Automatic Repair of Framework Applications

Zack Coker

CMU-CS-20-106
May 2020

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

This research was sponsored by the National Science Foundation under the Graduate Research Fellowship Program (grant numbers 0946825 and DGE1252522) and grant numbers CCF-1446966, and CCF-1618220, the Department of Defense under grant number FA872105C003, and the Air Force Research Laboratory under grant numbers FA87501910501 and FA87501620042.

Keywords: Frameworks, Automated repair, Debugging, APIs, Directives

To the people who gave to me without asking for anything in return.

Abstract

Developers commonly build framework applications. Developers who use frameworks get a productivity boost from being able to reuse parts of previous applications that have been shown to work in a given domain. However, the reuse provided by frameworks also comes with a variety of challenges. Frameworks impose many assumptions about a framework application on developers and require developers to write applications in a way that conforms to the framework's expectations. To provide the benefits of framework reuse to developers of new applications, frameworks require developers to build applications using programming concepts (such as inversion of control, where the framework calls the unique parts of the framework application only when required, instead of the unique framework application code controlling the execution flow of the application) that either do not occur in other programming situations or occur in a significantly higher frequency in framework development. The different programming concepts that developers encounter when creating framework applications present problems to developers. The goal of this dissertation is to address the problems that framework application developers encounter with tool support.

This dissertation presents a structured process for assisting framework application developers with tool support. First, I identified the challenges that developers faced in the framework application development process. This study demonstrated that an automated repair technique for framework applications would benefit developers. An automated repair technique needs to be able to both identify the problem and then create a possible repair for the problem. The second part of this dissertation discusses a strategy for automatically identifying problems in framework applications while the third part of the dissertation discusses an approach to generate repairs for framework application problems.

As mentioned earlier, the first step in assisting framework application developers was to gain an understanding of the problems that framework application developers encounter in the framework application development process. Online developer sites, like StackOverflow, demonstrated the difficulty that developers have fixing problems in framework applications and leads to the insight that improving the framework application debugging process would benefit developers. Based on these insights, I performed an exploratory study to investigate the framework application debugging process. The exploratory study followed a mixed-methods approach and adapted well-known qualitative research methodologies, such as grounded theory, to the debugging cases of interest. To approach framework application problems in a rigorous manner, I focused on directives, testable and possibly surprising requirements for how a framework application should interact with a framework. Through a preliminary study on framework directives, I found that violating directives can lead to seven different consequences — ways that errors are presented to developers. I then used the set of consequences to guide a debugging study on framework application debugging scenarios. I used directives to guide the problems in the debugging study. From the debugging study, I found that developers encountered challenges, such as understanding how inversion of control would affect objects passed to the framework application. The study also found that developers encountered particular difficulty debugging state-based issues, those where object instance variables contained values that conflicted with method call assumptions.

From this study, I theorized that framework application developers would benefit from an automated repair tool, tailored to fix state-based issues in framework applications. The automated repair tool would need both 1) a way to detect issues in framework applications and 2) to propose possible repairs. I created a static analysis tool to detect issues in framework applications. In this static analysis tool, a person knowledgeable with the framework converts the property to check into an API (Application Programming Interface) call, defined by a specification language I created. These API calls are then converted into composable checks to determine if the application violates the specification. I evaluated the language on a taxonomy of object protocols. I found that the language was able to easily support five out of seven categories. I also ran the checkers on real applications in the F-Droid dataset, which detected bugs in real applications.

The other part of the automated repair tool was the repair generation process. I created a tool, FrameFix, that uses common aspects of framework applications to propose repairs. The repair process includes moving methods to locations where objects may be in the right state and using similarly named methods from applications on GitHub to guide repairs. I found that the tool was able to repair a set of personally-created applications, problems injected into real applications, and problems found in real applications.

These projects were done with the goal that the techniques would apply to a wide variety of frameworks. The debugging study was conducted on two frameworks to ensure that conclusions were not framework-specific. While the static analysis check and the repair tool were built for only one framework (due to implementation limitations), the techniques the tool uses were designed in a way that were not framework-specific. The tools were evaluated over a large number (close to 2,000) real applications to demonstrate that they apply to real framework applications.

Acknowledgments

Thank you to everyone that supported me throughout my Ph.D. Claire Le Goues, my adviser, and Joshua Sunshine were instrumental in helping me become the scientist I am today, and I am grateful for their investment in my learning process. I would also like to thank my undergraduate adviser Munawar Hafiz, for helping me start the journey into research. My other committee members, David Garlan and Ciera Jaspan, and other professors who I interacted with during my PhD provided insights that made my experience more rewarding. My parents and the rest of my family provided a lot of support. The people in my research group who worked with me throughout the years (Cody, Chris, Afsoon, Deby, Jeremy, Rijnard, Leo, Mau, and the many undergrads and other temporary research group members) were important to my development and I enjoyed working with them. I also appreciate the people who helped me have a social life during the process, including Kelvin, Sanjana, Aram, Dougal, David, Sid, Goran, and Sol. Finally, thank you to the professors, bosses, and other people who took an interest in me and helped me develop into the person I am today. Thank you Katie, for proof-reading my dissertation. I am sure there are many people who have helped me along the way, and I am sorry I forgot to mention you. While I may have not mentioned them here, I am very grateful for what they did for me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Due to both the high demand for software products and the complexity of building these software products, our society repeatedly invests in improving the software development process. One of these innovations that has been widely adopted by the software industry is frameworks. A framework provides a partially complete application that can be extended to produce custom applications [34]. Frameworks improve the software development process through *architectural reuse* [8]. The *architecture* of the framework refers to the static class structure of internal framework classes, as well as the dynamic interactions between the framework and additional framework application code. The architectural reuse allows developers to re-purpose a software design that has been proven to work in a given domain to their unique application [34], which increases developer productivity [50].

1.1 Frameworks

1.1.1 Basic Terms

For the purposes of this dissertation, a *framework application* (often shortened to *application*) refers to a complete software program, built with one or more frameworks, that completes a task. When a developer builds a framework application with a software framework, the application consists of two parts. The first part is the *software framework*, often shortened to *framework*. The *framework* is a reusable piece of software that controls the execution flow and data flow of the rest of the application [21]. The second part of the application is the *plugin*, the application-specific part that extends the basic framework to perform a task. The framework interacts with the plugin through defined interaction points. In these interaction points, the framework calls the plugin and passes data about the state of the application to the plugin. Plugins then perform their application-specific task and relinquish control (and optionally pass data) back to the framework code.

1.1.2 Examples

To illustrate the relationship between a framework and a developer-built plugin, consider CoBot, a robot controlled by a framework application. CoBot is an autonomous navigation robot that contains sensors, wheels, and a screen for interacting with a human [89]. Researchers at Carnegie Mellon University use CoBot to perform many different navigational studies and to autonomously deliver candy to office workers on Halloween. CoBot was built using the Robotic Operating System (ROS [79]) framework, which is a framework for building robotic applications. ROS structures and controls the way messages are passed between different parts of the robot. To create the ROS application that controls CoBot, a developer would write the code for the unique parts of the robot (the plugin), such as how to sense and make decisions based on the current situation. ROS would provide the structure for communicating between the different parts of the robot, reducing the difficulty of coordinating a multi-part system. In this case, the plugin developers are able to use the event-based architecture provided by ROS to save development effort, allowing them to reuse the architecture that has been tested in previous robotic applications.

Another example is an Android application. Android is a framework for creating mobile applications [38]. Consider a developer who wants to build an application to receive medical alerts and updates for the user's current location. To build this application, the developer would build a plugin that shows medical alerts to users on top of the Android framework. The Android framework simplifies the process for interacting with user input, such as the process of starting the application when it is selected by the user. Android also provides a standard way to process and update an application with location information. Instead of requiring the developer to process GPS information from a satellite and monitor this information to keep the location information current, the framework provides location updates to the plugin. The plugin developer would then create the logic to update the application based on the location information, which is a large reduction in complexity compared to creating a complete location system.

1.1.3 What Makes Framework Development Unique

The process of developing framework plugins has major differences from the process of developing other programs, such as a program that calls out to libraries. The major differences are as follows:

- *Inversion of control* - In many programs, the code that developers write controls the execution flow and data flow of the application. However, this is not the case for many framework applications. Since the same execution flow and data flow can be reused across framework applications, the execution flow and data flow are managed by the framework. Due to the fact that the developer-created plugin is not managing these flows, the developer must instead interact with the framework through interaction points that are defined by the framework.

  Example: Android requires plugin developers to create a class that extends the Android `Activity` class. The framework will then call the methods in this new class that override the base methods in the original `Activity` class. Android documentation recommends that plugin developers override at least two methods: `onCreate(Bundle)` and `onPause()`. Figure 1.1 illustrates how the framework calls methods in the plugin, which

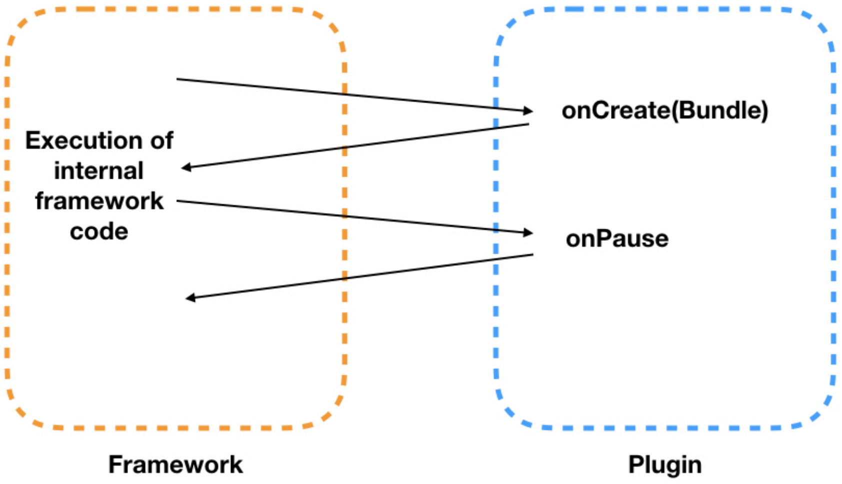


Figure 1.1: An example of how inversion of control is implemented in an Android application. The Android framework controls the execution of the application and then calls plugin methods, such as `onCreate(Bundle)` and `onPause` when the plugin needs to handle associated events. Data is passed from the framework to the plugin through parameters, such as `Bundle` and returned at the end of the method call. After executing each method, execution is returned to the internal framework code.

relinquish control when they are finished.

- *Object protocols* - Object protocols are partial ordering restrictions on method calls on a software object [85]. This means that a given object goes through state transitions where only certain method calls are allowed in each state. While object protocols occur in many programming contexts, they are heavily prevalent in framework development [8]. Framework application programming is further complicated by the way object protocols are implemented in frameworks. Frameworks will often change the state of objects in internal framework code, in a way that is not immediately obvious to developers. And, due to the way that data is passed through *inversion of control*, framework application developers will often need to keep track of the possible states of objects the framework passes to the plugin.

  Example: One of the most common example of an object protocol is a File object, as shown in Figure 1.2. In the File-object protocol, the file object must be opened before the program can read or write to the file. Once the file object has been closed, the read and write methods cannot successfully alter the file object.

- *Declarative artifacts* - Framework applications often need statically-defined files, often configuration files, e.g., static layout or string definition files [21]. While declarative artifacts create a standard place to define important application information, they also distribute important information into multiple places in the application. Plugin developers must understand how the static configuration interacts with dynamically executed application code.

  Example: Figure 1.3 shows an example declarative artifact from Android. This is a sample

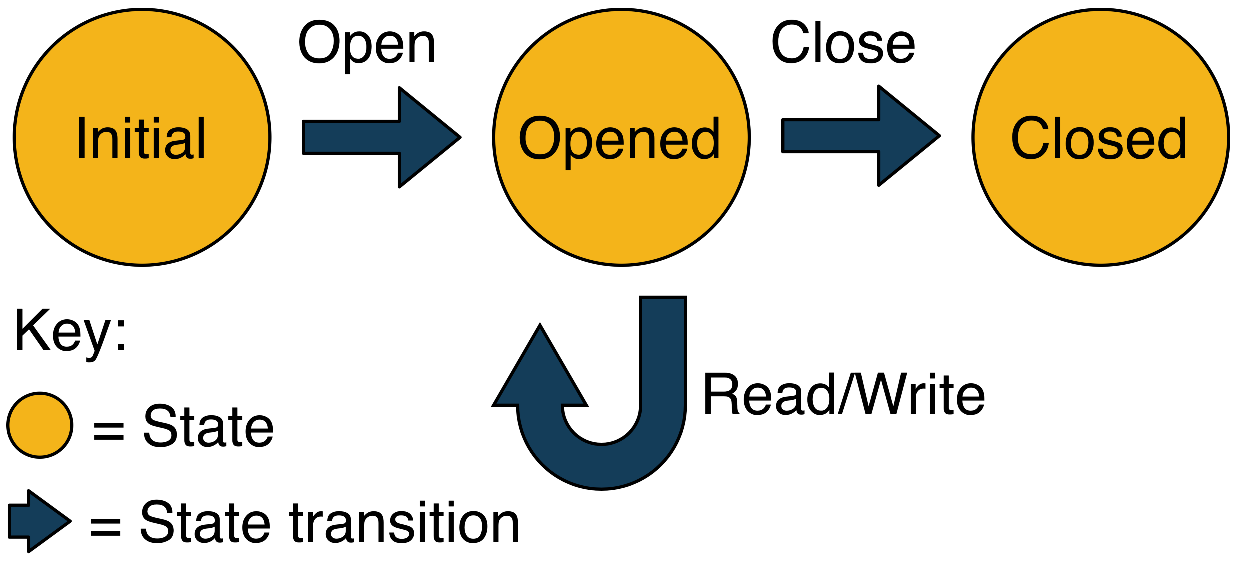


Figure 1.2: The File object protocol. Notice how certain method calls, such as `read` and `write`, are only available after opening the file and before closing the file.

layout file that statically defines how to show a sample string to the viewer. Android will then create a **`ViewGroup`** object from this layout definition when it should be displayed to the user.

- *Standard organization* - To provide architectural reuse, frameworks often impose a general architectural structure on an application. This structure requires plugin developers to organize the parts of the plugin that interact with the internal framework code in a similar manner across multiple applications. The similar structure across multiple plugins likely makes it easier to understand different applications for the same framework.

  Example: When creating a basic **`Activity`** in Android Studio, the official Integrated Development Environment of Android, the application is generated with a basic file structure. This file structure, shown in Figure 1.4, separates the source code files into the `java` folder, and the resources (such as the static layout files) into the `res` folder. Android imposes this file structure on plugins, so that the framework can access needed resources on request.

1.1.4 Current State of Framework Development

Due to the known and substantial productivity gains, frameworks are used by the software development industry to create a large number of diverse applications. For example, in 2019, the Google Play store generated 29.3 million dollars in yearly revenue [1] from about 2.9 million applications [2], all written using the Android framework. While frameworks are widely used, developers still encounter significant challenges when developing framework applications. Of the top 20 tagged question categories on StackOverflow (a popular developer question-and-answer site), shown in Figure 1.5, 6 correspond to framework categories, and these categories contain over 3 million questions [3]. Ko et al. investigated the challenges faced by plugin devel-

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="
    http://schemas.android.com/apk/res/android"
     xmlns:app="http://schemas.android.com/apk/res-auto"
     xmlns:tools="http://schemas.android.com/tools"
     android:layout_width="match_parent"
     android:layout_height="match_parent"
     app:layout_behavior="@string/appbar_scrolling_view_behavior"
     tools:context=".MainActivity"
     tools:showIn="@layout/activity_main">

    <TextView
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:text="Hello World!"
         app:layout_constraintBottom_toBottomOf="parent"
         app:layout_constraintLeft_toLeftOf="parent"
         app:layout_constraintRight_toRightOf="parent"
         app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Figure 1.3: An example declarative artifact. This is an example layout file written in XML. The file specifies to create a **`TextView`** with the contents *Hello World!* inside of a **`ConstraintLayout`**.

opers new to a framework (VisualBasic.NET), and found six different categories of challenges, demonstrating that new framework application developers encounter a diverse set of programming challenges (such as not knowing how to apply compiler feedback to get the program to compile) [55]. So while frameworks improve developer productivity overall, framework application developers still encounter many challenges.

1.2 Scope

While researchers have investigated problems in the framework application development process, none of them have investigated the challenges of debugging framework applications in-depth (Ko et al. included one debugging task in their study [55]). I hypothesized that the unique aspects of framework debugging would lead to unique debugging challenges. To investigate the unique challenges of framework application development, as the first contribution of this dissertation, I studied the challenges of debugging framework applications with a mixed methodology approach that used techniques from grounded theory and qualitative content analysis. I found that repairs concerning framework-imposed state constraints, that is, repairs that involved understand-

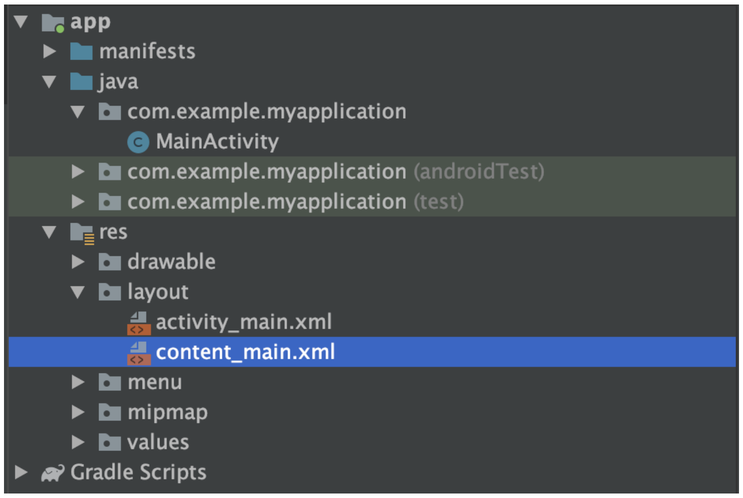


Figure 1.4: The project structure when a developer creates a basic `Activity`. Notice how the source code files, layout files, and build files are organized into different folders by default.

ing when values and method calls were valid in the framework application — were particularly challenging for developers. Notably, developers struggled with these repairs even when the failure location was provided [23].

These results motivate an automated technique for repairing state-based constraints. To address this need, I created, FrameFix, an automated repair technique for framework state constraint violations that exploits how framework state constraints limit the number of repair options. This repair technique consisted of two main parts: 1) a static analysis to identify the state constraint violations, with a new specification language consisting of a combination of known approaches, and 2) the repair generation process. The static analysis technique and the specification language were evaluated on both its generality to cover specifications in a specification taxonomy and its ability to scale to real applications. I evaluated the repair technique with respect to its accuracy and generality. *Accurate*, in the context of the automated repair technique, means that the repair technique removes the specified fault while retaining specified functionality. A repair technique can only be practically useful if it is accurate. *Generality*, in this case, means that the technique can apply to a large number framework applications, and that the technique could apply to more framework constraints than those addressed in the initial implementation, if the necessary software infrastructure were created. A state constraint repair technique should be applicable to more state constraints than those used to build it.

The previous paragraph is summarized into the following thesis statement:

Thesis statement: *Framework application developers encounter distinct challenges when reusing the architecture provided by frameworks, such as framework state constraints. An automated repair technique that exploits the similarities between framework state constraints can fix violations of these constraints in an accurate and general manner.*

1.3 Goals and Evaluation Criteria

1.3.1 Challenges of Framework Application Debugging

The goal of the debugging study was to identify the challenges that experienced developers face in the framework debugging process. At a high level, I created debugging scenarios from two frameworks and had multiple developers debug the scenarios. To guide the creation of debugging scenarios, I performed a preliminary study to investigate the different ways that framework applications present problems to developers. Then, to increase the chance that the scenarios represented problems that real developers encounter, I created the scenarios using questions from StackOverflow. I had multiple developers debug each scenario, and recorded their actions during the debugging process. I then analyzed the recordings and compiled them to identify the challenges that developers faced during the framework application debugging process. The results of from this analysis are discussed in Section 3.2.

1.3.2 Specification Language

My investigation into the challenges of framework application debugging found that developers especially struggled to repair state-based constraints in a framework application. This discovery led to the conclusion that developers would benefit from an an automated repair tool that fixes state-based constraints in framework applications. The current approaches for repairing framework applications require either both the framework and application to be written in a language that uses contracts [93] or use a set of transformations derived from an empirical study into common root-causes for Android application crashes [86]. These approaches require either a large overhead in using contract languages, or only fix application problems that cause the program's execution to stop. Instead, a technique should be able to detect more error conditions, including both silent failures and stopped executions.

An important part of repairing state-based framework problems is to identify applications with those problems, and to identify the location of the problem. To identify a problem, FrameFix needs a specification of the state-based interaction rules between the framework and framework application. I created a language to specify a static analysis that FrameFix uses to determine whether an application violates a state-based interaction rule. I evaluated the specification language by checking if the specification language could handle an example of each object protocol category in an object protocol taxonomy [13]. I chose to evaluate the specification language on the object protocol taxonomy to demonstrate that the specification language can generalize to the different type of framework state-based problems that the tool could encounter. The results of this evaluation are discussed in Section 4.3 and Section 4.5.

1.3.3 FrameFix

The last chapter of this dissertation describes the complete repair technique, with a focus on the novel repair process. The goal of the evaluation was to show that the repair technique could fix a wide range of plugin problems and that the technique works for problems in real applications. Thus, I evaluated FrameFix, the automated repair technique for plugins, with respect to three claims:

1. that the directive violations covered by FrameFix apply to a diverse set of constraints in the framework, which I evaluated through a manually created dataset of applications with constraint violations
2. that FrameFix is able to fix problems in real applications, which I evaluated with applications taken from the FDroid dataset [4].
3. that FrameFix can repair known problems in a diverse set of applications, which I evaluated by automatically injected problems into applications in the FDroid dataset.

The results of these experiments are elaborated in Section 5.4.

1.4 Contributions

The main contributions of this dissertation are as follows:

- A categorization of how violations of framework directives are presented to developers.
- An enumeration of the benefits and challenges in debugging misuses of framework APIs.
- A reproduction package and study notes from two human studies of debugging directive violations in the Android Fragment class and the Robotic Operating System (ROS).
- A specification language for specifying static analysis checks of state-based directive violations
- An automated repair technique based on frameworks' heavy use of object protocols and inversion of control to guide the automated repair process.
- An instantiation of the technique for Android applications, called FrameFix.
- An evaluation of FrameFix on F-Droid, a set of 1,964 open-source Android applications.

The goals presented in the thesis statement were to gain insights that generalize to a wide variety of frameworks and to demonstrate the accuracy of the specification language and repair technique. The following is a high-level summary of the ways that this dissertation demonstrates these goals.

- I investigated the Android `Fragment` class that investigated the ways that violations of expected framework-framework application interactions are presented to developers. By understanding the set of ways that violations are presented to developers, I was able to create a diverse set of debugging study scenarios that investigated the different cases.
- The debugging study covered framework-plugin interaction violations from two different frameworks.
- The debugging study included at least ten different developers for each framework. De-

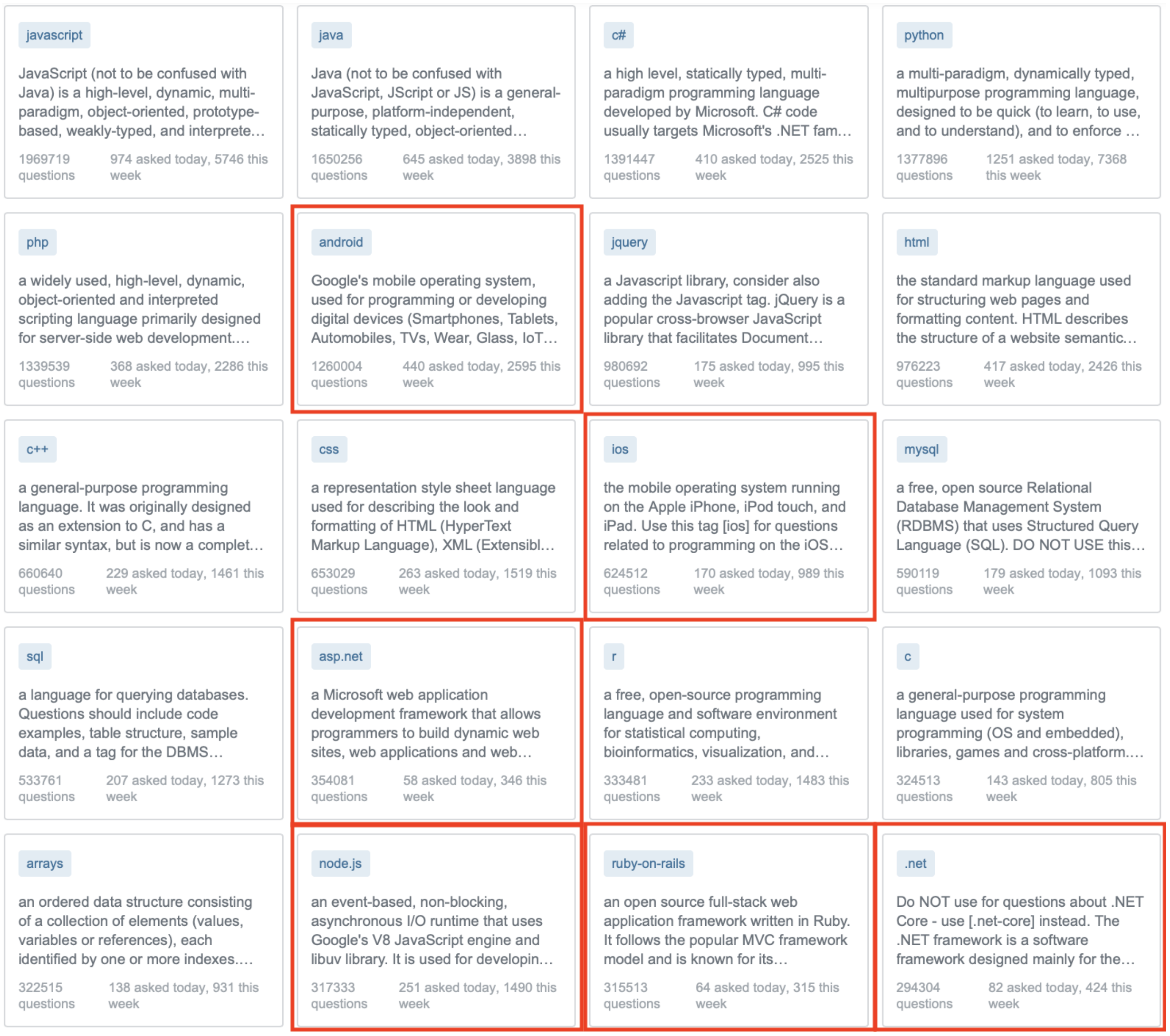
velopers solved violations in both frameworks, which ensures results from a more diverse set of developers. At least four participants attempted to repair each violation, providing multiple data-points from which to draw conclusions.

- I evaluated the expressiveness of the specification language on every category from a taxonomy of object protocols (excluding the catch-all 'other' category), created in prior research [13]. This evaluation was done to demonstrate that the specification language could apply to a diverse set of specification requirements.
- Both the specification language and repair technique, FrameFix, were evaluated on applications in the F-Droid dataset [4], an open-source collection of 1,964 Android applications, to demonstrate that the tools apply to a large percentage of real-world applications.

This dissertation finds that developers encounter debugging challenges related to unique aspects of framework development: inversion of control and the heavy use of object protocols. It also finds that a repair technique based on the unique aspects of framework development can fix plugin problems.

1.5 Outline

Chapter 2 discusses background and related work to this dissertation. Chapter 3 presents the debugging study to better understand the challenges that developers face when debugging framework application issues. Chapter 4 presents the specification language and static analysis used to identify state-based issues in framework applications. Chapter 5 presents the automated repair technique for state-based framework application bugs. Chapter 6 presents the conclusions and final thoughts of this dissertation.



javascript

JavaScript (not to be confused with Java) is a high-level, dynamic, multi-paradigm, object-oriented, prototype-based, weakly-typed, and interprete…

1969719 questions | 974 asked today, 5746 this week

java

Java (not to be confused with JavaScript, JScript or JS) is a general-purpose, platform-independent, statically typed, object-oriented…

1650256 questions | 645 asked today, 3898 this week

c#

a high level, statically typed, multi-paradigm programming language developed by Microsoft. C# code usually targets Microsoft's .NET fam…

1391447 questions | 410 asked today, 2525 this week

python

a multi-paradigm, dynamically typed, multipurpose programming language, designed to be quick (to learn, to use, and to understand), and to enforce …

1377896 questions | 1251 asked today, 7368 this week

php

a widely used, high-level, dynamic, object-oriented and interpreted scripting language primarily designed for server-side web development.…

1339539 questions | 368 asked today, 2286 this week

android

Google's mobile operating system, used for programming or developing digital devices (Smartphones, Tablets, Automobiles, TVs, Wear, Glass, IoT…

1260004 questions | 440 asked today, 2595 this week

jquery

a Javascript library, consider also adding the Javascript tag. jQuery is a popular cross-browser JavaScript library that facilitates Document…

980692 questions | 175 asked today, 995 this week

html

the standard markup language used for structuring web pages and formatting content. HTML describes the structure of a website semantic…

976223 questions | 417 asked today, 2426 this week

c++

a general-purpose programming language. It was originally designed as an extension to C, and has a similar syntax, but is now a complet…

660640 questions | 229 asked today, 1461 this week

css

a representation style sheet language used for describing the look and formatting of HTML (HyperText Markup Language), XML (Extensibl…

653029 questions | 263 asked today, 1519 this week

ios

the mobile operating system running on the Apple iPhone, iPod touch, and iPad. Use this tag [ios] for questions related to programming on the iOS…

624512 questions | 170 asked today, 989 this week

mysql

a free, open source Relational Database Management System (RDBMS) that uses Structured Query Language (SQL). DO NOT USE this…

590119 questions | 179 asked today, 1093 this week

sql

a language for querying databases. Questions should include code examples, table structure, sample data, and a tag for the DBMS…

533761 questions | 207 asked today, 1273 this week

asp.net

a Microsoft web application development framework that allows programmers to build dynamic web sites, web applications and web…

354081 questions | 58 asked today, 346 this week

r

a free, open-source programming language and software environment for statistical computing, bioinformatics, visualization, and…

333481 questions | 233 asked today, 1483 this week

c

a general-purpose programming language used for system programming (OS and embedded), libraries, games and cross-platform.…

324513 questions | 143 asked today, 805 this week

arrays

an ordered data structure consisting of a collection of elements (values, variables or references), each identified by one or more indexes.…

322515 questions | 138 asked today, 931 this week

node.js

an event-based, non-blocking, asynchronous I/O runtime that uses Google's V8 JavaScript engine and libuv library. It is used for developin…

317333 questions | 251 asked today, 1490 this week

ruby-on-rails

an open source full-stack web application framework written in Ruby. It follows the popular MVC framework model and is known for its…

315513 questions | 64 asked today, 315 this week

.net

Do NOT use for questions about .NET Core - use [.net-core] instead. The .NET framework is a software framework designed mainly for the…

294304 questions | 82 asked today, 424 this week

Figure 1.5: The top 20 tags on StackOverflow on March 4th, 2020. The six with red boxes around them are associated with frameworks. Frameworks are also the second largest tag category in the top 20 (11 — programming languages, 6 — frameworks, 1 — database systems, 1 — libraries, and 1 — programming concepts).

Chapter 2

Background and Related Work

In this section, I discuss topics that will be helpful for understanding and to situate the contributions presented in this dissertation in the literature. In this section, I first introduce important terms from the two frameworks I used in my dissertation (Section 2.1). Next, I introduce directives in Section 2.2, which I used to select general plugin problems for this work. The next section (Section 2.3) discusses architectural mismatches. Then, I discuss the related research into object protocols in Section 2.4, an important and well-studied aspect of framework programming that has a significant number of prior investigations. Object protocols are both referenced in the debugging study (Chapter 3), and in the FrameFix repair process (Chapter 5). Work in object protocols also provides context for my specification language (Chapter 4). The next topic is a brief introduction into research on the debugging process (Section 2.5.1), to describe studies related to my framework debugging study (Chapter 3). I end the background section by introducing automated program repair (Section 2.5.2) and previous work into Android and plugin repair approaches (Section 2.5.3) to provide context for my automated repair technique (Chapter 5).

2.1 Introduction to the Frameworks Used in this Work

To ensure that this work drew conclusions that apply to real frameworks, I both performed the debugging study on and built a prototype of the automated repair approach for real frameworks. Here I provide a brief introduction to the two frameworks I used, Android and ROS, to familiarize the reader with applicable terms that are used in later sections. While the list of terms for both frameworks is incomplete, they are terms that are used to explain the developer challenges and constraints imposed by frameworks that appear in later chapters. I revisit these frameworks in Section 3.1.1, where I explain the reasons for selecting these frameworks for the debugging study.

Android is a framework for building mobile applications, most notably the applications used on Android phones. In an Android application, the **`Activity`** class is the main entry point. An **`Activity`** may contain multiple **`Fragments`**. **`Fragments`** are reusable subcomponents of an **`Activity`** that allow for further organization of an application's logic. **`Intents`** are the objects that Android passes between applications, which facilities the communication between different applications. An **`OptionsMenu`** is the class that encapsulates the main menu of an Android application. **`OptionsMenus`** typically contain the ability to adjust application-wide settings, but the content of the `OptionsMenu` is left to the discretion of the plugin developer. A `View` is the class that dynamically encapsulates the behavior of a section of a screen the plugin presents to the user. When the `View` can be defined statically, the `View`'s contents are defined in a *layout file*, which is an XML file that defines the `View`'s subcomponents. The plugin developer can then reference the layout files using a resource identifier, for example **R.layout.sample_-view**. An Android application is deployed as one or more executable files called APKs (Android Packages).

ROS, the Robot Operating System, is a framework for building robotic applications. In a ROS application, different conceptual parts of the robot are organized into **`Nodes`**. The **`Nodes`** communicate with each other by passing **`Messages`** from one `Node` to another. A `Service` is a set of two messages: a request and a reply. This class encapsulates requesting information from a `Node` and then the `Node` responding to that request. When a response is not required, ROS supports indirect messages (th publish-subscribe pattern) with the `Topic` class. Instead of two **`Nodes`** connecting directly to one another, a `Node` can follow **`Messages`** posted to a topic. When another `Node` posts a `Message` to a `Topic`, all followers of the topic are sent the `Message`. ROS application are started from the configuration settings in a *launch file*. The launch file defines the starting **`Nodes`** and communication paths for a ROS application, although this initial setup may change at runtime.

2.2 Directives

I designed the debugging investigation and the automated repair of state-based framework applications to target framework-specific states, in a way that should be useful across many applications built against a given framework. The problems investigated in this dissertation meet the following criteria:

1. The problems are non-trivial. Trivial problems may make the debugging and fixing process so easy that there are no possible improvements on the process.
2. The problems are not application specific. While application-specific problems are interesting for case studies, results from a debugging study on application-specific problems are less likely to generalize. Likewise, a tool that only repairs a set number of application-specific issues are only useful for that subset of problems. Instead, problems that developers may make then repeat in other cases were of more interest.
3. The problems are testable and deterministic. While fixing non-testable or non-deterministic problems are important, it is easier to conduct the investigations if a test could deterministically demonstrate and reproduce the problem.

To meet these criteria, I focused on directives. *Directives* are unexpected or surprising API specifications for how to use a class or method correctly [27]. I was lenient with the "unexpected or surprising" requirement and included any framework directive that was not already checked by the type system or language-enforced runtime checks (e.g., an array out of bounds error). One type of directive I included were object protocol specifications. For example, *a `Fragment` cannot call `setArguments` after the `Fragment` has been initialized.* This means that the `Fragment` object instance can only call the `setArguments` method when it is in the uninitialized state. Directives are often documented by framework developers in READMEs, tutorials, and comments, but they may only be documented in the source code (often by throwing an error when a certain condition is met).

Prior studies into directives have proposed three different directive classification schemes. One classification scheme is based on the abnormal aspect specified in the directive (e.g., the calling restrictions, method limitations, or side-effects) [27] while another focused on the segment of code covered by the directive (e.g., line, method, or object) [69]. The third classification scheme is based on the keywords in the directive (e.g., directives with the word 'error' or 'illegal' are grouped into the same category) [61]. Another study showed that developers were more likely to successfully debug applications with directive violations when presented the directives important to the problem's context [28]. Other researchers have investigated certain directive categories: directives specifying how to extend objects to implement the framework [18] and parameter usage constraints [99]. One study found that about 58% of sentences in API documentation contains constraints [5]. My work adds to the collective knowledge on directives by showing that when developers are debugging plugins, notifying developers of the violated directive that is causing the problem does not lead to quickly fixing the problem. Instead, developers also need guidance on how to fix the directive violation.

2.3 Architectural Mismatch

When building framework applications, one source of difficulty is the assumptions that the framework imposes on plugins. Multiple approaches have investigated how to enforce framework constraints on plugins. Design patterns provide a way to catalog usage patterns in frameworks, which can be checked for compliance [32]. The Framework Constraint Language provides a function-based specification technique for enforcing structural constraints [44, 45]. Another specification technique, the Fusion analysis tool, specifies how to check object relationships in a framework [46], and the constraints imposed by declarative artifacts [47]. Another approach for specifying framework interactions with plugins include the text based Hook specification, where the interaction points between a framework and framework application are specified in a text format that describes the requirements of using the hook and how the hook affects the rest of the application [36]. FRED detects violations of framework interactions specified in diagrams [41]. Due to their design for a broader set of specifications than my specification language, previous techniques for specifying framework constraints require too much specification overhead for my use cases (further discussed in Section 4.7), and none of these technique try to automatically repair the detected problems.

Other research has investigated the challenges that arise when building software composed of components with differing assumptions. Researchers have proposed a technique to evaluate the architectural mismatch between a proposed framework application and the possible frameworks in which the application could be implemented [74]. This system compares the events the framework application needs to respond to with the events addressed by the framework's inversion of control mechanisms. The researchers found that the mismatch evaluation technique predicted the difficulty of creating applications in the proposed framework for two applications. A case study on combining reusable software components found that the assumptions made by components led to difficulties combining the different components into a software product [37]. Researchers have found message-passing incompatibilities are a main contributor to architectural mismatch in Service Oriented Architectures (SOA) [16]. Another study formalized architectural mismatches in cyber-physical systems as conflicts of automatically deduced invariants of the different system components [51]. While my work contributes to architectural mismatch literature by investigating the challenges that plugin developers encounter with framework constraints, it focuses on developer debugging issues, and not directly on the difficulties of combining components with different assumptions.

2.4 Object Protocols

When building applications for real-word situations, the developer or system must keep track of multiple *states*. For example, what needs to be saved so a user can can restore the application later? Which application needs to handle the current user interaction? Which window should be closed or saved for later? These various concerns are stored as states, so that the application can perform the correct action at the correct time. States are also important for individual objects to ensure that the objects produce the correct behavior. Objects need to be in the correct state before certain methods are called, to ensure that the method does not produce an error or nonsensical result. These state requirements for certain methods of an object are known as *object protocols* [13]. When developing an application with a framework, object protocols frequently arise, because of the way that frameworks are often implemented. Frameworks often implement extensibility by invoking part of the plugin at the right time [8]. For example, the Android framework invokes the **`onCreate`** method of an **`Activity`** class to perform any plugin-specific initialization steps when the application has been started by the user or another application. This invocation usually passes state information to the application, both by what method is called (for example, if an initialization method is called, the developer is inclined to assume that most of the application will either be uninitialized, or in a relatively refreshed state) and the state of objects that are passed to the method call. This object state change at different points in a framework application is used by the FrameFix repair process.

A human study has found that developers spend time trying to figure out which states are possible in an object protocol and how to cause certain state transitions in object protocols [85]. One approach to add tool support for object protocols is *typestates*. *Typestates* treat each important state of a programming abstraction as a separate but related type. A tool or compiler can then check whether method calls are allowed for a certain type of an object. As an example, *typestates* have been used to determine whether pointers can be dereferenced by treating an invalid pointer (a pointer that points to uninitialized memory or freed memory) and a valid pointer (a pointer that points to value) as two different types [84]. The typestate checking system then flags any pointer dereference on an invalid pointer type. Typestates have also been used to track different states of an object, assigning different types to different possible object variable values [29]. Another approach extended typestates to track the type of subtypes and the reduction of possible states to important states. Instead of treating each possible number of characters in a input buffer as a different state, the possible states reduced down to two possible states: 1) the empty state, where no content was in the buffer, and 2) the full state, where the buffer contained content

[14]. Another approach used access permissions (restricting aliases with read/write permissions) to improve the the soundness of typestate analysis [15]. While general typestate techniques can address changes in object protocol state, none of the general typestate techniques were designed for frameworks. Since internal framework code often changes the states of objects, there may not be a direct application of the technique to plugin code, or the technique may only address a small subset of the challenges of plugin development.

Session Types are a specification approach to sequenced interactions [42], and are a straightforward way to apply the concept of object protocols to functional programming. Prior research has demonstrated how session types can be applied to the interactions of multiple actors at once [43]. Dezani-Ciancaglini and de'Liguoro present a survey of session type papers and include demonstrations of how session types can be used to maintain values in sessions (correspondence assertions) and how session types can be applied in functional and object oriented languages [30]. Session types would have limited applicability to FrameFix because FrameFix is focused on actions that are allowed or disallowed in certain states and not the communication sequences of different components.

2.5 Program Repair

2.5.1 Debugging Process

Debugging is defined as the process of identifying and correcting the cause of a software failure [98]. Debugging is a wide field and includes many different subfields, including investigations into the debugging process and debugging tools. Prior work in debugging into the debugging process found that locating the failure, referred to as fault localization [24], was the main cost of the debugging process [90]. More recent investigations have found that developers use scent finding when locating the failure [57] and the ability to easily answer dataflow questions can significantly reduce debugging time [54]. Another study has found that developers encounter design decisions during the debugging process, such as choosing the correct location to fix incorrect data passed between multiple components [70].

2.5.2 Automated Program Repair

Automated Program Repair is an area of research that focuses on removing identified software failures through computer-generated patches. One family of automated repair approaches is generate-and-validate, typified by approaches like GenProg [59, 94]. Figure 2.1 illustrates a generate-and-validate repair process. The process uses a program and a set of specifications as input and produces a fixed program as output, in the successful case. When the process is unsuccessful, the process ends, usually due to a set timeout, without producing a fix In the generate-and-validate repair process, a repair technique creates possible repair options and then evaluates those possible repairs on a set of program specifications, such as test cases. These specifications are used to determine if the proposed repair successfully fixes the problem. In the case where techniques use test cases as specifications, the test case set usually consists of one or more failing test cases and multiple passing test cases. The techniques use the failing

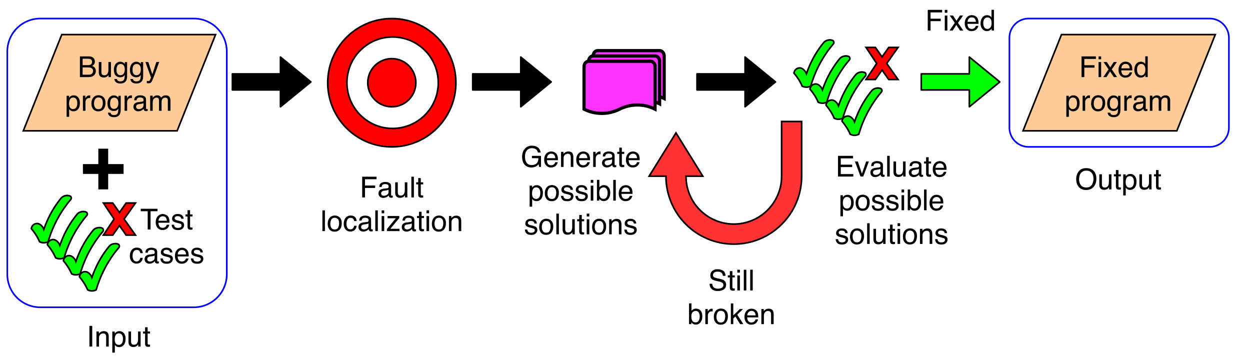


Figure 2.1: A diagram of a successful generate-and-validate automated repair process. The inputs are a program that needs to be fixed and a set of test cases that demonstrate the problem. The repair process localizes the fault and possible fix locations. Then it generates multiple possible fixes. These fixes are tested against the test cases. If none of the proposed fixes pass all test cases, then the technique generates more possible fixes. If one of the fixes pass all test cases, then it is considered a valid solution.

test cases to determine the likely problem location and to validate when the problem has been fixed. The techniques use the passing test cases to determine if possible fixes retain the required functionality of the application — if a fix causes one of the tests in the originally passing test set to fail, then the fix is invalid.

Automated repair techniques currently fall into three families of fix generation strategies. FrameFix is part of the *heuristic* repair family, which generates possible fixes by transforming the program through a a set of possible changes, often referred to as mutation operators. These mutation operators are usually a set of ways to change code (add and remove sections of code) or a set of common fixes for a known domain [53]. The typical heuristic repair process starts by estimating the possible change location and then generating one or more possible fixes. Some examples include AE [95], RSRepair [78], SPR [64], and SapFix [66].

In contrast to the heuristic generation approach, another major family is semantic-based repair techniques, such as Angelix [68], SemFix [72], DirectFix [67], Qlose [26] and S3 [58]. *Semantic-based program repair* uses dynamic semantic analysis, commonly symbolic execution, and a set of test cases to infer desired program behavior. These techniques treat the program as an equation and use the test cases as constraints on the equations. Semantic-based techniques produce a repair by solving the equation, adjusting the program so that the program produces all desired outputs from the provided inputs. These approaches are different from FrameFix, because FrameFix is not a semantic-based technique. Semantic techniques are currently either tailored to a subclass of problems (such as boolean or integer problems) that do not directly completely cover directive issues, or require multiple fix examples to create a repair specification to solve for (such as Refrazer [80]), which FrameFix does not require.

The final family is *learning-based repair*, where the techniques use machine learning and past fixes to propose repairs. These techniques turn past fixes into a format that can be handled by a classifier, for example by counting the number of operators added or removed in a fix. After training the classifier on these past fixes, when the classifier is given a new problem, the classifier generates new fixes based on past fixes applied to similar sections of code. Examples from this group include Prophet [65] and DeepFix [40]. FrameFix does not learn from past repairs, so it is not a learning-based repair technique.

Currently, there are two main approaches to determine whether a generated repair successfully fixes an issue. Dynamic validation runs the program to validate that the repair was successful. Most techniques perform dynamic validation with a set of test cases that usually consists of one or more failing test cases and multiple passing test cases. The techniques use the failing test cases to determine the likely problem location and to validate when the problem has been fixed. The techniques use the passing test cases to determine if possible fixes retain required functionality. If a candidate fix causes one of the tests in the originally passing test set to fail, then the fix is invalid. Most of the previously mentioned techniques use this approach. FrameFix uses dynamic validation when tests are available, after performing static validation.

The other main approach is static validation. This approach analyzes the code without running it to determine if the repair succeeded, usually with static analysis. One static validation approach that is similar to FrameFix is Phoenix [12]. Phoenix uses static analysis to identify faults and validate fault repairs. FrameFix differs from Phoenix in the types of bugs that each repair technique fixes and the general repair approach, Phoenix fixes errors caught by Findbugs [10] using a programming-by-example approach to synthesize repairs, while FrameFix avoids having to specify an example for each detected problem type (directive violation). Another static repair technique is FootPatch [88] which uses static analysis to identify heap-based problems. This technique uses static analysis, similar to FrameFix, but is focused on repairing a different subset of problems. A third static validation technique is Getafix [11], which generalizes a set of collected bug fixes to AST-based repair patterns. Getafix then applies those patterns to other detected problems. While FrameFix also uses static validation, Getafix repairs only uses local information to detect and fix the problem. Other approaches in this category include the abstract interpreter approach of Logozzo and Ball [63], a formal verification approach (Maple) [73], and a linear temporal logic approach by Jobstmann et al. [49].

2.5.3 Android and Framework Repair

Recent work on the Android framework has categorized a large number of Android exceptions and extracted common repair patterns for problems that produce exceptions [33]. Some of these repair patterns are useful for an Android specific implementation of FrameFix. Droidx repaired crashing Android applications using manually created patterns [86]. Droidx's repair patterns served as a source of inspiration for some of the framework repairs in FrameFix. This approach differs from FrameFix in that the repair process requires a recorded event sequence to produce a crash, and repairs were manually validated, — either by comparing the intermediate code change to developer changes or by running the application to determine if the crash was fixed. In contrast to these approaches, FrameFix was designed to be framework independent and thus handles a wider set of framework issues than those handled in the Android repair techniques.

Other research has taken a different approach to repairing framework applications than FrameFix. One approach to framework repair is to use contracts and dynamically created object behavior models to guide repairs [93]. This approach requires a significant developer investment to work for most frameworks, since both the framework and application would need to be written in a language that uses contracts. FrameFix instead only requires specifying the directive to check and how to translate the directive into code if a custom check is required. Another tool, SemDiff, fixes out-of-date API calls by recommending methods that were added in the version where the API call became out-of-date [25]. This tool addresses a different subclass of API problems than FrameFix.

Researchers have modeled the Android callback system for static analysis. One study evaluated how the changes between different versions of Android can be used to detect method call ordering bugs in callbacks [96]. Another automatically generated interprocedural call graphs from callbacks in an application [75]. An investigation into popular Android static analysis research tools found that incorrect modeling of the Android callback sequence can lead to unsound analysis results [91]. Instead of focusing on accurate static modeling of Android callbacks, this work uses the similarities between callbacks in different applications for repair.

Chapter 3

Framework Application Debugging

Debugging software is a common and time consuming task. The difficulty of debugging a software application has spawned numerous research areas. Some examples include debugging studies [54, 56], debugging strategies [97, 98], and automated debugging assistance (such as fault localization[7, 52]). Yet, debugging is still a difficult and time-consuming task. While studying the general debugging process has produced promising results, it is possible to produce scientific insights by investigating interesting programming situations that are likely to present unique debugging challenges. Framework application debugging is one subset that further study could lead to more insights. I hypothesize that the unique aspects of creating framework applications, discussed in Section 1.1.3, are likely to present unique debugging challenges and may also present unique benefits for the debugging process. Therefore, I performed an exploratory study to observe what debugging challenges developers encounter while debugging framework application problems. In this study, developers were given a debugging task — a framework application that contained a testable problem — and were asked to fix the problem. Since many qualitative study techniques are based on other forms of data collection than debugging trials, such as interviews, I performed the debugging study using a mixed methods approach. The mixed methods approach used techniques from both grounded theory [20], and qualitative content analysis [82]. These approaches were used to guide both the design of the study, as well as the analysis of the results. From this study, I found that developers have difficulty understanding how the framework has altered the state of variables passed to the problematic method, but that the similar structure of applications that used the framework aided developers' debugging efforts.[1]

[1]This chapter references work from *A Qualitative Study on Debugging* [23].

3.1 Methodology and Preliminary Study

I performed an exploratory study into framework API debugging to understand the unique benefits and challenges that framework application developers encounter during the debugging process. The exploratory study consisted of multiple participants performing lab-created debugging tasks. The main benefit of this approach is that it is possible to observe multiple developers approaching the same task, reducing the chance that observations are biased by the performance of an individual developer and increasing the chance that patterns are observed across multiple participants. The main risk of this approach is that the conclusions may not apply to real framework debugging scenarios, since the developers would not be debugging problems that they created in their own applications, but instead debugging problems in the application given to them for the scenario. To mitigate this risk, I took steps to make the tasks as realistic as possible, basing the tasks off debugging scenarios encountered by actual developers in practice. I further elaborate on the methodology I used to select the frameworks to study Section 3.1.1 and create the tasks in Section 3.1.2. The results from an investigation into the possible types of directive violations, used to inform the chosen tasks, are presented in Section 3.1.3. Once I created the tasks, I recruited study participants and conducted debugging trials, using the procedure described in Section 3.1.4. After recording participants performing the tasks, I coded the recordings using an iterative process. First, I coded the interesting actions of the first few participants in each framework that provided insight on the problem solving process — for example, I recorded if a participant ran the debugger or made a problem related statement but I did not include if a participant accidentally clicked on a wrong tab. Then, I defined coding frames of the interesting actions using qualitative content analysis, a technique that condenses verbal or visual data into important topics [82], and used the coding frames to code the recordings. After I finished coding, I performed theoretical sorting to condense the coded data into the benefits and challenges of framework debugging [20]. A summary of the coding process is described in Section 3.1.5.

3.1.1 Frameworks in the Study

This section provides a brief introduction to the two frameworks I used in the debugging study to explain my motivation for selecting these frameworks. This section complements the terms-of-interest for each framework covered in Section 2.1.

I focused the debugging study on scenarios from two frameworks: Android (v. 5.0 Lollipop–API 21, with a focus on the `Fragment` class), and the Robotic Operating System (ROS) (Kinetic Kame). The goal was to select two distinct frameworks to collect observations that do not apply to a single framework or subset of frameworks. Thus, I selected two frameworks with different application domains and architectures.

Google's *Android* [38] provides a Java framework for developing mobile applications. Android is a widely-used, mature framework, released in 2008 (over seven years at the time of the study). In the Android framework, the `Fragment` class represents a reusable component of an Android application's user interface. A picture of an Android `Fragment` in an example Android application is shown in Figure 3.1,[2] to illustrate its usage. I began with the Android framework

[2]Taken from https://developer.android.com/guide/components/fragments

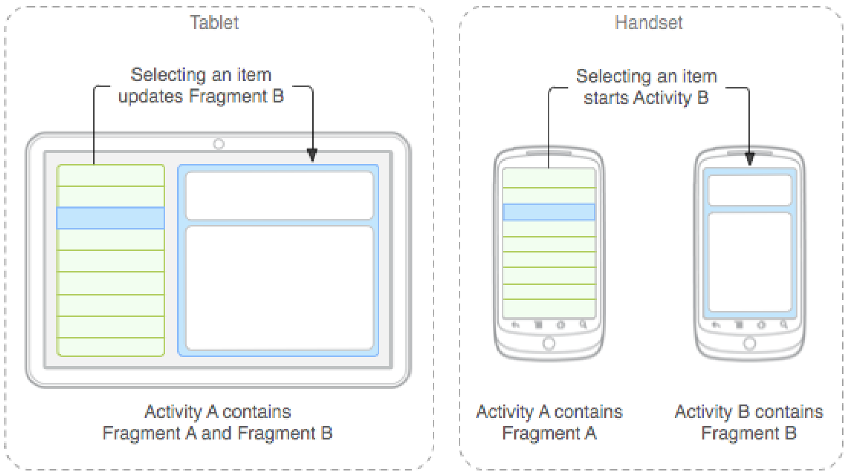

Tablet
Selecting an item updates Fragment B
Activity A contains Fragment A and Fragment B
Handset
Selecting an item starts Activity B
Activity A contains Fragment A
Activity B contains Fragment B

Figure 3.1: This diagram shows an example of the `Fragment` class, taken from the Android development documentation. This diagram demonstrates how the `Fragment` class is a reusable subcomponent of an `Activity`. An `Activity` is the class that controls the lifecycle of an Android application and is often the class that controls most of the communication between the framework and application specific code. The lifecycle of Android application is the major, state changes possible for an application — when the application is created, when the application is paused, when the application is destroyed, etc.

for three reasons:

- it is widely-used and well-established
- it makes heavy use of inversion of control and object protocols, key features that differentiate frameworks from libraries
- multiple developers express difficulties with the framework, as demonstrated by the many questions on StackOverflow about it.

After selecting the Android framework, I decided to perform an in-depth analysis into the directives of one class, to collect a diverse set of directive violation consequences. I queried StackOverflow for questions by class, finding that the `Activity` class had the highest number of posts and the `Fragment` class had the second highest number of posts. After investigating the questions for both, I focused on the `Fragment` class because the *Activity* keyword often turned up questions that were not related to the class. The `Fragment` class also requires developers to correctly implement the `Fragment` lifecycle inside of the `Activity` lifecycle, a fairly complicated workflow, which I speculated might lead to more or more interesting debugging challenges.

To increase the chance that the results of the study do not apply to a small subset of frameworks, I selected ROS as a second framework for study. The *Robot Operating System* [79] (ROS) is a framework for creating robotics applications, with a focus on the communication between various robotic components. ROS applications are built as a collection of nodes that communicate in an event driven model. ROS is also a mature framework and had been released for over eight years at the time of the study (released in 2007). ROS is written in both C++ and Python, but I focused on the C++ version of ROS in the ROS scenarios. The criteria for the second framework was that it should focus on a substantially different domain than Android, and have a different framework architecture. ROS satisfies these criteria, because: (1) ROS is designed for robotic applications instead of mobile applications, (2) ROS uses an event based architecture instead of the tiered architecture of Android, (3) ROS is written in C++ instead of Java, and (4) ROS has a smaller user base, but still sufficient users that I could find experienced participants for the study.[3]

3.1.2 Task Creation Methodology

The debugging tasks that developers solved consisted of framework directive violations. *Framework directives* are possibly surprising statements in a documentation source about how to use the framework (e.g., "`setHasOptionsMenu(true)` must be called to execute an overridden `onCreateOptionsMenu` method") [27]. Focusing on directive violations was key to this study because violations thereof capture mistakes specific to framework programming rather than bugs related to plugin logic. This moreover improves the chances that the results generalize to other framework API misuse errors. Framework directives are also likely require a non-trivial debugging process to fix, due to their surprising nature, increasing the chance of the study finding new insights on the application debugging process.

The process of extracting directives for both frameworks was performed with the assistance of one of my co-investigators, since having another person check the directive increases the validity that it is a directive. The process for extracting the directives from documentation consisted of one investigator extracting directives from the documentation and another investigator double-checking the extracted directives, similar to the coding process in prior work [83]. Both investigators extracted some directives and double-checked other directives. For the purposes of this study, I was lenient on the definition of "possibly surprising" and included any developer guidance that was not already checked by the type system or did not focus on the possible null value for a parameter. This means that extra constraints added by the framework on the framework application were included in the directive set. However, I restricted the directive set to directives that were testable, since directives that only cover recommended best practices may not produce a clear problem for developers to debug. Thus, in this study, the term directive violation refers a section of code or application that does not conform to a testable specification requirement.

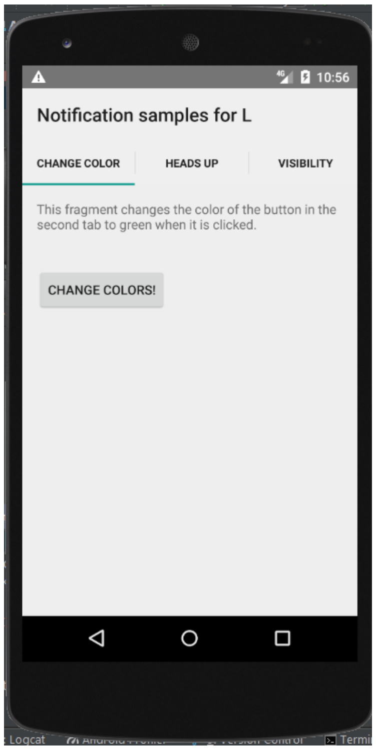
To inform the task selection, I first investigated the consequences of violating Android `Fragment` directives and how those violations were presented to developers. I collected 45 Android `Fragment` directives from three official documentation sources, in order: (1) 11 from the `Fragment` page in the developer's guide,[4] (2) 19 from the `Fragment` API page,[5] and (3) 15 from the `Fragment` class's source code. If directives were found to occur in two documents, I only included the directives once and listed them as occurring in the first document in the order shown above. I created violation scenarios for each directive through a manual

[3]This chapter references work from *A Qualitative Study on Debugging* [23]. Code associated with the project can be found at https://www.dropbox.com/sh/9b7ayp25oa248dn/AAB88KclAKjWxp3gft2pYgE3a?dl=0.

[4]developer.android.com/guide/components/fragments.html

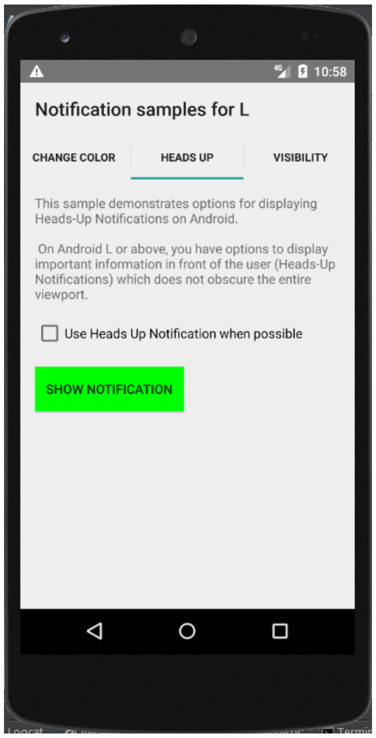
[5]developer.android.com/reference/android/app/Fragment.html

process of violating the directive and then confirming the directive violation, either through the scenario's output or through print statements. I then categorized the directives by the directive violation consequence, or the effect on the application that the developer sees when that directive is violated. The directive violation consequence categories are discussed in Section 3.1.3.



This fragment changes the color of the button in the second tab to green when it is clicked.

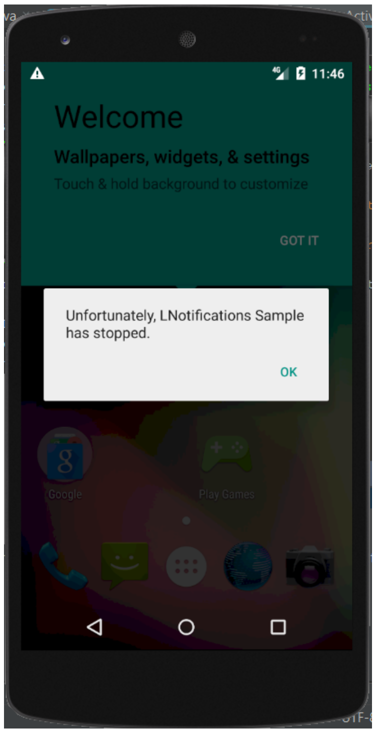
(a) The starting view of Android Task 5. The problem with the task is the behavior of the **CHANGE COLORS!** button.


10:58
Notification samples for L
CHANGE COLOR
HEADS UP
VISIBILITY
Use Heads Up Notification when possible
SHOW NOTIFICATION

This sample demonstrates options for displaying Heads-Up Notifications on Android.

On Android L or above, you have options to display important information in front of the user (Heads-Up Notifications) which does not obscure the entire viewport.

(b) The desired behavior of the application. When the **CHANGE COLORS!** button is pressed, it changes the **SHOW NOTIFICATION** button's color. However, at the start of the task, this functionally only worked if the user pressed the **CHANGE COLORS!** button before opening the **HEADS UP** tab.


11:46
Welcome
Wallpapers, widgets, & settings
Touch & hold background to customize
GOT IT
OK
Google
Play Games

Unfortunately, LNotifications Sample has stopped.

(c) The crash that occurs when participants first open the **HEADS UP** tab and then press the **CHANGE COLORS!** button. Participants were asked to remove this crash so that the button successfully changes colors, even if the participant had navigated to the **HEADS UP** tab first.

Figure 3.2: Android Task 5. Participants were given an application that crashed when participants interacted with the application in a certain way and asked to fix the crash.

To select tasks, I searched for StackOverflow questions that cover Android `Fragment` directives from a wide range of violation consequence categories. I used keywords taken from the directives or error messages produced by an author-written violation of the directive to find the StackOverflow questions. I found seven unique questions that covered directives in different consequence categories and provided enough information to reproduce the problem. These seven questions were used to create seven Android tasks. The seven tasks were created by taking an Android Lollipop (version 5.0) sample application[6] that demonstrated the various notifications available in Android Lollipop and changing the application to encompass the scenarios mentioned in the StackOverflow questions. An example of one of the tasks is shown in Figure 3.2.

For the ROS framework, I extracted 28 directives from two sources: (1) 9 from the official ROS C++ documentation,[7] and (2) 19 from ROS C++ source code.[8] Due to the relatively low number of online questions about ROS, I was unable to collect ROS directive scenarios from StackOverflow. Instead, I choose three directives that represented materially different cases, and manually created tasks for each. I created the first and third task by modifying the TurtleSim scenario,[9] a two node configuration where a virtual turtle in one window moves and publishes the movements so the virtual turtle in the other window mimics the movements. The second task involved a simple, custom-built directory reading application.

In the rest of this chapter, the Android tasks and participants will be prefixed with a “TA” and “PA” respectively. The ROS tasks and participants will be prefixed with a “TR” and “PR” respectively. Table 3.1 lists the number of participants per task and briefly explains the Android and ROS tasks. A link to the Android tasks and steps to recreate the ROS tasks can be found in the readme of https://github.com/cbogart/ViolationOfDirectives.

3.1.3 Directive Categorization By Consequence

To inform the debugging study trials, I investigated the ways that frameworks present errors to developers. This investigation provided a greater understanding of the debugging situations that developers encounter when debugging framework applications, and ensured a diversity of debugging study tasks. For the investigation, I manually violated 41 of the 45 Android `Fragment` directives, and then grouped the violation consequences into categories. 4 of the 45 directives were not violated, because I was unsure how to directly map the directives to code, and the goal of the study was to sample directive violation consequences to understand the different possible categories, not exhaustively catalog all violation consequences. I then collected three ROS directive violations and found that they fit into the previously created categories. Due to the limited investigation, I do not make claims that the categories will generalize to all frameworks. However, the results are useful as an initial investigation into the ways that frameworks present directive violations to developers.

The consequences of violating 41 Android `Fragment` directives are shown in Table 3.2 and explained below. I found that violating certain directives can produce multiple consequences (e.g., a violation can produce a tool warning and crash with reference to the directive), but each consequence is mutually exclusive (the same consequence could not be categorized in multiple categories - an application crash cannot be both classified as crash with reference to the directive and crash without reference to the directive). It was possible to violate one directive in two different ways to produce three possible consequences. I was also able to violate two directives in two ways, each with different consequences.

[6]github.com/googlesamples/android-LNotifications
[7]wiki.ros.org
[8]docs.ros.org/api
[9]http://wiki.ros.org/ROS/Tutorials/UsingRxconsoleRoslaunch

Task	Count	Goal	Violated Directive	Result of Directive Violation
TA1	5	The participant must connect user inputs to the output message when input components initially share the same ID.	Application components must have a unique ID to be referenced individually.	Any attempt to access one of the components with the same ID returned the last component added.
TA2	6	The participant must display the application start time on a tab without a warning.	The application should not pass time data through the constructor.	AndroidStudio displayed a warning and recommended a fix.
TA3	4	The participant must make the framework check for an updated `OptionsMenu`.	The framework only checks for an `OptionsMenu` if the application calls `setHasOptionsMenu(true)`.	The `OptionsMenu` does not appear, although an `OptionsMenu` is defined.
TA4	5	The participant must display the application's `Activity` (the entry point for an Android application) title in a pop-up message on a specific tab.	The `Fragment` could only access the `Activity` if the `Activity` was attached to the `Fragment`, and the `Activity` was not attached.	The application crashed with a notification that the `Activity` was not set.
TA5	4	The participant must fix a problem that occurs when the application tries to change the color of a button on a tab when the tab had been previously accessed.	A tab's arguments can only be set *before* the tab is accessed.	The application crashed, stating that arguments can only be set before the tab has started.
TA6	3	The participant must change a specified **`ContextMenu`** to an `OptionsMenu`.	Items should be added to the `OptionsMenu` in the `onCreateOptionsMenu` method.	The `OptionsMenu` would not appear.
TA7	5	The participant must fix an incorrect `inflate` method call.	In the application's current state, the last parameter of the `inflate` call must be `false`.	The application crashed with a stack trace that pointed towards core framework code.
TR1	8	The participant must fix an incorrect **`spinOnce()`** call.	**`spinOnce()`** cannot be used when the framework should perform the callback more than once.	A node in the application would quit unexpectedly without an error message.
TR2	8	The participant must fix an application node's parameter access.	Local namespaces are not checked if a global namespace is used in a parameter search.	The parameter search returned that the parameter does not exist.
TR3	6	The participant must fix an obsolete message type.	An incorrect message type was used for this version of ROS.	The application crashed with an incorrect type declaration error.

Table 3.1: Tasks in the debugging study. TA tasks indicate Android tasks; TR indicates ROS tasks. "Count" shows the number of participants per task. "Goal" indicates what the study participant must do to successfully complete the task. "Violated Directive" is a simplified explanation of the violated directive motivating the task. "Result of Directive Violation" explains how the application presented errors to participants.

Directive Violation Consequence	Count
Compiler Error	3
Crash With Reference To Directive	19
Crash Without Reference To Directive	2
Missing Feature	9
No Obvious Effect	5
Tool Warning	3
Wrong Value Returned	2

Table 3.2: Categorized consequences from violating 41 directives. A directive violation may have multiple consequences, but each consequence is mutually exclusive.

Compiler Error. When these directives were violated, the framework threw a compiler error, preventing the application from compiling. This consequence occurred when invalid semantics produced a directive violation. One example is when the documentation specified that a method could not be overridden. The compiler prevented a developer from overriding this method because the method was declared with the final modifier in the parent class.

Crash With Reference To Directive. When these directives were violated, the application crashed with an exception that notified the user of the directive violation either directly or indirectly. One example of this category is, *`getActivity()` should not be called when the `Fragment` is not attached to the `Activity`*. If this directive was violated, the application crashed with a null return from `getActivity()`. This category contains a high number of directives because all the directives found in the `Fragment` class's code were of this type.

Crash Without Reference To Directive. When these directives were violated, the application crashed with an exception that did not notify the developer that a directive was violated, usually with an error pointing to where the application crashed instead of the location where the application needed to be fixed. Violations in this category occur when a more general exception message is thrown, or violating the directive puts the application into an invalid state and the invalid state is caught in a later line. One example in this category is *when the result of the `inflate` method is used as the return result for `onCreateView`, the last parameter to the `inflate` method call must be `false`*. If this directive was violated, the application would crash with a stack trace that pointed to internal framework code and not the `inflate` line.

Missing Feature. When these directives were violated, the framework application did not produce the intended effect of the relevant section of code (i.e., the code ran without errors but the section of code with the directive violation did not produce the intended feature). The effect did not occur either because violating the directive caused the control flow to change or the semantics were changed. For example, one directive states that *an application will only execute the `Fragment`'s `onCreateOptionsMenu` method if the `Fragment` calls `hasOptionsMenu (true)` in the `onCreate` method*. If the `hasOptionsMenu(true)` call is removed, the `OptionsMenu` will not appear, even if the `Fragment` overrides `onCreateOptionsMenu`.

No Obvious Effect When these directives were violated, the framework correctly performed the intended action of the associated code segment without crashing the application. One example of this category is a directive that states that *if a `Fragment` does not have a user interface (UI), then the `Fragment` should be accessed by **`findFragmentByTag()`***, but the `Fragment` without a UI could be accessed by **`findFragmentById()`** without noticeable consequences. These directives may be for some other non-functional, quality attribute, such as performance or readability, but further investigation would be needed to draw definitive conclusions.

Tool Warning. While the application still compiled with these directive violations, if these directives are violated, the recommended tools for developing applications in the framework (in the case of Android, AndroidStudio, the recommended Android Integrated Development Environment) warn developers about the code, and sometimes recommend a possible fix. For example, the directive, *classes that subclass the `Fragment` class must have a public no-argument constructor* produces a warning when violated. An application will compile if the class subclassing `Fragment` lacks an empty constructor, but AndroidStudio displays a warning in the class's source file.

Wrong Value Returned. When these directives were violated, the application did not crash, but a reference to a part of the application was lost or used incorrectly. Any attempt to use the lost or incorrect reference returned a wrong value. For example, *when a developer dynamically added a UI element, the developer must assign a unique tag to the added UI element.* If the added UI element does not use a unique tag, the new tag overrides the matching tag of a previous UI element. The previous UI element is now unreachable through framework supported methods.

3.1.4 Debugging Task Methodology

I based the methodology of the debugging tasks on other previous studies of developers, such as study by Piorkowski et al. [76]. After I obtained IRB approval for the study, the debugging study process started with a pre-survey to document participants' framework experience. I provided participants a Surface Pro 3 tablet containing the tasks, and I instructed them to perform think-aloud debugging, vocalizing what they thought as they went through the debugging process [71].

I assigned a task to each participant, and asked them to fix the bug. I did not inform participants of the directive violation in the task because I was interested in also studying the fault localization process. If participants finished a task and could stay for another 20 minutes, I asked them to attempt another task. I did this to gain more insight on if certain tasks were hard or if certain participants were much more efficient at all the tasks. All but one participant was willing to spend more than twelve minutes to attempt the task once they started. I initially assigned tasks randomly, but later selected tasks that the fewest participants had attempted, to produce relatively even task coverage. Android participants participated in tasks for about two to three hours in total. Due to removing an unhelpful pre-trial application familiarization period, the ROS sessions lasted around one hour. Participants were permitted to quit at any time (I never stopped them in their work). I allowed participants to search online for anything, including the inspiration for the tasks, but I did not allow them to post questions. While searching online, no participant found the inspiration for any of the study's tasks. During think-aloud debugging, if participants stopped explaining their thought processes, I prompted them through questions like *"What are you thinking now?"* and *"What is your reasoning for that?"* Occasionally, I asked *"Why not?"* if participants commented they would normally do an action, but they were choosing not to. In addition to asking them to vocalize aloud their thoughts and strategies during the tasks, I asked participants about their approach in greater detail at the end of each participant's session.

For the Android study, I recruited a convenience sample of 15 participants. Eleven of the participants had over 2 years of industrial Java or Android experience, and 14 of the participants had more than a year of industrial Java or Android experience. Two participants were professional developers, and 13 were graduate students (12 with professional background). For the ROS study, I collected a convenience sample of 12 participants. Nine of the 12 participants preferred the C++ version of ROS over the Python version. Two of the participants had more than 2 years of ROS experience and 5 of the participants had over a year of experience. Three of the participants were research staff, 8 of the participants were graduate students, and 1 was an undergraduate student. None of the participants did both the Android and ROS study.

I made several procedure changes between the two case studies. For Android, I gave participants time to learn the application before attempting the tasks, while I did not provide a learning period for the ROS tasks. I made this change because I found that participants commonly spent the Android learning period exploring sections of the application that were not relevant to the tasks. In the Android study, I required participants to use the recommended Android Integrated Development Environment (IDE), AndroidStudio, because it provides warnings for directive violations. I did not require participants to use any particular IDE for the ROS tasks, because ROS does not have a recommended IDE.

3.1.5 Coding Process

To analyze the results of the human studies, I coded the actions and statements of participants from the trial video (participants' screens during the trials) and audio (all audio from the trials) recordings. I followed an iterative, open-coding practice based on techniques from Qualitative Content Analysis [82] and Constructivist Grounded Theory [20]. I made adaptations to the techniques, which are described for interview-focused studies, to render them suitable for the human trial method of data collection used in this study.

The iterative aspect of the coding process involved analyzing recordings of the first few participants for important concepts before conducting debugging trials with the remaining participants. After analyzing the recordings from these participants, I then conducted more debugging trials. I alternated between conducting and analyzing trials, and adjusted my coding frame based on the new trial results. As recommended by Contructivist Grounded Theory, I tried to reduce the impact of prior work on my coding process (to avoid analyzing the data with a pre-determined code and thus possibly bias the results) [20]. I thus limited my initial reading of prior work to a few similar studies, and investigated the rest of the related work after finishing the analysis.

The coding process consisted of two different investigators coding different aspects of the study. One investigator conducted and coded the Android trials while another investigator conducted and coded the ROS trials. I allowed the coding categories for both the Android and ROS study to emerge independently, although both were checked and guided by an expert in the area of frameworks, which led to variations in the categories and granularity of the different coding categories.[10]

[10]While the IRB approval prevents releasing the recorded trials, I have released the coding categories along with

The Android coding categories consisted of six categories that focused on the *mental aspects of the debugging process* ('deductions,' 'questions asked,' with a subcategory of 'hypothesis creation,' and 'hypothesis rejection') and the *actions* that participants took ('coding/testing actions,' and 'activities taken to learn the framework or code base'). Finally, a miscellaneous category included behaviors such as reading the scenario specific prompt, or a participant expressing an opinion (such as annoyance with the lack of documentation on a topic). The ROS coding categories consisted of four major categories: 'goals,' 'activities,' 'learning,' and 'miscellaneous.' Each category contained multiple sub-categories, for example, the subcategories under 'goals' were 'fix an identified problem,' 'answer a question,' 'test a hypothesis,' 'find a file,' or 'other goal' (not included in the previous four). The coding categories of both studies were refined over time. For example, the Android coding categories initially contained a 'goal' category, but this was eventually merged with the 'questions' category as participants often expressed goals in question form.

I then condensed the coding data from both studies into the final categories of framework benefits and challenges using theoretical sorting. There are multiple acceptable approaches for theoretically sorting data in constructivist grounded theory [20]. I choose to categorize the results into the benefits and challenges of framework debugging, as this frame provided the most insight.

3.2 Framework Debugging Benefits and Challenges

I first present the challenges that developers faced while debugging the tasks: those relating to dynamic behavior (Section 3.2), static structure (Section 3.2.1), and historical changes to the framework (Section 3.2.2). Next, I present the benefits of framework debugging: dynamic benefits (Section 3.2.3), static benefits (Section 3.2.4), and historical benefits (Section 3.2.5). Finally, I discuss the difficulty of debugging violations in relation to their consequence categories (Section 3.3).

Dynamic Challenges

Throughout the tasks in this study, participants struggled to determine the order in which a framework executes application code, which increased the difficulty of the debugging process. Participants seem to prefer a cause and effect ordering. However, framework application code does not typically follow a sequential ordering, instead it executes plugin code only when needed. This requires code to be structured as non-sequential event handlers. This can create uncertainty about which parts of project-specific code are called and when, and which project method will execute "next." Object protocols exacerbate this issue by requiring participants to understand which states various objects can be in when the framework calls their code.

Inversion of Control. In framework programs, plugin method execution order is not always transparent to the plugin developer. This sometimes led participants to misunderstand an application's control flow, which increased the difficulty of locating the error and fix locations. For example, in the ROS study, participants (PR18, PR20) assumed the framework would not call a

the identity-scrubbed and coded trial summaries to promote future researchers reproducing the study's results at https://github.com/squaresLab/frameworkStudyTranscripts.

section of code, when instead a problem in that code segment caused the application to terminate earlier than expected. In Android, two participants (PA10, PA11) tried to use the debugger to understand control flow, but struggled to do so. Both participants stepped past a plugin method and were unable to figure out how to step back from the framework code into the plugin code. This led participant PA10 to reach incorrect conclusions about which code executed in task TA7. In ROS, while trying to understand how two nodes communicated, PR22 did not realize that a third node linked two other nodes, because the nodes relied on the framework to handle communication. Participant PR22 read four files before understanding how the framework routed the nodes' communication. Another participant (PR23) made incorrect control flow deductions due to the way ROS redirects and filters statements printed to standard output.

Inversion of control also made localizing errors difficult. In Android, one participant (PA5) searched for an error message thrown by the application, but could not find it in the project. The search failed because the error message was generated from core framework code, not plugin code.

Some problems stemmed from participants' uncertainty about the hidden ordering of critical framework activity between events. When participants (PA4, PA12) saw the **`getActivity`** call returned **`NULL`**, they questioned whether the framework had incorrectly constructed its own reference to the parent **`Activity`**. In fact, **`getActivity`** was called in an event that occurred before the framework had attached the **`Activity`** to the **`Fragment`**.

Takeaway: The inversion of control in frameworks increases the difficulty of understanding the application's control flow, which increases the difficulty of debugging the application.

Object Protocols. Object protocols are object states that dictate how an object can be used. An example of object protocols in Android are lifecycles: state transitions between starting, active, and stopping for components. Participants experienced challenges with object protocols (e.g., accessing values before they were set). Object protocols are explained and diagrammed explicitly in the documentation, but implemented indirectly in the framework code, and invisible to non-framework code. This likely increases the difficulty of understanding the relevant object protocols.

Object protocol issues in tasks TA4 and TA5 significantly contributed to the amount of time those tasks took (see Section 3.3). Most participants assumed the application had performed an invalid action, rather than that an action that was invalid only in a given state. Object protocol misunderstandings also led participants to incorrectly conclude that certain values were available to use in the application. Three participants (PA4, PA6, PA11) mistakenly wrote code to access variables storing user selections before the user could have selected them. Participants were then confused when the accessed values did not match those they selected in testing. Participants (PA6, PA10) were confused about the circumstances in which they needed to commit and finalize a **`Fragment`** transaction (as opposed to the cases in which transactions were automatically committed). PA1 expressed exasperation on the difficulty of keeping track of when methods were called, stating *"You look at the Fragments documentation and it's like lifecycles inside of lifecycles inside of other lifecycles. It's annoying. It's like, how do I keep track of all this?"* In the ROS study, participant PR26 mentioned uncertainty about how to modify a method because of the states the application could be in when that method was called.

Takeaway: The way frameworks hide object protocols causes problems when diagnosing the problem location and when producing the correct fix.

3.2.1 Static Challenges

The static structure of frameworks, such as declarative artifacts and the mapping between source files and executable components, presented multiple challenges to participants. Participants commonly struggled to understand the separation between static structure and dynamic changes, determine the effects of the application's static configuration, and use that knowledge to solve problems. This led to uncertainty about whether errors should be addressed by modifying static files, or via a dynamic solution. Participants also often struggled with determining the correct framework terminology for their situation, which increased the difficulty of finding related documentation and online solutions.

In ROS, participants had difficulty understanding how source files map to executable components, partially because ROS executables consist of various components (Nodes, Services, and Topics) that do not map directly to source. ROS also does not provide an easy or well-known way to find source code corresponding to a given component. Participant PR16 struggled to understand how the publisher and subscriber methods in the C++ files integrated with the data redirections in the launch file. Other participants (PR17, PR18, PR20, PR22, PR25, PR27) similarly struggled to understand how the application remapped data between components. In one case, Participant PR17 diagnosed the problem but struggled to find the appropriate source file, due to both the organization of the ROS application files and their confusion about how to use the ROS filesystem commandline tools: *"I am looking for source code for [this node]... Unfortunately ROS is trying to isolate me from the file system, which I dislike, because it cannot isolate me fully."* About 16 minutes into the task, Participant PR17 exclaimed *"This is ridiculous, I can't even find the code that I am supposed to be debugging!"* They eventually used `grep` to find a node, more than half an hour into the task.

Multiple participants were confused about which framework concept applied to the current task. For example, both TA3 and TA6 asked participants to add an **`OptionsMenu`** to the application. Participants were given an application were the **`OptionsMenu`** did not appear and a picture that showed the application's **`OptionsMenu`**. Participants were asked to fix the problem in the application so that the **`OptionsMenu`** appeared. Android's **`OptionsMenu`** appears on the **`ActionBar`** but is not managed by the **`ActionBar`** class directly. At the beginning of the task PA9 commented *"I believe that is probably related to the ActionBar, because I think that is where the user defined menus can go."* PA9 ended up performing three different searches related to icons in the **`ActionBar`** which all initially looked promising to the participant but were ultimately unsuccessful. The participant's fourth search, *linearlayout action bar*, found a site that directed the participant to add the options through the **`OptionsMenu`** and lead to a successful completion of the task. PA1, PA2, PA3, PA9 all demonstrated difficulty determining the correct framework terms and concepts related to the issue.

Participants in the study were often unsure if changes to the application were performed dynamically with method calls or statically with adjustments to options in a *declarative artifact*, such as the XML layout specifications in Android. In the same tasks (TA3 and TA6), many participants (PA9, PA13, PA14, PA15) were initially confused if the **`OptionsMenu`** was added through a declarative artifact or through a method call. These participants looked through the

Android layout editor for an `OptionsMenu` or tried to add an `OptionsMenu` to a XML file, before realizing that it must be added dynamically. Another participant (PA9) investigated the `strings.xml` file after an online answer suggested that the problem may lie in an undefined icon title. Participant PA10 remembered that a specific theme setting could cause errors and checked if the theme caused the error.

Takeaway: Unclear mappings between declarative artifacts and the application's source code provides a source of debugging difficulty.

3.2.2 Historical Challenges

The fact that frameworks change over time can increase the difficulty of debugging framework errors: both participants' knowledge and online help or solutions may be out of date.

Legacy Challenges. Previous versions of both Android and ROS created issues for several participants. In Android, one participant (PA9) questioned whether a feature should be implemented in a backwards-compatible way, later discovering that the application was not configured to work with backwards-compatible components. A few participants (PA8, PA9, PA14) avoided online answers older than two years because they assumed they would no longer apply. Other participants (PA1, PA15) mentioned they were familiar with Android a couple years ago, but there had been many changes to the framework since they were proficient with it.

Some ROS participants incorrectly diagnosed the obsolete message type in task TR2 as correct because they had used it previously. Participant PR21 recognized that the message type caused an issue, searched the message type online, and found its official documentation, not realizing that the documentation was for an older ROS version. This was a problem because the documentation indicated that the file was using the message type correctly. The participant investigated four other possible error sources before realizing that a different message type was needed.

Past Experience. While past experience was often helpful, one participant in the Android study (PA10) misdiagnosed an error message due to previous experience. This caused the participant to conclude to *"not trust your experience."*

Takeaway: Out-of-date knowledge about a framework and similar error messages that cause developers to incorrectly diagnose the problem increase the difficulty of debugging framework applications.

3.2.3 Dynamic Benefits

Throughout the study, participants commonly used the framework to perform actions that would have been much more difficult to recreate without the help of the framework. When faced with a task, almost all participants tried to implement the correct, prescribed framework method for performing the required action (although PA1, PA3, PA5, PA8, and PA11 implemented custom solutions for certain tasks, such as implementing a custom message passing solution in TA1). For example, PA6 in TA1 correctly used the **`FindFragmentById`** method to access user input, instead of writing code to pipe the user input through the application. Overall, frameworks can provide developers with a prescribed way to address problems in their plugins.

Takeaway: Frameworks provide methods to address common task in a framework application. These methods can decrease the difficulty of implementing a bug fix.

3.2.4 Static Benefits

Study participants found the static organization of the framework helpful when trying to gain an overview of the application, which helped them find files of interest more easily than through unstructured search. In ROS, participants used the launch files as a way to start exploring the application. For example, participant PR27 looked through the ROS launch files to understand which nodes were involved in the application. Participant PR26 mentioned that they liked to use launch files to get an overview of the application. Multiple participants (PR17, PR18, PR19, PR22, PR26, PR27) used the launch files as a table of contents, using them to determine application components, how they interact, and locate source files of components.

Similarly, in Android, participants (PA1, PA2, PA3, PA6, PA8, PA9, PA10, PA11, PA13, PA14, PA15) used the structure of Android application to quickly find resource files and test case files. For example, PA8 was able to quickly look up the correct options menu layout file when writing the required options menu code. None of the tasks required participants to resolve bugs in declarative artifacts, so it's possible that this type of problem would be especially easy to debug.

Takeaway: The common structure of frameworks applications reduces the time to find necessary information when debugging.

3.2.5 Historical Benefits

Participants often found that past experience was helpful; some were able to correctly diagnose a ROS error simply by looking at the failing section of code and relating it to code or problems they had seen before. Multiple ROS participants (PR17, PR21, PR26, PR27, PR28) were able to diagnose an error and suggest a working alternative based on past experience. While working on task TR2, participant PR28 noticed the error in the code and said, *"I think the fact that there's a beginning slash means that instead of looking under this node's namespace it's gonna look under the global namespace [where] this parameter doesn't exist."* The participant was correct. Detailed knowledge of a framework, built up by through experience, can help mitigate barriers frameworks impose. Other participants (PR18, PR22, PR26, PR27) stated that past experience shaped their general ROS debugging strategy. One participant remembered to set framework environment variables, attributing past environment problems. Another participant (PR26) always used `grep` to find calls to a function modified over the course of a debugging session, to guard against unforeseen side effects, which was a problem they had faced in the past.

Takeaway: The common elements of applications created in a framework allows developers to build debugging experience in the framework.

Task	Times for Successful Completion (min)	Times for Unsuccessful Completion (min)
TA1	20, 41, 70, 89	35
TA2	4, 18, 20, 23, 24, 26	
TA3	8, 32, 42	33
TA4	17, 24	25, 46, 92
TA5	25	40, 64, 88
TA6	5, 27, 49	
TA7	8, 11, 13, 41	25
TR1	16, 22, 30, 42, 68	58, 62, 93
TR2	14, 34, 36, 43, 48	8, 45, 60
TR3	13, 15, 18, 25, 39, 40	

Table 3.3: Time spent on each task in minutes, grouped by successful and unsuccessful attempts.

3.3 Difficulty By Consequence

I analyzed participant results from both the Android and ROS tasks using the consequence categories collected from the Android investigation. I validated these categories by creating ROS trials and found that all of the ROS directive violation consequences fit into previously created categories. While I did not perform enough trials of each task for statistical significance, the results from the trials provide insight on the difficulties between the tasks, which is useful for directing research towards the tough framework problems. I present the time participants spent on each task and whether they successfully finished the task in Table 3.3. The mean time and success rate, grouped by consequence category, is shown in Table 3.4. The time participants spent on each violation category is presented as a number not to represent a precisely measured time for the task, but to present a better sense of the data than descriptive terms (i.e., much longer, longer, about the same, shorter, much shorter). Figure 3.3 show the time participants spent on tasks in the study, along with their success or failure rate. Figure 3.4 shows a box-and-whisker plot to provide a visual comparison of the average time spent on each task. There was a substantial difference in the mean time to complete tasks (ranging from 19 to 51 minutes) and the success rate on tasks (ranging from 33% to 100%) of difference consequences.

The consequence of violating a directive appears to influence how long it takes to debug the error as well as how likely a developer is to succeed in doing so over a short debugging session. A surprising result compared to prior research on debugging is that participants had significant difficulty with finding a fix once they were aware of the problem location. As an example of prior research in this area, Vessey [90] found that fault localization often took longer than finding the fix. However, developers who worked on the tasks in *Android: Crash With Reference To Directive* category were notified of the fault location immediately, yet this category had one of the longest mean times. One hypothesis that explains this phenomenon is that framework developers made error messages for these directives because the crash was extremely difficult to debug without referencing the directives. Further investigating this question, such as through a code archeology study, might provide further insight, but it was not in the scope of this work. However, even for other tasks that participants were immediately notified of the fault location,

Violation Consequence	Time (Mean)	Sessions Completed	Sessions Attempted	Success Rate (%)	Tasks
1. Android: Wrong Value Returned	51 min	4	5	80	TA1
2. Android: Crash With Reference To Directive	47 min	3	9	33	TA4, TA5
3. Android: Missing Feature	28 min	4	8	50	TA3, TA6
4. Android: Tool Warning	23 min	6	6	100	TA2
5. Android: Crash Without Reference To Directive	19 min	4	5	80	TA7
6. ROS: Missing Feature	49 min	5	8	63	TR1
7. ROS: Wrong Value Returned	36 min	5	8	63	TR2
8. ROS: Compiler Error	25 min	6	6	100	TR3

Table 3.4: Mean time on task and task completion rate, by consequence. Time on task includes failed attempts.

participants did not immediately determine the correct fix, often taking 20 or more minutes (TA2, TA4, TR3). In these cases, participants knew that certain directives were violated but they did not know how to fix the error. Participants found this frustrating, with one participant stating *"Why don't they [the documentation] tell me the right thing to use? They tell me it is going to cause a problem but they don't tell me what the alternative is."*

Takeaway: I observed that it appears important not only to notify developers of directive violations but also to help fix directive violations explicitly.

3.4 Limitations

External Limitations. I attempted to mitigate the risk that the results will fail to generalize to other frameworks or languages by investigating two different frameworks and a wide range of framework debugging problems. I have probably not discovered all of the benefits and challenges of framework debugging (or Android and ROS debugging), but I have reduced this risk by first investigating the space of error presentations that developers face (the consequence categories) and then investigating those categories in-depth with the human trials. The categorization of framework directives by violation consequence may not generalize (e.g., other frameworks may not have recommended development tools), and it may be incomplete; in particular, I did not consider potential non-functional violation effects, such as degraded performance. The lack of completeness could motivate future work.

The constructed tasks may not represent solving real-world debugging issues. This concern was reduced by basing the Android tasks on StackOverflow questions. Additionally, participants were new to the code in each task, possibly leading to unrealistic code familiarity problems. I sought to reduce this threat by providing Android participants with a learning period, but I note that, for example, one participant mentioned that it would be preferable to spend a day reading

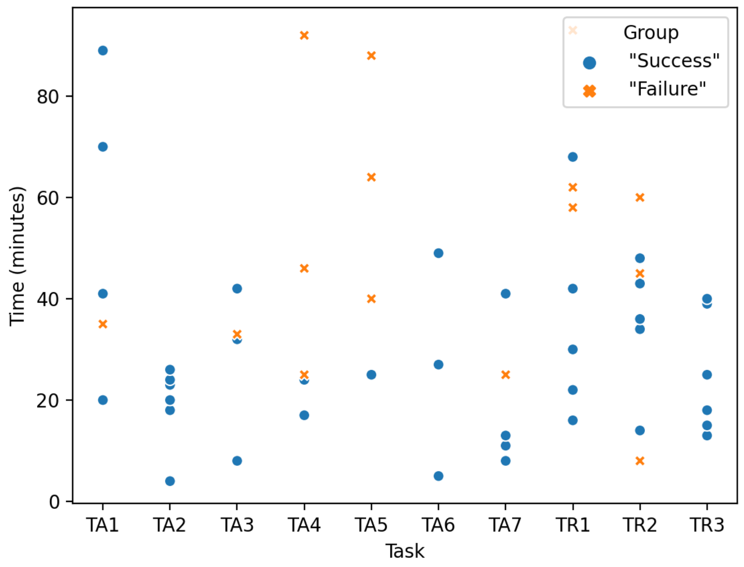


Figure 3.3: The time participants spent on tasks, with marked successes and failures.

documentation before tackling the tasks. As such, time limitations may have influenced the results. Finally, the participants in the study may not represent the population of framework users, and instead might be biased by the large number of student participants. I attempted to address this limitation by recruiting participants with framework experience: 14 of the Android study participants had over a year of industrial Android or Java experience and 7 of the ROS participants had over a year of ROS experience. It is likely that the study's results represent the challenges of developers that are moderately skilled in the framework, and may not apply to multi-year experts. The sample sizes of both number of trials in each category and number of tasks in each category are too low of a sample size to draw conclusions with statistical significance. While important in certain cases, my goal was to perform a qualitative-exploratory study, where statistical significance was not a primary concern. My goal was to determine what problems developers have instead of how often these problems occur. Achieving statistical significance with number of tasks or participants is left to future work.

Internal Limitations. Participants could freely decide, in a low-risk situation, when to quit a task. Participants were also asked to think-aloud, and prompted to do so by the researcher. These prompts may have altered the approach a participant would have taken absent the prompt. The think-aloud component may affect how long participants took to solve the tasks. I believe that the think-aloud component affected tasks roughly equally, such that tasks which took significantly longer than the others are likely to have taken longer in a non-think-aloud context. While the size of the different problems were restricted to directives, the amount of code required to fix the directive was not consistent across all tasks. However, all tasks could be fixed with ten additional lines or less. Finally, some participants mentioned they would have been more comfortable if the researcher were not watching, and if they were able to use their preferred IDE, OS, or laptop. These irritants may have caused participants to perform differently than they would have in their

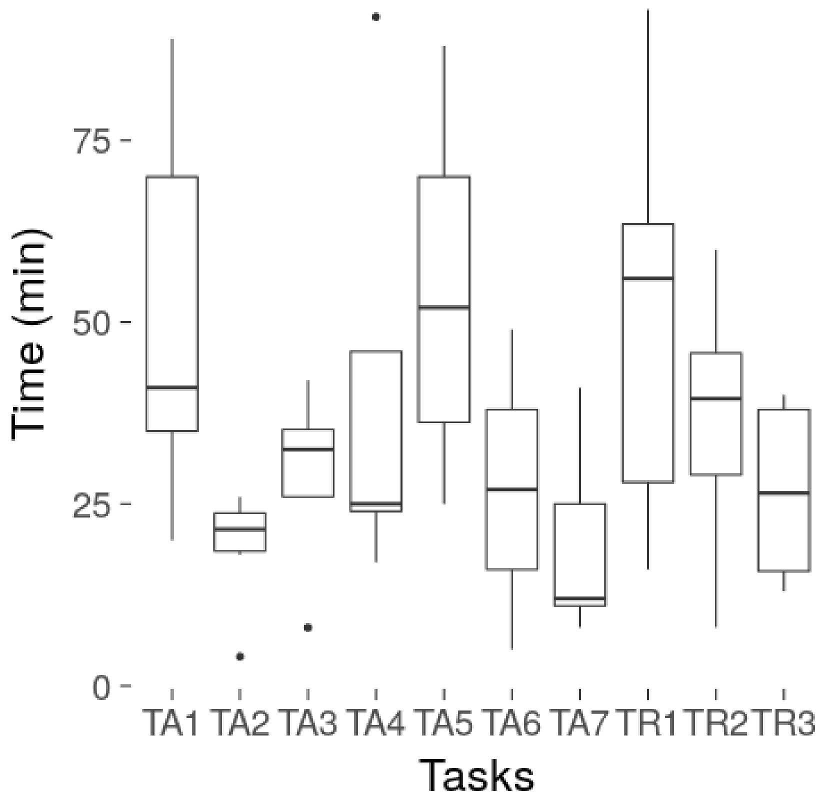


Figure 3.4: Average time spent per task. Times includes failed attempts. There are not enough points to demonstrate statistical significance, but the graphs are provided a way to quick compare average time trends.

preferred environment.

3.5 Summary

In the thesis statement (Section 1.2), the first goal was to understand the challenges of framework development. To investigate these challenges, I performed an exploratory study to better understand the challenges that developers face when debugging due to the unique aspects of framework application development. I found that developers encounter challenges when debugging plugins but also benefited from the methods and structure provided by the framework. The way that frameworks separate the execution flow, data flow, and definitions reduces the locality of the problem, requiring developers to understand multiple components of the framework application to be able to fix the problem. Developers seemed to have extra difficulty fixing state related issues, where the framework and plugin both expect changes that are non locally clear to the developer. Developers also would benefit from guidance on how to solve problems in a plugin, because developers occasionally find the way the framework expects the plugin developer to fix an issue to be not obvious. These results motivate an automated repair approach to fix plugin problems. I explore automatically identifying these problems in Section 4 and the full automated repair technique in Section 5.

Chapter 4

Automated Identification of State-Based Directive Violations

The insight from the debugging study lead to the conclusion that plugin developers would benefit from a tool to assist the framework application debugging process, specifically when debugging state-based directives. Based on this insight, I created an automated repair technique for plugin problems. The first step in creating an automated technique to assist plugin developers is to identify when a plugin contains a problem. To identify problems in plugins, I created a specification language for specifying how to detect state-based directive violations, cases where an object or other entity in the framework was in an invalid state for a particular method call. FrameFix then automatically converts specifications written in this language into static analysis checks. FrameFix or plugin developers can use this check to determine if the application contains a problem. I evaluated the expressivity this language using a taxonomy of object protocols to demonstrate that the language covers a wide variety of state-based issues. I then implemented the checks written in the specification language on the F-Droid dataset. I found that a test set of directive specifications applied to over 80% of real applications and that the specifications were able to detect multiple problems in real applications.[1]

4.1 Specification Language Approach

A specification language for testable, state-based directives needs to be able to state the specification goal in a checkable form. For example, to check the directive **`setPackage`** *and* **`setSelector`** *cannot be called on the same* **`Intent`** *instance*, the specification would need to be able to state that for all **`Intent`** instances, either **`setPackage`** or **`setSelector`** can be called but not both. However, directives written in natural language are often imprecise or use terms that do not clearly map to something that can be easily checked in static analysis. For example, in the directive **`getResources`** *cannot be called in a background* **`Fragment`**, a

[1]At the time of writing, the content of this chapter is currently under submission for peer review with the title *FrameFix: Automatically Repairing Statically-Detected Directive Violations in Framework Applications* [22]. Code associated with the project can be found at https://www.dropbox.com/sh/9b7ayp25oa248dn/AAB88KclAKjWxp3gft2pYgE3a?dl=0.

background `Fragment` is a `Fragment` that the plugin is not showing to the user. This concept is difficult to map directly to a static analysis check. There are multiple ways to address this issue, but I chose to require the specification to be written in a checkable form that maps to code to simplify the implementation. Thus there is some specification burden associated with translating a state-based directive into the specification language format. Due to the difficulty of translating the specification into a checkable form, a person knowledgeable in the framework (likely the framework developer) is needed to translate the directives into the specification language. While this translation may require expert knowledge, each directives only needs to be specified once, and then can be checked for all framework plugins or client applications.

Once the specification has been written in the specification language, the tool translates the specification into a set of functions that are composed and evaluated to determine if the application violates the specification. This approach allows the specification language to check a wide variety of state-based properties.

Figure 4.1 provides the grammar for the directive specification language, which includes both assertions (which define checks), and control statements, to logically compose assertions. Assertions can either be a *context* assertion or *simple* assertion.

A *context* assertion limits the assertion to a part of the code base. For example, to check that `Fragment` classes contain a definition for the `onCreateOptionsMenu` method, the context assertion would be written as *checkSubclassesOf(“*`Fragment`*”).defines(“* `onCreateOptionsMenu`*”)*. Some other examples of *context*s are nested classes and method definitions. Since multiple *context*s are often required to define the section of code to check (e.g., a method definition in subclasses of a certain Class), *context*s can be nested with the ‘.’ operator.

Simple assertions do not require a *context*, but can be modified by a *context*. For example, a *simple* assertion could check if the application defines a necessary configuration file. In contrast, a *simple-context-assertion* requires a context, thus it can only be part of a *context* assertion.

While a *context* is designed to encode a location to check in the plugin, a *state* is a list of methods where the desired object state is known to be true. The state is then checked using the methods in the stack trace (i.e. if the static path in the call-graph from the current method to the start of execution contains this method call, then the application is known to be in this state).

A *method-call*, which specifies the method call to look for in the code — when checking if `setHasOptionsMenu` is called in `onCreate`, `setHasOptionsMenu` is the *method-call*, while `onCreate` is the *context*. *method-call* has an optional *parameter-list*, which adds parameter requirements to the call. For example, if the checker is only interested in cases of `setHasOptionsMenu` when called with the parameter `true`, `true` would be defined as a required parameter. Required parameters can either be constant values, which only match if the constants are equal, or identifiers, which only match if the same object reference is used across multiple method calls (e.g., an object must unregister all receivers that it previously registered). If a parameter does not need to be matched, a wild card ‘*’ is used.

Two of the *simple-context-assertions*, *firstCannotFollowSecond* and *secondCannotOccurBeforeFirst*, are similar but differ on which parameter triggers the check — which means that the ordering constraint is not triggered if only the non-trigger method call is present. In the *firstCannotFollowSecond* check, once the first method call parameter is found, the analysis checks if the second method parameter preceded the first method call parameter in the context. If this condition occurs, then the check documents an error. Alternatively, the *secondCannotOccurBe-*

```
⟨expression⟩ ::= ⟨control-expression⟩ | ⟨assertion-expression⟩

⟨control-expression⟩ ::= and(⟨expression⟩, ⟨expression⟩)
   |  or(⟨expression⟩, ⟨expression⟩)
   |  if(⟨expression⟩) then (⟨expression⟩)
   |  not(⟨expression⟩)
   |  ⟨created-control-expression⟩

⟨assertion-expression⟩ ::= ⟨simple-assertion⟩ | ⟨context-assertion⟩

⟨simple-assertion⟩ ::= isDefined(⟨item⟩)
   |  ⟨created-simple-assertion⟩

⟨state⟩ ::= list(⟨methods-where-true⟩)
   |  ⟨created-state-representation⟩

⟨context-assertion⟩ ::= ⟨context⟩.⟨assertion-for-context⟩

⟨context⟩ ::= methodToCheck(⟨method-call⟩)
   |  instanceOf(⟨type⟩)
   |  checkSubclassesOf(⟨class⟩)
   |  checkClassesWithOuterClassThatAreSubclassOf(⟨class⟩)
   |  ⟨created-context⟩

⟨assertion-for-context⟩ ::= ⟨simple-assertion⟩
   |  ⟨simple-context-assertion⟩
   |  ⟨control-expression⟩
   |  ⟨context-assertion⟩

⟨simple-context-assertion⟩ ::= contains(⟨method-call⟩)
   |  in(⟨method-call⟩, ⟨state⟩)
   |  secondCannotOccurBeforeFirst(⟨method-call⟩, ⟨method-call⟩)
   |  firstCannotFollowSecond(⟨method-call⟩, ⟨method-call⟩)
   |  methodCallExclusiveOr(⟨method-call⟩, ⟨method-call⟩)
   |  ⟨created-assertion-for-context⟩

⟨item⟩ ::= ⟨file-type⟩
   |  ⟨method-call⟩

⟨method-call⟩ ::= ⟨method⟩
   |  ⟨method⟩(⟨parameter-list⟩)

⟨parameter-list⟩ ::= ⟨parameter⟩
   |  ⟨parameter⟩, ⟨parameter-list⟩

⟨parameter⟩ ::= ⟨wild-card⟩
   |  ⟨constant⟩
   |  ⟨identifier⟩
```

Figure 4.1: Grammar for creating the heuristic analyses of directives. Shown here are the analysis assertions and the possible control statements for combining the assertions.

foreFirst check is triggered when the second method parameter is found. The analysis checks the context to determine if the second method parameter was not preceded by the first method parameter, and throws an error if this occurs. These two *simple-context-assertions* could be combined into a three parameter check, such as an *enforceOrder* call, where the first two parameters are *method-call*s and the last parameter specifies the triggering *method-call* (which could be done through a new *triggering-parameter* rule). The two *simple-context-assertions* were not combined in the current version of the specification language to reduce the number of rules that needs to be defined for the language and in attempt to make the triggering method call obvious to a casual reader of the language.

Control-statements include standard first-order logic operators like `and`, `or`, and `not`. `if-then` is provided as convenient shorthand for compositions of assertions.

Since the specification language was not created with an exhaustive list of directives in mind, users of the language may want to extend the language to new static analysis checks. The specification language defines extension and customization points through the *created-** language terms, which could be defined by the user of the language.

4.2 Statically Identifying Directive Violations

To translate the specification to a static analysis check, the assertions are converted into composable functions, which are then combined using the control statements. The composable functions perform different checks depending on the check specified. Some check if a pattern exists in the code (for example, *contains* checks if a method call occurs in the expected location). Others checks for invalid method call ordering using data flow (e.g., *firstCannotFollowSecond*). The simple context assertion *in* checks if calls of a known state exist in all the paths in the call graph from the currently evaluated method to the start of execution. This combined check produces a final count of the number of violations of the directive in an application.

Example. Consider the directive:

"**`findViewById`** *cannot be called before* **`setContentView`** *has been called*"

This directive translated in the API specification language discussed in Section 4.1, would be

checkIfSubclassOf("`Activity`").methodToCheck("`onCreate`"). secondCannotOccurBeforeFirst("`setContentView`", "`findViewById`"))

This specification statement states to first check if the file contains a class that is a subclass of **`Activity`**. If the file is a subclass of **`Activity`**, then, in the **`onCreate`** method, require that **`setContentView`** is called before **`findViewById`**. However, only require the call order if a call to **`findViewById`** occurs in the method (meaning that **`findViewById`** is not required to follow **`setContentView`**). While the check to only investigate classes that are subclasses of **`Activity`** is not mentioned in the directive, this context assertion is included because the directive only applies to subclasses of the **`Activity`** class.

While not explicitly mentioned in the directive, the API specification restricts the check to

Category	Statement	Specification
Initialization	**`getEncoded`** cannot be called before **`init`** is called on an **`AlgorithmParameters`** instance	instanceOf("AlgorithmParameters"). secondCannotOccurBefore-First("init","getEncoded")
Deactivation	a closed **`BufferInputStream`** cannot be reopened	instanceOf("BufferInputStream"). firstCannotFollowSecond("open","close")
Type qualifier	a **`Collection.unmodifiableList`** instance cannot call the **`add`** method of its **`List`** superclass	instanceOf("Collection.unmodiableList"). not(contains("add"))
Dynamic preparation	For an **`Iterator`** instance, **`remove`** can only be called after **`next`**	instanceOf("Iterator"). firstMustOccurBe-foreSecond("next","remove")
Boundary	there a limited number of valid **`next`** calls to an **`Iterator`** instance	instanceOf("Iterator"). callLimit("next",7)
Redundant operation	an **`AbstractProcessor`** instance cannot call **`init`** twice	instanceOf("AbstractProcessor"). callLimit("init",1)
Domain mode	an **`ImageWriteParam`** instance can only call **`setCompressionType`** when the instance is in explicit compression mode	instanceOf("ImageWriteParam"). firstMustOccurBeforeSecond("setCompressionMode(MODE_-EXPLICIT)", "setCompressionType")

Table 4.1: API specifications for the examples of the different categories from the Beckman taxonomy.

the `onCreate` method because an `Activity` with a user interface should set the user interface by calling `setContentView` in `onCreate`. The checker will then check the execution paths of `onCreate` to make sure that `findViewById` is not called before `setContentView`.

4.3 Specification Evaluation

To evaluate if the API specification approach can apply to a diverse set of specification requirements, I evaluated the specification approach using the Beckman taxonomy of object protocols [13]. Beckman manually classified the different types of over 600 object protocols found in 4 programs or libraries. Beckman provided an example protocol for each category (except for the 'other' category, so I excluded it). I then checked if the example protocols could be encoded in the directive specification language.

The encoded API specifications for each of the Beckman taxonomy categories are shown in Table 4.1. Five of the seven examples translated easily to into the specification language. For the other examples, a new *simple-context-assertions* had to be defined to limit the number of method calls for an object. The Boundary category was the most difficult to translate into the directive specification language, because the example checked a dynamic property (limiting calls to the number of items in the `Iterator`). While not a perfect solution, if I assume that the number of valid calls can be determined statically (in this example, I use the value seven), then the specification language can handle this case. Incorporating dynamic checks to the language would address this issue, and is left to future work.

4.4 Directives Implemented in Tool

Another important evaluation of the check system is to show that the static analysis system can be used to check if real applications violate state-based directives. To evaluate the static analysis, I implemented checks for a set of directives taken from the Android `Activity`, `Fragment`, and `Intent` classes. The directives are listed here for reference.

---

1. Directive name: GetActivity

Directive text: `getActivity` *can only be called when the* `Fragment` *has been attached to an associated* `Activity`.

Explanation: A `Fragment` must be in an `Activity`-*attached* state, otherwise `getActivity` returns `null`

Result of violating the directive: `NullPointerException`

Directive specification:
checkSubclassOf("Fragment").in("getActivity", "activityAttachedState")

activityAttachedState = list("Fragment.onAttach", "Fragment.onCreate", "Fragment.onCreateView", "Fragment.onViewCreated", "Fragment.onActivityCreated", "Fragment.onStart", "Fragment.onResume", "Fragment.onPause", "Fragment.onStop", "Fragment.onDestroyView", "Fragment.onDestroy", "Fragment.onDetach", "Fragment.onActivityResult", "Fragment.onClick", "Fragment.onPreferenceClick")

---

2. Directive name: OptionsMenu

Directive text: *To use a* `Fragment`'s `OptionsMenu`, *the application must contain both the call* `setHasOptionsMenu(true)`, *and define* `onCreateOptionsMenu`

Explanation: If the `Fragment` is in the (static) *has*-`OptionsMenu` state, through the presence of a defined `onCreateOptionsMenu`, then the `Fragment` must call `setHasOptionsMenu` ; Alternatively, if the `Fragment` can request `OptionsMenu` (through a call to `setHasOptionsMenu(true)`), then it must start in the *has*-`OptionsMenu` state.

Result of violating the directive: The `OptionsMenu` will not appear.

Directive specification:
subClassFragment(Fragment).or(if(contains("onCreateOptionsMenu")) then (methodToCheck("onCreate").contains("setHasOptionsMenu(true)"), if(methodToCheck("onCreate").contains("setHasOptionsMenu(true)")) then (contains("onCreateOptionsMenu"))

---

3. Directive name: SetArguments

Directive text: `setArguments` *cannot be called until after the* `Fragment` *is instantiated.*

Explanation: Once a `Fragment` instance has passed out of its instantiation state, a call to `setArguments` is illegal

Result of violating the directive: `RuntimeException`

Directive specification:
or(checkSubclassOf("Fragment").in("setArguments", "fragmentInitializedState"),and(methodToCheck("onClick").contains("setArguments"), and (methodToCheck("onTabSelected"). contains("add")), methodToCheck("onTabUnselected").contains("hide"))

fragmentInitializedState = list("Fragment.onAttach", "Fragment.onCreate", "Fragment.onCreateView", "Fragment.onViewCreated", "Fragment.onActivityCreated", "Fragment.onStart", "Fragment.onResume", "Fragment.onPause", "Fragment.onStop", "Fragment.onDestroyView", "Fragment.onDestroy", Fragment.onDetach", "Fragment.onActivityResult", "Fragment.onClick")

---

4. Directive name: Inflate

Directive text: *The* `onCreateView` *method in a* `Fragment` *that will attach to the user interface, should return the result of a call to* `inflate` *where the last parameter is* `false`.

Explanation: If the `Fragment` will become part of the `Activity` user interface (a static state designation) then the fragment's `View` must be initialized correctly

Result of violating the directive: Crash due to an internal framework error.

Directive specification:
checkSubclassOf("Fragment").not(checkSubclassOf("DialogFragment")) .methodToCheck("onCreateView").contains("inflate(,*,false)")*

---

5. Directive name: FindViewById

Directive text: `setContentView` *must be called before* `findViewById`.

Explanation: Android does not associate an ID with a `View` until the `View` is in the initialized state.

Result of violating the directive: `RuntimeException`

Directive specification:
checkSubclassOf("Activity").methodToCheck("onCreate").
firstMustOccurBeforeSecond("setContentView", "findViewById")

---

6. Directive name: GetResources

Directive text: `getResources` *cannot be called in a background* `Fragment`.

Explanation: `getResources` can only be called on a `Fragment` when it is attached to an `Activity`, a state that a background `Fragment` is not in by definition.

Result of violating the directive: `RuntimeException`

Directive specification:
checkSubclassOf("AsyncTask").checkClassesWithOuterClassThatAreSubclassOf("Fragment").
not(contains("getResources"))

---

7. Directive name: SetInitialSavedState

Directive text: `setInitialSavedState` *cannot be called after the* `Fragment` *is attached to an* `Activity`.

Explanation: A `Fragment` cannot be initialized once it is in the *attached-to-*`Activity` state.

Result of violating the directive: `RuntimeException`

Directive specification:

subClass("Fragment").not(in("setInitialSavedState", "fragmentInitializedState"))

fragmentInitializedState = list("Fragment.onAttach", "Fragment.onCreate", "Fragment.onCreateView", "Fragment.onViewCreated", "Fragment.onActivityCreated", "Fragment.onStart", "Fragment.onResume", "Fragment.onPause", "Fragment.onStop", "Fragment.onDestroyView", "Fragment.onDestroy", Fragment.onDetach", "Fragment.onActivityResult", "Fragment.onClick")

---

8. Directive name: SetTheme

Directive text: `setTheme` *cannot be called after* `setContentView`.

Explanation: It is an error for an `Activity` to set an application theme after it has entered the `View`-*initialized* state

Result of violating the directive: `RuntimeException`

Directive specification:
checkSubclassOf("Activity").methodToCheck("onCreate").
firstCannotFollowSecond("setContentView", "setTheme")

---

9. Directive name: SetPackageSetSelector

Directive text: `setPackage` *and* `setSelector` *cannot be called on the same* `Intent` *instance.*

Explanation: `Intent` instance is already in the *package-set* or *selector-set* state, it cannot enter the other state.

Result of violating the directive: `RuntimeException`

Directive specification:
instanceOf("Intent").methodCallExclusiveOr("setPackage", "setSelector")

Directive violation	# of apps check applies	Percentage (%) of total	# with error	Error % of apps that apply
GetActivity	950	53.9	458	48.2
OptionsMenu	194	10.5	44	22.7
SetArguments	605	32.6	3	0.5
Inflate	435	23.4	13	3.0
FindViewById	368	19.8	0	0.0
GetResources	21	1.1	7	33.3
SetInitialSavedState	60	3.2	0	0.0
SetTheme	368	19.8	0	0.0
SetPackageSetSelector	554	29.8	0	0.0
At least one directive applies	1572	84.7	-	-

Table 4.2: This table shows the number of applications in the F-Droid dataset where the directive checks apply. It also shows the number of applications where the checks signaled a possible error. The second and third column are the count and percentage of applications in the dataset that applied to each directive violation specification. At least one check means the number of applications where at least one directive could apply to the application — applications contain the method call mentioned in the directive. The fourth and fifth column are the number of violations detected in the F-Droid dataset and the percentage of detected errors out of the number of applications where the check applies. - means that this value was not calculated. The directives selected in Section 4.4 occur in a large number of Android applications (about 85%) and are important for development. Since the static analysis is neither sound nor complete, and certain analyses have a high false positive rate, the number of detected errors may not accurately represent the number of directive violations in the application.

4.5 Applicability of Directives Covered

To investigate if the chosen directives apply to a large percentage of Android applications, I built an implementation of the specification language on top of FlowDroid, an Android static analysis tool [9]. I then collected 1,964 applications from F-Droid[2], a catalog of open source Android applications. FlowDroid threw an error when analyzing 108 of the applications, so those were removed from the dataset, leaving 1,856 applications to analyze. I downloaded the applications and determined if the applications used the methods mentioned in the directives discussed in Section 4.4. Table 4.2 shows how many applications use each directive in the dataset. These percentages are calculated out of the number of applications in the dataset for which FlowDroid was able to produce a call graph (1,865). The main takeaway from this table is that at least one directive applies to 84.7% of the applications in the dataset, demonstrating that the directives

[2]https://f-droid.org/en/ (dataset: https://doi.org/10.5281/zenodo.3698376)

covered in this investigation apply to wide range of applications.

4.5.1 Error Detection on F-Droid

With the goal of evaluating how well the checkers apply to a diverse set of applications, I ran the checkers on the applications in the F-Droid dataset. I found that 138 applications in the dataset produced call graph errors in Soot and were unable to be evaluated by the analysis. I then found that 65 of the applications produced other errors, often an analysis failure due to the application's language — the current implementation of FrameFix assumes that the application is written in Java, while some Android applications in the dataset were written in Kotlin. After removing the applications with errors from the evaluation, I ran the checkers over the remaining 1,761 applications in the dataset.

Only five checkers found errors in the applications. The errors counts detected by the five checkers are shown in Table 4.2. The high number of **GetActivity** errors is due to the fact that the checker heuristically checks if the method is used in the right context, and throws an error if **GetActivity** is used in any context that can not easily be determined to be correct.

4.6 Limitations

There are a few limitations to the current approach. The goal of the specification language is to be sufficiently expressive to support a study of automatic repair of directive violations. It does not include all possibly useful features in specifying all conceivable state-based directives. For example, I assume the number of arguments to a method are fixed; it also does not support explicitly stating alternative contexts created by program branching in the specification, but program branching can be checked in the underlying data flow analysis. The specification language supports extension and customization through the *created-** language terms. Another limitation is that it might be difficult for the average framework application developer to translate a directive into a check in the specification language. This difficulty occurs because natural-language directives are not always easily translated into a precise definition. My approach addresses a part of this limitation by allowing multiple developers to reuse the same specification, meaning that the specification only has to be written once by someone knowledgeable with the framework. The other part of this limitation is that it is easy to specify directive checks in a heuristic way that leads to neither sound nor complete results. This limitation was not addressed in this work and may be addressed in the future.

4.7 Inspiration for Specification Language

The specification language was inspired by fluent interfaces, such as Truth [39], and other framework constraint languages (SCL [45], the Fusion constraint language [21], and design patterns [32]). Fluent interfaces are an approach to specifying program steps, such as filtering a dataset, using method chaining, nested functions, and object scoping [35]. Fluent interfaces are commonly implemented in a way such that a chain of method calls perform modifications on an object. By contrast, my specification language instead uses the chain of method calls to add context restrictions to the final analysis check in the chain. These preceding method calls either add context restrictions (which are converted to `if` statements that limit the location of the static analysis checks) or describe how to combine independent code properties to check into a single detected violation. One similar fluent interface is Truth. Truth is a library for writing fluent Android assertions [39], which dynamically checks object values instead of the static check performed by my specification language.

This approach was also inspired by other framework constraint specification languages. I chose not to use them directly primarily because of the overhead associated with specifying a directive violation in those constraint languages. To illustrate the differences between my constraint language and other constraint languages that inspired my constraint language, I will use the directive "`setContentView` *must be called before* `findViewById`" as an example, In my specification language, this would be written as follows.

```
checkIfSubclassOf("Activity").methodToCheck("onCreate").
    firstMustOccurBeforeSecond("setContentView","findViewById")
```

This specification says to check that `findViewById` does not occur unless `setContentView` occurs first in the `onCreate` method of an `Activity` class. In this case, both the **checkIfSubclassOf** and the **methodToCheck** specification are context specifications, which limit the analysis to methods of that name in classes of that type. The `onCreate` method specification is added even though it was not mentioned in the directive because the `setContentView` method occurs in the `onCreate` method of an Android application.

One source of inspiration was the Structural Constraint Language (SCL), where program constraints are written as functions [45]. In Structural Constraint Language, the preceding directive is specified as follows.

```
def Objects as class(``Activity'')

for p: packages, c:classes(p), m: methods(c), ce: exprs(m)
(
  //if a defined onCreate method contains the call findViewbyID
  (name(m)==``onCreate'' & method(ce)=``findViewById'')
  =>
  //there exists a call to setContentView in the method
  (exists e: exprs(m)
    (method(e) =``setContentView'' &
    //and it occurs before findViewById
    (exists cde: cd(e)
      (method(cde)=``findViewById''))
    )
  )
)
```

This approach specifies to directly iterate over the classes and methods in a project, while my fluent style does this implicitly. My approach also reduces some of the required & operators through call chaining.

Another source of inspiration was the Fusion constraint language [21], which is designed to allow specifying constraints between multiple objects. Fusion is written as method annotations, which means that the annotation would need to occur before each overridden `onCreate` method. An example of specifying this constraint in Fusion is as follows, using the user-defined methods `precededBy(m)`, which defines that method m occurs before the current method call, and `FoundView`, which returns the correct instance of the view.

```
@Constraint(
  op="findViewById",
  trigger="{}",
  requires="precededBy(setContentView)",
  effect={"FoundView(v)"})
@Override
protected void onCreate(Bundle savedInstanceState){
```

This specification approach puts an annotation burden on developers, requiring developers to place the check in the correct spot. This approach, when compared to my approach, makes it easier for the developer to verify where the constraint is being checked. However, there may be cases where the same constraint applies to multiple method calls (such as overridden methods), where the developer would have to copy the constraint to multiple locations, increasing the chance that the developer mistakenly forgets to add the constraint to a method where it applies.

A third source of inspiration was design patterns [32]. This approach specifies constraints as an XML specified pattern. This approach adds overhead with the XML end tags and by listing methods in order, it is harder to specify that the first must precede the second method only if the second method exists.

While these three similar approaches can be used to specify directives of interest, my approach has a more minimal specification requirement, which I have shown with two examples.

4.8 Summary

In this chapter I presented an API-based specification language to specify static analysis checks of framework state constraints. While this section does not directly address any part of the thesis statement, the static analysis check is an important subcomponent of the automated repair process mentioned in the thesis statement. The static analysis is important because the automated repair technique needs to be able to identify the approximate fault location in plugins, to guide the repair location of generated possible fixes. The repair technique also needs to be able to determine when the problem has been removed from the plugin. For the static analysis to be useful for automated repair, I need an analysis technique that I can identify the plugin repair problems of interest in real applications. Thus, I evaluated the specification language on a taxonomy of object protocols and found to be able to support seven of the eight categories. I also evaluated the specification language and checker on a set of directives. I found that five of the nine static analysis checks were able to detect problems in real applications. This static analysis technique meets the requirements to be used as the fault localization component of the automated repair technique, discussed in the next chapter.

Chapter 5

FrameFix

5.1 Introduction

The debugging study (Section 3) showed that plugin developers would benefit from a system that recommended fixes to complement identifying the problem. An automated program repair technique for plugins is one way to recommend fixes to developers, and has been shown to help in the debugging process [87]. Automated program repair is a research area for generating possible solutions to known problems in a program. While a repair technique has been created to fix crashing Android applications, no technique has been created to fix the other problems that plugin developers encounter, such as the plugin silently missing a feature. To address this problem, I created FrameFix, an automated repair technique for framework plugins. FrameFix uses the static analysis checks discussed in Section 4 to locate the problem in a plugin and then determine if the problem is fixed. The unique approach of FrameFix is that FrameFix uses unique aspects of framework plugin development to guide its repair strategies. First, FrameFix tries to move problematic methods to the correct location in the plugin. If that approach does not fix the problem, FrameFix will then use similarly named methods off GitHub, an online source code repository tool, to guide proposed changes in the current application. While there are many important properties of an automated repair technique, such as usability, at minimum, an automated repair technique needs to be able to fix a diverse set of problems and be able to repair problems in real applications. Thus, FrameFix was evaluated on a set of manually built applications, problems detected in the F-Droid dataset, and problems randomly injected into F-Droid applications.[1]

[1]At the time of writing, the content of this chapter is currently under submission for peer review with the title *FrameFix: Automatically Repairing Statically-Detected Directive Violations in Framework Applications* [22]. Code associated with the project can be found at https://www.dropbox.com/sh/9b7ayp25oa248dn/AAB88KclAKjWxp3gft2pYgE3a?dl=0.

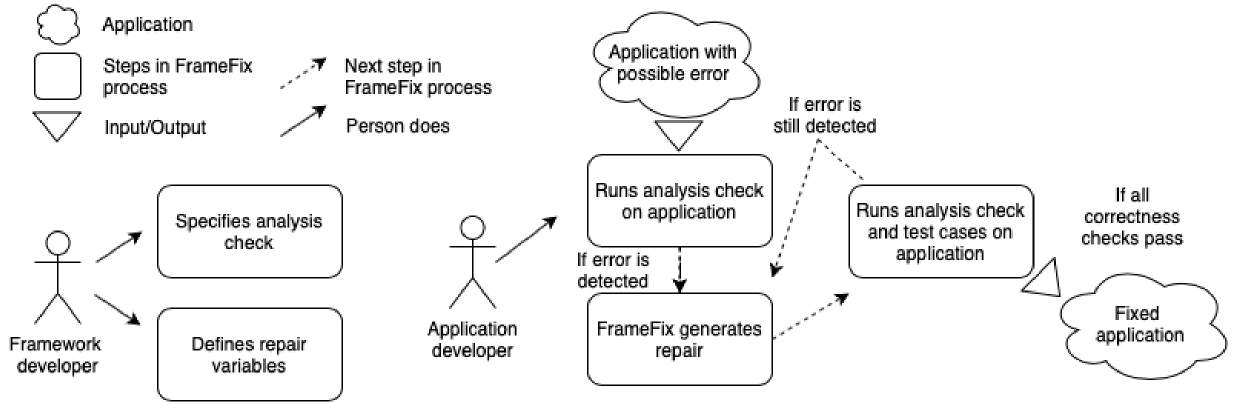

Application
Steps in FrameFix process
Input/Output
Next step in FrameFix process
Person does
Framework developer
Specifies analysis check
Defines repair variables
Application developer
Application with possible error
Runs analysis check on application
If error is detected
FrameFix generates repair
If error is still detected
Runs analysis check and test cases on application
If all correctness checks pass
Fixed application

Figure 5.1: The high-level FrameFix process. The framework developer creates the analysis check specification (Section 4.1) and defines the variables required to perform the repair for that problem. When a developer is developing an application using the framework, the developer can run the analysis check to see if the analysis check detects any problems in the application. If a problem is detected, FrameFix will attempt to repair the application by generation possible fixes (Section 5.3). If a fix fails to pass the checker and application test cases, FrameFix generates another possible fix. If FrameFix is able to generate a repair that passes both the analysis check and all the application's test cases, the application is considered to be fixed.

5.2 FrameFix Approach

FrameFix is an end-to-end approach for statically detecting and repairing certain classes of framework-specific errors. Figure 5.1 provides a high-level overview of the approach. A knowledgeable engineer (likely a developer of the framework) specifies directives relevant to correct usage of that framework, in a way that allows for statically detecting their violation (Section 4) and, ultimately, repairing them.[2] FrameFix requires the engineer to specify the static analysis check and a few other variables for the repair process, such as terms to guide the GitHub search. The cost of writing these specifications is highly amortized, and can be considered part of the process of documenting correct usage (which developers already do). Any application developer with this specification information can then use FrameFix to statically check for violations of the associated directives in client plugins (Section 4.2)

FrameFix attempts to construct repairs for detected violations (Section 5.3). The static violation report serves to localize the error. Once the error is localized, FrameFix leverages unique aspects of the framework application development to guide automated repair of the framework application. Specifically, FrameFix leverages the object protocols and inversion of control inherent in framework applications to repair framework application problems. Since frameworks often change object states in internal framework code, FrameFix can move method calls that have incorrect states to other parts of the application with different states, removing the state problem. Frameworks also implement inversion of control through callbacks, method calls with specific signatures, FrameFix can use the signatures to find applications that contain the same

[2]Inferring specifications from natural language documentation or source code analysis may be possible, but is left for future work.

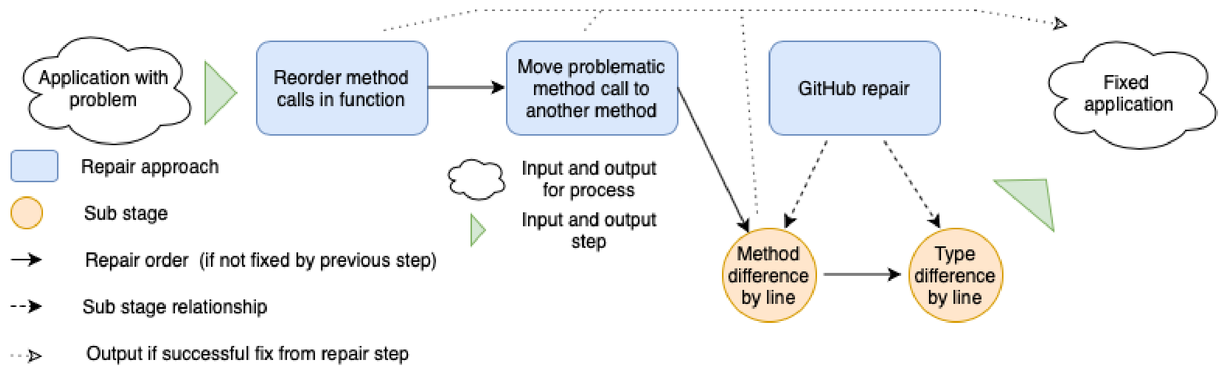

Application with problem
Reorder method calls in function
Move problematic method call to another method
GitHub repair
Fixed application
Repair approach
Sub stage
Repair order (if not fixed by previous step)
Sub stage relationship
Output if successful fix from repair step
Input and output for process
Input and output step
Method difference by line
Type difference by line

Figure 5.2: The steps and ordering of the FrameFix repair process.

call signature and use code from other applications which implemented the callbacks correctly.

FrameFix then validates generated repairs using the static analysis check, as well as by compiling the application and running any test cases associated with the application. Following a standard generate-and-validate repair paradigm [60], FrameFix attempts multiple repairs until either a time limit is exceeded, or one is found that satisfies all validation checks.

There are many properties of a successful automated repair technique.

5.3 Automatic Repair Technique

As shown in Figure 5.1, to initialize a repair, someone knowledgeable with the framework will need to define a few variables for the repair process. These variables include the method of interest and any possible second method of interest (the method or methods that were checked). The likely location of the methods of interest if one exists (such as the `onCreate` method). And search terms that can be used to narrow down the GitHub search. These variables should be defined when the check specification is written.

FrameFix tries three strategies to repair detected violations; Figure 5.2 provides an overview. FrameFix tries three high-level repair strategies. They are ordered by the likely worst-case time of each step — smallest to largest. The strategies in order:

1. **Method reordering.** Frameworks' heavy use of object protocols can lead to problems associated with calling methods while in the wrong state. One possible solution to this problem is to reorder the problematic method calls, to see if an incorrect method call order leads to one of the methods being called in the wrong state. FrameFix therefore starts by exploring rearrangements of the methods mentioned in the directive specification (Section 5.3.1).

2. **Method movement.** Directive violations can also be caused by calling methods when the application is not in the appropriate framework state. Thus, FrameFix next tries moving implicated method calls to alternative locations altogether, in an effort to move a necessary call to a location corresponding to an appropriate state (Section 5.3.2).

3. **Callback-informed search.** Inversion of control is typically implemented via methods with framework-specific type signatures and, as such, applications typically interact with

the framework through highly rigid API calls. This provides a useful filtering mechanism to constrain the search space of similar code suitably to inform repair. Thus, if the preceding attempts do not work, FrameFix searches GitHub for similar code to inform a candidate repair, based on the method signatures used in the code under repair (Section 5.3.3).

The goal of the repair technique is to fix the detected problem while also retaining the correct functionality of the application. Thus repairs are validated three different ways. The first validation step is the static analysis check. If the proposed fix passes the checker, then the problem has been removed. The second validation FrameFix performs is to compile the changed application, in attempt to verify that the proposed fix did not produce an invalid program. Unfortunately, it is still possible that the proposed repair removes the problem and removes other necessary parts of the application as well. For example, the technique could trivially remove the error by deleting the failing line and any other lines in the application that depend on the result of that line. To avoid this problematic result as a recommended solution, FrameFix's third validation step evaluates repairs on the available test cases for the application. FrameFix assumes that that the required functionality of the application is encoded in the test cases, and thus uses the test cases to ensure the proposed fix retains any required functionality. If none of the proposed repairs in a set amount of time produce a fix, then FrameFix was unsuccessful in fixing that error instance.

5.3.1 Method Order Repair

Since frameworks heavily use object protocols, in both internal framework code and in how objects are used in framework applications, an application bug may mean that a method call occurs in the wrong object state. For example, a call to get a resource view item in Android cannot occur before the view has been initialized — otherwise an exception will occur that can crash the application. In this case, the `Activity` is performing steps in the wrong order, but the calls would be valid if they were performed in the other order. FrameFix uses the insight that method order is important to generate repairs that reorder method calls in the current method definition. FrameFix only tries this repair approach if multiple methods are mentioned in the repair specification variables provided by the initial repair specifier. These multiple methods are intended to match the number of variables mentioned in the directive. FrameFix only tries to reorder the methods mentioned in the specification variables.

FrameFix generates three possible repairs for this strategy: 1) move the originally later method before the occurrence of the other method, 2) move the originally first method after the occurrence of the other method, 3) delete the occurrence of the later method. For example, assume that method **`foo`** should be called before method **`bar`**, but **`bar`** occurs on line 3 and **`foo`** occurs on line 6. Also assume that line 4 uses the result of **`bar`**, as shown in Figure 5.3a. In this strategy, the first proposed repair is to move **`foo`** before **`bar`** (move line 6 to be a new line 4 and move line 4 and 5 back 1 line), as shown in Figure 5.3b. The second proposed repair is to move line 3 and line 4 (the lines that use **`bar`**) after line 6 (the line that uses **`foo`**), as shown in Figure 5.3c. If neither of these repairs pass the validation check, the last proposed repair will propose to delete the line with the last method of interest (in this case, line 6 with **`foo`**), shown in Figure 5.3d.

```
public int exampleMethod() {
  ... //unimportant line
  int i = bar();
  someOtherCall(i);
  ...
  foo();
}
```

(a) The original example code.

```
public int exampleMethod() {
  ... //unimportant line
  foo();
  int i = bar();
  someOtherCall(i);
  ...
}
```

(b) The an example of moving `foo` before the `bar` method call.

```
public int exampleMethod() {
  ... //unimportant line
  ...
  foo();
  int i = bar();
  someOtherCall(i);
}
```

(c) The an example of moving `bar` after the `foo` method call.

```
public int exampleMethod() {
  ... //unimportant line
  int i = bar();
  someOtherCall(i);
  ...
}
```

(d) An example of deleting the second method call of interest (`foo`).

Figure 5.3: Examples of the different repairs tried by the method order repair technique.

5.3.2 Method Move Repair

Method reordering addresses another instance of bugs caused by object protocols. Since frameworks often change the state of objects internally, developers can call API methods in the right order, but at program points when the framework internals are in the incorrect state. This problem can occur when a method call occurs in the wrong point of the framework life-cycle or before other required state changes have occurred. One example is that the `Fragment` method `getActivity` can only be called after the `Fragment` has been completely initialized. This type of violation can be fixed by moving the problematic call to a different method definition, where hopefully the objects associated with the method call are in the correct state. Methods in the class that contain the problem under repair are tried first, followed by other methods in the application.

When an application implicitly calls a method on the current object instance, moving the method to a new class or `static` method requires adding the object instance to the method call. When moving an implicitly called method, FrameFix first tries to move the method call to locations where a variable instance of that type already exists. If those locations do not produce a fix, then, FrameFix creates a new instance of the object and calls the method on that object in each new tried location. These locations start with the `static` locations in the current class file and then other methods in the application.

For example, assume there is a class **`Foo`** with a failing implicit call to instance method **`bar`**, as shown in Figure 5.4a. When the **`bar`** method is moved to class **`Baz`**, first a new instance of **`Foo`** will be created. Then that instance will call the **`bar`** method, so the new code addition will compile, shown in Figure 5.4b. This is often needed for the `setArguments` directive, where `setArguments` needs to be called before the `Fragment` is instantiated.

When FrameFix moves certain method calls, it tries to ensure that the parameters used in the method call are still valid in a different context. FrameFix achieves this by scanning the preceding lines in the method definition for references to variables used in the method call to move. FrameFix will then copy any preceding lines in the current method where the parameters are altered. For example, if one of the parameters are set in the preceding line, then the preceding line will also be copied when moving the method (the line is copied in case that changed variable is still needed in a later line). If the moved code requires try-catch statements to compile, FrameFix will also copy the original try-catch statements to the next context.

5.3.3 Callback-Based Repair

The difficult part of using reference applications from GitHub is that there are many possible code sections to use as a reference. Since frameworks often implement inversion of control through methods that have the same signature across all applications, my insight is that there are likely other instances of the same callback that can be referenced to fix a directive violation. FrameFix will try to repair a faulty method by using examples of that method from GitHub.

If the previous repair techniques do not identify repairs, FrameFix searches GitHub for applications that implement similar methods (determined by the name of the method) and other keywords provided with the directive specification (e.g., for **inflate**, the keywords 'inflate' and 'Fragment' are also included). Once a matching application is found, the application code from

```java
public class Foo {

  public int m1() {
    bar();
    ... //unimportant line
  }

  public void bar() {
    ...
  }
}

public class Baz {
  public int m2() {
    ...
  }
}
```

(a) The original example code.

```java
public class Foo {

  public int m1() {
    ... \\unimportant line
  }

  public void bar() {
    ...
  }
}

public class Baz {
  public int m2() {
    Foo f = new Foo();
    f.bar();
    ...
  }
}
```

(b) An example of how FrameFix moves an implicit method to a new class, creating a new instance in the process, if no instance of the class is already found in the new context.

Figure 5.4: An example of the move method repair when moving a method to a new class.

To use a `Fragment`'s `OptionsMenu`, the application must contain both the call `setHasOptionsMenu(true)` and define `onCreateOptionsMenu`

(a) The directive used in the example.

```
public void onCreate(@Nullable Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  // set the user's search query
  searchQuery = getArguments().getString(QUERY);
}
  ...  //unimportant code
public void onCreateOptionsMenu(Menu menu, MenuInflater
   inflater) {
  ...
```

(b) Example code that violates the directive. `onCreateOptionsMenu` is defined but `setHasOptionsMenu(true)` is not called.

curl -n https://api.github.com/search/code?q=**onCreate**+**Fragment**+ **onCreateOptions-Menu**+in: file+language:java?page=1&per_page=100&sort=stars&order=desc

(c) The search query used to find similar code from GitHub. Terms in bold are developer defined repair variables related to the directive.

```
public void onCreate(
   @Nullable Bundle
  savedInstanceState) {
  super.onCreate(
     savedInstanceState);
  setHasOptionsMenu(true);
}
```

(d) The onCreate method from GitHub used for reference. This is compared to the onCreate method in the original code.

```
setHasOptionsMenu(true);
```

(e) The add list, which FrameFix creates by looking for lines in the reference application's method (`onCreate` method in Figure 5.5d) that are not used in the original application's method (`onCreate` method in Figure 5.5b).

```
public void onCreate(@Nullable Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  // set the user's search query
  searchQuery = getArguments().getString(QUERY);
  setHasOptionsMenu(true); //Added line!!!
}
```

(f) The repaired application with the line `setHasOptionsMenu(true)` added from the added list.

Figure 5.5: An example of how the callback-based repair fixes an application problem.

GitHub is reduced down to the method call of interest (e.g., if the repair is set to look for instances of the **onViewCreated** method, the application off GitHub is reduced down to only the **onViewCreate** method). FrameFix will then compare the method from GitHub to the method with the same signature in the application to repair. (using the same example, FrameFix compares the **onViewCreate** method from GitHub to the **onViewCreate** method in the application to repair). An example of this repair technique is shown in Figure 5.5.

To compare the two methods, the methods are converted to a list of method calls and a list of types per line, as shown in Figure 5.6. The intuition behind this approach is the application under repair likely needs to correct a method call or correct the types in the method call. Thus, the application under repair is first evaluated against the method from GitHub for method call differences, and then for type differences.

First, the method calls per line are compared. If the a line in the GitHub application's method is found to contain a method that is not in the application under repair, that line is saved as a possible line to add. If a line in the application under repair contains method call that is not used in the GitHub application's method, then that line is saved in the possible deletion list. The first proposed repairs will try to add a single line from the line to add list to the application under repair. If the proposed repairs do not pass the validation tests, then the next proposed repairs will try to delete a single line from the line to remove list. If deleting a single line does not produce a valid fix, then the next repairs generated will try all combinations of adding a single line from the line to add list and removing a single line from the line to delete list. This process repeats for adding and removing two or more lines until all add/delete combinations have been proposed. When lines are added from the GitHub application to the application under repair, the instance variables in the lines are changed to instance variable names with the same type in the application under repair.

If the application under repair is still not fixed, FrameFix will then recreate the addition and deletion list, but this time by comparing the types of variables in the lines of the method to compare. Lines are not directly compared (line 4 in one method is not always compared to line 4 in the other method). Instead, lines are compared on the order in which an instance of that type is found (e.g., if a **View** instance occurs on line 3 and line 5 one method and line 2 and 4 in the other method, then line 3 will be compared to line 2 and line 4 will be compared to line 5), to avoid the case where an added line always causes the following lines to not match. One exception

```
1 @Override
2 public View onCreateView(LayoutInflater inf, ViewGroup cont,
     Bundle saved) {
3   View v = inf.inflate(R.listView, cont, false);
4   dLE = new DLE(this, this, getActivity());
5   ((ListView) v.findView(R.DirList)).setAdapter(dLE);
6   return v;
7 }
```

(a) The original code.

```
3 inflate()
4 DLE(), getActivity()
5 findView(), setAdapter()
```

(b) The code converted to a list of method calls in each line. Constructors are included in the method call list and lines can contain multiple method calls.

```
3 View, LayoutInflater, constant_reference, ViewGroup, false
4 DLE
5 View, constant_reference, DirList
6 View
```

(c) The code converted to types per line. `R.` references become constant_references. `true` and `false` are separate type values. `this` references and method call return types are not included (unless the call is to a constructor) for ease of implementation purposes.

Figure 5.6: An example of how a method definition is turned into the list of method calls and types in each line. The repair technique uses both approaches to determine differences between the application to repair and the reference application from GitHub.

```
1 @Override
2 public View onCreateView(LayoutInflater inflater, ViewGroup
     container,
3   Bundle savedInstanceState) {
4   View v = inf.inflate(R.listView, cont, true);
5   dLE = new DLE(this, this, getActivity());
6   ((ListView) v.findView(R.DirList)).setAdapter(dLE);
7   return v;
8 }
```

(a) The original code with the problem

```
1 @Override
2 public View onCreateView(LayoutInflater inf, ViewGroup cont,
     Bundle saved) {
3   View v = inf.inflate(R.listView, cont, false);
4   dLE = new DLE(this, this, getActivity());
5   ((ListView) v.findView(R.DirList)).setAdapter(dLE);
6   return v;
7 }
```

(b) The example code off GitHub. This only differs from the original code by the type of the boolean on line 3.

```
3 View, LayoutInflater,
     constant_reference,
     ViewGroup, true
4 DLE
5 View, constant_reference,
     DirList
6 View
```

(c) The the original code converted to types per line.

```
3 View, LayoutInflater,
     constant_reference,
     ViewGroup, false
4 DLE
5 View, constant_reference,
     DirList
6 View
```

(d) The code from GitHub converted to types per line. Notice the difference from the original types on the first line.

```
1 View v = inf.inflate(R.
     listView, cont, true);
```

(e) The lines to delete list. These are the lines in the original code with type differences from the application on GitHub.

```
1 View v = inf.inflate(R.
     listView, cont, false);
```

(f) The lines to add list. These are the line in the code from GitHub that have type differences when compared to the original code.

Figure 5.7: An example of how the original code and the code from GitHub are converted in to lines to add and line to delete lists.

to evaluating the lines by type is that `true` and `false` are treated as different types. If a line is found to contain a type difference in the application under repair, then that line is added to the lines to delete. If a line is found to have a unique type in the application from GitHub, that line is added to the list of lines to add. All combinations of adding and deleting type lines are used to generate proposed fixes, the same way FrameFix generated repairs using method difference.

If FrameFix is unable to find a successful repair for the selected referenced application, another sample application is selected and the process is repeated for the new application. The sample applications are selected in the order returned from a GitHub API request for code that contains the method name of interest. This process is repeated until a set timeout is reached, currently set at one hour.

5.4 Evaluation

I evaluated FrameFix to demonstrate three claims: 1) that the directive violations covered by FrameFix apply to diverse set of applications in the framework, 2) that FrameFix works on real applications, and 3) FrameFix can repair problems in a diverse set of applications.

5.4.1 Repair of Manually Built Applications

To evaluate the repair process, I manually created applications that violated the nine Android directives mentioned in Section 4.4. I knew from the earlier investigation that these directives apply to a large number of real world applications (Section 4.2), so if the repair technique could fix these problems, then the fixes would apply to real-world applications. I created the repair scenarios by starting with a base application. The base application was either a sample application from the Google website[3] or, if the directive did not apply to the sample application, a default `Activity` application from AndroidStudio. Then, if I had a StackOverflow question that corresponded to the directive, I added code similar to the code/scenario mentioned in the question to the base application. For the violation that did not have a corresponding StackOverflow question, I created the application with a violation by manually creating code that violated the directive.

Framefix was able to fix seven out of the ten cases (counting the two ways to violate **OptionsMenu** as different cases). Three scenarios were not repaired due to limitations in the current approach. For the **OptionsMenu** case, the technique is unable to repair the case where `onCreateOptionsMenu` is undefined because a successful repair would require adding a new method to the application, which is not currently supported by the repair approach. The **GetResources** and **SetArguments** encounter similar problems. While the problematic methods could just be deleted to pass the check, often the application needs an alternative way to pass the data that the user intended with these calls, which requires adding significant code to fix. Addressing these problems are possible interesting areas of future work, particularly trying to guide repairs from the functionality of the failing method call.

[3]github.com/googlesamples/android-LNotifications

Application	Directive Violated	Repaired?	Repair Type
DeltaCamera	**GetActivity**	Yes	Move Method
PSLab	**GetActivity**	Yes	Move Method
RXDroid	**OptionsMenu**	(tests fail)	GitHub
RXDroid	**Inflate**	(tests fail)	GitHub

Table 5.1: The list of repairs on F-Droid applications with a problem detected.

Directive Violated	Repair Type	# Inj.	# Repaired
SetPackageSetSelector	Reorder Methods	4	4
SetContentView	Reorder Methods	22	22
SetTheme	Reorder Methods	19	19
SetInitialSavedState	Move Method	3	3
GetActivity	Move Method	4	1
Inflate	GitHub	3	1
OptionsMenu	GitHub	1	1

Table 5.2: The number of injected (inj.) F-Droid applications with the violations, and the number of injected applications repaired.

5.4.2 Repairs of F-Droid Applications

Table 5.1 shows that FrameFix is able to fix detected problems in real applications. Unfortunately, the number of repairs are small compared to the number of detected errors at this time. The main problem is that it is difficult to build most of the applications, due to dependencies not being found. Some of the other repair cases are situations that cannot be currently repaired — **GetResources** and **SetArguments**.

The successful repair cases are shown in Table 5.1. While FrameFix produces a fix for RXDroid, RXDroid's test cases fail because the testing code uses a version of Closure that is incompatible with more recent Java versions. I was unable to get the tests to work before or after the repair.

5.4.3 Repair of Injected Errors

To further evaluate FrameFix, I wanted to test the repair techniques in FrameFix on other directives that were not covered in the error dataset. To simulate repairing applications with problems, I created a script to inject the problem mentioned in the directive into a random but valid spot for violating the directive — if the problem could be injected into multiple spots in the application, a spot was chosen at random.

Next, I collected a set of application repositories where I could inject the problem in the source code. I collected these repositories from the open source applications in F-Droid that had a publicly accessible repository posted in the F-Droid metadata folder. 1,657 repositories met this requirement.

The list of available repositories was reduced down to the repositories in the dataset that were written in Java. The injection technique also needs to be able to build APKs, since the static analysis only worked on compiled Android executables. I only included applications that contained a set of test cases and passed the set of test cases without problems. Thus, I used the standard process to build a debug APK in an Android application and run the test cases for the build (the Gradle **`assembleDebug`** and **`test`** commands). Of the 1,657 repositories tested, 115 applications met all of these checks without errors.

Using those 115 application repositories, I checked which repositories used code that applied to the directives being violated, so injecting the problem did not require significantly changing the applications. Often only about five applications met all of these criteria for each directive that was tested. I also only tested the seven directive violation types that I are able to fix with the current FrameFix approach. The results of injecting repairs are shown in Table 5.2, showing that the repair technique is able to fix most of the injected problems. FrameFix was not perfect when repairing the **GetActivity** cases, due to the difficulty of finding a valid method to call **`getActivity`**, and the **Inflate** cases due to injecting the problem in a long method definition. This long method definition caused the repair to generate a large number of possible repairs, eventually timing out before successfully repairing the application.

5.5 Limitations

While I designed FrameFix to apply to multiple frameworks in multiple languages, I have not verified that FrameFix applies to other frameworks or languages, due to the overhead of incorporating FrameFix into another static analysis tool. While I believe that the size of the dataset is enough to demonstrate the feasibility of the claims that FrameFix handles a wide range of cases, a larger dataset could reduce the possibility of sampling error.

FrameFix's repair approach is currently limited to small change repairs. When a directive violation requires the addition of a new class or method to solve the problem, FrameFix is unable to repair those cases. A promising future approach that I have started to investigate is using template repair to fix the unsupported repair cases. Template repair is showing promise for one of the OptionsMenu cases, and might be able to fix the other two cases. FrameFix is also not able to fix problems due to environmental settings, such as the phone application not working because the application requires a permission that the user has not granted to the application.

5.6 Context and Prior Work

FrameFix uses static analysis to identify bugs and validate that bugs have been repaired. There exists other techniques that use static analysis at various points in the automated program repair process. One automated repair approach that uses static analysis validation is Phoenix [12]. Phoenix uses FindBugs [10], a static analysis tool for detecting problematic code patterns, to collect multiple examples of bug patterns with associated fixes from source control history. It uses these examples to synthesize transformation for newly-identified static violations.

Phoenix's approach could be used to address plugin issues, but requires multiple fix examples to determine the correct transformation steps, which FrameFix does not require. Phoenix would likely have difficulty with the directives that require moving method calls between functions also. FootPatch [88] uses separation logic and bi-directional static analysis to identify and fix heap-based problems. This technique uses static analysis to locate problems, calculate possible fixes, and to validate repairs. FrameFix uses static analysis to locate bugs and to validate repairs, but it does not use static analysis to calculate or identify the fix. Although it is conceivable that separation logic could encode the properties captured by directives, it is not a natural fit, and would certainly pose a significant annotation burden on the analysis developer, when compared to FrameFix. Infer [19], the current static analysis tool used in Footpatch, is also known to have technical limitations for Android lifecycles, which currently limit the tool's effectiveness for repairing Android plugin issues [31]. Getafix [11], generalizes a set of collected bug fixes to AST-based repair patterns. GetAFix, unlike FrameFix, requires a set of previous fixes to create repair pattern to fix a new bug category. ErrorProne uses static analysis to detect problems and recommend fixes, but the static analysis is limited to detecting bug patterns (which means it does not use dataflow or callgraphs to detect errors, as FrameFix does) [81]. There are other automated repair approaches that use static validation but manually define how to repair each error class. For example, the repair approach by Liu et al. [62], that uses FindBugs to identify issues and repairs problems with manually specified patterns, and Refaster [92], which defines refactoring patterns to fix code issues. While FrameFix is similar to these approaches, since they use static analysis to identify issues, FrameFix attempts to reduce the specification burden on developers by providing fix approaches that work for multiple directive violations and by not requiring developers to specify the fix (although developers currently have to specify variables for the repair process related to the directive).

FrameFix borrows from dynamic repair techniques, such as GenProg [48, 59], that randomly produce fixes within a given repair search space. Template-based repair, such as PAR [53], may be able to fix the directives addressed by FrameFix if a template were created for each directive. These approaches verify possible repairs using test cases, which adds extra specification burden on users to fix directives. These techniques would require that plugin developers write tests, that are often non-trivial, for particular directive violations. As a general comment, FrameFix is unique compared to other repair techniques in the way it blends elements of past heuristic and semantic repair techniques. FrameFix uses static analysis to locate and validate the error, but it does not use deep semantic reasoning, synthesis, or logic to produce possible repairs. Instead, FrameFix only limits the repair space using framework-specific heuristics.

FrameFix is also closely related to techniques that automatically repair problems in framework applications. One closely related work is Droix, which repairs crashing Android applications using manually created patterns [86]. While the repair patterns in Droix served as a source of inspiration for the repairs in FrameFix, Droix differs from FrameFix in that the repair process requires a recorded event sequence to produce a crash, and that repairs were manually validated - either by comparing the intermediate code change to developer changes or by running the application to determine if the crash was fixed (dynamic validation). Another difference is that Droix was designed to work on the Android byte code. FrameFix, instead, produces repairs in the source code, which is makes it easier for developers to integrate the fix into their project's source code repository. Fan et al. categorized a large number of Android exceptions and extracted common repair patterns for problems that produce exceptions, by looking at developer produced repairs [33]. This work did not not try to automatically repair any problems with their repair patterns, so it is not an automated repair technique. The move method repair (Section 5.3.2) in this work is based on the *Works in right callback* repair pattern found in study of developer produced repairs, but differs in that method calls can be moved to other methods that are not callbacks. In contrast to these approaches, FrameFix was designed to be framework independent, with the novel GitHub repair approach, and thus handles a wider set of framework issues than those handled in the Android repair techniques.

Other research has taken a different approach to repairing framework applications than FrameFix. One approach to framework repair is to use contracts and dynamically created object behavior models to guide repairs [93]. This approach requires a significant developer investment to work for most frameworks, since both the framework and application would need to be written in a language that uses contracts. FrameFix instead only requires specifying the directive to check and how to translate the directive into code if a custom check is required. Another tool, SemDiff, fixes out-of-date API calls by recommending methods that were added in the version where the API call became out-of-date [25]. SemDiff addresses an important issue in Android plugin debgugging as noted by the challenges that developers faced in the debugging study tasks due to different versions of Android (Section 3.2.2). However, SemDiff addresses a different bug class than FrameFix and should be used at different points the development process (Developers should use SemDiff when the framework API is updated, while developers could use FrameFix to address more fundamental development concerns).

5.7 Summary

In this chapter, I addressed the last goal of the thesis statement: to make an automated repair technique for plugins. This repair technique, FrameFix, tries to exploit the unique aspect of framework plugins to repair problems that may arise in plugin development. I evaluated FrameFix on a manually created application set, real applications from F-Droid, and problems automatically injected into applications from F-Droid. I showed that FrameFix was accurate in that it could fix all of the injected problems for five directives. I also showed that FrameFix's repair approach generalizes to multiple directives and is able to fix detected problems in real applications.

Chapter 6

Conclusion and Future Work

6.1 Summary

Due to the wide adoption of frameworks in the software development industry, small improvements to the plugin development process could have significant impact. To better understand the problems that plugin developers face, I performed a plugin debugging study where I analyzed the challenges developers encounter when fixing plugin problems. Some of the main findings from this study are:

- Developers have difficulty with the distributed nature of related debugging content. Inversion of control, object protocols, and declarative artifacts can interact in a way where the information required to solve a problem is not easily found near the fault location. This can make it harder for developers to correctly diagnose the problem and produce a correct fix.
- Developers would benefit from guidance on possible fixes. Frameworks both restrict the design of plugins, to interact correctly with them, and provide a common approach to solving problems that are regularly encountered in that domain. In the context of debugging plugins, developers encounter difficulty determining the expected or recommended way to achieve their goal. A tool could assist developers by proposing the recommended solution in the framework.
- Developers seem to encounter particular difficulty with state-based issues. Since frameworks perform many changes to object states that are not always easily visible to a plugin developer, developers have difficulty understanding both the states that lead to an error condition as well as the possible states that a correct solution must handle.

One of the goals of this dissertation is to produce results that generalize. For the debugging study, the goal was to produce conclusions that generalize to other plugin development contexts. This was achieved by focusing the study on directive violations, a set of problems that are not plugin specific, and by studying how developers solved problems in two fundamentally different frameworks.

The results from the previous study implied that plugin developers would benefit from an automated repair technique. An automated repair technique needs to be able to both identify a problem, so the technique can determine when the problem is fixed, and propose repairs that may fix it. To ease the burden of identifying problems, I created a way to statically check plugins for directive violations. In this technique, a person interested in creating a new check for a directive violation would use the specification language to guide the tool to identify problems in plugins. I evaluated the generalizability of the specification language and tool in two ways:

- A taxonomy of object protocols. I evaluated the specification language on the Beckman taxonomy of object protocols and found that five of the seven categories could be easily encoded in the language. This result implies that the language generalizes to a wide variety of cases in which a developer may use the language and that the language can be adapted to handle the cases.
- The F-Droid applications. I also evaluated the static analysis checks on a set of real application to show that the technique generalizes to real applications. I found that the set of directives covered by the tool apply to over 80% of applications in the dataset and that the technique is able to detect problems in real applications.

The other important part of a repair technique is the repair strategy. I created a tool, FrameFix, for fixing directive violations in a plugin. FrameFix is based on the unique aspects of framework programming, the heavy use of object protocols and similarly named methods to implement inversion of control, to guide its repair strategies. FrameFix uses method reordering and differences determined by comparing to a reference application to guide repairs. I evaluated FrameFix in three ways:

- On a reference set of manually built applications. I used this set of applications to demonstrate how well these repair techniques generalize across the types of problems encoded in directives. I found that the technique was able to repair seven of the nine cases.
- On applications in F-Droid. The goal of this evaluation was to determine how well FrameFix generalized to real applications. While FrameFix was only able to repair a small number of the detected problems, often due to difficulties building the application from source code, the technique demonstrated that it can work on real applications.
- On a set of problems injects into F-Droid applications. I also wanted to evaluate the accuracy of FrameFix. To do this, I randomly injected directive violation into applications where the directive applied. I found that FrameFix was able to repair all injected cases except for those for two directives.

This dissertation aims to apply a systematic method to improving the plugin development process, from scoping and identifying the problem to producing a proof-of-concept tool to address a significant challenge for developers. I hope this work can be used to guide future improvements in the plugin development process.

6.2 Discussion

In this section I discuss the kinds of bugs in framework application code that FrameFix can and cannot both detect and fix (Section 6.2.1). I then discuss benefits and challenges presented by the different implementation decisions in FrameFix (Section 6.2.2) providing a summary of the implementation trade-offs that were made to implement FrameFix. I then describe the essential parts of FrameFix (Section 6.2.3), which would be required to re-implement FrameFix, possibly for a new context.

6.2.1 Problems handled by FrameFix

Here I will attempt to characterize the problems that FrameFix can and cannot address in the current implementation. One class of problems that FrameFix is designed to fix is single-threaded, intra-procedural method call ordering issues. Another problem class FrameFix is designed to fix is small (measured in number of lines added or deleted) problems in methods that are commonly extended in applications. These problems likely need to stem from a method call problem (the method is missing, the parameters are incorrect, the call occurs in the wrong place). FrameFix is unlikely to fix issues by adding extra control statements, such as if blocks or try-catch blocks. FrameFix is not likely to be able to fix multi-threading issues, cases where fixes require large code additions (such as adding new method definition or adding a new way to pipe data across classes in the plugin), or problems that arise from environmental settings (such as an application lacking a required permission). FrameFix's use of static analysis also creates scope limitations. There are problems that are difficult for static analysis to detect, such as problems that arise from looping too many times. There are also limitations related to understanding developer intent — it might be difficult to determine something is a problem if it is difficult to determine whether the developer intended it or not. Finally, FrameFix's static analysis is currently not designed for declarative artifacts. While FrameFix could be extended to address problems in declarative artifacts, they are not not currently addressed by the tool.

6.2.2 Benefits and Limitations of Implementation Decisions

Directives
There are various approaches to studying plugin problems. For instance, one way would be to collect applications with known issues and then filter out problems that were application-specific. Directives provide a defined way to focus on plugin problems that are not application-specific. But, since directives are specified in natural language, there is an extra step to specify the directive in a checkable form.

Static analysis
FrameFix's use of static analysis provides both implementation benefits and difficulties. The benefits of the approach is that FrameFix can identify a large subset of directive violations and that the analysis provides useful information for the fix process, such as the possible valid method move locations (although static analysis is not required to get the information). Static analysis is not a perfect solution because it can be difficult to detect some problems with static analysis,

such as whether a `Fragment` is attached to in `Activity` in a given context, and it's harder to confirm the problem in the application. Incorporating other analysis techniques, such as dynamic analysis, would allow FrameFix to identify more problems and provide more validation to generated repairs.

Android

The decision to implement FrameFix for Android provided multiple benefits. The first benefit is the large number of Android applications that can be analyzed and used for reference. Another benefit is the large amount of documentation that makes collecting directives easier. There are some implementation difficulties associated with selecting Android. The first is that Android is a framework that uses it own byte code, meaning that the implementation of FrameFix for Android could not be tested on other frameworks, which makes it difficult to confirm that the technique generalizes to other frameworks. Another problem is that the Android framework and associated tool chain undergoes rapid evolution. Many of the directives collected applied to sections of the framework that became deprecated before FrameFix was finished. Simultaneously, Google was constantly updating the build tools; I had to make multiple updates to applications' build process files throughout the study. Another important source of implementation difficulty is that Android switched the preferred language from Java to Kotlin, so there are many applications in both languages, which increases the difficulty of statically analyzing Android applications. Android also adds other sources of difficulty when trying to analyze the application's call graph. Android frequently uses the observer pattern, which requires objects to dynamically register to respond to events and obscures the static call graph.

FlowDroid

FrameFix was built on top of FlowDroid (version 2.6.1). FlowDroid provides an analysis infrastructure for Android applications. FrameFix uses FlowDroid to analyze APKs, and provide the necessary data flow and call graph information to detect and fix the problems detected by the technique. FlowDroid also added some limitations to the implementation of FrameFix. One source of limitations was that FlowDroid cannot always generate call graphs for applications. I also encountered call graph limitations when trying to use the tool. One problem I encountered was that method calls of anonymous classes did not appear in the call graph. Another source of imprecision is that Android has an implied ordering of method calls, for example, that the **`onTabSelected`** method is called after the initialization function of a **`FragmentTabListener`** class. This order was lost from the call graph. Security researchers have uncovered some limitations of FlowDroid through mutation testing, such as FlowDroid not including the **`DialogFragment.onCreateDialog()`** callback in the call graph [17].

Source code repairs

Repairing applications at the source code level can make it easier to produce a repair — the fix is just a textual substitution in the right file — instead of repairing Android byte code. Source code level repairs can also be more useful for people developing an application, so they can incorporate the fix into the application. Fixes at the source code level also makes it easier to compare the current problematic application with reference applications from GitHub, since the reference applications do not have to be compiled. However, many plugin users will only have the final APK and not the source code. The current implementation of FrameFix could identify problems in the APK but could not fix the issue without the source code. Repairing applications at the source code level adds other difficulties, since evaluating the repairs requires compiling the source code of other applications. FrameFix could be extended to do fixes at the btye code level, but the fix techniques would need to be altered to work at the byte code level.

6.2.3 Essential Parts of Technique

At a high level, FrameFix has the same needs as any automated repair technique but with a focus on directive violations: the ability to detect the directive violation, fix the directive violation, and validate that the violation is fixed. To detect directive violations, FrameFix needs to be able to perform dataflow analysis (to determine method call ordering) and determine the states of objects relevant to the directive (which FrameFix did with call graphs). The different repair components, discussed in Section 5.3, each address different directives. These components either make assumptions or contain implementation details that increase their effectiveness. The move method repair approach can only work when objects in the problematic method call are in different states in different methods. If this assumption is not true for the plugin, a different approach to changing objects' state would be required. The GitHub repair technique requires other applications built in the framework to use as a reference. The repair technique also requires that the problem occurs in a method that is defined in multiple applications and can be found from either a matching method signature (for example, the **`onCreate`**'s method signature of an **`Activity`**) or a matching subset of the method signature. Another essential requirement is being able to find a correct implementation of the problematic code in a reference application. This requirement limits FrameFix to addressing problems that are common to plugins in the framework and have a context-independent fix (with the exception of variable names and other parameters). FrameFix could be extended to produce context-dependent fixes, if FrameFix could detect the right context and apply the right context-specific repair. FrameFix also needs to find a reference plugin with the necessary method differences to fix the problematic plugin. FrameFix attempts to find reference applications using a GitHub search query that incorporates developer provided keywords related to the directive, while also sorting the result by the most starred repositories and filtering by language. The query returns a list of reference applications.

The GitHub repair technique also employs strategies to reduce the detection of spurious differences between the current application and the reference application. While the reduction strategies are not essential to the technique, reducing spurious differences can be essential to whether the technique produces a repair before a set timeout. The repair technique tries to reduce spurious differences by first treating each type instance as a generic type variable and not the specific instance name (variable names are not likely to match between two different applications). Another strategy that the technique uses to reduce spurious differences is to match lines by types instead of line numbers. The technique matches lines by type through the process of comparing all lines where variables of a type are used with all lines in the reference application where the variable type is used, in-order. The reason that the technique does not compare by line number (the first line of the problematic method in the plugin is compared to the first line of the method in the reference plugin, and so on) is that the addition of an extra line to one of the methods, such as an early log statement, would cause the rest of the lines to not match.

6.3 Future Work

There are many possible ways to build on this dissertation. In the short term, there are projects that another researcher or I could do to further evaluate the current tool, and make the tool more usable, either by reducing human input or applying the tool to new situations. In the long term, there are also multiple research ideas that could build off this dissertation.

6.3.1 Further Evaluation

While FrameFix was evaluated for the qualities discussed in the thesis statement, there are other qualities that are important to developers. One important quality for developers to use the tool in practice is usability. FrameFix would benefit from a human study into how developers would use the tool in a development situation. This user-focused study would either provide evidence that the tool assists developers in the plugin development process or other challenges that arise in the development process when FrameFix is used. This study could uncover future research challenges and provide further validity to the claims in this study.

6.3.2 Possible Near-Term Improvements

There are couple of ways FrameFix could be improved. Currently, FrameFix is only able to address a limited set of repair situations. One case where FrameFix has difficulty is when the repair requires a significant amount of code, such as a new method. Repair templates, where a base fix is adjusted to the current code and problem, could be one way to add the necessary code. Preliminary investigations into one of the **OptionsMenu** cases has shown promise, but there were challenges associated with creating a template that would apply to the different **setArguments** and **getResources** cases that may arise. Another way to fix these directive cases may be to apply learning repair techniques, by proposing new fixes based on the code elements used in past fixes.

FrameFix currently has a relatively high specification cost, requiring developers to understand the framework to create a static analysis and repair specification. It may be possible to combine natural language processing and knowledge of the framework internals to automatically translate natural language directives into the information required by FrameFix. Further work in this direction could attempt to automatically infer the required knowledge from the internal framework code, reducing the required inputs to specify a directive to fix to only the framework and the natural language directive. Another possibly might be automatically inferring usage protocols [77] and then converting those into the inputs FrameFix requires.

The evaluation of FrameFix discovered that it is difficult to fix the source code and then build the fixed version of many Android applications. While FrameFix is currently implemented to produce source code repairs, a future version of FrameFix could be created to fix the application in byte code. This change would increase the number of applications that the tool could fix, since the source code may not be available for all applications.

6.3.3 Long-Term Ideas

There are some other research directions that FrameFix makes available. While FrameFix is able to address plugin problems, FrameFix addresses these problems from a method-call focused perspective — assuming that the problem is related to one or a few method calls in the application. It may be possible to adapt FrameFix or create another repair technique to automatically repair other types of problems that plugin developers encounter, such as problems that occur in declarative artifacts.

Another research direction is to investigate how to make repair techniques that use source code knowledge less language-dependent. One major limitation of FrameFix is the overhead required to use the tool on a framework in a new language. The technique currently needs a static analysis tool to identify problems in the application, as well as information about method call location and call graphs to produce repairs. For FrameFix, it may be possible to reduce this dependency through the use of other fault localization techniques, such as only using a set of test cases, and other repair strategies. For future automated repair research, improving language independence may involve investigating which repair approaches are language dependent and how the fix rate of the approach would change if a language-independent choices were used instead. This knowledge would help automated repair tool developers select the intended trade-off in their repair designs and also serve as a source of motivation for decreasing language dependence for automated repair techniques.

Another interesting research possibility for automated repair would be to create other techniques to find relevant code for reference and apply it to currently problematic section of code. Previous work has found sections of code with similar input-output characteristics [6] and this work used similarly named methods to find relevant sections. It may be possible to use another heuristic, like the prevalence of certain method calls in a section of code, to find a reference that can guide a successful repair.

Bibliography

[1] `https://www.statista.com/statistics/444476/google-play-annual-revenue/`, Accessed Mar. 16th, 2020.

[2] `https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/`, Accessed Mar. 4th, 2020.

[3] `stackoverflow.com/tags`, Accessed Mar. 4th, 2020.

[4] `https://f-droid.org/en/` (dataset: https://doi.org/10.5281/zenodo.3698376), Accessed Oct. 11th, 2019.

[5] Hadil Abukwaik, Mohammed Abujayyab, and Dieter Rombach. Towards seamless analysis of software interoperability: Automatic identification of conceptual constraints in api documentation. In *European Conference on Software Architecture*, ECSA '16', pages 67–83, 2016.

[6] Afsoon Afzal, Manish Motwani, Kathryn Stolee, Yuriy Brun, and Claire Le Goues. Sosrepair: Expressive semantic search for real-world program repair. *IEEE Transactions on Software Engineering*, (to appear), 2020.

[7] Hiralal Agrawal, Joseph R. Horgan, Saul London, and W. Eric Wong. Fault localization using execution slices and dataflow tests. In *International Symposium on Software Reliability Engineering*, ISSRE '95, pages 143–151, Oct 1995.

[8] Jonathan Aldrich. The power of interoperability: Why objects are inevitable. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! '13, pages 101–116, 2013.

[9] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Programming Language Design and Implementation*, PLDI '14, pages 259—269, 2014.

[10] Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, Sep. 2008.

[11] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *ACM on Programming Languages*, 3(OOPSLA), 2019.

[12] Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. Phoenix: Automated data-driven synthesis of repairs for static analysis violations. In *Joint Meeting of the European Software*

Engineering Conference and the Symposium on the Foundations of Software Engineering, ESEC/FSE '19, pages 613–624, 2019.

[13] Nels E. Beckman, Duri Kim, and Jonathan Aldrich. An empirical study of object protocols in the wild. In *European Conference on Object-Oriented Programming*, ECOOP'11, pages 2–26, 2011. ISBN 978-3-642-22654-0.

[14] Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. *SIGSOFT Software Engineering Notes*, 30(5):217—-226, September 2005.

[15] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Conference on Object-Oriented Programming Systems and Applications*, OOPSLA '07, pages 301—-320, New York, NY, USA, 2007.

[16] Kevin Bierhoff, Mark Grechanik, and Edy S. Liongosari. Architectural mismatch in service-oriented architectures. In *Workshop on Systems Development in SOA Environments*, SDSOA '07.

[17] Richard Bonett, Kaushal Kafle, Kevin Moran, Adwait Nadkarni, and Denys Poshyvanyk. Discovering flaws in security-focused static analysis tools for android using systematic mutation. In *USENIX Conference on Security Symposium*, SEC'18, pages 1263—-1280, USA, 2018. USENIX Association.

[18] M. Bruch, M. Mezini, and M. Monperrus. Mining subclassing directives to improve framework reuse. In *Mining Software Repositories*, MSR '10, pages 141–150, 2010.

[19] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods*, NFM '15, pages 3–11, 2015.

[20] Kathy Charmaz. *Constructing Grounded Theory*. Sage Publications, Los Angeles, CA, 2nd edition, 2014. ISBN 978-0-85702-914-0.

[21] Jaspan Ciera. *Proper Plugin Protocols*. PhD thesis, Carnegie Mellon University, 2011.

[22] Zack Coker, Joshua Sunshine, and Claire Le Goues. Framefix: Automatically repairing statically-detected directive violations in framework applications. In *(under submission)*. URL `https://drive.google.com/open?id=1VkCK5JokajdVxBermvq20rdeSa7vAJoA`.

[23] Zack Coker, David Gray Widder, Claire Le Goues, Christopher Bogart, and Joshua Sunshine. A qualitative study on framework debugging. In *International Conference on Software Maintenance and Evolution*, ICSME '19, pages 568–579, 2019.

[24] James S. Collofello and Larry Cousins. Towards automatic software fault location through decision-to-decision path analysis. In *International Workshop on Managing Requirements Knowledge*, AFIPS '87, pages 539–544, 12 1987.

[25] Barthelemy Dagenais and Martin P. Robillard. Semdiff: Analysis and recommendation support for api evolution. In *International Conference on Software Engineering*, ICSE '09, pages 599–602, 2009.

[26] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program repair with quan-

titative objectives. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 383–401. Springer International Publishing, 2016.

[27] Uri Dekel. *Increasing awareness of delocalized information to facilitate API usage*. PhD thesis, Carnegie Mellon University, 2009.

[28] Uri Dekel and James D. Herbsleb. Improving api documentation usability with knowledge pushing. In *International Conference on Software Engineering*, ICSE '09, pages 320–330, 2009.

[29] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*, ECOOP '04, pages 465–490, 2004.

[30] Mariangiola Dezani-Ciancaglini and Ugo de'Ligruoro. Sessions and session types: An overview. *Web Services and Formal Methods*, 6194:1–28, 2010.

[31] Facebook. `https://fbinfer.com/docs/limitations`, Accessed May 5th, 2020.

[32] George Fairbanks, David Garlan, and William Scherlis. Design fragments make using frameworks easier. In *Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 75–88, 2006.

[33] Lingling Fan, Ting Su, Sen Chen, Meng Guozhu, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. Large-scale analysis of framework-specific exceptions in android apps. In *International Conference on Software Engineering*, ICSE '18, pages 408–419, 2018.

[34] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, October 1997.

[35] Martin Fowler. Fluentinterface. `https://martinfowler.com/bliki/FluentInterface.html`, Accessed April 29th, 2020.

[36] Garry Froehlich, H. James Hoover, Ling Liu, and Paul Sorenson. Hooking into object-oriented application frameworks. In *International Conference on Software Engineering*, ICSE '97, pages 491—-501, 1997.

[37] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *International Conference on Software Engineering*, ICSE '95, pages 179—185, 1995.

[38] Google. Android. www.android.com, 2017. Accessed: 2/15/17.

[39] Google. `https://opensource.googleblog.com/2019/07/truth-10-fluent-assertions-for-java-and.html`, Accessed May 6th, 2020.

[40] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Association for the Advancement of Artificial Intelligence*, AAAI '17', pages 1345–1351, 2017.

[41] Markku Hakala, Juha Hautamäki, Kai Koskimies, Jukka Paakki, Antti Viljamma, and Jukka Viljamaa. Annotating reusable software architectures with specialization patterns. In *International Conference on Software Architecture*, ICSA '01, pages 171–180, 2001.

[42] Kohei Honda. Types for dyadic interaction. In *International Conference on Concurrency Theory*, CONCUR '93, pages 509–523, 1993.

[43] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *Journal of the ACM*, 63(1):9:1–9:67, March 2016. ISSN 0004-5411.

[44] Daqing Hou, H. James Hoover, and Eleni Stroulia. Supporting the deployment of object-oriented frameworks. In *Advanced Information Systems Engineering*, CAiSE 2002, pages 151–166. Springer Berlin Heidelberg, 2002.

[45] Daquing Hou and H. James Hoover. Using scl to specify and check design intent in source code. *IEEE Transactions on Software Engineering*, 32(6):404–423, 2006.

[46] Ciera Jaspan and Jonathan Aldrich. Checking semantic usage of frameworks. In *Library-Centric Software Design*, LCSD '07, pages 1–10, 2007.

[47] Ciera Jaspan and Jonathan Aldrich. Retrieving relationships from declarative files. 2009.

[48] Ciera Jaspan and Jonathan Aldrich. Are object protocols burdensome?: An empirical study of developer forums. In *SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '11, pages 51–56, 2011.

[49] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In *Computer Aided Verification*, CAV '05, pages 226–238, 2005.

[50] Ralph E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42, October 1997. ISSN 0001-0782.

[51] Taylor T. Johnson, Stanley Bak, and Steven Drager. Cyber-physical specification mismatch identification with dynamic analysis. In *International Conference on Cyber-Physical Systems*, ICCPS '15, pages 208–217, 2015.

[52] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *International Conference on Software Engineering*, pages 467–477, 2002.

[53] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering*, ICSE '13, pages 802–811, 2013.

[54] Amy J. Ko and Brad A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *International Conference on Software Engineering*, ICSE '08, pages 301–310, 2008.

[55] Amy J. Ko, Brad A. Myers, and Htet Htet Aung. Six learning barriers in end-user programming systems. In *Visual Languages - Human Centric Computing*, VLHCC '04, pages 199–206, 2004.

[56] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions of Software Engineering*, 32(12):971–987, December 2006. ISSN 0098-5589.

[57] Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector,

and Scott D. Fleming. How programmers debug, revisited: An information foraging theory perspective. *IEEE Transations of Software Engingeering*, 39(2):197–215, February 2013.

[58] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: Syntax- and semantic-guided repair synthesis via programming by examples. In *2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 593–604, 2017.

[59] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering*, ICSE '12, pages 3–13, 2012.

[60] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.

[61] Hongwei Li, Sirui Li, Jiamou Sun, Zhenchang Xing, Xin Peng, Mingwei Liu, and Zhao Xuejiao. Improving api caveats accessibility by mining api caveats knowledge graph. In *International Conference on Software Maintenance and Evolution*, pages 183–193, 09 2018.

[62] Kui Liu, Dongsun Kim, Tegawende F. Bissyande, Shin Yoo, and Yves Le Traon. Mining fix patterns for findbugs violations. *Transactions on Software Engineering*, (Early Access): 1–24, 2018.

[63] Francesco Logozzo and Thomas Ball. Modular and verified automatic program repair. In *Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 133—-146, 2012.

[64] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE '15, pages 166–178, 2015.

[65] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Principles of Programming Languages*, POPL '16, pages 298–312, New York, NY, USA, 2016. ACM.

[66] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. Sapfix: Automated end-to-end repair at scale. In *International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '19, pages 269–278, 2019.

[67] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *International Conference on Software Engineering*, ICSE '15, pages 448–458, 2015.

[68] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *38th International Conference on Software Engineering*, ICSE '16, pages 691–701, New York, NY, USA, 2016.

[69] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. What should developers be aware of? An empirical study on the directives of api documentation. *Empirical Software Engineering*, 17(6):703–737, 2012.

[70] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan. The design space of bug fixes

and how developers navigate it. *IEEE Transactions on Software Engineering*, 41(1):65–81, Jan 2015.

[71] B. A. Myers, A. J. Ko, T. D. LaToza, and Y. Yoon. Programmers are users too: Human-centered methods for improving programming tools. *Computer*, 49(7):44–52, July 2016.

[72] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, 2013.

[73] Thanh-Toan Nguyen, Quang-Trung Ta, and Wei-Ngan Chin. Automatic program repair using formal verification and expression templates. In *Verification, Model Checking, and Abstract Interpretation*, VMCAI '19, pages 70–91, 2019.

[74] Naoya Nitta, Izuru Kume, and Yasuhiro Takemura. A method for early detection of mismatches between framework architecture and execution scenarios. In *Asia-Pacific Software Engineering Conference (APSEC)*, volume 1 of *APSEC '13'*, pages 517–522, 2013.

[75] Danilo Dominguez Perez and Wei Le. Generating predicate callback summaries for the android framework. In *Mobile Software Engineering and Systems*, MOBILESoft '17, pages 68–78, 2017.

[76] David Piorkowski, Scott D. Fleming, Christopher Scaffidi, Margaret Burnett, Irwin Kwan, Austin Z. Henley, Jamie Macbeth, Charles Hill, and Amber Horvath. To fix or to learn? how production bias affects developers' information foraging during debugging. In *International Conference on Software Maintenance and Evolution*, ICSME '15, pages 11–20, Sep. 2015.

[77] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. Statically checking api protocol conformance with mined multi-object specifications. In *International Conference on Software Engineering*, ICSE '12, pages 925—935, 2012.

[78] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *International Conference on Software Engineering*, ICSE '14, pages 254–265, 2014.

[79] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[80] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *International Conference on Software Engineering*, ICSE '17, pages 404—-415, 2017.

[81] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM (CACM)*, 61 Issue 4:58–66, 2018.

[82] Margrit Schreier. *Qualitative Content Analysis in Practice*. EBL-Schweitzer. SAGE Publications, Thousand Oaks California, USA, 2012. ISBN 9781849205931.

[83] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded theory in software engineer-

ing research: A critical review and guidelines. In *International Conference on Software Engineering*, ICSE '16, pages 120–131, 2016.

[84] R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, January 1986.

[85] Joshua Sunshine. *Protocol Programmability*. PhD thesis, Carnegie Mellon University, 2013.

[86] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. Repairing crashes in android apps. In *International Conference on Software Engineering*, ICSE '18, pages 187–198, 2018.

[87] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. Automatically generated patches as debugging aids: A human study. In *Foundations of Software Engineering*, FSE '14, pages 64—-74, 2014.

[88] Rijnard van Tonder and Claire Le Goues. Static automated program repair for heap properties. In *International Conference on Software Engineering*, ICSE '18, pages 151–162, 2018.

[89] Manuela Veloso, Joydeep Biswas, Brian Coltin, and Stephanie Rosenthal. Cobots: Robust symbiotic autonomous mobile service robots. In *International Joint Conference on Artificial Intelligence*, IJCAI'15, pages 4423—-4429, 2015.

[90] Iris Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5):459 – 494, 1985.

[91] Yan Wang, Hailong Zhang, and Atanas Rountev. On the unsoundness of static analysis for android guis. In *State Of the Art in Program Analysis*, SOAP '16, pages 18–23, 2016.

[92] Louis Wasserman. Scalable, example-based refactorings with refaster. In *Workshop on Refactoring Tools*, WRT '13', 2013.

[93] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis*, ISSTA '10, pages 61–72, 2010.

[94] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, ICSE '09, pages 364–374, 2009.

[95] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering*, ASE '13, pages 356–366, 2013.

[96] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *International Conference on Software Engineering*, ICSE '15, pages 89–99, 2015.

[97] Andreas Zeller. Yesterday, my program worked. today it does not. why? In *European Software Engineering Conference Held Jointly with Foundations of Software Engineering*, ESEC/FSE '99, pages 253–267, 1999.

[98] Andreas Zeller. Automated debugging: Are we close? *Computer*, 34(11):26–31, 2001.

[99] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. Analyzing APIs documentation and code to detect directive defects. In *International Conference on Software Engineering*, ICSE '17, pages 27–37, 2017.