EXSYST: Search-Based GUI Testing

Florian Gross
Saarland University
Saarbrücken, Germany
fgross@cs.uni-saarland.de

Gordon Fraser
Saarland University
Saarbrücken, Germany
fraser@cs.uni-saarland.de

Andreas Zeller
Saarland University
Saarbrücken, Germany
zeller@cs.uni-saarland.de

***Abstract*—Test generation tools commonly aim to cover structural artefacts of software, such as either the source code or the user interface. However, focusing only on source code can lead to unrealistic or irrelevant test cases, while only exploring a user interface often misses much of the underlying program behavior. Our EXSYST prototype takes a new approach by exploring user interfaces while aiming to maximize code coverage, thus combining the best of both worlds. Experiments show that such an approach can achieve high code coverage matching and exceeding the code coverage of traditional unit-based test generators; yet, by construction every test case is realistic and relevant, and every detected failure can be shown to be caused by a real sequence of input events.**

***Keywords*-test case generation; system testing; GUI testing; test coverage**

I. Introduction

State of the art test generation tools can produce unit tests that achieve high code coverage, for example by using search [5] or constraint solvers [3]. However, such tools have a number of shortcomings that limit their widespread use. First, these tools produce executions and not test cases, as they are missing the essential test oracles that decide, whether an execution revealed a defect. The second problem is that generated test cases may be *nonsensical*—that is, represent executions that would never occur in real usage of the application. The reason for this is that often assumptions are not made explicit by the programmer. Even though this could be amended, for example in terms of preconditions, it is unlikely that programmers will protect their code against all possible automatic misuse. Such nonsensical executions not only add to the difficulty of the oracle problem, but may reveal *false failures* that cannot happen in real usage. In a recent study [4] on five applications, we found that *all* of the 181 failing generated unit tests were false failures—that is, failures which were created through violations of implicit preconditions, but that never occur in the actual application.

To address the issue of false failures, our EXSYST prototype leverages *system interfaces* such as GUIs as filters against nonsensical input—interfaces at which the program must cope with every conceivable input, and at which every failure is a true failure. While GUI testing has been around for a long time (typically with a focus of covering all GUI elements and states), EXSYST specifically aims for achieving the same high *code coverage* as unit test generators. For this purpose, it applies a *search-based* approach; that is, it systematically generates user interface events while *observing* which events correspond to which behavior in the code.

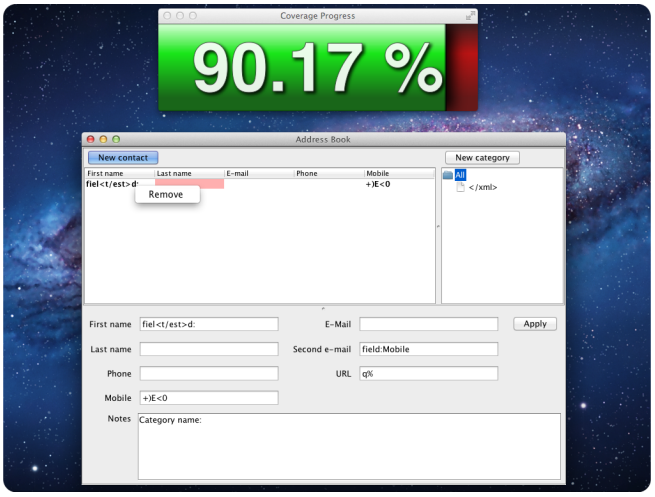

Figure 1. Screenshot of EXSYST in action, showing the current state of execution and the achieved branch coverage for the target application.

II. Search-based System Testing

EXSYST uses a genetic algorithm to evolve a population of GUI test suites with the goal of achieving maximum possible coverage. To achieve this, it combines the genetic algorithms from our EvoSuite test generation tool [2] with a new problem representation for handling GUI interactions.

A genetic algorithm is a meta-heuristic search technique which tries to imitate the natural process of evolution. A population of candidate solutions is evolved using search operators such as selection, crossover, and mutation, gradually improving the fitness value of the individuals, until a solution has been found or the search is stopped by other means. In EXSYST, an individual of this search is a set of GUI interaction sequences (test suites). Crossover creates offspring test suites by exchanging sequences between the two parent test suites. Mutation of test suites results in mutation of individual sequences with a probability related to the number of sequences in a test suite: Mutations of interaction sequences add, remove, or change individual interactions.

The individuals of the initial population are generated from targeted random walks on the GUI. The fitness of a test suite is measured by branch coverage on the underlying code, and is calculated as the sum of normalized branch distances [2], where the branch distance is an estimate of how close the predicate guarding a particular branch was to being fulfilled and thus how close the branch was to being covered.

To guide the exploration and the search operators, a model of the user interface is created and evolved alongside the test cases. This UI model represents our knowledge of the behavior of the application under test. The information is modeled using a non-deterministic state machine that we create from observing actual executions of the application under test. By construction, this model only describes a subset of possible application behavior, namely behavior we have already observed.

The states of this model represent components and actions available at a point of time in application interaction (i.e., windows that are visible, enabled, and not obscured by modality), as well as all of their respective interactable components (i.e., visible, enabled). Transitions are defined to represent the execution of actions, such as entering text into text fields, clicking buttons, opening menus, etc. An *action sequence* is a sequence of such actions, corresponding to a path through the automaton. If the action has not been executed so far, its transition leads to the unknown state $s_?$. As we generate new test cases, the model is updated with information from new observations of the application behavior. If an action takes us to a new state, the new state is added to the model, and a transition for the action, taking us from the old state to the new state will be added. If there previously was a transition to the unknown state, then it is removed.

The UI model serves several purposes:

- When inserting new interactions into an interaction sequence, we can give preference to unexplored parts of the GUI.
- Mutating an interaction sequence might result in an invalid sequence that cannot be executed. If this is the case, then we use the UI model to repair the sequence.

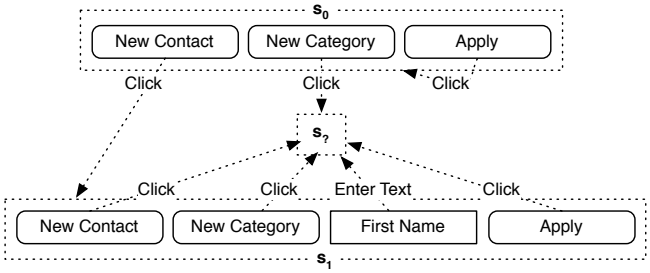

Figure 2. Example UI model: States represent active GUI elements, and transitions represent interactions; interactions not observed so far lead to a special unknown state $s_?$.

III. System Testing with Exsyst

EXSYST is a system test generator for interactive Java programs, that controls the program under test through its graphical user interface. It does so by synthesizing input actions, such as entering text or mouse clicks. The distinguishing feature of EXSYST is that it aims to *maximize coverage:* via search-based techniques, it strives to generate input sequences such that as much of the code of the program under test as possible is covered.

All it takes to use EXSYST is a computer with CPU cycles to burn. After invoking EXSYST with the program under test, it autonomously generates input sequences in an invisible virtual GUI, reporting any application failures as encountered. (For diagnostic and demonstration purposes, the GUI interaction can also be made visible.) Since every failure is tied to a real sequence of input events, every failure is real—it is characterized by a small number of user interactions that are easy to understand and to reproduce.

While EXSYST is easy to use and straight-forward to understand, it shares the same limitation as any other test generator: While it strives to *exercise* as many program aspects as possible, it cannot *check* the outcome of these interactions—unless the program or the runtime system detects an error itself (for instance, by raising an exception). EXSYST users would thus be well advised to include *assertions* in their code which check for the sanity of computing results. In the long run, such a setting implies a dramatic reduction in writing test cases: A tool like EXSYST generates executions and strives for coverage, while the provided assertions check for correctness of the results (and can be reused again and again). With EXSYST, any failure reported is (by construction) a true failure; the lack of false alarms finally makes automated test generation an effective and efficient alternative to manual test writing.

IV. Some Failures

Let us briefly discuss some results obtained with EXSYST. We ran EXSYST on five test subjects commonly used for GUI testing, the details of which are listed in Table I. On each of the five test subjects, we would run the respective tool for 15 minutes[1]. For these experiments, Table I lists:

1) the number of tests executed,
2) the number of failures encountered,
3) the *instruction coverage* achieved.

Applying EXSYST on the five study subjects listed in Table I, EXSYST generated a total of 6,373 tests (Table II), out of which 248 failed. These failures were caused by a total of six errors, all of which are true failures and can be recreated through a short sequence of user events:

[1]All times measured on a server machine with 8x 2.93 GHz Intel i7-870 Lynnfield CPU cores, and 16 GB RAM, running Linux kernel 2.6.38-11. As all approaches are driven by random generators, results reported are averages over three runs conducted.

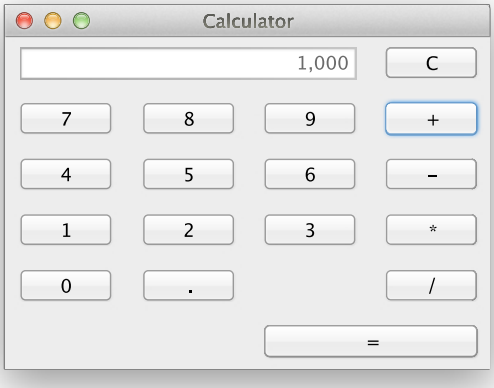

Figure 3. The *Calculator* application formats numbers with thousands separators, but can not parse such numbers produced by itself.

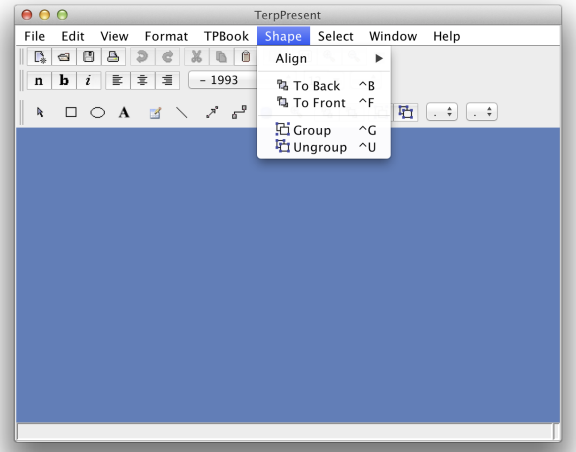

Figure 4. The *TerpPresent* application in an inconsistent state: No document windows are open, yet functionality to interact with documents remains enabled.

1) **Addressbook.** When no contact is selected in the address book, the application disables all the input fields in the bottom pane. However, it does not disable the "*Apply*" button. Pressing this button (for instance, right after application start) caused an uncaught `NullPointerException`.
2) **Calculator.** The *Calculator* application (Figure 3) raises a `NumberFormatExceptions` when applying any of its numerical operations to an intermediate result consisting of more than three digits. This is due to *Calculator* application using the `NumberFormat` of the English locale for formatting numbers (which uses thousands separators), and feeding these formatted strings into the `BigDecimal()` constructor, which does not support thousands separators. One input sequence leading to this problem is "*500 * 2 + 1 =*"; Figure 3 shows the application state before the failure. The application subsequently becomes unusable until it is returned to a sane state by pressing the "*Clear*" button.
3) **TerpPresent.** *TerpPresent,* a simple presentation program (Figure 4), uses a multi-document interface with a shared menu at the top. When the last presentation (in the blue area) is closed, menu entries pertaining to the document erroneously remain enabled. For instance, invoking *Shape/Group* without an open document results in an application failure (Uncaught `NullPointerException`).
4) **TerpPresent again.** A similar failure is related to object selection. By using "*Select/Invert*" twice when no shape was selected before, we arrive in an inconsistent application state: When we now use "*Edit/Delete*", the application assumes there still is a shape selected when no shape remains. "*Edit/Copy*" then leads to an uncaught `NullPointerException`. One input sequence demonstrating this problem is to create a new shape of some kind and then invoke "*Select/Invert, Select/Invert, Edit/Delete, Edit/Copy*".
5) and 6) **More failures.** EXSYST also was able to detect two issues related to opening and saving files with unspecified format and/or extension.

Table I
STUDY SUBJECTS

Name	Source	#Lines	#Classes
Addressbook	[6]	1,334	41
Calculator	[6]	409	17
TerpPresent	[7]	54,394	361
TerpSpreadSheet	[7]	20,130	161
TerpWord	[7]	10,149	58

Table II
TESTS GENERATED

Subject	Tests	Failures	Coverage
Addressbook	2,682	127	87.7%
Calculator	2,852	42	92.0%
TerpPresent	117	9	25.3%
TerpSpreadSheet	273	29	48.5%
TerpWord	449	41	53.9%
Total	6,373	248	61.5%

All these six issues are real failures which need to be fixed in the application code. In principle, all of them could also be discovered by a unit-testing tool like Randoop—but as we showed in our evaluation, they would likely get lost between hundreds of false alarms [4].

While testing through the GUI gets rid of false failures, would a search-based approach as in EXSYST still be able to achieve high coverage? The rightmost column of Table II lists the coverage achieved—on average, 61.5%, a significantly higher coverage than the 40.9% of Randoop [4]. Using EXSYST, one can thus expect to obtain the same high coverage as unit-based tests; yet, every failure is a true failure characterized by a short sequence of input events.

V. Related Work

Recent code-based techniques such as random testing [9], dynamic symbolic execution [3], or search-based testing [5] can achieve high code coverage, yet suffer from problems of nonsensical tests and false failures. EXSYST overcomes this problem by testing through the user interface, rather than at the API level. This, of course, requires the availability of a suitable user interface. Usually, automated techniques to derive test cases for graphical user interfaces (GUIs) first derive graph models that approximate the possible sequences of events of the GUI [8], and then use such models to derive representative test sets [10]. Mostly, these GUI test generation approaches consider *only* the GUI, thus allowing no direct conclusions about the relation of GUI tests and the tested program code. A notable exception is the work of Bauersfeld et al. [1], who attempt to link GUI tests with the code by optimizing individual test cases to achieve a maximum size of the call tree. In contrast, EXSYST explicitly tries to maximize the code coverage, while at the same time it aims to produce small test suites to reduce the oracle problem.

VI. Conclusions

Automated testing has made spectacular progress in the past decade—progress that makes its remaining shortcomings all the more painful. By testing through the user interface, EXSYST avoids false failures as they could be provoked by generated unit tests; at the same time, it achieves the same high coverage as these tools. EXSYST requires nothing but spare computer cycles and is straight-forward to use; Failures are reported in terms of user interaction and thus easy to understand. We recommend search-based system testing as a strong complement to generated unit tests.

We are currently integrating EXSYST into the EvoSuite test generation framework. Live demos of EXSYST and EvoSuite as well as additional data on the experiments described in this paper will be made available at

`http://www.exsyst.org/`

and

`http://www.evosuite.org/`

Acknowledgments

The work presented in this presentedaper was performed in the context of the Software-Cluster project *EMERGENT* (www.software-cluster.org). It was funded by the German Federal Ministry of Education and Research (BMBF) under grant no. “01IC10S01” and by an ERC Advanced Grant “Specification Mining and Testing”. EXSYST is additionally funded by the Google Focused Research Award “Test Amplification”. The authors assume responsibility for the content.

References

[1] S. Bauersfeld, S. Wappler, and J. Wegener. A metaheuristic approach to test sequence generation for applications with a GUI. In M. Cohen and M. O Cinneide, editors, *Search Based Software Engineering*, volume 6956 of *Lecture Notes in Computer Science*, pages 173–187. Springer Berlin / Heidelberg, 2011.

[2] G. Fraser and A. Arcuri. Evolutionary generation of whole test suites. In *International Conference On Quality Software (QSIC)*, pages 31–40, Los Alamitos, CA, USA, 2011. IEEE Computer Society.

[3] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, New York, NY, USA, 2005. ACM.

[4] F. Gross, G. Fraser, and A. Zeller. Exploring realistic program behavior. Technical report, Saarland University, 2012. Submitted for publication.

[5] P. McMinn. Search-based software test data generation: a survey: Research articles. *Software Testing Verification Reliability*, 14(2):105–156, 2004.

[6] R. Medina and P. Pratmarty. UISpec4J — Java/Swing GUI testing library. http://www.uispec4j.org/.

[7] A. Memon, I. Banerjee, N. Hashmi, and A. Nagarajan. Dart: A framework for regression testing "nightly/daily builds" of gui applications. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, pages 410–, Washington, DC, USA, 2003. IEEE Computer Society.

[8] A. M. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of The 10th Working Conference on Reverse Engineering*, Nov. 2003.

[9] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, New York, NY, USA, 2007. ACM.

[10] X. Yuan and A. M. Memon. Using GUI run-time state as feedback to generate test cases. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 396–405, Washington, DC, USA, 2007. IEEE Computer Society.