Automated identification and repair of state-based framework directive violations

Ph.D. Dissertation Proposal
Zack Coker
zfc@cs.cmu.edu

August 28th 2018

1 Abstract

Professional developers use software frameworks for the benefits gained from *architectural reuse*: the process of applying previously proven designs to new applications in a given domain, which saves developers time and reduces project uncertainty. Unfortunately, for frameworks to support architectural reuse, they must make a set of assumptions to interact with a diverse set of applications. These assumptions challenge developers because they create a set of constraints imposed by the framework. To investigate how framework imposed constraints affect developers, I conducted a human study on debugging violations of framework constraints. From this study, I found that the most time-consuming debugging difficulty developers faced was caused by the state restrictions on objects in the framework. I also found that developers had difficulty fixing state-based framework bugs, even when provided the failure location, implying that fixing the bug is the hardest step of the framework application debugging process. To address this issue, I propose FrameFix, a technique to automatically repair state violations in frameworks. The central innovations in this technique are a state-based fault localization approach and a template-based repair approach for these violations. I will evaluate the accuracy of the generated repairs on sample programs with framework state violations. The ultimate goal of the proposed work is to provide a way for framework designers to improve developer experience and reduce the challenges of framework development.

2 Introduction

Software frameworks increase the productivity of professional software developers [31]. Due to the known and substantial productivity gains, frameworks are an important tool of the software development industry. The importance of frameworks is demonstrated by the large number of applications built with them. For example, the Google Play Store contained 3.5 million Android applications in December 2017, all built on the Android framework [1]. One of the main aspects of frameworks that leads to large developer productivity gains is architectural reuse [4]. In the context of frameworks, *architectural reuse* means using the general parts of a proven framework application design in a new context. The *architecture* in architectural reuse refers to the static class structure of internal

framework classes, as well as the dynamic protocols of framework applications. Architectural reuse allows developers to re-purpose a software design that has been proven to work for a given domain to their unique application [21].

While developers receive a large benefit from architectural reuse in frameworks, developers that use frameworks still face unique challenges. Framework development differs from other forms of development due to designs based on inversion of control and the heavy use of object protocols. *Inversion of control* is the design style where the prebuilt framework controls the application's flow of data and execution. This design style is in contrast to development with libraries, where the developer's unique part of the application controls both the flow of data and execution and also uses library functions when needed. *Object protocols* are the limitations on the object's methods when the object is in different states (e.g., a file object can only use the write method when it is open). Object protocols occur in many forms of programming, but framework development is unique because frameworks heavily use object protocols.

Multiple studies have investigated aspects of the challenges of developing framework applications. Jaspan and Aldrich created an approach to specifying and enforcing the interaction constraints between frameworks and plugins (the unique aspects of an application that use the framework) [28, 29]. Fairbanks et al. captured common code patterns in framework with design fragments [19]. And Ko et al. categorized six learning barriers for new users of frameworks [36]. While these research efforts made progress on reducing the challenges of framework programming, multiple challenges currently still exist. A recent (August 14th, 2018) query on StackOverflow, a popular question and answer site for developers, shows that 7 of the top 24 tagged questions were on frameworks (shown in Figure 1) [2]. These framework categories had over three million questions.

To investigate the current challenges of framework development, my co-investigators and I studied the challenges of debugging framework applications. I found that repairs with framework imposed state constraints, repairs that involved understanding when values and method calls were valid in the framework application, were particularly challenging for developers. Developers struggled with these repairs, even when the failure location was provided [12].

These results motivate the need for an automated technique for repairing state-based constraints. I propose an automated repair technique for framework state constraints that exploits how framework state constraints limit the number of repair options. I will evaluate the proposed technique on 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, and thus the repair technique should be evaluated on its accuracy. *Generality* means that the technique can apply to both state constraints in the same framework and to state constraints in other frameworks. A state constraint repair technique should be able to apply to more state constraints than those that were used to build the technique, so the repair technique should also be evaluated on its generality.

Figure 1: The most popular tagged questions on StackOverflow on August 14th, 2018. The tags surrounded by red boxes are tags for questions about frameworks.

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.*

A key component of the automated repair technique, referred to as FrameFix, is a tool to detect framework state constraint violations. The state constraint detection tool consists of two subcomponents:

1. a language for specifying framework state constraints
2. a hybrid (static/dynamic) analysis to identify and localize violations in framework applications

The first subcomponent, a specification language for framework constraints, will allow the designers of a framework to specify the state constraints in ways that all users of FrameFix, the proposed automated repair approach, will benefit. This language will either be a language created for the proposed projects or an extension to a previous framework specification language. Currently, frameworks impose state constraints on applications in such a way that violations of these constraints are often caught only when the application crashes. Framework designers, or someone else that has knowledge of

important framework state, could specify the framework constraints once and have the constraints automatically checked for all developers that use the framework, improving the experience of multiple developers. The other subcomponent, the hybrid analysis, will allow the tool to check state-based faults that can be identified before running the application or through the execution of the application.

With the help of the state constraint detection tool, the repair technique will use insights from prior work, such as template-based repair [33] and the heuristic generate-and-evaluate design [64], to automatically repair state faults in framework applications. The automated repair approach will use the state constraint tool to both localize the fault for the automated repair process and to verify when the fault has been removed. Once the fault location is known, FrameFix will generate multiple possible fixes by automatically applying pre-created repair templates to these faults. The repair technique will evaluate the possible fixes against and set of test cases and the fault identified by the state constraint tool. The repair technique will repeatedly generate possible solutions until a repair that meets the evaluation criteria has been created.

I will evaluate both the accuracy and generality of the automated repair process. I will evaluate the technique's accuracy with a set of applications with framework state constraints. I will also evaluate the fault localization and the repair technique individually, to determine possible future improvements. Finally, I will analyze the technique's possible fixes of framework state constraints in other frameworks to evaluate the generality of the approach.

For this thesis, I will draw knowledge gained from my previous investigations into a genetic programming planner for self-adaptive systems, which proposed plans in a similar approach to how automated repair techniques propose possible fixes [34]. I will also use static analysis and dynamic analysis experienced from previous studies to build the directive violation detection tool [10, 11].

The proposed contributions of this thesis are:,

- A categorization of the functional effects of framework directive violations in the Android operating system (completed) [12]
- An analysis of the challenges that developers encounter when debugging framework directive violations through a human study (completed) [12]
- A tool that automatically identifies and localizes state-based directive violations in an application
- An automatic repair technique that repairs state violations in frameworks

The rest of this proposal is organized as follows. Section 3 first presents background and a motivating example. Then, I discuss the previously completed investigation into framework debugging challenges in Section 4, which provides motivation for the focus on framework state constraints and the proposed automated repair technique. Next I present a high-level overview of the proposed automated repair technique 5. After I explain the high-level overview of the proposed repair approach, I elaborate on the two proposed tools in the thesis:

1. the tool for automatically detecting framework state constraints (Section 6)
2. the tool for automatically repairing framework state constraint violations (Section 7)

I end the proposal with a discussion of possible risks in Section 8, the proposed thesis schedule in Section 9, and conclude the proposal in Section 10.

3 Background

In this section, I first discuss the topics and terminology used in this proposal (Section 3.1). Second, I provide an example that illustrates the principle problem addressed by this proposal (Section 3.2).

3.1 Relevant and Related Terminology

Frameworks provide a set of interfaces and classes that reduce the cost to achieve a general goal [37]. Application developers can use frameworks to achieve a specific task by writing a *plugin*, an extension of the framework to achieve a task. Developers create plugins to achieve specific goals through extending a defined framework interface. In this proposal, I will use the term *application* as a general term for a program. This program can be created with a framework or without one. I will use the term *plugin* to specify an extension to a framework that produces a complete application when attached to the framework. The framework typically calls plugin code through *inversion of control*, a design in which the core framework code, not the plugin-specific code, controls the data and execution flow of a plugin [32]. Frameworks usually achieve inversion of control through extending abstract methods.

Object protocols, the ordering constraints on object methods calls [6], are important when developing with frameworks. One typical example of an object protocol is the file object protocol, where a developer must open a file before closing the file. Prior work has demonstrated how object protocols are fundamental to programming with a framework [4].

Object protocols can be described as a graph of *typestates*, object states in which only a subset of the object's methods are allowed [59]. Using the file example, a file object's typestates would include the opened state and the closed state, and file reads could occur in the opened state but not the closed state. Object protocols, such as the file object protocol, occur in code that does not use a framework. However, frameworks rely on object protocols and change the typestates of objects in internal framework code [4], and thus, plugin developers must be aware of the possible object states when the framework calls their plugin.

Session Types are a specification approach to sequenced interactions [25]. Prior research has demonstrated how session types can be applied to the interactions of multiple actors at once [26]. 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 [18]. Session types would have limited applicability to the

proposed tools because the proposed tools are focused on actions that are allowed or disallowed in certain states and not the communication sequences of different components.

Directives are unexpected specifications for how to use a class or method correctly [16]. An example, with the necessary context, is explained in Section 3.2. Prior work has proposed classification schemes for directives. One classification scheme was based on the abnormal aspect specified in the directive (e.g., the calling restrictions, method limitations, or side-effects) [16] while the other focused on the segment of code covered by the directive (e.g., line, method, or object) [48]. Another investigation found that developers were more likely to successfully debug applications with directive violations when developers were presented the directives important to the problem's context [17]. Other researchers have investigated certain directive categories: directives specifying how to extend objects to implement the framework [9] and parameter usage constraints [68]. The tools in this proposal use directives as a way to identify framework state constraints.

Debugging is defined as the process of identifying and correcting the cause of a software failure [67]. Prior work in debugging found that locating the failure, referred to as fault localization [13], was the main cost of the debugging process [61]. More recent investigations have found that developers use scent finding when locating the failure [38] and the ability to easily answer dataflow questions can significantly reduce debugging time [35]. 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 [49].

Automated Program Repair is an area of research that focuses on removing identified software failures through proposed patches that are generated without human intervention. Many repair techniques use generate-and-validate to produce repairs, typified by approaches like GenProg [64, 41]. Approaches that use generate-and-validate create possible repair options and then evaluate those possible repairs on a set of program specifications, such as test cases. 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 test cases to to determine when the problem to repair and when the problem is fixed. The techniques use the passing test cases to determine if possible fixes retain the required functionality of the application. One notable family of generate-and-validate techniques is *heuristic repair techniques*, which repeatedly generate possible fixes until a valid fix is found. The typical heuristic repair process starts by locating a possible fault and then generating one or more possible fixes. The techniques then evaluate if the possible fixes are valid fixes. If a possible fix produces the required target functionality, then the possible fix is considered a successful repair. If none of the possible fixes produce the required target functionality then the techniques repeats the generate and evaluate process. Some other examples include AE [65], RSRepair [55], and SPR [44].

In contrast to the heuristic repair approach, the other major family of generate-and-validate techniques are semantic-based repair techniques, such as Angelix [47], SemFix [52], DirectFix [46], Qlose [15] and S3 [40]. *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

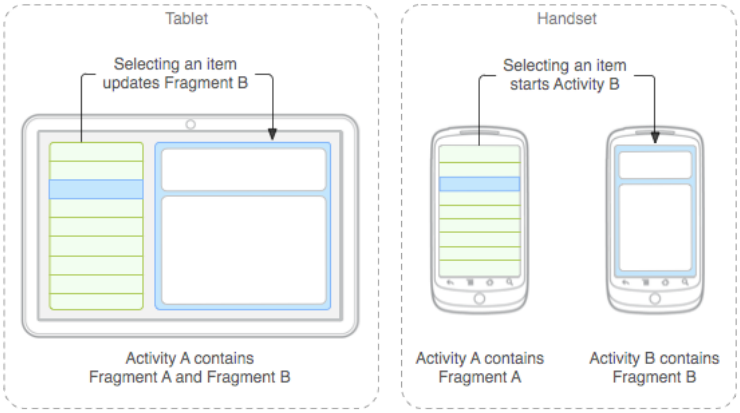


Figure 2: An example of the `Fragment` class taken from the Android developer documentation. This diagram demonstrates how the `Fragment` class is used in an `Activity`.

outputs from the provided inputs. While these techniques have shown promise, they are currently limited to fixing integer and boolean expression, and thus could only be used to repair a small set of state-based framework violations.

3.2 Motivating Example

An important part of using a framework is understanding how to use a framework correctly. Framework developers document the guidelines for how to use the framework API (Application Programming Interface) with directives. These directives can cover a wide range of guidelines, including state-based specifications.

One example of a state-based directive is a directive from the `Fragment` class in Android, a framework for developing mobile applications. The Android `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 2, which demonstrates how Fragments are a subcomponent of the larger `Activity`, which is the class that controls the lifecycle of an Android application. The documentation for the `Fragment` class states that `setArguments` can only be called on a `Fragment` before the `Fragment` is initialized. Developers can find this constraint challenging to follow, since developers may place a call to `setArguments` in a method that the framework could call after the `Fragment` has been initialized (this confusion could be caused by inversion of control, since developers would probably have a better understanding of the initialization state of the `Fragment` if the developer explicitly controlled all calls to the method in the application).

One example of developer difficulty with this directive is shown in a StackOverflow question[1], where a `Fragment` is incorrectly updated by calling `setArguments`. In this question, the application developer is trying to adjust the user interface of a `Fragment` based on the user's selection from a list. A quick summary of the code from the question is posted in Figure 3. The question also mentions that the error message reads *"Fragment already*

[1]https://stackoverflow.com/questions/19999172/fragment-already-active-when-trying-to-setarguments

```
1  @Override
2  public void onListItemClick(ListView l, View v, int position, long id) {
3    //finds a previously initialized DetailFragment
4    DetailFragment detailFragment = (DetailFragment)getFragementManager().findFragmentById(
         detailFragmentID);
5    //adjust the values in the bundle object based on user selection
6    ... //elided for simplicity
7    detailFragment.setArguments(bundle); //illegal setArguments call
8    detailFragment.setUpLayout(); //update the UI
```

Figure 3: A condensed summary of the code example from the StackOverflow question that misused the `setArguments` method.

active". In this question, the question asker is confused about the methods allowed in `Fragment` states. The question asker does not realize that the `setArguments` method is only allowed before the `Fragment` has already been initialized. Even more important is that the question asker does not realize what the alternatives are in the framework to update the desired `Fragment`, such as how to get the `Bundle` of the currently initialized `Fragment`, how to directly change the values of the `Fragment`, or how to move the `setArguments` method to a location in the code where the error would be removed while still retaining functionality.

4 Challenges of debugging directive violations

My goal with this thesis is to improve the framework development process. While there is previous research on the topic, the research does not provide enough insight on the possible areas of improvement in the framework development process. Prior work has found that developers have difficulty with framework constraints — developers ask questions on framework constraints in question and answer boards (and are willing to wait multiple hours for a response) [30] and developers miss important directives, even for small sections of code [17]. One of these works demonstrated that directive knowledge is helpful for debugging [17]. Another paper has created a taxonomy of the Android framework problems found in open source commits [20]. However, the related work provided limited help when understanding the problem that developers face while debugging framework applications, since those work investigated different scenarios or different sources of qualitative data. Thus, I and other co-investigators performed a human-focused study into the process that developers follow to debug framework application issues.

In this section, I first present the methodology for the human study into debugging framework application scenarios (Section 4.1). I then present a summary of the results from the human study in Section 4.2. Finally, I compare the study to related work in Section 4.3. A further explanation of the topics covered in this section can be found in Coker et al. [12].

4.1 Debugging Study Methodology

While there are multiple possible approaches to an exploratory study on frameworks, I decided to focus on controlled framework debugging tasks. This approach allowed me

to investigate the process that multiple developers took to debug the same problems, from which I could identify interesting patterns in developers' debugging processes. The trade-off for this approach is that if the scenarios do not accurately represent the situations that developers encounter when debugging framework applications, then the study may lead to conclusions that do not apply to real development situations. To make the scenarios as realistic as possible, I created framework debugging scenarios that focused on fixing violations of framework documentation requirements and that were based on real framework application bugs taken from StackOverflow when possible.

To conduct the human study into the challenges of debugging framework misuses, I first selected two frameworks:

1. Android

2. the Robotic Operating System (ROS), a robotic framework.

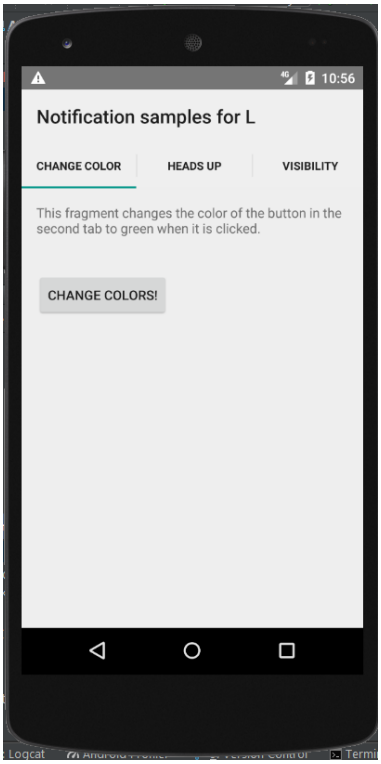
I selected the Android framework because Android contained many instances object protocols and is popular on StackOverflow. I selected the ROS framework because of the possible differences caused by the different architectural approach of ROS applications: ROS applications are organized as a collection of nodes that communicate in an event-driven architecture while Android applications are designed around the lifecycle callbacks of major components, such as the `Activity` class and the `Fragment` class. Both frameworks also had an active local user base so I could collect enough participants for the study.

The next step was to design the tasks that participants would perform. I started the task creation process by extracting a set of 45 Android `Fragment` directives and 28 ROS directives from official documentation sources which I could then use to narrow down to a smaller set of tasks for the study. For the Android study, I then looked at questions from StackOverflow that covered the directives interest and created tasks that mimicked the code in the questions. I tried to take a similar approach for ROS, but I was unable to find questions of interest. Thus, I created scenarios that violated the ROS directives.

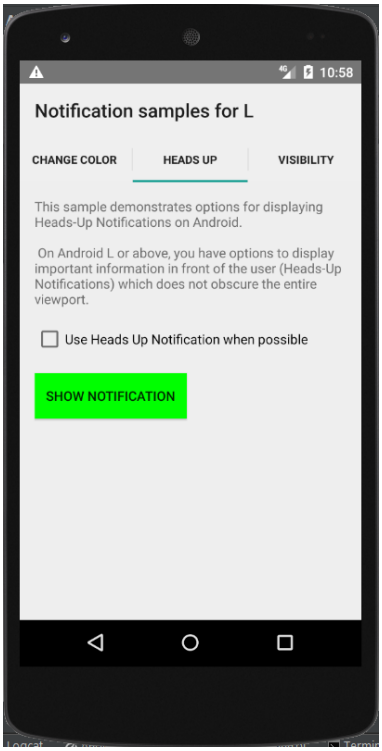
Once I created the scenarios, I recruited a convenience sample (a non-random collection of participants drawn from a close or easy-to-contact subset of the full population of interest) of 15 Android participants and 12 ROS participants to perform 7 Android tasks and 3 ROS tasks.

An example of a task is shown in Figure 4. When the user presses the `CHANGE COLORS!` button, the color of the `SHOW NOTIFICATION` button is set through a call to `setArguments`. When the `HEADS UP` tab is initialized after the button is pressed, this approach works. However, when the user has initialized the `HEADS UP` tab, by opening the tab, and then navigates back to the `CHANGE COLOR` tab and presses the `CHANGE COLORS!` button, the application violates the directive discussed in Section 3.2 and crashes. Participants were asked to fix the application so the crash is removed and the button successfully changes colors in that case.

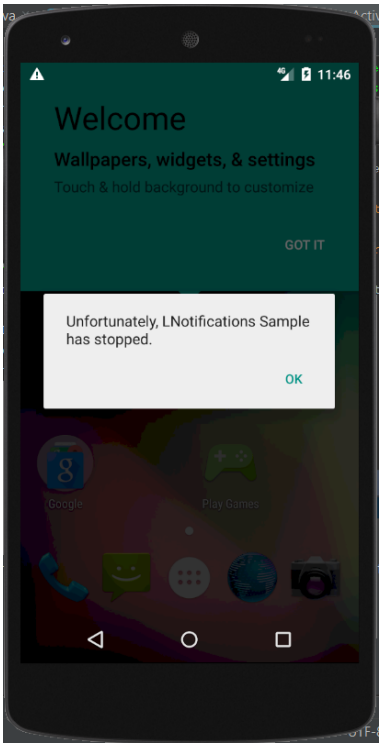
I assigned participants to the tasks so that multiple participants performed each task. The participants were told the general problem with the application and instructed to execute the application to perform the directive violation. During this process, I did not inform the participants of the specific directive violations in the application. I then instructed participants to perform think-aloud debugging, an approach where participants



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



(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.



(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 4: 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.

explain their thought process so the researcher can gain more insight into their problems [50]. During the sessions, I recorded the participants as they debugged the directive violations. Afterwards, I analyzed the recording to gain insight into the process of debugging framework directive violations.

4.2 Results from the human study on directive violations

From the human study, I found that participants encountered multiple difficulties with object protocols in the frameworks and also with the inversion of control organization of frameworks. For example, while creating the fix to the directive violation, three partici-

pants tried to access user input before the user could input a value, causing the program to eventually display the wrong value. While completely reducing these challenges would be difficult, a tool that could notify developers of incorrect state-based interactions would be a start in the right direction.

In the Android investigation, the tasks that participants spent the longest on were tasks that involved object states. For example, the task where participants had to display the user selected input was one of the most difficult tasks, partially because participants had to not only determine how to access the two different input components uniquely, but also how to access the components after the user had entered a value. This leads to the hypothesis that state-based directive violations are particularly difficult to debug, and a tool that could provide automated support would be helpful. Another exciting findings from the study is that contrary to the results presented by Vessey [61], I found that developers spent a significant amount of time determining how to fix a directive violation when developers were told the directive violation in the error message. Taken together, these results lead to the hypothesis that finding the fault may not be the most difficult aspect when fixing framework directives. Instead, producing the correct repair may be the most challenging step for humans when debugging framework directive violations. Thus, an automated repair tool would be helpful in providing possible fixes for state-based directive violation.

4.3 Related Work

The closest study to the human study discussed in this section is the investigation into the challenges faced by new framework users when developing framework plugins [36]. While the human study discussed in this section focused on debugging challenges and more experienced developers, some of the problems encountered in both studies were the same, such as the difficulty of understanding how the internal framework code affected the code segment of interest and difficulties using debuggers. Jaspan and Aldrich found that developers had difficulties with object protocols by examining the questions asked on Spring and ASP.NET forums [30]. Jaspan and Aldrich also noted that developers were willing to wait multiple hours for an answer to their questions about the frameworks, demonstrating the degree of difficulties that framework application developers face. While this study and my study both used developer forums, my study investigated developers in person, leading to a better understanding of the challenges that developers face during the framework application debugging process. Another closely related work was done by Dekel and Herbsleb [17]. This study demonstrated that knowledge of relevant directives increased the number of developers that finish library and framework debugging tasks in a set time limit. This study focused on evaluating a tool that notified developers of relevant directives, and not the challenges that developers faced while finding or applying directives.

A couple of papers have addressed the topic of *Android Application Debugging*. Tan et al. [60] investigated automatic repair of crashing Android applications, a subset of the violation consequences presented in the human study of this section. Fan et al. [20] collected Android exception traces and classified the exceptions based on the type of error (e.g., Lifecycle error, UI Update Error, Framework Constraint Error). Instead of only fo-

cusing on exceptions, I investigated the way that framework applications handle directive violations and found a more diverse set of functional effects.

5 Brief overview of proposed tools

The human study into debugging directive violations found that participants had particular difficulty with state-based directives, similar to the directive shown in Section 3.2 [12]. Participants' difficulties were caused by multiple factors, e.g., a framework changing the state of objects in internal framework code, or the interaction of multiple objects in different states [12]. While some directive violations could be automatically enforced by simple static analysis, state-based directives are more difficult to check automatically, since they require a specification of the important states of an object and the object's state transitions. Developers would benefit from an approach that automatically repairs state-based directive violations.

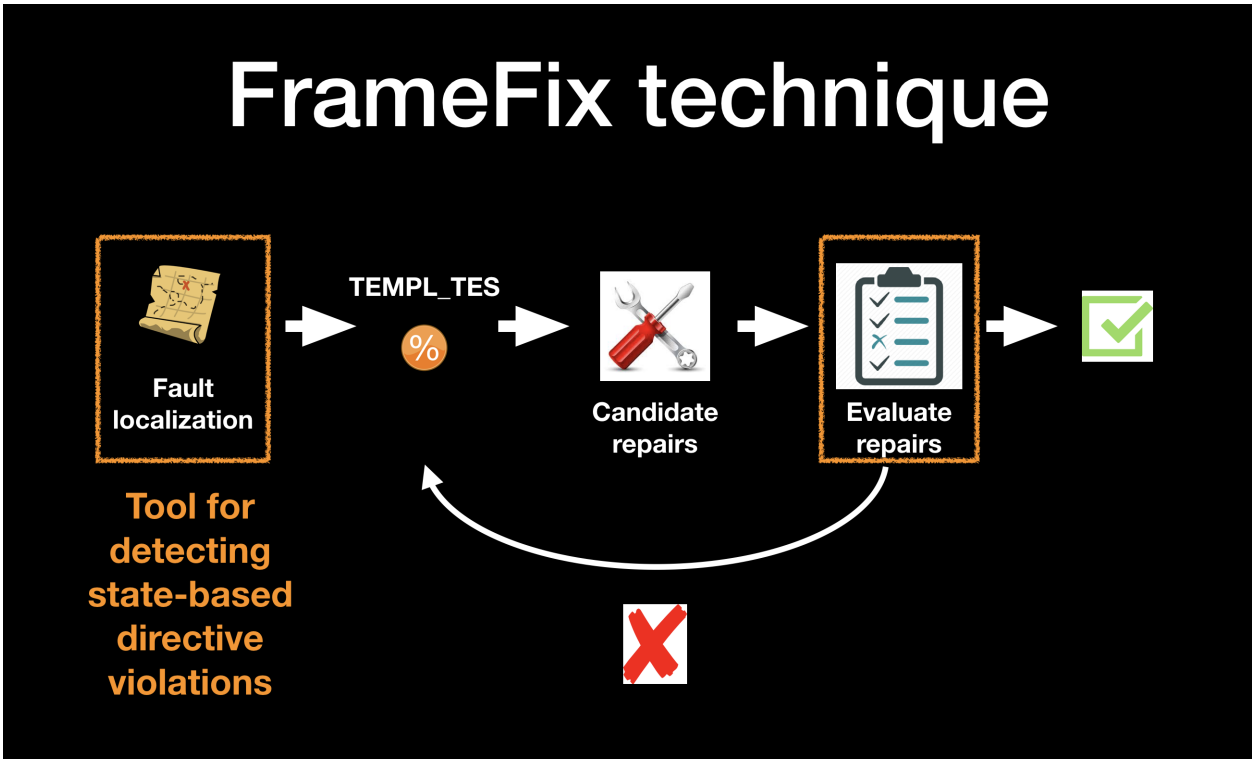


Figure 5: The process diagram of the FrameFix repair technique. The state-based directive violation detection tool is used to both identify the fault location and to evaluate possible repairs. The technique starts by identifying the fault location. The fault information is used with a template repair approach (or a statistical model repair approach) to produce a set of possible repairs. The repairs are then evaluated against a set of test cases and the violation detection tool to see if the repair is valid. The repair approach will continue to produce and evaluate repairs until it produces a valid fix. This technique is further described in Section 7.1.

This motivates FrameFix, an automated repair approach to state based framework directive violations. The components of the proposed automated repair tool, FrameFix, are shown and explained in Figure 5. One key aspect of this diagram is how the repair process will use the state based framework directive violation tool, designated by the orange boxes. The state based directive violation tool is needed in both the identification of the fault to repair and to verify that the problem has been removed.

6 Automatic detection of state-based directive violations

The state-based detection tool must be able to determine if the application violations the directive to be useful in the automated repair process. This requirement consists of two main subcomponents. The first is that *the tool must allow developers to specify directives in a way that can be unambiguously checked by the tool.* A goal of this specification would be that directives could be specified once and then checked in different contexts. This single specification goal would reduce the requirements for the tool to be useful, increasing the chance it will be accepted by the development community. In this case, one knowledgeable user, such as the framework designer, could encode the rules so that they are automatically checked for all users of the tool. This approach could also be used to assist new developers of the framework with extra checks. The second subcomponent is that *the tool must be able to determine if the application violates the directive.* Since some directives are difficult to determine statically, and may be undecidable, this means the tool will need to perform both static and dynamic directive violation checks.

Section 6.1 discusses the current results on this project. Section 6.2 discusses the proposed evaluation for this research thrust. Then, Section 6.3 compares the proposed tool and evaluation to related work.

6.1 Current results for the directive violation identification tool

At the time of this proposal, I am investigating and understanding state-based directives, to inform the creation of the directive violation identification tool. I have collected a set of 16 state-based directives from the Android developer's guide and the documentation for four classes: `Activity`, `Fragment`, `Dialog`, and `Context`. I have used this sample of directives to understand the variety of state-based directives and understand the requirements of a state-based specification language for an automated checking tool. I am also investigating a set of 15 ROS bug to also support directives in ROS.

Once I have finished investigating state-based directives, the next step will be to create the static analysis. I will create a tool that implements the static analysis defined by the dataflow analysis specification in Figure 6. Dataflow analysis is a technique to track possible variable values in different locations of a method or program. Figure 6 describes the static analysis using the dataflow analysis format in *Compilers: Principles, Techniques, and Tools* [3]. This dataflow analysis will allow the tool to statically determine if method calls violate state-based directives.

For example, consider the directive *only call `getActivity()` when the `Fragment` is attached to an `Activity`.* Using a sample application from the human study [12], the application

Domain: Set of important states in the application, which can often be determined from method calls, a previous textual scan of the application's source, or are pre-defined

Lattice definition: For a single directive, $\perp = \emptyset$, each subsequent level is all combinations in $\binom{n}{k}$ where n is the level from the top of the lattice (for second level $n = 1$) and k is the number of important states. At the top of the lattice, $n = k$, which means all states are unknown (e.g. *anyState* for all objects that are important to the directive).

Direction: Forward

Transfer function: $additionalStates \bigcup currentStates - (statesEnded)$ where $additionalStates$ is the important states added by a method call, $currentStates$ is the set of previous important states, and $statesEnded$ is the set of states that end at the current method transition or are canceled by the states in $additionalStates$.

Boundary: $OUT[ENTRY] = \emptyset \vee$ the annotated input $\vee$ the input determined through control flow

Meet operator: $\bigcup$

Initialize: $OUT[othernodes] = \emptyset$

Interprocedural meet: annotations, context sensitive dataflow from previous methods, or computed from method's control flow

Figure 6: The dataflow analysis specification for our static analysis approach to checking state-based directive violations.

crashes when a `Fragment` is selected. This is due to a call to `getActivity()` that occurs before the `Fragment` is attached. However, the `Fragment` calls `getActivity()` in multiple places, all but one of them when the `Activity` is attached to the `Fragment`. Using knowledge of the control flow graph for the application and the Android `Fragment` lifecycle, it can be deduced that all but one call is certainly correct (e.g. the call occurs in lifecycle phases where the `Activity` is always attached. The only uncertain case occurs in a method that is called from the parent `Activity`.

Figure 7 shows an example application where a method that calls `getActivity()` before the `Fragment` is attached to an `Activity`. Using provided annotations from the Framework designers, the tool could deduce that all `Fragments` in the application are in a possibly

```
1  @Override
2  public void onTabSelected(ActionBar.Tab tab, FragmentTransaction ft) {
3    ft.replace(R.id.container, fragment);
4    if(fragment instanceof  OtherMetadataFragment){
5      //the next line calls getActivity before the Fragment is attached
6      ((OtherMetadataFragment)fragment).displayActivityTitle();
7    }
8  }
```

Figure 7: An example call to `getActivity()` before the `Fragment` is attached to an `Activity`, violating the directive and causing the application to crash.

unattached state at the start of `onTabSelected`. The tool would start the dataflow analysis with all `Fragment`s in the possibly unattached state, except for the initial `Fragment` that is shown at the start of the application, since it was attached in a previous method that is not shown. However, for this application, it can be deduced that an `OtherMetadataFragment` is not the starting `Fragment` and thus is possibly unattached.

The lattice for this dataflow analysis would have a $\top$ of *anyState*, which represents when the state cannot be determined for a `Fragment`, which is equal to the possibly unattached state mentioned above. The states of a `Fragment` would consist of *attachedToActivity* and *notAttachedToActivity*. Certain methods, can produce transfer states, while other methods are only safe in specific states. Because `displayActivityTitle` always calls `getActivity`, the `displayActivityTitle` is only safe when the `fragment` is in the *attachedToActivity* state. Due to the fact that an `OtherMetadataFragment` is in the *anyState* initially, dataflow analysis can be used to deduce that the `Fragment` state will not change when the `displayActivityTitle` method is called since the `replace` method is not annotated as a method that changes the attached state of a `Fragment` (the annotations are provided by the framework designer in this example). Thus, `displayActivityTitle` will be called when the `Fragment` is in a possibly uninitialized state, and the tool will display an error to the application developer. If the tool performed a previous scan of the methods that could attach an `Activity` to a `Fragment`, the tool would find that the application could only attach an `Activity` to a `Fragment` in two locations: where the starting `Fragment` is attached (not shown), and at the end of this method — `onTabSelected`. In that case, the tool could determine that the `OtherMetadataFragment` has to be attached at the end of the method, so the call to `displayActivityTitle` would violate the directive at least once.

Unfortunately, the static analysis works well only for directives that can be enforced syntactically or whose relevant control flow can be determined unambiguously. Dynamic analysis will be used to check the enforcement of directives which do not fit those two categories, such as restricting method calls when when a database is closed. The dynamic analysis technique will execute applications, check if directives are violated during the application execution, and notify developers if a violation occurs. A formal specification of our dynamic analysis is written in Figure 8, and is based on the operational specifications defined by Leinhard et al. [43].

I have currently designed a first draft of the specification language (Figure 9) and am now creating a tool that parses this language into the information required by an automated checking tool. The tool is currently able to parse the specification language. I am also working on creating static and dynamic analyses for specific directives.

6.2 Proposed evaluation of tool for automatic detection of state-based directive violations

The state-based directive violations tool will be evaluated on four criteria:

1. recall
2. precision
3. generality

Variables in specification

m - a method

c - a class

$\mathcal{D}$ - a function `<class, method, environment>` $\rightarrow$ `boolean`; a check if the class and method are in the set of `<class, method>` tuples that trigger a directive check and if the directive is violated in the current environment

e - the current environment that contains the app configuration information, outgoing connections to databases, etc.

`notify` - a method that notifies tool users of a directive violation if one occurs

s - the statements in a method

`MethodLookup` - a function `<c, m>` $\rightarrow$ `<s,c'>`; a conversion from the class and method of the object to the statements in the object and the class for which the method is defined

Σ - the set of `<class address, parameter address, method, stack frame>` tuples

`Stack Frame` - $\{\sigma \in \Sigma \bigcup \emptyset\}$

`Address` - $\{i \in \mathbb{N} | i \; is \; an \; address \; of \; valid \; memory\}$

`Heap` - a function `<i>` $\rightarrow$ `c`; a mapping of memory locations to classes defined at those locations

$\mathcal{H} \in$ `Heap`

Method call semantics

Given:

$c = \mathcal{H}(i)$ //determine the class of the object in the method call statement

$(s, c') = \texttt{MethodLookup}(c, m)$ //convert a method and class to the expressions in the method and the class where the method is defined

$d = \mathcal{D}(c', m, e)$ //check if the method has a directive check and if the directive is violated in the current environment; using c' because the directives are specified in the documentation for classes that are often parent classes of the current object, d is true when there is a directive violation

$\sigma' = (i, i', m, \sigma)$ //create a simplified stack frame with the new method and arguments

$\sigma' * s, \mathcal{H} \rightarrow \sigma' * i'', \mathcal{H}'$ //evaluating the method statement produces the return result i''

Result:

`if d then notify`; $\sigma * i.m(i'), \mathcal{H} \rightarrow \sigma * i'', \mathcal{H}'$

Figure 8: The dynamic analysis specification of the directive violation tool using operational semantics. This tool checks for directive violations when a program calls methods mentioned in state-based directives. If a directive violation is found, the analysis displays a notification.

⟨*directive-specification*⟩ ::= ⟨*method-call-specification*⟩
| ⟨*field-specification*⟩

⟨*method-call-specification*⟩ ::= ⟨*method-call*⟩ [NOT] (ALLOW_WHEN | REQUIRE) ⟨*reference-identifier-list*⟩ [NOT] IN ⟨*state-list*⟩

⟨*method-call*⟩ ::= ⟨*class*⟩.⟨*method-signature*⟩

⟨*reference-identifier-list*⟩ ::= ⟨*reference*⟩
| ⟨*reference*⟩, ⟨*reference-identifier-list*⟩

⟨*reference*⟩ ::= @⟨*ref-term*⟩

⟨*ref-term*⟩ ::= *this*
| *super*
| ⟨*ref-term*⟩.⟨*class-variable*⟩

⟨*state-list*⟩ ::= ⟨*state*⟩
| ⟨*state*⟩, ⟨*state-list*⟩

⟨*field-specification*⟩ ::= ⟨*object-field*⟩ MUST_BE_SET_IN ⟨*state-list*⟩ [TO ⟨*value*⟩]

Figure 9: The Backus-Naur Form formalism for state-based directive specifications..

4. suitability for use in automated repair approach.

The tool should be evaluated on recall and precision because that is an established way to evaluate a fault localization technique [39]. Generality and suitability are important because the eventual goal of this tool is to be used in a state-based automated repair approach. Both are required to create a repair technique that can fix state-based directive violations in framework applications. The tool will be designed to run on the full source code of a framework application, so all experiments will test full applications.

I will evaluate the precision and recall of the tool by creating a dataset of state-based directive violations from a single framework that contains enough examples of a applications with state-based directive violations to evaluate the tool. This framework will likely be the Android framework, due to the prevalence of state-based directive violations in the Android component lifecycles and the prior work I have done with the Android frameworks. The ROS framework or another framework that contains enough examples for evaluation are current alternative options. I will create the dataset using bug collected from open-source sites, such as GitHub, along with applications created through the combination of questions from developer forums, such as StackOverflow, and sample applications. When collecting applications from GitHub and questions from StackOverflow, I will select the most popular applications and questions from the sites, using the search options available on each site, that contain state-based directive violations (with some filtering to ensure that the dataset does not consist of only a few directive viola-

tions repeated in different applications). The dataset will consist of at least 100 instances of state-based directive violations that cover at least 15 different state-based directives in a single framework. The dataset will contain applications that I take directly from GitHub as well as applications that are based on StackOverflow questions. The bugs collected from GitHub will allow evaluating the technique on large applications but may not contain enough scenarios of interest, since it is currently difficult to search open-source projects for bugs that are not easily found with a simple code pattern, and developers may not commit these directive violations, since the directive violations may only occur in the development period between commits. To include the application issues that developers encounter but do not commit, I will also create applications based on StackOverflow questions and sample applications taken from official framework samples and tutorials, if possible, or popular applications that use the framework on GitHub if I am unable to find official applications. I will create applications in the dataset by modifying the sample application to contain the scenario discussed in the StackOverflow question. I will also add 10 applications to this dataset that contain correct state-based directive implementations. The 10 applications will contain at least 5 unique state-based directive instances. I will use this dataset of 110 applications to evaluate recall and precision.

The first metric for evaluating the automated detection tool is recall. Recall is defined as

$$recall = \frac{\#\_of\_caught\_violations}{\#\_of\_caught\_violations + \#\_of\_uncaught\_violations}$$

This metric is important because the tool should be able to identify a state-based directive problem in the application when one exists. For the static analysis evaluation, the goal will be to catch at least 80% of the violations (similar to other publications in the area [66, 24, 53]) in the 100 faulty applications at the location where the directive violation occurs. The numerical specification of this goal, as well as other evaluation numbers in the proposal, is designed to provide context on a reasonable standard that has been demonstrated in similar prior work, and not to bind me or the committee to these evaluation numbers. For the dynamic analysis evaluation, the goal will be to ensure that each directive statement is tested at least once, and any directive violations are identified at the correct location. By testing the recall of the tool, I will ensure that the method works correctly for the directives it was designed for and ensure that the tool can catch real world bugs.

The second metric for evaluating the automated detection tool is precision. Precision is defined as

$$precision = \frac{\#\_of\_correctly\_identified\_violations}{\#\_of\_correctly\_identified\_violations + \#\_of\_incorrectly\_identified\_violations}$$

Precision is an important metric for this tool because false error reports have been shown to cause developers to lose trust in tools[57]. I will use 10 applications with correct instances of state-based directives , as well as any correct instances of state-based directives in the 100 faulty applications, to determine if the tool reports an error when an error does not exist. The goal will be to incorrectly identify one or fewer incorrect directive violations static and dynamic analysis (marking a directive violation as undecidable will not

count as an incorrectly identification), meeting the false positive rates at Google [57] and Coverity [7].

The third metric for evaluating the automated detection tool is generality. The goal of the proposed technique is to create a technique that could apply to a significant portion of state-based directives and across multiple frameworks. Thus the technique should be evaluated on generality.

To determine how well technique applies across directives in the same framework, I will evaluate the tool on how well the automatic approach to state-based directive violations generalizes on ten new directive violations that I will collect after creating the tool. This approach will test the generality of the approach on a single framework. The goal will be to automatically catch 80% of a set of ten collected directives. I will also further investigate the directives that are not supported, which will inform possible tool improvements.

To evaluate how well the technique will apply to other frameworks, I will test the tool on ten state-based directives in another framework. If I do not have to rebuild the static and dynamic infrastructure for a new framework (i.e., The analysis tools can be reused across frameworks in the same language) then the tests will be conducted in an automated manner. However, I am currently considering building the initial framework infrastructure for Android, and due to differences in the Android Runtime from the Java Virtual Machine, the infrastructure I am building may be limited to Android. In that case, I will manually inspect ten state-based directives in another framework and determine whether or not our technique could support those directives if the appropriate framework tooling infrastructure was built. This framework will likely be the ROS framework, due to prior work with the framework, but may be any framework with clearly documented state-based directives and easy to collect examples of state-based directive violations. For this experiment, I will collect ten state-based directives from other frameworks (the first ten collected — not chosen on any other criteria) and verify that the tool could catch 50% of the directives, assuming the tool contained the necessary framework-specific changes. This evaluation would provide confidence that the technique could generalize to other frameworks while also providing information on possible tool improvements.

The final goal of the evaluation is to determine that the tool will be useful for automated repair. While the usefulness of the tool cannot be completely evaluated without an accompanying automated repair approach, I can evaluate if the tool will be able to be used in a generate-and-validate automated repair approach [64]. Evaluating if the tool is suitable for automated repair involves evaluating if the tool is able to localize the fault [54], which is covered by the previous three evaluations, but also requires evaluating the tool on the time required to run the analysis. In the proposed approach, the technique will run the state-based directive violation detection tool multiple times, and thus, the state-based directive violation detection tool must be able to identify the fault in a short amount of time. To ensure that an automated approach can finish in a reasonable amount of time, I will test that a prototype implementation of this tool can run 1000 or more times in day. Due to the limitations of running real applications and the start up time of emulators, this may not always be feasible for the dynamic analysis. If the tool meets these minimum requirements, then the detection tool is likely suitable for use in an automated repair approach. A full evaluation of the usefulness of the automated detection tool in an

automated repair context will be left until the supporting automated repair tool is built, further discussed in Section 7.

6.3 Related Work

There are framework specification languages that have been created in prior work, each with its own limitations for the case proposed here. SCL is a specification language and analysis approach for checking method-level syntax patterns [27]. This approach is limited to interprocedural analysis, and would require significant adjustments to adapt the language to the tool proposed here. Declarative event patterns is an approach to dynamically check framework function call sequences specified in AspectJ [62]. This approach does not perform static specification checks and is limited to specifications that can be enforced through traces (i.e., the directive *"The* `Fragment` *must call* `setHasOptionsMenu(true)` *if the application overrides* `onCreateOptionsMenu` *cannot be enforced with this specification approach"*). The specification language created by Jaspan [28] is the closest language that will be useful for the tool. This language is also limited to specifications that can be enforced through traces, but could be extended to cover other specifications. I currently lean towards using a new language, since it avoids the specification overhead of a broader use case, but I am considering using the specification language created by Jaspan as an alternative.

Other papers have created static [58], dynamic [14], and hybrid [8] typestate checkers. However, they have not been applied directly to the framework context, where frameworks may also change the state of objects of interest, increasing the complexity of the analysis. Another similar work that enforces object protocols through static and dynamic analysis in APIs was not applied to frameworks [23].

There is some prior work on typestates in frameworks, such as DroidStar, which automatically finds a state machine for Android classes using user-specified callins and callbacks [56]. While not focused on frameworks, Nanda et al. automatically collect typestate specifications [51]. Other researchers have experimented with incorporating typestate as a first-class component of a programming language [5]. The language approach to typestate is similar to the proposed type specification in this proposal, but requires the overhead of specifying the type states of all objects, and not just the most important ones. The approach also requires developers to document how all methods change the typestate of objects, instead of the methods that cause changes to the typestates of objects.

7 Automated repair of state-based directive violations

Once it is possible to automatically identify state-based directive violations, the next step is to automatically repair these violations. Section 7.1 discusses the proposed repair approach, FrameFix, and 7.2 discusses the proposed evaluation of the technique. Section 7.3 then discusses other related automated repair approaches.

7.1 FrameFix design

The main insight of FrameFix is the state information gained from the violation detection tool (Section 6) and the limited set of possible repairs of state-based directives will significantly reduce the difficulty of automatically repairing state-based directive violations. Prior research has shown that the time required to produce a repair is correlated with the number of possible fault locations, which means that the fault locations identified by the violation detection tool will help make finding the fix feasible [22]. The limited number of possible repairs will further increase the change that the tool will produce a successful repair.

Once the fault has been determined by the violation detection tool, FrameFix will use a heuristic generate-and-validate strategy [64] to repair the fault, which consists of generating multiple possible repairs and then validating each repair against a set of specifications. FrameFix will generate possible repairs using a template based approach [33]. These templates will either be created manually by the researcher, using knowledge of possible fixes for different directive, or through an automated approach that looks for patterns in fixes to past directive violations. This template strategy is likely to work because directives often have a limited number of ways to correctly implement the desired functionality and the object protocols of software entities in the directives further limit the valid number of method call, and thus possible repairs, in each state. Also, prior research has created templates for framework application development and demonstrated that these templates are useful in the context of frameworks [19]. If the template based approach does not work, then I will use a model-based repair approach [45]. This approach uses information on common code elements (such as the number of `if` statements) to guide proposed automatic repairs.

FrameFix will use the application specification information from two sources to determine if the fix is successful:

1. test cases that accompany the buggy applications, which will be used to determine the required functionality of the application
2. the state-based directive violation tool, which will identify if the directive violation has been removed.

A proposed repair will be considered a successful repair if it is able to pass the specifications from both sources. If the template based repair approach does not work, then the tool could use a machine learning based repair approach [45] or a genetic programming based approach [41].

7.2 FrameFix evaluation

FrameFix will be evaluated on both the accuracy of the proposed repairs, as well as the generality of the repair technique. Accuracy, in this context, will be defined as changing the application in a way that retains specified functionality while removing the detected undesirable behavior. Generality will be defined as applying the technique to other state-based directives in the current framework and in other frameworks.

It is important that the repair technique is accurate, because the technique should fix the problem without otherwise changing functionality. If the proposed repairs did not fix the directive violation or arbitrarily changed the functionality of the application, then developers would not likely trust our tool. The technique should also generalize to other applications with state-based framework directive faults, both within the framework and in a new framework. If the tool could only work for the applications that were considered when creating the tool, then the technique would only address a small set of directives in a specific framework. Thus, the tool should be both accurate and general.

To evaluate the accuracy of FrameFix, I plan to first create a dataset of test programs, reusing as many applications collected in Section 6.2 as possible (which means I will perform both evaluations on the same framework). The dataset will contain 20 programs that violate at least 10 different state-based directives and contain a test suite, which I will consider a specification of the required functionality of the application. If I need to collect more applications than those previously collected for the evaluation in Section 6.2, then I will use the same criteria to collect applications (specified in Section 6.1) with the additional criteria that the application has an associated test suite.

The goal for the tool is to to fix 30% of the state-based programs in the dataset, where a fix means that the proposed repair removes the state-based directive violation without removing the required functionality encoded in the application's test cases. While the accuracy of techniques cannot be compared when using different datasets, previous work has achieved a fix rate between about 20–50% on their chosen datasets, which indicates that 30% will be a reasonable goal [45, 40, 41, 42].

I will evaluate the generality of the technique with six new applications with state-based directive violations. If the state-based directive detection tool infrastructure supports running the analysis on programs in a different framework, then these six applications will contain state-based directive violations in another framework. If the state-based directive violation tool infrastructure does not support another framework, then I will evaluate the tool using six applications from the current framework that contain different state-based directive violations than were in the original dataset. I will then manually evaluate if the technique could apply to six applications from anther framework, which will either be the ROS framework or another framework with clearly documented state-based directives and easy to collect examples of state-based directive violations. In all cases, the goal will be to fix, or possibly fix, two of the six applications, maintaining the 30% fix rate discussed previously.

As a final evaluation, I will compare the new technique to GenProg, a well-known approach for general automatic program repair. This will demonstrate that FrameFix is better suited for the state-based framework repairs that the tool is designed to fix. This evaluation could also provide insight on possible future areas of improvement.

7.3 Related Work

Some examples of heuristic generate-and-validate techniques that I will use as inspiration in this proposal are PAR [33], short for Pattern-based Automatic Program Repair, and Prophet [45]. PAR is a template based repair technique, which applies a group of repair templates to possible fault locations in the code. The authors created the templates by

manually defining repair templates (these templates are based on patterns in past repairs and domain knowledge). Prophet automatically learns the common elements in a group of past fixes (such as the number of if statements) and uses those common elements to guide possible repairs. These approaches apply to fixing state-based directive, since this approaches specialize in fixing problems with limited repair options, and state-based repair has limited valid repairs in each state.

Recent work on the Android framework has categorized a large number of Android exceptions and extracted common repair patterns for exceptions [20]. Some of these repair patterns are likely to be useful when repairing state-based directive violations, such as adding extra condition checks, while other state-based directive issues are not likely to be covered by these repair patterns, such as fixing directive violations that do not cause exceptions. Tan et al. [60] have also found common repair patterns for faults in Android applications that lead to crashes, by manually creating repair templates from a set of fixes for previously crashing Android applications, and may serve as a source of inspiration for some of the framework repairs in this thesis. FrameFix is designed to be framework independent and thus will handle a wider set of framework issues than those handled in the Android repair technique. Another related approach involves fixing programs that are specified with contracts [63]. A part of this approach involves dynamically building an object behavior model (represented by object states) for passing tests cases and using that to guide repairs. This approach does not support a static analysis approach to checking directives and is limited in the states that are represented by the accessor functions of the class (e.g., `getFoo()` function calls in Java).

8 Risks

As with any proposed project, there are possible risks that could cause changes to the proposed thesis. In this section, I will cover a few possible risks and discuss possible mitigation strategies.

Risk: Lack of test cases with applications
The automated repair evaluation requires a specification of both the violated directive and the functionality that the application is required to maintain after the proposed fix to the directive violation. The required functionality of an application is often specified through test cases that demonstrate the behavior of the application in successful conditions. It may be difficult to collect enough applications with an associated test suite to validate the repair technique. In this case, I could create a test suite for the applications of interest that encodes the required functionality. If the bias produced by a single person creating the automated repair technique and creating the test suite cause experts in the field to lose confidence in the results of this evaluation, I could recruit another person to create the test suite for those applications.

Risk: Running applications may require specific configurations that are not available
One risk of using real applications is that the applications may have specific requirements, such as specific hardware, that are unavailable to someone who did not create the project

or may be difficult to obtain. This risk can be mitigated by collecting more applications with state-based directive violations, since if more examples are collected, there is a better chance that I will be able to collect enough applications that I can use with the resources available. Another mitigation strategy would be to create applications that remove the configuration-specific aspect of the applications while retaining the directive violations of interest. I would create these applications in a similar approach to the applications created based on questions off StackOverflow — alter a base application to contain the problem of interest.

Risk: Identifying directive violations or repairing directive violations is more difficult than expected
The currently proposed evaluations depends on the ability of the proposed tools to identify directive violations and repair directive violations. If I am unable to identify and repair a significant number of directive violations, then significant changes to the evaluation could be required. The first mitigation for this problem arises from the fact that there are multiple strategies for identifying problems through static analysis and to generate possible application repairs. If the currently proposed techniques do not work, then I will try other options, which will be selected when I have a better understanding of the reasons the proposed approach is failing. If these other techniques do not work, then I will analyze why this problem is more challenging than expected and use that analysis to draw conclusions on the limitations of this problem space.

9 Schedule

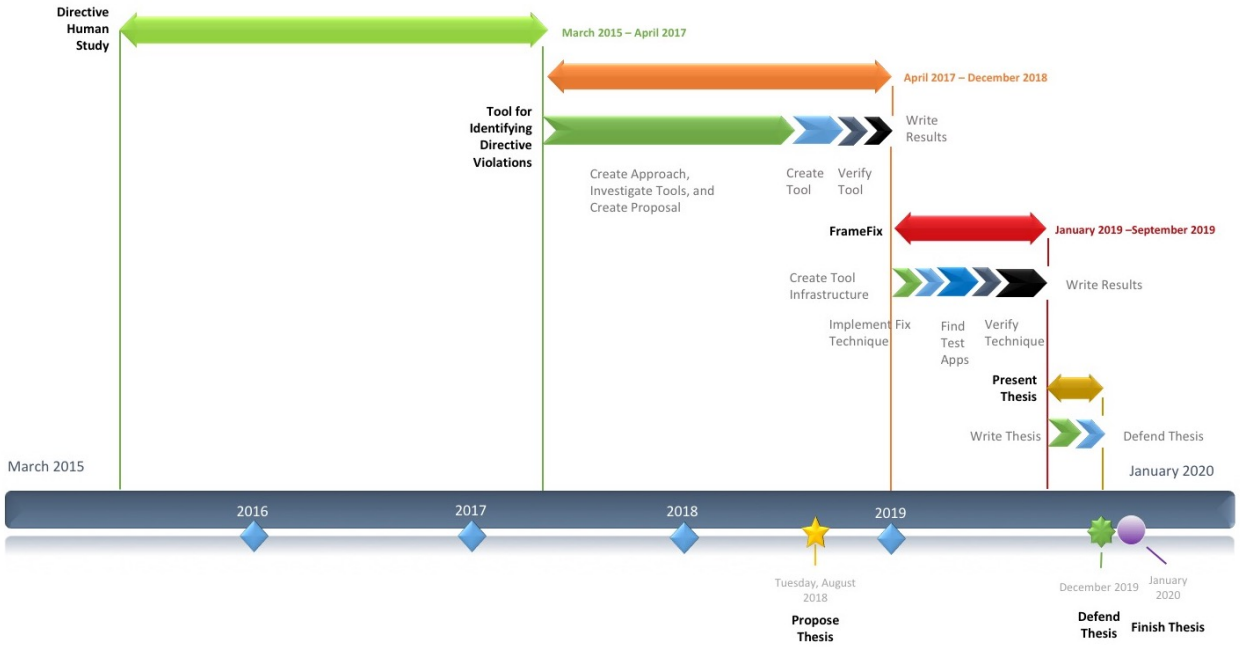


Figure 10: Proposed timeline for the thesis.

Figure 10 shows the proposed schedule for this thesis with an expected graduation date of January 2020. My current goal is to finish the tool for automatically detecting directives by December 2018 and FrameFix by September 2019. At the time of writing this proposal, the directive human study and part of the tools for identifying directive violations work has been completed.

The estimate of the two uncompleted projects, as well as the thesis presentation process, is broken down into the different components to provide insight on the time required for each step of the process. For example, *create tool infrastructure* and *find test apps* are each month-long tasks in the automatically repair direction violations project. This plan also provides three months at the end of the plan for writing and presenting the thesis, as well as any extra work that may arise during the dissertation process.

During this timeline, the work from these projects will be submitted to software engineering conferences and journal such as the International Conference on Software Engineering (ICSE), Foundations of Software Engineering (FSE), and Transactions on Software Engineering (TSE).

10 Conclusion

I have demonstrated that developers face challenges with object states in frameworks and I have proposed a feasible tool to address these problems. This thesis proposes three main research projects:

1. a human study investigation into the challenges of debugging directives
2. a tool that can automatically detect application errors due to not following state-based directives
3. FrameFix: a tool to automatically repair state-based directives in applications.

The proposed techniques will provide developers with a new approach to reducing the challenges of framework programming. Another benefit of these techniques is that they will allow developers to incrementally add and adjust the enforced state specifications, enabling the tools to adapt to changes in the framework and to problems that framework application developers face. If extra work is put into making these tools available to developers after the initial proof of concept, these tools would address a significant problem when developing with frameworks and increase developer productivity.

References

[1] `https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/`, Accessed Jan. 4th, 2018.

[2] `stackoverflow.com`, Accessed Nov. 20th, 2017.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321486811.

[4] 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.

[5] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 1015–1022, 2009.

[6] 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.

[7] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[8] Eric Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *International Conference on Software Engineering - Volume 1*, ICSE '10, pages 5–14, 2010.

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

[10] Zack Coker and Munawar Hafiz. Program transformations to fix c integers. In *International Conference on Software Engineering*, ICSE '13, pages 792–801, 2013.

[11] Zack Coker, Michael Maass, Tianyuan Ding, Claire Le Goues, and Joshua Sunshine. Evaluating the flexibility of the java sandbox. In *Annual Computer Security Applications Conference*, ACSAC 2015, pages 1–10, 2015.

[12] Zack Coker, David Gray Widder, Claire Le Goues, Christopher Bogart, and Joshua Sunshine. Debugging famework applications: Benefits and challenges. *Computing Research Repository*, https://arxiv.org/abs/1801.05366, 2017.

[13] 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.

[14] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. Generating fixes from object behavior anomalies. In *Automated Software Engineering*, ASE '09, pages 550–554, 2009.

[15] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program repair with quantitative objectives. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 383–401. Springer International Publishing, 2016.

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

[17] 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.

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

[19] 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.

[20] 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.

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

[22] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *Genetic and Evolutionary Computation*, GECCO '09, pages 947–954, 2009.

[23] Madhu Gopinathan and Sriram K. Rajamani. Enforcing object protocols by combining static and runtime analysis. In *Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 245–260, 2008.

[24] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. Machine-learning-guided selectively unsound static analysis. In *International Conference on Software Engineering*, ICSE '17, pages 519–529, 2017.

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

[26] 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.

[27] 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.

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

[29] Ciera Jaspan and Jonathan Aldrich. Checking framework interactions with relationships. In *European Conference on Object-Oriented Programming*, ECOOP '09, pages 27–51, 2009.

[30] 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.

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

[32] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.

[33] 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.

[34] Cody Kinneer, Zack Coker, Jiacheng Wang, David Garlan, and Claire Le Goues. Managing uncertainty in self-adaptive systems with plan reuse and stochastic search. In *Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '18, pages 40–50, 2018.

[35] Andrew 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.

[36] Andrew 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.

[37] Craig Larman. *Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Upper Saddle River, N.J. : Prentice Hall Professional Technical Reference, Upper Saddle River, New Jersey, USA, 3rd ed edition, 2004. ISBN 0131489062.

[38] 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.

[39] Tien-Duy B. Le and David Lo. Will fault localization work for these failures? an automated approach to predict effectiveness of fault localization tools. In *International Conference on Software Maintenance*, ICSM '13, pages 310–319, 2013.

[40] 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.

[41] 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.

[42] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

[43] A. Lienhard, T. Girba, and O. Nierstrasz. Specifying dynamic analyses by extending language semantics. *IEEE Transactions on Software Engineering*, 38(3):694–706, May 2012.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] Mangala Gowri Nanda, Christian Grothoff, and Satish Chandra. Deriving object typestates in the presence of inter-object references. In *Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 77–96, 2005.

[52] 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.

[53] Frolin S. Ocariza, Jr., Karthik Pattabiraman, and Ali Mesbah. Detecting inconsistencies in javascript mvc applications. In *International Conference on Software Engineering - Volume 1*, ICSE '15, pages 325–335, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL `http://dl.acm.org/citation.cfm?id=2818754.2818796`.

[54] Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 191–201, 2013.

[55] 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.

[56] Arjun Radhakrishna, Nicholas Lewchenko, Shawn Meier, Sergio Mover, Kirshna Chaianya Sripada, Damien Zufferey, Bor-Yuh Evan Chang, and Černỳ. Droidstar: Callback typestates for android classes. In *International Conference on Software Engineering*, ICSE '18, pages 1160–1170, 2018.

[57] 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.

[58] Paulo Salem. Practical programming, validation and verification with finite-state machines: A library and its industrial application. In *International Conference on Software Engineering Companion*, ICSE '16, pages 51–60, 2016.

[59] 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.

[60] 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.

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

[62] Robert J. Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In *Foundations of Software Engineering*, FSE 04, pages 159–169, 2004.

[63] 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.

[64] 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.

[65] 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.

[66] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. Spain: Security patch analysis for binaries towards understanding the pain and pills. In *International Conference on Software Engineering*, ICSE '17, pages 462–472, 2017.

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

[68] 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.