AFRL-RI-RS-TR-2020-143

FIXR: MINING AND UNDERSTANDING BUG FIXES TO ADDRESS APPLICATION-FRAMEWORK PROTOCOL DEFECTS

UNIVERSITY OF COLORADO BOULDER

AUGUST 2020

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ AIR FORCE MATERIEL COMMAND ■ UNITED STATES AIR FORCE ■ ROME, NY 13441

NOTICE AND SIGNATURE PAGE

/ S /	**/ S /**
STEVEN DRAGER	GREGORY J. HADYNSKI
Work Unit Manager	Assistant Technical Advisor
	Computing & Communications Division
	Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* AUGUST 2020	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* SEP 2014 – DEC 2019

4. TITLE AND SUBTITLE FIXR: MINING AND UNDERSTANDING BUG FIXES TO ADDRESS APPLICATION-FRAMEWORK PROTOCOL DEFECTS	**5a. CONTRACT NUMBER** FA8750-14-2-0263
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 61101E
6. AUTHOR(S) Bor-Yuh Evan Chang, Kenneth M. Anderson Pavol Cˇerný, Sriram Sankaranarayanan Tom Yeh, Sergio Mover	**5d. PROJECT NUMBER** MUSE
	5e. TASK NUMBER DC
	5f. WORK UNIT NUMBER OL
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Colorado Boulder Department of Computer Science 1111 Engineering Dr, 430 UCB Boulder CO 80309-0430	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITA 525 Brooks Road Rome NY 13441-4505 DARPA 675 North Randolph St. Arlington VA 22203-2114	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI **11. SPONSOR/MONITOR'S REPORT NUMBER** AFRL-RI-RS-TR-2020-143

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The Fixr project addressed the problem of mining and understanding from large code corpora to realize automatically transferring bug fixes from one application (app) to another. Fixr was particularly concerned with pernicious protocol bugs that arise from programming against rich software frameworks like Android. While frameworks are crucial for enabling developers to efficiently create complex apps on sophisticated platforms, they are complicated, making programming apps against them error-prone. Yet, the fact that rich frameworks have vibrant communities of app developers presents a unique opportunity for tools to improve software quality. We envisioned a reality where intelligent automated systems help app developers synthesize fixes to defects by mining public code repositories. To this end, the Fixr project developed a suite of pattern mining, test-case generation, code synthesis, and verification tools that cooperatively realize this vision. Fixr advanced the state of the art through publication at top tier venues and has moved the state of the practice by transitions with industry partners GitHub and MuseDev.

15. SUBJECT TERMS
big code, software specification and verification, program analysis and synthesis,automated and probabilistic reasoning, application testing, big-data engineering, developer tools, mining source repositories, application-programming protocols, event-driven software frameworks, syntactic-semantic representations of code

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**STEVEN DRAGER**
U	U	U	UU	81	**19b. TELEPHONE NUMBER** *(Include area code)* **N/A**

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 SUMMARY

Rich libraries and frameworks—like Android, Swing, Spring, Struts, Standard Widget Toolkit (SWT), Eclipse Rich Client Platform (RCP), Play, and Akka—enable software engineers to create complex applications on sophisticated computing platforms (e.g., smart phones with a broad range of sensors and rich interactivity). While rich frameworks are an important tool to deal with the ever-rising complexity of creating software systems, they themselves are complicated, making programming applications against them challenging and error-prone. A client application (app) must carefully follow the rules of a complex protocol to obtain the desired behavior. To properly use a framework, the app developer must understand the protocols, invariants, and idioms intended by the framework designer. A violation of these (often implicit) rules leads to pernicious and unexpected bugs.

Yet, the fact that rich frameworks have vibrant communities of app developers (e.g., Android developers on StackOverflow) presents a unique opportunity for tools to improve software quality. Imagine this scenario: an app developer discovers a bug in her app where she inadvertently broke a framework protocol rule. She, through great effort, discovers a *fix* for her app that she then commits to a public code repository. Now right after her commit, an automated system recognizes her fix and synthesizes candidate *patches* to other apps that appear to violate the same framework rule—conceptually, *transferring* the original fix to these other apps. In the end, this automated system *amplifies* the human effort of the original fix—reifying in tools the diffusion of communal knowledge that today spreads only through informal means like developer forums.

In the Fixr project, an effort under the Defense Advanced Research Projects Agency (DARPA) Mining and Understanding Software Enclaves (MUSE) program, we undertook the fundamental research to realize the scenario described above to address application-framework protocol defects. In particular, we developed a suite of pattern mining, test-case generation, code synthesis, and verification tools that cooperatively address app-framework protocol bugs. The unique aspect of our work is the use of a feedback loop to iteratively improve the confidence of repairs. That is, our approach leveraged the large and continuously increasing enclave of applications that program against any given framework— crucially using the MUSE database itself to generate high-quality artifacts with which to populate it. The missing link that we addressed in this project is deriving high-quality specifications of the framework rules. Unlike many alternative approaches, the novel aspect of our work was to infer such specifications from the volume and variety of bug fixes made by the crowd of client app developers.

Filling in this missing link heightens the impact of verification tools to improve software quality. In particular, the breadth and depth of what today's program verification tools of all varieties can prove are astounding, as long as there is a clear property to prove. Today's tools are quite effective on fixed, language-level defects, like proving the absence of null-pointer dereference or array-out-of-bounds exceptions, but much less so for framework "misuses." This situation is not necessarily out of an unwillingness to provide specifications, but frameworks like Android have become so massive that even the framework developers themselves may unknowingly break their own protocol guidelines [1, 2].

In the end, the result of our work enables leveraging existing corpora of client apps developing against a framework to (1) mine protocol usage patterns as rich syntactic-semantic artifacts [3–5]; (2) flag anomalous code as potential protocol misuses in other apps [3–5]; (3) generate user-level, end-to-end test cases to validate misuses [6–8]; (4) synthesize models of frameworks grounded in actual framework implementations [9–11]; (5) inform how the framework could evolve to avoid misuses [1, 12–14]; (6) verify that apps conform to protocols [1, 12–14]; and (7) most importantly, increase the confidence in app-framework interfaces.

2 INTRODUCTION

The central goal of the Fixr project was to automatically transfer bug fixes of app-framework protocol errors from one code location to another (as alluded to Section 1). There are two major challenges in realizing this vision:

Challenge 1: Mining at Scale and with Relevance. Frameworks such as Android are large, involving hundreds of packages, thousands of classes, and tens of thousands of application programming interface (API) methods with asynchronous interactions driven by callbacks. In Figure 1, we illustrate both challenges in the scale of the framework and of a corpus spanning many, many projects. Fixr must find relevant patterns from a large, messy corpus in the context of a large, messy framework

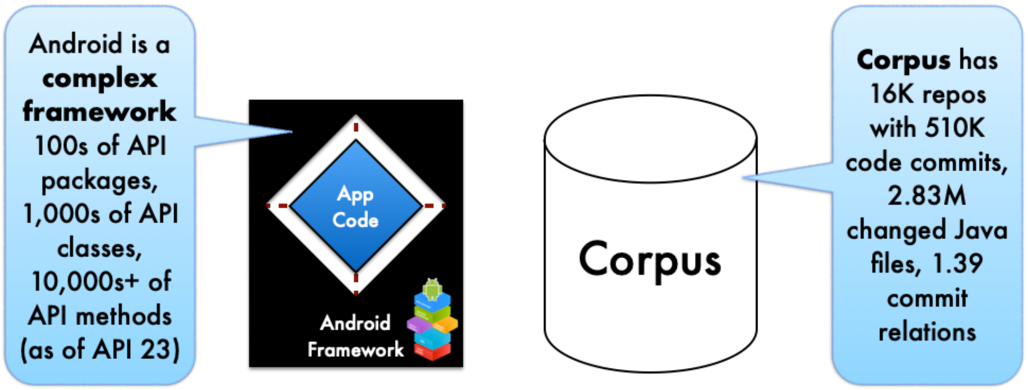

Android is a complex framework 100s of API packages, 1,000s of API classes, 10,000s+ of API methods (as of API 23)
App Code
Android Framework
Corpus
Corpus has 16K repos with 510K code commits, 2.83M changed Java files, 1.39 commit relations

Figure 1: Mining at Scale and with Relevance (Challenge 1)

Challenge 2: Understanding Complex Event-Driven Framework Behavior. These frameworks are often insufficiently documented and lead to serious protocol usage bugs that are common across many usages of the same function within a single project in a software repository, as well as across many projects in the software repository as a whole. The event-driven framework uses callbacks to notify the application of events and the application uses callins to affect how the framework invokes future callbacks. As illustrated in Figure 2, event-driven frameworks with callbacks make understanding the possible behaviors of the app particularly challenging, in that possible next events are determined by a dynamic interaction between the app and the framework. Callback ordering constraints are not static in the way they are typically documented.

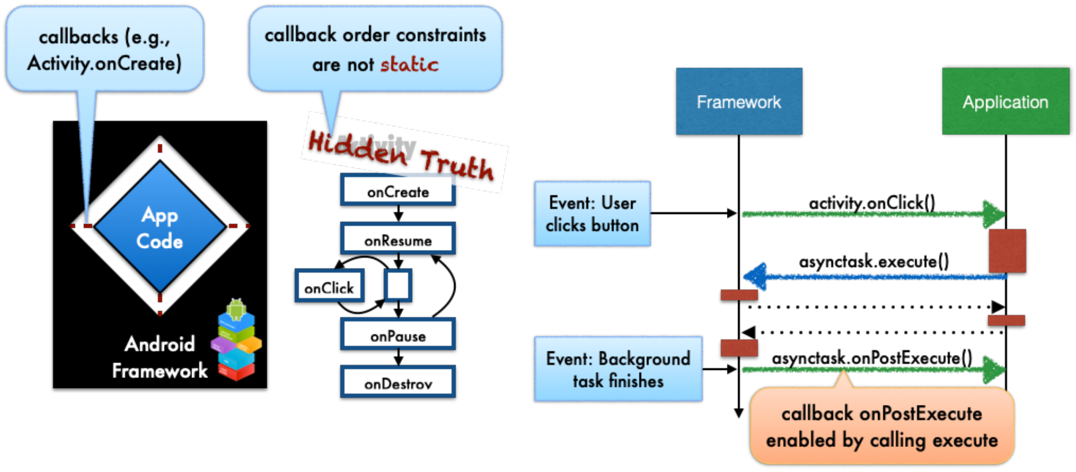

callbacks (e.g., Activity.onCreate)
callback order constraints are not static
Hidden Truth
App Code
Android Framework
onCreate
onResume
onClick
onPause
onDestroy
Framework
Application
Event: User clicks button
activity.onClick()
asynctask.execute()
Event: Background task finishes
asynctask.onPostExecute()
callback onPostExecute enabled by calling execute

Figure 2: Understanding Complex Event-Driven Framework Behavior (Challenge 2)

The Fixr effort consists of multiple research thrusts that investigate how to realize the process of transferring fixes. Such a transfer relies on multiple areas: (a) mining, specifying, and succinctly representing API usage patterns, especially for a complex asynchronous API such as Android; (b) understanding how to use these patterns to detect potential misuses and search for fixes that can resolve these misuses in a large corpus of Android source code; and (c) engineering a pipeline that can fetch source code, parse and maintain them in a searchable database as the code in the repositories evolve over time, in order to support both mining and understanding activities. Figure 3 illustrates Fixr developed novel techniques both for mining relevant patterns and understanding the behavior of those patterns.

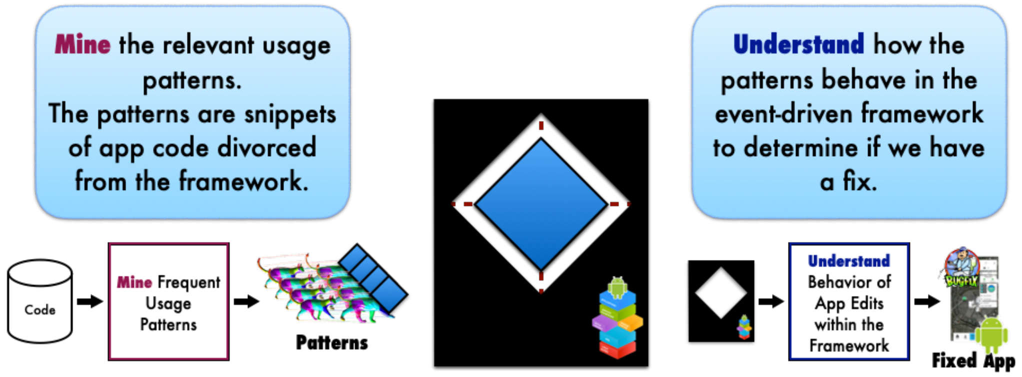
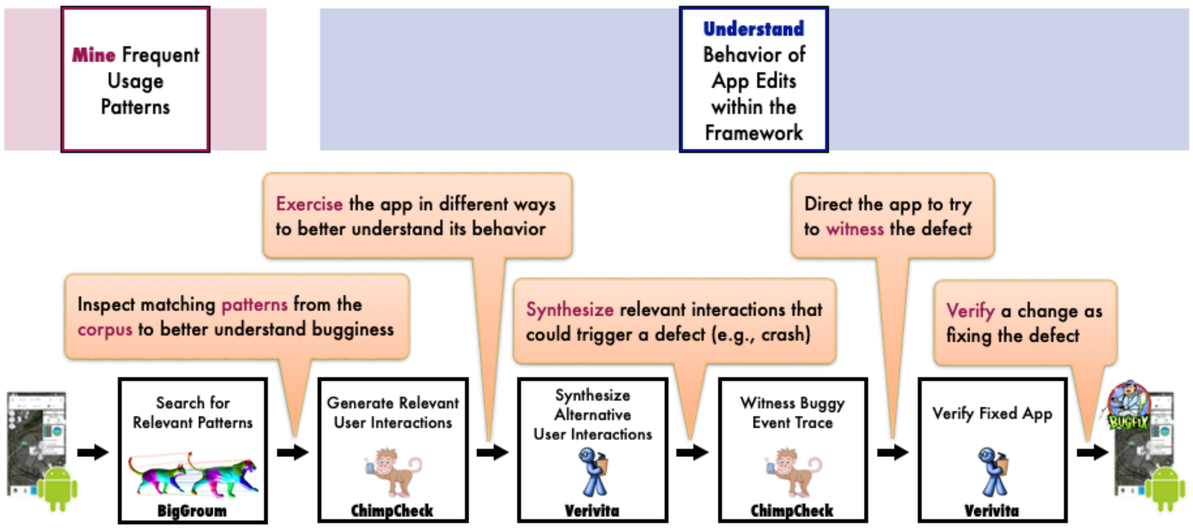

Mine the relevant usage patterns.
The patterns are snippets of app code divorced from the framework.
Understand how the patterns behave in the event-driven framework to determine if we have a fix.
Code
Mine Frequent Usage Patterns
Patterns
Understand Behavior of App Edits within the Framework
Fixed App

Figure 3: Fixr: Mining and Understanding



Mine Frequent Usage Patterns
Understand Behavior of App Edits within the Framework
Exercise the app in different ways to better understand its behavior
Direct the app to try to witness the defect
Inspect matching patterns from the corpus to better understand bugginess
Synthesize relevant interactions that could trigger a defect (e.g., crash)
Verify a change as fixing the defect
Search for Relevant Patterns
BigGroum
Generate Relevant User Interactions
ChimpCheck
Synthesize Alternative User Interactions
Verivita
Witness Buggy Event Trace
ChimpCheck
Verify Fixed App
Verivita
BUGFIX

Figure 4: The Fixr Workflow: Interactively Transfer a Bug Fix

The multiple research thrusts together realize the Fixr Workflow. Figure 4 illustrates the interactive search, test, drive, witness, edit, and verify to transfer a bug fix and imagines the interactive interleaving of mining and understanding activities supported by a suite of tools, enabling the app developer to:

1. Inspect patterns from a corpus to understand a potential bug (with BigGroum[1]);
2. Exercise the app in different ways to better understand its behavior (with ChimpCheck[2]);
3. Synthesize relevant interactions that attempt to manifest the defect (with Verivita[3]);
4. Direct the app to witness the defect (with ChimpCheck); and finally
5. Verify a change as fixing the defect (with Verivita).

As shown in Figure 5, these top-level tools for mining and understanding app-framework protocol defects are supported by a big-data pipeline orchestration services (Fixr Services), user-interactive interfaces (Interactive Fixr), and framework-model learning engines (DroidStar[4]). In the remainder of this section, we summarize some of the key research activities that were undertaken to realize these goals in the Fixr project and the Fixr Workflow.

[1] The University of Colorado Programming Languages and Verification Group (CUPLV). BigGroum. Project Home: http://plv.colorado.edu/biggroum. Sources: http://github.com/cuplv/biggroum
[2] The University of Colorado Programming Languages and Verification Group (CUPLV). ChimpCheck. Project Home: http://plv.colorado.edu/chimpcheck. Sources: http://github.com/cuplv/chimpcheck
[3] The University of Colorado Programming Languages and Verification Group (CUPLV). Verivita. Project Home: http://plv.colorado.edu/verivita. Sources: http://github.com/cuplv/verivita
[4] The University of Colorado Programming Languages and Verification Group (CUPLV). DroidStar. Project Home: http://plv.colorado.edu/droidstar. Sources: http://github.com/cuplv/droidstar

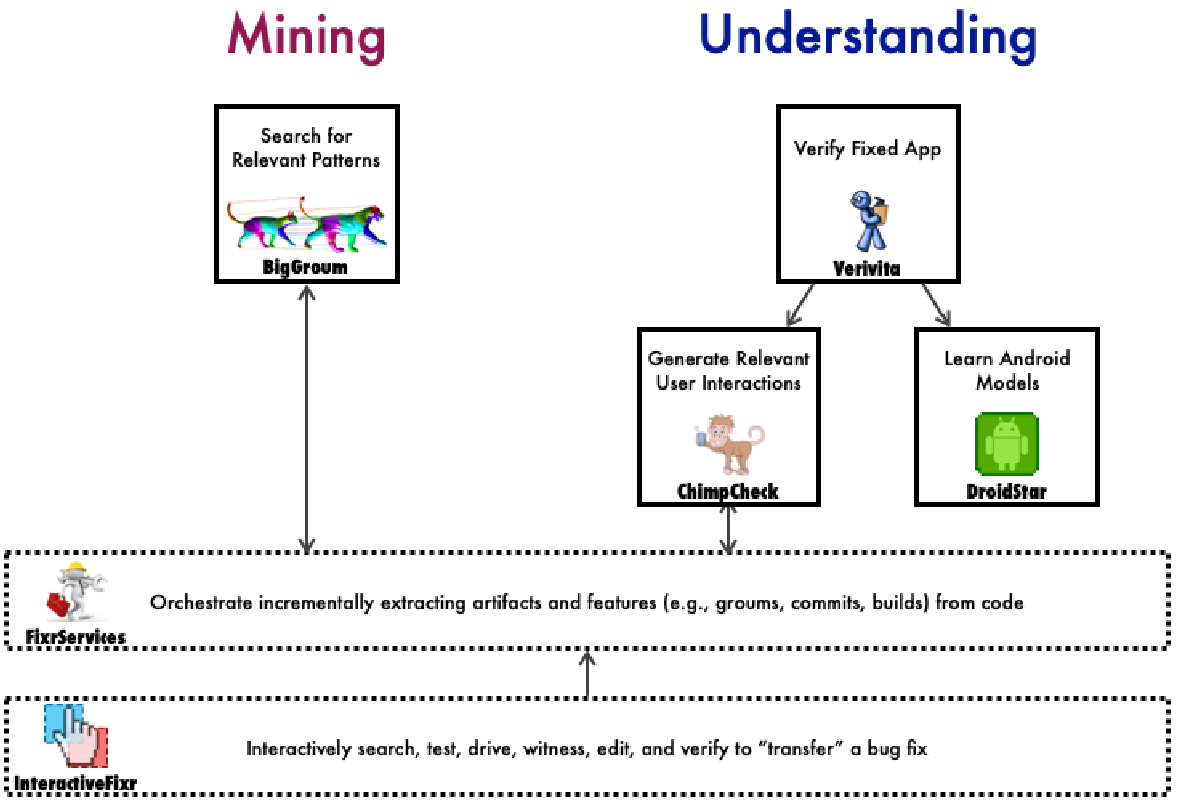

Mining
Understanding
Search for Relevant Patterns
BigGroum
Verify Fixed App
Verivita
Generate Relevant User Interactions
ChimpCheck
Learn Android Models
DroidStar
Orchestrate incrementally extracting artifacts and features (e.g., groums, commits, builds) from code
FixrServices
Interactively search, test, drive, witness, edit, and verify to "transfer" a bug fix
InteractiveFixr

Figure 5: The Fixr Tools

2.1 BigGroum: Mining Framework Usage Graphs from App Corpora

Behind the BigGroum tool [3, 4], a key thrust of the project was investigating data representations of common API usage patterns called a graph-based object usage model (groum) and how common patterns of API usages are located in and mined from a large corpus. A groum is a generalization of a control-data flow graph (CDFG) whose nodes represent API function calls, and the objects that are involved in these calls. The edges represent control and data dependencies between these objects. Thus a groum is a comprehensive description of a set of related API function calls that share a common purpose or idiom for an API.

Modern software libraries and frameworks expose their functionalities to programmers via well-defined APIs. For example, Java APIs are a set of packages containing classes exposing public methods. A programmer needs to understand the behavior of the APIs to write *correct* and *secure* code. This learning task is challenging and error prone because the documentation of a complex library is inevitably incomplete (e.g., the documentation of an API almost never describes its internal behavior, like the correct order of method invocations) and ambiguous (e.g., documentation is written in natural language). An alternative source of documentation for popular APIs like the Java standard library and the Android framework are the *example* programs that use the APIs. App code represents the application programmer's knowledge about an API and can often be used by another programmer to infer the API's behavior. One of the main challenges of exploiting such "documentation" is that each program represents a single use of the API and does not fully capture the API's behavior. The source code found "in the wild" on a single software repository may also represent an incorrect use of the API. While the extraction of the API's usages from existing code corpora seem promising, one needs to consider a large number of examples to learn "general" correct API usage.

We focused on the problem of mining rich application-programming protocols for complex APIs from source code corpora. In our context, a programming protocol describes how developers should write their code containing external libraries to avoid correctness and security issues. While the API pattern mining problem has been well-studied in the literature (e.g., [15–25]), each proposed technique has an *expressivity* versus *efficiency* tradeoff. To advance the state-of-the-art in API pattern mining, we investigated an efficient approach to mine grooms using a sub-graph isomorphism solver that was built using Boolean satisfiability (SAT) based techniques. However, the key challenge lies in scalability. It is well known that sub-graph isomorphism is a combinatorically hard problem without known polynomial time algorithms. Secondly, in a typical corpus with hundreds of thousands of source files, a pairwise comparison would involve a prohibitively large (approximately 10^{10}) number of comparisons to find common patterns. Finally, the common usage patterns themselves need to be classified and counted efficiently in order to find the most frequent patterns. The main contribution of our API usage mining approach is to propose algorithmic solutions to these problems. Our implementation of the software pattern mining approach has been applied to a large corpus of Android projects downloaded from GitHub.

2.1.1 Syntactic-Semantic Representations of API Usage

Here, we adopt grooms [26] as an expressive representation of API usage that describes (1) the control flow of the API methods called, including control structures such as branches and loops, and (2) the data flow showing how objects returned during a method call are used in another method call. That is, a groom is an abstract control-flow and data dependency graph where nodes are either API method calls or program variables and edges represent control flow between method nodes or data dependencies.

While a groom is a rich syntactic-semantic program representation, the existing approaches restricted their scope to mine groom patterns *in a single program* (or project). That is, the problem of identifying the API usage from a relatively small number of developers, already working on the same source code base has been previously considered.

With BigGroum, we are instead interested in programming patterns that are highly conserved across a large corpus of heterogeneous applications. Our underlying hypothesis is that an API usage pattern that is highly conserved across heterogeneous projects represents a "good" or "correct" usage pattern, essentially algorithmically extracting example program-based documentation. Thus we need to scale groom mining to large source code corpora consisting of code from a large volume of a variety of different projects.

> We solve the problem of mining *correct* syntactic-semantic API protocols such as grooms from large corpora of app code from *heterogeneous* projects.

The main challenges when mining grooms from a large, heterogeneous corpus are to mine patterns involving related sets of APIs, as the API methods that should be used together are not known *a priori* and to define (and then validate experimentally) criteria for identifying “correct” and “wrong” API usage patterns. While we have the intuition that patterns that appear “frequently” in a corpus represents the “correct” use for the APIs, it is not clear how we should consider partial instantiations of the “popular” patterns (e.g., missing a method call) or “super” instantiation of the patterns (e.g., calling other API methods together). Furthermore, as noted above, the existing algorithms to mine groums are computationally expensive since they require computing sub-graph isomorphism’s among the candidate patterns and the program representations of the corpus.

2.1.2 Applications and Capabilities of BigGroum

BigGroum by itself can provide useful insights into API usage patterns that are popular and potentially anomalous. We further demonstrate how we solve various developer-oriented tasks exploiting the mined patterns. In particular, we focus on three inter-related tasks:

Similar Source Code Search. Given the developer’s source code, we find the source code in the corpus that uses the API in a similar manner.

Automatic Anomaly Detection. We classify the developer source code as *common* or *anomalous* further providing API patterns to explain this labeling.

Automatic Repair Suggestion. We suggest a set of modifications to the developer’s source code in the case where the code was flagged as anomalous.

Standard workflows for software development are integrated with software repository platforms (e.g., GitHub[5]), where developers collaborate by contributing source code to the same code-base. The software repository platform further provides services for developers to review a version of the code before adopting it in the main code base and to run tasks such as continuous integration (e.g., running regression tests on each new version of the code). A key problem to achieve a successful demonstration for real developers is to integrate BigGroum on a platform where developers collaborate, such as GitHub. The integration on the GitHub platform required developing new software components to automatically process the developer code as soon as a new version was available (e.g., download, compile, analyze statically using BigGroum) and to provide the analysis results to the developer on GitHub.

To demonstrate the BigGroum anomaly detection and repair suggestion capabilities, we have partnered with GitHub and MuseDev[6], which develops a platform integrating static analysis tools in the developer workflow.

2.1.3 Key Contributions of BigGroum.

Our main contributions to API pattern mining in the Fixr project is the BigGroum methodology where:

[5] GitHub. The world’s leading software development platform. https://github.com/
[6] muse.dev. MuseDev | Better code forever. https://muse.dev/

- We developed a pattern mining and classification algorithm for syntactic-semantic API patterns (groums). The algorithm solves the scalability issue that arise with large corpora of heterogeneous code. We also proposed a novel classification of the mined patterns based on the patterns' frequency in the corpora and the patterns relations to identify correct and anomalous API usages.
- We further evaluated the effectiveness of the classification algorithm, validating the hypothesis that "popular" usage of the API usually corresponds to correct usages, and the algorithm's scalability to mine a large corpora of programs.
- We developed a similar source code search, based on finding similar patterns via sub-graph isomorphism. Given the developer source code, we can find mined patterns that use the API in a similar manner. Code search allows a developer to query the pattern database comparing her code to other code that uses the same set of API methods, with the goal of finding differences (i.e., possible anomalies or suggestions on API usage).
- We integrated the BigGroum methodology into the MuseDev platform to demonstrate the viability of transferring this technology into practical applications.
- We released an open-source toolset that implements the BigGroum methodology [5].

2.2 ChimpCheck: Property-Based Randomized Test Generation for Interactive Apps

With the ChimpCheck tool [6, 7], we address the problem of generating *relevant*, user-level, end-to-end test cases for rich interactive applications like Android apps. Testing Android apps to expose relevant behavior is hard. Android apps are complex, stateful pieces of software built on an expansive Android Framework that enables interacting in a rich environment ranging from sensors to cloud-based services. Even in carefully procured mock environments, the app must have a sufficiently large suite of user-interaction event sequences to test its ability to handle the multitude of possible asynchronous events (e.g., button clicks but also screen rotations and app suspensions).

Industrial practice of app testing today can largely be divided into two approaches: (1) brute-force random user interface (UI) testing (e.g., Android UI Exerciser Monkey [27]) and (2) low-level UI scripting of test cases (e.g., Android Espresso [28], Robotium [29]). So-called monkey testing is extremely low effort for the tester but is also undirected—spending lots of time doing irrelevant work. UI scripting, on the other hand, is directed to get relevance executions but often requires substantial effort to develop and maintain. By *relevance*, we mean using application-specific knowledge to exercise the app-under-test in a more sensible way. For instance, to test a music-streaming service app, trying one failed login attempt is almost certainly sufficient.

Then, to exercise the interesting part of the app requires using a test account to get past the login screen to check, for example, that the media-player behaves in a way that users expect for a music service. While rich library support (e.g., Android JUnit, Espresso, and Robotium) and integrated development environment (IDE) integration (e.g., Android Studio) can make custom UI scripting more manageable, implementing test cases one-at-a-time to cover all corner cases of an app is still a tedious process.

From the academic literature, advanced approaches for automating test generation has been a significant focus: model-based techniques (e.g., Android Ripper [30, 31], Dynodroid [32], evolutionary testing techniques (e.g., Evodroid [33]) and search-based techniques (e.g., Sapienz [34]). While each of these techniques easily out performs purely random testing, the development of all of these techniques have almost been entirely focused on total automation. Less attention has been given to developing techniques that simplify programmability and allow higher-levels of customizability that empowers the test developer to inject her app-specific knowledge into the test generation technique. As a result, while these techniques offer the promise for effective automated solutions for testing generic functionality, they are unlikely to replace manual UI scripting because of their omission of the test developer's human insight, app-specific knowledge, and intent.

2.2.1 A New Paradigm for UI Testing

The premise of this work is that we cannot forsake human insight and app-specific knowledge. Instead, we must fuse scripted and randomized UI testing to derive *relevant* test-case generators. While improving and refining automated test generation techniques is indeed a fruitful endeavor, an equally important research thread is developing expressive ways to integrate human knowledge into these techniques.

As a demonstration of this fused approach, we developed ChimpCheck, a proof-of-concept testing tool for programming, generating, and executing property-based randomized test cases for Android apps. Our key insight is that this tension between less relevant but automated and more relevant but manual can be eased or perhaps even eliminated by lifting the level of abstraction available to UI scripting. Specifically, we make *generators* of UI traces available to UI scripting, and we then discover that a brute-force random tester can simply be expressed as a particular generator. From a technical standpoint, ChimpCheck introduces property-based test case generation [35] to Android by making user-interaction event sequences (i.e., UI traces) first-class objects, enabling test developers to express (1) properties of UI traces that they deem to be relevant and (2) app-specific properties that are relevant to these UI traces. From a test developer's perspective, ChimpCheck provides a high-level programming abstraction for writing UI test scripts and deriving app-specific test generators by integrating with advanced test generation techniques—all from a single and simple programming interface. Furthermore, regardless of the underlying generation techniques used, this integrated framework generates a unified representation of relevant test artifacts (UI traces and generators), which effectively serves as both executable and human-readable specifications.

2.2.2 Key Contributions of ChimpCheck

Our main contributions to app test generation in the Fixr project is the ChimpCheck methodology where:

- We formalized a core language of user-interaction event sequences or *UI traces* [6]. This core language captures what must be realized in a platform-specific test runner. In ChimpCheck, the execution of UI traces is realized by the ChimpDriver component built on top of Android JUnit and Espresso. This formalization provides the foundation for generalizing property-based randomized test generation for interactive apps to other user-interactive platforms (e.g., iOS, web apps).

- Then, building on the formal notion of UI traces, we defined *UI trace generators* that lifts scripting user-interaction event sequences to scripting sets of sequences—potentially infinite sets of infinite sequences, conceptually. This lifting enables the seamless mix of scripted and randomized UI testing. It also captures the platform-independent portion of ChimpCheck that compiles to the core UI trace language. This component is realized by building on top of ScalaCheck, which provides a means of sampling from generators.
- Driven by case studies from real Android apps and real reported issues, we demonstrate in Section 4.2 how ChimpCheck enables expressing customized testing patterns in a compact manner that direct randomized testing to produce relevant traces and targets specific kinds of bugs.
- We released an open-source prototype tool for property-based Android UI testing that implements the ChimpCheck principles [8].

2.3 Verivita: Event-Driven Protocols and Callback Control Flow

With the Verivita tool [1, 12], we consider the critical problem of checking that an app programmed against an event-driven framework respects the required application-programming protocol. In such frameworks, apps implement *callback* interfaces so that the app is notified when an *event* managed by the framework occurs (e.g., a UI button is pressed). The app may then delegate back to the framework through calls to the API, which we term *callin* by analogy to callback. To develop working apps, the programmer must reason about hidden *callback control flow* and often implicit asynchronous programming protocols. The same asynchronous, implicitly defined, control flow that makes it difficult for the app developer to reason about his app is also what makes verifying the absence of such protocol violations hard.

2.3.1 Lifecycle Automata are Insufficient for Modeling Callback Control Flow

Lifecycle automata are the common representation used to model callback control flow that is both central to Android documentation [36, 37] and prior Android analysis techniques—both static and dynamic ones (e.g., [38–40]). In Figure 6, we show a lifecycle automaton for the Activity class of the Android framework. The black, solid edges are the edges present in the Android documentation [36] showing common callback control flow. These edges capture, for example, that the app first receives the onStart callback before entering a cycle between the onResume and the onPause callbacks. But this clean and simple class-based model quickly becomes insufficient when we look deeper.

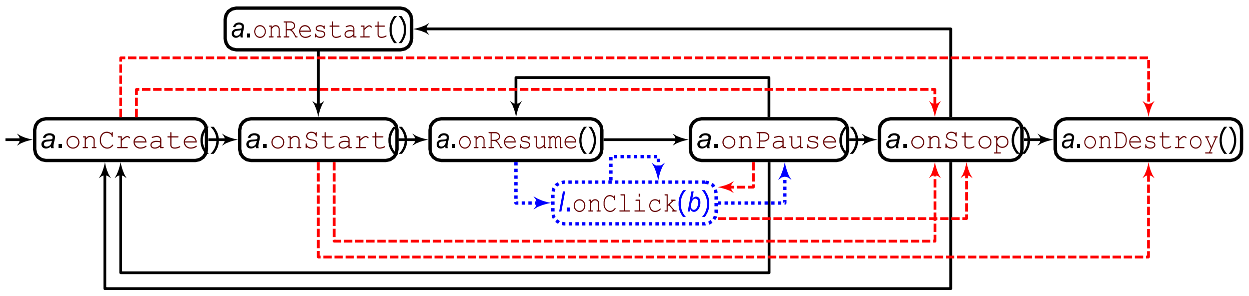


Figure 6: The Activity Lifecycle Automaton

First, there are complex relationships between the callbacks on "related" objects. For example, an OnClickListener object l with an onClick callback may be "registered" on a View object v that is "attached" to an Activity object a. Because of these relationships, the callback control flow we need to capture is somewhat described by modifying the lifecycle automaton for Activity a with the additional blue, dotted edges to and from onClick (implicitly for OnClickListener l) in Figure 6. This modified lifecycle encodes framework-specific knowledge that the OnClickListener l's onClick callback happens only in the "active" state of Activity a between its onResume and onPause callbacks, which typically requires a combination of static analysis on the app and hard-coded rules to connect callbacks on additional objects such as OnClickListeners to component lifecycles such as Activity. We refer to such callback control-flow models based on such refined lifecycle automatons as lifecycle^{++} models.

Second, there are less common framework-state changes that are difficult to capture soundly and precisely. For example, an analysis that relies on a callback control-flow model that does not consider the intertwined effect of an a.finish() call may be unsound. The red, dashed edges represent callback control flow that are not documented (and thus missing from typical callback control flow models). Each one of these edges specifies different possible callback control flow that the framework imposes depending on *if and when* the app invokes the finish callin inside one of the Activity's callbacks. Of course, the lifecycle automaton can be extended to include these red edges. However, this lifecycle automaton is now quite imprecise in the common case because it does not express precisely when certain callback control-flow paths are spurious (i.e., depending on where finish is not called). Figure 6 illustrates why developing callback control flow models is error prone: the effect of calls to finish are subtle and poorly understood.

It is simply too easy to miss possible callback control flow. While lifecycle automata are useful for conveying the intuition of callback control flow, they are often insufficiently precise and easily unsound.

2.3.2 Key Contributions of Verivita

Our main contributions to modeling and reasoning about event-driven callback control flow in the Fixr project is the Verivita methodology where:

- We gave a careful re-examination of callback control flow models. In prior work, modeling callback control flow was almost always a secondary concern in service to, and often built into, a specific program analysis where the analysis abstraction may reasonably mask unsound callback control flow. Instead, we consider modeling callback control flow independent of any analysis abstraction—we identify and formalize the key aspects to effectively model event-driven application-programming protocols at the app-framework interface, such as the effect of callin and callback invocations on the subsequent callback control flow. The main result of such a theoretical framework is a specification language and methodology for callback control flow models called *lifestates* that we can *validate* for soundness against real execution traces from the event-driven framework implementation.

- With the lifestate methodology, we further defined the dynamic lifestate verification problem: given an app-framework interaction trace and a lifestate model, dynamic lifestate verification attempts to prove the absence of a rearrangement of the recorded events that could cause a protocol violation. Rearranging the execution trace of events corresponds to exploring a different sequence of external inputs and hence discovering possible protocol violations not observed in the original trace.
- We conducted a thorough experimental evaluation validating the hypotheses that lifecycle models, by themselves, are insufficiently precise to verify Android apps as conforming to the specified protocols, that model validation on large corpora of traces exposes surprising unsoundnesses, and that lifestates are indeed useful.
- We released an open-source toolset [13, 14] that implements application-execution trace instrumentation, lifestate model validation, and dynamic lifestate verification that demonstrates the Verivita methodology.

2.4 Fixr Services and Interactive Interfaces

A core part of the Fixr infrastructure is the persistence layer where Android app code repositories, the features extracted from them, and all derived information products (such as groums) are stored. The Fixr architecture features a centralized store of information that allows each of the major Fixr components to access any of the artifacts it needs. In actuality, the Fixr repository was heterogeneous and made use of multiple storage mechanisms across the life of the project.

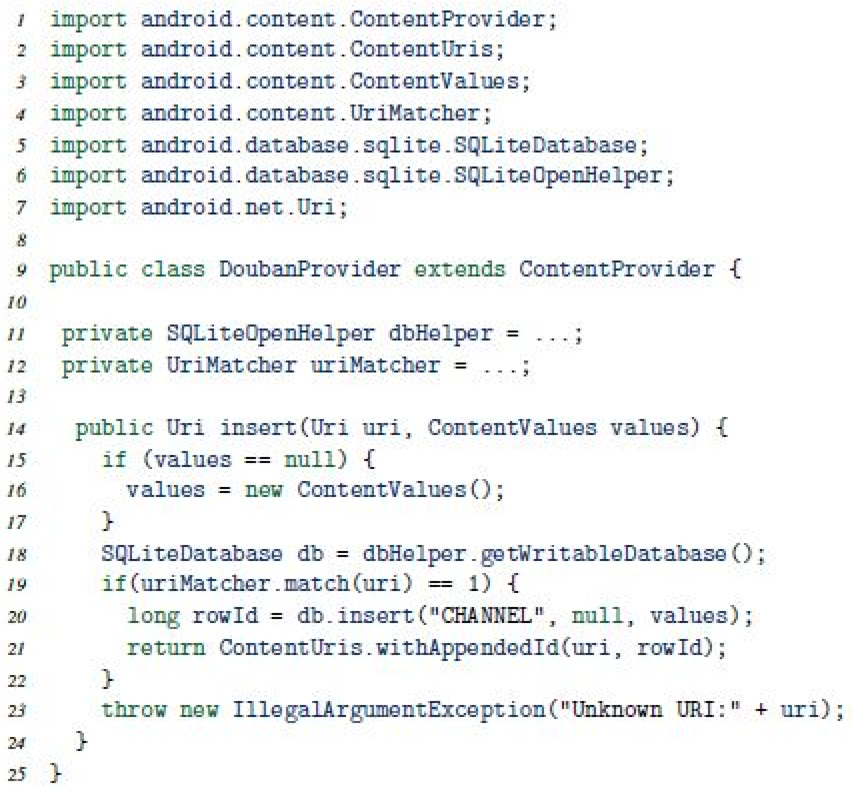
A closely related aspect of the storage layer are the systems that download large numbers of Android repositories and extract the features needed by the other Fixr components from the software contained in those repositories. The approach we took to download, process, generate, and store the repositories and their associated features evolved significantly over the life of the project. We present the choices that we made related to the storage layer and these collection and generation tasks in the sections to follow.

Finally, after artifacts—Android repositories, extracted features, and derived information—are derived and stored in the storage layer, we want to make it possible for users, which are software developers, to easily access, interact with, analyze, and make sense of those artifacts. In the subsequent sections, we present the design and development of Interactive Fixr, a web-based interface that allows software developers to (a) browse code snippets, (b) view API usage trends, (c) analyze code commits, and (d) query API call patterns, to support their tasks to identify, understand, and fix bugs.

3 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Mining Rich Syntactic-Semantic Protocol Usage Patterns with BigGroum

Consider being the developer of the Douban-FM-sdk Android app [41] and receiving complaints about the app being slow. By debugging the app, the developer discovers that the performance issue is due to high memory usage, and the executions that have an anomalous memory consumption are the ones that invoke operations on the DoubanProvider class. In turn, this class heavily uses database operations from the SQLiteOpenHelper Android framework package. In Figure 7, we show the code of the insert method from the DoubanProvider class that uses such database operations. The insert method inserts new values in the "CHANNEL" table only if the uniform resource identifier (URI) uri passed as input is a channel. In the method, the developer opens a database connection but he forgets to close it; not explicitly closing the database connections may retain resources during the execution of the Android app (as documented on StackOverflow [42, 43]).



```java
import android.content.ContentProvider;
import android.content.ContentUris;
import android.content.ContentValues;
import android.content.UriMatcher;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.net.Uri;

public class DoubanProvider extends ContentProvider {

 private SQLiteOpenHelper dbHelper = ...;
 private UriMatcher uriMatcher = ...;

  public Uri insert(Uri uri, ContentValues values) {
    if (values == null) {
      values = new ContentValues();
    }
    SQLiteDatabase db = dbHelper.getWritableDatabase();
    if(uriMatcher.match(uri) == 1) {
      long rowId = db.insert("CHANNEL", null, values);
      return ContentUris.withAppendedId(uri, rowId);
    }
    throw new IllegalArgumentException("Unknown URI:" + uri);
  }
}
```

Figure 7: An Open-Source Android App Showing Buggy API Usage

At a first glance, the code shown in Figure 7 seems correct and the developer is confused trying to understand how the API should be used to avoid the issue. At this point, the developer *compares* his code with the code of other developers that used the same API, trying to understand the behavior of the API, and how to use it correctly. This practice is enabled by the access to large code repositories (e.g., GitHub has more than 31 million developers and 96 million repositories [44]). The implicit assumption of the developer is that other developers that used the same API could have a better understanding of the behavior of the API or have experienced and fixed similar issues. For example, the developer of the code in Figure 7 may search for Java code on GitHub that use the methods getWritableDatabse, from the class SQLiteOpenHelper, and insert, from the class SQLiteDatabase.[7] Then, the developer compares his code with different source code examples returned by the search, trying to find discrepancies that can explain the performance issue, and hopefully gets insights needed to fix the issue. Eventually, the developer finds several source code examples that open a database connection and invoke the insert methods. The developer also believes that most of the code examples further contain the invocation to the close method. And indeed the bug in the code from Figure 7 is that it does not explicitly close the database connections. As a consequence, the database connection uses the system's resources during the execution of the Android app [42, 43]. In particular, during this process the developer believes the particular API usage to be correct if several developers have adopted it, while he trusts less source code examples that do not appear as often in the search.

API pattern mining algorithms try to automate this workflow to find such kind of usages from existing source code and synthesize a *pattern* describing them. While there are works that mine API usage patterns for a domain-specific library from a single program (e.g., [26]), we focus on the problem of mining API usage patterns for a large framework (like Android) from a corpora of applications (i.e., from a large corpora of applications written by many different developers). We are interested in validating the hypothesis that for this API mining problem, we can infer correct API usage patterns by mining for the *popular* patterns–those seen more frequently in the corpus. In particular, we try to answer the following question:

> Can we infer the correct API usage patterns for a complex framework by mining the popular patterns across a large volume and variety of applications?

If such hypothesis is true, we can use such popular patterns as documentation or to compare-and-contrast with new developer code. Similarly, we ask if we can infer buggy API usage patterns by mining for anomalous patterns—those that occur very rarely in the corpus.

In order to answer the previous research questions, we developed BigGroum, a methodology to mine the popular and anomalous APIs patterns *from different apps*. We then use BigGroum to evaluate the hypothesis that popular patterns are "correct" and that anomalous patterns are "buggy." As described in Section 2.1, we use graph-based object usage models (groums) to represent API usage in both the source code and in a pattern. In Figure 8, we show the groum for the code in Figure 7. The groum captures in the nodes all of the API method invocations (e.g.,

    db = dbHelper.getWritableDatabase())

and connects a method invocation to another (with a solid, black edge) when that method invocation may happen before. The groum represents that the db =

[7] An example of such a search queries is https://github.com/search?l=Java&p=2&q=SQLiteOpenHelper+getWritableDatabase+SQLiteDatabase+insert&type=Code

dbHelper.getWritableDatabase() is executed before rowId = db.insert(''CHANNEL'', null, values). Furthermore, a groum represents how the data is "produced and consumed" by API calls. The oval-shaped nodes (e.g., ContentValues values) represents an item of data (e.g., a variable defined in the program or a constant). A data node has a green edge to a method node when the method node uses that data—a use edge. For example, db = dbHelper.getWritableDatabase() uses the variable dbHelper. An edge in the other direction represents that the variable has been assigned by the invocation of the method–a definition (def) edge. The node values = new ContentValues() defines the variable values. Groums abstract all the app-specific (i.e., non-API defined) code.

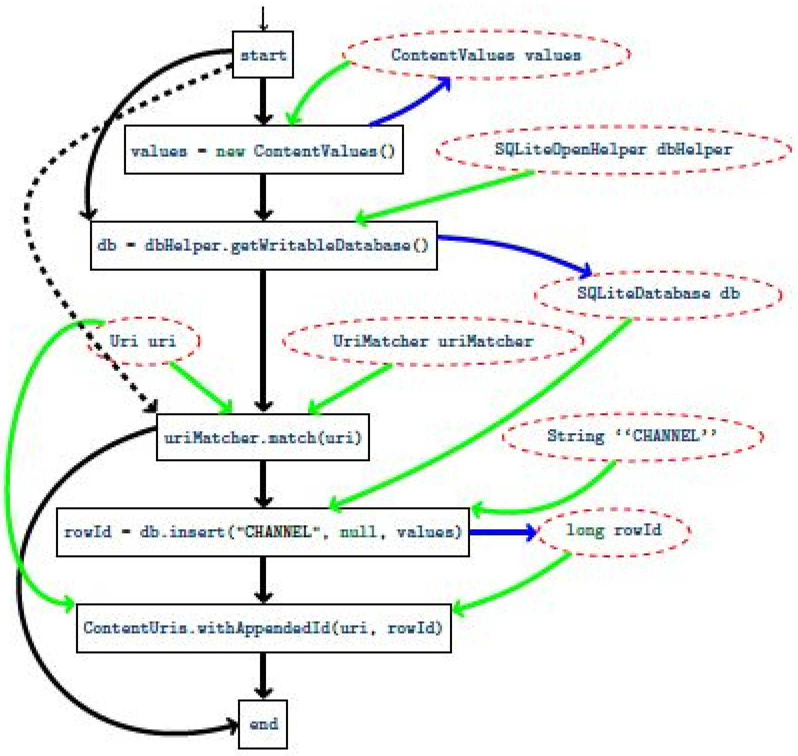

start
ContentValues values
values = new ContentValues()
SQLiteOpenHelper dbHelper
db = dbHelper.getWritableDatabase()
SQLiteDatabase db
Uri uri
UriMatcher uriMatcher
uriMatcher.match(uri)
String ''CHANNEL''
rowId = db.insert("CHANNEL", null, values)
long rowId
ContentUris.withAppendedId(uri, rowId)
end

Figure 8: The Groum Representing API Usage from Figure 7

We frame the problem of mining patterns of API usages from a corpora of Android applications as the problem of mining groum patterns from the groum dataset representing the source code in the corpora. For each method declaration in the corpora, we obtain a groum (e.g., as for the insert method from Figure 7 and its groum shown in Figure 8) and hence the input for the mining algorithm is a collection of groums. Then, the mining problem is to construct a set of popular and anomalous groums. We consider a "frequentist" approach where we define if a pattern is popular or anomalous depending on the number of groums in the corpora that support such a hypothesis.

In Figure 8, the dashed, oval nodes (*data nodes*) represent data (variables and constants) used by or assigned by the invocation of an API method. The solid, rectangular nodes (*method nodes*) represent the invocation of an API method; solid black edges (*control edges*) between two method nodes are control flow edges (i.e., the program can invoke the method in the source node of the edge before invoking the method in the destination node of the edge). The dashed black edges (*transitive edges*) represents the transitive closure of the control flow edges—the figure shows only one transitive edge for readability; a blue edge (*def edge*) from a method node to a data node indicates that the invocation of the method in the method node assigned the value of the variable in the data node; a green edge instead (*use edge*) indicates that a method invocation uses a specific data value.

The problem is similar to the mining problem of Nguyen et al. [26], with the main difference that we mine patterns for the usage of the same API across multiple repositories. These settings are more challenging since the code utilizing the same API methods is developed: (1) in different contexts and by different developers; and (2) in a large number of repositories. Our approach tries to overcome both of these challenges.

3.1.1 Clustering Groums Using Similar Functionalities.

The code in Figure 7 uses API methods from the android.content, android.database, and android.net packages. The classes from android.content are related to the database API but only some of the classes, and hence methods, are relevant to describe how to properly open and close a database. For example, the pattern that describes how to manage a database connection will not include the method match from the UriMatcher class even if it is from the android.content package.

For this reason, we find the methods that are frequently invoked together in the groum dataset. This data mining problem is well known and is called *frequent itemset mining* [45]. The frequent itemset mining produces a collection of sets of method invocations. For example, suppose one of the frequent itemsets contains the methods invocations getWritableDatabase (from SQLiteOpenHelper), methods insert and close (both from SQLiteDatabase), and put (from ContentValues). In a nutshell, this set means that getWritableDatabase, insert, close, and put are frequently invoked together in the source code from the corpus.

We use the frequent methods invoked together to *cluster* the groums that use (all) such methods. In practice, we create (non-disjoint) subsets of groums, each one containing all and only the groums that invoke all the frequent methods. After all, if two groums do not use the same set of frequent methods, then they cannot contribute to the same frequent patterns.

We further simplify the groums contained in the cluster. Consider again the groum of Figure 7 and the frequent itemset getWritableDatabase, insert, close, and put. The method nodes of the groum that are not in this set (e.g., match) are not important for the mining process since they will never contribute to a popular pattern. Thus, BigGroum cleans the data by *slicing* each groum, as shown in Figure 9 on the left for the groum from Figure 8. The slicing process first removes the method nodes that are not in the frequent itemset and then removes any dangling edges and nodes. The effect of such clustering and slicing is to create a reduced dataset of groums where unrelated methods are not considered together. Then, we mine the patterns from *each cluster* individually. Clearly, the dataset we are starting from after each clustering and slicing is smaller than the original corpora, since it contains less groums, and "easier," since each groum is sliced with respect to the set of important methods.

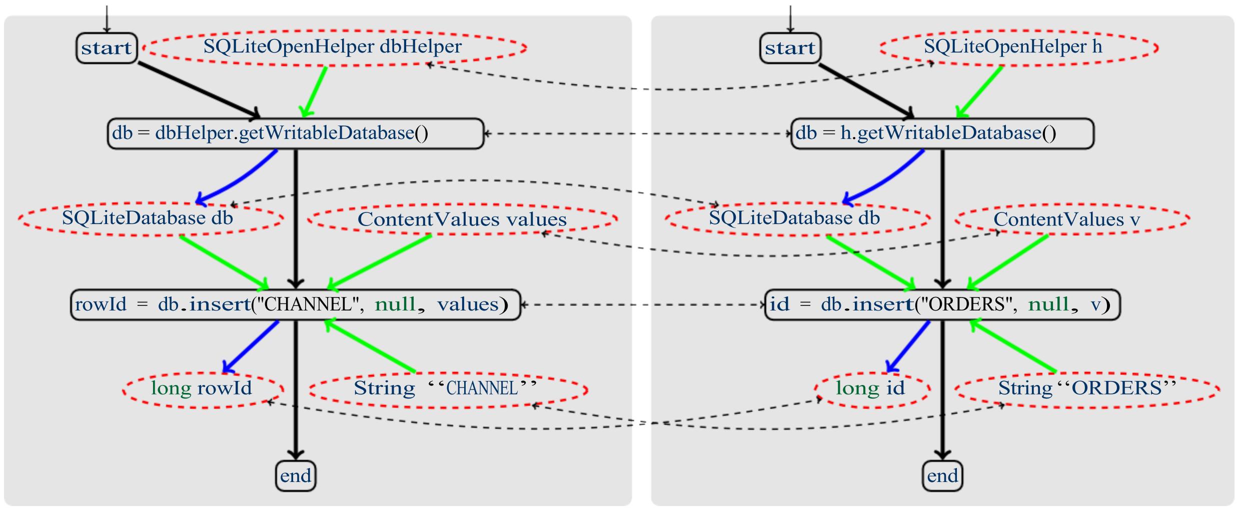

start
SQLiteOpenHelper dbHelper
db = dbHelper.getWritableDatabase()
SQLiteDatabase db
ContentValues values
rowId = db.insert("CHANNEL", null, values)
long rowId
String ‘‘CHANNEL’’
end
start
SQLiteOpenHelper h
db = h.getWritableDatabase()
SQLiteDatabase db
ContentValues v
id = db.insert("ORDERS", null, v)
long id
String ‘‘ORDERS’’
end

Figure 9: An Isomorphism Relation between Two Grooms

In Figure 9, the groum is sliced using the methods getWritableDatabase, insert, close, and put, as shown in the *Left Graph,* for slicing the one from Figure 8. Corresponding to the buggy code, the *Right Graph,* is a groum isomorphic to the left one. The *graph isomorphism relation* for each node of the left groum is mapped to one, and only one, node of the right groum and vice-versa. The mapping is shown with black dashed arrows. We do not show the mapping on the edges, even though the isomorphism also maps them.

3.1.2 Mining API Usage Patterns via Lattice Construction.

The mining and classification of pattern from a dataset of groums proceeds with three subsequent phases:

Finding Identical API Usage Patterns. The input of the mining task is a dataset of groums that have already been sliced with respect to a set of methods. The goal of the mining process is to find the “popular” and “anomalous” API usages. Consider the frequent itemset consisting of getWritableDatabase, insert, close, and put as in Figure 9. Each groum in the dataset is a candidate pattern for the use of this set of API methods. Our goal is to find the API usages that occur more frequently in the dataset. Thus, we first find how many instances we have for each single API usage pattern. That is, we count how many times we find the same groum in the dataset. Clearly, groums extracted from different source code repositories will be syntactically different (e.g., they use different variable names). We use *graph isomorphism*, a bijection between the nodes and edges of two graphs, as an equality operator between groums. For example, the two groums shown in Figure 9 are isomorphic, even if the variable names are different. In the figure, we show the isomorphism mapping between nodes as a doubly-headed, black-dashed, arrow (we do not show the isomorphism on edges for readability). Notice that the isomorphism is aware of the kinds of the nodes and edges. For example, a method invocation node can only be mapped to another method invocation node and similarly, a variable def edge can only be mapped to another def edge. The result of the first step of the mining process is shown in Figure 11a. The figure represents the “lattice structure” produced during the mining process. In Figure 11a, each rectangle represents a bin of groums from the dataset. All the identical groums are grouped in the same bin and the blue number on the right of the bin is the bin's cardinality. In the example shown in Figure 11a, the bin B_1 contains the isomorphic groums from Figure 9, bin B_2 the left groum from Figure 10, and bin B_3 the right groum from Figure 10.

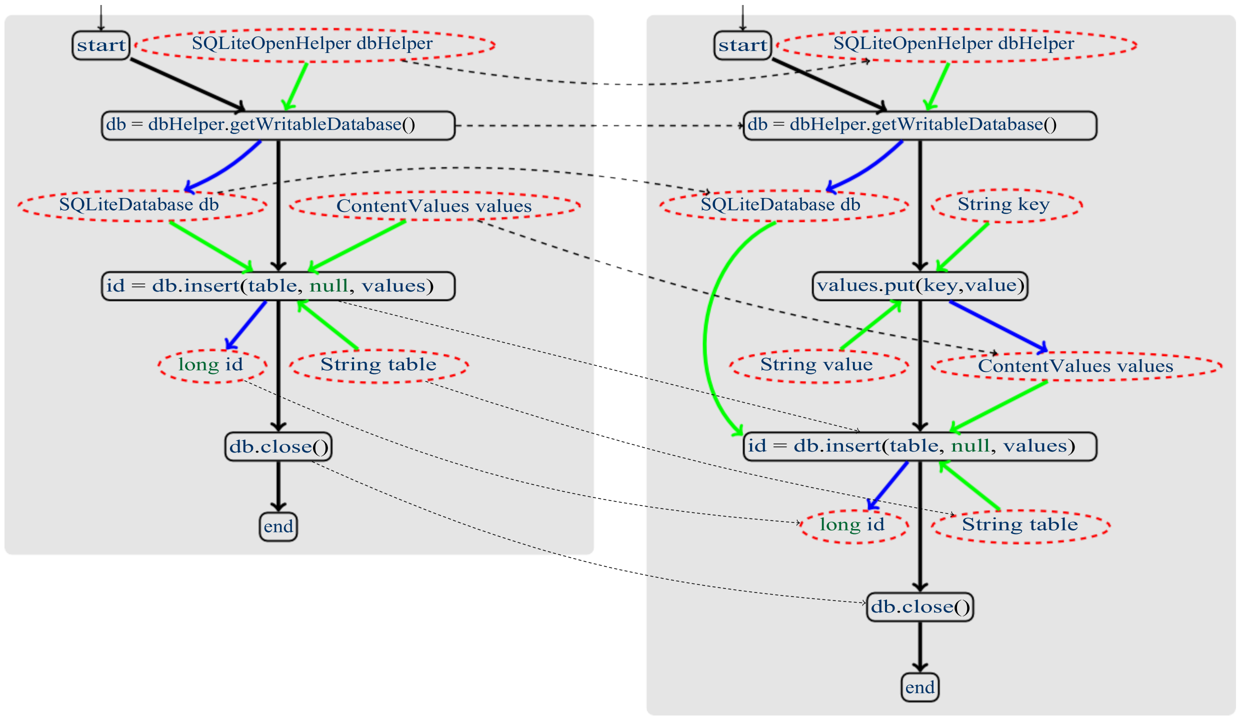
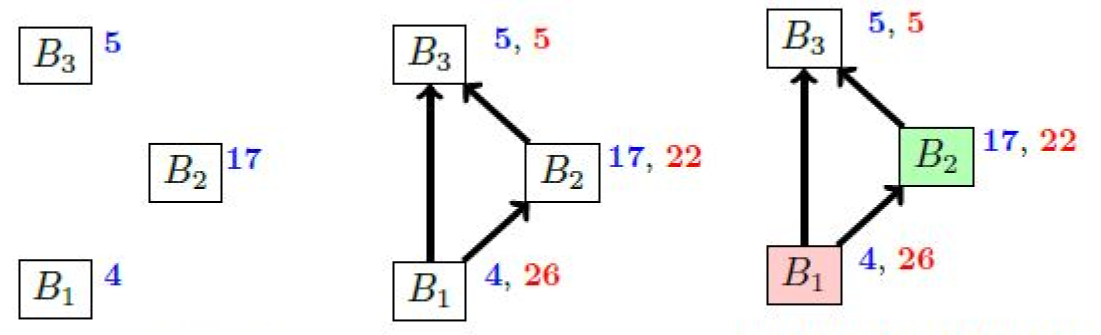


Figure 10: An Embedding Relation between Two Groums

In Figure 10, the *Left Graph* shows the groum representing a popular usage of the SQLiteDatabase Object, including the final call to close. The *Right Graph* shows the groum including an additional call to the put Method from ContentValues. The *Graph Embedding Relation* shows each node of the left groum is mapped exactly to one node of the right groum; the mapping is shown with black dashed arrows.

Computing API Usage Pattern Relations. At this stage, each bin just represents how many times we saw a specific pattern in the dataset. However, a pattern may appear inside other patterns. In that case, more examples from the dataset contain the pattern and should be accounted for when considering the popularity of the pattern. For example, in Figure 10 the left groum "appears again" inside the groum on the right. We consider *graph embedding* as a containment relation between two groums. Intuitively, a groum is embedded in another groum if there is a mapping from all its nodes and edges to other nodes and edges in the other graph. In Figure 10, we show an embedding between two groums. We use the groum embedding relations to construct a lattice where the elements are bins of isomorphic groums. In the lattice in Figure 11b, we represent that a bin (i.e., an equivalence class of isomorphic groums) is embedded in another bin with a direct edge. In Figure 11b, we further see that bin B_2 is embedded in bin B_3 and bin B_1 is embedded in B_2 (and, by transitivity of the embedding, B_1 is embedded in B_3). Moreover, in this phase we compute the frequency of a pattern (shown in the red number in Figure 11b). Given the lattice structure, we compute the popularity for a bin B summing all the cardinalities of the bins subsuming B.



(a) Lattice After Identifying the Identical Groums

(b) Lattice with the Containment Relations

(c) Lattice After Pattern Classification, Using $f = 20$ as the Frequency Threshold

Figure 11: Phases of Mining for Popular and Anomalous APU Usage Patterns

Figure 11 shows how we mine a dataset of sliced groums *(a)* identifying the identical (up to isomorphism) groums, *(b)* find the containment relations among patterns, and *(c)* classify the patterns according to their popularity. The figure represents the resulting lattice data structure after each phase, where each rectangle is a bin representing an equivalence class of groums with the blue number on the right of a bin being the cardinality of the bin (i.e., the number of groums in the equivalence class) and the red number the frequency of the bin (i.e., the total number of times the pattern appeared in the dataset, also considering subsuming groums). The solid, black lines represent an embedding relation between bins; the red bin in *(c)* is classified as an ANOMALOUS pattern, while the green bin is a POPULAR pattern. The bin B_1 contains the groums from Figure 9, the bin B_2 contains the left groum from Figure 10, and the bin B_3 contains the right groum from Figure 10.

Classifying the API Usage Patterns According to Popularity. In the final step, we classify the patterns depending on their popularity. We consider a pattern POPULAR if two conditions hold. First, several developers frequently use the pattern (we set a threshold f that we compare to the frequency of the bin). Then, we require that no bin subsuming the pattern's bin is popular. Consider Figure 11c and 20 as a frequency threshold f. The bin B_2 is then a POPULAR pattern, while B_3 is not. We then classify patterns that represent anomalous usages. We call usages that are rare, below a fixed threshold L, ISOLATED, which we consider a pre-condition for incorrect API usages. If a rare usage is then also embedded in a POPULAR pattern, then it represents a violation of a pattern used frequently that we consider ANOMALOUS. In Figure 11c, the bin B_3 is an ANOMALOUS pattern. The final output of the mining process is then a list of POPULAR and ANOMALOUS (as well as ISOLATED) patterns.

3.1.3 Summary of the BigGroum Mining Process.

In Figure 12, we summarize the phases of the BigGroum approach, where White, Green, Red, Yellow colored bins represent unlabeled, POPULAR, ANOMALOUS, and ISOLATED patterns, respectively:

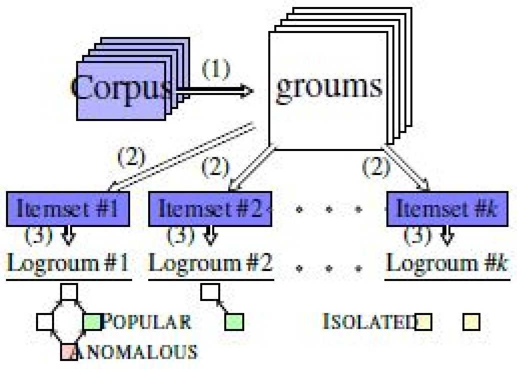


Figure 12: Summary of the BigGroum Mining Process

(1) Each method in the corpus is compiled into a groum (i.e., the approach is, by default, intra-procedural),

(2) The corpus of groums is clustered by using frequent itemset mining in order to perform the pattern computation on subsets of the entire corpus. Frequent itemset mining computes the set of API method calls (itemsets) where the number of groums (i.e., app defined methods) containing all API method calls of the itemsets exceeds some threshold f_I. The corpus of groums can then be clustered based on the computed frequent itemsets. Each itemset selects the groums that share at least some number K_I of API method calls.

(3) The patterns are mined from each cluster of groums.

   a. The groums in each cluster first are binned according to the embedding relation, computed among each pair of groums in the cluster. Isomorphic groums belong to the same bin, and a bin is subsumed by another bin if the groums that it contains are embedded in the groums of the other bin.

   b. The bins in the lattice are labeled POPULAR if the number of groums in the bin exceeds some threshold f; ANOMALOUS or ISOLATED labels are applied to bins below some threshold L subject to whether or not the bin is embedded a popular bin. The main output of BigGroum are lattice-ordered groums (logroums)—the lattices of isomorphic groums in the labeled bins that represent POPULAR, ANOMALOUS, or ISOLATED API usage patterns.

The mined lattices can then be used to search for patterns similar to developer code.

3.1.4 Searching for Buggy API Usages and Synthesizing Bug-Fix Suggestions.

The mined logroum structures are the key ingredient to automatically find anomalous API uses in developer code and provide suggestions to repair those anomalies.

At the high level, the anomaly search takes as input an existing snippet of code and performs the following steps:

1. Extract a query groum q from the input code snippet;
2. Search the mined logroum for an ANOMALOUS pattern g_a identical to the developer

code (using graph isomorphism as a primitive to determine equality);

3. Find the POPULAR pattern g_p subsuming the ANOMALOUS pattern in the logroum; and
4. Produce a list of edit operations that would make the query groum g isomorphic to the popular pattern g_p.

As an example, the BigGroum anomaly search applied to the code in Figure 7 from Douban-FM-sdk [41] would find that the code is identical (isomorphic) to the ANOMALOUS pattern B_1. Furthermore, the mined lattice structure tells that the pattern in the bin B_1 is subsumed by the POPULAR pattern in the bin B_2. The embedding relation between the pattern of bin B_1 and the pattern of bin B_2 (i.e., the sub-graph embedding between the left groum of Figure 9 and the left groum of Figure 10) highlights that the user code misses a call to the `close` method that releases the database resource.

3.2 Proof of Concept: Integrating BigGroum into MuseDev and onto GitHub

Given the promise of the BigGroum approach, we undertook developing an advanced proof of concept (POC) implementation of integrating the BigGroum approach into the MuseDev Platform and onto GitHub—as a demonstration for potential technology transfer. [8] The result of the integration is that a developer on GitHub can use the MuseDev platform to automatically get the list of anomalies and repair suggestions in her code.[9]

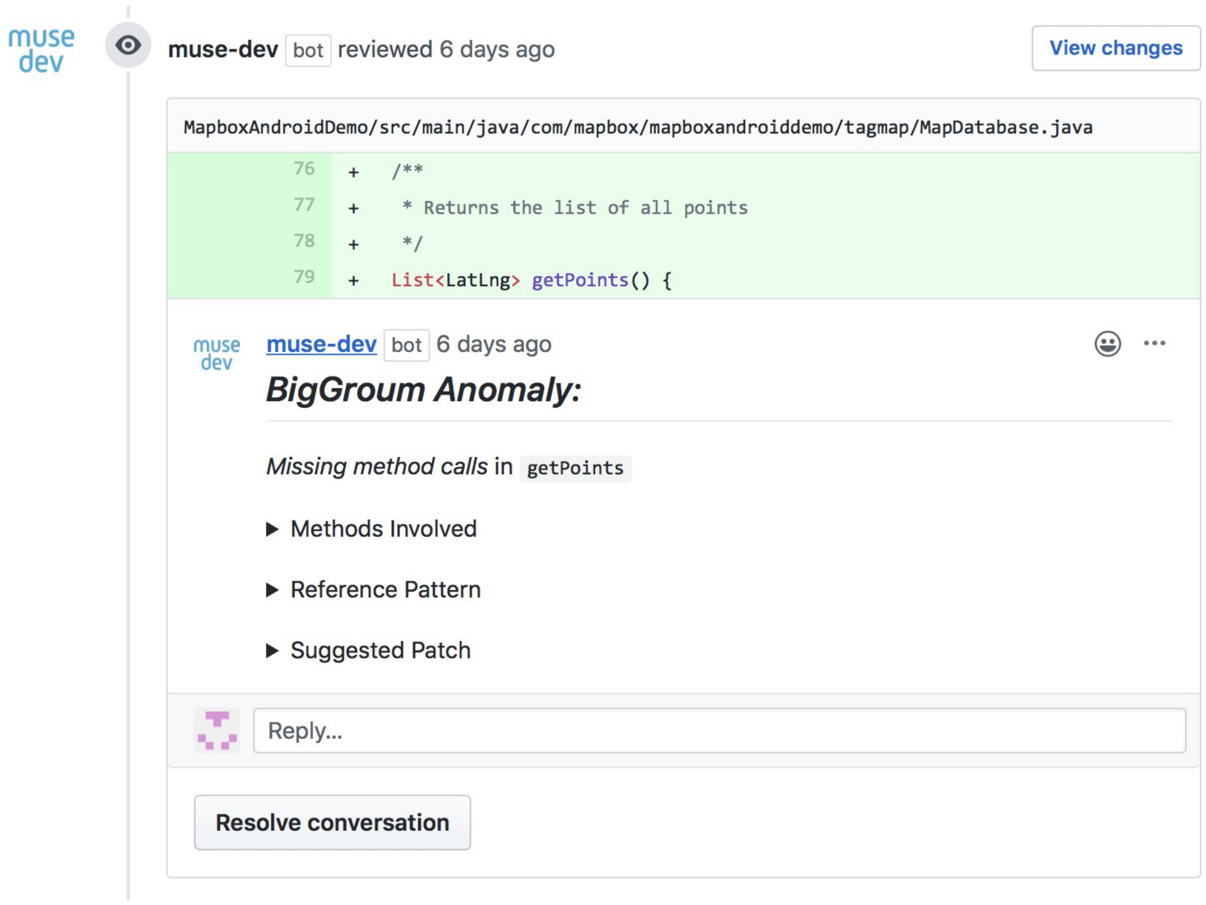
We demonstrated anomaly detection and repair suggestion in the pull request process of GitHub. A developer creates a pull request to integrate his latest changes to a source code project in another version of the existing code base. During a pull request, another developer can review the new code and decide if the new changes can be integrated (i.e., merged) in the existing code base. We integrated the BigGroum analysis in this part of the developer's workflow: when a developer asks for merging his new contributions, BigGroum analyzes the new contributions to find anomalies and propose fixes. In this way, the developer receives automatic feedback and can solve potential issues in her code before merging it, avoiding introducing new bugs in the project.

We illustrated the final result of the integration in the pull request [46]. The pull request integrates an artificial addition to the MapBox application demo [47] for Android by adding a sample Activity to save different positions on a map. The BigGroum tool generates 8 new anomalies, involving Android APIs to work with a database and the Android UI. Figure 13 shows one of the anomaly reports on GitHub, which shows that the `getPoints` method misses invoking some API methods and that the detailed information (i.e., methods involved, reference pattern, and the suggested patch) are hidden in drop-down sections. The anomaly reports that the method `getPoints` misses some method calls and provides the reference pattern that was violated in the anomaly and the suggestion for fixing the anomaly. The anomaly contains detailed

[8] This advanced POC was developed as part of an 18-month extension to the original agreement via "Engineering Change Proposal: Technology Transfer: BigGroum Protocol Pattern Mining and Repair." We, the performer, collaborated separately with GitHub and MuseDev through this extended agreement

[9] At the time of this writing, the MuseDev platform is in private beta and not yet publicly available

information for the developer:



muse-dev bot reviewed 6 days ago

View changes

MapboxAndroidDemo/src/main/java/com/mapbox/mapboxandroiddemo/tagmap/MapDatabase.java

```
+   /**
+    * Returns the list of all points
+    */
+   List<LatLng> getPoints() {
```

muse-dev bot 6 days ago

BigGroum Anomaly:

Missing method calls in `getPoints`

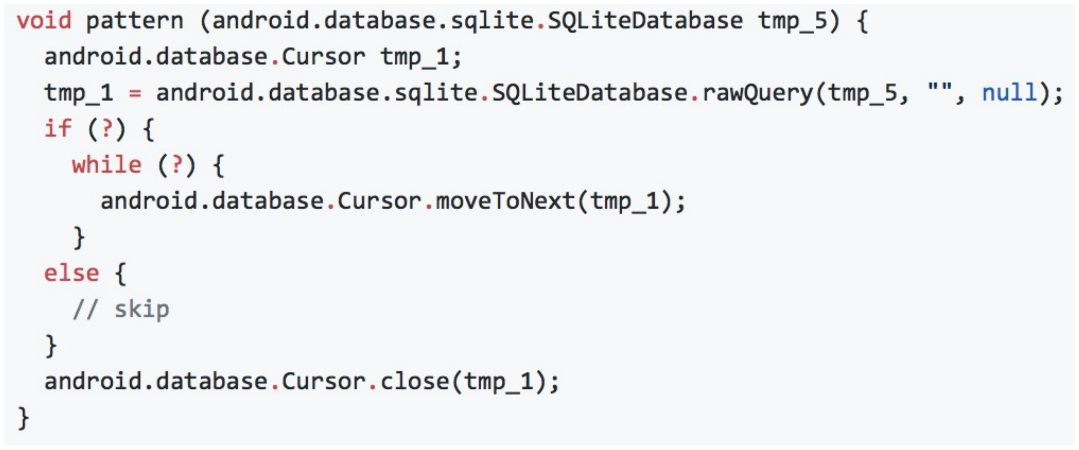
▶ Methods Involved

▶ Reference Pattern

▶ Suggested Patch

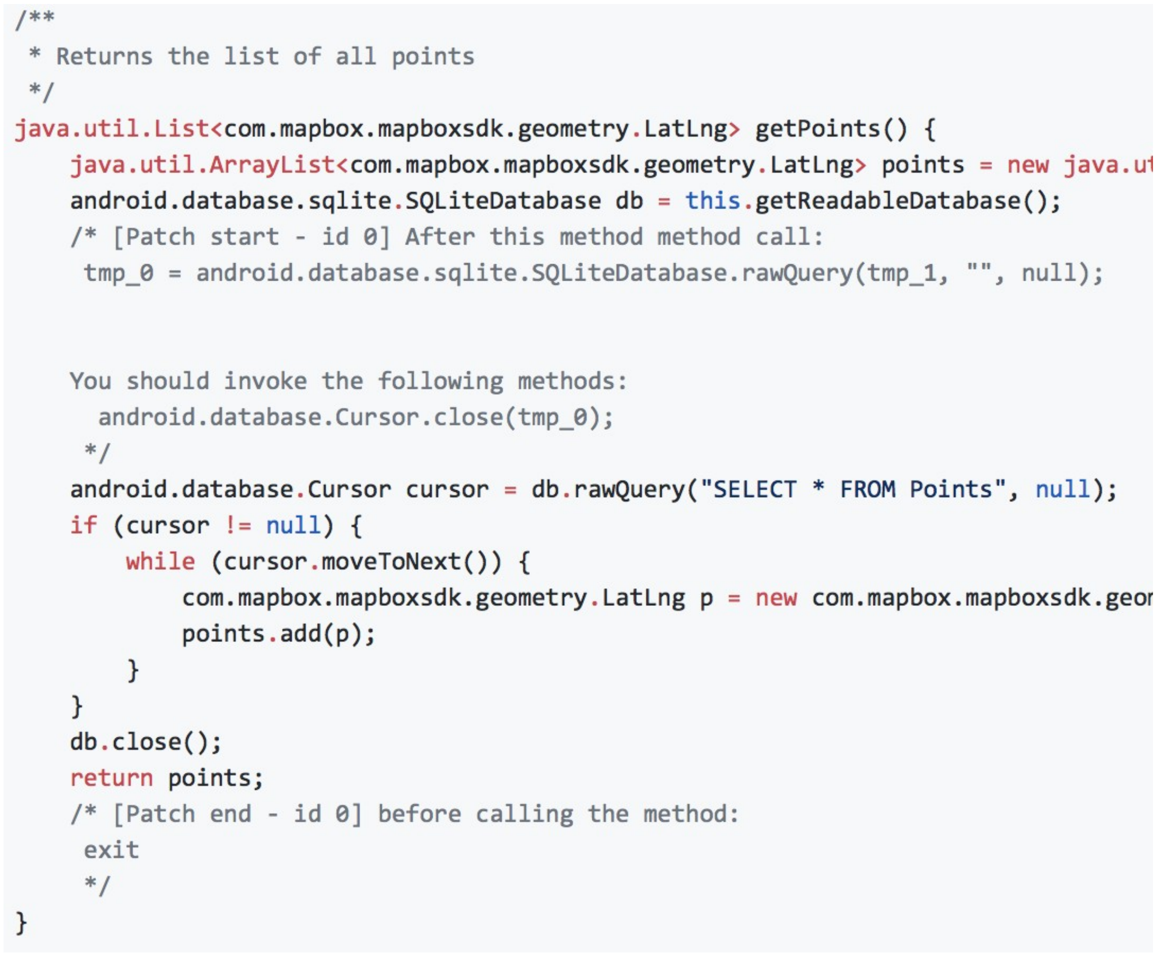
Figure 13: A BigGroum Anomaly Report on GitHub

- The set of methods involved in the pattern (in this case methods SQLiteDatabase.rawQuery, Cursor.close, and Cursor.moveToNext).
- Source code describing the reference pattern that the user code violated (i.e., the pattern that the developer did not implement and that BigGroum identified as the correct API usage). Figure 14 shows a POPULAR, reference pattern describing that after performing a query on a database (with method SQLiteDatabase.rawQuery) the developer can repeatedly access the results of the query (with a while loop invoking the method Cursor.moveToNext some number of times), and finally release the Cursor object (with method Cursor.close).
- A repair suggestion showing a commented version of the developer code (from the analyzed commit). The comments suggest modifications (i.e., addition and removal of method invocations) to the code to fix the anomaly. Figure 15 shows the code of the getPoints method and two comments suggesting to release the Cursor object (by inserting the method Cursor.close at the end of the method implementation).



```
void pattern (android.database.sqlite.SQLiteDatabase tmp_5) {
  android.database.Cursor tmp_1;
  tmp_1 = android.database.sqlite.SQLiteDatabase.rawQuery(tmp_5, "", null);
  if (?) {
    while (?) {
      android.database.Cursor.moveToNext(tmp_1);
    }
  else {
    // skip
  }
  android.database.Cursor.close(tmp_1);
}
```

Figure 14: A BigGroum Reference Pattern Shown on GitHub



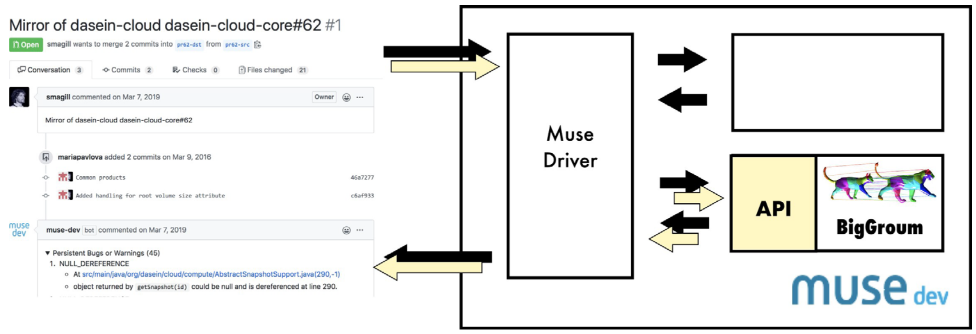
```
/**
 * Returns the list of all points
 */
java.util.List<com.mapbox.mapboxsdk.geometry.LatLng> getPoints() {
    java.util.ArrayList<com.mapbox.mapboxsdk.geometry.LatLng> points = new java.ut
    android.database.sqlite.SQLiteDatabase db = this.getReadableDatabase();
    /* [Patch start - id 0] After this method method call:
     tmp_0 = android.database.sqlite.SQLiteDatabase.rawQuery(tmp_1, "", null);


    You should invoke the following methods:
      android.database.Cursor.close(tmp_0);
     */
    android.database.Cursor cursor = db.rawQuery("SELECT * FROM Points", null);
    if (cursor != null) {
        while (cursor.moveToNext()) {
            com.mapbox.mapboxsdk.geometry.LatLng p = new com.mapbox.mapboxsdk.geor
            points.add(p);
        }
    }
    db.close();
    return points;
    /* [Patch end - id 0] before calling the method:
     exit
     */
}
```

Figure 15: A BigGroum Suggested Patch Shown on GitHub

The MuseDev platform automates the integration of static analysis tools in GitHub. At a high level, the platform provides a generic API to integrate static analyzers, allows a developer to configure a set of analysis to run on her GitHub repository, and triggers a new analysis for every new commit in a pull request. In each analysis, the MuseDev platform downloads the developer repository, compiles the repository code, executes the configured static analyzer tools, collecting their results, and provides the *new* analysis results for the analyzed commit (i.e., the platform reports only the results that have been introduced in the latest change).

After a new commit on the source branch of a pull request (image on the left), GitHub invokes the MuseDev platform (the Muse driver box on the right) as shown in Figure 16. The Muse driver instantiates a new analysis, downloads the developer code from GitHub, compiles it, and invokes a static analysis tool (e.g., BigGroum). Each analysis tool produces a set of results for the commit and are sent back to the Muse driver and then are posted as comments on the pull request on GitHub. A significant step of the integration was providing an implementation of the MuseDev APIs (the yellow box with label API). Figure 16 shows the architecture of the MuseDev platform at a high level and how BigGroum integrates in the platform. BigGroum implements the Talk and Per-File (TAPFI) API. In particular, TAPFI defines an end-point (*finalize*) to process all the compiled artifacts that the MuseDev platform produced for the git repository. In the *finalize* implementation, BigGroum runs the anomaly detection search on the compiled artifact. The BigGroum software architecture is distributed and is implemented with a main web service that takes as input the bytecode of the Android application to analyze and produces as output a set of anomalies (hence, the web service is stateless). The MuseDev platform only runs a "thin" implementation of the TAPFI API that invokes an instantiation of the BigGroum service—such an architecture will allow the integration to scale easily and support a higher number of requests.


Mirror of dasein-cloud dasein-cloud-core#62 #1

Open smagill wants to merge 2 commits into pr62-dst from pr62-src

Conversation 3 · Commits 2 · Checks 0 · Files changed 21

smagill commented on Mar 7, 2019 Owner

Mirror of dasein-cloud dasein-cloud-core#62

mariapavlova added 2 commits on Mar 9, 2016

```
Common products                                   46a7277
Added handling for root volume size attribute     c6af933
```

muse-dev bot commented on Mar 7, 2019

▾ Persistent Bugs or Warnings (45)
1. NULL_DEREFERENCE
   - At src/main/java/org/dasein/cloud/compute/AbstractSnapshotSupport.java(290,-1)
   - object returned by `getSnapshot(id)` could be null and is dereferenced at line 290.

Figure 16: BigGroum on the MuseDev Platform

3.3 Fusing Scripted and Randomized UI Testing with ChimpCheck

We introduce our fused approach and ChimpCheck by means of an example testing scenario from the app-test developer's perspective. The purpose of this example is not necessarily to show every feature of ChimpCheck but rather to demonstrate the need to fuse app-specific knowledge with randomized testing.

3.3.1 An Example UI Testing Scenario: Signing In.

Consider the problem of testing a music service app as shown in Figure 17, which requires (a) fusing fixture-specific flows with random interrupts and (b) asserting app-specific properties. Testing this app requires using test fixtures for getting past user authentication and asserting properties of the user interface specific to being a music player. For concreteness, let us consider testing a particular user story for the app: (1) Click on button Enter; (2) Type in test and 1234 into the username and password text boxes, respectively; (3) Click on button Sign in; and (4) Click on buttons ▶ and ⏸ to try out starting and stopping the music player, respectively. Observe that the first few steps (Steps 1–3) describe setting up a particular test fixture to get to the "interesting part of the app," while the last step (Step 4) finally gets to testing the app component of interest.

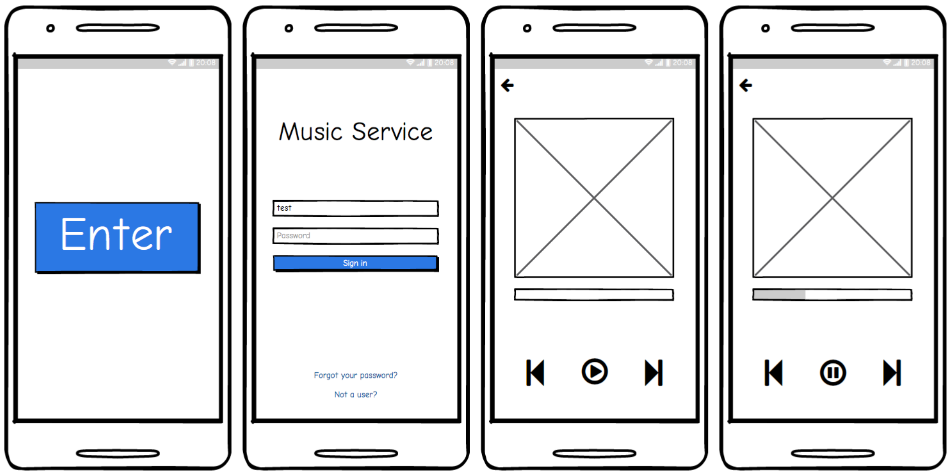


Figure 17: Testing a Music Service App

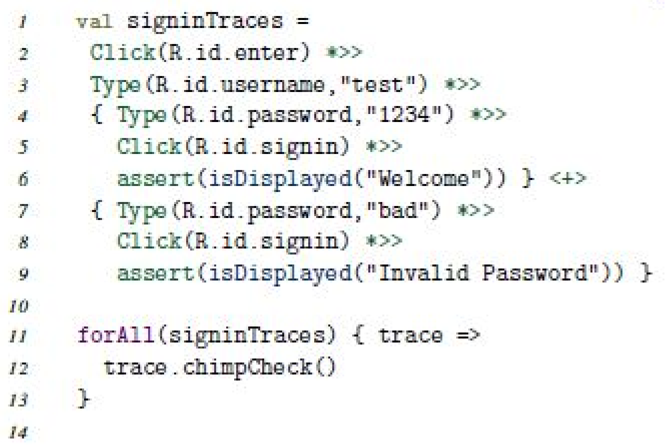
This description captures the user story that the test developer seeks to validate, but an effective test suite will likely need more than one corresponding test case to see that that this user-flow through the app is robust. With ChimpCheck, the test developer describes essentially the above script, but the script can be fused with fuzzing and properties to specify not a single test case but rather a family of test cases. For example, suppose the test developer wants to generate a family of test cases where:

a. The sign-in phase (Steps 1–3) is robust to interrupt events such as screen rotations.

b. The state of ▶ and ⏸ buttons in the user interface corresponds in the expected manner to the internal state of an android.media.MediaPlayer object.

Fusing Fixture-Specific Flows with Interrupts (a). As a test developer, we want coverage that samples from both valid and invalid scenarios. While invalid sign-ins are easily stumbled upon by brute-force random techniques, many important scenarios require testing after a valid sign-in, and the most reasonable means of testing beyond a valid sign-in is to hard-code a test fixture account. While we would like a concise way of expressing such fixtures, we also want some way to generate variations that test the sign-in process with interrupt events (e.g., screen rotate, app suspend, and resume) inserted at various points of the sign-in process. Such variations are important to test because interrupts often are sources of crashes (e.g., null pointer exceptions) as well as unexpected behaviors (e.g., characters keyed into a text box vanishes after screen rotation). Developing such test cases one-at-a-time (via Espresso or Robotium [28, 29]) is too time consuming, and most test-generation techniques (via [30, 32–34]) would not be effective at finding valid sign-in sequences.

A key contribution of ChimpCheck is that it empowers the test developer to define the skeleton of the kind of UI traces of interest. Concretely in Figure 18, shows how ChimpCheck focuses test development on the valid and invalid sign-in flows for the music service app from Figure 17, we specify not only a hard-coded sign-in sequence but variations of it that include interrupt actions that an app-user could potentially interleave with the sign-in sequence.



```
1   val signinTraces =
2    Click(R.id.enter) *>>
3    Type(R.id.username,"test") *>>
4    { Type(R.id.password,"1234") *>>
5      Click(R.id.signin) *>>
6      assert(isDisplayed("Welcome")) } <+>
7    { Type(R.id.password,"bad") *>>
8      Click(R.id.signin) *>>
9      assert(isDisplayed("Invalid Password")) }
10
11  forAll(signinTraces) { trace =>
12    trace.chimpCheck()
13  }
14
```

Figure 18: Specifying the Skeleton of Relevant User Interaction

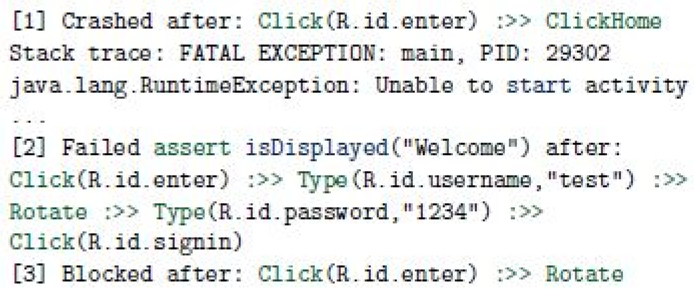
On line 1, the test developer defines a value signinTraces that is a *generator* of sign-in traces where a trace is a sequence of UI events that drives the app in some way. To define signinTraces, the test developer describes essentially the sign-in flow outlined earlier: (1) Click on button Enter with Click(R.id.enter) on line 2; (2) Type in test with Type(R.id.username,"test") on line 3 and type 1234 with Type(R.id.password,"1234") on line 4; and (3) Click on button Sign in with Click(R.id.signin) on line 5. In Android, the R class contains generated constants that uniquely identify user-interface elements of an app. Like an Espresso-test developer today, we use these identifiers to name the user-interface elements.

The user-interaction events like Click and Type specify an individual user action in a UI trace. These events can be composed together with a core operator like :>> that represents sequencing or, as on line 6, with the operator <+> that implements a non-deterministic choice between its left operand (lines 4–5) and its right operand (lines 7–9). Thus, signinTraces does not represent just a single trace but a set of traces. Here, we union two sets of traces to get some valid sign-ins (lines 4–6) and some invalid ones (lines 7–9). As the test developer, we can check for the correctness of the sign-in scenarios with an assert for a welcome message in the case of the valid sign-ins (line 6) or for an “Invalid Password” error message in the case of the invalid ones (line 9). The isDisplayed expressions specify properties on the user interface that are then checked with assert.

Finally, the UI events are sequenced together with the *>> combinator rather than the :>> operator. The *>> combinator represents interruptible sequencing, that is, sequencing with some non-deterministic insertion of interrupt events. Because these generator expressions represent sets of traces rather than just a single trace, this interruptible sequencing combinator *>> becomes a natural drop in for the core sequencing operator :>>. And indeed, interruptible sequencing *>> is definable in terms of sequencing :>> and other core operators.

ChimpCheck is built on top of ScalaCheck [48], extending this property-based test generation library with UI traces and integration with Android test drivers running emulators or actual devices.

From a practical standpoint, by building on top of ScalaCheck, ChimpCheck inherits ScalaCheck's test generation functionalities. The signinTraces generator is then passed to a forAll call on line 12. The forAll combinator comes directly from the ScalaCheck library that generically implements sampling from a generator. Tests are executed by invoking the ChimpCheck library operation .chimpCheck() on UI-trace samples. Here, UI trace samples are bound to trace and executed on line 12 (i.e., trace.chimpCheck()). When this operation is executed, it triggers off a run of the app on an actual emulated Android device and submits the trace trace (a trace sampled from signinTraces) to an associated test driver running in tandem with the app. When problems are encountered during each run, details of crashes or assertion failures are reported back to the user. Figure 19 shows examples of bug reports, including the UI trace that leads to the crash or assertion failure, from ChimpCheck.



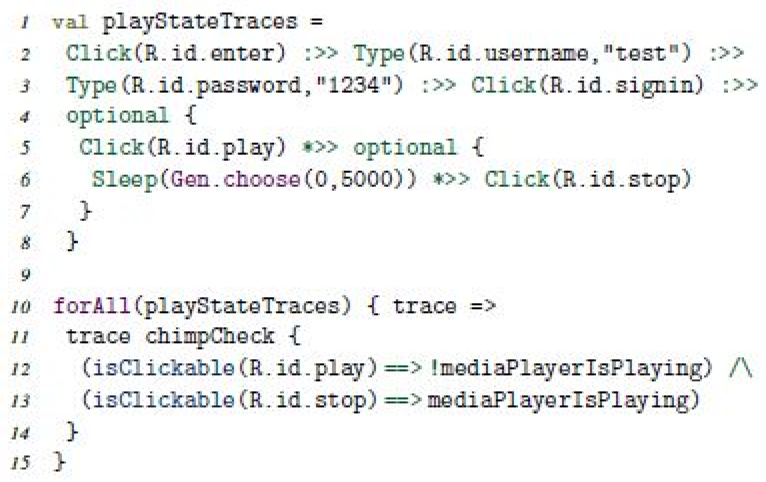
```
[1] Crashed after: Click(R.id.enter) :>> ClickHome
Stack trace: FATAL EXCEPTION: main, PID: 29302
java.lang.RuntimeException: Unable to start activity
...
[2] Failed assert isDisplayed("Welcome") after:
Click(R.id.enter) :>> Type(R.id.username,"test") :>>
Rotate :>> Type(R.id.password,"1234") :>>
Click(R.id.signin)
[3] Blocked after: Click(R.id.enter) :>> Rotate
```

Figure 19: A CimpCheck Report for a Failed Test

An important contribution of ChimpCheck is that all reports contain the actual user-interaction trace executed up to the point of failure. In an interactive application, a stack trace is severely limited because it contains only the internal method calls from the callback triggered by the last user interaction. This UI trace is essentially an executable and concise description of the sequence of UI events that led up to the failure. And thus, these executed UI traces are valuable in the debugging process as they can guide the developer in reproducing the failure and ultimately understanding the root causes.

Asserting App-Specific Properties (b). Many problems in Android apps do not result in run- time exceptions that manifest as crashes but instead lead to unexpected behaviors. It is thus critical that the testing framework provide explicit support for checking for app-specific properties. To check for unexpected behaviors, the developer must write custom test scripts and invoke specific assertions at particular points of executing the Android app.

ChimpCheck provides explicit support for property-based testing, which enables the test developer to simultaneously express generators for describing relevant UI traces and assertions for specifying app-specific properties to check (ie. enables simultaneously describing the trace of the relevant user interactions to drive the app to a particular state and checking properties on the resulting state). In Figure 20, we show playStateTraces, another UI trace generator for the same music-service app that focuses on testing that the UI state with the ▶ and ⏸ toggle is consistent with the internal MediaPlayer object. Since the signinTraces from Figure 18 already tests the sign-in process, these tests simply wire-in the test fixture to get past the sign-in screen into the music player with the plain sequencing operator :>> (lines 2–3). Past the sign-in screen, lines 4–8 implement the optional-cycling between the ▶ and ⏸ music player states via Clicking the corresponding buttons. In this particular example, we insert a random idle of 0 to 5 seconds (Sleep(Gen.choose(0,5000))) between a cycle. Note that Gen.choose(0,5000) is not special to ChimpCheck; it is part of the ScalaCheck library to generate an integer between 0 and 5,000. Finally, on lines 10–15, we execute the actual test runs as before, but this time we supply a property expression (lines 12–13) comprised of two implication rules that asserts the expected consistency between the UI state and the underlying MediaPlayer state. Note that while isClickable is a built-in atomic predicate that accesses the Android run-time view-hierarchy, mediaPlayerIsPlaying is a predicate defined by the developer. Its implementation is simply a call to the MediaPlayer object's isPlaying method.



```
1  val playStateTraces =
2   Click(R.id.enter) :>> Type(R.id.username,"test") :>>
3   Type(R.id.password,"1234") :>> Click(R.id.signin) :>>
4   optional {
5    Click(R.id.play) *>> optional {
6     Sleep(Gen.choose(0,5000)) *>> Click(R.id.stop)
7    }
8   }
9
10 forAll(playStateTraces) { trace =>
11  trace chimpCheck {
12   (isClickable(R.id.play) ==> !mediaPlayerIsPlaying) /\
13   (isClickable(R.id.stop) ==> mediaPlayerIsPlaying)
14  }
15 }
```

Figure 20: Describing Relevant User Interactions and Checking Properties

3.3.2 Implementing UI Trace Generators.

Figure 21 illustrates the entire test generation and execution architecture of ChimpCheck, from the test developer writing ChimpCheck UI trace generator scripts; the underlying ScalaCheck library sampling the UI traces from the UI trace generators; the test coordinator issuing each UI trace sample to a unique Android emulator for independent exercise of the instance; and the execution outcome each UI trace being reported back to the test developer. The test developer writes UI trace generators rather than a single UI trace (as natively in Espresso or Robotium). To generate concrete instances of traces from these generators, we have implemented the generators on top of the ScalaCheck [48] library— Scala's active implementation of property-based test generation [35]. Since the testing framework now potentially needs to manage multiple test executions, the ChimpCheck library includes a coordinator library that coordinates the test executions: it coordinates the testing efforts across multiple instances of the test controller and Android devices (physical or emulated), scheduling test runs of traces across the test execution pool, and consolidating concretely executed traces and test results. This coordinator library is developed with Scala's high-performance Akka actor library.

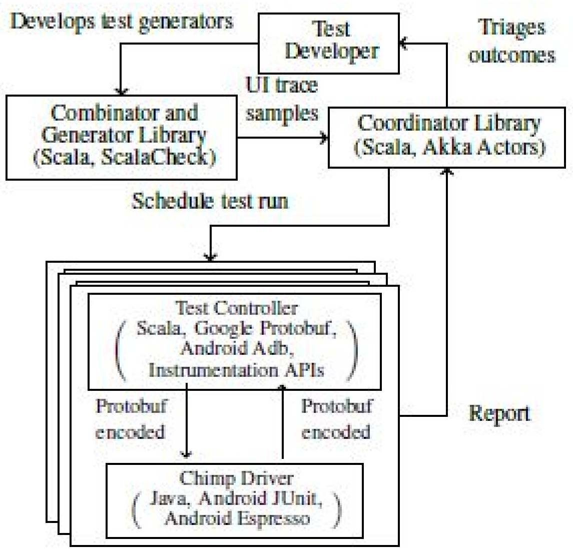

Develops test generators
Test Developer
Triages outcomes
UI trace samples
Combinator and Generator Library (Scala, ScalaCheck)
Coordinator Library (Scala, Akka Actors)
Schedule test run
Test Controller (Scala, Google Protobuf, Android Adb, Instrumentation APIs)
Protobuf encoded
Protobuf encoded
Report
Chimp Driver (Java, Android JUnit, Android Espresso)

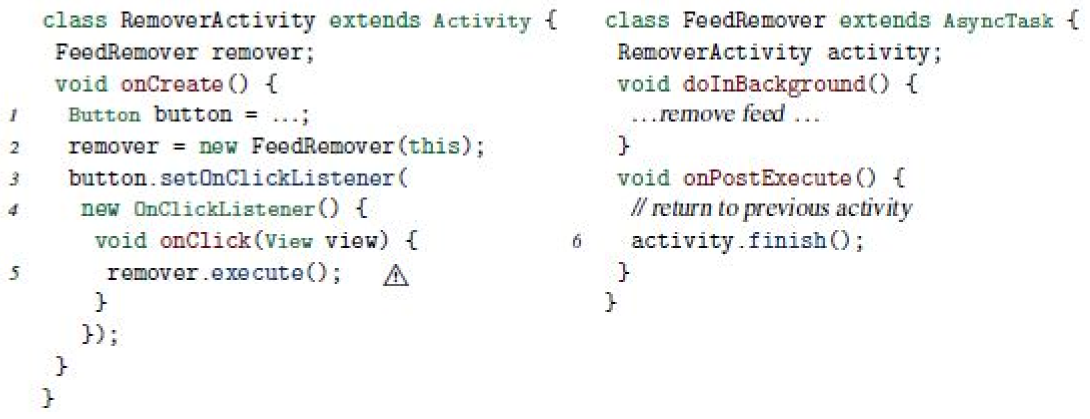
Figure 21: Architecture of ChimpCheck Test Generation

The design choice of building the ChimpCheck Generator library on top of ScalaCheck has proven to be extremely beneficial for our current and anticipated future development efforts of ChimpCheck: rather than developing randomized test generation from scratch, we leverage ScalaCheck's extensive library support for generating primitive types (e.g., strings and integers). This library support includes many utility functions from generating arbitrary values of the respective domains, to uniform/weighted choosing operations. The ScalaCheck library is also highly extensible, allowing developers to extend these functionalities to user-defined algebraic datatypes (e.g., trees, heaps) and in our case, extending test generation functionalities to UI trace combinators. This approach not only offers the ChimpCheck developer compatible access to ScalaCheck library of combinators, it makes ChimpCheck reasonably simple to maintain and incrementally developed.

3.4 Reasoning about Callback Control Flow with Verivita

3.4.1 Verifying Application Protocol Violations.

In Figure 22, we show a code snippet from an example Android app containing a protocol violation (the bug is inspired by an existing issue in AntennaPod [49], a podcast manager app with 100,000+ installs, and the Facebook SDK for Android [50]). The code implements the asynchronous removal of a feed: when the user clicks the Button button, the app creates an AsyncTask remover that deletes the feeds in the back-ground. Note that the removal happens in the doInBackground callback implemented in the FeedRemover class (which extends the Android AsyncTask class). The bug happens when calling the method remover.execute() (marked with ⚠): the call throws an IllegalStateException if the AsyncTask *t* instance, pointed-to by remover, is already running (i.e., the exception happens if the remover.execute() is called twice on the same object instance). This exception is an example of *protocol violation*: the AsyncTask implementation, provided by the Android framework, does not allow the app code to invoke remover.execute() twice!



```
  class RemoverActivity extends Activity {
   FeedRemover remover;
   void onCreate() {
1   Button button = ...;
2   remover = new FeedRemover(this);
3   button.setOnClickListener(
4    new OnClickListener() {
      void onClick(View view) {
5      remover.execute();   ⚠
      }
     });
   }
  }
```

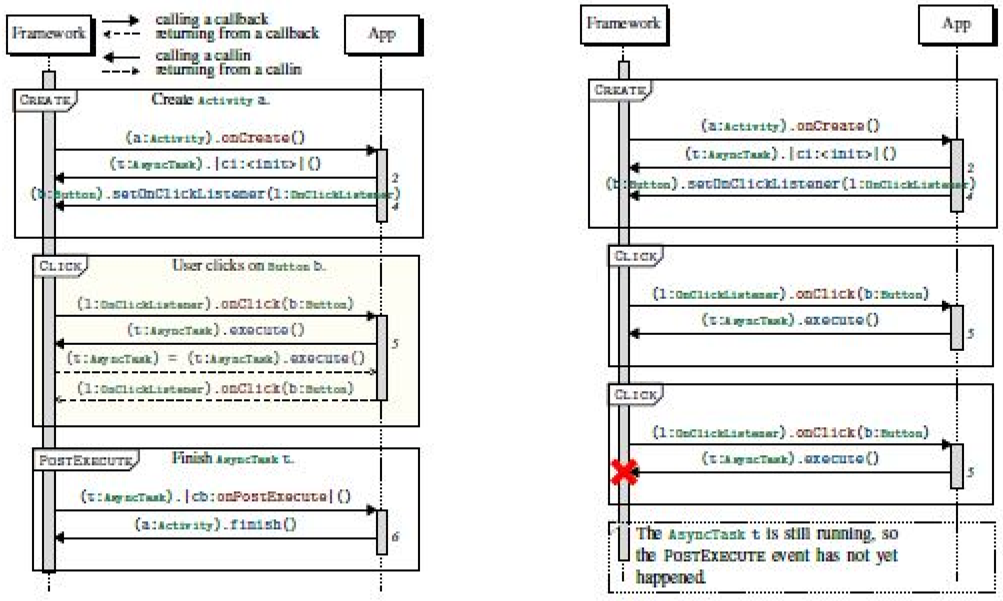
```
  class FeedRemover extends AsyncTask {
   RemoverActivity activity;
   void doInBackground() {
    ...remove feed ...
   }
   void onPostExecute() {
    // return to previous activity
6   activity.finish();
   }
  }
```

Figure 22: An App Containing a Protocol Violation

While in this case, we can "easily" characterize the protocol violation, what makes event-driven programming challenging is understanding under which conditions a protocol violation will happen in an app. In fact, even in our example the exception does not happen at every execution of the app but only when the app executes a particular sequence of events. We visualize the interface between an event-driven framework and an app as a dialog between two components, with execution time flowing downwards as a sequence of events, as shown in Figure 23. Control begins on the left with the framework receiving an event, focusing on the highlighted Click event in Figure 23a. When a user Clicks on the button corresponding to Object b of Type Button, the onClick Callback is invoked by the Framework on the registered listener l. For clarity, we write method invocations with type annotations (e.g., (l:OnClickListener).onClick(b:Button)), and variables b and l stand for some concrete instances (rather than program or symbolic variables). The app then delegates back to the framework by calling an API method to start an asynchronous task t via (t:AsyncTask).execute(); to connect with the app source code. We label the callins originating from the app timeline with the corresponding program point numbers in Figure 22. Here, we can clearly see a *Callback* as any app method that the framework can call (i.e., with an arrow to the right), and a *Callin* as any framework method that an app can call (i.e., with an arrow to the left). We show returns with dashed arrows (but sometimes elide them when they are unimportant).

In Figure 23a we show a correct execution (i.e., an execution not ending with the IllegalStateException, while in Figure 23b, we show a buggy execution. How do they differ? The correct execution consists of the sequence of events Create;Click;PostExecute (i.e., the Activity starts, the user clicks once the Button, and the background AsyncTask terminates its execution). The buggy execution contains the sequence of events Create;Click;Click: clicking the button a second time before the FeedRemover task completes and generates the post-execute event (PostExecute) causes the bug. In practice, the bug manifests only if the FeedRemover is not "fast enough" (or, alternatively, the user clicks the Button very fast) — a condition difficult to test, either manually or automatically. A fix for the above app would be to simply "disable" the Click event after we call it the first time: we obtain this effect if we invoke the method (b:Button).setEnabled(false) in the implementation of the onClick callback. Calling (b:Button).setEnabled(false) has the effect of disabling the Button and hence prohibiting the sequence of events Create;Click;Click. This example shows two key aspects of event-driven programming:

- To reason about the protocol violation in the app code, we need to understand the possible order of execution of the callbacks the app implements (i.e., when will the app code be executed?). We refer to the possible order execution of callbacks as *callback control flow*.
- Both the framework and the app determine the order of execution of the app's callbacks (e.g., the event onClick cannot happen before the event onCreate, and onClick cannot happen after the invocation to (b:Button).setEnabled(false)).



Framework
calling a callback
returning from a callback
calling a callin
returning from a callin
App
CREATE
Create Activity a.
(a:Activity).onCreate()
(t:AsyncTask).[c1:<init>]()
(b:Button).setOnClickListener(l:OnClickListener)
CLICK
User clicks on Button b.
(l:OnClickListener).onClick(b:Button)
(t:AsyncTask).execute()
(t:AsyncTask) = (t:AsyncTask).execute()
(l:OnClickListener).onClick(b:Button)
POSTEXECUTE
Finish AsyncTask t.
(t:AsyncTask).[cb:onPostExecute]()
(a:Activity).finish()
Framework
App
CREATE
(a:Activity).onCreate()
(t:AsyncTask).[c1:<init>]()
(b:Button).setOnClickListener(l:OnClickListener)
CLICK
(l:OnClickListener).onClick(b:Button)
(t:AsyncTask).execute()
CLICK
(l:OnClickListener).onClick(b:Button)
(t:AsyncTask).execute()
The AsyncTask t is still running, so the POSTEXECUTE event has not yet happened

(a) A Trace that Does Not Witness a Protocol Violation Since the Callin (t:AsyncTask).execute() on t Is Executed Only Once

(b) The CREATE;CLICK;CLICK Interaction Sequences Witnesses the No-execute-Call-on-Already-Executing-AsyncTask Protocol Violation

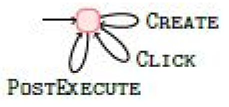
Figure 23: Interaction Traces Witnessing and Not Witnessing a Protocol Violation

We consider the problem of *dynamic verification* of protocol violations: given an execution trace that does not expose the violation, the dynamic verification explores all the possible sequences of interactions that can be obtained by replicating, removing, and reordering the events in a trace. For example, when we consider the buggy version of the app and the (correct) execution trace Create;Click;PostExecute, the verification algorithm should rearrange the events in the trace to find the error trace Create;Click;Click. More importantly, when verifying the same trace on the correct version of the app, the verification algorithm *should not* find the error trace (i.e., that result would be a false alarm). To achieve such a result, we need to capture the essential, hidden framework state—tracking the set of enabled callbacks and the set of disallowed callins. In practice, after the first Click, the application disables the button to prevent a second Click via the call to (b:Button).setEnabled(false), which at that point removes l.onClick(b) from the set of enabled callbacks the framework can trigger.

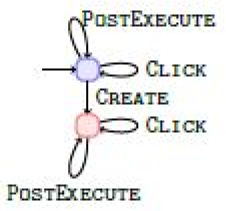
Verivita addresses the dynamic verification problem by reducing it to a model checking problem. The model is a transition system with (i) states abstracting the set of enabled callbacks and disallowed callins and (ii) transitions capturing the possible replication, removing, and reordering of a given interaction trace. The safety property of interest is that the transition system never visits a disallowed callin. How can we construct such a transition system that over-approximates concrete behavior while being precise enough? A technical problem to solve to enable such precise automatic reasoning for event-driven applications is to have adequate callback control flow modeling.

3.4.2 Precision and Soundness of Callback Control Flow Models.

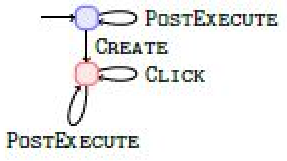
In the following, we describe how existing control flow models affect the precision and soundness of the verification problem. In Figure 24, we illustrate the essence of callback control-flow modeling as finite-state automata that over-approximate rearrangements of the Create;Click;PostExecute trace when considering the *fixed version* of the app. In most prior work, models are generated for an application by restricting the possible order of callbacks. We show four sound abstractions with different levels of precision, indicating whether they are precise enough to verify our fixed application (as well as one unsound abstraction).



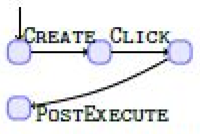
(a) False Alarm on the Trivially Sound, Unconstrained, "Top" Abstraction



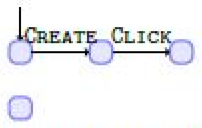
(b) False Alarm on the Activity Lifecycle-Refined Abstraction



(c) False Alarm on the Lifecycle with the CLICK Restricted to the Active Activity State (as Shown in Figure 6)



(d) Verified Safe When We Consider the Effect of Button.setEnabled()



(e) This Unsound Abstraction is Missing the POSTEXECUTE Edge

Figure 24: Different Models for the Callback Control Flow

Automaton 24a exhibits the trivially sound, unconstrained, "top" abstraction that considers all replications, removals, and reordering's of the trace. This abstraction is the one that assumes all callbacks are always enabled. Since a possible trace in this abstraction includes two Click events, a sound verifier must alarm.

Meanwhile, the Automaton 24b shows a refined abstraction encoding the Android-specific Activity lifecycle. The abstraction is framework-specific but application-independent and captures that the Create event cannot happen more than once. The abstraction shown by Automaton 24b is also insufficient to verify the trace from the fixed app because two Click events are still possible.

Automaton 24c shows a refined, lifecycle++ abstraction that considers the Activity lifecycle with additional constraints on an “attached” Click event. This abstraction is representative of the current practice in callback control-flow models (e.g., [38–40]). While Automaton 24c restricts the Click event to come only after the Create event, the abstraction is still too over-approximated to verify that the trace from the fixed app is safe—two Click events are still possible with this model. But worse is that this model is still, in essence, a lifecycle model that is constrained by Android-specific notions like View attachment, Listener registration, and the “live” portion of lifecycles. In existing analysis tools, such constrained lifecycle models are typically hard-coded into the analyzer.

3.4.3 Lifestate: A New Perspective on Callback Control Flow Models.

We need a better way to capture how the application may affect callback control flow. In our example, we need to capture the effect of the callin button.setEnabled(false), which is the only difference with the buggy version in Figure 22. The modeling needs to be expressive to remove such infeasible traces and compositional to express state changes independently.

We propose a new specification language, *lifestate*, that describes how the internal model state is updated by observing the history of intertwined callback and callin invocations. In lifestate, we write:

$$(\ell_b\text{:Button}).\text{setEnabled}(\text{false}) \nrightarrow (\ell_l\text{:OnClickListener}).\text{onClick}\ (\ell_b\text{:Button})\ \text{(for all }\ _l, \ell_b)$$

to model the statement that when $(\ell_b\text{:Button}).\text{setEnabled}(\text{false})$ is invoked, the click callback is *disabled* on the same button ℓ_b (on all listeners ℓ_l). Also, we can similarly specify the safety property of interest:

$$(\ell_t\text{:AsyncTask}).\text{execute}() \nrightarrow (\ell_t\text{:AsyncTask}).\text{execute}()\ \text{(for all } \ell_t)$$

that when $(\ell_t\text{:AsyncTask}).\text{execute}()$ is called on a task ℓ_t, it *disallows* itself. And analogously, lifestates include specification forms for *enabling callbacks* or *allowing callins*.

Lifestate uniformly models callback control flow and specifies event-driven application-programming protocols. The rules that enable and disable callbacks model what callbacks the framework can invoke at a specific point in the execution of the application, while the rules that disallow and allow callins specify what callins the application must invoke to respect the protocol. What makes lifestate unique compared to typestates [51] or lifecycle automata is this unification of the intertwined effects of callins and callbacks on each other. We refer to our paper [1] for the definition of the formal syntax and semantics of the lifestate language.

The complexity of implicit callback control flow is what makes expressing and writing correct models challenging. An issue whose importance is often under-estimated when developing callback control-flow models is how much the model faithfully reflects the framework semantics. How can we *validate* that a lifestate specification is a correct model of the event-driven framework?

3.4.4 Validating Event-Driven Programming Protocols.

As argued above, a key concern when developing a framework model is that it must over-approximate the possible real behavior of the application. The "top" model as shown in Automaton 24a trivially satisfies this property, and it may be reasonable to validate an application-independent lifecycle model like Automaton 24c. However, as we have seen, verifying correct usage of event-driven protocols typically requires callback control-flow models with significantly more precision.

Automaton 24d shows a correct lifestate-abstraction that contains an edge labeled PostExecute. We express this edge with the rule shown below:

$(\ell_t\texttt{:AsyncTask}).\texttt{execute}() \rightarrow (\ell_t\texttt{:AsyncTask}).\texttt{onPostExecute}()$ (for all ℓ_t)

This rule states that when $(\ell_t\texttt{:AsyncTask}).\texttt{execute}()$ is called, its effect is to enable the callback

$(\ell_t\texttt{:AsyncTask}).\texttt{onPostExecute}()$ on the AsyncTask ℓ_t.

If we do not model this rule, we obtain the abstraction in Automaton 24e. The lifestate model is *unsound* since it misses the PostExecute edge. The trace of the app Create;Click;PostExecute is a witness of the unsoundness of the abstraction. Automaton 24e accepts only proper prefixes of the trace (e.g., Create;Click), and hence the abstraction does not capture all the possible traces of the app. We can thus use interaction traces to *validate* lifestate rules: a set of lifestate rules is valid if the abstraction accepts all the interaction traces. The validation applied to the abstraction shown in Automaton 24e demonstrates that the abstraction accepts Create;Click as the longest prefix of the trace Create;Click;PostExecute. This information helps to localize the cause for unsoundness since we know that after the sequence Create;Click, the callback PostExecute is (erroneously) disabled.

The encoding of the abstraction from lifestate rules is a central step to perform model validation and dynamic verification. At this point, we still cannot directly encode the abstraction since the lifestate rules contain universally-quantified variables. How can we encode the lifestate abstraction as a transition system amenable to check language inclusion for validation, and to check safety properties for dynamic verification?

3.4.5 From Specification to Validation and Verification.

Generalizing slightly, we use the term *message* to refer to any observable interaction between the framework and the app. Messages consist of invocations to and returns from callbacks and callins. The abstract state of the transition system is then a pair consisting of the *permitted-back messages* from framework to app and the *prohibited-in messages* from app to framework. And thus generalizing the example rules shown above, a lifestate specification is a set of rules whose meaning is,

> *If* the message history *matches* r, *then* the abstract state is updated according to the specified *effect* on the set of permitted-back and prohibited-in messages.

There are many possible choices and tradeoffs for the matching language r. As is common, we consider a regular expression-based (i.e., finite automata-based) matching language.

We exploit the structure of the validation and dynamic verification problem to encode the lifestate abstraction. In both problems, the set of possible objects and parameters is finite and determined by the messages recorded in the trace. We exploit this property to obtain a set of *ground* rules (rules without variables). We can then encode each ground rule in a transition system. Since the rule is ground, the encoding is standard: each regular expression is converted to an automaton and then encoded in the transition system, changing the permitted-prohibited state as soon as the transition system visits a trace accepted by the regular expression, which implicitly yields a model like Automaton 24d.

Our paper [1] describes in depth the formal encoding of both the validation and verification problems as a model checking problems on a finite-state transition system. This problem can be solved using off-the-shelf symbolic model checking tools [52]. Our implementation of the verification tool Verivita is also available on GitHub [13].

3.4.6 Event-Driven App-Framework Interfaces.

Lifestate offers a general and flexible way to specify the possible future messages in terms of observing the past history of messages.

It, however, essentially leaves the definition of messages and what is observable abstract. What observables characterize the interactions between an event-driven framework and an app that interfaces with it? And how do these observables define event-driven application-programming protocols and callback control flow?

Lifestate rules are agnostic to the kinds of messages they match and effects they capture on the internal abstract state. To give meaning to lifestates, we formalize the essential aspects of the app-framework interface in an abstract machine model called λ_{life}. This abstract machine model formally characterizes what we consider an *event-driven framework*. The λ_{life} abstract machine crisply defines the messages that the app and the framework code exchange and a formal correspondence between concrete executions of the program and the app-framework interface. We use this formal correspondence to define the semantics of the lifestate framework model, its validation problem, and protocol verification.

As-is, we have also left open the problem of efficiently specifying lifestate models. That is, how do we obtain sound and precise lifestate models with minimum manual effort? While this problem is extremely challenging in general, we have developed DroidStar [9–11] that can infer useful models in particular restricted contexts using active learning techniques.

3.5 Supporting Mining and Understanding Tools with Fixr Services

For the Fixr storage layer, we eventually settled on the use of multiple persistence technologies to store our Android repositories, extracted features, and generated downstream artifacts. For the entire project, we always stored Android repositories in standard file systems downloading these repositories from GitHub using git. The file system is optimized to handle this type of storage and git assumes that repositories are stored there. With respect to extracting features from those repositories, we first started by writing programs that would traverse the repositories and generate the information needed to extract features by placing that information into large text files that were stored (and replicated) by the Hadoop Distributed File System (HDFS). These files could then be processed by batch processing jobs running in Spark allowing for features to be extracted via large MapReduce jobs. When the MapReduce jobs were finished, a program extracted the Protocol Buffers (protobuf) files that had been generated from HDFS back to a regular file system where individual analysis tools could then make use of them in service to the goals of other aspects of the Fixr project. This extraction included generating a file that could be imported into Solr to search for commits in our Android repositories that might be related to an Android framework bug. Solr was used to do the searching for relevant commits and generated Hypertext Markup Language (HTML) output that could then be viewed in a Web browser for presentation to other Fixr researchers.

As we moved away from batch processing technologies, we stopped making use of HDFS and instead wrote incremental, asynchronous software that would extract information from repositories and store it directly in Solr. This included storing extracted features in protobuf format in Solr with metadata that made it easy to find the extracted feature based on repository name and commit identifier (id) and then the ability to pull the protobuf out of Solr to then instantiate and process the information of that particular feature. Additional processing took the features as input and generated the HTML representation of that feature directly in Solr; that processing used to take place outside of Solr and was then imported into it by a separate tool.

Ultimately, the final processing workflow that was implemented to process all files of all commits of all repositories to generate all required features and their HTML representations consisted of the following steps:

1. Check the input list of the Uniform Resource Locator (URL) and see if any edits are detected. This list of URLs pointed to Android repositories on GitHub.
2. Check if we need to delete an existing repository or create a directory for a new repository.
3. Clone from GitHub any new repositories that appeared as the result of the previous two steps.
4. Perform a "git pull" operation on all repositories stored in our local file system.
5. Perform a series of git operations to detect if new commits are available for a given repository.
6. Create a new record in Solr to store metadata about any new commits detected.
7. Examine all new commits for the files modified (or created/deleted) within the new commits.
8. Create a new record in Solr to store metadata about all such files.
9. Perform feature extraction on all new file records generated by the previous step.
10. Generate HTML representations for all new files created by the previous step.

11. Generate HTML representations for all new “deltas” created by the new commits. That is if file A in commit Z is a new version of file A in commit Y, then generate an HTML representation of the difference between those two files. Do this for all diffs created by the new commits.

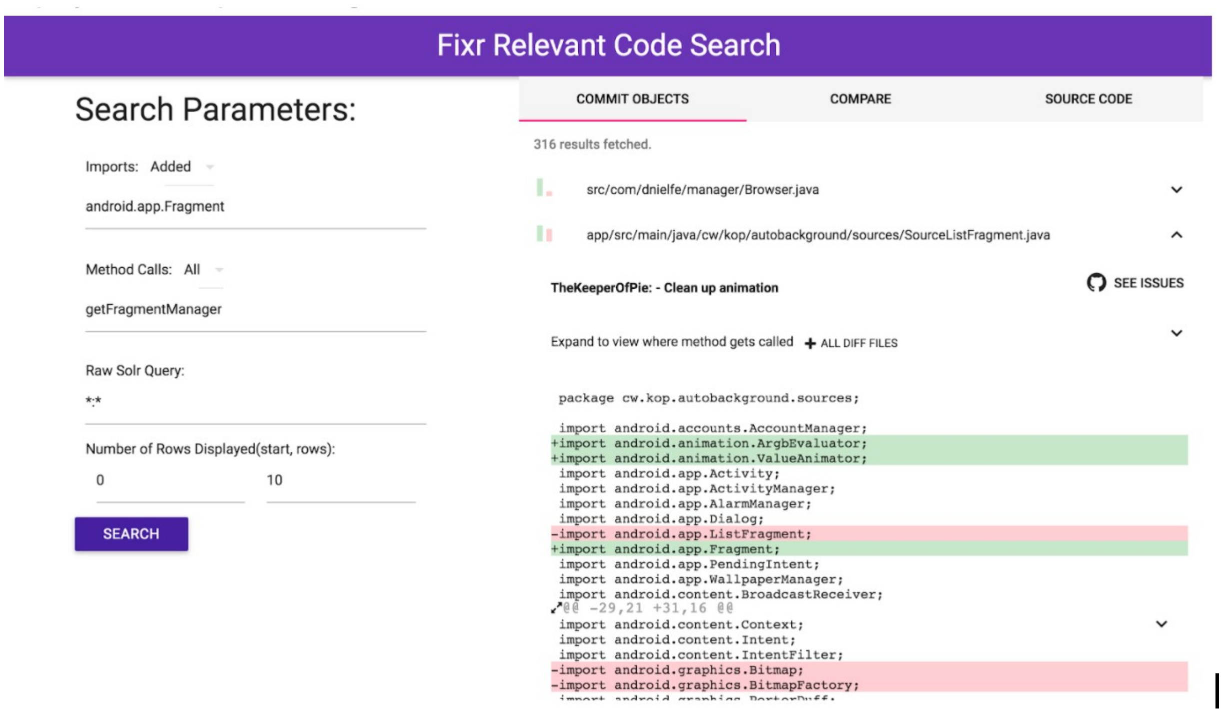
An important thing to note is that these steps could be run in any order and could all be active at the same time. Each step has a way to query Solr asking for artifacts of a given type that are produced by the step “ahead” of it. If no such artifacts are available, then the step did nothing. If artifacts were available, then the step would process just a small number of them and then circle back to see if it had more artifacts to process. In this way, any particular invocation of a processing step would check to see if it had work to do and then work until it had completed processing of all its input artifacts. It was, of course, potentially generating output artifacts of its own and, if so, before it ended, it would notify the next step that there was work for it to perform.

The infrastructure was configured such that the first step would be run once a week since the list of URLs in our corpus changed very slowly while the step to perform a “git pull” was configured to run once per day. When placed on a powerful machine with many cores, this infrastructure could max out the work of all cores when performing the initial processing of the corpus and then could very quickly process any new commits coming in on a given day. The steady state of the infrastructure was then one that did not involve a lot of computational power for most processing of new commits but had built in elasticity if a large number of commits showed up at once.

3.6 Presenting Relevant Code Search with Interactive Fixr

Use Cases. The main user of the tool is an app developer (though in principle the tools could support users in exploring the data produced by Fixr). The main use case involves an app developer who wants to address a bug (i.e., an incorrect use of some API method) figures out the correct usage of that API method. For example, the method used to start the search can be the one (or the ones) involved in a bug the developer is experiencing.

The relevant commit search system, see Figure 25, is an interactive tool that allows a developer to iteratively refine a snippet of code (the query): from an existing snippet of code (it could be a single method). The developer performs a search, observes the results of the search and uses them to refine the snippet of code. This in turns triggers a new search, and this process is iterated until the developer finds the snippet of code he is interested in. The *query* at the end is not just the snippet of code, but it is formed by several objects selected by the developer (e.g., the snippet of code, the list of commits that are more interesting, the period of time considered in the search).


Fixr Relevant Code Search

Search Parameters:

Imports: Added

android.app.Fragment

Method Calls: All

getFragmentManager

Raw Solr Query:

:

Number of Rows Displayed(start, rows):

0 10

SEARCH

COMMIT OBJECTS | COMPARE | SOURCE CODE

316 results fetched.

src/com/dnielfe/manager/Browser.java

app/src/main/java/cw/kop/autobackground/sources/SourceListFragment.java

TheKeeperOfPie: - Clean up animation

SEE ISSUES

Expand to view where method gets called + ALL DIFF FILES

```
 package cw.kop.autobackground.sources;

 import android.accounts.AccountManager;
+import android.animation.ArgbEvaluator;
+import android.animation.ValueAnimator;
 import android.app.Activity;
 import android.app.ActivityManager;
 import android.app.AlarmManager;
 import android.app.Dialog;
-import android.app.ListFragment;
+import android.app.Fragment;
 import android.app.PendingIntent;
 import android.app.WallpaperManager;
 import android.content.BroadcastReceiver;
@@ -29,21 +31,16 @@
 import android.content.Context;
 import android.content.Intent;
 import android.content.IntentFilter;
-import android.graphics.Bitmap;
-import android.graphics.BitmapFactory;
```

Figure 25: A Prototype Implementation of Relevant Code Search with Interactive Fixr

Interface Design. The user interface consists of for different panels: (1) the code snippet panel, (2) the API trend panel, (3) the commits panel, and (4) the patterns panel:

Code Snippet Panel. The code snippet panel allows the developer to insert a snippet of code. The snippet of code corresponds to the "current knowledge" the developer has to complete her task. For example, when the search starts the snippet of code could be just a single API method.

API Trend Analysis Panel. This panel contains the graphs obtained from the trend analysis on the commits documents found by querying our dataset using the content of the code snippet panel. For example, if the code snippet panel contains the API getResources, the API trend panel will show a graph that represents how many commits involve the use of getResources have been added/removed/changed in a given period of time. The trend graph is in the form of a bar chart where each bar correspond to a pre-determined period (e.g., 3 months), and the content of the bar represents how many commits involve the query terms that have been added/removed/modified in that period of time.

Commits Panel. The commit panel shows a ranked list of commits that have been obtained with the current query.

Patterns Panel. The panel shows a list of the most likely patterns of usage for the current query.

Iterative Refinement of a Query. The ultimate goal of the developer is to search for the right pattern of usage for a set of API methods. Hence, the result is really what will be shown in the Patterns panel. The user can refine the search using the interactive panels as follows:

- The user writes a code snippet (e.g., some import statement, or just an API method call). After updating the snippet of code, the system should search for commits relevant to the code snippet and display them in both the API Trend Analysis panel and the Commits panel.

- The user selects a range of time she is interested in (e.g., from the API Trend Analysis). In turn, this filters the relevant commits shown in both the API trend analysis and the Commits panel. Also, this possibly triggers the search of a better pattern of usage.
- The user inspects the relevant commits (the one in the commits panel), selecting the ones that seem more promising for his task. This updates the search.
- The user inspects the relevant commits, and from one of them selects part of the involved commits (e.g., a set of important API method calls). This action triggers a new search and can in principle update the code snippet used as query. By selecting a commit and some of its program elements, the user is trying to build her own snippet of code.

Ideally, the developer iteratively constructs her own snippet of code by iteratively exploring the data found by the relevant code search. The system works as a feedback loop: the user changes the input (the query) by performing some actions on the UI, which also have an effect back on results of the search.

Prototype Implementation. The interactive tool presented above was developed with popular open-source web application frameworks that consist of React.js, Webpack, and Apache Solr. React.js is a powerful front-end tool for rendering huge amount of data, providing a better visualization of fetched results instead of presenting the results in raw JavaScript Object Notation (JSON) form.

4 RESULTS AND DISCUSSION

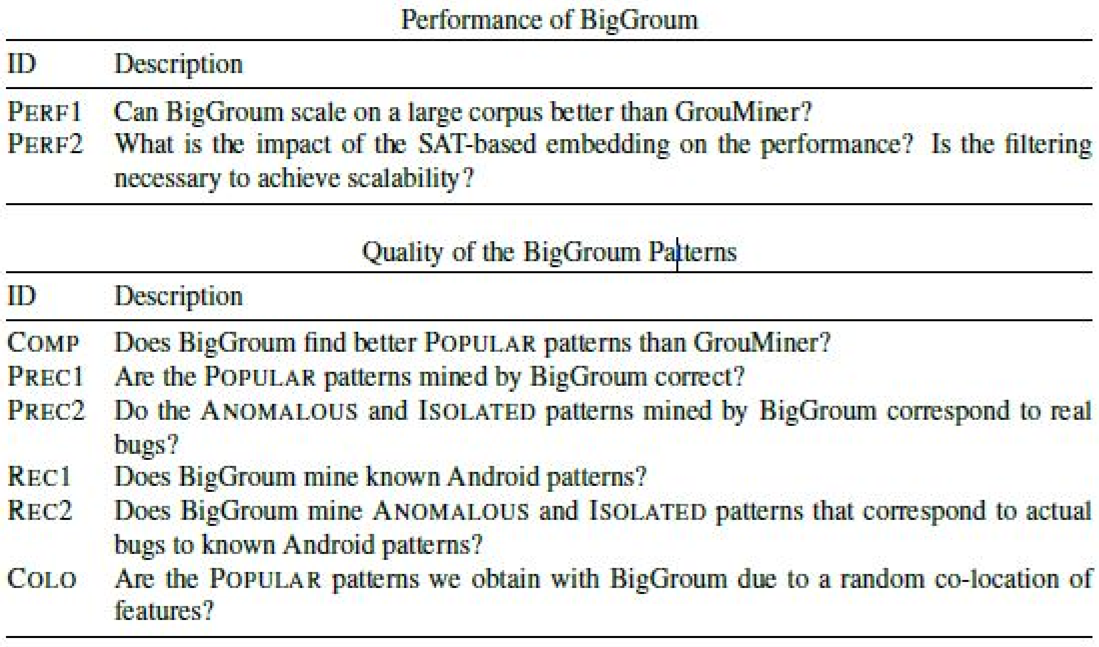
4.1 Evaluating the Performance and Quality of BigGroum Mining

We implement the BigGroum approach with different tools, written in Java, Python and C++. BigGroum extracts a groum for each method of each class of an Android app. BigGroum works directly with Java and Android bytecode using the Soot front end for Java [53]. The groums are sliced to remove statements and control structures irrelevant to the Android API. BigGroum implements the clustering using frequent itemset mining and the construction of bins, the lattice, and the resulting classification algorithm as described in Section 3.1.2. The embedding computation uses the Z3 satisfiability modulo theories (SMT) solver [54] as the underlying SAT solver. BigGroum presents the resulting patterns and anomalies as HTML pages to allow a user to examine them.

Research Questions. The main research question we want to address in this evaluation is as follows:

> Can we mine patterns of usage as grooms for a complex framework such as
> Android from a large, heterogeneous corpus of apps?

We decompose this research question with the questions shown in Figure 26. Throughout this section, we refer to the approach used in this report as BigGroum and to the approach of Nguyen et al. [26] as GrouMiner.



Performance of BigGroum

ID	Description
PERF1	Can BigGroum scale on a large corpus better than GrouMiner?
PERF2	What is the impact of the SAT-based embedding on the performance? Is the filtering necessary to achieve scalability?

Quality of the BigGroum Patterns

ID	Description
COMP	Does BigGroum find better POPULAR patterns than GrouMiner?
PREC1	Are the POPULAR patterns mined by BigGroum correct?
PREC2	Do the ANOMALOUS and ISOLATED patterns mined by BigGroum correspond to real bugs?
REC1	Does BigGroum mine known Android patterns?
REC2	Does BigGroum mine ANOMALOUS and ISOLATED patterns that correspond to actual bugs to known Android patterns?
COLO	Are the POPULAR patterns we obtain with BigGroum due to a random co-location of features?

Figure 26: Research Questions for Evaluating BigGroum Mining

We first ask if BigGroum scales better than GrouMiner when mining a large corpus of groums (PERF1), and the role of the SAT-based embedding check and the filtering on the overall performance of BigGroum (PERF2).

We then focus on the quality of the patterns mined by BigGroum. The COMP question compares the POPULAR patterns mined by BigGroum and GrouMiner (when it can compute the patterns), in terms of number and quality (e.g., size, frequencies) of the patterns found. In the PREC1 and PREC2 questions, we evaluate the *precision* of the mined patterns. We evaluate the precision of a pattern by assigning it to one of the following categories:

OBLIGATORY. Common usage patterns that lead to serious defects such as crashes or security vulnerabilities when not respected.

BESTPRACTICES. Common usage patterns that lead to undesirable user experience when not respected.

CUSTOMARY. Common usage patterns that are followed by Android developers to achieve an accepted user experience (e.g., color schemes, windows with titles, notifications).

UNTRUE. Patterns formed by a purely accidental collocation of methods or weakly related methods.

Intuitively, OBLIGATORY, BESTPRACTICES, and CUSTOMARY are all "correct" patterns of usage of the APIs, while UNTRUE patterns are "wrong" patterns of usages. This classification is more fine grained than the one that partitions the patterns as "correct" or "wrong" and further helps to understand how the mined patterns could be used in a client (e.g., a bug detector may only use OBLIGATORY patterns, while a code search or automatic code completion tool may consider all of them).

Questions REC1 and REC2 ask if well documented patterns in Android can be found by BigGroum, and they measure the recall of the approach. Clearly, we do not know *a priori* the patterns represented in our corpus or all the existing patterns in Android. Thus, we compute the recall measure on a subset of known reference patterns.

Finally, the research question COLO asks if the POPULAR patterns BigGroum mines are due to an accidental placement of program features. We empirically disprove the previous hypothesis testing that the POPULAR patterns are statistically significant. That is, we test that it is very unlikely to obtain a POPULAR pattern assuming a random generative model for groums. Statistical significance implies that our mined data cannot be produced from a random process. Clearly, statistical significance does not imply that the mined patterns are correct, since other generative models, different from the random one, have not been rejected.

Technically, we first define a (generative) *null model* for groum patterns, and we set a level of significance α (it is set arbitrary to 0.01). Then, we test if the probability to obtain a POPULAR pattern assuming the null model is lower than α. If this condition holds, then we reject the statement that popular patterns can be likely generated using the null model, hence discharging such assumption. We present our null model, the p-value computation, and the results we obtained later in this section.

Experimental Setup. We consider a corpus of 542 Android open source apps from GitHub. We crawled GitHub searching for repositories containing Android apps that were rated with at least 5 stars to bias our corpus towards "good quality" apps. Since BigGroum works on bytecode, we tried to compile the apps (with the `gradlew` command), keeping only the ones that compile, for a total of 542 apps. We extracted a total of $70{,}000$ groums from each declared method of the 542 apps that contained at least one call to an Android API method.

We obtained the input required by the GrouMiner tool by slicing the app code in the corpus to produce Java code containing only the method of interest and the class members accessed by the method. However, this slicing applied on the source code does not remove calls to app defined methods. In Android apps usually the calls to app methods constitute a small part of a single app, and hence we expected to also obtain patterns for Android APIs from GrouMiner.

We first performed an initial experiment where we run both GrouMiner and the mining algorithm of BigGroum (i.e., BigGroum without the clustering phase) on the entire corpus of groums. After 72 hours of execution, both the approaches failed to terminate. This result implies that (i) the original implementation of GrouMiner [26] *cannot scale* on the corpus of $70{,}000$ groums; and (ii) the clustering of groums using the frequent itemset *is necessary* to scale to large corpora of apps.

All the data presented in the rest of the evaluation is obtained by first computing the clusters using the frequent itemset computation, and then running the groum mining algorithm of BigGroum and GrouMiner on each cluster separately. We refer to this configuration of GrouMiner as GrouMiner$^+$, since it extends the original approach of Nguyen et al. [26] with up-front groum clustering. We set a timeout of 5 hours for the computation of the patterns of a single cluster (i.e., for each cluster, we run GrouMiner$^+$ and BigGroum for at most 5 hours). In the experiments, we used the parameters $f_I = 20, f = 20$, $L = 5$, and $K_I = 2$ (i.e., we created clusters of groums that share 2 or more methods with the corresponding itemsets) for BigGroum. We chose these values running BigGroum on a smaller corpus of groums.

The frequent itemset computation generated 194 clusters in 60 seconds. The largest cluster had $1{,}730$ groums with 22% of the clusters having 100 groums or more. The smallest cluster had 48 groums. Likewise, the largest itemset had 20 Android API methods in it, whereas the smallest itemset had 3 methods.

4.1.1 Results: BigGroum Performance.

Performance Comparison with GrouMiner$^+$ (PERF1). In the scatter plot shown in Figure 27, we compare the performance of BigGroum and GrouMiner$^+$. Each point in the plot compares the times (seconds) for BigGroum (y-axis) and GrouMiner+ (x-axis) to compute a cluster, where a point is on the dashed line if the corresponding approach did not terminate before the timeout (300 minutes, note that GrouMiner+ timed out on 11 clusters). Overall, BigGroum computed the frequent sub-graphs for *all* the clusters in 95 minutes, whereas GrouMiner$^+$ took 413 minutes for 183 out of 194 clusters, timing out for the remaining 11 (the time out is 300 minutes). On average, BigGroum computed the patterns for a cluster in 0.5 minutes, while GrouMiner$^+$ took 2.3 minutes (the average for GrouMiner$^+$ is only computed for the clusters where GrouMiner$^+$ did not time out).

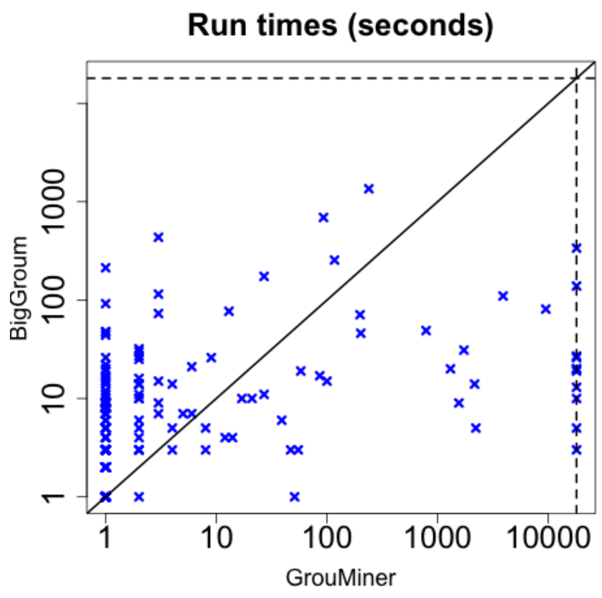


Figure 27: Comparing the Execution Time of BigGroum and GrouMiner

We conclude that BigGroum scales better than GrouMiner$^+$. We conjecture that the bottom-up mining approach of GrouMiner$^+$ must enumerate a large number of smaller patterns before finding the larger popular patterns, requiring more computational effort. We note that there are also some clusters wherein GrouMiner$^+$ computes the patterns almost immediately (i.e., in less than a second) and faster than BigGroum. However on these clusters, BigGroum always terminates well within the timeout.

SAT Solver Performance (PERF2). We compare the total time taken by BigGroum for the pattern mining and classification phase with the total time taken by the calls to the SAT solver. We also compare the total number of checks and the total number of checks that had to call the SAT solver in Table 1.

Table 1: SAT Solver Performance in BigGroum Mining

Average/Maximum Graph Sizes	Total Time (s)	SAT Time (s)	Total $\preceq$ Checks	$\preceq$ Checks with SAT
186/456	5695	3042	6744259	119250

About 1.77%, a tiny fraction, of the checks had to directly call the SAT solver. At the same time, however, the total time taken by the SAT solver for these calls is about 53.4% of the overall computation time (95 minutes).

4.1.2 Results: Quality of BigGroum Patterns.

Table 2: Popular Patterns Found by BigGroum and GrouMiner

POPULAR patterns		
Approach	Total Patterns	Minimum/Average/Maximum Pattern Size
BigGroum	410	4/4.9/10
GrouMiner$^+$	85	2/2.4/6

Pattern Comparison with GrouMiner (COMP). In Table 2, we compare the number and sizes of POPULAR patterns found by GrouMiner$^+$ against those found by BigGroum, while in Figure 28 we compare the distribution of groum sizes in terms of number of method nodes. BigGroum finds 410 POPULAR patterns, while GrouMiner$^+$ finds 85. On average, BigGroum patterns have method nodes versus 2.4 API methods for GrouMiner$^+$. For each GrouMiner$^+$ pattern, we examine if BigGroum can find the same or a more complete pattern. BigGroum finds 72/85 patterns found by GrouMiner$^+$. We manually examined the remaining 13 patterns to understand why BigGroum did not discover them: (i) 7 patterns involved an API method call that *was not part* of a frequent itemset and thus was sliced away in BigGroum (changing the frequency cutoff for popular patterns could address these discrepancies); (ii) 2 patterns contained app-specific methods, 2 others contained methods from the Java (and not Android) API, and 2 patterns contained methods without a precise type signature.

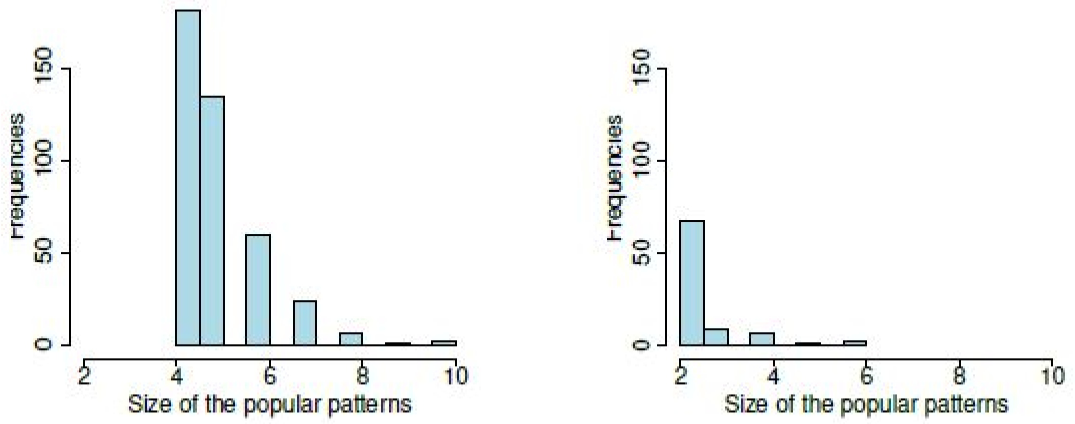


Figure 28: Size Distribution of Popular Patterns Found by BigGroum (left) and GrouMiner (right)

BigGroum finds more patterns than GrouMiner$^+$ for the following reasons: (i) BigGroum tracks base types for app classes that inherit from an Android class, enabling us to compare object types across apps, unlike GrouMiner$^+$; (ii) even though we slice the GrouMiner$^+$ input, some of the app-specific method calls are left over, nevertheless. These are sometimes popular enough for the given cutoff frequencies.

Precision of the BigGroum Patterns (PERF1 and PERF2). To evaluate the precision of the approach, we manually inspected the patterns found by BigGroum for 30 (out of the 194) randomly selected clusters. We first analyze the POPULAR patterns. We manually assigned the category, OBLIGATORY, BESTPRACTICES, CUSTOMARY and UNTRUE, to the *most* frequent and the *least* frequent POPULAR pattern in the clusters (a cluster may contain several POPULAR patterns). For the OBLIGATORY patterns, we then investigated the other ISOLATED and ANOMALOUS patterns in those clusters to check if they were actual defects.

The bar chart in Figure 29 summarizes the outcome of our manual evaluation for the POPULAR patterns: the first bar in the plot shows the distribution of the patterns with the highest frequency, while the second bar shows the distribution of the patterns with the lowest frequency, both divided in the 4 different categories. The plot shows that the precision of BigGroum does not change if we consider patterns with different frequency (i.e., the frequency cutoff *f* is adequate).

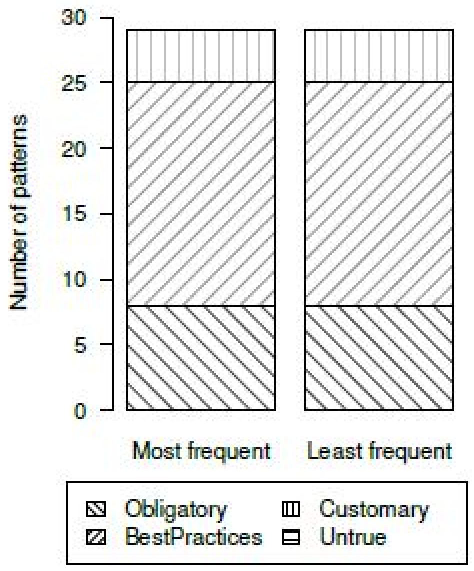
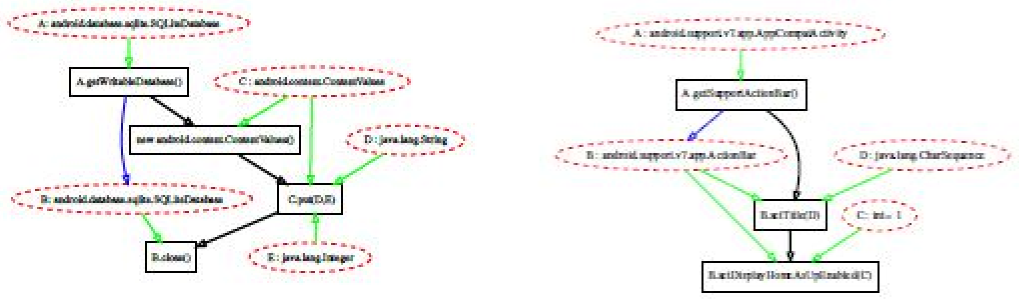


Figure 29: Categorization of the Most and Least POPULAR Patterns

In the following, we discuss the results for the most frequent POPULAR patterns. We found at least one popular pattern in 29 out of the 30 clusters examined. The cluster defined by the methods setOnClickListener, setText, and setTextColor failed to have any popular patterns. There is no prescribed order for the three setter methods involved, and further, only a subset of these methods may be called. This yields a large number of possible patterns, none of which exceed our frequency cutoff to be popular.

We found 8/29 OBLIGATORY patterns, and Figure 30a shows one of them: the pattern [55] shows the protocol for opening a database, creating a new value, inserting the value in the database, and closing the database.

Most of the patterns examined (17/29) are BESTPRACTICES that describe code snippets to accomplish a well-defined, specific task. Figure 30b shows an example of a BESTPRACTICES pattern to retrieve an Activity toolbar, setting its title and adding a navigation button back to the app's home screen [56]. Clearly, the pattern is used by several apps. A deviation from this pattern does not necessarily cause a serious defect but presumably leads to a poorer user experience.


A: android.database.sqlite.SQLiteDatabase
A.getWritableDatabase()
C : android.content.ContentValues
new android.content.ContentValues()
D : java.lang.String
B: android.database.sqlite.SQLiteDatabase
C.put(D,E)
B.close()
E : java.lang.Integer
A : android.support.v7.app.AppCompatActivity
A.getSupportActionBar()
B : android.support.v7.app.ActionBar
D : java.lang.CharSequence
B.setTitle(D)
C: int.. 1
B.setDisplayHomeAsUpEnabled(C)

(a) An OBLIGATORY Protocol for Working with an `SQLiteDatabase` Object in Android

(b) A BESTPRACTICES Protocol to Set an Activity Toolbar

Figure 30: OBLIGATORY and BESTPRACTICES Patterns Mined by BigGroum

Four patterns out of 29 were categorized as CUSTOMARY. In one such pattern the method android.util.Log.d is frequently called with the method putExtra used to create and modify an android.content.Intent object (used for interprocess communication). It is clear that developers often insert log messages to help them better debug Intents.

We did not find any POPULAR pattern categorized as UNTRUE, although an ongoing thorough examination of all the 410 patterns may provide us such examples.

Table 3: Bugs in ANOMALOUS and ISOLATED Patterns

Category	Total Patterns	Total Patterns Containing a Bug
ANOMALOUS	34	2
ISOLATED	492	21

Next, we manually examined one representative groum (chosen randomly) for each pattern inside the clusters categorized as ANOMALOUS and ISOLATED, searching for violations that could be potential bugs. In Table 3, we see that 6% of the ANOMALOUS and 4% of the ISOLATED patterns correspond to real bugs in the usage of the APIs. We found several bugs wherein the developer omitted a call to the close method on a database object in the protocol shown in Figure 30a [42]. We also encountered several patterns that did not contain a bug: in almost all these cases the database was eventually closed by another method in the same class. These results show a limitation of our current approach. From our manual inspection, the cause of imprecision of our approach is caused by the fact that the groums are obtained from a single method in the app (i.e., the groum extraction is intra-procedural), and hence does not capture the real execution of an app (e.g., what methods are invoked before and after).

Recall of the BigGroum Patterns (REC1 and REC2). We evaluated the recall of BigGroum by considering 15 known reference patterns. We collected the names of the Android methods contained in our corpus, we selected a subset of them randomly, and we then searched for their usages on the Android documentation and StackOverflow. The list of patterns is reported in Table 4, together with their categorization; OBL: OBLIGATORY, BES: BESTPRACTICES, T: Total Number of Patterns Matching the Reference Pattern, and f: Average Frequency.

Table 4: BigGroum Recall on Reference Patterns

	Reference Pattern	Category	POPULAR		ANOMALOUS	ISOLATED
			T	f	T	T
1	DB Transaction	OBL	1	38	1	0
2	Get/Release Cursor	OBL	8	37.6	7	78
3	Fragment Transaction	OBL	10	66	2	30
4	Show Toast	OBL	19	52	4	99
5	Show/AlertDialog	OBL	20	32	21	250
6	Retrieve/Release Parcel	OBL	1	29	0	0
7	Create/Send Intent	BES	1	26	0	0
8	Retrieve from backstack	BES	0	0	0	0
9	Query ContentProvider	BES	3	35	0	24
10	Insert ContentProvider data	BES	0	0	0	0
11	Update ContentProvider data	BES	0	0	0	0
12	Delete ContentProvider data	BES	0	0	0	0
13	Build/send notification	OBL	3	42.3	0	32
14	Restore Preferences	BES	2	38.5	0	12
15	Edit Preferences	OBL	5	53	2	81

To evaluate REC1, we first searched for the occurrence of each reference pattern among the POPULAR patterns discovered by BigGroum. Then, we evaluate REC2 by analyzing the ANOMALOUS and the ISOLATED patterns in the same clusters where the reference patterns were found to be popular.

Table 4 shows the total number of POPULAR patterns that contains a reference pattern, with their average frequencies, and the number of ANOMALOUS and ISOLATED patterns found in the same clusters (of the POPULAR patterns). We see that BigGroum finds at least one POPULAR pattern for 11 out of the 15 reference patterns. However, 4 out of 15 reference patterns did not have any corresponding POPULAR pattern, since there are few instances of these patterns in the corpus and hence they did not pass the frequency cutoff to be labeled as POPULAR. Thus, it seems that we miss 4 reference patterns because we do not have enough data in the dataset of apps, and not because BigGroum does not mine them. BigGroum also finds ANOMALOUS and ISOLATED patterns discovering possible wrong usages of the reference patterns.

Accidental Co-location of POPULAR Patterns (COLO). We show that BigGroum did not mine POPULAR patterns only because of a random co-location of features (e.g., the existence of a method call). We experimentally validate our claim first defining a *null model* H_0, a probabilistic model that generates random groum patterns, and then computing the probability of obtaining the mined POPULAR patterns with such a model. If this value is lower than a level of significance threshold α, we can reject the null hypothesis (i.e., that it is likely that the POPULAR patterns have been obtained with the model H_0).

Since generative models for graphs are difficult to interpret and use, we define a simpler generative model that considers the existence of Boolean features in a groum. We include what we consider fundamental and core features of the groum: the presence in the groum of an API method invocation and the presence in the groum of a control (and transitive control) edge.

Consider again the set of all the API methods M = {f$_1$, . . . , f$_K$}. We assume a finite set of method invocations that we obtain over the observed corpus $\mathscr{L}$: $\{H_1, \ldots, H_{N_g}\}$ of N_g groums. The set of features we consider are:

$$\mathcal{X}_m := \{x_f \mid f \in M\} \tag{1}$$

$$\mathcal{X}_e := \{x_{(f_i, f_j)} \mid f_i, f_j \in M\} \tag{2}$$

$$\mathcal{X} := \mathcal{X}_m \cup \mathcal{X}_e \tag{3}$$

where $\mathcal{X}_m$ and $\mathcal{X}_e$ are sets of discrete random variables with a Bernoulli distribution with values 0 and 1. Let $x_f = 1$ if the pattern contains the method invocation f, and otherwise is 0. Similarly, let $x_{(f_1, f_j)} = 1$ if the pattern contains the control edge from a method invocation node f_i to a method invocation node m_j.

H_0 model assumes a Bernoulli distribution for each feature. The probability of generating a groum H assuming the null model H_0 (p-value) is as follows:

$$P(H) \coloneqq P\Big(\bigcap_{x \in \mathcal{X}_H} x = 1 \ \cap \bigcap_{x \notin \overline{\mathcal{X}_H}} x = 0\Big). \tag{4}$$

where $\mathcal{X}_H$ is the set of features from $\mathcal{X}$ that appears in H.

Computing such a joint probability is difficult, so we make the simplifying assumption that the events of seeing an edge $x_{(f_i, f_j)}$ is conditionally independent from seeing another edge $x_{(f_l, f_m)}$. The appearance of an edge is still dependent to the appearance of a method invocation node. We rewrite $P(H)$ with the independence assumption as follows:

$$P(H) \coloneqq \prod_{e \in \mathcal{X}_e \cap \mathcal{X}_H} P(e = 1, m_1 = 1, \ldots, m_n = 0) \tag{5}$$

$$\prod_{e \in \mathcal{X}_e \cap \overline{\mathcal{X}_H}} P(e = 0, m_1 = 1, \ldots, m_n = 0) \tag{6}$$

where $m_1 = 1, \ldots, m_n = 0$ are the random variables of the method invocation nodes (i.e.,

$$\bigcap_{m \in \mathcal{X}_m \cap \mathcal{X}_H} m = 1 \ \cap \bigcap_{m \in \mathcal{X}_m \cap \overline{\mathcal{X}_H}} m = 0\,).$$

We can then compute the join probability for each single edge as follows:

$$P(e = 1, m_1 = 1, \ldots, m_n = 0) \coloneqq \frac{P(e=1 \mid m_1=1, \ldots, m_n=0)}{P(e=1)} P(m_1 = 1, \ldots, m_n = 0) \tag{7}$$

and similarly for $P(e = 0, m_1 = 1, \ldots, m_n = 0)$. We estimate such probability from the corpus of groums $\mathscr{L}$.

If the $P(H)$ is less than α (we set it to 0.01), then we conclude that it is very unlikely to obtain H assuming the null model, and hence that H is statistically significant.

We extracted the feature set $\mathcal{X}$ from the dataset $\mathscr{L}$ and we obtained 2,357,857 edge features $\mathcal{X}_e$

(unique pairs of method nodes) and 376,701 node features $\mathcal{X}_m$. We then compute the probability for each one of the POPULAR patterns assuming H_0. Under H_0, the probability $p(H)$ for all the POPULAR patterns is lower than 0.01. An experimental package describing additional details is available from our source code repository.

4.1.3 Discussion and Threats to Validity.

We discuss the possible threats to the validity of our experimental evaluation. The choice of the corpus of apps may affect the performance and the quality of the results. We minimized the issue by selecting real, quality apps (i.e., apps with more than 5 stars on GitHub and that compile). Our evaluation is for Android, and we could have different results for other frameworks.

The experimental settings could also affect the results. First, we observe that GrouMiner was designed to work inside a single project rather than work across a larger corpus. We addressed this by slicing the source code to retain parts relevant to the Android API. Then, we tried to avoid any selection bias on the BigGroum parameters choosing the parameters on a small corpus of apps before mining the full corpus and evaluating our technique.

We manually evaluated the precision (research questions PREC1 and PREC2) on 30/194 randomly chosen patterns, finding evidence that a vast majority of the BigGroum's patterns are OBLIGATORY and BESTPRACTICES. The number of UNTRUE patterns is highly unlikely to be a large percentage, given that none were found in our sample. Furthermore, we recognize that the categorization of the patterns (in OBLIGATORY, BESTPRACTICES, CUSTOMARY, and UNTRUE) is not formal (and it cannot be defined formally) and hence it is open to interpretation. The evaluation (and categorization) of the patterns was manual and hence, prone to validity threats, such as our knowledge of complex (and often undocumented) Android API behavior. Three persons validated the mined patterns and their categorization, consulting additional documentation (e.g. StackOverflow posts, the Android source code) and in the most uncertain cases, an Android developer (a person from our research team with experience in Android development).

For the research question REC1, we further did our best to select the reference patterns in an unbiased way. We recognize that our understanding of the API usage and some of the online sources may in fact be erroneous. Two persons collected and validated the reference patterns (again, consulting additional documentation).

Finally, for the COLO research question, we defined a probabilistic model for the null hypothesis. We recognize the difficulty in defining a null hypothesis for our domain and that the assumptions we made to simplify the model, in particular the independence assumption among the edge features, may not hold in general (e.g., clearly the existence of an edge in a groum may influence the existence of another edge in the groum). Modeling such kinds of dependencies however is too complicated and counter-productive, since it would require estimating the joint probability of very specific events (e.g., that a precise set of edges exist in a groum).

4.2 Expressing Customized UI Testing Patterns

In this section, we demonstrate the utility of UI trace generators as a higher-order combinator library. We present four novel ways that a generator derived from the core language is applied to address a specific issue in Android app testing. For each, we discuss real, third-party reported issues in open-source apps that motivate the conception of this generator.

4.2.1 Exceptions on Resume.

A common problem in Android apps is the failure to handle suspend and resume operations. These failures are most commonly exhibited as app crashes caused by (1) *illegal state exceptions*, when an app's suspend/resume routines do not make correct assumptions on a sub-component's lifecycle state, or (2) as *null pointer exceptions*, typically when an app's suspend/resume routines wrongly assumes the availability of certain object resources that did not survive the suspend operation or was not explicitly restored during the resume operation. From a software testing perspective, the Android app's test suite should include sufficient test cases that exercises it's suspend/resume routines. As illustrated in our example in Section 3.3.1, Android apps are stateful event-based systems, so conducting suspend and resume operations at different points (states) of an Android app may result in entirely different outcomes. Test cases must provide coverage for suspend/resume routines at crucial points of the test app (e.g., login pages, while performing long background operations).

To simplify the development of trace generator sequences that test the app's ability to handle interrupt events, we derive a specific combinator *>> similar to sequencing :>> but additionally insert suspend and resume events in a *non-deterministic* manner.

We have observed numerous issue tracker cases on open Android GitHub projects that report failures that are exactly caused by this issue (illegal state exceptions or null pointer exceptions on app resume). One example is found in Tower [57], which is a popular open-source mobile ground control station app for UAV drones. A past issue[10] of the Tower app documents a null-pointer exception that occurs when the user suspended and resumed the app from the app's main map interface. This failure is caused by a wrong assumption that references to the map UI display fragment remain intact after the suspend/resume cycle.

Another example worth noting is the Nextcloud open-source Android app[11], which provides rich and secured mobile access to data (documents, calendars, contacts, etc) stored (by paid users) in the company's proprietary cloud data storage service. A recent issue[12] reports a crash of the app during a file selection routine when the user re-orients the app from portrait to landscape. This crash is the result of a null-pointer exception on a reference to the file selector UI object (OCFileListFragment), caused by the failure of the app's resume operation to anticipate the

[10] Fredia Huya-Kouadio. FlightActivity NPE in onResume #1036. https://github.com/DroidPlanner/Tower/issues/1036. September 2, 2014
[11] Nextcloud. https://github.com/nextcloud/android
[12] Andy Scherzinger. FolderPicker - App crashes while rotating device #448. https://github.com/nextcloud/android/issues/448. December 13, 2016

destruction of the object after suspension.

We have developed test generators for Nextcloud and reproduced this crash (#448) with the following trace generator (with the login sequence omitted for simplicity):

```
(Login Sequence) *>>
LongClick(R.id.linearlayout) *>>
Click("Move")
```

Note that other recent work [58] has also identified injecting interrupt events as being critical to testing apps. While our (current) implementation of *>> is less sophisticated than Adamsen et al. [58], the advantage of the ChimpCheck approach is that *>> is simply a derived combinator that can be placed alongside other scripted pieces (e.g., for the login sequence). The *>> provides a basic demonstration of how more complex test generators that address real app problems can be implemented in ChimpCheck.

4.2.2 Preserving Properties.

An app's proper functionality may frequently depend on its ability to preserve important properties across certain state transitions that it is subjected to. For instance, it would be a very visible defect if key navigational buttons of the app vanish after a user suspended and resumed the app. While this issue seems very similar to the previous (i.e., a failure caused by interrupt events), the distinction we consider here is that the issue does not result in an app crash. Hence, simply testing against interrupt events (via the *>> combinator) may not detect any faults. Since the decision of which UI elements should "survive" interrupt events is app specific, we cannot expect to build in such a property but instead derive customizable generators that allow the test developer to program such tests more effectively and efficiently. Rather than writing boiler-plate assertions before and after specific events, we derive a generator that expresses the test sequences in a more general manner.

We have observed many instances of this "vanishing UI element" scenario described above. Just to name a few popular apps in this list: Pokemap[13], a supporting map app for the popular Pokémon Go game; c:geo[14], a popular Geocaching app with 1M-5M installs from the Google Play Store as of August 23, 2017; and Tusky[15], an Android client for a popular social-media channel Mastodon. Each app at some point of its active development contained a bug related to our given scenario. For Pokemap, issue #202[16] describes a text display notifying your success in locating a Pokémon permanently disappears after screen rotation. For CGeo, issue #2424[17] describes an opened download progress dialog is wrongly dismissed after screen rotation, leaving the user uncertain

[13] Omkar Moghe. Pokemap. https://github.com/omkarmoghe/Pokemap
[14] c:geo. https://github.com/cgeo/cgeo
[15] Andrew Dawson. Tusky. https://github.com/Vavassor/Tusky
[16] Andy Cervantes. Flipping Screen Orientation Issue - Pokemon Found Stops Being Displayed #202. https://github.com/omkarmoghe/Pokemap/issues/202. July 26, 2016
[17] Ondˇrej Kunc. Download from map is dismissed by rotate #2424. https://github.com/cgeo/cgeo/issues/ 2424. January 22, 2013

about the download progress. For Tusky, issue #45[18] states that replies typed into text inputs are not retained after screen rotation. We found that we could reproduce issue #45 in Tusky with the following simple generator:

```
. . . :>> Type(R.id.edit_area,"Hi") :>>
Rotate preserves hasText(R.id.edit_area,"Hi")
```

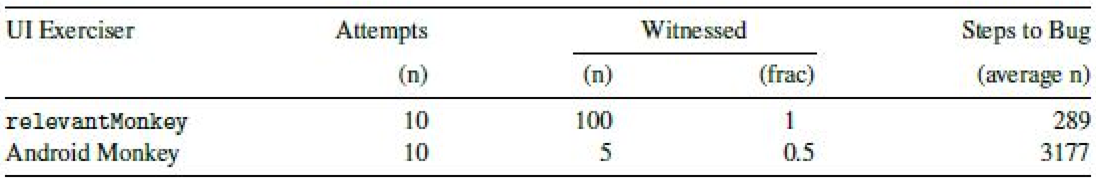
Integrating Randomized UI Testing. While writing custom test scripts is often necessary for achieving the highest possible test coverage, black-box techniques like pure randomized UI testing [27] and model learning techniques [30, 32] are nonetheless important and an effective means in practice for providing basic test coverage. Industry-standard Android testing frameworks (e.g., Espresso and Robotium) provides little (or no) support for integrating with these test generation techniques, which unfortunately forces the test developer to use the various possible testing approaches in isolation.

In ChimpCheck, we can easily define a monkey combinator that is similar to Android's UI Exerciser Monkey [27] in that it generates random UI events applied to random locations on the device screen. But we can also just as easily define a combinator that generates random but more relevant UI events by relying on ChimpCheck's primitive * combinator to infer relevant interactions from the app's run-time view hierarchy, which we will call relevantMonkey. Having randomized test generators like these as combinators provides the developer with a natural programming interface to integrate these approaches with her own custom scripts. For instance, revisiting our example in Section 3.3.1 (or similarly for the Nextcloud app in Section 4.2.1), getting past a login page is the hurdle to using a brute-force randomized testing, but we can implement the necessary traces to the media pages by simply the following generator:

```
Click(R.id.enter) :>>
Type(R.id.username,"test")  :>>
Type(R.id.password,"1234") :>>
Click(R.id.signin) :>> relevantMonkey 50
```

This generator simply applies the relevantMonkey combinator (arbitrarily for 50 steps) after the login sequence—thus generating multiple UI traces that randomly exercises the app functionalities after applying the login sequence fixture.

Table 5: Comparing a ChimpCheck relevant and Android UI Exerciser Monkeys



UI Exerciser	Attempts	Witnessed		Steps to Bug
	(n)	(n)	(frac)	(average n)
`relevantMonkey`	10	100	1	289
Android Monkey	10	5	0.5	3177

Our preliminary studies have found that even for simple apps (e.g., Kistenstapeln[19], a score tracker

[18] Julien Deswaef. When writing a reply, text disappears if app switches from portrait to landscape #45. https://github.com/Vavassor/Tusky/issues/45. April 2, 2017
[19] Fachschaft Informatik. Kistenstapeln-Android. https://github.com/d120/Kistenstapeln-Android

for crate stacking game, and Contraction Timer[20], a timer that tracks and stores contraction data), we require generating numerous UI event sequences from Android Monkey before we get acceptable coverage results. Preliminary experiments using the relevant monkey combinator (that accesses the view hierarchy) have shown promising results shown in Table 5, where we ran each exerciser 10 times, for up to 5,000 UI events, averaging the number of steps taken only over successful attempts in witnessing the bug. For the Kistenstapeln app, the relevant-monkey combinator witnesses the bug from issue #1[21]) in an order of magnitude less generated events than the Android UI Exerciser Monkey. Note that Android Monkey failed to witness the bug in under 5,000 events in half of the attempts.

We also note that these promising preliminary results are achieved with a simplistic, light- weight implementation with a few lines of ChimpCheck script together with less than 200 lines of library code that implements view hierarchy access for the * combinator, developed within a span of two days, including time to learn the Android Espresso framework.

Injecting Custom Generators. In Section 4.2.2, we demonstrated how random UI testing techniques can be added to ChimpCheck as black-box generators. However in practice, many situations require more tightly coupled interactions between the custom scripts and the black-box techniques. For instance, many modern Android apps can contain features requiring user authentication with her account and such authentication procedures are often requested in an on-demand manner (only when the user requests contents that requires two-factor authentication). Such dynamic behaviors makes it difficult or impractical to simply hard-code and prepend a custom login script as we did in the previous sections.

From the relevantMonkey combinator, we refine it into the gorilla combinator with the ability to inject customized scripting logic into randomized testing. The gorilla combinator accepts an additional argument: a generator that is prepended before every randomized event, hence allowing the developer to "inject" custom *directives* to handle situations where a purely randomized technique may be mostly ineffective. For example, we can define a simple combinator that generates traces that randomly explores the app unless a login page is displayed. When the login page is displayed, it will append a hard-coded login sequence to effectively proceed through the login page:

```
val login = Click("Login")  :>>
  Type("User","test") :>>
  Type("Password","1234") :>> Click("Sign-in")
gorilla 50 (isDisplayed("Login") then login)
```

An example of a simple app with an authentication page is OppiaMobile[22], a mobile learning app. We have developed test generators using the refined gorilla with a hard-coded login directive (as described above). In sample runs, we observed that the gorilla occasionally logs out and logs back in between unauthenticated and authenticated portions of the page. Though no bugs were found, such log in/out loops are very rarely tested and potentially hides defects.

[20] Ian Lake. Contraction Timer. https://github.com/ianhanniballake/ContractionTimer
[21] Tobias Neidig. Crash on timer-event on other fragment #1. https://github.com/d120/Kistenstapeln-Android/issues/1. March 19, 2015
[22] Digital Campus. OppiaMobile Learning. https://github.com/DigitalCampus/oppia-mobile-android

```
gorilla 100 {
 isDisplayed("SD Card Access Framework") then {
  { Swipe(R.id.scroll,Down) :>>
    Click("Select SD Card")     } <+> ClickBack
 }
}
```

Figure 31: Getting Past a Modal Dialog by Injecting a Custom Action

This app also contains an information, modal dialog that is unfavorable for randomized testing techniques. In particular, the only way to navigate through this page is a lengthy scroll to the end of the UI view and hitting a button labeled "Select SD Card". Attempts at random exercising often ends up stuck in this problematic page. To help the gorilla function more effectively, we injected a non-deterministic choice between the only two possible actions when this dialog page is visible: (1) proceed forward by scrolling down and click the button or (2) hit the device back navigation button. In Figure 31, we show the few lines that implement injection of this application-specific way of getting past a particular modal dialog. The ChimpCheck generator says to do a random UI, exercising unless the app is showing the "SD Card Access Framework" page; and in this case, injects the specific action to either scroll to the bottom of the page and click a specific button to continue or click the back button.

Finally, another application of gorilla is that from Android 6.0 (API level 23) onwards, device permissions (e.g., SD card usage, camera, location) are granted at run-time, as opposed to on installation. The app developer is free to decide how these *dynamic* permissions are requested. Typically, an app will use a modal dialog box with an acknowledgment and reject button. Dealing with these dynamic permissions is straight-forward with the gorilla combinator using code, for example, similar to Figure 31.

4.3 Evaluating the Precision and Generality of Lifestate and Validating Models

We implement the Verivita [13] to verify protocol violations for Android applications and for validating callback control flow models. The tool: (i) instruments an Android app to record observable traces, (ii) validates a lifestate model for soundness against a corpus of traces, and (iii) assesses the precision of a lifestate model with dynamic verification. We use the following research questions (RQs) to demonstrate that lifestate is an effective language to model event-driven protocols, and validation is a crucial step to avoid unsoundness.

RQ1 *Lifestate Precision.* Is the lifestate language adequate to model the callback control flow of Android? We hypothesize that carefully capturing the app-framework interface is necessary to obtain precise protocol verification results.

RQ2 *Lifestate Generality.* Do lifestate models generalize across apps? We want to see if a lifestate model is still precise when used on a trace from a new, previously unseen app.

RQ3 *Model Validation.* Is validation of callback control-flow models with concrete traces necessary to develop *sound* models? We expect to witness unsoundnesses in existing (and not validated) callback control-flow models and that validation is a crucial tool to get sound models.

Our experimental evaluation was validated in peer review, and the full artifact containing code and data is publicly available [14].

4.3.1 RQ1: Lifestate Precision.

The bottom line of Table 6 is that lifestate modeling is essential to improve the percentage of verified traces to 83%—compared to 57% for lifecycle++ and 27% for lifecycle modeling. In Table 6, the sensitive callin column lists protocol properties by the callin that crashes the app when invoked in a bad state. A total of 50 traces from 5 applications with no crashes were collected and the sensitive column lists the number of traces where the application invokes a sensitive callin; to provide a baseline for the precision of a model, counting the number of traces without a manually-confirmed real bug in the verifiable column. There are four columns labeled verified showing the number and percentage of verifiable traces proved correct using different callback control-flow models. The lifestate columns capture our contribution, while the lifecycle++ columns capture the current practice for modeling the android framework and the bad column lists the number of missed buggy traces and is discussed further in RQ2.

Table 6: Precision of Callback Control-Flow Models

properties	non-crashing traces		callback control-flow models								
			top		lifecycle		lifestate		lifecycle++		
sensitive callin	sensitive (n)	verifiable (n)	verified (n)	verified (%)	verified (n)	verified (%)	verified (n)	verified (%)	verified (n)	verified (%)	bad (n)
AlertDialog											
dismiss	16	6	0	0	0	0	6	100	6	100	0
show	43	34	17	50	17	50	28	82	24	71	0
AsyncTask											
execute	4	4	0	0	4	100	4	100	0	0	0
Fragment											
getResources	10	10	0	0	0	0	10	100	4	40	0
getString	10	10	0	0	0	0	2	20	0	0	0
setArguments	19	19	1	5	1	5	19	100	13	68	0
total	102	83	18	22	22	27	69	83	47	57	0

Methodology. We collect execution traces from Android apps and compare the precision obtained verifying protocol violations with four different callback control-flow models. The first three models are expressed using different subsets of the lifestate language. The *top* model is the least precise (but clearly sound) model where any callback can happen at all times, like in the Automaton 24a in Section 3.4. The *lifecycle* model represents the most precise callback control-flow model that we can express only using back-messages, like in Automaton 24b. The *lifestate* model uses the full lifestate language, and hence also in-messages like in the Automaton 24d, to change the currently permitted back-messages. It represents the most precise model that we can represent with lifestate. To faithfully compare the precision of the formalisms, we improved the precision of the lifecycle and lifestate models minimizing the false alarms from verification. And at the same time, we continuously run model validation to avoid unsoundnesses, as we discuss below in RQ3. As a result of this process, we modeled the behavior of several commonly-used Android classes, including Activity, Fragment, AsyncTask, CountdownTimer, View, PopupMenu, ListView, and Toolbar and their subclasses. Excluding similar rules for subclasses, this process resulted in a total of 167 lifestate rules.

We further compare with an instance of a *lifecycle*$^{++}$ model, which refines component lifecycles with callbacks from other Android objects. Our model is a re-implementation of the model used in FlowDroid [38] that considers the lifecycle for the UI components (i.e., Activity and Fragment) and bounds the execution of a pre-defined list of callback methods in the active state of the Activity lifecycle, similarly to the example we show in Figure 6. We made a best effort attempt to faithfully replicate the FlowDroid model (the details of our implementation is available in an extended version of the Verivita paper [59]).

To find error-prone protocols, we selected *sensitive callins*, shown in the first column of Table 6, that frequently occur as issues on GitHub and StackOverflow [49, 60–64]. We then specify the lifestate rules to allow and disallow the sensitive callins.

To create a realistic trace corpus for RQ1, we selected five apps by consulting Android user groups to find those that extensively use Android UI objects, are not overly simple (e.g., student-developed or sample-projects apps), and use at least one of the sensitive callins. To obtain realistic interaction traces, we recorded manual interactions from a non-author user who had no prior knowledge of the internals of the app. The user used each app 10 times for 5 minutes (on an x86 Android emulator running Android 6.0)—obtaining a set of 50 interaction traces. With this trace-gathering process, we exercise a wide range of behaviors of Android UI objects that drive the callback control-flow modeling.

To evaluate the necessity and sufficiency of lifestate, we compare the verified rates (the total number of verified traces over the total number of verifiable traces) obtained using each callback control-flow model. We further measure the verification run time to evaluate the trade-off between the expressiveness of the models and the feasibility of verification.

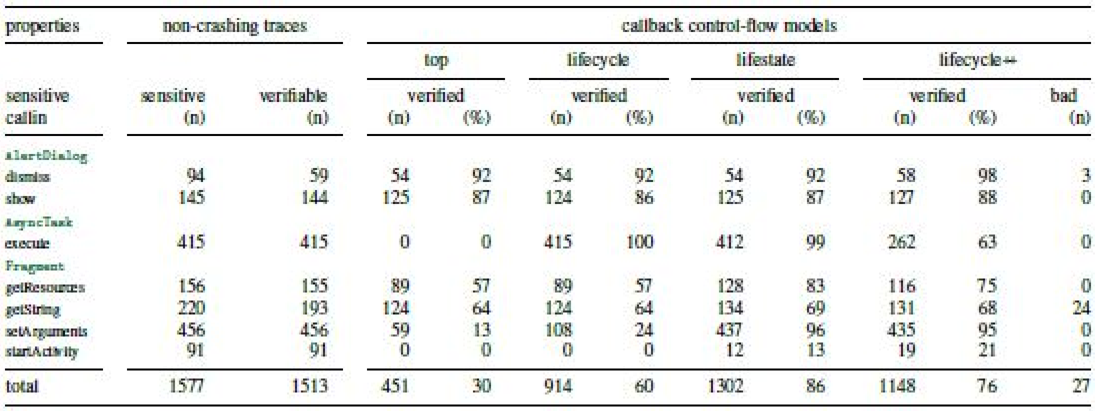
Discussion. In Table 6, we show the number of verified traces and the verified rates broken down by sensitive callins and different callback control-flow models—aggregated over all apps. As stated earlier, the precision improvement with lifestate is significant, essential to get to 83% verified. We also notice that the lifecycle model is only slightly more precise than the trivial top model (27% versus 22% verified rate). Even with unsoundnesses discussed later, lifecycle^{++} is still worse than the lifestate model, with 57% of traces proven.

Lifestate is also expressive enough to prove most verifiable traces—making manual triage of the remaining alarms feasible. We manually examined the 14 remaining alarms with the lifestate model, and we identified two sources of imprecision: (1) an insufficient modeling of the attachment of UI components (e.g., is a View in the View tree attached to a particular Activity), resulting in 13 alarms; (2) a single detail on how Android options are set in the app's Extensible Markup Language (XML), resulting in 1 alarm. The former is not fundamental to lifestates but a modeling tradeoff where deeper attachment modeling offers diminishing returns on the verified rate while increasing the complexity of the model and verification times. The latter is an orthogonal detail for handling Android's XML processing (that allows the framework to invoke callbacks via reflection).

4.3.2 RQ2: Lifestate Generality.

Table 7 shows the precision results for the 1577 non-crashing traces that contained a sensitive callin from a total of 2202 traces that we collected from 121 distinct open source app repositories. Note that lifestate takes slightly longer than lifecycle; for this reason, lifestate performs slightly worse than lifecycle on execution. The bad column is 0 for models other than lifecycle++ because of continuous validation. The bottom line of Table 7 is that the lifestate model developed for RQ1 as-is generalizes to provide precise results (with a verified rate of 86%) when used to verify traces from 121 previously unseen apps. This result provides evidence that lifestates capture general behaviors of the Android framework. While the lifecycle++ model verifies 76% of traces, it also misses 27 out of 64 buggy traces (i.e., has a 42% false-negative rate).

Table 7: Generality of Lifestate Specification



properties	non-crashing traces		callback control-flow models								
			top		lifecycle		lifestate		lifecycle++		
			verified		verified		verified		verified		bad
sensitive callin	sensitive (n)	verifiable (n)	(n)	(%)	(n)	(%)	(n)	(%)	(n)	(%)	(n)
AlertDialog											
dismiss	94	59	54	92	54	92	54	92	58	98	3
show	145	144	125	87	124	86	125	87	127	88	0
AsyncTask											
execute	415	415	0	0	415	100	412	99	262	63	0
Fragment											
getResources	156	155	89	57	89	57	128	83	116	75	0
getString	220	193	124	64	124	64	134	69	131	68	24
setArguments	456	456	59	13	108	24	437	96	435	95	0
startActivity	91	91	0	0	0	0	12	13	19	21	0
total	1577	1513	451	30	914	60	1302	86	1148	76	27

Methodology. To get a larger corpus, we cloned 121 distinct open source app repositories from GitHub that use at least one sensitive callin (the count combines forks and clones). Then, we generated execution traces using the Android UI Exerciser Monkey [27] that interacts with the app issuing random UI events (e.g., clicks, touches). We attempted to automatically generate three traces for each app file obtained by building each app.

Discussion. From Table 7, we see that the lifestate verified rate of 86% in this larger experiment is comparable with the verified rate obtained in RQ1. Moreover, lifestate still improves the verified rate with respect to lifecycle, which goes from 60% to 86%, showing that the expressivity of lifestate is necessary.

Critically, the lifecycle++ model does not alarm on 42% of the traces representing real defects. That is, we saw unsoundnesses of the lifecycle++ model manifest in the protocol verification client. The verified rate for the lifecycle model is higher in this larger corpus (60%) compared to the rate in RQ1 (27%), and the precision improvement from the top abstraction is more substantial (60% to 30% versus 27% to 22%). This difference is perhaps to be expected when using automatically-generated traces that may have reduced coverage of app code and bias towards shallower, "less interesting" callbacks associated with application initialization instead of user interaction. In these traces, it is possible that UI elements were not exercised as frequently, which would result in more traces provable solely with the lifecycle specification. Since coverage is a known issue for the Android UI Exerciser Monkey [65], it was critical to have some evidence on deep, manually-exercised traces as in RQ1.

Bug Triage. We further manually triage every remaining alarm from both RQ1 and RQ2. Finding protocol usage bugs was not necessarily expected: for RQ1, we selected seemingly well-developed, well-tested apps to challenge verification, and for RQ2, we did not expect automatically generated traces to get very deep into the app (and thus deep in any protocol).

Yet from the RQ1 triage, we found 2 buggy apps out of 5 total. These apps were Puzzles [66] and SwiftNotes [67]. Puzzles had two bugs, one related to AlertDialog.show and one for AlertDialog.dismiss. Swiftnotes has a defect related to AlertDialog.show.

In the RQ2 corpus, we found 7 distinct repositories with a buggy app (out of 121 distinct repositories) from 64 buggy traces (out of 2202). We were able to reproduce bugs in 4 of the repositories and strongly suspect the other 3 to also be buggy. Three of the buggy apps invoke a method on Fragment that requires the Fragment to be attached. This buggy invocation happens within unsafe callbacks. Audiobug [68] invokes getResources. NextGisLogger [69] and Kistenstapeln [70] invoke getString. We are able to reproduce the Kistenstapeln bug.

Interestingly, one of the apps that contain a bug is Yamba [71], a tutorial app from a book on learning Android [72]. We note that the Yamba code appears as a part of three repositories where the code was copied (we only count these as one bug). The tutorial app calls AlertDialog.dismiss when an AsyncTask is finishing and hence potentially after the Activity used in the AlertDialog is not visible anymore. We found similar defects in several actively maintained open source apps where callbacks in an AsyncTask object were used either to invoke AlertDialog.show or AlertDialog.dismiss. These apps included OpenStreetMap (OSM) Tracker [73] and Noveldroid [74]. Additionally, we found this bug in a binary library connected with the PingPlusPlus android app [75]. By examining the output of our verifier, we were able to create a test to concretely witness defects in 4 of these apps.

4.3.3 RQ3: Model Validation.

Figure 32 displays the results of validating the lifecycle++ model on all traces, plotting the cumulative traces grouped by (intervals of) the number of steps validated, with the number of traces further divided into categories either indicating that validation succeeded, "No errors," or the cause of failure of the validation process. The plot in Figure 32 highlights the necessity of applying model validation: lifecycle++ based on a widely used callback control-flow model does not validate (i.e., an unsoundness is witnessed) on 58% of 2183 traces (and the validation ran out of memory for 19 out of the total 2202 traces).

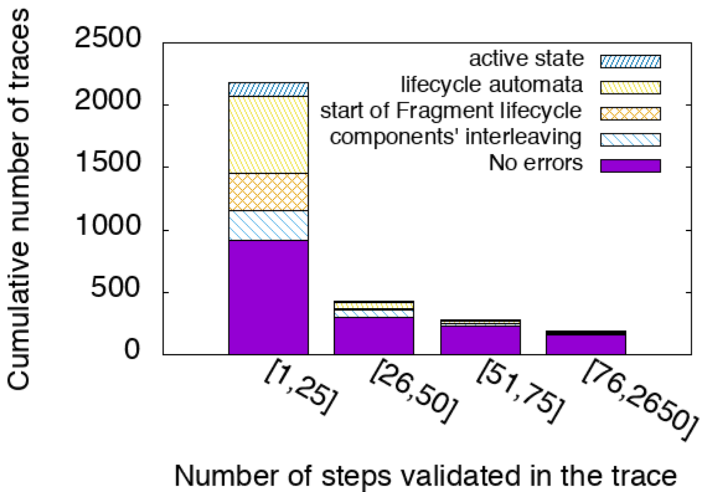


Figure 32: Unsoundnesses Detected in Common Callback Control-Flow Models

Methodology. We first evaluate the need for model validation by applying our approach to lifecycle++ and quantifying its discrepancies with the real Android executions.

Our first experiment validates the lifecycle++ model on all the traces we collected (bounding each validation check to 1 hour and 4 GB of memory). We quantify the necessity of model validation collecting for each trace if the model was valid and the length of the maximum prefix of the trace that the model validates. Since there are already some known limitations in the lifecycle++ model (e.g., components interleaving), we triage the results to understand if the real cause of failure is a new mistake discovered with the validation process.

Our second experiment qualitatively evaluates the necessity of model validation to develop sound lifestate specifications. To create a sound model, we started from the empty model (without rules) and continuously applied validation to find and correct mistakes. In each iteration, we model the callback control flow for a specific Android object, we validate the current model on the entire corpus of traces (limiting each trace to one hour and 4 GB of memory), and when the model is not valid for a trace, we inspect the validation result and repair the specification. We stop when the model is valid for all the traces. We then collected the mistakes we found with automatic validation while developing the lifestate model. We describe such mistakes and discuss how we used validation to discover and fix them.

Discussion: lifecycle++ Validation. From the first bar of the plot in Figure 32, we see that the lifecycle++ model validates only 42% of the total traces, while validation fails in the remaining cases (58%). The bar shows the number of traces that we validated for at least one step, grouping them by validation status and cause of validation failure. From our manual triage, we identified 4 different broad causes for unsoundness: (i) *outside the active lifecycle*: the model prohibits the execution of a callback outside the modeled active state of the Activity; (ii) *wrong lifecycle automata*: the model wrongly prohibits the execution of an Activity or Fragment lifecycle callback; (iii) *wrong start of the* Fragment *lifecycle*: the model prohibits the start of the execution of the Fragment lifecycle; (iv) *no components interleaving*: the model prohibits the interleaved execution of callbacks from different Activity or Fragment objects. The plot shows that the lifecycle++ model is not valid on 25% of the traces because it does not model the interleaving of components (e.g., the execution of callbacks from different Activity and Fragment objects cannot interleave) and the start of the Fragment lifecycle at an arbitrary point in the enclosing Activity object. With FlowDroid [38], such limitations are known and have been justified as practical choices to have feasible flow analyses. But the remaining traces, 33% of the total, cannot be validated due to other reasons including modeling mistakes. In particular, the FlowDroid model imprecisely captures the lifecycle automata (for both Activity and Fragment) and erroneously confines the execution of some callbacks in the active state of the lifecycle.

The other bars in the plot of Figure 32 show the number of traces we validated for more than 25, 50, and 75 steps, respectively. In the plot, we report the total number of steps in the execution traces that correspond to a callback or a callin that we either used in the lifestate or the lifecycle++ model, while we remove all the other messages. From such bars, we see that we usually detect the unsoundness of the lifecycle++ model "early" in the trace (i.e., in the first 25 steps). This result is not surprising since most of the modeling mistakes we found are related to the interaction with the lifecycle automata and can be witnessed in the first iteration of the lifecycle. We further discovered that the lifecycle++ model mostly validates shorter execution traces, showing that having sound models for real execution traces is more challenging (see [59] for further details).

Discussion: Catching Mistakes During Modeling. We were able to obtain a valid lifestate specification *for over 99.9%* of the traces in our corpus. That is, we were able to understand and model the objects we selected in all but two traces.

Surprisingly, we identified and fixed several mistakes in our modeling of the Activity and Fragment lifecycle that are due to undocumented Android behaviors. An example of such behavior is the effect of Activity.finish and Activity.startActivity on the callback control flow for the onClick callback. It is unsound to restrict the enabling of onClick callbacks to the active state of the Activity lifecycle (i.e., between the execution of the onResume and onPause callbacks). This is the behavior represented with blue edges in Figure 6, what is typically understood from the Android documentation, and captured in the existing callback control-flow models used for static analysis.

We implemented a model where onClick could be invoked only when its Activity was running and found this assumption to be invalid on several traces. We inferred that the mistake was due to the wrong "bounding" of the onClick callbacks in the Activity lifecycle since in all the traces: i) the first callback that was erroneously disabled in the model was the onClick callback; and ii) the onClick callback was disabled in the model just after the execution of an onPause callback that appeared before in the trace, without an onResume callback in between (and hence, outside the active state of the Activity.) It turns out that both finish and startActivity cause the Activity to pause without preventing the pending onClick invocations from happening, as represented in the red edges connected to onClick in Figure 6. We validated such behaviors by writing and executing a test application and finding its description in several StackOverflow posts [76, 77]. The fix for this issue is to detect the finishing state of the Activity and to not disable the onClick callback in this case.

5 CONCLUSIONS

The underlying vision of the Fixr project has consistently been to leverage social programming and the vibrant communities of app developers to create tools and techniques that ultimately improve software quality. That is, we undertook the challenge to mine *and* understand software repositories at scale *and* with relevance to help app developers interactively search, test, drive, witness, edit, and verify to confidently *transfer* bug fixes.

The Fixr effort required multiple complex and interleaved research thrusts as has been documented in this report. While this effort has certainly made significant steps along these thrusts, open challenges remain to fully realize the Fixr vision. In the remainder of this section, we comment on such future challenges and lessons learned along each thrust.

5.1 Rich Syntactic-Semantic Pattern Mining

BigGroum overcame the scalability issues due to the size of the app corpus by clustering the groums using frequent itemset mining and building a lattice that represents the embedding relationship among groums. Furthermore, we show that simple filtering techniques reduce the cost of the embedding computation. We presented a detailed experimental evaluation of BigGroum, demonstrating its scalability and the quality of the mined patterns.

Many open challenges remain that need to be addressed through further investigation. A first set of challenges come from the difficulty of running program analysis tasks on large corpora. For instance, many Android program analysis tools require compilation into an Android Package (APK). However, doing so changes the structure of the program in many ways making the process of recovering back some structure from the original code quite challenging. Powerful analyses that work directly on Android Java source code will enable our ideas to work faster and on a larger scale. Existing frameworks that can achieve this are limited in their nature.

Another challenge arises in distinguishing patterns. Although a null model based approach was investigated to rapidly estimate the probability that a pattern could have been generated by pure chance, our results indicate that a more sophisticated generative model of API usage patterns may be necessary in order to carefully disentangle useful patterns (the signal) from accidental ones (the noise). Finally, our approach did not consider existing documentation available online. However, parsing such documentations will require a mix of natural language processing and program analysis techniques.

5.2 Automated Testing of Interactive Apps

ChimpCheck addresses the problem of efficiently and effectively exercising interactive apps to generate relevant user-interaction traces. Our key observation is the need to fuse scripted and randomized UI testing—that is, we need automation but we cannot forsake human insight and app-specific knowledge. To realize this vision, the critical technical step was a lifting of UI scripting from traces to generators drawing on ideas from property-based test-case generation [35].

In particular, we formalized the notion of user-interaction event sequences (or UI traces) as a first-class object. First-class UI traces naturally lead to UI trace generators that abstract sets of UI traces. The sampling from UI trace generators is then platform-independent and can leverage a property-based testing framework such as ScalaCheck.

Driven by real issues reported in real Android apps, we demonstrated how ChimpCheck enables easily building customized testing patterns out of compositional components. The resulting testing patterns like interrupting sequencing, property preservation, brute-force randomized monkey testing, and a customizable monkey tester seem broadly applicable and useful. Preliminary experiments provide evidence that simple specializations expressed in ChimpCheck can drive apps to witness bugs with many fewer events.

Generalizing from lessons learned during development and experimentation of ChimpCheck, we have distilled two design principles, namely higher-order automated generators and custom reifications of test inputs. We believe that these design principles will serve as a guide to future development of highly relevant testing frameworks based on the basic idea of expressing property-based test generators via combinator libraries.

5.3 Event-Driven Protocols and Callback Control Flow

Verivita addresses the specification problem for event-driven application programming protocols. Our main contribution is a careful distillation of what is observable at the interface between the framework and the app. This distillation leads to the abstract notions of permitted messages from the framework to the app (e.g., enabled callbacks) and prohibited messages into the framework from the app (e.g., disallowed callins). Such formalization allows us to define a clean specification language Lifestate and a procedure to formally validate the framework specifications—a fundamental step to guarantee that the model is sound, despite the high complexity of real-world frameworks. We further provided a tool that automatically analyzes protocol violations. While our current analysis is dynamic, our formalization opens the door for static analysis tools using lifestate specifications to statically verify the app code.

We see several open challenges in the analysis of event-driven applications. One of the main open challenges is in obtaining both the callback control flow models and the application-programming protocols for a large and continuously evolving framework as Android. Our experiments demonstrate that precise and sound modeling is time consuming and error prone. We see a real need for more research in the automatic synthesis of framework models.

The other challenge consists of generalizing the framework models to represent Android inter-process communications and different event-driven frameworks, such as the Robotic Operating System (ROS), React.js for web programming, or TensorFlow for machine learning applications. In the inter-process communication model, multiple components, either from the framework or the app, may invoke each other and thus determine different app execution paths. ROS adopts a similar architecture, where single computation nodes "communicate" through different mechanisms. An open problem is to extend the lifestate specification formalism to express such app-framework interactions.

5.4 Big Code Versus Big Data

The main lessons learned from our work on the Fixr storage layer involved understanding over time the overwhelming need to make use of techniques and technologies that could perform all needed processing in an incremental and asynchronous fashion. Furthermore, while it was always understood that any storage mechanism selected for the project would need to be scalable, it became increasingly clear across the project that the storage layer needed to consist of multiple, scalable persistence technologies that were all selected to meet specific requirements imposed by the Fixr components that accessed them. Indeed, as many data-intensive software engineering projects learn, being able to meet the needs of a large project using one uniform persistence technology is not possible; extreme heterogeneity is the norm [78].

Furthermore, we started the project making use of batch processing technologies, such as Apache Spark, to process all Android repositories over multiple days and extracting all desired features from their source code at once. Such techniques were useful in bootstrapping the project to a point where there was lots of data available to support work by all of the other Fixr components but quickly became too difficult to maintain as each individual repository would receive new commits over time. The amount of work required to process one new commit using a batch processing approach was completely out of alignment with the benefits gained.

As such, the software infrastructure had to be designed to perform all steps of the feature extraction workflow asynchronously and incrementally. After an initial investment of time (days) and cycles to process an initial set of repositories from scratch, all subsequent work to handle new commits happened quickly (seconds) and only required updating information artifacts that were related to the code that had been updated by a new commit. The challenge, of course, in developing this infrastructure was changing the mindset of its developers to one in which all programming tasks had to be written such that they could occur asynchronously. The key difficulty with that approach to development is how much software engineers have grown to depend on thinking of the form "A happens before B" and learning how to step away from that and truly write software that does not care whether B happens before A or vice versa. A novel aspect of this work was adopting software frameworks (such as Elixir's agent framework) that allowed for asynchronous software processing to be specified using first class concepts.

5.5 Querying Big Code

The overarching goal of the design and development of Interactive Fixr is to enable software developers to make use of the large number of artifacts mined by Fixr, including Android repositories, extracted features, and derived information. We have demonstrated the usefulness of Interactive Fixr with some real-world use cases.

However, in the process of designing, developing, and testing Interactive Fixr, weencountered a number of challenges that suggest future work. Interactive Fixr does not support the full-range of artifacts produced by the Fixr tools. For example, the BigGroum tool is able to label syntactic-semantic API patterns as "popular" or "anomalous," but Interactive Fixr does not expose this information fully. A significant human-centered design challenge continues to be exposing the breadth of program analysis artifacts that contain an enormous wealth of information in a manner that is approachable for the app developer (i.e., the end user) and is useful in her workflow.

REFERENCES

[1] Shawn Meier, Sergio Mover, and Bor-Yuh Evan Chang. Lifestate: Event-driven protocols and callback control flow. In *Object-Oriented Programming (ECOOP)*, volume 134, pages 1:1–1:29, 2019.

[2] Yan Wang, Hailong Zhang, and Atanas Rountev. On the unsoundness of static analysis for Android GUIs. In *State of the Art in Program Analysis (SOAP)*, pages 18–23, 2016.

[3] Sergio Mover, Sriram Sankaranarayanan, Rhys Braginton Pettee Olsen, and Bor-Yuh Evan Chang. Mining framework usage graphs from app corpora. In *Software Analysis, Evolution and Reengineering (SANER)*, pages 277–289, 2018.

[4] CUPLV (University of Colorado Programming Languages and Verification). BigGroum. http://plv.colorado.edu/biggroum, 2018.

[5] CUPLV (University of Colorado Programming Languages and Verification). GitHub - cuplv/biggroum. https://github.com/cuplv/biggroum, 2020.

[6] Edmund S. L. Lam, Peilun Zhang, and Bor-Yuh Evan Chang. ChimpCheck: property-based randomized test generation for interactive apps. In *New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 58–77, 2017.

[7] CUPLV (University of Colorado Programming Languages and Verification). ChimpCheck. http://plv.colorado.edu/chimpcheck, 2018.

[8] CUPLV (University of Colorado Programming Languages and Verification). GitHub - cuplv/chimpcheck. https://github.com/cuplv/chimpcheck, 2018.

[9] Arjun Radhakrishna, Nicholas V. Lewchenko, Shawn Meier, Sergio Mover, Krishna Chaitanya Sripada, Damien Zufferey, Bor-Yuh Evan Chang, and Pavol Cerný. DroidStar: callback typestates for Android classes. In *International Conference on Software Engineering (ICSE)*, pages 1160–1170, 2018.

[10] CUPLV (University of Colorado Programming Languages and Verification). DroidStar. http://plv.colorado.edu/droidstar, 2018.

[11] CUPLV (University of Colorado Programming Languages and Verification). GitHub - cuplv/droidstar. https://github.com/cuplv/droidstar, 2018.

[12] CUPLV (University of Colorado Programming Languages and Verification). Verivita. http://plv.colorado.edu/verivita, 2018.

[13] CUPLV (University of Colorado Programming Languages and Verification). GitHub - cuplv/verivita. https://github.com/cuplv/verivita, 2018.

[14] Shawn Meier, Sergio Mover, and Bor-Yuh Evan Chang. Lifestate: Event-driven protocols and callback control flow (artifact). *DARTS*, 5(2):13:1–13:3, 2019.

[15] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated API property inference techniques. *IEEE Trans. Software Eng.*, 39(5):613–637, 2013.

[16] Amir Michail. Data mining library reuse patterns in user-selected applications. In *Automated Software Engineering (ASE)*, page 24, 1999.

[17] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Trans. Software Eng.*, 31(6):429–445, 2005.

[18] Annie T. T. Ying, Gail C. Murphy, Raymond T. Ng, and Mark Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Software Eng.*, 30(9):574–586, 2004.

[19] João Eduardo Montandon, Hudson Borges, Daniel Felix, and Marco Tulio Valente. Documenting APIs with examples: Lessons learned with the APIMiner platform. In *Reverse Engineering (WCRE)*, pages 401–408, 2013.

[20] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal API rules from imperfect traces. In *International Conference on Software Engineering (ICSE)*, pages 282–291, 2006.

[21] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE)*, pages 35–44, 2007.

[22] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: mining and recommending API usage patterns. In *Object-Oriented Programming (ECOOP)*, pages 318–343, 2009.

[23] Sriram Sankaranarayanan, Swarat Chaudhuri, Franjo Ivancic, and Aarti Gupta. Dynamic inference of likely data preconditions over predicates by tree learning. In *Software Testing and Analysis (ISSTA)*, pages 295–306, 2008.

[24] Rajeev Alur, Pavol Cerný, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for Java classes. In *Principles of Programming Languages (POPL)*, pages 98–109, 2005.

[25] Hila Peleg, Sharon Shoham, Eran Yahav, and Hongseok Yang. Symbolic automata for static specification mining. In *Static Analysis (SAS)*, pages 63–83, 2013.

[26] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. In *European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE)*, pages 383–392, 2009.

[27] Android Studio. UI/Application Exerciser Monkey. https://developer.android.com/guide/components/activities.html, 2010.

[28] Android Developers. Testing UI for a single app. https://developer.android.com/training/testing/ui-testing/espresso-testing.html, 2016.

[29] Renas Reda. Robotium: User scenario testing for Android. http://www.robotium.org, 2009.

[30] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software*, 32(5):53–59, 2015.

[31] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine,

and Atif M. Memon. Using GUI ripping for automated testing of Android applications. In *Automated Software Engineering (ASE)*, pages 258–261, 2012.

[32] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: an input generation system for Android apps. In *European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE)*, pages 224–234, 2013.

[33] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. Evodroid: segmented evolutionary testing of Android apps. In *Foundations of Software Engineering (FSE)*, pages 599–609, 2014.

[34] Ke Mao, Mark Harman, and Yue Jia. Sapienz: multi-objective automated testing for Android applications. In *Software Testing and Analysis (ISSTA)*, pages 94–105, 2016.

[35] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming (ICFP)*, pages 268–279, 2000.

[36] Android Developers. The Activity lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle.html, 2008.

[37] Steve Pomeroy. The complete Android Activity/Fragment lifecycle v0.9.0. https://github.com/xxv/android-lifecycle, 2014.

[38] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Programming Language Design and Implementation (PLDI)*, page 29, 2014.

[39] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race detection for Android applications. In *Programming Language Design and Implementation (PLDI)*, pages 316–325, 2014.

[40] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Selective control-flow abstraction via jumping. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 163–182, 2015.

[41] ampm. Douban-FM-sdk. https://github.com/ampm/Douban-FM-sdk, 2017.

[42] StackOverflow. StackOverflow - Android error-close() was never explicitly called on database. http://stackoverflow.com/questions/4464892/android-error-close-was-never-explicitly-called-on-database, 2012.

[43] StackOverflow. StackOverflow - what is "SQLiteDatabase created and never closed" error? https://stackoverflow.com/questions/7632047/what-is-sqlitedatabase-created-and-never-closed-error, 2011.

[44] GitHub Inc. The state of the octoverse 2018. https://blog.github.com/ 2018-10-16-state-of-the-octoverse/, 2018.

[45] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Very Large Data Bases (VLDB)*, pages 487–499, 1994.

[46] Sergio Mover. GitHub pull request - implement the tag map activity.

https://github. com/cuplv/mapbox-android-demo/pull/36, 2020.

[47] GitHub mapbox-android-demo. mapbox-android-demo. https://github.com/mapbox/mapbox-android-demo, 2020.

[48] Rickard Nilsson. ScalaCheck: Property-based testing for Scala. http://scalacheck.org/, 2015.

[49] Martin Fietz. FeedRemover: already running - issue #1304 - AntennaPod/AntennaPod. https://github.com/AntennaPod/AntennaPod/issues/1304, 2015.

[50] Vladislav Kaplun. Update RequestAsyncTask.java by kaplad - pull request #315 - facebook/facebook-android-sdk. https://github.com/facebook/facebook-android-sdk/pull/315, 2014.

[51] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

[52] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. In *Computer-Aided Verification (CAV)*, pages 334–342, 2014.

[53] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Computer Science and Software Engineering (CASCON)*, pages 13–, 1999.

[54] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963, pages 337–340, 2008.

[55] StackOverflow on getWritableDatabase. http://stackoverflow.com/questions/35068292/do-i-need-call-close-when-i-use-getwritabledatabase, 2017.

[56] Google Inc. Android documentation on setDisplayAsUpEnabled. https://developer.android.com/training/implementing-navigation/ancestral.html#NavigateUp, 2017.

[57] DroidPlanner. Tower. https://github.com/DroidPlanner/Tower, 2017.

[58] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. Systematic execution of Android test suites in adverse conditions. In *Software Testing and Analysis (ISSTA)*, pages 83–93, 2015.

[59] Shawn Meier, Sergio Mover, and Bor-Yuh Evan Chang. Lifestate: Event-driven protocols and callback control flow (extended version). *CoRR*, abs/1906.04924, 2019.

[60] Red Reader. Crash during commenting #467 RedReader. https://github.com/QuantumBadger/RedReader/issues/467, 2017.

[61] Android Topeka. Crash if rotate device right after press floating action button #4 Topeka for Android. https://github.com/googlesamples/android-topeka/issues/4, 2015.

[62] OneBusAway. IllegalStateException: Fragment BaseMapFragment not attached to Activity #570 OneBusAway. https://github.com/OneBusAway/onebusaway-android/issues/ 570, 2016.

[63] StackOverflow Post. Alertdialog creating exception in android. https://stackoverflow.com/questions/15104677/alertdialog-creating-exception-in-android, 2013.

[64] StackOverflow Post. Got exception: fragment already active. https://stackoverflow.com/questions/10364478/got-exception-fragment-already-active, 2012.

[65] Yauhen Leanidavich Arnatovich, Minh Ngoc Ngo, Hee Beng Kuan Tan, and Charlie Soh. Achieving high code coverage in Android UI testing via automated widget exercising. In *Asia-Pacific Software Engineering Conference, (APSEC)*, pages 193–200, 2016.

[66] Chris Boyle. Simon Tatham's puzzles. https://github.com/chrisboyle/sgtpuzzles/blob/658f00f19172bdbceb5329bc77376b40fe550fcb/app/src/main/java/name/boyle/chris/sgtpuzzles/GamePlay.java#L183, 2014.

[67] Adrian Chifor. Swiftnotes. https://fdroid.org/en/packages/com.moonpi.swiftnotes/, 2015.

[68] Matthias Urhahn. AudioBug. https://github.com/d4rken/audiobug, 2017.

[69] NextGis. NextGisLogger. https://github.com/nextgis/nextgislogger, 2017.

[70] D120. Kistenstapeln. https://github.com/d120/Kistenstapeln-Android, 2015.

[71] Marko Gargenta. Yamba. https://github.com/learning-android/Yamba/blob/429e37365f35ac4e5419884ef88b6fa378c023f8/src/com/marakana/android/yamba/StatusFragment.java, 2014.

[72] Marko Gargenta and Masumi Nakamura. *Learning Android*. O'Reilly Media, 2014.

[73] Nicolas Guillaumin. OSMTracker for Android. https://github.com/nguillaumin/osmtracker-android/blob/d80dea16e456defe5ab62ed8b5bc35ede363415e/app/src/main/java/me/guillaumin/android/osmtracker/gpx/ExportTrackTask.java, 2015.

[74] sh1ro. NovelDroid. https://github.com/sh1r0/NovelDroid/blob/f3245055d7a8bcc69a9bca278fbe890081dac58a/app/src/main/java/com/sh1r0/noveldroid/SettingsFragment.java, 2016.

[75] PingPlusPlus. Ping Plus Plus. https://github.com/PingPlusPlus/pingpp-android, 2017.

[76] StackOverflow Post. Android: click event after Activity.onPause(). https://stackoverflow.com/questions/38368391/android-click-event-after-activity-onpause, 2016.

[77] StackOverflow Post. OnClickListener fired after onPause? https://stackoverflow.com/questions/31432014/onclicklistener-fired-after-onpause, 2015.

[78] Kenneth M. Anderson. Embrace the challenges: Software engineering in a big data world. In *Big Data Software Engineering (BIGDSE)*, pages 19–25, 2015.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

API	Application Programming Interface
APK	Android Package
app	application
BES	Best Practices
CDFG	Control-Data Flow Graph
COLO	Quality of BigGroum, are patterns due to random co-location?
COMP	Quality of BigGroum, find better POPULAR patterns than GrouMiner?
DARPA	Defense Advanced Research Projects Agency
Def	definition
f	Average Frequency
groum	graph-based object usage model
HDFS	Hadoop Distributed File System
HTML	HyperText Markup Language
ID	identifier
id	identifier
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
logroum	lattice-ordered groum
MUSE	Mining and Understanding Software Enclaves
OBL	Obligatory
OSM	OpenStreetMap
PERF1	Performance of BigGroum, scale compared to GrouMiner
PERF2	Performance of BigGroum, SAT-based embedding and scalability
POC	Proof of Concept
PREC1	Quality of BigGroum, are POLULAR patterns mined correct?
PREC2	Quality of BigGroum, are ANOMALOUS and ISOLATED patterns real bugs?
protobuf	Protocol Buffers
RCP	Rich Client Platform
REC1	Quality of BigGroum, are known patterns mined?
REC2	Quality of BigGroum, are ANOMALOUS and ISOLATED patterns actual bugs in known patterns
ROS	Robotic Operating System
RQ	research question
SAT	Boolean satisfiability
SMT	Satisfiability Modulo Theories
SWT	Standard Widget Toolkit
T	Total Number of Patterns Matching the Reference Pattern
TAPFI	Talk and Per-File
UI	User Interface
URI	Uniform Resource Identifier

URL	Uniform Resource Locator
XML	Extensible Markup Language

GLOSSARY OF TERMINOLOGY

Callback	A callback is an application-implemented method that is invoked by a software frame- work as an extension point (e.g., for application-specific processing of a user event).
Callin	A callin, by analogy to callback, is a framework-implemented method that is invoked by an application to affect the state of the software framework.
Framework	A software framework is an abstraction that provides generic functionality for client applications through callback extensions.