JOT
JOURNAL OF
OBJECT TECHNOLOGY

Evolution of Bad Smells in LabVIEW Graphical Models

Saheed Popoola, Xin Zhao, and Jeff Gray
University of Alabama, Tuscaloosa, USA

ABSTRACT Bad smells often indicate potential problems in software, which may lead to long-term challenges and expensive maintenance efforts. Although bad smells often occur in source code, bad smells also exist in representations of design descriptions and models. We have observed that many users of graphical modeling environments (e.g., LabVIEW) are systems engineers who may not be aware of core software engineering techniques, such as refactoring of bad smells. Systems engineers often focus on implementation correctness and may be unaware of how their designs affect long-term maintenance properties that may increase design smells. There exists a large body of research focused on analysing bad smells embedded in the source code of textual languages, but there has been limited research on bad smells in systems models of graphical languages. In this paper, we present a semi-automated approach for extracting design smells across versions of LabVIEW graphical models through user-defined queries. We describe example queries that highlight the emergence of design smells that we discovered from posts in the LabVIEW user's forum. We then demonstrate the use of the example queries in understanding the evolution of seven bad smells we found in 81 LabVIEW models stored in 10 GitHub repositories. We analyze the evolution of these smells in order to understand the prevalence and introduction of bad smells, as well as the relationship between bad smells and the structural changes made to the models. Our results show that all of the models contain instances of at least one type of bad smell and the number of smells fluctuates as the size of a model increases. Furthermore, the majority of the structural changes across different versions of LabVIEW models involve the addition of new elements with a corresponding increase in the presence of design smells. This paper 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.

KEYWORDS LabVIEW models, bad smells, user queries.

1. 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 (Roberts et al. 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 et al. 2012). Therefore, bad smells represent a major challenge to the quality and maintenance of large and complex software systems.

Software repositories usually provide a rich source of historical data related to software analysis and quality evaluation due to the version histories of software systems preserved in these repositories. Historical data in a software repository also depicts the evolution of software design and the various refactoring activities that have been performed over the life cycle of a system. Hence, it is not surprising that much research has been devoted to extracting, manipulating, and analysing the data embedded in repositories in order to support software maintenance, detect and refactor bad software designs, and conduct empirical

JOT reference format:
Saheed Popoola, Xin Zhao, and Jeff Gray. *Evolution of Bad Smells in LabVIEW Graphical Models.* Journal of Object Technology. Vol. 20, No. 1, 2021. http://dx.doi.org/10.5381/jot.2021.20.1.a1

validation of new research techniques (Robbes et al. 2017).

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 et al. 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 & Manakos 2014; Tahmid et al. 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.

Graphical models have been used for many decades in software engineering to capture concepts within the problem space, and abstract external complexities such as infrastructure dependencies or programming environments that simplify software design concepts (Ludewig 2003). The graphical nature of models also aids in visualizing different aspects of a system, thereby easing the learning curve for developing new models to capture essential system properties. As a result, paradigms such as Model-Based Systems Engineering (MBSE) have been developed to promote the use of models as first-class artifacts for virtually any software development task such as implementation, testing, and analysis. The analysis of systems models is very important because a model primarily abstracts the design concepts of a system to be developed; therefore, design problems such as bad smells may emerge in the model used to represent a system. Unfortunately, there is limited research on the evolutionary analysis of bad smells in graphical models and systems developed via graphical 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 15,000 companies all over the world (Services n.d.; Falcon 2017). LabVIEW provides a graphical programming environment for developing test 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 & Scaffidi 2013).

This paper introduces an approach for analysing the evolution of bad smells in LabVIEW graphical models via user-defined queries. We focus our work on LabVIEW because it has a large number of industrial and academic users. In this paper, we describe an approach for mining LabVIEW models in software repositories via user-defined queries to detect and analyse the presence of four of the design smells identified by end users in (Zhao & Gray 2019). We extend a tool named Hawk (Barmpis & Kolovos 2014)(García-Domínguez et al. 2019) to query histories of LabVIEW models stored in GitHub repositories. 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[1]. 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 paper 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.

The remainder of the paper is organized as follows: Section 2 offers an overview of bad smells across graphical models and text-based programs, while Section 3 provides an overview of the LabVIEW modelling platform. Section 4 discusses the research questions that motivate our study. Section 5 gives a detailed description of the seven bad smells that have been selected for analysis in this paper, while Section 6 introduces the Hawk framework for mining repositories of graphical models. We also discuss how Hawk queries were generated for the selected bad smells. Section 7 presents the evaluation of the queries on 81 models in 10 LabVIEW repositories and the analysis of the results returned by the queries across the version history of the models. The section also presents answers to the research questions that motivate this research. Section 8 mentions the threats that may challenge the results analysed in Section 7, while Section 9 discusses related work and highlights how our work extends existing literature. Finally, Section 10 offers our concluding comments and also outlines our future plans to enhance the evolution analysis of graphical models.

2. 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 et al. 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

[1] http://bit.ly/LMeta

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 language (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 & Gray 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 & Scaffidi 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.

3. Overview of LabVIEW

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). 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 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 1 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 NXG version stores models in XML, making it more amenable to read and analyse by external tools. This study focuses on LabVIEW programs developed in the new NXG version. Listing 1 shows how the VI in Figure 1 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). Similarly, Lines 8 to 13 represent 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.

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

Listing 1 Simplified Block Diagram Part of a VI file

4. Research Questions

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 et al. (Chambers & Scaffidi 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,

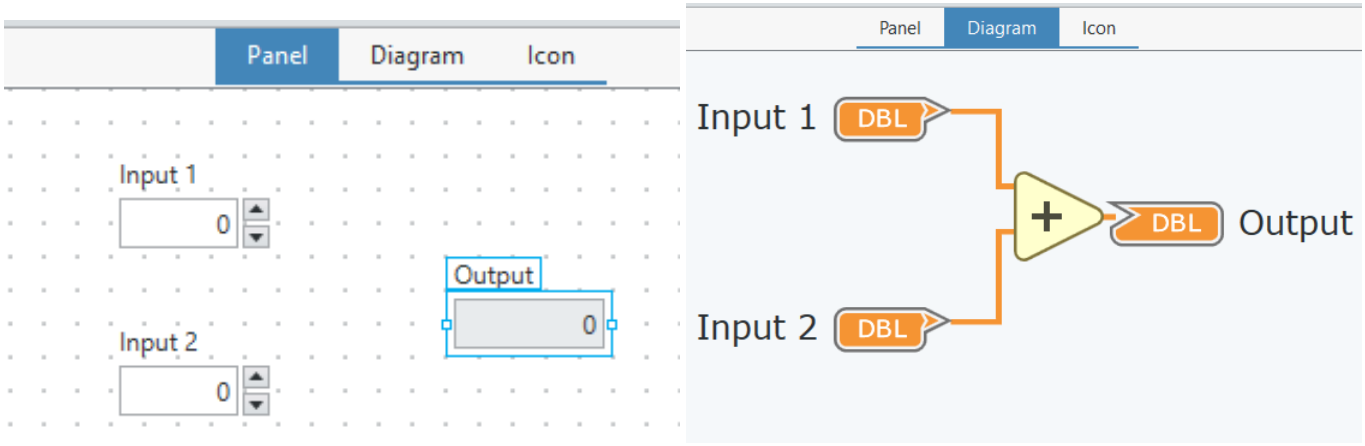

Figure 1 A Front Panel and Block Diagram to Add Two Numbers

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 (Roberts et al. 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 et al. 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 (Roberts et al. 1999) and Tufano et al. (Tufano et al. 2015) suggest the 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 extended the Hawk tool (see Section 6) to support querying the evolution history of LabVIEW models. This makes it easy to extract the history of LabVIEW models in selected GitHub repositories.
3. We constructed seven Hawk queries to detect instances of seven smells in LabVIEW. The execution of the queries in Hawk will produce the evolution history of smells in selected models.
4. We manually analysed the results of the queries to answer the research questions discussed above.

5. Smell Selection

Many smells in LabVIEW have been identified in previous works. Chambers and Scaffidi (Chambers & Scaffidi 2013) identified smells based on an interview conducted with experienced LabVIEW developers while Carcao et al. (Carçao 2014)(Carçao et al. 2014) 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 & Gray 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 & Gray 2019) identified 15 bad smells, while Chambers and Scaffidi (Chambers & Scaffidi 2013) identified 13 smells. Six of these smells were reported in both papers; thereby, resulting in a net total of 22 smells. We selected seven of the smells for this study due to the limited cost and time required to effectively analyse all of the 22 smells. The seven smells studied in this paper 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 program consumes while executing. *No Wait in a Loop* is likely to cause synchronization issues (Chambers & Scaffidi 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 & Scaffidi 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 also 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.

A Large Variables

Large Variables 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.

B 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 modeling, 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.

C 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.

D 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.

E 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 & Scaffidi 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.

F **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 (Quilleré et al. 2000) (Karunaratne et al. 2018). Similar to other programming languages, the use of nested loops in LabVIEW models reduces readability (Chambers & Scaffidi 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[2], design issue[3]).

G **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 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 et al. 2015).

6. Queries to Detect Smells in Model Repositories

To support querying of LabVIEW models in repositories, we extended the Hawk platform to support LabVIEW models, and developed a metamodel that captures the properties and relationships across elements in LabVIEW models. The following subsections discuss how the Hawk framework and LabVIEW metamodel help to simplify the writing of queries to detect smells in LabVIEW models.

6.1. Hawk

Hawk is a platform for querying version histories of models stored in file-based repositories such as the models that are available in Version Control Systems (VCS). A VCS (e.g., Git) provides capabilities for tracking changes made to files and handling conflicts due to file edits by multiple users. However, a traditional VCS does not provide support for managing the complex relationships among elements in models that are stored as files; thereby, leaving such complexities to be handled by the user (Bartelt 2008).

Hawk provides 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. The queries supported by Hawk include: detecting changes across files in a repository, adding new model files, and extracting models that conform to some metamodel. Figure 2 provides a graphical overview of Hawk.

The Hawk tool adopts a component-based architecture where each component can be extended towards a more specific requirement. The VCS manager component detects and retrieves changes in files stored in a VCS, such as Git, while the Model Resource Factory (or model interpreter) translates the files into a compatible EMF representation. The model updater compares the models with the latest version that has been stored in a backend database. The model updater then performs an incremental update on the backend database to reflect the newly added models. The query engine provides capabilities to retrieve information about the models that have been stored in the backend database. The time-aware dialect of Hawk is able to support time-aware indexing where changes in the models are associated with a timestamp that corresponds to when the changes were made. Hawk also provides a time-aware query engine that can extract historic information about the models.

In this paper, we extend the time-aware dialect of Hawk to support LabVIEW models via a model interpreter and a LabVIEW metamodel that captures the properties and relationships across elements in existing LabVIEW models. The model interpreter 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 3 is a graphical overview of our extended Hawk tool. The next subsection discusses the LabVIEW metamodel in detail.

6.2. LabVIEW Metamodel

To the best of our knowledge, there is no open-source metamodel for LabVIEW nor is there any available specification document that captures the concepts in LabVIEW and the relationship among these concepts. Hence, in our previous work we developed a LabVIEW metamodel (Popoola & Gray 2019) via example-driven techniques (such as those used in (López-Fernández et al. 2015)) by curating over 100 LabVIEW models from the LabVIEW examples repository that is present in the LabVIEW IDE. The metamodel captures several types of LabVIEW dependencies and the relationships that exist across these

[2] https://forums.ni.com/t5/LabVIEW/Nested-while-loops/td-p/2167378?profile.language=en

[3] https://forums.ni.com/t5/LabVIEW/Nested-FOR-loops-and-flat-sequence-A-good-design/td-p/2454042?profile.language=en

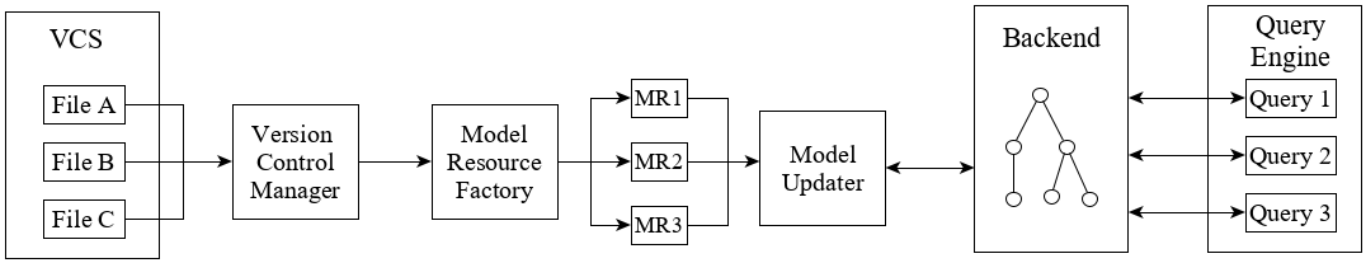

Figure 2 The Hawk Tool (Barmpis et al. 2020)

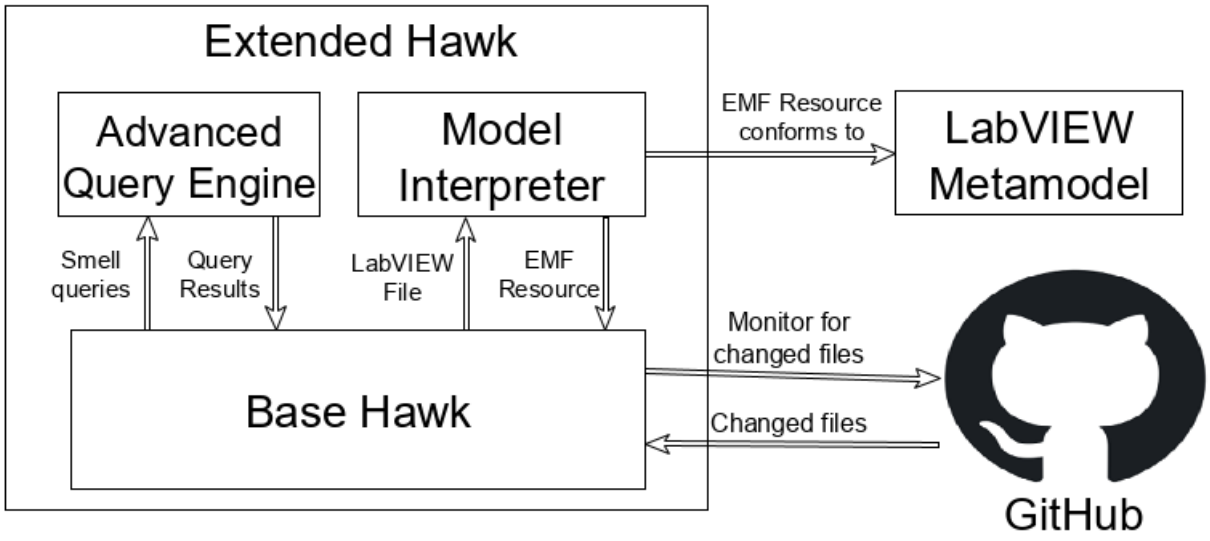

Figure 3 The Extended Hawk Tool

dependencies. Figure 4 shows a simplified view of the metamodel.

We simplify the amount of effort required to translate a bad smell into an efficient query method via an extensible and carefully designed metamodel that captures the relationships among the elements in a LabVIEW model. 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. 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 (from Section 5) to detect the presence of the bad smells in the repositories. 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. Furthermore, because the LabVIEW metamodel was developed by curating sample models, it is possible that the metamodel may not cover all possible features in a LabVIEW model. However, the metamodel is open-source and extensible; therefore, it can be easily extended by anyone to accommodate more features that are not currently covered.

For *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 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.

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 the 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. The complete open-source LabVIEW metamodel, model interpreter, and all of the seven queries developed to extract the bad smells are publicly available online (please see http://bit.ly/LMeta).

Repo \ Properties	URL for Repository	# of Models	# of Elements
Repo 1	https://github.com/ni/webvi-examples	1	680
Repo 2	https://github.com/JKISoftware/JKI-State-Machine-NXG	10	5537
Repo 3	https://github.com/ni/labview-nxg-jenkins-build	14	270
Repo 4	https://github.com/rajsite/ webvi-hack	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 1 Overview of 10 Repositories and Identified Smells.

7. Research Questions 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 4, 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. An initial set of selected projects were identified and the models embedded in these projects were validated against our LabVIEW metamodel. The metamodel was also updated to accommodate new relationships that were observed across the 10 public projects containing 81 LabVIEW models. Table 1 gives an overview of the repositories. The following subsections address each of the research questions

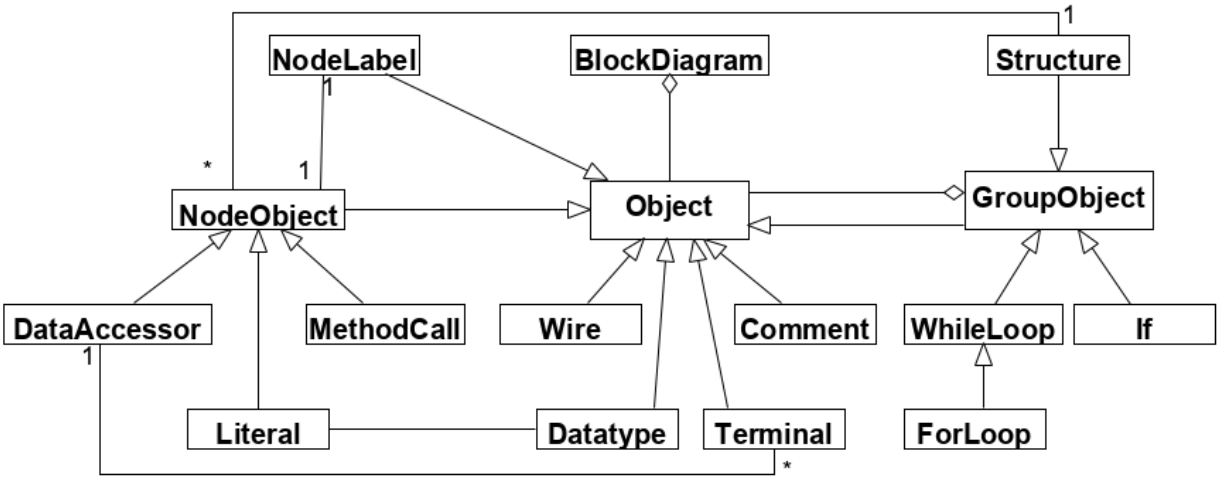

Figure 4 A simplified LabVIEW metamodel (Popoola & Gray 2019)

and provides an in-depth study of the evolution of bad smells in LabVIEW models.

7.1. 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 2 shows that most of the repositories contain instances of 2 or 3 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 developers are not very familiar with software engineering design principles because bad smells are often associated with bad software engineering practices (Van Emden & Moonen 2002). Table 2 gives a breakdown of the presence of each of the bad smells in the life cycle of the models stored in the repositories.

7.2. 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 was 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 summarizes the results corresponding to when a bad smell is introduced.

The number of bad smell instances also tends to increase steadily over time, and then decline after a peak period. Figures 6, 7, 8, 9 shows the evolution of four types of smells across the version history of the repositories. It should be noted that the smells *String Concatenation in 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 7 versions; hence, there may be a constant value from the last version number to the "current version" specified in the figures. 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 10 gives an overview of the evolution of smells in repository 2, while Figure 11 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.

7.3. 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.

We used EMFCompare (Toulmé & Inc 2006) to extract the structural differences between different versions of LabVIEW Models. EMFCompare is a model differencing framework for extracting differences between two or three sets of models. Two-way differencing involves the direct comparison of two models,

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 2 Detection of Each Smell across 10 Repositories.

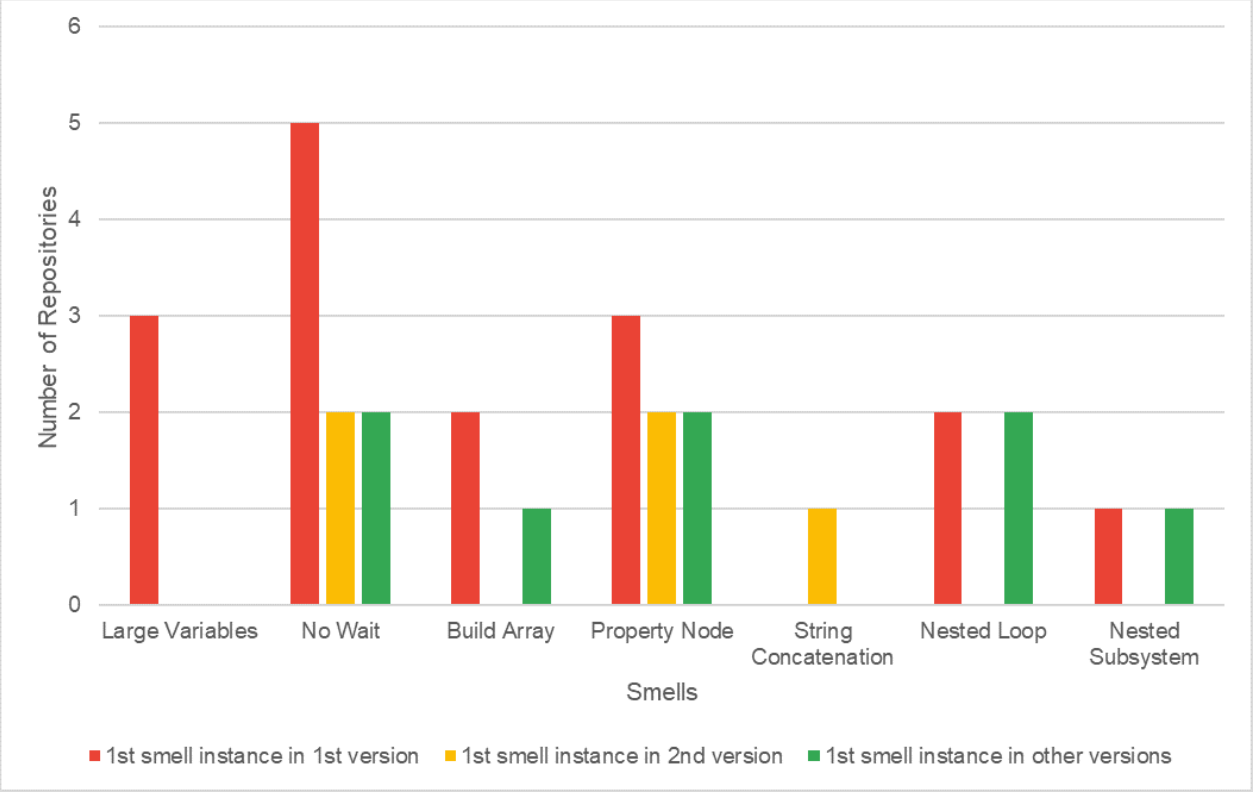

Figure 5 Smell Introduction across Repositories

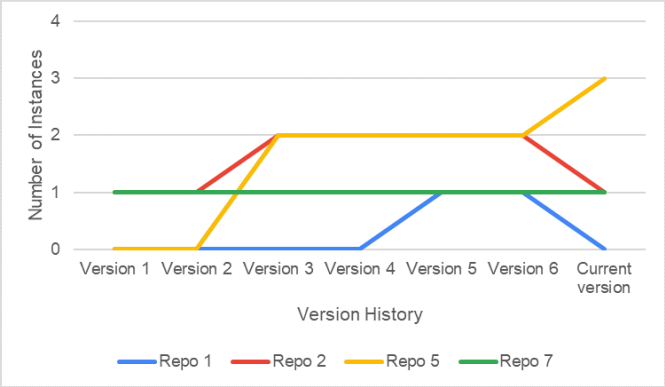

Figure 6 Evolution of *Multiple Nested Loops* Across Version History of LabVIEW Models

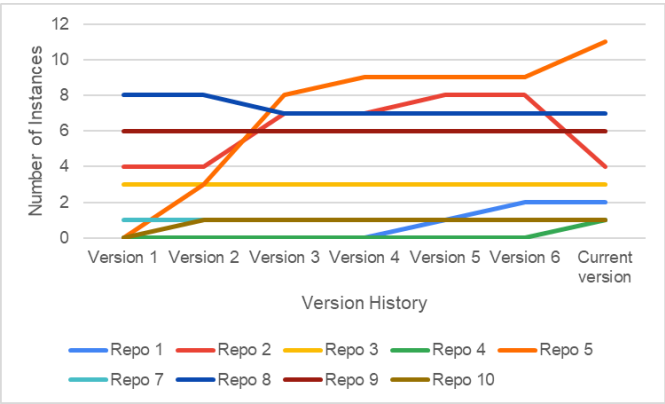

Figure 7 Evolution of *No Wait in a Loop* Across Version History of LabVIEW Models

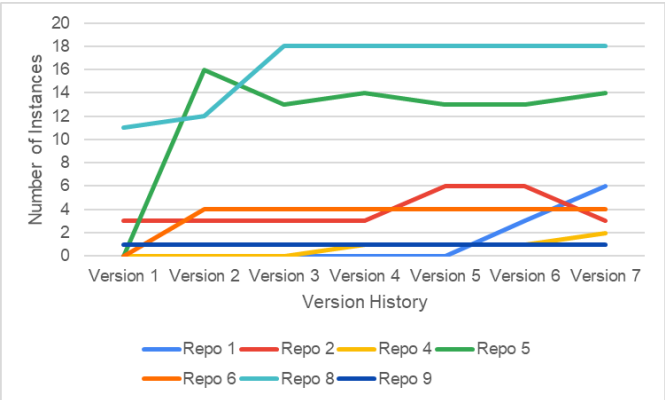

Figure 8 Evolution of *Excessive Property Nodes* Across Version History of LabVIEW Models

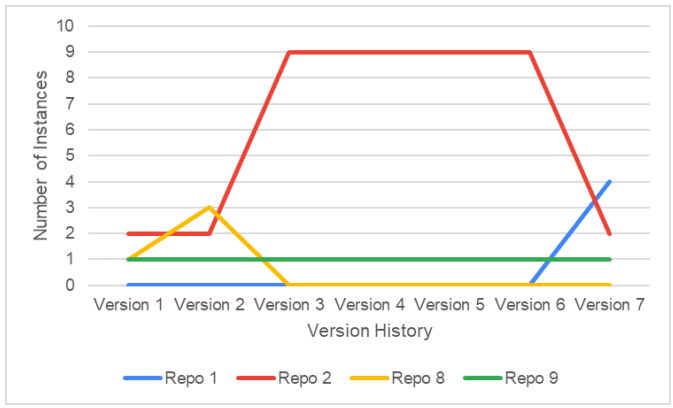

Figure 9 Evolution of *Large Variables* Across Version History of LabVIEW Models

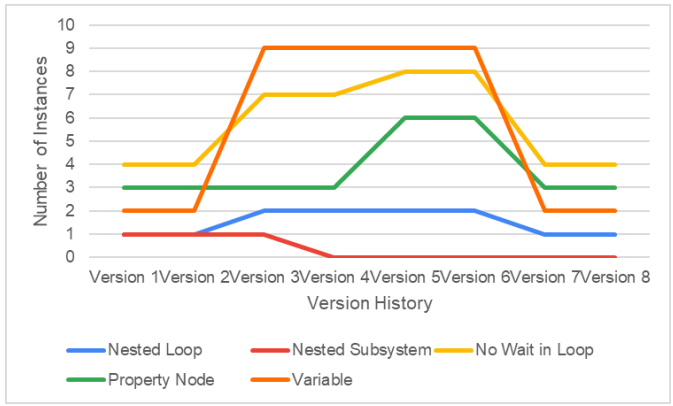

Figure 10 Evolution of Bad Smells in Repository 2

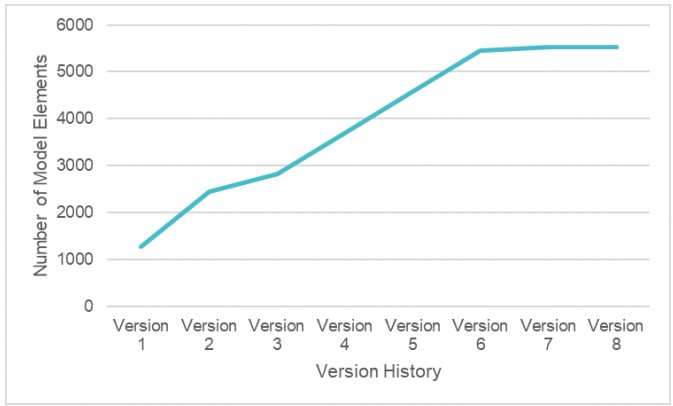

Figure 11 Evolution of the Model Size in Repository 2

Repo \Diff Kind	Avg ADD	Avg DELETE	Avg CHANGE	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 3 Average Number of Difference Types Across Version History.

while three-way differencing involves the comparison of two models based on their evolution via a third common model. For this study, we used the two-way differencing for direct comparison between models in a specific version and models in the succeeding version. 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 elements.

To analyse the differences in models across various versions, each version of the models in the identified repository 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 were then aggregated and analysed for all versions in a repository. This analysis makes it easy to understand the changes that necessitate the evolution of LabVIEW programs. 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 3 shows the average number of modifications that are performed across each version in the repositories.

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 12 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

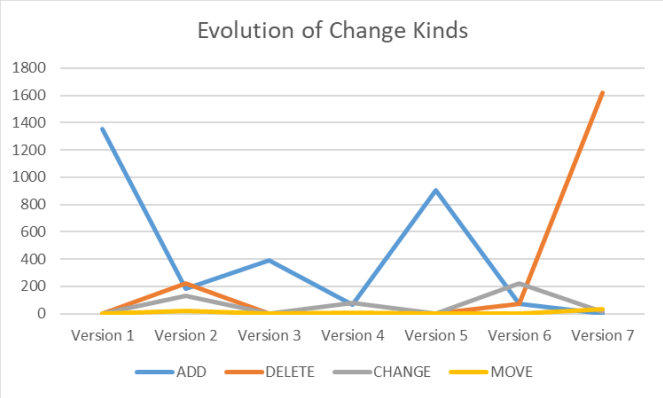

Figure 12 Evolution of Change Kinds in Repository 2

in the number of additions. This corresponds to the decrease in bad smells shown in Figures 6 to 9.

8. Threats to Validity

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

A 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.

B 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 developers 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.

C 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.

D 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, 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 paper provide significant insight into the general evolution of bad smells in systems models.

9. Related Work

Much work exists on bad smell identification in text-based languages. Chatzigeorgiou et al. (Chatzigeorgiou & Manakos 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 et al. 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 et al. 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 languages. Simulink provides an interactive, graphical environment for modeling, simulating, and analysing of 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 et al. 2015). In this work, the authors summarized 21 bad smells into 5 categories: name, partition, interface, signal flow and signal structure. Two tools, Artshopr (Thomas et al. 2016) and SLRefactor (Tran & Dziobek 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 & Cordy 2015).

Chambers and Scaffidi (Chambers & Scaffidi 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 & Gray 2019). However, both (Chambers & Scaffidi 2013) and (Zhao & Gray 2019) do not investigate the presence of the identified smells in code repositories nor do they study the evolution of such smells. This paper 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.

10. Conclusion and Future Work

We 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 69 models in 10 GitHub repositories. The evaluation process includes the analysis of four 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.

In the future, we plan to carry out an extensive analysis on more GitHub projects and other graphical modeling tools such as Simulink. The majority of the smells discussed in this paper are specific to LabVIEW; hence, we plan to cover more smells such as "Duplicate Model" that can be generalizable to other graphical modelling environments. We also plan to conduct qualitative studies that will complement the results presented in this paper. We will conduct studies to understand the context in which bad smells are introduced, the relationship between author demographics and bad smells, and the level of consideration given to bad smells by LabVIEW developers. Furthermore, we will implement and study the benefits of a refactoring mechanism to address the challenges of bad smells from the perspective of a systems engineer. Finally, we plan to improve our study to identify refactoring activities that specifically target the reduction of bad smells, discover the impacts of bad smells on maintenance activities, and highlight the differences in the analysis of bad smells in graphical and textual languages.

Acknowledgements

The authors would like to appreciate the contribution of Dr. Taylor Riché (National Instruments) and Dr. Antonio Garcia-Dominguez (Aston University, UK) for their help with general questions for LabVIEW and Hawk respectively. We would also like to thank the editors and reviewers for their time and insightful feedback.

References

Barmpis, K., García-Domínguez, A., Bagnato, A., & Abherve, A. (2020). Monitoring model analytics over large repositories with hawk and measure. In *Model Management and Analytics for Large Scale Systems* (pp. 87–123). Elsevier.

Barmpis, K., & Kolovos, D. S. (2014). Towards scalable querying of large-scale models. In *10th European Conference on Modelling Foundations and Applications* (pp. 35–50).

Bartelt, C. (2008). Consistence preserving model merge in collaborative development processes. In *Proceedings of the International Workshop on Comparison and Versioning of Software Models* (pp. 13–18).

Carçao, T. (2014). Measuring and visualizing energy consumption within software code. In *IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 181–182).

Carçao, T., Cunha, J., Fernandes, J. P., Pereira, R., & Saraiva, J. (2014). 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, C., & Scaffidi, C. (2013). Smell-driven performance analysis for end-user programmers. In *IEEE Symposium on Visual Languages and Human Centric Computing* (pp. 159–166).

Chatzigeorgiou, A., & Manakos, A. (2014). Investigating the evolution of code smells in object-oriented systems. *Innovations in Systems and Software Engineering*, *10*(1), 3–18.

Exman, I. (2014). Linear software models: standard modularity highlights residual coupling. *International Journal of Software Engineering and Knowledge Engineering*, *24*(02), 183–210.

Falcon, J. (2017). Facilitating modeling and simulation of complex systems through interoperable software. In *Keynote address at 20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems.*

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. USA: Addison-Wesley Longman Publishing Co., Inc.

García-Domínguez, A., Bencomo, N., Parra-Ullauri, J. M., & García-Paucar, L. H. (2019). Querying and annotating model histories with time-aware patterns. In *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (pp. 194–204).

Gerlitz, T., Tran, Q. M., & Dziobek, C. (2015). Detection and handling of model smells for MATLAB/Simulink models. In *Modeling in Automotive Software Engineering Workshop at the 18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (pp. 13–22).

Johnson, G. W. (1997). *LabVIEW Graphical Programming*. Tata McGraw-Hill Education.

Karunaratne, M., Tan, C., Kulkarni, A., Mitra, T., & Peh, L.-S. (2018). Dnestmap: mapping deeply-nested loops on ultra-low power cgras. In *Proceedings of the 55th Annual Design Automation Conference* (pp. 1–6).

López-Fernández, J. J., Cuadrado, J. S., Guerra, E., & De Lara, J. (2015). Example-driven meta-model development. *Software and Systems Modeling*, *14*(4), 1323–1347.

Ludewig, J. (2003). Models in software engineering–An introduction. *Software and Systems Modeling*, *2*(1), 5–14.

Olbrich, S., Cruzes, D. S., Basili, V., & Zazworka, N. (2009). The evolution and impact of code smells: A case study of two open source systems. In *3rd International Symposium on Empirical Software Engineering and Measurement* (pp. 390–400).

Opdyke, W. F. (1992). *Refactoring object-oriented frameworks* (Unpublished doctoral dissertation). University of Illinois at Urbana-Champaign Champaign, IL, USA.

Popoola, S., & Gray, J. (2019). A LabVIEW metamodel for automated analysis. In *International Conference on Computational Science and Computational Intelligence* (p. 1127-1132).

Quilleré, F., Rajopadhye, S., & Wilde, D. (2000). Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, *28*(5), 469–498.

Robbes, R., Kamei, Y., & Pinzger, M. (2017). Guest Editorial: Mining software repositories. *Empirical Software Engineering*, *22*(3), 1143–1145.

Roberts, D., Opdyke, W., Beck, K., Fowler, M., & Brant, J. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Boston.

Services, E. (n.d.). *Companies using labview.* https://enlyft.com/tech/products/labview. (Accessed: January 2020)

Stephan, M., & Cordy, J. R. (2015). Identification of Simulink model antipattern instances using model clone detection. In *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (pp. 276–285).

Tahmid, A., Nahar, N., & Sakib, K. (2016). Understanding the evolution of code smells by observing code smell clusters. In *23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering* (Vol. 4, pp. 8–11).

Thomas, G., Norman, H., Christian, D., & Stefan, K. (2016). Artshop: A continuous integration and quality assessment framework for model-based software artifacts. In *Dagstuhl Workshops – Model Based Engineering of Embedded Systems* (pp. 13–22).

Toulmé, A., & Inc, I. (2006). Presentation of EMF Compare utility. In *Eclipse Modeling Symposium* (pp. 1–8).

Tran, Q. M., & Dziobek, C. (2013). 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).

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., & Poshyvanyk, D. (2015). When and why your code starts to smell bad. In *37th ACM/IEEE International Conference on Software Engineering* (Vol. 1, pp. 403–414).

Van Emden, E., & Moonen, L. (2002). Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering* (pp. 97–106).

Zaman, S., Adams, B., & Hassan, A. E. (2012). A qualitative study on performance bugs. In *9th IEEE Working Conference on Mining Software Repositories* (pp. 199–208).

Zhao, X., & Gray, J. (2019). BESMER: An approach for bad smells summarization in systems models. In *Models and Evolution Workshop at the 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (pp. 304–313).

About the authors

Saheed Popoola is a PhD candidate in the Department of Computer Science at the University of Alabama. His doctoral research deals with the evolution and design analysis of systems models. His broad research interests deals with the application of software engineering techniques to address software development challenges in systems engineering. Prior to joining the University of Alabama, he was a member of the Enterprise Systems Group, University of York where he obtained an MSc (By Research) in Computer Science, under the su-

pervision of Prof. Dimitris Kolovos. You can contact him at sopopoola@crimson.ua.edu or visit http://popoola.cs.ua.edu.

Xin Zhao is a PhD candidate in the Department of Computer Science at the University of Alabama. His doctoral research investigates the evaluation of systems models, including bad smells categorization, model refactoring and metrics analyses of systems models. He is also interested in software product lines and CS education. He obtained his B.E. in the Hebei Normal University, China. Before joining the University of Alabama, he was a member of the National Engineering Research Center for Multimedia Software, Wuhan University, China. You can contact him at xzhao24@crimson.ua.edu or visit http://xzhao24.students.cs.ua.edu.

Jeff Gray is a Professor in the Department of Computer Science at the University of Alabama. His research interests are in the areas of software engineering, model-driven engineering, and computer science education. Jeff is a Distinguished Member of the ACM and a Senior Member of the IEEE. He is the co-Editor in Chief of the Journal of Software and Systems Modeling (SoSyM). You can contact him at gray@cs.ua.edu or visit http://gray.cs.ua.edu.