Proceedings Paper:
Walkinshaw, N. (2020) Improving automated GUI testing by learning to avoid infeasible tests. In: Proceedings of the 2020 IEEE International Conference On Artificial Intelligence Testing (AITest). 2020 IEEE International Conference On Artificial Intelligence Testing (AITest), 03-06 Aug 2020, Oxford, UK. IEEE , pp. 107-114. ISBN 9781728169859

https://doi.org/10.1109/AITEST49225.2020.00023

Improving Automated GUI Testing by Learning to Avoid Infeasible Tests

Neil Walkinshaw
Department of Computer Science
The University of Sheffield, UK
Email: n.walkinshaw@sheffield.ac.uk

***Abstract*—Most modern end-user software applications are controlled through a graphical user interface (GUI). When it comes to automated test selection, however, GUIs present two major challenges: (1) It is difficult to automatically identify feasible, non-trivial sequences of GUI interactions (test cases), and (2) each attempt at a test case execution can take a long time, eliminating the possibility of rapidly attempting large numbers of alternatives. In this paper we present an iterative approach that infers state-machine models from previous test executions, and increases the utility of tests by learning which sequences to avoid. The approach is evaluated on a selection of Java applications, and the results indicate that our approach is successful at achieving higher code coverage and longer sequences than the state of the art, albeit with a time-overhead caused by the repeated invocation of a Machine Learner.**

I. Introduction

GUIs raise several challenges when it comes to automated software testing. They can comprise a large variety of windows with different combinations of of widgets (e.g. buttons, check-boxes, text-entry fields, etc.), where the appearance or contents of certain windows and widgets can depend upon previous inputs. Accordingly, test cases that seek to fully explore the behaviour of the underlying system can be required to include complex sequences of selections and inputs.

There exists a very large number of GUI-testing tools, spanning mobile apps, web-apps, and desktop GUIs. The goal is the same as with any testing technique - to identify a manageably small set of test cases that is sufficiently rigorous and diverse to expose any faults. However, the challenge with GUIs is especially challenging because (1) GUI-based applications can take a long time to initialise and execute, (2) the GUI interface is invariably dynamic – the input 'surface' can change from one interaction to the next, and as a result (3) 'test cases' amount to potentially complex sequences of widget clicks, drags, gestures, etc.

In this paper we investigate a solution for scenarios where there is no capability of analysing and querying the run-time GUI state. We may, for example, be interested in testing an application across a multitude of platforms. We consider the scenario where we are able to supply the SUT with a range of inputs (in a programmable way via some testing interface). We also presume that we start from a possibly large set of potential test sequences, which may arise from some GUItar-style analysis of the SUT [1], be randomly generated, they may be a product of fuzzing [2]. In any case, a large proportion of these test cases are liable to be trivially invalid, and lead to (expensive) application re-starts after only few interactions. We do *not* assume that we are able to query or scrutinise the state of the system under test during a test execution (e.g. to determine which inputs are feasible at any given point).

Our solution is superficially similar to existing solutions [3], [4]. We use a state machine inference to infer models of what has been tested so far and use this model to inform the selection of new test cases. However, in our scenario it is especially important that the inferred model is able to discriminate between sequences of events that have been explored so far, and sequences of events that have been explored but should be avoided in future executions because they will lead to some undesirable outcome (e.g. a time-out). To address this, our paper makes the following contributions:

- We show how GUI test-executions can be fed to the EDSM state machine inference algorithm [5](previously used for Android SwiftHand [3]), in a way that takes advantage of its capacity to distinguish between positive *and negative* examples to produce models that are capable of distinguishing between interaction sequences that have been attempted, and sequences that are likely to be invalid or lead to time-outs.
- We present an algorithm that uses the resulting model to filter-out and prioritise GUI test cases.
- We have developed an openly available implementation of the approach.
- We present an empirical evaluation on five GUI-based Java applications, which demonstrates that the use of our approach leads to longer interactions and greater code coverage than a quasi-random use of GUItar.

II. Background and Related Work

We start with a brief introduction to the landscape of automated GUI testing. Since state machine inference has played a reasonably prominent role in GUI testing (and forms the basis for our approach too) we provide a brief introduction to state machines and state machine inference. We then discuss some of the specific ways in which state machine inference has been used for GUI testing, and discuss some of the weaknesses (or missed opportunities) of these approaches.

A. Automated GUI Testing

There has been a gradual evolution of GUI-testing tools. Early GUI-testing tools, most notably GUItar [6] and its mobile version MobiGuitar [7], worked in two phases. In the first phase, an analysis of the source code, perhaps enhanced by a dynamic analysis, would construct a model of capturing the possible range of interactions with the SUT. This is then followed by a test selection process [1], where the goal is to meet various objectives - to achieve maximum coverage of the model (and code), with the fewest possible number of test cases (because application restarts for new tests are especially time-consuming).

In GUITar and MobiGUItar the model of the target GUI was encoded as an Event Flow Graph [1]. This is a graph that contains labels corresponding to GUI events, where transitions indicate the order in which these events are deemed to be possible.

Definition 1: An EFG is a directed graph (V, E, I), where each element $v \in V$ corresponds to an event in the GUI. E is a set of edges (v_i, v_j), indicating that event v_i can be succeeded by v_j. $I \subseteq V$ is a set of initial vertices, indicating that these can act as starting points for a GUI interaction.

One major limitation of this approach is the fact that the model constructed in the first phase is not entirely accurate. The use of static analysis invariably means that the model will indicate that certain sequences of events should be possible when they are not. As a result, such approaches can end up attempting large numbers of test cases that are futile [8], [9], leading to many re-starts of the application without achieving significant coverage [4].

In recent years, GUI-testing tools have worked around the limitations posed by such a-priori models by exploiting the emergence of increasingly sophisticated technical facilities within APIs to query and log GUI-interactions. As a result a large range of Android-based testing tools [3], [10], [11], [12], [2], [4] have emerged, which take advantage of such capabilities, and are able to successfully generate long, exploratory test sequences. Similarly for Windows applications, Testar [13], [14] has emerged as a leading tool, able to query the state of a GUI during execution via the Windows accessibility layer.

This progress does however come at a cost. These techniques and tools tend to be tied to the underlying platform for which they have been developed, vulnerable to any sudden changes to interfaces within the target API or OS platform. They can only be re-engineered to an alternative platform if it offers a comparable interface with runtime access to the underlying GUI state.

B. State Machines

Definition 2: A Deterministic Finite Automaton (DFA) is a quituple $(Q, \Sigma, \Delta, F, q_0)$, where Q is a finite set of states, Σ is a finite alphabet, $\Delta : Q \times \Sigma \rightarrow Q$ is a partial function, $F \subseteq Q$ is a set of final (accepting) states, and $q_0 \in Q$. A DFA can be visualised as a directed graph, where states are the nodes, and transitions are the edges between them, labelled by their respective alphabet elements.

Algorithm 1: Basic State Merging Algorithm

Input: Two samples S^+ and S^- containing positive and negative examples respectively
Result: A DFA consistent with S^+ and S^-

```
1  Infer(S+, S-) begin
2      PTA ←initialize(S+, S-);
       // Let N denote the number of states
          in the PTA
3      π ← {{0}, {1}, ..., {N − 1}};
4      while (B_i, B_j) ← ChoosePair(π) do
5          π_new ← Merge(π, B_i, B_j);
6          if Compatible(PTA/π_new, S-) then
7              π ← π_new;
8      return PTA/π

       // Merge pair of states and ensure that
          the result is deterministic
9  Merge(π, B_i, B_j) begin
10     π ← π \ {B_i, B_j} ∪ {B_i ∪ B_j};
11     while
       (B_k, B_l) ←FindNonDeterminism(π, B_i, B_j) do
12         π ←Merge(π, B_k, B_l);
```

When discussing the *behaviour* of a DFA, we are referring to the possible (and impossible) sequences of elements in Σ (denoted Σ^*). The set of all possible sequences in a DFA is referred to as its *language*. To define this formally, we draw on the inductive definition for an extended transition function $\hat{\delta}$ used by Hopcroft *et al.* [15]. For a state q and a string w, the extended transition function $\hat{\delta}$ returns the state p that is reached when starting in state q and processing sequence w. For the base case $\hat{\delta}(q, \epsilon) = q$. For the inductive case, let w be of the form xa, where a is the last element, and x is the prefix. Then $\hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a)$.

Definition 3: The language $L(A)$ of a DFA A is the set of strings reaching any state in A from its initial state. $L(A)$ is defined as follows: $L(A) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}$. The complement of a language L is denoted L^C (i.e. the set $\Sigma^* \setminus L$ of strings that do not belong to L). Sequences $w \in \Sigma^*$ for which $\hat{\delta}(q_0, w)$ is not defined are considered to be rejected by the automaton.

C. State Machine Inference

Although the challenge of inferring an exact state machine has been shown to be NP-hard [16], several algorithms have emerged that have been shown to be capable of inferring reasonably accurate approximations. It has been shown that, given a sufficiently diverse set of positiven *and negative* examples, it is possible to infer a state machine that is 'Probably Approximately Correct' [17] in polynomial time [18], [19].

Amongst a variety of inference algorithms, the State Merging algorithm [19], [5] is is particularly prevalent in Software Engineering [20], [21], [22], [3], [23], [24], [25], [26], [27],

[28], [29], [30], and is detailed in Algorithm 1. In essence, the approach starts from two sets of sequences: S^+ - a set of sequences that are accepted by the subject, and S^- - a set of sequences that are not. From these it constructs a tree-shaped automaton (a 'prefix-tree automaton') that exactly represents a given set of sequences (line 2). It then adopts some form of heuristic to select which pairs states to merge with each other (lines 4-5, 9-12), thereby producing a state machine that generalises on the initial set of sequences. If the resulting machine correctly rejects all sequences in S^- (it accepts all strings in S^+ by construction), then the merge is accepted and the process iterates (lines 6-7) until no further merges can be identified and the final machine is returned (line 8).

Software engineering applications, including the various inference-based testing approaches, have largely been based on situations where there are no 'negative' sequences for S^-, but only instances of observed execution traces belonging to S^+. In such situations, to prevent the merging process from over-generalising to produce a single-state machine that trivially accepts all sequences, it is necessary to constrain the `ChoosePair` function. To this end, techniques such as k-tails [21], [22] tend to only select merge-candidates if their outgoing paths fulfill some sort of 'similarity' criterion (e.g. outgoing paths must match each other up to some specified length k).

D. State Machine Inference and GUI Testing

State machine inference has been successfully applied to test sequential software systems that are not GUI-driven (notably network protocols) in the past [31]. There is a natural link between GUIs and state machines, which has been the subject of an extensive amount of research [32], and which makes GUIs apparently ideal candidates for testing approaches that incorporate state machine inference. The idea was first explored by Mariani *et al.* with the AutoBlackTest approach [12], which used QLearning [33] to infer behaviours from the GUI under test as it is being tested, and to then use this as a basis for selecting new inputs. Subsequently, Choi *et al.* developed the SwiftHand [3] Android testing tool (which is based upon a state merging algorithm). The subsequent FSMDroid [4] was also based on a similar premise, whilst including stochastic weights in the state machine.

The evidence to corroborate the performance of these approaches is mixed. In the domain of Android testing, a 2015 study [10] saw techniques such as SwiftHand comprehensively outperformed by the very Android Monkey tool. Studies that examine the relative performance of AutoBlackTest and GUITar subject to the caveat that their relative performance can vary significantly depending on configurations and subject systems [34]. However results by FSMDroid appear to be promising [4] (outperforming successful tools such as Sapienz [2]). Invariably, when comparing techniques it can be difficult to disentangle performance gains that are due to tool implementation details from gains that are due to the underlying technique.

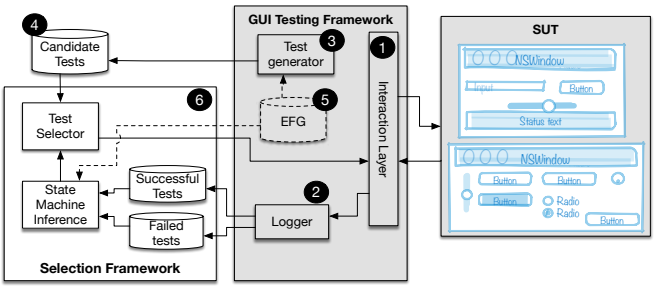

Fig. 1. Testing set up.

One characteristic that applies to all of these techniques is that they build a model from a *single set* of test executions (or dynamic traces obtained before testing). All test executions are treated the same – regardless of whether they terminated successfully or ended in a time-out and had to be aborted. In the terms of the state merging algorithm presented in Algorithm 1, all of the traces belong to set S^+ and S^- is empty.

This severely hampers model inference; without any negative examples, the inference algorithm is vulnerable to over-generalisation [16], [19]. In a pathological worst-case this would result in a single-state DFA where all sequences lead to the same positive outcome. To avoid this scenario, techniques are reduced to either crudely guessing whether two states are equivalent (e.g. by means of the k-tails heuristic [21], [22]), or by using run-time GUI querying APIs to inspect the current GUI state.

Aside from the problems of inference-accuracy, there is also a consequence for the semantics of the inferred model. Without any negative information the inferred models tend to be 'prefix-closed', meaning that any sequence and prefix thereof through the inferred model is valid. This leads to a simplified form of DFA (formally a Labelled Transition System (LTS) [35]) where $F = Q$; every state is a potential final state, and there is no ability to discriminate between sequences that are valid, and those that should be avoided (e.g. because they lead to costly time-outs and restarts [9]).

III. Inference with Negative GUI Test Sequences

This paper is based on the observation that the context of GUI-testing offers plenty of sources of 'negative' information. In an inference-supported testing context, these sources of information can be easily incorporated into well-established inference techniques. This raises the possibility of inferring more accurate models, and using these models to suppor the selection of better test-cases.

A. Testing Scenario

We demonstrate this process with respect to the traditional "gui-ripping" testing scenario [1]. For the sake of practicality, we seek to limit our practical requirements where possible. We describe the key components (and distinguish between those that are absolutely necessary and those that are desirable) with

respect to the GUI-testing setup illustrated in the grey-shaded elements in Figure 1.

The most important requirement is access to a GUI testing setup that is able to interact with the SUT (we refer to this as the "interaction layer" - 1). This is what enables us to supply test sequences (sequences of interactions) and for them to be applied as GUI interactions with the SUT. One particularly important requirement is that we are able to surmise whether or not an attempted test execution has completed or not. In GUITar, for example, there is a logging facility (2) that records, for each test execution, which events were executed and at what point (if any) an attempted interaction failed or resulted in a time-out. We do not assume run-time access to the test execution, or an ability to query the GUI during test executions.

We assume that there is some test generator (3) by which to generate a set of potential test sequences (4). Since we are operating in a "gui-ripping" setting, we assume that the ripped information (obtained by some mixture of static and dynamic analysis of the SUT) is available in the form of an EFG (5 - see also Definition 1). Although the EFG itself is not essential to our approach, it can be helpful during state machine inference. An EFG-supported extension to the state merging algorithm in Algorithm 1 is provided in Appendix A.

B. Adaptive Test Selection with Input from Negative Inputs

The goal is to identify a manageable sub-set of test cases that will, reach the widest possible range of GUI functionalities. This is challenging because test cases can require a long time to execute. Although GUI Rippers may have the EFG graph, these can be of limited practical use because of their scale, and the fact that many paths through them are infeasible in practice. For example, the ripped EFG for the smallest of our case study systems (Rachota) contains 149 possible events (nodes) and 1344 edges connecting them.

We use state machine inference to address this problem by developing a test selection framework (6 in Figure 1). As with previous learning-based GUI testing approaches [12], [3], [4], our approach uses the inferred model to identify test cases. However, there are two key differences with our approach – one obvious, the other subtle. The obvious difference is that our approach explicitly incorporates negative information, by learning models that distinguish between failures to execute a test properly, and test executions that terminated without problems. The more subtle difference is that we frame our approach as a 'test selection' approach; we use the inferred model to filter an existing set of proposed test sequences, where the construction of these sequences is delegated to some external test-generation algorithm (for example, an existing EFG-based test generator [1]).

The details of our test selection process are presented in Algorithm 2. The approach takes three inputs: A set of candidate test cases T (e.g. as generated by some GUI testing framework), a number of iterations i representing the number of test-inference loops to be run, and j the number of tests to be generated per iteration (the choice of values for these parameters will be discussed after we briefly present the various steps in algorithm). The algorithm proceeds as follows:

Algorithm 2: Test Selection Algorithm

Input: A set of candidate test cases T, iterations i, number of tests per iteration j.
Result: A set $FinalTestSet \subseteq T$, where $|FinalTestSet| = i * j$

```
1  Select(T, i, j) begin
2      FinalTestSet ← ∅;
3      S+ ← ∅;
4      S− ← ∅;
5      for 1 to i do
6          Potential ← T \ FinalTestSet;
7          if S+ ∪ S− ≠ ∅ then
8              DFA ← inferDFA(S+, S−);
9              Potential ← Potential ∩ L(DFA);
10         Tests ← randomSelection(Potential, j);
11         FinalTestSet ← FinalTestSet ∪ Tests;
12         for t ∈ Tests do
13             e ← execute(t);
14             if e = t then
15                 S+ ← S+ ∪ {e};
16             else
17                 S− ← S− ∪ {e};
18     return FinalTestSet;
```

2-4 Before iterating, the set of test cases to be returned $FinalTestSet$ is initialised, along with the set of positive and negative test executions (S^+ and S^-).

5-6 For each iteration, we remove the set of tests executed so far $FinalTestSet$ from the pool of generated tests T, and store this as the set $Potential$.

7-8 If this is not the first iteration (i.e. $S^+ \cup S^- \neq \emptyset$), a DFA DFA is inferred from the sequences in S^+ and S^-, using the inference algorithm in Algorith 1.

9 The pool of potential tests $Potential$ is filtered by retaining only those tests that are accepted in the inferred DFA (i.e. belong to $L(DFA)$).

10-11 A random sub-set of size j is selected from $Potential$, is stored as a separate set $Tests$, and is added to $FinalTestSet$.

12-13 Each of the tests $t \in Tests$ is executed. The $execute$ function returns the sub-sequence of elements e in t that are successfully executed. In practice tests are executed by using whatever test execution mechanism is built into the GUI testing framework (e.g. the JFCReplayer in GUItar).

14-17 If t and e are identical, then the sequence e can be added to S^+. Otherwise it is added to S^-.

18 After the i iterations, the final test set $FinalTestSet$ is returned.

TABLE I
SUBJECT SYSTEMS

Name	Version	LOC	Windows	Events
Rachota	2.3	8,803	10	149
Buddi	3.4.0.8	9,588	11	185
JabRef	2.10b2	52,032	49	680
JEdit	5.1.0	55,006	20	457
DrJava	20130901-r5756	92,813	25	305

C. Implementation

The proof of concept tool was implemented as an approximately 1KLOC extension to the MINT EFSM inference tool[1] [36]. Our implementation is tailored to GUItar, but is implemented to be adaptable to alternative GUI testing frameworks. The test-inference loop happens through command-line invocations, and parsing of log-files, there are no API dependencies.n Although we use the EFG-driven compatibility function (see Appendix A), this could in principle be replaced with alternative state machine inference algorithms that do not require the EFG.

To obtain the initial large set of potential test sequences from which we are selecting tests, we generate all shortest paths from the EFM using the Floyd-Warshall algorithm [37]. In principle, any test-generation algorithm could be used at this point. It is however desirable that the base set of test cases exercises every event in the GUI, and this level of coverage is guaranteed by the Floyd-Warshall algorithm (accepting, of course, that many of the proposed tests will invariably not be feasible).

IV. PRELIMINARY EVALUATION

The goal of our technique is to identify test sets that are 'efficient'. By skipping tests that are infeasible, it should be possible to spend a greater proportion of the testing effort on meaningful tests that ultimately exercise the behaviour of the underlying program to a greater extent. To investigate whether this is indeed true, we pose the following research questions:

RQ1 Does our approach enable longer sequences of GUI interactions?

RQ2 Does our approach cover the underlying source code to a greater extent?

RQ3 What is the time overhead incurred by our approach?

A. Subject Systems

To evaluate our approach, we chose five GUI-based Java applications, shown in Table I. We selected these applications on the basis that they were used by Gao *et al.* [8] for their GUI testing paper. The exact versions (along with accompanying versions of GUItar) were made available by Gao *et al.* online[2]. `Rachota` is a time-tracking tool that can produce time-management reports. `Buddi` is a financial budget management tool. `JabRef` is a bibliography reference manager. `JEdit` is an extensible text editor, and `DrJava` is an educational Java IDE.

[1]https://github.com/neilwalkinshaw/mintframework

[2]http://cse.unl.edu/myra/artifacts/Repeatability/

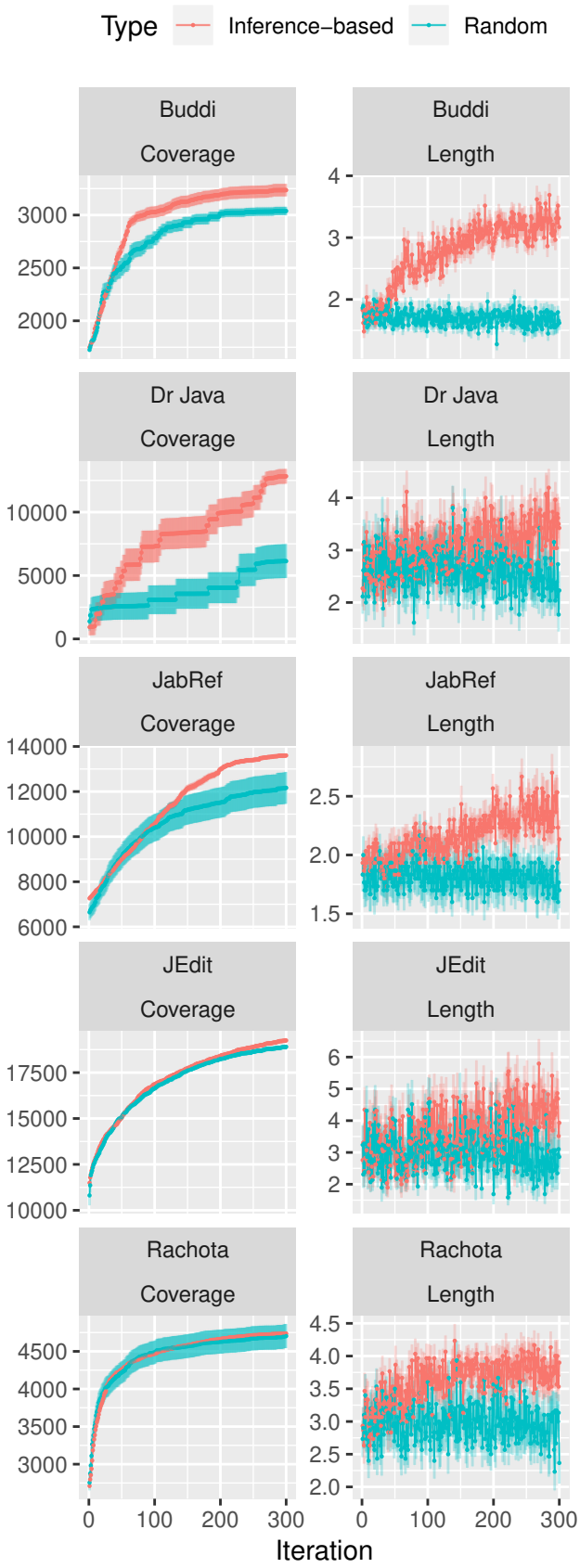

Fig. 2. Code branches covered and test sequence lengths and per iteration.

TABLE II
FINAL SEQUENCE LENGTHS, COVERAGE, INPUTS EXECUTED AND TIME TAKEN AFTER 300 ITERATIONS.

System	Mean sequence length			Mean coverage			Mean inputs executed			Mean time (minutes)		
	Inf.	Rnd.	p	Inf.	Rnd.	p	Inf.	Rnd.	p	Inf.	Rnd.	p
Rachota	3.9	2.37	**0.0006**	4,736.63	4,703.30	**0.0002**	1082.67	887.87	**<0.0001**	211.12	94.9	**<0.0001**
Buddi	3.17	1.62	**<0.0001**	3,236.06	3,038.52	**0.0399**	830.38	507.93	**<0.0001**	178.52	90.87	**<0.0001**
JabRef	2.13	1.7	**0.0199**	4,736.63	4,703.30	**0.0002**	649.9	543.37	**<0.0001**	629.77	155.08	**<0.0001**
JEdit	3.93	2.86	0.1547	19,250.93	18,906.14	**0.013**	1101.28	890.73	**<0.0001**	535.09	227.71	**<0.0001**
DrJava	3.42	2.23	**0.027**	12,827.73	6,136.92	**0.0002**	929.08	780.85	**<0.0001**	304.59	136.07	**<0.0001**
All	3.062	2.408		8841.57	7741.035		917.72	720.83		373.89	140.81	

B. Methodology

For each of the systems in Table I we ran our tool for 300 iterations, producing 5 tests per iteration (i.e. we generated a total of 1500 tests per run). Since our approach involves some random sampling (tests are picked at random from the large pool of tests), we repeat each run 30 times with different random seeds.

As our baseline, we randomly pick the same number of test cases from the pool of candidate test cases that collectively cover the vertices of the EFG (as described in Section III-C). This amounts to a generic coverage-driven GUI testing technique.

To answer RQ1, we record for each individual test execution the number of separate GUI events successfully executed and the length of each test sequence. To answer RQ2, we record the code coverage, using the Cobertura extensions of the GUItar framework. To answer RQ3 we record the number of milliseconds taken for each iteration.

To measure statistical significance in our comparisons of length and coverage, we use the Mann-Whitney U-Test to compare the lengths and coverage respectively at the final (300th) iteration. This statistical test was chosen because a Shapiro-Wilks test indicated that the data is not normally distributed. We report a statistically significant difference if $p < 0.05$.

The experiments were run in parallel on the ALICE HPC facility at the University of Leicester. GUI interactions were executed with the `xvbf` virtual frame buffer. To guard against any side-effects from previous tests affecting subsequent tests, a core copy of the program was copied on to the test node for every experiment. The subject systems and test harness were run using Oracle JDK 8.

C. Results

The mean results after 300 iterations and the p-values for the statistical significance of the U-Tests are shown in Table II. The per-iteration means (and standard deviation error-bars) for sequence-length and coverage are shown in Figure 2. The final times and total number of interactions executed are shown in Figure 3.

RQ1: Length of GUI interactions: Figure 2 indicates that, for each system, the inference-based tests achieve longer GUI interactions than those that are selected at random. When the test runs from all the systems are taken together, the average sequence length achieved from the random selections at the final iteration is 2.41, versus 3.06 for inference-based testing. For all systems apart from JEdit the difference in sequence length is statistically significant. In the case of Buddi the difference is especially pronounced, with the inference-based tests leading to a mean sequence length of 3.17 against a mean of 1.62 for the randomly-selected tests.

RQ2: Code coverage: Table II shows that, after 300 iterations, the mean coverage achieved with the help of inference is (statistically) significantly higher across all subject systems than that achieved from random test selection. The extent of this improvement, as in RQ1, differs significantly between systems. Figure 2 shows that whereas the difference is substantial for Buddi, Dr Java, and JabRef, it is hard to distinguish visually for Rachota, and only in the latter 50-70 iterations for JEdit.

RQ3: Time: Table II shows that the inference-based approach took significantly longer than the random approach for 300 iterations. The average times have to be interpreted with caution because they vary significantly for each system. This is illustrated in Figure 3, which plots the time taken against the number of events executed.

On average, the time difference between inference-driven and random test execution is 233.08 minutes. Over 300 iterations (at five test executions each) this amounts to a 42 second difference per iteration. Although the execution time of longer valid test sequences will be a factor, it is likely that the majority of this time is spent inferring the state machine.

It should be noted that the 300 iteration cut-off is an arbitrary limit. Looking at the sequence-length and code-coverage time-series in Figure 2, significant improvements over random testing are already evident between 100 and 150 iterations, in which case the time-overhead would be significantly lower.

Summary

The findings are promising. The inclusion of inference supported by negative results leads to longer sequences, which probe aspects of software behaviour that are not reached by random executions. As a result this leads to higher code-coverage. There is a time-overhead involved, largely because of the need to run a Machine Learner at every iteration.

D. Threats to Validity

a) External threats to validity: The baseline used in our experiments is a test set generated by coverage-guided GUI Ripping. Although we have convincingly shown that the use of

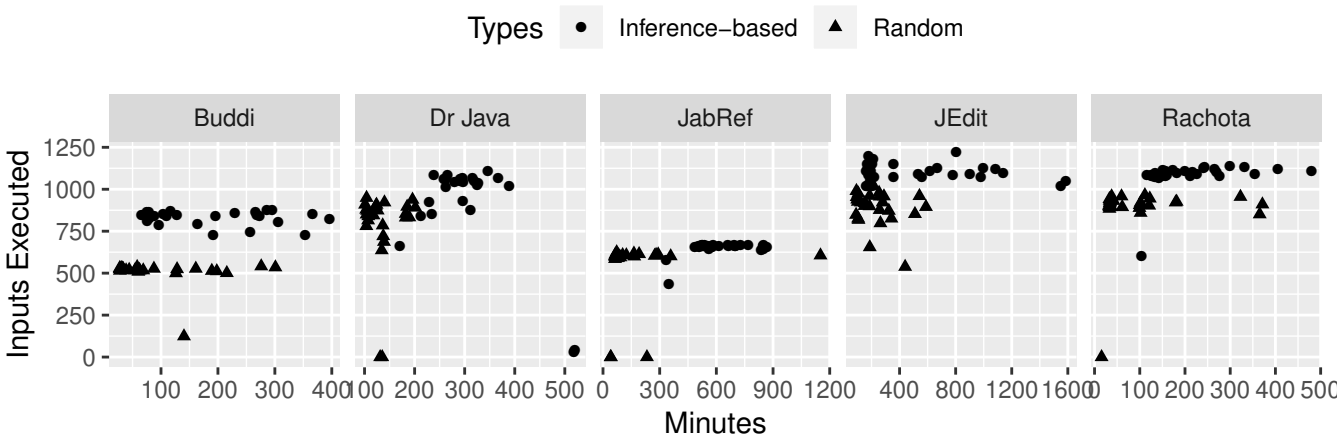

Fig. 3. Times taken for 300 iterations, versus total number of inputs executed.

inference (with negative examples) produces better results, this does not demonstrate that the use of negative examples produces better results than conventional (non-negative example)-based inference approachces such as SwiftHand or FSMDroid. This will require a separate controlled experiment, and is part of our plans for future work.

The experiments are based on five Java (Swing / AWT) programs, and cannot be taken to represent, for example, the performance obtained with respect to mobile devices. However, as far as desktop GUI applications are concerned, they are all diverse in terms of their domain and size, and have all been used in previous studies on GUI testing.

b) Internal threats to validity: We use statement coverage to gauge the extent to which the behaviour of the underlying source code has been executed. This is notoriously imprecise at estimating test adequacy; test set can achieve a high level of statement coverage but still miss out on many aspects of program behaviour. Since we are more interested in using code coverage as a *relative* measure as opposed to an absolute one, we would nevertheless argue that it is reasonable to presume that a test set that achieves statement coverage that is higher than another test set is exercising a greater range of behaviour.

V. Conclusions and Future Work

In this paper we have introduced a technique whereby state machine learners can be incorporated into an automated testing cycle to increase the likelihood of selecting valid, longer test sequences. We have demonstrated a proof-of-concept implementation, and have successfully applied it to a selection of Java Swing / AWT programs.

Our work has specifically considered the "GUI-ripping" setting, but can be adapted to other settings. model inference has been used successfully in "active" Android testing settings [3], [4], for example. These also offer sources of negative information that can be easily used to improve and refine models, and the test sets that are derived from them as a result.

References

[1] A. M. Memon, "An event-flow model of gui-based applications for testing," *Software testing, verification and reliability*, vol. 17, no. 3, pp. 137–157, 2007.

[2] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 94–105.

[3] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in *Acm Sigplan Notices*, vol. 48, no. 10. ACM, 2013, pp. 623–640.

[4] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 245–256.

[5] K. J. Lang, B. A. Pearlmutter, and R. A. Price, "Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm," in *Proceedings of the 4th International Colloquium on Grammatical Inference*, V. Honavar and G. Slutzki, Eds., vol. 1433. Springer-Verlag, 1998, pp. 1–12.

[6] A. Memon, I. Banerjee, and A. Nagarajan, "What test oracle should i use for effective gui testing?" in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 164–173.

[7] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "Mobiguitar: Automated model-based testing of mobile apps," *IEEE software*, vol. 32, no. 5, pp. 53–59, 2014.

[8] Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang, "Making system user interactive tests repeatable: When and what should we control?" in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1. IEEE, 2015, pp. 55–65.

[9] S. Huang, M. B. Cohen, and A. M. Memon, "Repairing gui test suites using a genetic algorithm," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*. IEEE, 2010, pp. 245–254.

[10] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?(e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 429–440.

[11] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM, 2014, pp. 204–217.

[12] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro, "Autoblacktest: Automatic black-box testing of interactive applications," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 81–90.

[13] T. E. Vos, P. M. Kruse, N. Condori-Fernández, S. Bauersfeld, and J. Wegener, "Testar: Tool support for test automation at the user

interface level," *International Journal of Information System Modeling and Design (IJISMD)*, vol. 6, no. 3, pp. 46–83, 2015.
[14] A. I. Esparcia-Alcázar, F. Almenar, T. E. Vos, and U. Rueda, "Using genetic programming to evolve action selection rules in traversal-based automated software testing: results obtained with the testar tool," *Memetic Computing*, vol. 10, no. 3, pp. 257–265, 2018.
[15] J. Hopcroft, R. Motwani, and J. Ullman, *Introduction to Automata Theory, Languages, and Computation, Third Edition*. Addison-Wesley, 2007.
[16] D. Angluin, "On the complexity of minimum inference of regular sets," *Information and Control*, vol. 39, pp. 337–350, 1978.
[17] L. Valiant, "A theory of the learnable," *Communications of the ACM*, vol. 27, no. 11, pp. 1134–1142, 1984.
[18] D. Angluin, "Learning Regular Sets from Queries and Counterexamples," *Information and Computation*, vol. 75, pp. 87–106, 1987.
[19] J. Oncina and P. Garcia, "Inferring regular languages in polynomial updated time," in *Pattern recognition and image analysis: selected papers from the IVth Spanish Symposium*. World Scientific, 1992, pp. 49–61.
[20] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *POPL 2002*, Portland, Oregon, Jan. 16–18, 2002, pp. 4–16.
[21] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behaviour," *IEEE Transactions on Computers*, vol. C, no. 21, pp. 592–597, 1972.
[22] A. W. Biermann and R. Krishnaswamy, "Constructing programs from example computations," *IEEE Transactions on Software Engineering*, no. 3, pp. 141–153, 1976.
[23] J. Cook and A. Wolf, "Discovering models of software processes from event-based data," *ACM Transactions on Software Engineering and Methodology*, vol. 7, no. 3, pp. 215–249, Jul. 1998.
[24] C. Damas, B. Lambeau, P. Dupont, and A. van Lamsweerde, "Generating annotated behavior models from end-user scenarios," *IEEE Transactions on Software Engineering*, vol. 31, no. 12, 2005.
[25] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun, "Classification of software behaviors for failure detection: a discriminative pattern mining approach," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2009, pp. 557–566.
[26] D. Lo and S. Maoz, "Scenario-based and value-based specification mining: better together," *Automated Software Engineering*, vol. 19, no. 4, pp. 423–458, 2012.
[27] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *ACM/IEEE 30th International Conference on Software Engineering, 2008. (ICSE'08)*. ACM, 2008, pp. 501–510.
[28] S. Reiss and M. Renieris, "Encoding program executions," in *ICSE*. IEEE Computer Society, 2001, pp. 221–230.
[29] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin, "Reverse engineering state machines by interactive grammar inference," in *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*. IEEE, 2007, pp. 209–218.
[30] N. Walkinshaw, B. Lambeau, C. Damas, K. Bogdanov, and P. Dupont, "STAMINA: a competition to encourage the development and assessment of software model inference techniques," *Empirical Software Engineering*, pp. 1–34, 2012.
[31] N. Walkinshaw, K. Bogdanov, J. Derrick, and J. Paris, "Increasing functional coverage by inductive testing: A case study," in *Testing Software and Systems (ICTSS'10)*, 2010, pp. 126–141.
[32] R. K. Shehady and D. P. Siewiorek, "A method to automate user interface testing using variable finite state machines," in *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*. IEEE, 1997, pp. 80–88.
[33] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998, vol. 1, no. 1.
[34] G. Bae, G. Rothermel, and D.-H. Bae, "Comparing model-based and dynamic event-extraction based gui testing techniques: An empirical study," *Journal of Systems and Software*, vol. 97, pp. 15–46, 2014.
[35] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*. Wiley, 1999.
[36] N. Walkinshaw, R. Taylor, and J. Derrick, "Inferring extended finite state machine models from software executions," *Empirical Software Engineering*, vol. 21, no. 3, pp. 811–853, 2016.
[37] R. W. Floyd, "Algorithm 97: Shortest path," *Communications of the ACM*, vol. 5, p. 345, 1962.

APPENDIX A
USING THE EFG FOR STATE MACHINE INFERENCE

It is in our interest that the inferred DFA is as precise as possible. It should generalise upon the set of traces in S^+, but not *over*-generalise to the point that it accepts too many sequences that are infeasible. Existing state machine inference-based GUI-testing approaches such as SwiftHand [3] take advantage of the run-time GUI state. When deciding whether a pair of states can be merged (i.e. as part of the `ChoosePair` function in Algorithm 1), they take advantage of the ability to compare which events are possible at any given point; if different events are possible, the states are not behaviourally equivalent and should not be merged. In our setting, we do not presume access to the run-time state. However, if we have access to the EFG, it is possible increase the efficiency and accuracy of this process by in a similar manner to the use of the live test-information used by tools such as SwiftHand.

Algorithm 3: EFG-supported compatibility function

Input: A DFA D and an EFG E.
Result: A boolean.

```
1  ChoosePair(E, D) begin
2      merge ← false;
3      while (B_i, B_j) ← ChoosePair(Q_D) ∧ ¬merge do
4          Events ← in(D, B_i)∪in(D, B_j);
5          Dest_D ←out(D, B_i)∪out(D, B_j);
6          Dest_E ← ∅;
7          for e ∈ Events do
8              Dest_E ← Dest_E∪ out(E, e);
9          if Dest_D ⊆ Dest_E then
10             merge ← true;
11     return (B_i, B_j);
```

Algorithm 3 shows a version of the `ChoosePair` function that can be used as a wrapper for the original `ChoosePair` function in Algorithm 1. For every pair of states considered (line 3), it identifies the set of GUI events $Events$ (vertices in the EFG) that would need to be considered by identifying the set of incoming events to each candidate state in the DFA (denoted by function `in`, line 4). It then predicts the set of all events that should be possible from the merged state in the DFA by taking the union of the outgoing transition events from both candidate states (line 5). It constructs a corresponding union of events are possible according to the EFG by taking the union of events possible after every event $e \in Events$ (lines 7-8). If the set of events possible from the merged state is a subset of the set of events in the EFG (line 9), then the merge is allowed (line 10), otherwise the process is repeated for some alternative pair.