Support for Real-Time Auto-Completion of Refactorings." In *Proceedings of the 34th International Conference on Software Engineering*, pp. 222-232. 2012.

Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional. 2018

Fowler, Martin, Kent Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional. 1999.

Frank, Eibe, Mark Hall, Geoffrey Holmes, Richard Kirkby, Bernhard Pfahringer, Ian H. Witten, and Len Trigg. "Weka-A Machine Learning Workbench for Data Mining." In *Data Mining and Knowledge Discovery Handbook*, pp. 1269-1277. 2009.

Friedenthal, Sanford, Regina Griego, and Mark Sampson. "INCOSE Model-based Systems Engineering (MBSE) Initiative." *INCOSE International Symposium*, 2007.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. 1995.

García-Domínguez, Antonio, Bencomo, Nelly and Garcia Paucar, Luis García-Paucar. "Reflecting on the Past and the Present with Temporal Graph-Based Models." In *Proceedings of the International Workshop on Models@Runtime at the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pp. 46-55. 2018.

García-Domínguez, Antonio, Nelly Bencomo, Juan Marcelo Parra-Ullauri, and Luis Hernán García-Paucar. "Querying and Annotating Model Histories with Time-Aware Patterns." In *Proceedings of the ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems*, pp. 194-204. 2019.

Gau Pagnanelli, Christi A., Barbara J. Sheeley, and Ronald S. Carson. "Model-Based Systems Engineering in an Integrated Environment." *International Council on Systems Engineering*, pp. 633-649. 2012.

Ge, Xi, Saurabh Sarkar, and Emerson Murphy-Hill. "Towards Refactoring-Aware Code Review." In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, pp. 99-102. 2014.

GEMOC Studio. http://gemoc.org/studio. (Accessed March 2021).

Gerlitz, Thomas, Quang Minh Tran, and Christian Dziobek. "Detection and Handling of Model Smells for MATLAB/Simulink models." In *Proceedings of the International Workshop on Modelling in Automotive Software Engineering at the ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems*, pp. 13-22. 2015.

Ghannem, Adnane, Marouane Kessentini, and Ghizlane El Boussaidi. "Detecting Model Refactoring Opportunities using Heuristic Search." In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research*, pp. 175-187. 2011.

Gharbi, Sirine, Mohamed Wiem Mkaouer, Ilyes Jenhani, and Montassar Ben Messaoud. "On the Classification of Software Change Messages using Multi-Label Active Learning." In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pp. 1760-1767. 2019.

Gough, Kerry M., and Nipa Phojanamongkolkij. "Employing Model-Based Systems Engineering (MBSE) on a NASA Aeronautic Research Project: A Case Study." *Aviation Technology, Integration, and Operations Conference*, pp. 3361, 2018.

Haskins, Cecilia, Kevin Forsberg, Michael Krueger, D. Walden, and D. Hamelin. "The INCOSE Systems Engineering Handbook." *International Council on Systems Engineering*. 2006.

Hartmann, Thomas, Francois Fouquet, Matthieu Jimenez, Romain Rouvoy, and Yves Le Traon. "Analyzing Complex Data in Motion at Scale with Temporal Graphs." In *Proceedings of the 29th International Conference on Software Engineering and Knowledge Engineering*, pp. 596-601. 2017.

Hassan, Ahmed E. "Automated classification of change messages in open source projects." In *Proceedings of the 23rd ACM symposium on Applied computing*, pp. 837-841. 2008.

Herzig, Kim, Sascha Just, Andreas Rau, and Andreas Zeller. "Classifying Code Changes and Predicting Defects using Change Genealogies." Technical report *Saarland University*. 2013.

Herzig, Sebastian JI, Ahsan Qamar, and Christiaan JJ Paredis. "An Approach to Identifying Inconsistencies in Model-based Systems Engineering." *Procedia Computer Science*, vol. 28, pp. 354-362. 2014.

Hindle, Abram, Daniel M. German, Michael W. Godfrey, and Richard C. Holt. "Automatic Classification of Large Changes into Maintenance Categories." In Proceedings of the *17th International Conference on Program Comprehension*, pp. 30-39. 2009.

Holte, Robert "Very Simple Classification Rules Perform Well on Most Commonly Used Datasets." *Machine Learning*, vol. 11, no. 1, pp. 63-90. 1993.

Hooman, Jozef, Nataliya Mulyar, and Ladislau Posta. "Coupling Simulink and UML Models." In *Proceedings of Joint Symposium on Formal Techniques for Railway Management Systems, and Formal Methods for Automation and Safety in Railway and Automotive Systems*, pp. 304-311. 2004.

Horváth, Akos, István Ráth, and Rodrigo Rizzi Starr. "Massif-The Love Child of MATLAB Simulink and Eclipse." *EclipseCon NA.* 2015. https://www.eclipsecon.org/na2015/session/ massif-love-child-matlab-Simulink-and-eclipse.

Hussain, Shahid, Humaira Afzal, Muhammad Rafiq Mufti, Muhammad Imran, Amjad Ali, and Bashir Ahmad. "Mining Version History to Predict the Class Instability." *PLoS One*, vol. 14, no. 9, pp 1-21. 2019.

ISO/IEC. "International standard-ISO/IEC 14764:2006; Software Engineering- Software Lifecycle Processes and Maintenance." *International Standard Organization*, pp. 1–46. 2006.

Jansen, Anton, Jan Bosch, and Paris Avgeriou. "Documenting After the Fact: Recovering Architectural Design Decisions." *Journal of Systems and Software*, vol. 81, no. 4, pp. 536-557. 2008.

Javed, Faizan, Marjan Mernik, Jeff Gray, and Barrett R. Bryant. "MARS: A Metamodel Recovery System using Grammar Inference." *Information and Software Technology*, vol. 50, no. 9-10, pp. 948-968. 2008.

Jenkins, Z. I. "A Project-Oriented Model-Based Systems Engineering (MBSE) Approach for Naval Decision Support." PhD dissertation, The George Washington University, 2021.

Johnson, Gary. *LabVIEW Graphical Programming*. Tata McGraw-Hill Education, 1997.

Jouault, Frédéric, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. "ATL: A Model Transformation Tool." *Science of Computer Programming*, vol. 72, no. 1-2, pp. 31-39. 2008.

Karsai, Gábor, Miklos Maroti, Ákos Lédeczi, Jeff Gray, and Janos Sztipanovits. "Composition and Cloning in Modeling and Meta-modeling." *IEEE Transactions on Control Systems Technology*, vol. 12, no. 2, pp. 263-278. 2004.

Karunaratne, Manupa, Cheng Tan, Aditi Kulkarni, Tulika Mitra, and Li-Shiuan Peh. "Dnestmap: Mapping Deeply-Nested Loops on Ultra-Low Power Cgras." In *Proceedings of the 55th Annual Design Automation Conference*, pp. 1-6. 2018.

Kchaou, Dhikra, Nadia Bouassida, and Hanene Ben-Abdallah. "A MOF-based Change Meta-model." In *Proceedings of the 13th International Arab Conference on Information Technology*, pp. 134-141. 2012.

Keller, Anne, and Serge Demeyer. "Change Impact Analysis for UML Model Maintenance." In *Emerging Technologies for the Evolution and Maintenance of Software Models, IGI Global*, pp. 32-56. 2012.

Khomh, Foutse, Massimiliano Di Penta, and Yann-Gael Gueheneuc. "An Exploratory Study of the Impact of Code Smells on Software Change-Proneness." In *Proceedings of the 16th IEEE Working Conference on Reverse Engineering*, pp. 75-84. 2009.

Kim, Sunghun, James Whitehead, and Yi Zhang. "Classifying Software Changes: Clean or Buggy?." *IEEE Transactions on Software Engineering,* vol. 34, no. 2, pp. 181-196. 2008.

Kodosky, Jeffrey. *"LabVIEW."* In *Proceedings of the ACM on Programming Languages, History of Programming Languages,* vol 4, no. 78, pp. 1-54. 2020.

Kolovos, Dimitrios S., Richard F. Paige, and Fiona AC Polack. "Novel Features in Languages of the Epsilon Model Management Platform." In *Proceedings of the Workshop on Models in Software Engineering at the 30th International Conference on Software Engineering,* pp. 69-73, 2008.

Kolovos, Dimitrios S., Davide Di Ruscio, Alfonso Pierantonio, and Richard F. Paige. "Different Models for Model Matching: An Analysis of Approaches to Support Model Differencing." In *Proceedings of the Workshop on Comparison and Versioning of Software Models at the 31st International Conference on Software Engineering,* pp. 1-6. 2009.

Kowalewski, Thomas Gerlitz1 Norman Hansen1Christian Dernehl1Stefan. "Artshop: A Continuous Integration and Quality Assessment Framework for Model-Based Software Artifacts." In *Tagungsband des Dagstuhl-Workshops*, pp. 13. 2016.

Langley, Pat, Wayne Iba, and Kevin Thompson. "An Analysis of Bayesian Classifiers." In *Proceedings of the 10th National Conference on Artificial Intelligence,* pp. 223-228. 1992.

Lee, Larix, and Philippe Kruchten. "A Tool to Visualize Architectural Design Decisions." In *Proceedings of the International Conference on the Quality of Software Architectures*, pp. 43-54. 2008.

Li, Daoyuan, Li Li, Dongsun Kim, Tegawendé F. Bissyandé, David Lo, and Yves Le Traon. "Watch Out for this Commit! A Study of Influential Software Changes." *Journal of Software: Evolution and Process,* vol. 31, no. 12, pp. 2181. 2019.

Lin, Yuehua, Jeff Gray, and Frédéric Jouault. "DSMDiff: A Differentiation Tool for Domain-Specific Models." *European Journal of Information Systems,* vol. 16, no. 4, pp. 349-361. 2007.

Lin, Heng-You, Seppo Sierla, Nikolaos Papakonstantinou, Anatoly Shalyto, and Valeriy Vyatkin. "Change Request Management in Model-Driven Engineering of Industrial Automation Software." In *Proceedings of the 13th International Conference on Industrial Informatics*, pp. 1186-1191. 2015.

Liu, Yiguang, Zhisheng You, and Liping Cao. "A Novel and Quick SVM-Based Multi-Class Classifier." *Pattern Recognition,* vol. 39, no. 11, pp. 2258-2264. 2006.

López-Fernández, Jesús , Jesús Sánchez Cuadrado, Esther Guerra, and Juan De Lara. "Example-driven meta-model development." *Software and Systems Modeling*, vol. 14, no. 4, pp. 1323-1347. 2015.

Ludewig, Jochen. "Models in Software Engineering–An Introduction." *Software and Systems Modeling*, vol. 2, no. 1, pp. 5-14. 2003.

Lytra, Ioanna, Huy Tran, and Uwe Zdun. "Supporting Consistency between Architectural Design Decisions and Component Models through Reusable Architectural Knowledge Transformations." In *Proceedings of the 7th European Conference on Software Architecture*, pp. 224-239. 2013.

Malavolta, Ivano, Henry Muccini, and V. Smrithi Rekha. "Supporting Architectural Design Decisions Evolution through Model Driven Engineering." In *Proceedings of the 3rd International Workshop on Software Engineering for Resilient Systems*, pp. 63-77. 2011.

Malone, Robert, Brittany Friedland, John Herrold, and Daniel Fogarty. "Insights from Large Scale Model Based Systems Engineering at Boeing." *International Council on Systems Engineering*, vol. 26, no. 1, pp. 542-555. 2016.

Mathworks. "Company Overview." https://www.mathworks.com/content/dam/mathworks/handout/2020-company-factsheet-8-5x11-8282v20.pdf. (Accessed: March 2021)

Mockus, Audris, and Lawrence G. Votta. "Identifying Reasons for Software Changes using Historic Databases." In *Proceedings of the IEEE International Conference on Software Maintenance*, pp. 120-130. 2000.

Montgomery, Paul R. "Model-Based System Integration (MBSI)–Key Attributes of MBSE from the System Integrator's Perspective." *Procedia Computer Science*, vol. 16, pp. 313-322. 2013.

Müller, Klaus, and Bernhard Rumpe. "A model-based approach to impact analysis using model differencing." *ArXiv preprint arXiv:1406.6834.* 2014.

Murgia, Alessandro, Giulio Concas, Roberto Tonelli, Marco Ortu, Serge Demeyer, and Michele Marchesi. "On the Influence of Maintenance Activity Types on the Issue Resolution Time." In *Proceedings of the 10th international conference on predictive models in software engineering*, pp. 12-21. 2014.

Nejati, Shiva, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela Zave. "Matching and Merging of Statecharts Specifications." In *Proceedings of the 29th International Conference on Software Engineering*, pp. 54-64. 2007.

Olbrich, Steffen, Daniela S. Cruzes, Victor Basili, and Nico Zazworka. "The Evolution and Impact of Code Smells: A Case Study of Two Open Source Systems." In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, pp. 390-400. 2009.

Opdyke, William. "Refactoring Object-Oriented Frameworks." PhD dissertation. University of Illinois at Urbana-Champaign. 1992.

Palomba, Fabio, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. "Detecting Bad Smells in Source Code using Change History Information." In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 268-278. 2013.

Palomba, Fabio, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. "A Large-Scale Empirical Study on the Lifecycle of Code Smell Co-Occurrences." *Information and Software Technology*, vol. 99, pp. 1-10. 2018.

Palomba, Fabio, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. "On the Diffuseness and the Impact on Maintainability of Code Smells: A Large Scale Empirical Investigation." *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188-1221. 2018b.

Paige, Richard, Dimitrios Kolovos, Louis Rose, Nicholas Drivalos, and Fiona Polack. "The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering." In *Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems*, pp. 162-171. 2009.

Paige, Richard, Dimitrios Kolovos, and Fiona Polack. "A Tutorial on Metamodelling for Grammar Researchers." *Science of Computer Programming*, vol. 96, pp. 396-416. 2014.

Paige, Richard F., Nicholas Matragkas, and Louis M. Rose. "Evolving Models in Model-Driven Engineering: State-of-the-Art and Future Challenges." *Journal of Systems and Software*, vol. 111, pp. 272-280. 2016.

Paixao, Matheus, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. "Are Developers Aware of the Architectural Impact of their Changes?." In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 95-105. 2017.

Pandey, Nitish, Debarshi Kumar Sanyal, Abir Hudait, and Amitava Sen. "Automated Classification of Software Issue Reports using Machine Learning Techniques: An Empirical Study." *Innovations in Systems and Software Engineering*, vol. 13, no. 4, pp. 279-297. 2017.

Pearl, Judea. "Bayesian Networks." In *The Handbook of Brain Theory and Neural Networks*, MIT Press, pp. 149-153. 1998.

Peters, Ralph, and Andy Zaidman. "Evaluating the Lifespan of Code Smells using Software Repository Mining." In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, pp. 411-416. 2012.

Platt John, "Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines," In *Advances in Kernel Methods—Support Vector Learning*, MIT Press, pp. 185–208, 1999.

Popoola, Saheed, and Jeff Gray. "A LabVIEW Metamodel for Automated Analysis." In *Proceedings of the International Conference on Computational Science and Computational Intelligence*, pp. 1127-1132. 2019.

Prete, Kyle, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. "Template-based Reconstruction of Complex Refactorings." In *Proceedings of the IEEE 26th International Conference on Software Maintenance*. pp. 1-10. 2010.

Qamar, Ahsan, Carl During, and Jan Wikander. "Designing Mechatronic Systems: A Model-based Perspective, an Attempt to Achieve SysML-MATLAB/Simulink Model Integration." In *Proceedings of the International Conference on Advanced Intelligent Mechatronics*, pp. 1306-1311. 2009.

Quilleré, Fabien, Sanjay Rajopadhye, and Doran Wilde. "Generation of Efficient Nested Loops From Polyhedra." *International Journal of Parallel Programming*, vol. 28, no. 5, pp. 469-498. 2000.

Quinlan, Ross. *C4. 5: Programs for Machine Learning*. Elsevier. 2014.

Rajabi, Bassam Atieh, and Sai Peck. "Change Management Framework to Support UML Diagrams Changes." *The International Arab Journal of Information Technology, vol. 16, no. 4*. pp. 720-730. 2019.

Rapos, Eric, and James Cordy. "SimPact: Impact Analysis for Simulink Models." In *Proceedings of the IEEE 33rd International Conference on Software Maintenance and Evolution.* pp. 489-493. 2017.

Reichmann, Clemens, Markus Kiihl, Philipp Graf, and K. D. Muller-Glaser. "GeneralStore: A CASE-tool Integration Platform Enabling Model Level Coupling of Heterogeneous Designs for Embedded Electronic Systems." In *Proceedings of the 11th IEEE International Conference on the Engineering of Computer-Based Systems*, pp. 225-232. 2004.

Selic, Bran. "The Pragmatics of Model-Driven Development." *IEEE Software*, vol. 20, no. 5, pp.19-25. 2003.

Services, E. (n.d.). Companies using LabVIEW. https://enlyft.com/tech/products/labview. (Accessed: March 2021)

Shahbazian, Arman, Youn Kyu Lee, Duc Le, Yuriy Brun, and Nenad Medvidovic. "Recovering Architectural Design Decisions." In *Proceedings of the IEEE 15th International Conference on Software Architecture*, pp. 95-9509. 2018.

Shahin, Mojtaba, Peng Liang, and Zengyang Li. "Do Architectural Design Decisions Improve the Understanding of Software Architecture? Two Controlled Experiments." In *Proceedings of the 22nd International Conference on Program Comprehension*, pp. 3-13. 2014.

Shi, Jianlin, Martin Törngren, David Servat, Carl-Johan Sjöstedt, DeJiu Chen, and Henrik Lönn. "Combined usage of UML and Simulink in the Design of Embedded Systems: Investigating Scenarios and Structural and Behavioural Mapping." In *Proceedings of the 4th Workshop of Object-oriented Modeling of Embedded Real-time Systems*, 2007.

Sidhu, Brahmaleen K., Kawaljeet Singh, and Neeraj Sharma. "A Catalogue of Model Smells and Refactoring Operations for Object-Oriented Software." In *Proceedings of 2nd International Conference on Inventive Communication and Computational Technologies*. pp. 313-319. 2018.

Silva, Danilo, Nikolaos Tsantalis, and Marco Tulio Valente. "Why We Refactor? Confessions of GitHub Contributors." In *Proceedings of the 24th ACM International Symposium on Foundations of Software Engineering*, pp. 858-870. 2016.

Simulink, "MATLAB Simulink." [Online]. Available: https://www.mathworks.com/products/simulink.html

Steinberg, Dave, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

Stephan, Matthew, and James R. Cordy. "Identification of Simulink Model Antipattern Instances using Model Clone Detection." In *Proceedings of the ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems*, pp. 276-285. 2015.

Stevanetic, Srdjan, Konstantinos Plakidas, Tudor B. Ionescu, Fei Li, Daniel Schall, and Uwe Zdun. "Tool Support for the Architectural Design Decisions in Software Ecosystems." In *Proceedings of the 9th European Conference on Software Architecture Workshops*, pp. 1-6. 2015.

Stevens, Reinout, Coen De Roover, Carlos Noguera, and Viviane Jonckers. "A History Querying Tool and its Application to Detect Multi-Version Refactorings." In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering*, pp. 335-338. 2013.

Tahmid, Ahmad, Nadia Nahar, and Kazi Sakib. "Understanding the Evolution of Code Smells by Observing Code Smell Clusters." In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, vol. 4, pp. 8-11. 2016.

Telea, Alexandru, and Lucian Voinea. "Visual Software Analytics for the Build Optimization of Large-Scale Software Systems." *Computational Statistics*, vol. 26, no. 4, pp. 635. 2011.

Toulmé, Antoine, and I. Inc. "Presentation of EMFCompare Utility." In *Eclipse Modeling Symposium*, pp. 1-8. 2006.

Tran, Quang Ming, and Dziobek Christian. "Approach to Constructing and Maintaining Simulink Models Based on the use of Transformation/ Refactoring and Generation Operations." In *Dagstuhl Workshops – Model Based Engineering of Embedded Systems*, pp. 1–12. 2013.

Treude, Christoph, Stefan Berlik, Sven Wenzel, and Udo Kelter. "Difference Computation of Large Models." In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering*, pp. 295-304. 2007.

Tsantalis, Nikolaos, Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig. "Accurate and Efficient Refactoring Detection in Commit History." In *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering*, pp. 483-494. 2018.

Tsantalis, Nikolaos, Ameya Ketkar, and Danny Dig. "RefactoringMiner 2.0." In *IEEE Transactions on Software Engineering (Early Access)*, 2020. doi: 10.1109/TSE.2020.3007722.

Tufano, Michele, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. "When and Why Your Code Starts to Smell Bad." In *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 403-414. 2015.

Tyree, Jeff, and Art Akerman. "Architecture Decisions: Demystifying Architecture." In *IEEE Software*, vol. 22, no. 2, pp. 19-27. 2005

Van der Ven, Jan Salvador, Anton GJ Jansen, Jos AG Nijhuis, and Jan Bosch. "Design Decisions: The Bridge between Rationale and Architecture." In *Rationale Management in Software Engineering*, pp. 329-348. 2006.

Van Emden, Eva, and Leon Moonen. "Java Quality Assurance by Detecting Code Smells." In *Proceedings of the 9th Working Conference on Reverse Engineering*, pp. 97-106. 2002.

Van Heesch, Uwe, and Paris Avgeriou. *A Pattern-Based Approach against Architectural Knowledge Vaporization*. University of Groningen, Johann Bernoulli Institute for Mathematics and Computer Science. 2009.

Völter, Markus, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley and Sons. 2013.

Wang, Haoqi, Vincent Thomson, and Chengtong Tang. "Change Propagation Analysis for System Modeling using Semantic Web Technology." *Advanced Engineering Informatics, vol. 35*. pp. 17-29. 2018.

Wen, Zhihua, and Vassilios Tzerpos. "An Effectiveness Measure for Software Clustering Algorithms." In *Proceedings of the IEEE 12th International Workshop on Program Comprehension,* pp. 194-203. 2004.

Xia, Xin, David Lo, Xinyu Wang, and Xiaohu Yang. "Collective Personalized Change Classification with Multiobjective Search." *IEEE Transactions on Reliability*, vol. 65, no. 4, pp.1810-1829. 2016.

Xing, Zhenchang, and Eleni Stroulia. "UMLDiff: An Algorithm for Object-Oriented Design Differencing." In *Proceedings of the IEEE/ACM 20th International Conference on Automated Software Engineering*, pp. 54-65. 2005.

Xue, Dingyü, and YangQuan Chen. *Modeling, Analysis and Design of Control Systems in MATLAB and Simulink*. World Scientific Publishing. 2015.

Yadav, Jitender, and Arvind Raizada. "Systems Engineering Challenges in Simulation Based Aviation Training: A Few Insights." In *International Council on Systems Engineering*, vol. 26, no. 1, pp. 177-192. 2016.

Yan, Meng, Ying Fu, Xiaohong Zhang, Dan Yang, Ling Xu, and Jeffrey D. Kymer. "Automatically Classifying Software Changes via Discriminative Topic Model: Supporting Multi-Category and Cross-Project." *Journal of Systems and Software*, no. 113, pp. 296-308. 2016.

Yohannis, Alfa, Dimitris Kolovos, and Fiona Polack. "Turning models inside out." In *3rd Flexible MDE Workshop at the 20th International Conference on Model Driven Engineering Languages and Systems*, pp. 430-434. 2017.

Yohannis, Alfa, Horacio Hoyos Rodriguez, Fiona Polack, and Dimitris Kolovos. "Towards efficient comparison of change-based models." *Journal of Object Technology,* vol. 18, no. 2, pp. 1-21. 2019.

Zaman, Shahed, Bram Adams, and Ahmed E. Hassan. "A Qualitative Study on Performance Bugs." In *Proceedings of the IEEE 9th Working Conference on Mining Software Repositories*, pp. 199-208. 2012.

Zhang, Jing, Yuehua Lin, and Jeff Gray. "Generic and Domain-Specific Model Refactoring using a Model Transformation Engine." In *Model-Driven Software Development*, pp. 199-217. 2005.

Zhao, Xin, and Jeff Gray. "BESMER: An Approach for Bad Smells Summarization in Systems Models." In *Proceedings of the Models and Evolution Workshop at the ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems*, pp. 304-313. 2019.

Zhu, Liming, and Ian Gorton. "UML Profiles for Design Decisions and Non-Functional Requirements." In *Proceedings of the 2nd Workshop on Sharing and Reusing Architectural Knowledge-Architecture, Rationale, and Design Intent at the 29th International Conference on Software Engineering*, pp. 8-8. 2007.

Zimmermann, Olaf. "Architectural Decisions as Reusable Design Assets." *IEEE Software*, vol. 28, no. 1, pp. 64-69. 2010.

APPENDIX A

QUERIES TO DETECT BAD SMELLS IN LABVIEW MODELS

This appendix contains the seven queries that were used to identify instances of bad smells in LabVIEW models. This appendix was referenced at the end of Section 5.4.

A.1 Large Variables.

```
var x = 0;
var me:Set;
for (i in Model.allInstances){
    if (i.isKindOf(NodeLabel)){
        var t= Model.allInstances.select(t:DataAccessor| t.Label==i.AttachedTo);
        if(t.size>0){
            for(j in i.`p.Text`){
                if(me.includes(j)){
                    x=x+1;
                }
                else{
                    me.add(j);
                }
            }
        }

    }
}
return x;
```

Listing A.1. Query to Detect Instances of Large Variables Smell

A.2. No Wait in a Loop

```
var i=0;
var j=0;
for (t in WhileLoop.allInstances){
    if (t.Wait.size()>0){
        continue;
    }
     else{
         i= i+getRecursiveWait(t);
     }
}
return i;

// recursive function to check for Wait
operation getRecursiveWait(element:Any): Integer{
    // check if the element as any children (containment references)
    if (Model.getChildren(self).size()<1){
        return 1;
    }
    //check if the element has a Wait
    else if (element.Wait.size()>0){
        return 0;
     }
     else{
      // recursively go through element's children and see if it has a Wait
        for (j in Model.getChildren(self)){
            if(getRecursiveWait(j)==0){
                return 0;
            }
        }

         //return 1 if none of the children has a Wait
        return  1;
    }
}
```

Listing A.2. Query to Detect Instances of No Wait in a Loop Smell

A.3 Build Array in a Loop

```
var i=0;
var j=0;
for (t in WhileLoop.allInstances){
    if (t.BuildArray.size()>0){
        i=i+1;
    }
    else{
         i=i+getRecursiveBuild(t);
    }

}
return i;

// recursive function to check for BuildArray
operation getRecursiveBuild(element:Any): Integer{
    // check if the element as any children (containment references)
    if (Model.getChildren(self).size()<1){
        return 0;
    }
    //check if the element has a Build array
    else if (element.BuildArray.size()>0){
        return 1;
    }
    else{
        // recursively go through element's children and see if it has a BuildArray
        for (j in Model.getChildren(self)){
            if(getRecursiveBuild(j)==1){
                return 1;
            }
        }

        //return 0 if none of the children has a BuildArray
        return 0;
    }
}
```

Listing A.3. Query to Detect Instances of Build Array in a Loop Smell

A.4 Property Nodes

```
return PropertyNode.allInstances.size;
```

Listing A.4. Query to Detect Instances of Excessive Property Nodes Smell

A.5 String Concatenation in a Loop

```
var x=0;
for (i in WhileLoop.allInstances){
    if(i.ConcatenateStrings.size()>0){
        x=x+1;
    }
    else{
        x=x+getRecursiveString(i);
    }
}
return x;

// recursive function to check for ConcatenateStrings
operation getRecursiveString(element:Any): Integer{
     // check if the element as any children (containment references)
     if (Model.getChildren(self).size()<1){
        return 0;
    }
    //check if the element has a Concatenate String
    else if (element.ConcatenateStrings.size()>0){
        return 1;
    }
    else{
        // recursively go through element's children and see if it has a ConcatenateStrings
        for (j in Model.getChildren(self)){
            if(getRecursiveString(element)==1){
                return 1;
            }
        }
        //return 0 if none of the children has a ConcatenateStrings
        return 0;
    }
}
```

Listing A.5. Query to Detect Instances of String Concatenation in a Loop Smell

A.6 Multiple Nested Loop

```
var x=0;
for (i in WhileLoop.allInstances()){
    if(i.WhileLoop.size()>0 or i.ForLoop.size()>0){
        x=x+1;
    }
    else{
        x= x+ getRecursiveLoop(i);
    }
}
return x;

// recursive function to check for loops
operation getRecursiveLoop(element:Any): Integer{
    // check if the element as any children (containment references)
    if (Model.getChildren(self).size()<1){
        return 0;
    }
    //check if the element has a while loop or for loop
    else if (element.WhileLoop.size()>0 or element.ForLoop.size()>0){
        return 1;
    }
    else{
        // recursively go through element's children and see if it has a loop'
        for (j in Model.getChildren(self)){
            if(getRecursiveLoop(j)==1){
                return 1;
            }
        }
        //return 0 if none of the children has a loop
        return 0;
    }
}
```

Listing A.6. Query to Detect Instances of Multiple Nested Loop Smell

A.7 Deeply Nested Subsystem Hierachy

```
var x=0;
for (i in MethodCall.allInstances){
    var name=i.Target;
    var back= '\\\\';
    var nm =name.replace(back,"");
    for (j in BlockDiagram.allInstances.select(t|t.file == nm)){
        if (j.MethodCall.size>0){
            x=x+1;
        }
    }
}
return x;
```

Listing A.7. Query to Detect Instances of Deeply Nested Subsystem Hierarchy Smell

APPENDIX B

QUERIES TO DETECT BAD SMELLS IN SIMULINK MODELS

This appendix lists the four queries that were used to detect instances of bad smells in Simulink models. The appendix was referenced in Section 6.3.

B.1 Superfluous Subsystem

```
var x=0;
for(y in SubSystem.allInstances){
    if (y.subBlocks.size<1){
        x=x+1;
        continue;
    }
    var a=0;
    var b=0;
    for (j in y.subBlocks){
        if(j.isKindOf(VirtualBlock)){
            a=a+1;
        }
        else{
            b=b+1;
        }
    }
    if(a>0 and b< 2){
        x=x+1;
    }
}
return x;
```

Listing B.1. Query to Detect Instances of Superfluous Subsystem Smell

B.2 Long Port List

```
var x=0;
for(y in SubSystem.allInstances){
    if(y.ports.size>0){
        var a=0;
        var b=0;
        for(j in y.ports){
            if(j.isKindOf(InPort)){
                a=a+1;
            }
            else if (j.isKindOf(OutPort)){
                b=b+1;
            }
        }
        if (a>2 or  b> 2){
            x=x+1;
        }
    }
}
return x;
```

Listing B.2. Query to Detect Instances of Long Port List Smell

B.3 Deeply Nested Subsystem

```
var x=0;
for(y in SubSystem.allInstances){
    for (a in y.subBlocks){
        if (a.isKindOf(SubSystem)){
            for(b in a.subBlocks){
                if (b.isKindOf(SubSystem)){
                    for (c in b.subBlocks){
                        if (c.isKindOf(SubSystem)){
                            x=x+1;
                            continue;
                        }
                    }
                }
            }
        }
    }
}
return x;
```

Listing B.3. Query to Detect Instances of Deeply Nested Subsystem Smell

B.4 Superfluous Bus Signal

```
var x=0;
for (i in BusCreator.allInstances){
    for (j in i.parameters.select(t| t.name='InputSignalNames')){
        j.value.println();
        if(j.value.split(',').size <3){
            x=x+1;
        }
    }
}
return x;
```

Listing B.4. Query to Detect Instances of Superfluous Bus Signal Smell

APPENDIX C

OVERVIEW OF SIMULINK REPOSITORIES

This appendix provides a brief overview of the 31 Simulink repositories that have used for the analysis of bad smells and design decisions discussed in Chapters 6 and 7. This appendix have been referenced in Section 6.4.

Index	URL	# Commits	# Versions	Domain	# Models	# Model Elements	# Branches	Contributors
Repo 1	https://github.com/CUGravity/ACS-Full-System-Simulation	135	42	space	8	207673	1	8
Repo 2	https://github.com/nasa/SIL	37	10	space	44	79664	4	2
Repo 3	https://github.com/analogdevicesinc/MathWorks_tools	589	24	electronics	23	138858	48	6
Repo 4	https://github.com/microgrid/Simulink-microgrid	55	11	electronics	3	149653	3	3
Repo 5	https://github.com/CPFL/Autoware_Toolbox	109	11	automotive	8	9931	1	5
Repo 6	https://github.com/stozaki-mathworks/SetInheritedPortName-Simulink-Utility	36	6	Simulink library	10	48583	2	1

Repo 7	https://github.com/tue-robotics/simulink_models	20	11	robotics	3	22725	1	1
Repo 8	https://github.com/petercorke/machinevision-toolbox-matlab	370	9	machine vision	6	22725	2	1
Repo 9	https://github.com/nasa/T-MATS	225	4/26	thermo dynamics	91	691776	1	5
Repo 10	https://github.com/verivital/slsf_randgen	217	16	cyber physical	12	22618	6	3
Repo 11	https://github.com/Smithsonian/swarm-firmware	311	4	astrophysics	1	118538	19	3
Repo 12	https://github.com/coco-team/cocoSim2	963	3	safety	174	830193	14	7
Repo 13	https://github.com/petercorke/robotics-toolbox-matlab	1168	10	robotics	23	194239	8	4
Repo 14	https://github.com/bdrozdenko/ieee80211aZynq	48	18	electronics	7	68449	1	1
Repo 15	https://github.com/luojunxun/simulink2	188	90	others	5	5592	1	1
Repo 16	https://github.com/darrahts/uavModel	131	42	avionics	2	8815	1	1
Repo 17	https://github.com/dostaji4/EnergyPlus-co-simulation-toolbox	72	10	power	5	8184	1	1
Repo 18	https://github.com/forresti/ee249_lab2	277	19	avionics	2	17693	1	2
Repo 19	https://github.com/SchaplM/robotics-toolbox	315	11	robotics	25	291511	1	1

Repo 20	https://github.com/courtin/16_82_Simulink_2019	94	50	avionics	15	52618	13	4
Repo 21	https://github.com/msiplab/EmbVision	54	9	machine vision	4	133349	1	1
Repo 22	https://github.com/simenandresen/Simulink-Underwater-Robotics-Simulator	37	20	automotive	28	233898	1	1
Repo 23	https://github.com/roslovets/GMP	31	10	robotics	14	30842	1	1
Repo 24	https://github.com/aa4cc/ert_linux	62	7	Linux	4	4259	2	3
Repo 25	https://github.com/kyak/launchpad_ert	87	21	electronics	11	47060	1	1
Repo 26	https://github.com/cookacounty/simulink-utils	22	3	Simulink library	1	911	1	1
Repo 27	https://github.com/voldemoriarty/mrac	34	19	others	10	61552	1	1
Repo 28	https://github.com/orhuntas/threephaseinverter	40	29	power	12	192121	1	1
Repo 29	https://github.com/Henricus-Basien/F16Sim	66	9	avionics	4	124612	1	2
Repo 30	https://github.com/pa872d/ArduCopterSim	57	24	avionics	12	226441	2	2
Repo 31	https://github.com/geez0x1/CompliantJointToolbox	668	16	robotics	8	208954	7	2

Table C.1. List of Selected Simulink Repositories