FrameFix: Automatically Repairing Statically-Detected Directive Violations in Framework Applications

Zack Coker  
School of Computer Science,  
Carnegie Mellon University  
Pittsburgh, PA  
zfc@cs.cmu.edu

Joshua Sunshine  
School of Computer Science,  
Carnegie Mellon University  
Pittsburgh, PA  
sunshine@cs.cmu.edu

Claire Le Goues  
School of Computer Science,  
Carnegie Mellon University  
Pittsburgh, PA  
clegoues@cs.cmu.edu

ABSTRACT
Software frameworks make developing applications for a specific domain easier than doing so from scratch. Unfortunately, frameworks can also place unexpected requirements on a developer’s application, which can in turn lead to application bugs in the development process. We propose an automated technique for repairing violations of state-based framework requirements, FrameFix. First, developers of a framework can encode state-based framework requirements. Then FrameFix automatically checks whether a developer’s application follows the encoded requirements. Once a violation of these requirements has been detected, FrameFix tries three different approaches — reordering method calls, moving method calls to different method definitions, and comparing the faulty method to similarly defined methods on GitHub. These repair approaches are mainly based on the key aspects of how frameworks interact with framework applications: object protocols and inversion of control. To demonstrate that object protocols and inversion of control can be used to aid automated repair for framework applications, we created a sample implementation of FrameFix for Android applications. We evaluated FrameFix by injecting problems into open source Android applications. We found that FrameFix was able to fix at least one of the cases for each repair type. We also used FrameFix to fix errors found in an real applications.

CCS CONCEPTS
- Software and its engineering → Object oriented frameworks; API languages; Software maintenance tools; Search-based software engineering; Software testing and debugging; Automatic programming.

KEYWORDS
Frameworks, Automated Repair, API

1 INTRODUCTION
Software frameworks make developing applications for a specific domain easier than doing so from scratch. A framework is a generic piece of software or set of libraries that developers can use to develop client applications in a domain, typically by conforming to a specific prescribed architecture. [1]. For example, the Android framework provides a reusable interface to interact with the hardware of the mobile device, allowing application developers to only code the unique parts of their application. Software frameworks are widely used, and substantially increase the productivity of professional software developers [2]. The importance of frameworks is demonstrated by the large number of framework applications. For instance, the Google Play Store contained 2.9 million applications in December 2019, all built on the Android framework [3].

While developers commonly use frameworks, they can still encounter significant challenges in using them correctly. Of the top 20 tagged question categories on Stack Overflow, 6 correspond to framework categories, containing more than 3 million questions [4]. New framework application developers encountered difficulties understanding how their changes would affect the application and selecting the right framework interface to solve their problem [5]. Developers have been shown to wait days for answers to framework-specific questions on forums [6], demonstrating both the difficulty of developing framework applications and the challenges associated with finding the information necessary to do so. Framework development also poses unique challenges to developers, such as in debugging violations of framework-specific API requirements [7]. For example, state-based issues, problems that arise from the state of objects in the framework (a common feature of framework programming), are especially challenging, because developers have to track how a framework (usually invisibly) modifies the states of objects in internal code. Overall, debugging support for framework development is still inadequate, and certain features of framework applications, like state manipulation, are particularly challenging.
Despite these challenges, we observe that frameworks also exhibit key features that can help in developing automated tooling to improve the debugging process. These key features are

1. The heavy use of object protocols - requirements that objects must be in the correct state to perform an action. Because frameworks heavily use object protocols, methods commonly fail because they are performed in the wrong state, and can be moved to a location where objects are in the correct state.

2. Due to inversion of control, the way frameworks call application to perform application specific actions, framework applications often contain the same callback method signatures. If a problem occurs in one of these callback methods, there is likely an application that implements the callback correctly on GitHub.

In this paper, we propose a novel, end-to-end static program repair technique that uses these insights to automatically repair state-based framework errors, a particularly challenging class of debugging problems for framework application developers. In this approach, someone knowledgeable with the framework (likely the framework developer) would encode a directive (8) (a method call requirement not enforced by the type system) using a specification language and provide the starting parameters of the repair process. When an application developer of the frameworks wants to check an application, the application developer would run the analysis generated from the specification language. If the analysis finds an error, then FrameFix generates possible repairs guided by the key features that differentiate framework application repair. These proposed repairs are evaluated with the analysis and application’s test cases to determine the correct fix.

We instantiate our approach in a proof-of-concept tool called FrameFix, which addresses such defects in Android applications. We evaluate FrameFix on a large subset of the F-Droid dataset [9]. We demonstrate in particular applicability, in terms of the prevalence of the types of potential defects we address, and utility in terms of the variety and type of repairs for those errors it can construct. We find that FrameFix is able to repair 4 problems in real application and 91% of automatically injected errors.

Our approach is novel in its end-to-end treatment of the problem of finding and fixing state-based framework mistakes. Both static [10, 11] and dynamic [12] checkers have been proposed to identify improper interactions between an application and its underlying framework. These techniques are useful, but evidence suggests that developers struggle to fix framework mistakes even once the problem has been identified or pointed out to them explicitly [7]. More recent work targets the repair of crashing Android applications when a user-provided event sequence can recreate the crash [13], but without the end-to-end static checking we propose that can detect problems that do not cause application crashes; our approach is also lighter-weight in its specification burden on the framework developers, and does not require framework developer and application developers to write contract specifications [14] to guide repair.

Our primary contributions are:

- An instantiation of the technique for Android applications, called FrameFix\(^1\).
- An evaluation on F-Droid, a set of 1,964 open source Android applications

We start the paper by discussing the necessary background on framework bugs in Android in Section 2. We then discuss the approach we used to create FrameFix in Section 3 and our evaluation of FrameFix in Section 4. We end the paper with a discussion of limitations (Section 5), related work (Section 6), and finish with conclusions (Section 7).

2 STATE-BASED FRAMEWORK BUGS

We begin by introducing background information on framework development (Section 2.1), as well as the types of defects we target with our repair approach (Section 2.2). We instantiate our approach for Android and evaluate it in that context, and so include specifics for the Android domain throughout.

2.1 Framework Development

Frameworks provide a set of interfaces and classes that reduce the time required to create a new program in a particular domain [15]. Developers create applications to achieve specific goals by writing plugins, or client applications, that use the framework. For example, Android applications are plugins to the Android framework. In practice, the framework typically calls the plugin code through inversion of control, a design in which the core framework code, not the application-specific code, controls the data and execution flow of an application [1]. Inversion of control is typically implemented by defining callback methods in the framework application, which the framework calls when needed. The callback methods have a specific method signature that the framework can recognize, a key insight we used to implement FrameFix. Another important aspect of frameworks is object protocols, or ordering constraints on calls to an object’s methods [16]. While object protocols appear in many applications, object protocols are especially prevalent in framework applications, where frameworks change the state of objects in internal framework code. Both inversion of control and object protocols require developers to keep track of often-invisible object states, increasing the chance of encountering a state-based problem. Finally, though not directly germane to our approach, frameworks commonly require applications to conform to a specified structure, and to contain certain declarative artifacts, non-source code files that contain configuration information [1].

**Android.** To create a proof-of-concept repair technique, we created an implementation of FrameFix for the Android framework. We implemented FrameFix for Android because it was the most popular framework on StackOverflow, and thus most likely to have a demonstrable need for state-based automated repair. We also chose the Android framework because of the state-based development issues found in a previous study [7].

We provide a brief summary of relevant Android development terms, for reference for the rest of this paper. In an Android application, the `Activity` class is the main entry point of the application. Fragments are reusable subcomponents of an `Activity`. Intents

\(^1\)https://www.dropbox.com/sh/9b7ayyp25oa248dn/AAB88KzdAJQXmp3flgft2pYgEUa/idi-0
are the objects that Android passes between applications, which facilitates the communication between different applications. The directives covered in this paper prominently feature these classes. An Android application is packaged into one or more executable files called APKs (Android Packages), which are used in the FrameFix process to evaluate if a source-code level repair succeeded.

2 State-based directive violations in Android
Our repair approach targets framework-specific mistakes, in a way that should be useful across many applications built against a given framework. Directives are unexpected or surprising API specifications for how to use a class or method correctly [8]. By focusing on framework directives, we are able to focus on documented development challenges that are not dependent on a specific framework application. We are particularly interested in addressing violations of state-based directives, since state-based problems have been shown to be difficult for developers [7].

Directives are often subtle, and expressed by the framework developers in READMEs, tutorials, comments, or other forms of documentation surrounding a framework; it is presently the responsibility of the application developers to follow them correctly. For illustration (here) and (in subsequent sections) evaluation, we collected a set of Android-specific directives that can be violated, leading to bugs in real applications. We collected a set of directives in the Android API and application creation documentation using a subset of directives from prior work [7]. We further validated these directives by finding StackOverflow questions on them, indicating that and how they have been violated in real development situations. We were able to find questions that corresponded to all but one of the directives in the dataset; we further replicated the violations using sample Android applications (as we revisit in Section 4.3). Table 1 summarizes and describes the Android state-based directives we use as case studies.

3 FRAMEFIX APPROACH
FrameFix is an end-to-end approach for statically detecting and repairing certain classes of framework-specific errors. Figure 1 provides a high-level overview of the approach. At framework development time, knowledgeable engineer (likely a developer of the framework) specifies directives relevant to correct usage of that framework, in a way that allows for statically detecting their violation (Section 3.1) and, ultimately, repairing them. The cost of writing these specifications is highly amortized, and can be considered part of the process of documenting correct usage (which developers already do). An application developer can then use FrameFix to statically check for violations of the associated directives in client plug-ins (Section 3.2).

FrameFix attempts to construct repairs for detected violations (Section 3.3). The static violation report serves to localize the error. We leverage unique aspects of the framework application development to guide automated repair for framework applications. Specifically, FrameFix leverages the object protocols and inversion of control inherent in framework applications to repair framework application problems. Since frameworks often change object states in internal framework code, FrameFix can move method calls that have incorrect states to other parts of the application with different states, removing the state problem. Frameworks also implement inversion of control through callbacks, method calls with specific signatures, FrameFix can use the signatures to find applications that contain the same call signature and use code from other applications which implemented the callbacks correctly.

FrameFix then validates generated repairs using the static analysis check, as well as any test cases associated with the application. Following a standard generate-and-validate repair paradigm [17], FrameFix attempts multiple repairs until either a time limit is exceeded, or one is found that satisfies all validation checks.

3.1 Specification language
The first step in the FrameFix repair process is statically identifying directive violations to be repaired. Since directives written in natural language are often imprecise and use meta-terms that do not clearly map to code, a person knowledgeable in the framework (likely the framework developer), presently needs to translate the directives into the specification language. While this translation may require expert knowledge, the directives only need to be specified once, and then can be checked for all framework plugins or client application. FrameFix thus takes as input a specification of how to check for framework-specific directive violations. Towards this end, we created a specification language that can check a wide variety of properties between a framework and framework application.

Figure 2 provides the grammar for our specification language, which includes both assertions (which define checks), and control statements, to logically compose assertions. Assertions can either be a context assertion or simple assertion.

A context assertion limits the assertion to a part of the code base. For example, to check that Fragment classes contain a definition for the onCreateOptionsMenu method, the context assertion would be written as checkSubclassesOf("Fragment").defines("onCreateOptionsMenu"). Some other examples of contexts are nested classes and method definitions. Since multiple contexts are often required to define the section of code to check (e.g., a method definition in subclasses of a certain Class), contexts can be nested with the "\" operator. A simple assertion could check if the application defines a necessary configuration file. Simple assertions do not require a context, but can be modified by a context. In contrast, a simple-context-assertion is a check that must have an associated context, making it part of a context assertion.

Differing from a context, a state is a list of methods where the desired object state is known to be true. The state is then checked using the methods in the stack trace (i.e. if the static path in the call graph from the current method to the start of execution contains this method call, then the application is known to be in this state). A method-call, which specifies the method call to look for in the code — when checking if setHasOptionsMenu is called in onCreate , setHasOptionsMenu is the method-call, while onCreate is the context. method-call has an optional parameter-list, which adds parameter requirements to the call. For example, if the checker is only interested in cases of setHasOptionsMenu when called with the parameter true, true would be defined as a required parameter. Required parameters can either be constant values, which only
### Table 1: The directives used in our current study. The first column is the label which we will use to refer to the directive later in the paper, the second is the directive. The third column is further explanation of the directive and the last column is the result of violating the directive.

<table>
<thead>
<tr>
<th>Label</th>
<th>Directive</th>
<th>Explanation</th>
<th>Typical Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetActivity</td>
<td>getActivity can only be called when the Fragment has been attached to an associated Activity.</td>
<td>A Fragment must be in an Activity-attached state, otherwise getActivity returns null.</td>
<td>NullPointerException</td>
</tr>
<tr>
<td>OptionsMenu</td>
<td>To use a Fragment’s OptionsMenu, the application must contain both the call setHasOptionsMenu(true), and define onCreateViewOptionsMenu.</td>
<td>If the Fragment is in the (static) hasOptionsMenu state, through the presence of a defined onCreateViewOptionsMenu, then the Fragment must call setHasOptionsMenu. Alternatively, if the Fragment can request OptionsMenu (through a call to setHasOptionsMenu(true)), then it must start in the hasOptionsMenu state.</td>
<td>The OptionsMenu will not appear.</td>
</tr>
<tr>
<td>SetArguments</td>
<td>setArguments cannot be called until after the Fragment is instantiated.</td>
<td>Once a Fragment instance has passed out of its instantiation state, a call to setArguments is illegal.</td>
<td>RuntimeException</td>
</tr>
<tr>
<td>Inflate</td>
<td>The onCreateView method in a Fragment that will attach to the user interface, should return the result of a call to inflate where the last parameter is false.</td>
<td>If the Fragment will become part of the Activity user interface (a static state designation) then the Fragment’s View must be initialized correctly.</td>
<td>Crash due to an internal framework error.</td>
</tr>
<tr>
<td>FindViewById</td>
<td>setContentView must be called before findViewById.</td>
<td>Android does not associate an ID with a View until the View is in the initialized state.</td>
<td>RuntimeException</td>
</tr>
<tr>
<td>GetResources</td>
<td>getResources cannot be called in a background Fragment.</td>
<td>getResources can only be called on a Fragment when it is attached to an Activity, a state that a background Fragment is not in by definition.</td>
<td>RuntimeException</td>
</tr>
<tr>
<td>SetInitialSavedState</td>
<td>setInitialSavedState cannot be called after the Fragment is attached to an Activity.</td>
<td>A Fragment cannot be initialized once it is in the attached-to-Activity state.</td>
<td>RuntimeException</td>
</tr>
<tr>
<td>SetTheme</td>
<td>setTheme cannot be called after setContentView.</td>
<td>It is an error for an Activity to set an application theme after it has entered the View-initialized state.</td>
<td>RuntimeException</td>
</tr>
<tr>
<td>SetPackageSetSelector</td>
<td>setPackage and setSelector cannot be called on the same Intent instance.</td>
<td>If an Intent instance is already in the package-set or selector-set state, it cannot enter the other state.</td>
<td>RuntimeException</td>
</tr>
</tbody>
</table>

**Figure 1:** The high-level FrameFix process. The framework developer creates the analysis check specification (Section 3.1) and defines the variables required to perform the repair for that problem. When a developer is developing an application using the framework, the developer can run the analysis check to see if the analysis check detects any problems in the application. If a problem is detected, FrameFix will attempt to repair the application by generation possible fixes (Section 3.3). If a fix fails to pass the checker and application test cases, FrameFix generates another possible fix. If FrameFix able to generate a repair that passes both the analysis check and all the application’s test cases, the application is considered to be fixed.

**3.2 Statically identifying directive violations**

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

**Example.** Consider the directive (Section 2.2):
While not explicitly mentioned in the directive, the API specification restricts the check to the `onCreate` method because an `Activity` with a user interface should set the user interface by calling `setContentView` in `onCreate`. The checker will then check the execution paths of `onCreate` to make sure that `findViewById` is not called before `setContentView`.

### 3.3 Automatic repair of detected violations

As shown in Figure 1, to initialize a repair, someone knowledgeable with the framework will need to define a few variables for the repair process. These variables include the method of interest and any possible second method of interest (the method or methods that were checked). The likely location of the methods of interest if one exists (such as the `onCreate` method). And search terms that can be used to narrow down the GitHub search. These variables should be defined when the check specification is written. FrameFix tries three strategies to repair detected violations; Figure 3 provides an overview. FrameFix tries three high-level repair strategies. They are ordered by the likely worst-case time of each step — smallest to largest. The strategies in order:

1. **Method reordering.** Frameworks' heavy use of object protocols can lead to problems associated with calling methods while in the wrong state. One possible solution to this problem is to reorder the problematic method calls, to see if an incorrect method call order leads to one of the methods being called in the wrong state. FrameFix therefore starts by exploring rearrangements of the methods mentioned in the directive specification (Section 3.3.1).

2. **Method movement.** Directive violations can also be caused by calling methods when the application is not in the appropriate framework state. Thus, FrameFix next tries moving implicated method calls to alternative locations altogether, in an effort to move a necessary call to a location corresponding to an appropriate state (Section 3.3.2).

3. **Callback-informed search.** Inversion of control is typically implemented via methods with framework-specific type signatures and, as such, applications typically interact with the framework through highly rigid API calls. This provides a useful filtering mechanism to constrain the search space of similar code suitably to inform repair. Thus, if the preceding attempts do not work, FrameFix searches GitHub for similar code to inform a candidate repair, based on the method signatures used in the code under repair (Section 3.3.3).

If any proposed repair passes both the static checker and any available test cases for the application, then the application is considered successfully repaired. If none of these changes produce a fix, then FrameFix was unsuccessfully in fixing that error instance.

#### 3.3.1 Method order repair

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

FrameFix generates three possible repairs for this strategy. Assume that method foo should be called before method bar, but bar occurs on line 3 and foo occurs on line 6. Also assume that line 4 uses the result of bar. In this strategy, the first proposed repair is to move foo before bar (move line 6 to be a new line 4 and move line 4 and 5 back 1 line). The second proposed repair is to move line 3 and line 4 (the lines that use bar) after line 6 (the line that uses foo). If neither of those repairs pass the validation check, the last proposed repair will propose to delete the line with the last method of interest (in this case, line 6 with foo).

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

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

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

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

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

If the previous steps do not identify repairs, FrameFix searches GitHub for applications that implement similar methods (determined by the name of the method) and other keywords provided with the directive specification (e.g., for inflate, the keywords ‘inflate’ and ‘Fragment’ are also included). Once a matching application is found, the application code from GitHub is reduced down to the method call of interest (e.g., if the repair is set to look for instances of the onCreateViewCreated method, the application off GitHub is reduced down to only the onCreateViewCreate method). FrameFix will then compare the method from GitHub to the method with the same signature in the application to repair. (using the same example, FrameFix compares the onCreateViewCreate method from GitHub to the onCreateViewCreate method in the application to repair).

To compare the two methods, the methods are converted to a list of method calls and a list of types per line, as shown in Figure 4. The intuition behind this approach is the application under repair likely needs to correct a method call or correct the types in the method call. Thus, the application under repair is first evaluated against the method from GitHub for method call differences, and then for type differences.
FrameFix: Automatically Repairing Statically-Detected Directive Violations in Framework Applications

ESEC/FSE 2020, 8 - 13 Nov., 2020, Sacramento, California, U.S.A.

Figure 4: An example of how a method definition is turned into the list of method calls and types in each line. The repair technique uses both approaches to determine differences between the application to repair and the reference application from GitHub.

First, the method calls per line are compared. If the a line in the GitHub application’s method is found to contain a method that is not in the application under repair, that line is saved as a possible line to add. If a line in the application under repair contains method call that is not used in the GitHub application’s method, then that line is saved in the possible deletion list. The first proposed repairs will try to add a single line from the line to add list and remove a single line from the line to delete list. If the proposed repairs do not pass the validation tests, then the next proposed repairs will try to delete a single line from the line to remove list. If deleting a single line does not produce a valid fix, then the next repairs generated will try all combinations of adding a single line from the line to add list and removing a single line from the line to delete list. This process repeats for adding and removing 2 or more lines until all add/delete combinations have been proposed. When lines are added from the GitHub application to the application under repair, the instance variables in the lines are changed to instance variable names with the same type in the application under repair.

If the application under repair is still not fixed, FrameFix will then recreate the addition and deletion list, but this time by comparing the types of variables in the lines of the method to compare. Lines are not directly compared (line 4 in one method is not always compared to line 4 in the other method). Instead, lines are compared on the order in which an instance of that type is found (e.g., if a View instance occurs on line 3 and line 5 one method and line 2 and 4 in the other method, then line 3 will be compared to line 2 and line 4 will be compared to line 5), to avoid the case where an added line always causes the following lines to not match. One exception to evaluating the lines by type is that true and false are treated as different types. If a line is found to contain a type difference in the application under repair, then that line is added to the lines to delete. If a line is found to have a unique type in the application from GitHub, that line is added to the list of lines to add. All combinations of adding and deleting type lines are used to generate proposed fixes, the same way FrameFix generated repairs using method difference.

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

4 EVALUATION

We evaluated if the specification language could apply to object protocols that it could encounter. We also evaluated FrameFix to demonstrate three claims: 1) that the directive violations covered by FrameFix apply to diverse set of applications in the framework, 2) that FrameFix works on real applications, and 3) FrameFix can repair problems in a diverse set of applications.

4.1 Specification evaluation

To evaluate if the API specification approach can apply to a diverse set of specification requirements, we evaluated the specification approach using the Beckman taxonomy of object protocols [16]. Beckman manually classified the different types of over 600 object protocols found in 4 programs or libraries. Beckman provided an example protocol for each category (except for the ‘other’ category, so we excluded it). We then checked if the example protocols could be encoded in our specification language.

The encoded API specifications for each of the Beckman taxonomy categories are shown in Table 2. Five of the seven examples translated easily into the specification language. For the other examples, a new simple-context-assertions had to be defined to limit the number of method calls for an object. The Boundary category was the most difficult to translate into our specification language, because the example checked a dynamic property (limiting calls to the number of items in the Iterator). While not a perfect solution, if we assume that the number of valid calls can be determined statically (in this example, we use the value seven), then the specification language can handle this case. Incorporating dynamic checks to the language would address this issue, and is left to future work.
4.2 Applicability of directives covered

To investigate if the chosen directives apply to a large percentage of Android applications, we collected 1,964 applications from F-Droid\(^1\), a catalog of open source Android applications. FlowDroid threw an error when analyzing 108 of the applications, so those were removed from the dataset, leaving 1,856 applications to analyze. We downloaded the applications and determined if the applications used the methods mentioned in the directives discussed in Table 1. Table 3 shows how many applications use each directive in the dataset. These percentages are calculated out of the number of applications in the dataset for which FlowDroid was able to produce a call graph (1,865). The main takeaway from this table is that at least one directive applies to 84.7% of the applications in the dataset, demonstrating that the directives covered in this investigation apply to wide range of applications.

### 4.3 Repair of manually built applications

To evaluate the repair process, we manually created applications that violated the nine directives mentioned in Table 1. We created the repair scenarios by starting with a base application. The base application was either a sample application from the Google website\(^2\) or, if the directive did not apply to the sample application, a default Activity application from Android Studio. Then, if we had a StackOverflow question that corresponded to the directive, we added code similar to the code/scenario mentioned in the question to the base application. For the violation that did not have a corresponding StackOverflow question, we created the application with a violation by manually creating code that violated the directive.

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

### 4.4 Error detection and repair on F-Droid

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

We found that only five checkers found errors in the applications. The errors counts detected by the five checkers are shown in Table 3. The high number of GetActivity errors is due to the fact that the checker heuristically checks if the method is used in the right context, and throws an error if GetActivity is used in any context that can not easily be determined to be correct.

Unfortunately, our initial tests were not able to repair many of the detected errors at this time. The main problem is that it is difficult to build most of the applications, due to dependencies not being found. Some of the other repair cases are situations that cannot be currently repaired — GetResources and SetArguments.

The successful repair cases are shown in Table 4. RXDroid’s test cases fail because the testing code uses a version of Closure that is incompatible with more recent Java versions, so we are unable to get the test cases to work before or after the repair.

### 4.5 Repair of injected errors

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

Next, we collected a set of application repositories where we could inject the problem in the source code. We collected these repositories from the open source applications in F-Droid that had

<table>
<thead>
<tr>
<th>Category</th>
<th>Statement</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>getEncoded cannot be called before init is called on an</td>
<td>instanceOf(&quot;AlgorithmParameters&quot;)</td>
</tr>
<tr>
<td></td>
<td>AlgorithmParameters instance</td>
<td>secondCannotOccurBeforeFirst(&quot;init&quot;,&quot;getEncoded&quot;)</td>
</tr>
<tr>
<td>Deactivation</td>
<td>a closed BufferInputStream cannot be reopened</td>
<td>instanceOf(&quot;BufferInputStream&quot;), firstCannotFollowSecond(&quot;open&quot;,&quot;close&quot;)</td>
</tr>
<tr>
<td></td>
<td>a Collection.unmodifiableList instance cannot call the add method</td>
<td>instanceOf(&quot;Collection.unmodifiableList&quot;) notContains(&quot;add&quot;)</td>
</tr>
<tr>
<td>Dynamic preparation</td>
<td>For an Iterator instance, remove can only be called after next</td>
<td>instanceOf(&quot;Iterator&quot;), firstMustOccurBeforeSecond(&quot;next&quot;, &quot;remove&quot;)</td>
</tr>
<tr>
<td></td>
<td>a AbstractProcessor instance cannot call init twice</td>
<td>instanceOf(&quot;AbstractProcessor&quot;), callLimit(&quot;init&quot;,1)</td>
</tr>
<tr>
<td>Domain mode</td>
<td>an ImageWriteParam instance can only call setCompressionType when the instance is in explicit compression mode</td>
<td>instanceOf(&quot;ImageWriteParam&quot;) firstMustOccurBeforeSecond(  &quot;setCompressionMode(MODE_EXPLICIT)&quot; , &quot;setCompressionType&quot;)</td>
</tr>
</tbody>
</table>

Table 2: API specifications for the examples of the different categories from the Beckman taxonomy.
Table 3: This table shows the number of applications in the F-Droid dataset where the directive checks apply. It also shows the number of applications where the checks signaled a possible error. The second and third column are the count and percentage of applications in the dataset that applied to each directive violation specification. At least one check means the number of applications where at least one directive could apply to the application — applications contain the method call mentioned in the directive. The fourth and fifth column are the number of violations detected in the F-Droid dataset and the percentage of detected errors out of the number of applications where the check applies. - means that this value was not calculated.

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

Table 4: The list of repairs on F-Droid applications with a problem detected.

<table>
<thead>
<tr>
<th>Application</th>
<th>Directive Violated</th>
<th>Repaired?</th>
<th>Repair Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeltaCamera</td>
<td>GetActivity</td>
<td>Yes</td>
<td>Move Method</td>
</tr>
<tr>
<td>PSLab</td>
<td>GetActivity</td>
<td>Yes</td>
<td>Move Method</td>
</tr>
<tr>
<td>RXDroid</td>
<td>OptionsMenu</td>
<td>(tests fail)</td>
<td>GitHub</td>
</tr>
<tr>
<td>RXDroid</td>
<td>Inflate</td>
<td>(tests fail)</td>
<td>GitHub</td>
</tr>
</tbody>
</table>

Table 5: The number of injected (inj.) F-Droid applications with the violations, and the number of injected applications repaired.

<table>
<thead>
<tr>
<th>Directive Violated</th>
<th>Repair Type</th>
<th># Inj</th>
<th># Repaired</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetPackageSetSelector</td>
<td>Reorder Methods</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>SetContentView</td>
<td>Reorder Methods</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>SetTheme</td>
<td>Reorder Methods</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>SetInitialSavedState</td>
<td>Move Method</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>GetActivity</td>
<td>Move Method</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Inflate</td>
<td>GitHub</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>OptionsMenu</td>
<td>GitHub</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

5 LIMITATIONS

There are a few limitations to the current FrameFix approach. The goal of our specification language is to be sufficiently expressive to support a study of automatic repair of directive violations. It does not include all possibly useful features in specifying all conceivable state-based directives. For example, we assume the number of arguments to a method are fixed; it also does not support explicitly stating alternative contexts created by program branching in the specification, but program branching can be checked in the underlying data flow analysis. We do support extension and customization through the created-* language terms, and envision extending it a publicly accessible repository posted in the F-Droid metadata folder. 1,657 repositories met this requirement.

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

Using those 115 application repositories, we checked which repositories used code that applied to the directives being violated, so injecting the problem did not require significantly changing the applications. Often only about five applications met all of these criteria for each directive that was tested. We also only tested the seven directive violation types that we are able to fix with the current FrameFix approach. The results of injecting repairs are shown in Table 5, showing that the repair technique is able to fix most of the injected problems. FrameFix was not perfect when repairing the GetActivity cases, due to the difficulty of finding a valid method to call getActivity, and the Inflate cases due to injecting the problem in a long method definition. This long method definition caused the repair to generate a large number of possible repairs, eventually timing out before successfully repairing the application.
suitably in future work. Another limitation is that it might be difficult for the average framework application developer to translate a directive into a check in the specification language. This difficulty occurs because natural-language directives are not always easily translated into a precise definition. Our approach addresses this limitation by only allowing multiple developers to reuse the same specification, meaning that the specification only has to be written once.

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

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

6 RELATED WORK
6.1 Automated repair

Automated Program Repair is an area of research with the goal to remove identified software failure without human intervention. In automated repair, there are three main categories: heuristic repair, constraint-based repair, and learning-based repair. Many repair techniques use a heuristic to generate possible repairs. Notable examples in this category include GenProg [18, 19], AE [20], BSeqRepair [21], SPR [22], and PAR [23]. FrameFix differs from these approaches by the choice of heuristic changes used to generate repairs, since FrameFix is based on framework insights.

Another major family is semantic-based repair techniques. Examples of semantic-based repair techniques include Angelix [24], SemFix [25], DirectFix [26], Qlose [27], S3 [28], and FootPatch [29]. Semantic-based program repair uses semantic analysis, commonly symbolic execution, and a set of test cases to infer desired program behavior. These techniques calculate the repair based on the given constraints. While these techniques have shown promise, most could only repair a small set of state-based framework violations. One exception is a synthesis-based approach that is similar to FrameFix, Phoenix [30]. Phoenix uses static analysis to identify faults and validate fault repairs. FrameFix differs from Phoenix in the types of bugs that each repair technique fixes and the general repair approach. Phoenix fixes errors caught by FindBugs [31] using a programming-by-example approach to synthesize repairs, while FrameFix avoids having to specify an example for each detected problem type (directive violation).

The final family is learning-based repair, where the techniques use machine learning and past fixes to propose repairs. Examples from this group include Prophet [32] and DeepFix [33]. FrameFix does not learn from past repairs, so it is not a learning-based repair.

6.2 Android and Framework Repair

Recent work on the Android framework has categorized a large number of Android exceptions and extracted common repair patterns for problems that produce exceptions [34]. Some of these repair patterns are useful for an Android specific implementation of FrameFix. Tan et al. have repaired crashing Android applications using manually created patterns [13]. These repair patterns served as a source of inspiration for some of the framework repairs in FrameFix. This approach differs from FrameFix in that the repair process required a recorded event sequence to produce a crash, and that repairs were manually validated - either by comparing the intermediate code change to developer changes or by running the application to determine if the crash was fixed. In contrast to these approaches, FrameFix was designed to be framework independent and thus handles a wider set of framework issues than those handled in the Android repair techniques.

Other research has taken a different approach to repairing framework applications than FrameFix. One approach to framework repair is to use contracts and dynamically created object behavior models to guide repairs [14]. This approach requires a significant developer investment to work for most frameworks, since both the framework and application would need to be written in a language that uses contracts. FrameFix instead only requires specifying the directive to check and how to translate the directive into code if a custom check is required. Another tool, Samediff, fixes out-of-date API calls by recommending methods that were added in the version where the API call became out-of-date [35]. This tool addresses a different subclass of API problems than FrameFix.

Past researchers have modeled the Android callback system for static analysis. One study evaluated how the changes between different versions of Android can be used to detect method call ordering bugs in callbacks [36], another automatically generated interprocedural call graphs from callbacks in an application [37]. An investigation into popular Android static analysis research tools found that incorrect modeling of the Android callback sequence can lead to unsound analysis results [38]. Instead of focusing on accurate static modeling of Android callbacks, our work uses the similarities between callbacks in different applications for repair.

6.3 Directives

Prior work has proposed three different directive classification schemes. One classification scheme was based on the abnormal aspect specified in the directive (e.g., the calling restrictions, method limitations, or side-effects) [8] while another focused on the segment of code covered by the directive (e.g., line, method, or object) [39]. The third classification scheme is based on the keywords in the directive (e.g., directives with the word ‘error’ or ‘illegal’ are grouped into the same category) [40]. Another study on directives 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 [41]. Other researchers have investigated certain directive categories: directives specifying how to extend objects to implement the framework [42],
and parameter usage constraints [43]. The tools in this proposal use directives as a way to identify framework state constraints.

7 CONCLUSION
We have demonstrated we can use the insight that frameworks heavily use of object protocols and inversion of control to guide the repair of state-based framework application problems. We demonstrated this through our implementation of Framefix, which was able to repair both detected bugs and bugs injected into real applications. Possible expansions on this work include automatically inferring part or all of the checker specifications and addressing other types of framework application issues.

8 ACKNOWLEDGMENTS
Thank you to everyone who provided help along the way.

REFERENCES