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 of architectural reuse: the process of applying previously proven designs to new applications in a given domain. Unfortunately, for frameworks to support architectural reuse, frameworks must impose constraints on developers applications. The constraints derive from the assumptions that frameworks make to interact with a diverse set of applications. 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 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 are a language for specifying constraints along with a hybrid (static/dynamic) analysis to identify and localize violations in framework clients. I will then evaluate the accuracy of the generated repairs on sample programs with framework state violations. The ultimate goal of the tool 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 [18]. 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 frameworks. 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 lead to large developer productivity gains is architectural reuse [4]. Architectural reuse allows developers to re-purpose a software design that has been proven to work for a given domain to their unique application [13].

While developers receive a large benefit from architectural reuse in frameworks, developers still face challenges with frameworks. Multiple research projects have tried to
address aspects of these challenges. Jaspan and Aldrich investigated the interaction constraints between frameworks and plugins (the unique aspects of an application that use the framework) [15, 16]. Fairbanks et al. captured patterns of framework interactions in design fragments [11]. And Ko et al. categorized six learning barriers for new uses of frameworks [22]. While these research efforts made progress on reducing the challenges of framework programming, multiple challenges still exist. On StackOverflow, a popular question and answer site for developers on August 14th, 2018, seven 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 state-based constraint violations were particularly challenging for developers and that developers were not easily able to fix framework constraint violations, even when the failure location was provided [8]. These results led to the following thesis statement:

**Thesis statement:** Framework application developers encounter distinct challenges when reusing the architecture provided by frameworks. One important manifestation of these challenges is framework state constraints. I proposed to automatically repair applications that violate framework state constraints with a technique that exploits the similarities between framework state constraints.
The proposed technique consists of two key insights: 1) a language for specifying framework state constraints and 2) a hybrid (static/dynamic) analysis to identify and localize violations in framework applications. The first insight, 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 will benefit. Currently, frameworks impose constraints on applications in a way that violations of these constraints are not always caught at compile time, and often only when the application crashes. The proposed specification language will catch these constraint violations earlier. The specification language has the additional benefit that a constraint specified once can be used by multiple developers. This means the work of one person, such as the designer of the framework, would improve the developer experience of all users of the framework. The other insight, the hybrid analysis, will solve one of the leading challenges of automated program repair: identifying the fault location. Once the fault location is known, FrameFix will use knowledge of the violated constraint to propose a valid fix of the program.

I will evaluate both the accuracy and generality of the technique. I will evaluate the technique’s accuracy by scrutinizing the proposed repairs for a set of applications with constraint violations. During this evaluation, I also evaluate the fault localization and the repair technique individually, to determine possible future improvements. Finally, I’ll manually analyze the technique’s possible fixes of constraint violations in other frameworks, assuming the appropriate framework specific infrastructure existed. This manual analysis will provide insight on how well the technique generalizes to new frameworks.

The proposed contributions are:

- A categorization of how frameworks present directive violations to framework application developers in the Android operating system (completed) [8]
- An analysis of the challenges that developers encounter when debugging framework directive violations through a human study (completed) [8]
- A notation for specifying framework directives in a software application
- A tool that automatically notifies developers of state constraint 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 presents background and related work. I discuss the previously completed investigation into directive debugging challenges in Section 4. I then present the two proposed tools of the thesis: 1) the directive violation detection tool (Section 5) and 2) the automated repair tool for directive violations (Section 6). I then present the proposed schedule for the thesis in Section 7 and conclude the proposal in Section 8.

3 Background

In this section, I first discuss the topics required to understand the proposal and how the proposal is similar to but different from previous research (Section 3.1). Second, I provide
an example that illustrates the principle problem of this proposal (Section 3.2).

3.1 Related Work

Frameworks provide a set of interfaces and classes that reduce the cost to achieve a general goal [23]. 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 by adding new functionally within a framework template. In this paper, 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 a program that was written using a 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 [19]. Frameworks usually achieve inversion of control through extending abstract methods. Frameworks commonly require plugins to conform to a specified plugin structure. Frameworks also commonly use declarative artifacts, non-source code files that contain configuration information [17]. In this proposal, I will refer to the process of using the framework provided architecture and domain-specific libraries as framework-based reuse. This is in contrast to library-based reuse, which only uses libraries without a provided architecture to save time and effort when developing applications.

Object protocols, the ordering constraints on calls to an object’s methods [6], are important to framework-based reuse. 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 described how objects are important to implement the architectural reuse provided by most frameworks [4]. Another investigation found open-source programs and libraries use object protocols, which suggest that object protocols are commonly used in programming [6].

Object protocols can be described as a graph of typestates, object states in which only a portion of the object’s methods are allowed [38]. 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 frameworks. However, frameworks rely on object protocols and change the typestates of objects in internal framework code [4]. 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 [37]. While not focused on frameworks, Nanda et al. automatically collect typestate specifications [34].

Directives are specifications for how to use a class or method correctly. Two works have proposed a classification scheme for directives, with one focusing on the abnormal aspect specified in the directive (e.g., the calling restrictions, method limitations, or side-effects) [9] while the other focused on the code unit covered by the directive (e.g., line, method, or object) [31]. Another work performed a human study that found that developers were more likely to successfully debug applications with directive defects when developers were presented the directives important to the problem’s context [10]. Other works on directives have focused on mining certain directive categories: directives
specifying how to extend objects to implement the framework [7] and parameter usage constraints [44].

Debugging is defined as the process of identifying and correcting the cause of a software failure [43]. Prior work in debugging has found that locating the failure was the main cost of the debugging process [40]. More recent investigations have found that developers use scent finding when locating the failure [24] and the ability to easily answer dataflow questions can significantly reduce debugging time [21]. Another work has also found that developers encounter design decisions during the debugging process, such as when the fix incorrect data passed between multiple components [32].

A couple of papers has addressed the topic of Android Application Debugging. Tan et al. [39] investigated automatic repair of crashing Android applications, a subset of the violation consequences presented in the human study of Section 4. Fan et al. [12] collected Android exception traces and classified the exceptions based on the type of error (e.g. Lifecycle error, UI Update Error, Framework Constraint Error). This paper focuses on framework constraint errors and identifies how violations of these constraints are presented to developers. Instead of only focused on exceptions, I investigated the way that framework applications handle constraint violations and found a more diverse categorization.

Automated Program Repair is the approach to remove identified failures through proposed patches that are generated without human intervention. One of the notable family of techniques in automated repair is generate-and-validate, typified by approaches like GenProg [42]. The generate-and-validate family of techniques first generates possible fixes by locating a possible fault and generating one or more possible fixes. Second, these techniques use information from other sources, such as a set of test cases that contain at least one failing test, to determine the requirements for a correct fix. The techniques use the initial failing tests to determine when the problem is fixed. The passing test cases constrain what modifications to the program are possible since the passing test cases must still pass after the change. If a modification is able to pass the specifications, such as the test case set, then the techniques considers the modification a successful repair. If none of the modification fit all the specification requirements then the techniques generated more modifications.

Some examples of generate-and-validate techniques that will be used as inspiration in this proposal are template based repair [20] and learning models of correct code [28]. Template based repair is a technique were multiple common fix templates are created and then applied to a failing program to fix the bug. Learning models of correct code consists of learning a vector of elements in successful patches, which contains the characteristics of the patches, such as if the number of local variables used in the patch. The learning models technique then uses those models to repair other problems by creating a search space of possible repairs and to select the most likely patches from the search space for the program. These approaches apply to fixing state-based directive issues because they focus on the problem of selecting the correct repair out of a set of options, instead of generating a possible repair from scratch. State-based repair is likely to have a limited number of known repairs for an identified problem, and thus the challenge becomes selecting the right repair for the current situation. Recent work on the Android framework has categorized a large number of Android exceptions and extracted common repair patterns [12].
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.

Although these repair patterns are too broad to be directly used in a state-based repair approach, I may be able to use similar repair patterns in an automated framework repair approach, such as adding extra condition checks. Tan et al. [39] have also found common repair patterns for Android issues and may serve as a source of inspiration for some of the framework repairs in this thesis.

Another family of techniques for automated repair is semantic-based program repair, such as Angelix [30], SemFix [35], DirectFix [29], and S3 [25]. Semantic-based program repair uses dynamic semantic analysis, commonly symbolic execution, and a set of test cases to infer desired program behavior. While these techniques have shown promise, they are currently limited to fixing integer and boolean expression.

3.2 Motivating Example

Developers encounter multiple challenges when developing a framework application, such as how to use the states of various objects in a framework application correctly. Framework designers document these state guidelines in directives.

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, setArguments can only be called on a Fragment before the Fragment is initialized. Developers can find this constraint challenging to follow, due to inversion of control, since an application developer may have difficulty determining if the Fragment has been initialized.

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

---

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

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`.

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 [17] and developers miss important directives, even for small sections of code [10]. One of these works demonstrated that directive knowledge is helpful for debugging [10]. However, based on the limited related work, I thought I could get novel insights into the framework development process though a human study. I and other co-investigators decided that watching developers debug framework application issues in a controlled setting would provide the most insight on uncovering possible areas of improvement. Work that was published after this investigation was completed but not yet published has categorized Android framework problems from open-source commits into a taxonomy, providing further possible areas of focus and confirming that developers have difficulty with Android lifecycles and framework constraints [12].

To determine if alternative approaches would be more effective or if developers encounter other problems than awareness of the violated directive, it is important to understand the process that developers take when debugging directive violation problems. Since the process of debugging directive violations was not documented in any source, a human study into the process of debugging directive violations was the logical next step.

In this section, I first discuss the methodology of the preliminary investigation that informed the scenarios in the human study (Section 4.1). Second, I present the methodology for the human study into debugging framework application scenarios (Section 4.2). Finally, I present a summary of the results from the investigations in Section 4.3.
4.1 Methodology for preliminary study on how frameworks present framework misuse

While prior work has investigated similar problems, such as the learning challenges of new framework users [22], my coauthors and I hypothesized that an exploratory study into the challenges of framework debugging would provide new insights into these challenges. While there are multiple possible approaches to an exploratory study, 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 the debugging scenarios may not be realistic and thus cause me to draw conclusions that do not apply to a real development situation. In attempt to make the scenarios as realistic as possible, I created framework debugging scenarios that focused on fixing violations of framework documentation requirement and that were based on real framework application bugs taken from StackOverflow when possible. But first, I needed to narrow down which violations to focus on, and I wanted the tasks to cover important variations in the framework application debugging process. Thus, before performing the human study, I performed a preliminary investigation into the ways that frameworks present directive violations to developers.

The first important design decision for the preliminary investigation methodology was the frameworks to use in the study. For the preliminary study, I only focused on the Android framework because Android contained many instances object protocols and is popular on StackOverflow. I only investigated the Android framework in the preliminary investigation because I only needed enough variations to inform interesting scenario differences.

The second decision was how to sample the API misuses. I decided to use testable documentation statements, known as directives, that specified how to use the framework, since they provided specific and testable instances and were relatively easy to find. To avoid the significant overhead of a thorough and manual investigation of a large code base, I had to decide between automatically searching the whole framework for a type of violation or a thorough investigation into an important part of the framework (i.e., an automated pattern match search that will find instances throughout the code base but miss patterns that were not provided in the original search or a manual investigation into a small section that catches all instances). I decided to fully investigate a small section of the framework, the Android Fragment class, which is one of the key classes to an Android application and a class that was mentioned in many questions on StackOverflow.

After choosing the study’s focus, I manually extracted directives from three documentation sources. I then manually created scenarios that violated each directive and then categorized the effects that that violated directive had on the application. A further elaboration on the methodology for the preliminary study is contained in Coker et al. [8].

4.2 Debugging Study Methodology

After I understood the different ways that frameworks could present directive violations to developers 4.1, I began to create the human study scenarios to investigate the chal-
challenges that developers face when fixing directive violations in plugins. To conduct the human study into the challenges of debugging framework misuses, I first selected two frameworks: 1) Android, due to the reasons described in the preliminary study methodology, and 2) the Robotic Operating System (ROS), a robotic framework. 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. ROS 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. For the Android study, I looked at questions from StackOverflow that covered the framework mismatches of 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 framework API specifications.

Once I created the scenarios, I recruited a convenience sample of participants to perform the tasks. I assigned participants to the task so that multiple participants perform 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 explain their thought process so I could gain more insight into their problems [33]. 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. A further elaboration on the methodology of the experiment is contained in Coker et al. [8].

4.3 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 participants 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.

One of the most exciting findings from the study is that contrary to the results presented by Vessey [40], I found that developers spent a significant amount of time determining how to fix a directive violation after the developers knew what directive was violated. In the scenarios where participants were immediately notified of the failing location, participants were faster but still spent a comparable amount of time to the other tasks where participants had to both locate and fix the error.

In the Android investigation, the tasks that participants spent the longest on were tasks that involved object states. 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 very helpful. These results also 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 helpful in providing possible fixes for state-based directive violation. A further elaboration of the study’s results can be found in Coker et al. [8].

5 Automatic detection of state-based directive violations

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 [8]. Participants’ difficulties were caused by multiple reasons, e.g., a framework changing the state of objects in internal framework code, or the interaction of multiple objects in different states [8]. While some directive violations could be automatically enforced by simple static analysis, state-based directives are more difficult to automatically detect, since they require a specification of the important states of an object and the object’s state transitions. To alleviate these issues, a tool is needed that can identify state-based directive violations and overcome the challenges associated with automatically determining the current state.

This tool must address some important challenges to be useful. 1) The tool must be able to specify directives in a way that can be unambiguously checked by the tool. A goal of this specification would be to make it where directives only had to be specified once and could be checked in many different contexts. This single specification goal would reduce the requirements for the tool to be useful, increasing the chance that 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. 2) The tool must be able to determine if the directive will be violated. Since some directives are difficult to determine statically, this means the tool will need to perform both static and dynamic directive violation checks.

Section 5.1 discusses the proposed evaluation for this tool. Section 5.2 discusses the current results on this project.

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

The goal of the evaluation of the automated detection of state-based directive violations tool will be to determine the accuracy and generality of the tool. This tool will be designed to run on the full source code of a framework application, so all experiments will test full applications. First, I will ensure that the tool works on a sample set of directive violations. The tool will be tested on 100 instances of state-based directive violations taken from GitHub or created based on StackOverflow questions. The goal will be to catch at least 90% of these violations. By testing the tool on these applications, 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.
One important goal of the study is to determine the generality of the technique, to make sure that the technique applies to the directives in the framework and not only the example directives that were used to create the tool. Thus, I will also 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.

A limitation of this experiment is that state-based directives in one framework are likely to be more similar than state-based directives in another framework. To reduce this limitation, 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 (there is likely to be a large overhead for building the static and dynamic analysis infrastructure for each framework, so automatically testing multiple frameworks may be infeasible). For this experiment, I will collect ten state-based directives from other frameworks and manually verify that the tool could catch 50% of the found 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.

Another limitation of this approach is that the usefulness of the tool for automated repair was not evaluated. The tool is likely to be useful to automated repair techniques, since determining the fault is a difficult challenge for automated repair and the ability to automatically determine the fault location significantly reduces the challenge of the repair strategy [36, 5]. However, a full evaluation of this claim will be left until the supporting automated repair tool is built, further discussed in Section 6.

5.2 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 understanding the state-based directives, I will create a tool that implements the static analysis defined by the dataflow specification in Figure 4 based on the dataflow specifications in Compilers: Principles, Techniques, and Tools [3]. This dataflow analysis will allow the tool to 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 [8], the application 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
**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, $\bot = \emptyset$, 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:** $additionalStates \cup currentStates - (statesEnded)$ 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:** $OUT[ENTRY] = \emptyset \lor$ the annotated input $\lor$ the input determined through control flow

**Meet operator:** $\cup$

**Initialize:** $OUT[othernodes] = \emptyset$

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

---

Figure 4: The dataflow analysis specification for our static analysis approach to checking state-based directive violations.

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 5 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 unattached state at the start of onTabSelected. The tool would start the dataflow analysis with all Fragments 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

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

Figure 5: An example call to $getActivity()$ before the Fragment is attached to an Activity, violating the directive and causing the application to crash.
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 uninitialzed state, and the tool will display an error to the application developer.

Unfortunately, the static analysis cannot catch all cases, such the current state of a database in the environment. This state can be determined through an analysis of the application at runtime. To catch directive violations that are difficult to check statically, I propose a dynamic analysis technique that will check for directive violations and notify developers if a violation occurs. A formal specification of our dynamic analysis is written in Figure 6 and is based on the operational specifications defined by Leinhard et al. [27].

I have currently designed a first draft of the specification language 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 in Figure 7. I am also working on creating static and dynamic analyses for specific directives.

### 6 Automated repair of state-based directive violations

An automated repair tool will significantly reduce the time required to fix state-based directive repairs, since prior work has shown that automatically generated repairs reduce the time required to verify the correctness of a proposed fix [41].

This automated repair tool, FrameFix, will use the state information gained from the violation detection tool (Section 5) and the limited set of possible repairs for state-based directives to automatically repair state-based directives. The evidence provided by the detection tool will be used to pinpoint the fault location. Knowledge of the fault location will greatly aid the tool when creating a proposed fix. 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 reduction in possible fault locations will make finding the fix feasible [14]. After the tool determines the fault location, it must repair the fault. The results from our human study into debugging directive defects (Section 4) led me to hypothesize
Variables in specification

- m - a method
- c - a class
- \(D\) - a function \(<\text{class}, \text{method}, \text{environment}> \rightarrow \text{boolean}\); a check if the class and method are in the set of \(<\text{class}, \text{method}>\) 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
- \(\Sigma\) - the set of \(<\text{class address}, \text{parameter address}, \text{method}, \text{stack frame}>\) tuples

Stack Frame - \{\(\sigma \in \Sigma \cup \emptyset\}\}

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

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

Method call semantics

Given:

- \(c = \mathcal{H}(i)\) \hspace{1cm} //determine the class of the object in the method call statement
- \((s, c') = \text{MethodLookup}(c, m)\) \hspace{1cm} //convert a method and class to the expressions in the method and the class where the method is defined
- \(d = \mathcal{D}(c', m, e)\) \hspace{1cm} //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
- \(\sigma' = (i, i', m, \sigma)\) \hspace{1cm} //create a simplified stack frame with the new method and arguments
- \(\sigma' \ast s\), \(\mathcal{H} \rightarrow \sigma' \ast i'', \mathcal{H}'\) \hspace{1cm} //evaluating the method statement produces the return result \(i''\)

Result:

\(\text{if } d \text{ then notify; } \sigma \ast i.m(i'), \mathcal{H} \rightarrow \sigma \ast i'', \mathcal{H}'\)

Figure 6: 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.
that the location of different object states in the code limits the possible repair locations. The repair options are further limited by the possible repairs for a directive violation — since the repair technique has knowledge of the directive that was violated, the technique can propose alternatives that can be used to achieve a similar goal.

The violation detection tool will provide a way to verify that a fix has been achieved when the violation has been repaired. To determine if the repair is acceptable, I will use other sources of information, such as tests that come with the application or a successful patch to the problem in source control, to verify that the change does not break the functionality of the application.

### 6.1 Automated repair of state-based directives

FrameFix will produce possible patches for faulty state-based directive code. I will evaluate FrameFix on both the accuracy of the proposed repairs, as well as the generality of the repair technique. The repairs should be accurate, because they should fix the problem without otherwise changing functionality. The technique should also generalize to other applications with state-based framework directive faults. These two aspects are important because without either one, the technique would have limited applicability. 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. If the tool could only work for the applications that were considered when creating the tool, then the technique
would be very limited. Thus, the tool should be both accurate and general.

To test the accuracy, I will evaluate the technique on a set of collected state-based directive violations, as well as non-state based directives to test the tool’s ability to determine that a fix does not apply. I will also evaluate the accuracy of components of the automated repair technique, to understand the benefits gained from the novel contributions and possible areas of improvement.

To evaluate the automated repair of state-based directives, I plan to first create a benchmark of test programs, reusing as many applications collection in 5.1 as possible. Creating the benchmark is necessary because no openly available bug dataset, of which I am aware of, currently contains a high enough prevalence of state-based framework directive misuses to thoroughly evaluate our technique. The benchmark dataset will contain 20 programs that violate state-based directives, 4 program that contain violations of directives that are not state-based, and 4 bugs in applications that are not violations of directives. I will use the variety of programs in the dataset to evaluate the technique on both situations that the technique should be able to fix and also situations that I did not design the technique to handle. The purpose of evaluating the tool on situations that I did not design the technique to handle is to make sure the tool handles these non-perfect situations correctly. In these non-perfect cases, the tool should determine that the repair technique was not meant for these cases and notify the tool user that the another repair strategy is required. Through this approach, I will be able to get a sense of which applications cause the tool to produce false positives (when the tool applies a fix to an incorrect situation) and false negatives (when the tool is unable to propose a necessary fix), which should inform possible areas of improvement. The goal for the tool is to fix 30% of the state-based programs in the dataset. 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 [28, 25, 26]. I will consider the fix a successful fix if the repair causes the application to pass all specification sources: the associated test cases and the state-based directive violation checker. If it is possible, I will then try to collect six more state-based directive programs and make sure that the technique applies to these new programs. This additional test will provide confidence on the generality of the proposed technique.

I will create the benchmark using bug collected from open-source sites such as GitHub along with applications created through the combination of questions from StackOverflow and sample applications. 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. To also cover the interesting cases that were not covered by the collected open-source projects, we will take questions on StackOverflow that demonstrate real problems of interest and modify sample applications to contain the scenarios presented in the questions. A benchmark consisting of applications from these two groups should allow a thorough evaluation of the technique for automatically repairing state-based directive violations.

Along with evaluating the complete automated repair technique, I also plan to evaluate the gains in the automated repair process produced by the tool that automatically identifies state-based directive violations. This tool provides an extra test or specification
that can be used in the automated repair process with two main benefits: 1) pinpointing the fault, and 2) limiting the number of repair options. I plan to evaluate both of these benefits individually. For pinpointing the fault, I plan to test how often the technique pinpoints the location that either needs to be fixed or contains the failure that informs the fix location. For limiting the number of repair options, I plan to evaluate the repair technique in two different modes: with and without information on the directive violation location. I will evaluate how many programs each approach can fix in the dataset well as the time required to achieve the fixes. These evaluations will allow us to quantify the improvements gained by this new approach to automated repair and identify the current bottleneck of the automated repair process.

7 Schedule

Figure 8: Proposed timeline for the thesis.

Figure 8 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).

8 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, and 3) FrameFix: a tool to automatically repair state-based directives in applications.

The techniques proposed in this paper will provide developers with a new approach to reducing the challenges of framework programming. Another benefit of these techniques is that they 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


