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

Zack Coker*, Joshua Sunshine*, Claire Le Goues*

*Carnegie Mellon University

{zfc, sunshine, 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 based on the principles of how frameworks interact with framework applications: object protocols and inversion of control. To demonstrate that these principles can be used to aid automated repair for framework applications, we created a sample implementation of FrameFix for Android applications. Our evaluation shows that FrameFix is effective, repairing both real bugs in real applications, and a large number and variety of injected defects.**

Index Terms—**Frameworks, Automated Repair, API**

I. 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 mobile device hardware, 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 StackOverflow, 6 correspond to framework categories, containing more than 3 million questions [4]. New framework application developers encountered difficulties understanding how their changes affect an application and in selecting the right framework interface to solve

their problem [5]. Developers have been known 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 debugging challenges to developers [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 exhibit key features that can help in developing automated tooling to improve the debugging process:

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 occur 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 applications 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.

We propose an end-to-end static program repair technique that automatically repairs state-based framework errors, a particularly challenging class of framework debugging problems. In our approach, someone knowledgeable with the framework (likely the framework developer) encodes *directives* [8] (method call requirements not enforced by the type system) using a specification language and provides the starting parameters of the repair process. Application developers then run analyses generated from the specification language to find and fix defects in their client applications. If the analysis finds an error, then FrameFix generates possible repairs guided by the key features that differentiate framework application repair. Proposed repairs are evaluated with the analysis and an

Label	Directive	Explanation	Typical Error
GetActivity	getActivity *can only be called when the* Fragment *has been attached to an associated* Activity.	A Fragment must be in an Activity-*attached* state, otherwise getActivity returns null	NullPointerException
OptionsMenu	*To use a* Fragment's OptionsMenu, *the application must contain both the call* setHasOptionsMenu(true), *and define* onCreateOptionsMenu.	If the Fragment is in the (static) *has*-OptionsMenu state, through the presence of a defined onCreateOptionsMenu, then the Fragment must call setHasOptionsMenu; Alternatively, if the Fragment can request OptionsMenu (through a call to setHasOptionsMenu(true)), then it must start in the *has*-OptionsMenu state.	The OptionsMenu will not appear.
SetArguments	setArguments *cannot be called until after the* Fragment *is instantiated.*	Once a Fragment instance has passed out of its instantiation state, a call to setArguments is illegal	RuntimeException
Inflate	*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.	If the Fragment will become part of the Activity user interface (a static state designation) then the fragment's View must be initialized correctly.	Crash due to an internal framework error.
FindViewById	setContentView *must be called before* findViewById.	Android does not associate an ID with a View until the View is in the initialized state.	RuntimeException
GetResources	getResources *cannot be called in a background* Fragment.	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.	RuntimeException
SetInitialSavedState	setInitialSavedState *cannot be called after the* Fragment *is attached to an* Activity.	A Fragment cannot be initialized once it is in the *attached-to*-Activity state.	RuntimeException
SetTheme	setTheme *cannot be called after* setContentView.	It is an error for an Activity to set an application theme after it has entered the View-*initialized* state	RuntimeException
SetPackageSetSelector	setPackage *and* setSelector *cannot be called on the same* Intent *instance.*	If an Intent instance is already in the *package-set* or *selector-set* state, it cannot enter the other state.	RuntimeExcpeption

TABLE I: Directives used in our study. The first column is a shorthand label for each directive; the second is the directive proper, further explained in the third column. The last column shows the result of violating the directive.

application's test cases.

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 prevalance 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 causes application crashes. Our approach also does not require framework developer and application developers to write contract specifications [14] to guide repair.

Our primary contributions are:

- An automated repair technique based on framework's heavy use of object protocols and inversion of control to guide the automated repair process.
- An instantiation of the technique for Android applications, called FrameFix[1].
- An evaluation on F-Droid, a set of 1,964 open source Android applications

We begin by discussing the necessary background on framework bugs in Android (Section II). We then discuss the approach we used to create FrameFix in Section III and our evaluation of FrameFix in Section IV. We end the paper with a discussion of limitations (Section V), related work (Section VI), and finish with conclusions (Section VII).

II. STATE-BASED FRAMEWORK BUGS

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

A. 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
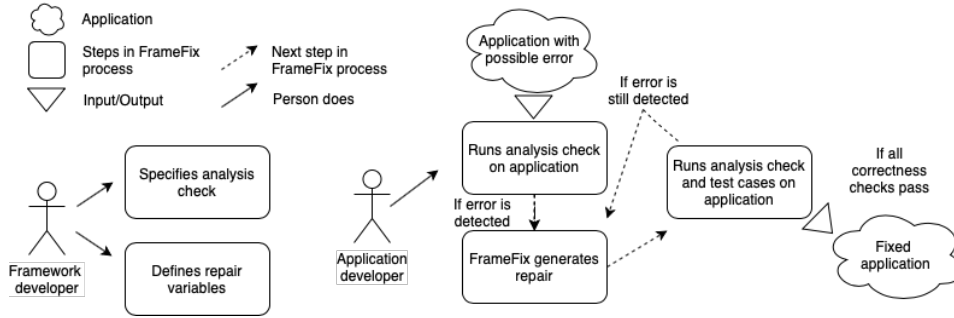
[1]http://bit.ly/3qg7M9v

Fig. 1: The high-level FrameFix process. A framework developer creates a check specification (Section III-A) and defines the variables required to perform associated repairs. A developer of a framework client application can run the resulting analysis check to detect problems. FrameFix will attempt to repair detected problems by generating possible fixes (Section III-C). If FrameFix is able to generate a repair that passes both the analysis check and all the application's test cases, the application is considered fixed.

specific goals by writing *plugins*, or client applications, that use the framework. For example, Android applications are plugins to the Android framework. A framework typically calls plugin code through *inversion of control*, a design in which the core framework code, not the application-specific code, controls an application's data and execution flow [1]. Inversion of control is typically implemented by defining *callback methods* with well-specified method signatures in the framework application, which the framework calls when needed. 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 state-based problems. 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].

We implemented our proof-of-concept for Android because it is the most popular framework on StackOverflow, and thus most likely to have a demonstrable need for state-based automated repair. Issues in state-based development issues in Android have also been previously studied [7]. In an Android application, the `Activity` class is the main application entry point. `Fragments` are reusable subcomponents of an `Activity`. `Intents` are the objects that Android passes between applications, which facilities inter-application communication. The directives we discuss 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 by FrameFix in the static analysis and to evaluate if a source-code level repair succeeded.

B. 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 which are particularly 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 further validated these directives by finding StackOverflow questions about them, indicating that and how they have been violated in real development situations. We were able to find questions corresponding to all but one of the directives in the dataset; we further replicated the violations using sample Android applications (as we revisit in Section IV-C). Table I summarizes our case study Android state-based directives.

III. FRAMEFIX APPROACH

Figure 1 shows a high-level overview of FrameFix, an end-to-end approach for statically detecting and repairing certain classes of framework-specific errors. At *framework development time*, a knowledgeable engineer specifies directives relevant to correct framework usage, in a way that allows for statically detecting violations (Section III-A) and, ultimately, repairing them.[2] 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 directive violations in client plug-ins (Section III-B)

FrameFix attempts to construct repairs for detected violations (Section III-C). The static violation report serves to localize the error. FrameFix leverages the unique aspects of framework application development to guide repair of framework applications: the object protocols and inversion of control

---

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 2: Grammar for creating heuristic directive analyses, including assertions and control statements for combining them.

inherent in framework applications. Since frameworks often internally change object states, FrameFix can move method calls that have incorrect states to other parts of the application with different states. Frameworks also implement inversion of control through callbacks. FrameFix can search GitHub for applications containing the same callback method signatures, and then reuse code implementing the callbacks correctly.

FrameFix validates generated repairs using the static check, as well as any application test cases. 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.

For some errors, it may be possible for a developer to specify a fix template for that error type to fix the issue. However, our goal was to create a repair process that could apply to as many error types as possible and could could repair more errors than the directives evaluated in this study.

A. Specification language

The first step in the FrameFix repair process is to statically identify directive violations. Since directives written in natural language are often imprecise and use meta-terms that do not clearly map to code, a knowledgeable developer must translate

directives into the specification language. While this translation may require expert knowledge, the directives only need to be specified once per framework. Thus, the specifications that we have already encoded do not need to rewritten.

Figure 2 provides the grammar for our specification language, which includes both assertions (which define checks), and control statements, to logically compose assertions. 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")`. Other examples of *context*s include nested classes and method definitions. *Context*s can be nested with the '.' operator (e.g., a method definition in subclasses of a certain class). 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. The *created-* checks are possible extension points, such as *created-context* where users can define their own *context* check.

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* specifies the method call to look for in the code, e.g., 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. For example, if the checker is only interested in cases of `setHasOptionsMenu` when called with the parameter `true`, `true` is specified as a required parameter. Required parameters can either be constant values, which must be syntactically equal to match, or identifiers, which match if the same object reference is used across multiple method calls. A wild card '*' denotes a parameter that does not need to be matched.

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

B. 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 functions 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 directive violations an application.

To illustrate, consider the **FindViewById** directive (Section II-B). This directive translates to the specification language (Section III-A) as

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

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

While not explicitly mentioned in the directive, the API specification restricts the check to 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 `onCreate`'s execution paths to make sure that `findViewById` is not called before `setContentView`.

C. Automatic repair of detected violations

As shown in Figure 1, to initialize a repair, the framework-knowledgable developer defines select variables, including methods of interest (up to two, the method or methods that were checked), the likely methods containing calls to 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. The strategies are ordered by the likely worst-case time required:

1) **Method reordering.** Frameworks' heavy use of object protocols can lead to problems associated with calling methods while in the wrong state, sometimes because methods are called in the wrong order. One possible solution is to reorder the problematic calls. FrameFix therefore starts by exploring rearrangements of the methods mentioned in the directive specification (Section III-C1).

2) **Method movement.** Directive violations can also be caused by calling methods when the application is not in the appropriate state. Thus, FrameFix next tries moving implicated method calls to alternative locations altogether, seeking one that corresponds to an appropriate state (Section III-C2).

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 filtering mechanism to constrain the code of the repair search space. Thus, if the preceding attempts do not work, FrameFix searches GitHub for code to inform a candidate repair, based on the method signatures used in the code under repair (Section III-C3).

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

1) Method order repair: Since frameworks heavily use object protocols, in both internal framework code and in client applications, a bug may correspond to a method call occuring when an object is in the wrong state. For example, a call to get a resource view item in Android cannot occur before the view has been initialized (**GetResources**). In this case, the `Activity` is performing steps in the wrong order, and the calls could be valid if called in a different order. FrameFix thus attempts to reorder method calls to produce possibly interprodecural repairs. FrameFix only tries this approach if multiple methods are mentioned in the directive specification, and only tries to reorder the mentioned methods.

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, and that line 4 uses the result of `bar`. The first proposed repair moves `foo` before `bar`. The second proposed repair is to move line 3 and line 4 (the lines that call and use `bar`) after line 6 (the line that uses `foo`). If neither of these repairs succeed, FrameFix tries to delete the second method of interest (line 6 with `foo`).

2) Method move repair: 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 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 (**GetActivity**). This type of violation can be fixed by moving the problematic call to a different method body, 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. FrameFix uses this approach to generate possible intraprocedural repair.

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.
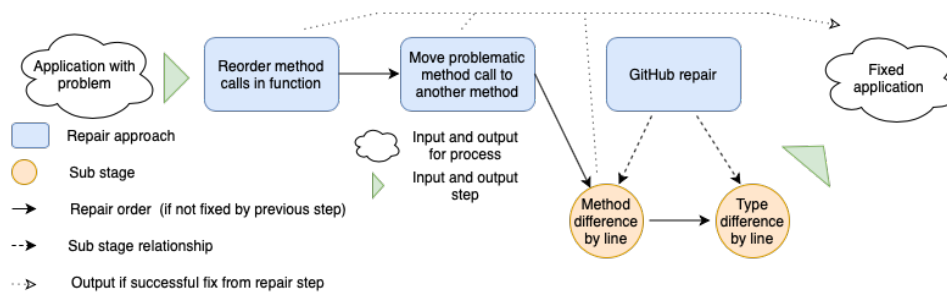
Fig. 3: The FrameFix repair process.

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 heuristically tries to ensure that the parameters used in the method call are still valid. It does this by scanning the preceding lines in the method definition for references to variables used in the moved call. 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, this preceding line will also be copied when moving the method. FrameFix also copies any relevant try-catch statements to the new context.

3) Callback-based repair: The difficult part of using reference applications from GitHub is choosing the section of code that can be used to obtain the fix. Our novel insight to reduce this search space problem is that frameworks often implement inversion of control through methods that have the same signature across all applications. Our technique, FrameFix, exploits this by trying to repair a faulty method using methods with a similar signature off 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 `onViewCreate` method, the application off GitHub is reduced down to only the `onViewCreate` method). FrameFix will then compare the method from GitHub to the method with the same signature in the application to repair. (using the same example, FrameFix compares the `onViewCreate` method from GitHub to the `onViewCreate` method in the application to repair).

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.

First, the method calls per line are compared. If the line in the GitHub application's method is found to contain a method that is not in the application under repair, that line is saved as a possible line to add. If a line in the application under repair contains method call that is not used in the GitHub application's method, then that line is saved in the possible deletion list. The first proposed repairs will try to add a single line from the line to add list to the application under repair. If the proposed repairs do not pass the validation tests, then the next proposed repairs will try to delete a single line from the line to remove list. If deleting a single line does not produce a valid fix, then the next repairs generated will try all combinations of adding a single line from the line to add list and removing a single line from the line to delete list. This process repeats for adding and removing 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, the process

```
1  @Override
2  public View onCreateView(
3      LayoutInflater inf, ViewGroup cont,
4      Bundle saved) {
5   View v =
6      inf.inflate(R.listView, cont, false);
7   dLE = new DLE(this, this, getActivity());
8   ((ListView) v.findView(R.DirList)).
       setAdapter(dLE);
9   return v;
10 }
```

(a) The original code (simplified for presentation).

```
6  inflate()
7  DLE(), getActivity()
8  findView(), setAdapter()
```

(b) Code as a list of method calls per line. Constructors are included; lines can contain multiple calls.

```
6  View, LayoutInflater, constant_reference,
       ViewGroup, false
7  DLE
8  View, constant_reference, DirList
9  View
```

(c) Converted to types. **R.** references become constant_references. **true** and **false** are separate type values. **this** references and method call return types are excluded (unless the call is to a constructor) for implementation ease.

Fig. 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 the differences between the application to repair and the reference application from GitHub.

is repeated on another sample application in the order returned from a GitHub API request that contains the method name of interest, any specified key terms from the directive, and the most repository stars. This process is repeated until a set timeout is reached, currently set at one hour.

IV. EVALUATION

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

A. 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 attempted to encode the example protocols in our specification language.

The encoded API specifications for each of the Beckman taxonomy categories are shown in Table II. 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 seven), the specification language can handle this case. Incorporating dynamic checks to the language would address this issue, and is left to future work. This evaluation shows our specification language can handle a diverse set of specification requirements, but cannot perfectly address all specification cases.

B. Applicability of directives covered

To investigate if the chosen directives apply to a large percentage of Android applications, we implemented the specification language and static analyses using FlowDroid, an Android static analysis tool [18]. We then collected 1,964 applications from F-Droid[3], a catalog of open source Android applications. FlowDroid, and thus the static analysis tools, 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 I. Table III 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.

C. Repair of manually built applications

To evaluate the repair process, we manually created repair scenarios which consisted of applications that violated the nine directives mentioned in Table I. We created the repair scenarios by starting with a base application that was either a Google sample application[4] or, if the directive did not apply to the sample application, a default **Activity** application from AndroidStudio. 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 question, we manually created 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

---

[3]https://f-droid.org/en/ (dataset: https://doi.org/10.5281/zenodo.3698376)
[4]github.com/googlesamples/android-LNotifications

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

TABLE II: API specifications for the examples of the different Beckman taxonomy categories.

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

TABLE III: Applications in the F-Droid dataset where directive checks apply, as well as applications where checks signaled possible errors. The second and third columns show count and percentage of applications in the dataset that applied to each directive violation specification. At least one directive applies 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 columns show 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.

undefined because a successful repair would require adding a new method to the application, which is not currently supported by the repair approach. The **GetResources** and **SetArguments** encounter similar problems. While the 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.

D. 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

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

TABLE IV: Repairs on F-Droid applications with detected problems.

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 III. The high number of **GetActivity** errors is due to a heuristic context check that throws an error if **GetActivity** is used in any context that cannot 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

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

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

situations that cannot be currently repaired — **GetResources** and **SetArguments**.

The successful repair cases are shown in Table IV. RX-Droid's test cases fail before and after the repair because the test code uses a version of Closure that is incompatible with more recent Java versions.

E. 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 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 test cases without problems. Thus, we used the standard process to build a debug APK and run the test cases for the build (the Gradle `assembleDebug` and `test` commands). Of the 1,657 repositories tested, 115 applications met these criteria without errors.

Using those 115 application repositories, we checked which repositories used code that applied to the seven directive violation types that we are able to fix with the current FrameFix approach. The number of applications that met these criteria for each directive and the results of injecting repairs are shown in Table V, 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.

V. LIMITATIONS

There are some limitations to the current 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. To address this, we support extending the language through the *created-** language terms. Our technique relies on a person knowledgeable in the framework to be able to define a context that is not overly broad or narrow for each specification. Another limitation is that a framework expert may be required to translate directives into specification language checks, because natural-language directives are not always easily translated into a precise definition. This limitation is mitigated because multiple developers can use the same specifications, meaning specifications only have 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. 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. The current FrameFix implementation is not designed to address multi-threading issues, but it could be expanded to do so. Finally, the Github repair technique has a computational complexity issue when testing the differences between large methods. This problem is not addressed in the current FrameFix implementation and left as a possible future improvement

VI. RELATED WORK

A. Automated repair

Automated Program Repair is an area of research with the goal to remove identified software failures 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 [19], [20], AE [21], RSRepair [22], SPR [23], SapFix [24], and PAR [25]. 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 [26], SemFix [27], DirectFix [28], Qlose [29], S3 [30], the tools PHPQuickFix and PHPRepair [31], and FootPatch [32]. *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 the synthesis-based approach Phoenix [33]. Although FrameFix is similar to Phoenix, 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 [34] using a programming-by-example approach to synthesize repairs, while FrameFix avoids specifying an example for each detected problem type (directive violation). ACS is another related technique that extracts *if* conditions from GitHub, limiting the reference code differently than FrameFix [35].

The final family is learning-based repair, which uses machine learning and past fixes to propose repairs. Examples include Prophet [36] and DeepFix [37]. FrameFix does not learn from past repairs, so it is not a learning-based repair.

B. Android and Framework Repair

One closely related work is Droix, which repairs crashing Android applications using manually created patterns [13]. While the repair patterns in Droix served as a source of inspiration for the repairs in FrameFix, Droix differs from FrameFix in that repairs were manually validated, and that Droix repairs Android byte code. FrameFix, instead, produces repairs at the source code, which is easier for developers to save into their project's source code repository. Other recent work on the Android framework has categorized Android exceptions and extracted common repair patterns [38]. Some of these repair patterns are useful for FrameFix.

Another approach to framework repair is to use contracts and dynamically created object behavior models to guide repairs [14], [39]. 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 supports contracts. FrameFix instead only requires specifying the directive to check and how to translate the directive into code if a custom check is required. Another tool, SemDiff, uses versioning to fix out-of-date API calls [40]. Other approaches check [41] or detect anomalies [42] in runtime behavior. One study used MuBench to compare tools that statically analyzed programs for infrequent API usage patterns to identify API misuse [43]. These tools addresses a different subclass of API problems than FrameFix, rely on infrequent patterns to identify misuse, or require multiple application runs to generate a fix.

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 ordering bugs in callbacks [44]. Another automatically generated interprocedural call graphs from callbacks in an application [45]. An investigation into popular Android static analysis research tools found that incorrect modeling of the Android callback sequence can lead to unsound analysis results [46]. Pasket [47] and SynthesiSE [48] are related techniques that synthesize a representation of the framework for symbolic execution analysis. Instead of focusing on accurate modeling of Android callbacks, our work uses the similarities between callbacks in different applications for repair.

C. 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) [49]. 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) [50]. 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 [51]. Other researchers have investigated certain directive categories: directives specifying how to extend objects to implement the framework [52] and parameter usage constraints [53]. FrameFix, instead, uses directives as a way to identify framework state constraints.

VII. CONCLUSION

We have demonstrated that we can use the insight that frameworks heavily use 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.

ACKNOWLEDGMENT

REFERENCES

[1] C. Jaspan, "Proper plugin protocols," Ph.D. dissertation, Carnegie Mellon University, 2011.
[2] R. E. Johnson, "Frameworks = (components + patterns)," *Communications of the ACM*, vol. 40, no. 10, pp. 39–42, Oct. 1997.
[3] www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, Accessed Mar. 4th, 2020.

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

[5] A. J. Ko, B. A. Myers, and H. H. Aung, "Six learning barriers in end-user programming systems," in *Visual Languages - Human Centric Computing*, ser. VLHCC '04, 2004, pp. 199–206.

[6] C. Jaspan and J. Aldrich, "Are object protocols burdensome?: An empirical study of developer forums," in *SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU '11, 2011, pp. 51–56.

[7] Z. Coker, D. G. Widder, C. Le Goues, C. Bogart, and J. Sunshine, "A qualitative study on framework debugging," in *International Conference on Software Maintenance and Evolution*, ser. ICSME '19, 2019, pp. 568–579.

[8] U. Dekel, "Increasing awareness of delocalized information to facilitate API usage," Ph.D. dissertation, Carnegie Mellon University, 2009.

[9] https://f-droid.org/en/, Accessed Oct. 11th, 2019.

[10] D. Hou and H. J. Hoover, "Using scl to specify and check design intent in source code," *IEEE Transactions on Software Engineering*, vol. 32, no. 6, pp. 404–423, 2006.

[11] C. Jaspan and J. Aldrich, "Checking semantic usage of frameworks," in *Library-Centric Software Design*, ser. LCSD '07, 2007, pp. 1–10.

[12] R. J. Walker and K. Viggers, "Implementing protocols via declarative event patterns," in *Foundations of Software Engineering*, ser. FSE 04, 2004, pp. 159–169.

[13] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing crashes in android apps," in *International Conference on Software Engineering*, ser. ICSE '18, 2018, pp. 187–198.

[14] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *International Symposium on Software Testing and Analysis*, ser. ISSTA '10, 2010, pp. 61–72.

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

[16] N. E. Beckman, D. Kim, and J. Aldrich, "An empirical study of object protocols in the wild," in *European Conference on Object-Oriented Programming*, ser. ECOOP'11, 2011, pp. 2–26.

[17] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[18] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Programming Language Design and Implementation*, ser. PLDI '14, 2014, pp. 259–269.

[19] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *International Conference on Software Engineering*, ser. ICSE '09, 2009, pp. 364–374.

[20] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *International Conference on Software Engineering*, ser. ICSE '12, 2012, pp. 3–13.

[21] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *Automated Software Engineering*, ser. ASE '13, 2013, pp. 356–366.

[22] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *International Conference on Software Engineering*, ser. ICSE '14, 2014, pp. 254–265.

[23] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '15, 2015, pp. 166–178.

[24] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "Sapfix: Automated end-to-end repair at scale," in *International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '19, 2019, pp. 269–278.

[25] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *International Conference on Software Engineering*, ser. ICSE '13, 2013, pp. 802–811.

[26] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *38th International Conference on Software Engineering*, ser. ICSE '16, New York, NY, USA, 2016, pp. 691–701.

[27] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 International Conference on Software Engineering*, ser. ICSE '13, 2013, pp. 772–781.

[28] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *International Conference on Software Engineering*, ser. ICSE '15, 2015, pp. 448–458.

[29] L. D'Antoni, R. Samanta, and R. Singh, "Qlose: Program repair with quantitative objectives," in *Computer Aided Verification*, S. Chaudhuri and A. Farzan, Eds. Springer International Publishing, 2016, pp. 383–401.

[30] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: Syntax- and semantic-guided repair synthesis via programming by examples," in *2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, 2017, pp. 593–604.

[31] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and H. Laurie, "Automated repair of html generation errors in php applications using string constraint solving," in *International Conference on Software Engineering*, ser. ICSE '12, 2012, pp. 277–287.

[32] R. van Tonder and C. Le Goues, "Static automated program repair for heap properties," in *International Conference on Software Engineering*, ser. ICSE '18, 2018, pp. 151–162.

[33] R. Bavishi, H. Yoshida, and M. R. Prasad, "Phoenix: Automated data-driven synthesis of repairs for static analysis violations," in *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '19, 2019, pp. 613–624.

[34] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, Sep. 2008.

[35] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *International Conference on Software Engineering*, ser. ICSE '17, 2017, pp. 416–426.

[36] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: ACM, 2016, pp. 298–312.

[37] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Association for the Advancement of Artificial Intelligence*, ser. AAAI '17', 2017, pp. 1345–1351.

[38] L. Fan, T. Su, S. Chen, M. Guozhu, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale analysis of framework-specific exceptions in android apps," in *International Conference on Software Engineering*, ser. ICSE '18, 2018, pp. 408–419.

[39] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," *IEEE Transactions on Software Engineering*, vol. 40, no. 5, pp. 427–449, 2014.

[40] B. Dagenais and M. P. Robillard, "Semdiff: Analysis and recommendation support for api evolution," in *International Conference on Software Engineering*, ser. ICSE '09, 2009, pp. 599–602.

[41] O. Riganelli, D. Micucci, and L. Mariani, "Controlling interactions with libraries in android apps through runtime enforcement," *Transactions on Autonomous and Adaptive Systems*, vol. 14, pp. 1–29, 12 2019.

[42] V. Dallmeier, A. Zeller, and B. Meyer, "Generating fixes from object behavior anomalies," in *Automated Software Engineering*, ser. ASE '09, 2009, pp. 550–554.

[43] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static api-misuse detectors," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1170–1188, 2019.

[44] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in android applications," in *International Conference on Software Engineering*, ser. ICSE '15, 2015, pp. 89–99.

[45] D. D. Perez and W. Le, "Generating predicate callback summaries for the android framework," in *Mobile Software Engineering and Systems*, ser. MOBILESoft '17, 2017, pp. 68–78.

[46] Y. Wang, H. Zhang, and A. Rountev, "On the unsoundness of static analysis for android guis," in *State Of the Art in Program Analysis*, ser. SOAP '16, 2016, pp. 18–23.

[47] J. Jeon, X. Qiu, J. Fetter-Degges, J. S. Foster, and A. Solar-Lezama, "Synthesizing framework models for symbolic execution," in *International Conference on Software Engineering*, ser. ICSE '16, 2016, pp. 156–167.

[48] X. Gao, S. H. Tan, Z. Dong, and A. Roychoudhury, "Android testing via synthetic symbolic execution," in *International Conference on Automated Software Engineering*, ser. ASE 2018, pp. 419–429.

[49] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini, "What should developers be aware of? An empirical study on the directives of api documentation," *Empirical Software Engineering*, vol. 17, no. 6, pp. 703–737, 2012.

[50] H. Li, S. Li, J. Sun, Z. Xing, X. Peng, M. Liu, and X. Zhao, "Improving api caveats accessibility by mining api caveats knowledge graph," in *International Conference on Software Maintenance and Evolution*, ser. ICSME'18, Sep. 2018, pp. 183–193.

[51] U. Dekel and J. D. Herbsleb, "Improving api documentation usability with knowledge pushing," in *International Conference on Software Engineering*, ser. ICSE '09, 2009, pp. 320–330.

[52] M. Bruch, M. Mezini, and M. Monperrus, "Mining subclassing directives to improve framework reuse," in *Mining Software Repositories*, ser. MSR '10, 2010, pp. 141–150.

[53] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing APIs documentation and code to detect directive defects," in *International Conference on Software Engineering*, ser. ICSE '17, 2017, pp. 27–37.