Automated Testing with Targeted Event Sequence Generation

Casper S. Jensen[*,†]
Aarhus University, Denmark
semadk@cs.au.dk

Mukul R. Prasad
Fujitsu Laboratories of America, USA
mukul@us.fujitsu.com

Anders Møller[†]
Aarhus University, Denmark
amoeller@cs.au.dk

ABSTRACT

Automated software testing aims to detect errors by producing test inputs that cover as much of the application source code as possible. Applications for mobile devices are typically event-driven, which raises the challenge of automatically producing event sequences that result in high coverage. Some existing approaches use random or model-based testing that largely treats the application as a black box. Other approaches use symbolic execution, either starting from the entry points of the applications or on specific event sequences. A common limitation of the existing approaches is that they often fail to reach the parts of the application code that require more complex event sequences.

We propose a two-phase technique for automatically finding event sequences that reach a given target line in the application code. The first phase performs concolic execution to build summaries of the individual event handlers of the application. The second phase builds event sequences backward from the target, using the summaries together with a UI model of the application. Our experiments on a collection of open source Android applications show that this technique can successfully produce event sequences that reach challenging targets.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Languages, Verification

Keywords

Symbolic execution; test generation; mobile applications; Android

[*]This author was an intern at Fujitsu Laboratories of America for a part of this work.

[†]Supported by Google, IBM, and the Danish Research Council for Technology and Production.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the author/owner(s). Publication rights licensed to ACM.

ISSTA'13, July 15–20, 2013, Lugano, Switzerland
ACM 978-1-4503-2159-4/13/07
http://dx.doi.org/10.1145/2483760.2483777

1. INTRODUCTION

Mobile applications are often structured as collections of screens where user interactions and other events trigger transitions from one screen to another and cause updates of the internal application state. To test such applications, the developers face the challenge of constructing test inputs that exercise the functionality and cover all reachable application code. A test input consists of a sequence of events, each with some parameters depending on the kind of event, for example, coordinates for click events and string values when text fields are filled in. In contrast to other kinds of software, a key challenge in mobile application testing is managing the explosion in the number of possible event sequences [3]. Since it can be difficult and tedious to construct these test inputs we wish to automate the work. We focus on Android, which has become the most widely used platform for mobile software. More specifically, our goal is to improve automated testing for Android applications that are not computationally heavy but may have complex user interaction patterns.

One popular technique is black-box random testing or crawling [2, 13, 15]. Other related approaches are based entirely on abstract models of the applications [21, 24] or only involve the application code for extracting, for example, available event handlers and basic information on how they access shared state [4, 18]. Common to these approaches is that they cannot effectively reach branches of the application code that are highly constrained by the event parameters. In contrast, symbolic execution, which analyzes the application code in more detail, has shown to be a powerful approach to find appropriate event parameters. However, most existing techniques that do apply symbolic execution are not able to effectively construct event sequences that consist of many events. For example, the experiments reported by Anand et al. [3] are limited to event sequences of length 4. Others, for example, Mirzaei et al. [20] use symbolic execution, but only for deriving the event parameters, not for the sequencing of events. As a consequence, these existing approaches are not able to effectively reach parts of the application code that require many events and with highly constrained event parameters.

We propose a targeted approach to generation of event sequences. Given a target location in the application code, for example, a branch that is not reachable with the existing automated testing techniques, we wish to find an event sequence that leads from the application entry to the target. Note that it may be easy to reach the entry of an event handler that contains the target but significantly more difficult

to reach the target itself, since it may be guarded by conditionals that depend on events earlier in the event sequence.

Our approach is inspired by the work of Ma et al. who consider the line reachability problem for C programs [17]. They present an algorithm that works backward in the call graph from a given target, using traditional forward symbolic execution of each function, until it finds a feasible path from the start of the program. Call graphs of C programs resemble finite-state UI models of event-driven applications, however with the important difference that calls in a C program are controlled by the program itself whereas navigation in event-driven applications is largely controlled by the user. This means that a simple backward search toward the application entry would likely lead to an explosion of different paths to consider. To address this problem, we draw inspiration from another source: the model-based testing technique by Arlt et al. [4]. In model-based testing of event-driven applications, test inputs are constructed from a finite-state abstraction of the user interface behavior that shows how the event handlers are connected. In the technique by Arlt et al., the conventional UI model is augmented by event dependence information that for each pair of event handlers gives an indication of how much state may be written by one of them and read by the other. This information provides the basis for construction of abstract event sequences, which are subsequently extended to executable event sequences using the UI model. In our approach, when we search backward through the event handlers from the target, we use this idea of exploiting UI models and event dependence information to narrow the search space – although with some fundamental differences that we explain in Section 8. For the kind of UI models we use, each state represents a combination of registered event handlers, and transitions correspond to execution of event handlers. Previous work has shown that it is often possible to infer such UI models automatically [27].

Combining these ideas, our approach to targeted event sequence generation works in two phases:

1) We first preprocess the application by performing concolic execution [11] of each event handler to infer path conditions and symbolic states for its paths. The result is a summary for each event handler, reminiscent of the use of function summaries in compositional symbolic execution [10]. This phase is independent of the selected target.

2) Given a target location in the application code, the main phase uses the event handler summaries together with a UI model of the application to build a concrete event sequence that leads from the entry state of the application to the target. This is structured as a worklist algorithm where each worklist item consists of a path through one or more event handlers ending at the target. Each path is extended incrementally by searching for an event handler that may be triggered in front of the path to satisfy some of the constraints in the path condition, following the idea from Ma et al. [17]. This search uses the UI model and the symbolic states to bypass event handlers that are likely not relevant for the path condition. When candidate event sequences are found, we compose the path summaries and check satisfiability. This process continues until the entry state of the UI model is reached.

Our contributions can be summarized as follows:

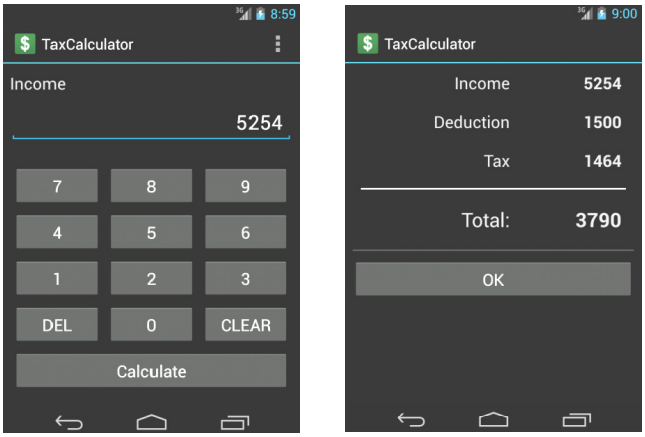
- We present a framework for automated testing of event-driven applications that combines concolic execution and UI models for targeted generation of test inputs to reach application code that may require many events and with highly constrained event parameters. An important part of this approach is how concolic execution is applied to individual event handlers and the resulting summaries are composed for reasoning about event sequences. Another central idea is to extract information about data dependence between event handlers from the concolic execution and exploit this to narrow the search space.
- The framework can be tuned by the prioritization mechanism of the worklist. We suggest three example prioritization heuristics that consider different aspects of how execution of event handlers affect application state.
- We provide an experimental evaluation involving five Android applications. Our prototype implementation uses a novel approach to concolic execution that utilizes the debugging interface of the Android emulator. The experimental results show that the approach can successfully cover challenging targets that are beyond the reach of random testing and conventional model-based test sequence generation.

Targeted generation of application inputs can be useful not only for maximizing coverage in automated testing but also for reproduction of reported errors and evolution of test suites.

Although our work is motivated by practical challenges in Android application development and our experimental tool is built for this specific platform, we believe that our approach may also be applicable to other kinds of event-driven programs, such as, JavaScript web applications and desktop GUI applications. However, our approach is particularly suitable for mobile applications, where event sequences are often longer and event handlers are smaller than in web or GUI applications.

2. MOTIVATING EXAMPLE

In this section we introduce a simple Android application, *TaxCalculator*, that we use as a motivating example to illustrate different aspects of our approach. *TaxCalculator* is a personal tax calculator used to compute the income tax



(a) INCOME (b) RESULT

Figure 1: Two screens in *TaxCalculator*: (a) the income entry screen and (b) the result screen displaying the income, deductions, and resulting tax.

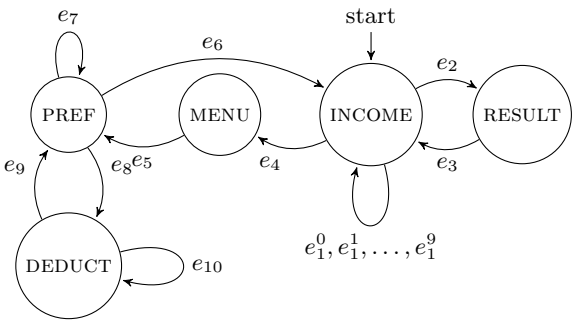


Figure 2: UI model of a part of *TaxCalculator*.

liability for a given income amount. Figure 1 shows screenshots illustrating its simplest use case.

On the entry screen, denoted INCOME, the user enters an income amount through a numeric keypad. Clicking the Calculate button takes the user to a result screen, denoted RESULT, which displays the calculated tax amount. By default this is a fixed percentage of the income amount. Figure 2 shows a part of the UI model that captures the relevant event sequences in this application. Intuitively, the states denote principal GUI screens and the transitions denote user actions, such as, clicks on buttons or changes to text fields. For example, $e_1^0, e_1^1, \ldots, e_1^9$ denote a click on the 0...9 buttons in the numeric keypad, e_2 is a click on Calculate, and e_3, e_6, and e_9 are clicks on the device's *back* button (in the lower left corner on the screen).

The default tax calculation can be modified by optionally specifying an income tax deduction amount that is deducted from the income before calculating the tax. To do this, the user must press the device's *menu* button (in the top right corner of the screen), corresponding to e_4 in the UI model, to get to the MENU screen, and from there click a Settings widget, e_5, to access the preferences screen, PREF. That screen contains a radio button for toggling tax deduction calculation. The user needs to click this button, e_7, and then click another button, e_8, which opens a dialog box denoted by DEDUCT. Here, the user can specify the deduction amount via a text field, e_{10}. The value being entered here is available as an event parameter, which is abstracted away in the UI model. The *back* button can then be used to navigate back to PREF and further to INCOME, corresponding to e_9 and e_6, respectively. The user can now enter the income amount and click Calculate to perform the modified tax calculation.

Figure 3 shows a fragment of code from *TaxCalculator*, used for performing the tax calculation. It is executed each time the user clicks Calculate on the INCOME screen. This code, though simple, is not trivial to test. Specifically, the calculation contains an if-statement with the predicate `taxable < 0`. In order to reach line 8 the application must be configured to take tax deduction into account, and the tax deduction amount must be greater than the income entered. The shortest sequence of user actions that can fulfill this constraint contains 8 events: $e_4, e_5, e_7, e_8, e_{10}, e_9, e_6, e_2$. Moreover, the branch on line 7 depends on the text value entered at e_{10}. In other words, reaching the target requires not only a long event sequence but also specific values in event parameters earlier in the sequence. Such a combination of requirements on event sequences and event parameters makes it difficult for existing automated testing techniques to reach the target line.

```
 1 income = this.appState.enteredAmount;
 2 deduction = 0;
 3 if (Settings.getEnableTaxDeduction()) {
 4   deduction = Settings.getTaxDeduction();
 5 }
 6 taxable = income - deduction;
 7 if (taxable < 0) {
 8   taxable = 0; // example target
 9 }
10 tax = taxable * TAX_RATE;
11 result = income - tax;
```

Figure 3: Snippet from the `onCreate` method in the `TaxResult` activity in *TaxCalculator*.

In our approach, we start at the given target at line 8. Concolic execution infers a path constraint that involves three variables in the application state: the income value, the deduction value, and the flag that controls whether tax deduction is enabled. It also infers event handler summaries that show which events may influence these variables. This information is then used when constructing event sequences.

3. UI MODELS FOR EVENT-DRIVEN APPLICATIONS

The literature on automated testing contains many different views on what constitutes an event-driven application. This section establishes the essential terminology that we use in the description of our proposed approach.

We view an event-driven application, in particular, an Android application, as a collection of *event handler* methods. During execution, event handlers can be attached to GUI widgets. An *event handler registration* is a triple of a GUI widget object, an event kind (click, text input, etc.), and an event handler method that has been attached to the widget. At any point during execution of the application we thus have a set of such event handler registrations. For simplicity, we assume that a single main method acts as entry point to the application for setting up the initial event handler registrations. The application is then driven by a sequence of events, each triggering the execution of an event handler from the current set of event handler registrations. Events with no corresponding event handler registration are ignored. We focus on user events, which represent a human user's interaction with the application, but our approach is equally applicable to system events that arise, for example, when new activities are created or paused. Some events are parameterized, for example, to indicate coordinates for click events or string values for text field alterations.

Our approach falls under the category of model-based testing. It operates on a *UI model* of the behavior of the graphical user interface of the Android application under test. Figure 2 shows a graphical view of the UI model for our motivating example. Formally, a UI model $\mathcal{M}$ is a finite-state machine denoted by a 4-tuple $\mathcal{M} = (S, s_0, E, T)$. Here, S is a finite set of abstract states representing different GUI screens, where $s_0 \in S$ is the initial state that describes the opening screen after the main method has been executed. E is a finite set of event handler registrations, as defined above, and $T \subseteq S \times E \times S$ is a transition relation, corresponding to the edges in the graphical view. Each abstract state $s \in S$ is uniquely characterized by its set of event handler registrations defined by $R_s = \{e_i \in E \mid (s, e_i, _) \in T\}$. We sometimes refer to event handlers and event handler reg-

istrations simply as events when the meaning is clear from the context.

A sequence of events $\langle e_1, \ldots, e_n \rangle$ is *consistent* with a sequence of states $p_0, \ldots, p_n$ where each $p_i \in S$ if for each $i = 1, \ldots, n$, either $(p_{i-1}, e_i, p_i) \in T$ or $e_i \notin R_{p_{i-1}}$. The latter case accounts for ignored events. In this way, every given sequence of events gives rise to a non-empty set of state sequences through $\mathcal{M}$.

A UI model is sound if it represents an over-approximation of the possible behavior of the application. More precisely, for any sequence of events $e = \langle e_1, \ldots, e_n \rangle$, let R_e denote the set of event handler registrations that exist after executing e on the concrete application starting from its entry state. For $\mathcal{M}$ to be sound we now require that there exists a state sequence $p = p_0, \ldots, p_n$ that is consistent with e and where $p_0 = s_0$ and $R_{p_n} = R_e$. Using an unsound UI model may prevent exploration of valid event sequences. Conversely, over-approximation could suggest infeasible event sequences, however, such sequences will be rejected by our algorithm, which tests candidates using concrete execution.

4. APPROACH OVERVIEW

Given an Android application under test, a UI model of the application, and a set of targets, the objective of our technique is to generate a test case for each target, that is, an event sequence that brings the application from its initial state to the target. Such targets, which can be lines or branches in the application code, arise in a number of different scenarios, as discussed in Section 1. The UI model could be specified manually or generated automatically through one of the model generation techniques proposed in the literature (see Section 8).

The motivating example in Section 2 demonstrates that reaching a target generally requires execution of a series of event handlers that mutate the program state, sometimes based on strings or numbers provided by the user in the form of event parameters, and navigation between these event handlers, ultimately executing the event handler that contains the target. More generally, our study of Android applications suggests that executions exercising specific targets often have a particular structure:

- There exists a small set of events, which we call *anchor events*, that are responsible for setting the necessary program state for a target to be executed.
- There is a disjoint set of events used only for connecting the initial state, the anchor events, and the target. These *connector events* do not affect the program state used at any anchor event or at the target.

For example, in the test case $\langle e_4, e_5, e_7, e_8, e_{10}, e_9, e_6, e_2 \rangle$ for the target at line 8 in Figure 3, e_7 and e_{10} are the anchor events, e_4, e_5, e_8, e_9, and e_6 are connector events, and e_2 exercises the target.

These observations motivate the key idea of our target event sequence generation algorithm. We identify a series of anchor events in reverse chronological order, starting at the target. The anchor events guide the search for a feasible test case by focusing on identifying events and paths in the application that are indispensable for reaching the target. In effect, this prunes away many sequences that can never reach the target. Further, we need to find suitable connector events to connect the initial state with the sequence of anchor events to the target. Thus, our approach works backward from the target, iteratively identifying anchor events and connector events, extending a partial sequence, until the initial state is reached.

We use symbolic analysis of the application source code to identify anchor events, build feasible paths exercising the target, and compute appropriate values for user event parameters. The UI model is used as the basis for selecting suitable connector events to connect the initial state, the anchor events, and the targets. To build a test case exercising a target, our analysis reasons at the level of individual execution paths within the event handlers. We refer to an execution path in an event handler that is triggered by an anchor event as an *anchor path* (or simply, an *anchor*). Similarly, an execution path for a connector event handler is called a *connector path* (or simply, a *connector*).

We note that the same event handler may be considered many times in the construction of a test case. To exploit this, we compute a symbolic summary of each event handler, once, in a target agnostic manner. This is a key ingredient of our approach. The construction of test cases now uses these summaries, without considering the actual application code.

Our overall approach is thus divided into two phases: a target agnostic *symbolic summarization phase*, followed by a *sequence generation phase* that searches for a test case for each target:

Symbolic Summarization This phase operates on the executable Android application. Symbolic analysis is applied to each event handler in turn to produce an *event handler summary* characterizing its behavior. This summary ideally includes necessary data and control-flow information about every execution path in the event handler code.

Sequence Generation This phase uses the event handler summaries generated in the first phase, along with the UI model to find a test case for a given target. In this search, the UI model and event handler summaries are used both to limit the search space, and as guides for the search space exploration order. The event sequence generation algorithm starts from the target and builds a sequence of events backward until it reaches the initial state, combining individual paths from the event handler summaries compositionally. In order to avoid false positives, candidate event sequences are executed concretely, using the executable application.

The algorithms for these two phases are presented in the following sections.

5. SYMBOLIC SUMMARIZATION

The symbolic summarization phase preprocesses the application code to produce a symbolic characterization, called an *event handler summary*, for each event handler. The event handlers can be located either using the UI model or by a simple static analysis of the Dalvik bytecode. An event handler summary is a set of *path summaries*, one for each execution path within the event handler code. Execution paths and their summaries encompass not only the event handler method itself but also other methods that may be called directly or indirectly from that method. A path summary $\mathcal{W}$ for a path P is a symbolic representation of the behavior of P, as in classical symbolic execution [16].

```
1: function SEQUENCESEARCH(target, summaries, model)
2:     worklist = INITIALIZE(target, summaries, model)
3:     while worklist is not empty do
4:         partialSequence = DEQUEUE(worklist)
5:         extendedPartialSequences = empty list of sequences
6:         for anchor in ANCHORS(partialSequence, summaries, model) do
7:             for path in PATHS(anchor, partialSequence, summaries, model) do
8:                 newPartialSequence = COMBINE(anchor, path, partialSequence)
9:                 if ISCOMPLETE(newPartialSequence) then
10:                    potentialTestCase = EXTRACTTESTCASE(newPartialSequence)
11:                    if REACHESTARGET(potentialTestCase, target) then
12:                        return potentialTestCase
13:                    end if
14:                end if
15:                APPEND(extendedPartialSequences, newPartialSequence)
16:            end for
17:        end for
18:        ENQUEUE(worklist, extendedPartialSequences)
19:        PRIORITIZE(worklist, extendedPartialSequences)
20:    end while
21:    return no test case found
22: end function
```

Figure 4: The event sequence generation algorithm. The input *target* denotes the target of interest, *summaries* is the set of all handler summaries produced in the symbolic summarization phase, and *model* is the UI model of the application. The algorithm either returns a test case that reaches the target, returns that it is unable to find a test case, or it diverges.

More formally, a path summary is denoted by a triple $\mathcal{W} = (pc, \sigma, \tau)$, where pc is the symbolic *path condition* of that path, σ is the symbolic state at the end of the execution of P, and τ is a log of bytecodes executed in the path, which serves as a unique signature of the path itself. The symbolic state, σ, is a map from variables in the application state to symbolic expressions, such that $\sigma(v)$ represents the value of v at the end of the execution of P. The values of event parameters and object fields are treated symbolically.

Event handler summaries can be computed by performing concolic execution [11]. Each iteration of concolic execution symbolically explores one path and hence computes its path summary. In this way, both the state that is shared between event handlers and the event parameters are treated symbolically at the entry of the event handler, so the event handler summary characterizes the event handler in its most general environment, independent of the preceding event sequence and event parameters.

In practice, concolic execution may not be able to cover all possible execution paths within a given event handler. This means that our event handler summaries may be incomplete, which can potentially affect the efficacy of our overall approach. However, this possibility is mitigated by the fact that event handlers in mobile applications are often relatively small, with much of the complexity of the application code lying in the dependencies between the event handlers.

Note that reachability of a given target in an event handler e_i cannot be decided based on the summary of e_i alone. In case a path from the entry point in e_i to the target has a nontrivial path condition pc, we need to produce an event sequence that brings the application from its entry state to a state where e_i can be triggered, i.e. it exists as an event handler registration, and moreover, pc is satisfied. We address this challenge in the following section.

6. SEQUENCE GENERATION

The event sequence generation phase generates a test case for each given target, based on the event handler summaries generated in the symbolic summarization phase and the UI model. For this phase, we propose an algorithm that gradually explores sequences of events backward, from the target to the application entry point. The algorithm, given in Figure 4, is organized around a prioritized worklist of partial sequences that are gradually extended until a complete sequence is found. The prioritization mechanism guides the selection of worklist items to be explored next.

A *partial sequence* is a sequence of path summaries representing a concrete path $\langle \tau_1, \tau_2, \ldots, \tau_n \rangle$ through the application, combined with an abstract state s in the UI model. Each τ_i is a complete path in an event handler for an event e_i of the UI model, where the segment τ_n exercises the target. The event sequence $\langle e_1, e_2, \ldots, e_n \rangle$ is consistent with a state sequence starting from s in the UI model.

The INITIALIZE function (line 2) initializes the worklist as follows. For each path summary $\mathcal{W}$ that exercises the target of interest (that is, the bytecode log of the path summary contains the target) and each abstract state s in the UI model where s has an outgoing transition labelled e_i such that $\mathcal{W}$ belongs to e_i, we add the partial sequence of length 1 defined by $\mathcal{W}$ and s to the worklist. For the example in Section 2, the target is the event handler for e_2, which appears as an outgoing edge from INCOME in Figure 2. Only a single path summary exercises the target in this example, so the worklist will be initialized to a single partial sequence defined by that path summary and the INCOME abstract state.

Next, the main search loop is entered (lines 3–20). A partial sequence is selected from the worklist and extended into a number of new partial sequences. This extension is conducted in two steps:

1. A set of anchors for the partial sequence is found using the ANCHORS function (line 6) described in Section 6.1. This function provides a set of event handler paths, each of which (1) write to some program state that the partial sequence depends on according to its path condition, and (2) has a symbolic state that is consistent with the path condition of the partial sequence.
2. For each anchor, we extract a set of feasible sequences of connectors that lead from the anchor to the partial sequence (line 7), using the PATHS function described in Section 6.2. For each sequence, we construct a new partial sequence consisting of the anchor, the sequence of connectors, and the original partial sequence.

We say that a partial sequence is *complete* if it starts at the entry state of the application and reaches the target. Such a sequence may give rise to a concrete test case for the target. The ISCOMPLETE function checks if an extended partial sequence is complete (line 9). In that case, we extract a potential test case using the EXTRACTTESTCASE function (line 10) and check that it reaches the target when executed concretely by the REACHESTARGET function. If the new partial sequence is not complete, it is added to a list of extended partial sequences (line 15). On lines 18–19, these partial sequences are added to the worklist, and their priorities are computed using the PRIORITIZE function that we describe in Section 6.3.

6.1 Construction of Anchors

The ANCHORS function produces a set of anchors for a partial sequence. Recall from Section 4 that an anchor is an execution path in an event handler that writes to some program state that the partial sequence depends on, according to its path condition. We define the *dependency set* of a partial sequence as the set of variables that occur in its path condition. In this way, an anchor corresponds to an execution path in an event handler that affects the values in the dependency set and thereby potentially discharges some of the clauses in the path condition of the partial sequence.

The anchors are identified using the UI model and the event handler summaries. First, we perform a breadth-first backward traversal in the UI model, starting from the abstract state of the partial sequence, until the nearest anchors are located. More precisely, at each traversed transition in the UI model, the dependency set of the partial sequence is compared with each path summary that belongs to the event handler of the transition. A path summary is marked as an anchor if it affects the dependency set.

In the example from Section 2, the target in the event handler e_2 shown in Figure 3 depends on the symbolic constraint variable `Settings.enableTaxDeduction`. A partial sequence containing a path summary for e_2 will include this variable in its dependency set. Since the path summaries for the event handler e_7 all affect this particular variable, they will be identified as anchors for the partial sequence.

Some of these anchors, however, can safely be pruned away. If the symbolic state of an anchor is inconsistent with the path conditions of the current partial sequence (i.e. their conjunction is unsatisfiable), then using this anchor for extending the partial sequence would not lead to any feasible paths. By removing such anchors from further consideration, we effectively reduce the search space of the sequence generation algorithm. The resulting set of anchors is returned by the ANCHORS function.

As mentioned, the idea of using anchors is to guide the sequence generation from the target backward toward the entry state. For a complete sequence, that is, a partial sequence that has reached the goal, the dependency set is empty. The idea in our algorithm is that putting an anchor in front of a partial sequence will likely reduce the dependency set. However, there is no guarantee that the dependency set is in fact reduced by this step, since the anchor itself may introduce additional dependencies. Our experimental evaluation in Section 7 investigates how the use of anchors guides the search in practice.

6.2 Construction of Connector Sequences

The PATHS function generates a set of possible connector sequences between the given anchor and partial sequence. For this, we use the UI model to find all sequences of connectors between the two. Each of these sequences have the following two properties: (1) it corresponds to an acyclic path in the UI model from a transition that has the anchor as label to the abstract state at the beginning of the partial sequence, and (2) none of the connectors, where each corresponds to a single transition in the UI model, is an anchor. The first property can be ensured using a basic graph traversal algorithm. Section 6.1 provides the information for ensuring the second property.

Each connector sequence that has these properties is a candidate for connecting the given anchor and partial sequence. Not all of these candidates are feasible, however. If the symbolic state of the anchor is inconsistent with the composition of the symbolic summaries of the connectors, then no corresponding concrete path exists. The remaining feasible paths are then returned by the PATHS function.

Continuing the example of the partial sequence containing a path summary for e_2 and an anchor for e_7 from Section 6.1, a path summary for e_6 is a connector, since it connects the two in the UI model and it does not affect the dependency set of the partial sequence.

6.3 Prioritization

A key part of the algorithm is the PRIORITIZE function that assigns priorities to newly added partial sequences. This function initially selects the priority of a new sequence as the priority of the sequence it extends. The priority is then adjusted using a series of *reprioritization functions* representing different heuristics that we describe in the following.

6.3.1 Equivalent-Anchors Reprioritization

An event handler summary consists of a set of path summaries. When extending a partial sequence with anchors, we look at their path summaries to determine if they write any program state that the partial sequence depends on. Since multiple paths in an event handler can result in the same mutation of the variables that appear in the dependency set of the partial sequence, the resulting set of anchors will likewise contain multiple candidates with the same effect. Each of these anchors results in a new partial sequence in the worklist.

As an example, if we assume the dependency set is {`income`} and we consider the event handler in Figure 3, there exist multiple paths through the event handler that all have the same effect on `income`, so giving them the same priority would lead to redundant work.

Our first reprioritization function exploits this observation by lowering the priority of all the involved partial sequences, except one that we pick arbitrarily. Finding the anchors that have an equivalent effect relative to the dependency set can be done by comparing the constant values and symbolic values in their symbolic states.

6.3.2 Connector Reprioritization

There may exist multiple sequences of connectors between a given anchor and a partial sequence. Recall our observation in Section 4 that these connectors only navigate between screens in the application, without affecting the program state that the partial sequence depends on. In many cases, any of these paths will suffice, and it would only lead to a path explosion if we try to follow all of them.

Based on this, we introduce a second reprioritization function that exploits this observation by lowering the priority of all the involved partial sequences, except one that we pick arbitrarily, similar to the previous reprioritization function.

6.3.3 Increment-Decrement Reprioritization

Another common pattern is pairs of path summaries, in which one changes some state, and another reverts those changes. This general pattern manifests itself in a number of concrete instances, such as add/remove buttons that mutate a collection of items, buttons for incrementing or decrementing a number, or buttons toggling a value.

Extending partial sequences with path summaries that simply undo changes is not productive and may lead to unnecessary exploration of paths. Note that we only care about parts of the program state that are in the dependency set of the current partial sequence. Our third reprioritization function aims at decreasing the priority of any partial sequence where this pattern is found. This is not trivial to detect precisely, however. A simple approximation is to consider only numeric counters and boolean flags. Whenever the reprioritization function identifies a pair of path summaries where one increments some variable the other decrements the same variable, then the priority of event sequences that mix the two path summaries is lowered, and similarly for boolean flags.

7. EVALUATION

To evaluate the practical usefulness of our approach, we have implemented the proposed event sequence generation algorithm and supporting infrastructure in a tool called *Collider*. We now consider the following research questions:

Q1. Is our algorithm able to generate test cases for challenging targets in real-world Android applications? We view a target as being "challenging" if it cannot be reached with traditional random testing or model-based testing techniques.

Q2. Does the use of anchors and connectors have an effect on the the ability to reach the targets? A simple alternative would be a backward breadth-first search in the UI model.

Q3. Do the prioritization heuristics have an effect on the ability to reach the targets? If that mechanism is disabled, the partial sequences in the worklist will be treated in a random order.

7.1 Implementation

Collider is implemented with approximately 8,000 lines of Java code excluding libraries. The part implementing the sequence generation phase closely follows the pseudo-code from Section 6, whereas the part for symbolic summarization requires more explanation.

A central part of Collider is the concolic execution engine for symbolically summarizing event handlers as described in Section 5. As the concolic execution is performed at the level of event handlers (including methods called in the process), application state that may be shared between event handlers, in particular, all object fields, are initialized with symbolic values. Collider operates directly on the Dalvik bytecode of the compiled Android applications. We do not differentiate between application code, Android library code, and the Java standard library, however, the symbolic execution uses mocks for more precise treatment of some basic library methods.

The concolic execution engine must be able to evaluate Android applications concretely, inspect the evaluation and program state, and modify the program state in order to explore new branches. In Collider, concrete execution is handled by the Android emulator provided by the Android SDK, which ensures a correct execution of the application. All interaction between Collider and the Android emulator is handled by a combination of the ordinary instrumentation framework for testing Android applications and the debugging interface in the Android VM. Via the debugger, breakpoints are inserted after each bytecode instruction, such that the symbolic execution can be performed in parallel with the concrete execution in a lock-step manner. Using this technique, neither the application nor the emulator needs to be modified in any way, which simplifies the implementation.

For the symbolic execution, we reuse parts of the solver infrastructure from Symbolic Java PathFinder,[1] which in turn relies on underlying solvers, such as, Yices.[2] The solver infrastructure is also used to check the feasibility of partial sequences, as described in Section 6. This implementation currently supports basic constraints on numbers, booleans, strings, and arrays. The Smali[3] disassembler is used for extracting various pieces of information about the Dalvik bytecode, and the testing library Robotium[4] is used for simulating user interactions with the application.

7.2 Benchmark Applications and Targets

Our evaluation has been conducted on five Android applications selected using the following criteria: (1) the source code for the applications must be available to allow us to manually inspect the application behavior, (2) we only consider applications that are UI driven and not computationally intensive, so we exclude games and system services, (3) to get interesting targets, the applications must contain branches that depend on previous events or event parameters, and (4) the applications should represent different application categories, such as productivity, entertainment, and tools, and from different repositories. The five applications are: TippyTipper[5] (1,800 LOC), a tip percentage calculator including tax calculation and functionality

[1] http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc
[2] http://yices.csl.sri.com/
[3] http://code.google.com/p/smali/
[4] http://code.google.com/p/robotium/
[5] http://code.google.com/p/tippytipper/

Table 1: The first two columns of numbers show the targets that remain unreached after running both Monkey tool and the crawler. The third and fourth columns show the additional targets reached by Collider, and targets that we believe could potentially be reached if improving the symbolic execution implementation, respectively. The fifth and sixth columns show the average sequence length of the produced test cases and the average number of connector events in the test cases. The seventh column shows the percentage of anchors pruned from the search during sequence generation. The numbers in the rightmost three columns are for the reached targets only.

	Targets depending on				Average size		
Benchmark	Sequence	Parameters	Reached	Potential	Test case	Connectors	Pruning of anchors
TippyTipper	15	1	7	9	13	9	71%
ConnectBot	17	25	16	26	8	4	58%
Munchlife	5	5	6	4	29	7	66%
OpenManager	11	7	9	9	8	4	39%
DieDroid	2	11	8	5	10	4	38%

for splitting a bill between a group of people; ConnectBot[6] (33,000 LOC), a SSH client with support for public/private key management; MunchLife[7] (400 LOC), a utility for keeping score in a card game; OpenManager[8] (2,500 LOC), a file manager with support for viewing, moving, and copying files; and DieDroid[9] (1,900 LOC), an application for virtual dice rolling using a number of different systems and conditions on the die rolls.

We have manually built a UI model of each application. This could in principle be done automatically, as discussed in Section 8, but no suitable tool was available to us when conducting the experiments. These UI models are sound in the sense described in Section 3.

To obtain a baseline for comparison, we combine two existing approaches. First, we use a a simple crawler that produces events systematically based entirely on the UI models, without considering the application code. Second, we use the random testing tool Monkey provided by the Android SDK.[10] This tool fires a large number of random events and periodically restarts the application. When we run these tools on the benchmark applications, we observe that coverage first rises rapidly and then stabilizes. We select a time budget that allows the stabilization to be reached. As a result, the different benchmarks have been exercised using 3,000 to 6,000 events each. Now, we define that a branch in a benchmark application is "challenging" if neither of these two tools is capable of producing event sequences that reach the target. Although the tools involve randomization, this classification appears to be reasonably robust.

For our evaluation of Collider, we only consider the challenging targets. In practice, this means that all targets of interest depend on the sequencing of events beyond what is expressible in the UI models alone.

We focus on branches in the application code, not on those in the Android SDK and external libraries. Moreover, we exclude dead branches, i.e. those that cannot be reached with any sequence of events according to a manual inspection. As our focus is on user events, we also exclude branches that depend on external data, such as, the file system or device configuration. For the same reason, we populate the initial application states with meaningful data, such as, files for OpenManager and a valid SSH connection for ConnectBot.

[6] `http://code.google.com/p/connectbot/`
[7] `https://github.com/sensae/MunchLife`
[8] `https://github.com/nexes/Android-File-Manager`
[9] `https://github.com/logomancer/diedroid`
[10] `http://developer.android.com/tools/help/monkey.html`

The first two columns of numbers in Table 1 show for each benchmark (1) the number of targets of interest that depend on event sequencing, but not on event parameters, and (2) the number of targets that depend on both event sequencing and event parameters. This classification is obtained by a manual inspection of each target.

7.3 Results

Q1: We answer research question Q1 by applying our event sequence generation algorithm on the selected targets. The column named Reached in Table 1 shows the number of targets where a successful test case is generated. For example, 7 of the 16 targets of interest in TippyTipper are reached. As we only consider targets where the baseline tools fail, we conclude that our proposed algorithm is capable of producing test cases for challenging targets.

As an example of a challenging target, the DieDroid benchmark contains a screen that shows a number of rolled dice, marked red, green or gray depending on user-defined winning and failure thresholds. In one event handler, a particular branch can only be reached if the winning threshold is larger than the loosing threshold. To reach this target branch, our algorithm identifies a path summary exercising the branch. The algorithm then continues to extend the partial sequence backward, finding anchor points in two separate dialogs that set these thresholds, and inserting connector events as necessary. Each of these anchors is parameterized by user input, for which the solver identifies two values that satisfy the path condition. After extending the partial sequence to the application entry point, the tool outputs a concrete event sequence that reaches the target.

We have manually inspected all the targets that were not reached by Collider to investigate whether the reason is due to limitations in the symbolic summarization phase or due to the assumptions we make in the sequence generation phase. In the former case, the limitations can perhaps be remedied by improving the concolic execution engine; in the latter case, more fundamental changes to our approach might be necessary to increase the coverage further. The number of missed targets in the first category are shown for each benchmark in the column named Potential in Table 1: We observe that none of the missed targets are in the second category. Our prototype implementation only supports symbolic reasoning of numeric values and booleans, resulting in imprecise treatment of, for example, strings and objects. A closer inspection reveals that this particular source of imprecision is a dominant cause of missed targets. For this reason, we believe that many of the challenging targets that are not

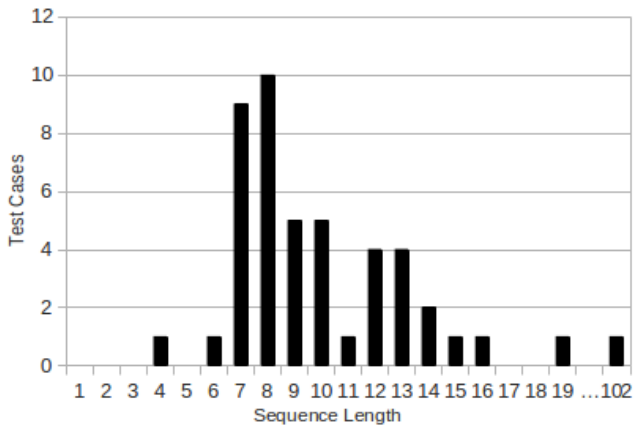


Figure 5: Distribution of event sequence lengths in the generated test cases.

reachable with our current implementation can potentially be reached with realistic improvements of the symbolic analysis, without requiring modifications of the main sequence generation phase. Naturally, we will pursue this in our further work.

The event sequence generation phase, which is the central part of our algorithm, typically takes less than one minute per target where a matching event sequence is found (running on an ordinary i5 3.1GHz PC). A few outliers, for example in TippyTipper, require up to 30 minutes. This is caused by a large number of connectors between partial event sequences and anchors. An adjustment of the prioritization mechanism could perhaps give a performance improvement in this case, but a more thorough experimental study would be necessary to investigate this further.

The symbolic summarization phase is more time consuming. Our simple prototype implementation runs for 3 to 5 hours on each benchmark. However, note that this phase is doing preprocessing, independent of the choice of targets. Moreover, the current implementation naively analyzes each event handler in isolation, without taking into account that event handler methods often share common functionality via other methods. Thus a considerable amount of time is spent re-analyzing shared methods. With additional implementation effort, we believe that well-known techniques can be adapted to avoid this redundancy, which we return to in Section 8. Another reason is the implementation approach: Basing the concolic execution engine on single-stepping via the Android VM debugger does lead to a relatively simple implementation, but it naturally incurs a substantial overhead.

Q2: To answer research question Q2, we investigate how the use of anchors and connectors reduces the search space compared to a simple backward breadth-first search.

The distribution of observed test case lengths is shown in Figure 5. The sequence lengths range from 4 to 102 events, however, with the test case containing 102 events being an outlier. On average, a test case consists of 10 events if we exclude this outlier. The average test input length for each benchmark is shown in the 'Average size Test case' column in Table 1. With such relatively long event sequences, a simple backward breadth-first search would lead to an explosion of possible paths.

Of all the generated test cases excluding the outlier, 53% of the events are connector events. The average number of connector events per test case is shown in the Connectors column in Table 1. Since a considerable part of the events are connectors, the ability to jump between anchors is an advantage compared to a backward breadth-first search.

The idea of pruning potential anchors by checking consistency of the symbolic states and the path conditions further reduces the search space. The rightmost column in Table 1 lists the pruning of anchors for each benchmark. The pruning eliminates 38%-71% of the anchors. Since this happens in each step of the backward search, it adds up to a substantial reduction of the search space.

Q3: For research question Q3, we disable the prioritization functions and run Collider again. In theory, we are still able to reach the same targets, however, we expect a slower pace due to the larger number of paths that need to be considered before finding test cases that reach the targets. We want to test if Collider is still able to reach the same targets, even if we allow ten times as many iterations of the worklist algorithm compared to number of iterations used when the prioritization functions are enabled.

Running our algorithm again, 21 of the 46 targets are now unreachable. Moreover, for the remaining 25 branches that are still reached, the total running time for the sequence generation has increased from 45 seconds to 2.5 hours. Thus, we conclude that the prioritization heuristics have a considerable impact on the ability to reach targets within reasonable time.

A possible threat to validity in our evaluation is whether the selected benchmarks represent the range of real-world applications in use. All of the selected benchmarks are real-world applications publicly available in the Android marketplace, and they have been selected in accordance with the criteria stated in Section 7.2. The nature of our evaluation, involving manual inspection of the benchmarks and manual construction of UI models, reduces the feasibility of scaling the evaluation to a larger number of benchmarks. However, these preliminary experiments demonstrate the potential of our algorithm.

8. RELATED WORK

Our work builds on a significant body of work in symbolic execution and model-based testing. We first discuss related work involving symbolic execution for event-driven programs.

As in our approach, the ACTEve technique by Anand et al. [3] performs automated testing for Android applications using concolic execution. However, their approach explores the application starting from the its entry point, not aiming for particular targets. Concolic execution is applied at the level of the entire application rather than on individual event handlers. Moreover, concolic execution is used for reasoning about low-level properties of events, such as coordinates for tap events, which we can treat more abstractly by the use of UI models. Despite applying a subsumption mechanism to filter away certain event sequences, their approach apparently does not scale beyond event sequences consisting of more than four events.

The Barad framework by Ganov et al. [9] performs automated testing for SWT GUI applications, which are also event-driven. It first symbolically executes each event handler, not to produce path summaries as in our approach, but

to discover registered event handlers and build a model of the application similar to the UI models we use. Next, a set of abstract event sequences are produced from the model, and symbolic execution is performed on each sequence to produce concrete test inputs. Mirzaei el al. [20] generate tests for Android applications using a similar approach by first producing abstract event sequences based on application models and then running Symbolic PathFinder to perform symbolic execution on each sequence. In contrast, our approach utilizes information from symbolic execution also when constructing the sequencing of events. Several other symbolic execution tools have been build specifically for Android [14, 25]. Related tools for automated testing of web applications, which are also driven by user events, include Apollo [6] for PHP and Kudzu [23] for JavaScript. Common to these frameworks and tools is that they do not create event sequences in a targeted manner but explore the given application from its entry point.

As mentioned in Section 1, our targeted approach to generation of event sequences resembles call-chain-backward symbolic execution by Ma et al. [17], although we consider relations between events rather than function calls. In their approach, call sequences are generated backward from the target one function at a time. We also construct event sequences backward, but using anchors and connectors to narrow the search, as explained in Section 6.

The idea of guiding automated testing using data dependence appears in many techniques [3–5, 7, 8]. Of particular relevance is the one by Arlt et al. [4] that we also mentioned in Section 1. In their technique, abstract event sequences are constructed based on how event handlers read and write shared state and subsequently concretized using a UI model, but reasoning at the level of entire event handlers rather than individual paths through event handlers. A novel feature of our approach is that path-specific data dependence information is extracted from event handler summaries that have been created using concolic execution.

For the symbolic summarization phase, we currently use traditional concolic execution, also called dynamic symbolic execution, or directed automated random testing [11], at the level of event handlers. We can in principle benefit from the numerous improvements that have been proposed to that basic technique. Specifically, we suspect that performance of the symbolic summarization phase can be improved using compositional dynamic test generation [10], which involves method summaries, orthogonal to our use of event handler summaries.

Alternatives to symbolic execution for automated testing include random testing, search-based testing, and model-based testing. Monkey, which we used for the experiments in Section 7, is a popular random testing tool for Android that has been shown to be effective for bug finding [13]. The tools A^2T^2 [1], AndroidRipper [2], iCrawler [15], and EXSYST [12] enhance random testing by using the application GUI to guide the testing. These light-weight techniques can be a good starting point for automated testing. However, as they have a black-box view on the application code, they are generally unable to reach the challenging targets that require many events and with constrained event parameters and specific execution paths in the event handlers, as shown in Section 7.

Tools such as Artemis [5] and to some extent also Dynodroid [18] employ feedback-directed automated testing, which is based on random testing but prioritizing using information gathered during the testing. Such techniques can often obtain good coverage with fewer test inputs than traditional random testing and faster than techniques that involve symbolic execution, yet they are not suitable for the more challenging targets that we focus on here.

Model-based testing approaches [22] organize the testing around a model of the application under test. For the event-driven applications we consider, the models express over-approximations of the relevant event sequences by abstracting away from the event parameters and the different execution paths that exist in the event handlers. Some tools extract tests for Android applications directly from such models using random or combinatorial approaches [21, 24], without involving symbolic execution.

The models used in model-based testing may be specified manually or generated automatically. The GUITAR tool by Memon et al. [19] is among the earliest and most well known approaches for reverse engineering models of GUI applications. It extracts the model using automated crawling. A recent extension, AndroidGUITAR, supports Android applications. The Orbit tool by Yang et al. [27] is a variant that builds models that are tailored to the Android event system. Several of the other techniques that we have mentioned above also automatically construct models [3, 9, 20]. Although the various techniques involve different kinds of models, each of them can in principle provide the information we need for the UI models described in Section 3.

We distinguish between anchor events and connector events, however, other classifications exist. As an example, Xie and Memon [26] categorize events according to whether they manipulate the GUI while we focus on how the events modify data.

9. CONCLUSION

We have presented a targeted algorithm for automated testing of event-driven systems, in particular Android applications. The algorithm is tailored to targets that require long event sequences and reasoning about event parameters. We have evaluated the effectiveness of this algorithm on a small suite of real-world Android applications, aiming for targets that are beyond reach for traditional random testing and model-based testing techniques. Our prototype implementation, Collider, successfully produces event sequences for many of the challenging targets.

Moreover, we believe that a large part of the remaining targets can also be reached using the algorithm, provided that the symbolic constraint solver component is extended with better support for, in particular, strings and arrays. We leave that for future work. Also, we plan to apply some of the techniques suggested in the literature on concolic execution, for example, compositional dynamic test generation, to improve performance of the symbolic summarization phase. Another practical limitation of our current prototype is that it requires UI models as input. This can in principle be remedied by integrating existing algorithms for automatic UI model construction. Such an extension of the implementation would enable a larger scale experiment in which Android applications are automatically analyzed and tested. For this purpose, it is practical that our approach works on bytecode and does not need access to the source code of the applications.

10. REFERENCES

[1] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. A GUI crawling-based technique for Android mobile application testing. In *Proc. 3rd International Workshop on Testing Techniques and Experimentation Benchmarks for Event-Driven Software*, 2011.

[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using GUI ripping for automated testing of Android applications. In *Proc. 27th International Conference on Automated Software Engineering*, 2012.

[3] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proc. 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2012.

[4] Stephan Arlt, Andreas Podelski, Cristiano Bertolini, Martin Schäf, Ishan Banerjee, and Atif Memon. Lightweight static analysis for GUI testing. In *Proc. 23rd IEEE International Symposium on Software Reliability Engineering*, 2012.

[5] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A framework for automated testing of JavaScript web applications. In *Proc. 33rd International Conference on Software Engineering*, 2011.

[6] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit M. Paradkar, and Michael D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Transactions on Software Engineering*, 36(4), 2010.

[7] Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. RWset: Attacking path explosion in constraint-based test generation. In *Proc. 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[8] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 1996.

[9] Svetoslav R. Ganov, Chip Killmar, Sarfraz Khurshid, and Dewayne E. Perry. Event listener analysis and symbolic execution for testing GUI applications. In *Proc. 11th International Conference on Formal Engineering Methods*, 2009.

[10] Patrice Godefroid. Compositional dynamic test generation. In *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007.

[11] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[12] Florian Gross, Gordon Fraser, and Andreas Zeller. Search-based system testing: high coverage, no false alarms. In *Proc. 21st International Symposium on Software Testing and Analysis*, 2012.

[13] Cuixiong Hu and Iulian Neamtiu. Automating GUI testing for Android applications. In *Proc. 6th International Workshop on Automation of Software Test*, 2011.

[14] Jinseong Jeon, Kristopher K. Micinski, and Jeffrey S. Foster. SymDroid: Symbolic execution for Dalvik bytecode. Technical report, 2012.

[15] Mona Erfani Joorabchi and Ali Mesbah. Reverse engineering iOS mobile applications. In *Proc. 19th IEEE Working Conference on Reverse Engineering*, 2012.

[16] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

[17] Kin-Keung Ma, Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In *Proc. 18th International Static Analysis Symposium*, 2011.

[18] Aravind MacHiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for Android apps. Technical report, 2012.

[19] Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proc. of The 10th Working Conference on Reverse Engineering*, November 2003.

[20] Nariman Mirzaei, Sam Malek, Corina S. Pasareanu, Naeem Esfahani, and Riyadh Mahmood. Testing Android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 37(6), 2012.

[21] Cu D. Nguyen, Alessandro Marchetto, and Paolo Tonella. Combining model-based and combinatorial testing for effective test case generation. In *Proc. 21st International Symposium on Software Testing and Analysis*, 2012.

[22] Mauro Pezzè and Michal Young. *Software testing and analysis - process, principles and techniques*. Wiley, 2007.

[23] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Stephen McCamant, Dawn Song, and Feng Mao. A symbolic execution framework for JavaScript. In *Proc. 31st IEEE Symposium on Security and Privacy*, 2010.

[24] Tommi Takala, Mika Katara, and Julian Harty. Experiences of system-level model-based GUI testing of an Android application. In *Proc. 4th IEEE International Conference on Software Testing, Verification and Validation*, 2011.

[25] Heila van der Merwe, Brink van der Merwe, and Willem Visser. Verifying Android applications using Java PathFinder. *ACM SIGSOFT Software Engineering Notes*, 37(6), 2012.

[26] Qing Xie and Atif M. Memon. Using a pilot study to derive a GUI model for automated testing. *ACM Transactions on Software Engineering and Methodology*, 2008.

[27] Wei Yang, Mukul R. Prasad, and Tao Xie. A grey-box approach for automated GUI-model generation of mobile applications. In *Proc. 16th International Conference on Fundamental Approaches to Software Engineering*, 2013.