Search-Based System Testing: High Coverage, No False Alarms

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

Modern test case generation techniques can automatically achieve high code coverage. If they operate on the unit level, they run the risk of generating inputs infeasible in reality, which, when causing failures, are painful to identify and eliminate. Running a unit test generator on five open source Java programs, we found that all of the 181 reported failures were *false failures*—that is, indicating a problem in the generated test case rather than the program. By generating test cases at the GUI level, our EXSYST prototype can avoid such false alarms by construction. In our evaluation, it achieves higher coverage than search-based test generators at the unit level; yet, every failure can be shown to be caused by a real sequence of input events. Whenever a system interface is available, we recommend considering search-based system testing as an alternative to avoid false failures.

Categories and Subject Descriptors

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

General Terms

Algorithms, Experimentation

Keywords

Test case generation; system testing; GUI testing; test coverage

1. INTRODUCTION

In the past years, the field of *test case generation* has made tremendous progress. Techniques such as random testing [33], dynamic symbolic execution [15], or search-based testing [24] are now able to generate executions that easily achieve high coverage at the unit test level. Still, test case generation tools have a number of shortcomings that limit their widespread use. First and foremost is the *oracle problem:* Test case generators normally do not generate test cases, but only *executions;* the *oracle* which assesses the test result is missing. Unless the program has good run-time checks, failures may thus go unnoticed; recent work addresses this issue by suggesting oracles together with test cases [11, 13].

ISSTA '12, July 15–20, 2012, Minneapolis, MN, USA

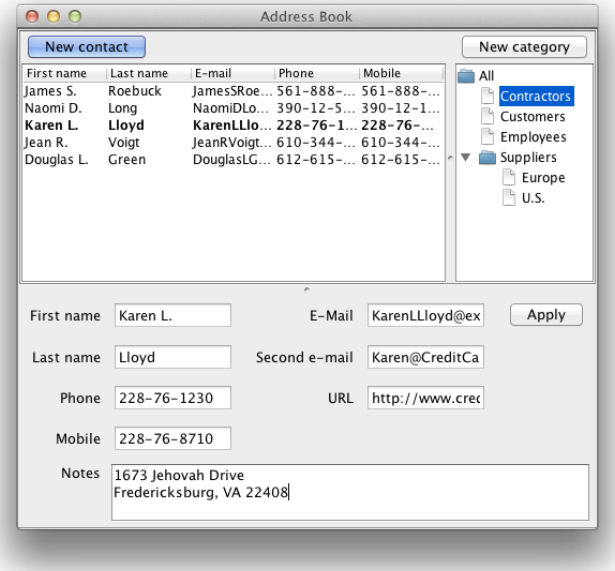


Figure 1: The *Addressbook* application for which unit test generators create infeasible failing tests.

The second problem is that generated test cases may be *infeasible*—that is, represent an execution that would never occur in reality. As an example, consider the *Addressbook* program [26] shown in Figure 1. It is a simple Java application which manages a set of contacts that can be entered, searched, and grouped into categories. Let us assume we want to use test case generation to test the main `AddressBook` class; for this purpose, we use the *Randoop* random test case generator, a tool which randomly combines method calls to cover large portions of code.

Figure 2 shows one of those tests generated by Randoop. It consists of 88 individual calls, with the last one raising an exception—that is, the test fails and must now be analyzed by the developer to understand the failure cause. Using delta debugging and event slicing [5], we can automatically reduce this test to five lines listed in Figure 3. We now can see how the test fails: It creates two `AddressBook` objects as well as a category in each; however, the category for the second address book is based on the category from the first address book. This mix-up of address books is not handled by the code and raises an exception.

The reason why this test code is infeasible is that the actual application only has *one* address book, and consequently, only one `AddressBook` object. The *Randoop* test case is infeasible because it violates this implicit assumption—it is like creating a car with two steering wheels and finding that this causes conflicts.

```
public class RandoopTest0 extends TestCase {
    public void test8() throws Throwable {
      AddressBook var0 = new AddressBook();
      EventHandler var1 = var0.getEventHandler();
      Category var2 = var0.getRootCategory();
      Contact var3 = new Contact();
      ⟨. . . 75 more calls. . . ⟩
      AddressBook var79 = new AddressBook();
      EventHandler var80 = var79.getEventHandler();
      Category var81 = var79.getRootCategory();
      String var82 = var81.getName();
      var77.categorySelected(var81);
      Category var85 = var65.createCategory(var81,
          "hi!");
      String var86 = var85.toString();
      Category var88 = var0.createCategory(var85, . . .);
          // raises NameAlreadyInUseException
    }
}
```

Figure 2: A failing test case for the `AddressBook` class

To avoid infeasible tests, the programmer could make assumptions explicit—in our example, she could make `AddressBook` a singleton class, for instance, or encode preconditions that would be checked at runtime. One could also claim that the test case simply represents some possible future use, which should be accounted for. However, it is unlikely that programmers would anticipate and enumerate all possible misuses and protect their code against them.

Given how these test cases are generated, the existence of infeasible tests is not a surprise. However, infeasible failing tests are widespread. In an exploratory study of five common study subjects, we found *100% of failing Randoop tests to be false failures*—one has to suffer through up to 112 infeasible failing test cases and not find a single true failure.[1] Such false positives effectively prohibit the widespread use of test generation tools.

In this paper, we explore how to leverage *system interfaces* such as GUIs as filters against infeasible executions. As system input is controlled by third parties, the program must cope with every conceivable input. If the program fails, it always is the program's fault: At the system level, every failing test is a true positive. The system interface thus acts as a *precondition* for the system behavior.

Test case generation at the system level and through GUIs has been around for a long time. The issue of such approaches is the large code distance between the system interface and the behavior to be covered; intuitively, it appears much easier to call a method directly rather than trying to achieve its coverage through, say, a user interface; on top of that, interfaces may be governed by several layers of complex third-party code for which not even source code may be available.

We overcome these problems by applying a *search-based* approach; that is, we systematically generate user interface events while *learning* which events correspond to which behavior in the code. On five programs commonly used to evaluate GUI testing approaches, our EXSYST prototype[2] achieves the same high code coverage as unit-based test generators and the same high GUI code coverage as GUI testers. Yet, in our evaluation, it finds more bugs than typical representatives of these approaches, and by construction, has no false alarms whatsoever. Finally, it is future-proof: Should the *Addressbook* GUI ever evolve to support multiple address books, EXSYST would generate tests that cover this situation—but only then.

[1] For the full numbers, see Table 2 in Section 2.

[2] EXSYST = EXplorative SYStem Testing

Table 1: Study Subjects

Name	Source	#Lines	#Classes
Addressbook	[26]	1,334	41
Calculator	[26]	409	17
TerpPresent	[27]	54,394	361
TerpSpreadSheet	[27]	20,130	161
TerpWord	[27]	10,149	58

```
public class RandoopTest0 extends TestCase {
    public void test8() throws Throwable {
      AddressBook a1 = new AddressBook();
      AddressBook a2 = new AddressBook();

      Category a1c = a1.createCategory(
          a1.getRootCategory(), "a1c");
      Category a2c = a2.createCategory(a1c, "a2c");
          // raises NameAlreadyInUseException
    }
}
```

Figure 3: Simplified `AddressBook` test case from Figure 2

This paper makes the following contributions:

1. We explore and quantify the problem of false failures caused by infeasible tests (Section 2);
2. We use a new testing tool, EXSYST, to demonstrate coverage-driven generation of test suites through a GUI (Section 3);
3. We introduce the general concept behind EXSYST, namely *search-based system testing,* which uses system test generation to maximize test quality based on coverage metrics, while avoiding false failures by construction (Section 4); and
4. We show that our approach achieves *better coverage* than state-of-the-art unit or GUI test generators, as evaluated on a standard suite of five Java programs (Section 5). This is
   - a *surprising result,* as one might assume it would be easier to achieve high coverage when invoking methods directly rather than going through the GUI, and
   - our *most important message,* as it establishes system testing as a viable alternative to unit-level testing.

In the remainder of the paper, Section 6 discusses the related work in unit test generation and GUI testing. Section 7 closes with conclusion and consequences, as well as an outlook on future work and alternate applications.

2. FALSE FAILURES

In the introduction, we have discussed the problem of infeasible tests—method sequences that violate some implicit assumption, which, however, is never violated in the actual application context. Is this a true problem, and how widespread is it? To this effect, we have conducted an exploratory study on five small-to-medium Java programs frequently used for GUI testing purposes (Table 1). *Addressbook* is the address book application discussed in Section 1. *Calculator* is a simple application mimicking a pocket calculator with basic arithmetic operations. *TerpPresent*, *TerpSpreadSheet*, and *TerpWord* are simple, yet fully functional programs for presentations, spreadsheet usage, and word processing previously used to explore GUI testing [27]. All five programs are written in Java.

In our study, we applied the *Randoop* random test case generator on all five programs. Randoop [33] applies *feedback-directed* random test generation; it comes with basic contracts for a software's

Table 2: Failure and Issues detected by Randoop

Name	Test Failures	Total Issues	JDK Issues	Unit Issues	App Issues
Addressbook	112	1	0	1	0
Calculator	2	2	2	0	0
TerpPresent	32	11	2	9	0
TerpSpreadSheet	34	10	0	10	0
TerpWord	1	1	0	1	0
Total	**181**	**25**	**4**	**20**	**0**

correct operation and then tries to find inputs that differ from these contracts—that is, inputs which are likely to result in failures. Using Randoop's default settings, we had it run for 15 minutes on each of the five programs. We would then examine each reported failure, isolate the issue that caused it (a single issue can cause several similar failures), and classify each failure into one of three categories:

JDK issues are *false failures* coming from violating assumptions of the Java runtime library—for instance, calling methods in the wrong order, or passing insufficiently instantiated objects as parameters.

Unit issues also are *false failures,* but this time coming from misuse of application classes. One such example is the failing test case shown in Figure 2 and Figure 3.

App issues are *true failures* that could actually be achieved by using the application; in other words, there exists some user input that would trigger the defect causing this failure.

Overall, Randoop reported 181 failures. For each failure, we identified the failure-causing issue, which we classified into one of the three categories. This task took us 3–4 minutes per failure, even for an application like *Addressbook,* where we found all 112 failures to be caused by the same single issue described in Section 1.[3]

The results of our classification are summarized in Table 2. The striking observation is the rightmost column: None of the issues reported indicates a defect in the application. *All failures reported by Randoop are false failures*—that is, failures that are impossible to achieve through a real input.

> *In our study of generated unit test suites,* ***all*** *failing test cases are* ***false failures,*** *indicating an error in the test case rather than in the application under test.*

Such false failures are not a feature of Randoop alone; indeed, they would be produced by *any approach that generates tests at the method level,* including CUTE [34], Pex [35], or EvoSuite [11]—in fact, by any tool on any library or application as long as the interfaces lack formal specifications on which interactions are legal, and which ones are not. The explicit preconditions in EIFFEL programs, however, identify false failures by design; and consequently, the AUTOTEST random test case generator for EIFFEL [6] does not suffer from false failures. In programs without contracts, the effects we saw for Randoop would show up for any of these tools, especially as real defects are fixed while false failures remain.

3. SYSTEM TESTING WITH EXSYST

Is there a way we can get rid of false test failures without annotating all our programs with invariants and contracts? The answer is simple: Just test the program at the *system level* rather than at the unit level[4]. At the system level, input is not under the program's control; hence, it *must* properly check against illegal inputs. These checks effectively form *preconditions at the system level*—preconditions which make every remaining failure a true failure.

The big advantage of generating tests at the unit level is that it is very easy to cover a specific method—all the test generator needs to do is call it. When generating tests at the system level, it is much harder to reach a specific place in the code, as one has to go through layers and layers, and satisfy dozens of individual conditions along the way—which is the more a challenge when source code is not available, as is frequently the case for user interface code. The challenge for system test generators is thus to achieve the same high test coverage as unit test generators.

In this section, we present EXSYST, a system test generator for interactive Java programs, first shown at the ICSE 2012 demonstration track [16].[5] EXSYST operates the program under test through its graphical user interface by synthesizing input actions. EXSYST is special in that it aims to *maximize coverage:* via search-based techniques, it tries to generate input sequences such that as much code of the program under test as possible is covered.

Before we discuss the foundations of EXSYST in Section 4, let us first demonstrate EXSYST from a user's perspective. To use EXSYST, all one has to do is to specify the name of the program under test as well as the class of its main window. EXSYST then autonomously generates input sequences in a virtual GUI, reporting any application failures it encountered. 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. Applying EXSYST on the five study subjects listed in Table 1, EXSYST detects a total of six errors[6], out of which four shall be presented in the remainder of this section.

3.1 Addressbook

The *Addressbook* application's user interface consists of three parts (see Figure 1): It has a list of contacts at the top left, a list of categories on the top right, and a panel for editing the currently selected contact at the bottom. When no contact is selected, the application disables all the input fields in the bottom pane, but does not disable the "*Apply*" button. Pressing it results in an uncaught `NullPointerException`.

Due to no contact being selected at the beginning this behavior can be reproduced simply by pressing the enabled "*Apply*" button right after application start.

3.2 Calculator

Calculator fails with `NumberFormatException`s when applying any of its supported mathematical operations to an intermediate result with more than three digits. This is because the application uses the default `NumberFormat` of the English locale for formatting numbers, which uses thousands separators. In reading back the number from its display for subsequent operations, it directly passes the formatted number string to the `BigDecimal()` constructor, which does not support thousands separators.

One input sequence demonstrating this problem is "*500 * 2 + 1 =*". Figure 4 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] In practice, one would add an explicit assumption to the code whenever encountering a false failure, instructing test case generators to avoid this issue in the future. Hence, the total time spent on false failures would not be 3–4 minutes per *failure*, but maybe 5–10 minutes per *issue*, including the extra time it takes to identify and specify the assumption. Note however that the test generation tool must be rerun after adding such assumptions.

[4] Note that this is only suitable when we know what the whole program will be.

[5] This four-page paper presents EXSYST from a user's perspective, briefly discussing its operation and failures found.

[6] GUITAR detected a total of three errors, all also found by EXSYST.

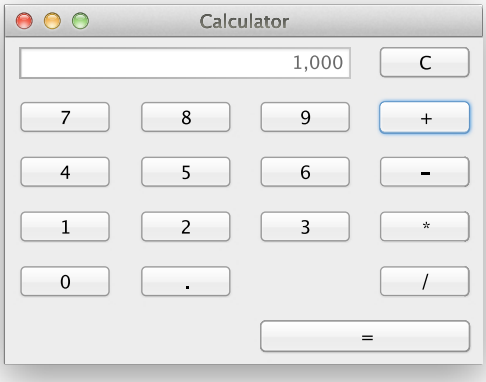


Figure 4: The *Calculator* application outputs numbers with thousands separators, but fails to parse such numbers.

3.3 TerpPresent: Failure 1

The *TerpPresent* application employs a multi-document interface: There is a shared area with menus and toolbars at the top, the blue area at the bottom contains floating windows, one per open document. When closing a window by clicking the close button in its titlebar, the functionality of the shared area pertaining to concrete documents gets disabled. When closing a window by selecting "*File/Close*", this functionality erroneously remains enabled (see Figure 5). Using any of this functionality without an open document results in application failure, namely an uncaught `NullPointerException`.

One input sequence demonstrating this problem is "*File/Close, Shape/Group*" right after application start.

3.4 TerpPresent: Failure 2

A similar failure happens in object selection. Usually, functionality pertaining to shapes is disabled when no shape is selected. 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 thinks that we still have 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 right after application start, then proceed with "*Select/Invert, Select/Invert, Edit/Delete, Edit/Copy*".

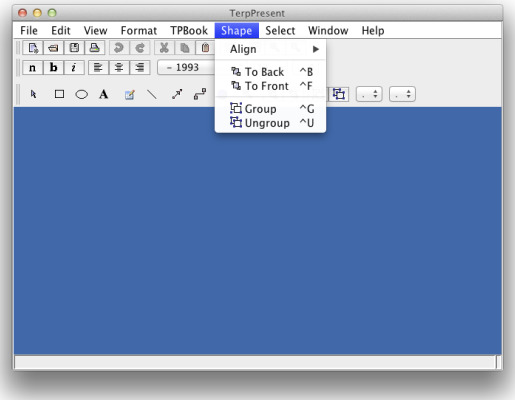


Figure 5: The *TerpPresent* application in an inconsistent state: No document windows are open, yet functionality to interact with documents remains enabled.

Would a tool like Randoop be able to find these failures? The answer is "yes"—if you give it enough time. Unfortunately, these true failures will be buried in thousands of false failures.

4. SEARCH-BASED SYSTEM TESTING

After describing the user's perspective on EXSYST, let us now discuss its technical foundations. EXSYST employs a *search-based approach* to system testing. In detail, it uses a genetic algorithm to evolve a population of GUI test suites towards achieving highest possible coverage. To achieve this, the EXSYST prototype extends our EVOSUITE test generation tool [9] with a new problem representation layer suitable to handle GUI interactions. In this section, we describe the details of this approach.

4.1 Genetic Algorithms

A genetic algorithm is a meta-heuristic search technique which tries to imitate natural evolution. A population of candidate solutions is evolved using genetics-inspired operations such as selection, crossover, and mutation. Each individual is evaluated with respect to its fitness, which is an estimate of how good the individual is with respect to the objective of the optimization. Individuals with better fitness have a higher probability of being selected for crossover, which exchanges genetic material between parent individuals, or for mutation, which introduces new information. At each iteration of the algorithm individuals are selected and evolved until the next generation has reached its desired size. The algorithm stops if either a solution has been found, or if a given limit (e.g., number of generations, number of fitness evaluations) has been reached. When applying a genetic algorithm to a particular problem instance, it is necessary to find a suitable representation for the candidate solutions, define search operators on this representation, and to encode a fitness function that evaluates the individuals.

4.2 Problem Representation

Our overall objective is to produce a set of test cases, such that the set optimizes a code-based coverage criterion. In the past, a common approach to test generation was to generate one individual at a time, and then to merge the resulting individuals to a test suite in the end; however, it has been shown [9] that a granularity of test suites as individuals can be more suitable. In particular, this choice avoids problems that would otherwise arise when trying to apply crossover to sequences of GUI actions. We thus follow this approach, although in principle our techniques could also be applied to a scenario where individual test cases are evolved.

In our context, a test case is a sequence of GUI interactions from our model (see Section 4.5). A test suite is represented as a set T of test cases t_i. Given $|T| = n$, we have $T = \{t_1, t_2, \ldots, t_n\}$. The length of a test suite is defined as the sum of the lengths of its test cases, i.e., $\mathit{length}(T) = \sum_{t \in T} \mathit{length}(t)$.

4.3 Initial Population

The initial population consists of randomly generated test suites. For each test suite, we generate k test cases randomly, where k is chosen randomly out of the interval $[1, 10]$.

Test cases are generated via random walks on the real application UI. Starting in the initial state of the application, we inspect the available GUI elements and their actions, and randomly choose one action, giving precedence to actions previously unexplored. In the resulting state, we again randomly choose an action out of the available ones. This process is repeated until the test case has reached the desired length. This length is randomly chosen from the interval $[1, L]$, where L is a fixed upper bound on the length of test cases.

4.4 Search Operators

Individuals selected for reproduction are evolved using crossover and mutation with certain probability.

4.4.1 Crossover

The crossover operator produces two offspring test suites O_1 and O_2 from two parent test suites P_1 and P_2. To avoid unproportional growth of the number of test cases [9], we choose a random value α from $[0, 1]$; then, the first offspring O_1 contains the first $\alpha|P_1|$ test cases from the first parent, followed by the last $(1-\alpha)|P_2|$ test cases from the second parent. The second offspring O_2 contains the first $\alpha|P_2|$ test cases from the second parent, followed by the last $(1-\alpha)|P_1|$ test cases from the first parent. This way, no offspring will have more test cases than the largest of its parents. It is important to note that crossover only changes test suites while leaving test cases unaffected—crossover at the test cases level can be significantly more complicated, requiring complex repair actions.

4.4.2 Mutation

The mutation operator is applied at both the test suite and the test case level. A test suite can be mutated either by adding new test cases, or by changing existing test cases. Insertion of a new test case happens with probability $\sigma = 0.1$. If a new test case was inserted, we insert another test case with probability σ^2 and so on, until no more test cases are inserted. There is no explicit deletion operator at the level of test suites, but we delete a test case once its length is 0.

In addition to the insertion, each test case of a test suite T is mutated with probability $1/|T|$. A mutation of a test case can result in (up to) three mutation operators being applied to the interaction sequence t, each with probability $1/3$:

- *Deletion operator:* For each action in the sequence there is a chance of it being removed with probability $1/\mathit{length}(t)$.
- *Change operator:* For each action in the sequence there is a chance of its parameters being randomly changed with probability $1/\mathit{length}(t)$.
- *Insert operator:* With decreasing probability (1.0, 0.5, 0.25) a new action is inserted into the sequence. Each insertion happens at a random position of the interaction sequence. A random unexplored action available in that state is chosen and inserted into the interaction sequence just after that state, if available, else we fallback to a random action.

The changes done to an interaction sequence by mutation can result in an action sequence that is not possible on the actual application. To overcome this problem we create a model of the GUI, which we use to repair test cases.

4.5 UI Model

The UI model represents our knowledge of the possible application behavior. This information is contained in a *state machine* that we create from observing actual executions of the application under test. Note that by definition the model only describes a subset of the possible application behavior, namely behavior we have already observed.

Our models are similar to EFG models [29] in spirit, but different in how they are represented: The states of the model serve to tell us which components and actions are available at a point of time in application interaction. Transitions then represent the execution of actions, such as pressing buttons, entering text into text fields, selecting menu items, etc. An *action sequence* is a sequence of such actions, and corresponds to a path through this automaton.

A *state* consists of a hierarchy of all windows which can be interacted with in that state (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). Components are identified by properties such as their type, label, or internal name.

For each action available in a state s, we have a transition leading from s to the state the app. is in after executing the action. If we have not executed an action so far, it leads to the unknown state $s_?$.

Figure 6 shows how a sample model progresses as we continue to discover more information about application behavior: Initially, we start with a model containing only the initial state s_0, corresponding to the application's status right after it has been launched, as well as the unknown state $s_?$. All possible actions in s_0 lead to $s_?$. While generating new test cases we iteratively update the application model: If an action leads to a new state then that state is added to the model, and the transition for that action leads from the old state to the new state. If there previously was a transition to the unknown state, then that transition is removed.

A state in the model represents only the state of the user interface, which might not fully capture the variables determining the program states. Consequently, the model may be non-deterministic, and executing an action a in a specific state might not always lead to the same resulting state. We handle this by allowing multiple outgoing transitions.

4.6 Use of the UI model

As we have previously seen, the insert mutation operator inserts a new random action at some random point p of the target action sequence. In order to find a new action to insert from the model, we first need to find the state s_p we would be in if we executed the action sequence up to p: This is done by following the transitions identified by the actions of the interaction sequence, starting from

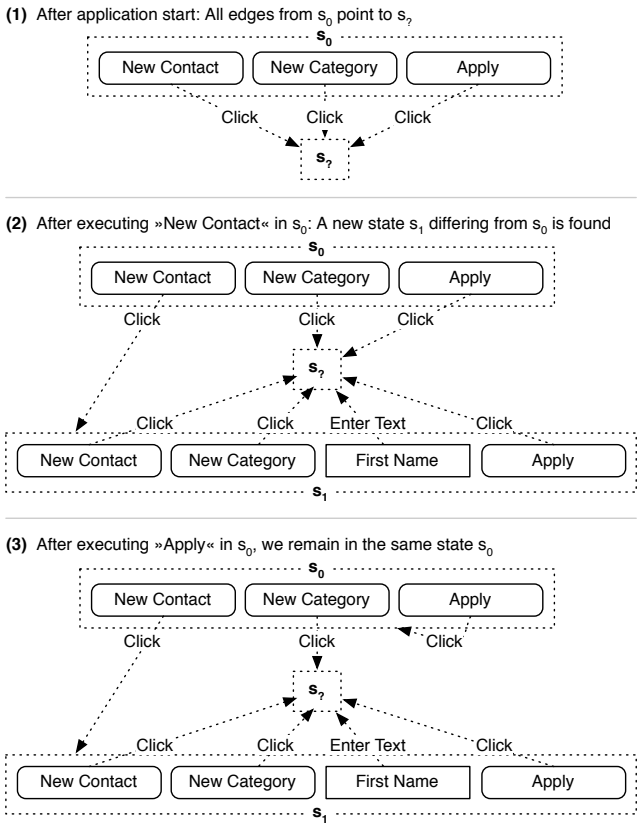


Figure 6: Discovering new application behavior leads to the addition of new states and transitions to the UI model.

s_0. We then need to find the set of possible actions for s_p. The state s_p may be the unknown state, in case the action sequence up to that point contains an action that has not yet been executed.

We define the set of actions $A(s_p)$ available at s_p as follows:

For a known state: All the actions available for all components available in that state.

For the unknown state: Set of all actions available for the last known state along the action sequence.

As explained in Section 4.4.2 the application of a mutation operator can result in an interaction sequence that is not possible in the actual application.

We define the feasibility of an action in state s_p as follows:

For a known state: If the action is contained in $A(s_p)$.

For the unknown state: We assume the action to always be potentially feasible.

If we know any action of the interaction sequence to certainly be infeasible, we consider the whole sequence to be infeasible. Our model allows us to repair a subset of such infeasible action sequences before their actual execution.

We repair an infeasible action sequence by deleting all actions known to certainly be infeasible from it. Afterwards only potentially feasible actions remain. Note that during this repair we do not need to execute the test at all — as the execution of interaction sequences against a GUI is quite costly, we can save significant execution time by repairing infeasible action sequences beforehand.

To run an interaction sequence we execute its actions against the concrete application. If we discover a potentially feasible interaction sequence to be infeasible at execution time, we suspend its execution and update our model. In order to execute actions and gather component information for state abstraction, we employ the open source UISpec4J Java/Swing GUI testing framework [26]. Its window interception mechanism allows us to transparently perform GUI testing in the background.

4.7 Fitness Function

As we aim at maximizing code coverage with our test suites, we base the fitness function on our previous work on whole test suite generation [9]. Rather than optimizing test cases for each branch at a time, the objective is to optimize an entire test suite with respect to *all* branches.

For a given execution, each branch can be mapped to a *branch distance* [24], which estimates how close the predicate guarding it was to evaluating to this particular branch.

For a given test suite T the fitness value is calculated by determining the minimum branch distance $d_{min}(b, T)$ for every target branch b. The fitness function estimates how close a test suite is to covering all branches of a program, and therefore requires that each predicate has to be executed at least twice so that each branch can be taken. Consequently, the branch distance $d(b, T)$ for branch b on test suite T is defined as follows [9]:

$$d(b,T) = \begin{cases} 0 & \text{if the branch is covered,} \\ \nu(d_{min}(b,T)) & \text{if the predicate has been executed at least twice,} \\ 1 & \text{otherwise.} \end{cases}$$

To avoid individual branches dominating the search, each branch is normalized in the range $[0, 1]$ using a normalization function ν.

This results in the following fitness function [9], for a given set of branches B:

$$\text{fitness}(T) = |M| - |M_T| + \sum_{b_k \in B} d(b_k, T) \qquad (1)$$

Here, M is the set of methods, and M_T is the set of methods that are executed by test suite T; this information is included as there can be methods that contain no branching predicates.

The primary objective of our search is code coverage, but we have the secondary objective of producing small test suites. In accordance with our previous experiments [10] we do not include the test length explicitly in the fitness function, but implicitly in terms of the selection function (when ranking individuals, two test suites with identical fitness are sorted according to their length).

5. EVALUATION

5.1 Study Setup

To compare the effectiveness of search-based system testing with state-of-the-practice techniques, we compared EXSYST against three tools:

Randoop 1.3.2 as a representative of random unit test generators. We chose Randoop (described earlier in this paper) over arguably more sophisticated tools for the very pragmatic reason that it ran out of the box. Tools requiring symbolic execution such as CUTE [34] would falter in the presence of classes heavily tied to the system and require specific adaptation (which we wanted to avoid lest our adaptation would influence the result). Pex [35], which would also run out of the box, unfortunately is restricted to .NET programs.

EVOSUITE [11] as a representative of search-based test generators at the API level. As EXSYST is based on EVOSUITE, programs that can be tested with EXSYST can also be tested with EVOSUITE. Comparing EXSYST against EVOSUITE illustrates the effects of applying the same search strategy at the API level and at the system level.

GUITAR 1.0 [1] as a representative of test generators at the GUI level. The choice of GUITAR was purely pragmatic: It was readily available, ran out of the box, had excellent support by its developers, and worked very well on four of our five programs (some of which were already used to test GUITAR itself).[7]

On each of our five test subjects (Table 1), we would run the respective tool for 15 minutes[8] and determined

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

The more instructions a test covers, the more it exercises aspects of internal program behavior. We wanted to see whether EXSYST would be able to exceed the code coverage of Randoop as well as the coverage of GUITAR using the same amount of time. To account

[7]The GUITAR tool works a little differently than the other tools, in that it includes an offline phase, where it tries to dynamically extract a complete EFG model [29] from the application, by invoking all actions it sees as available upfront. The EFG model contains descriptions of all encountered actions. Test case generation then just chains together actions from the model. In our evaluation, we let GUITAR extract the EFG model and let it generate all combinations of actions up to a reasonable depth (usually 3) to obtain a large pool of test cases. For each run iteration, we then kept selecting previously unexecuted random test cases from the pool until the time limit was reached. We did not include the time needed for creating the EFG model and test pool as it was usually larger than 15 minutes and due to how GUITAR works out of the box it could actually generate a larger pool than is strictly necessary.

[8]All times measured on a server machine with 8x 2.93 GHz Intel Core i7-870 Lynnfield CPU cores, and 16 GB RAM, running Linux kernel 2.6.38-11.

Table 3: Tests Generated (avg. of 5 runs)

Subject	Randoop	EvoSuite	GUITAR	EXSYST
Addressbook	3,984	74,940	237	2,682
Calculator	11,310	91,487	254	2,852
TerpPresent	n/a[8]	12,637	n/a[8]	117
TerpSpreadSheet	5,384	26,110	225	273
TerpWord	1,759	26,729	107	449
Total	**22,437**	**231,903**	**823**	**6,373**

Table 4: Failing Tests (avg. of 5 runs)

Subject	Randoop	EvoSuite	GUITAR	EXSYST
Addressbook	146	15	95	127
Calculator	28	20,578	40	42
TerpPresent	n/a[8]	1,173	n/a[8]	9
TerpSpreadSheet	196	2,043	214	29
TerpWord	28	513	106	41
Total	**398**	**24,322**	**455**	**248**

for the randomness of the approaches, we repeated each of the experiments five times with different random seeds, and statistically analyzed the results with respect to coverage.

5.2 Failing Tests

Table 3 lists the number of tests generated by each tool, i.e., for EvoSuite and EXSYST these are the tests executed during the evolution and not only the tests in the final test suite. It is clear to see that while EXSYST produces 7.7 times more test cases than GUITAR, both are dominated by unit test generators. However, each tool uses different maximum lengths (e.g., EvoSuite uses a variable size of up to 40 statements per test, Randoop was configured to use 100 statements per test, while EXSYST was configured with a limit of 50 user interactions and GUITAR uses a limit of three interactions plus any required prefix sequences), so a comparison between the tools is not possible. However, the numbers suggest that unit level testing tools have higher performance due to calling methods directly rather than simulating input events.

Unit test generators seem to be more productive, but are they also more effective? Table 4 lists the failures found by each of the tools. As EvoSuite does not report failures out of the box, we counted all undeclared exceptions (except `NullPointerExceptions` or `IllegalArgumentExceptions`). We did not look into the 24,322 failures found by EvoSuite, but we examined a handful of the 398 failures encountered by Randoop. The failures we investigated all turned out to be false failures from our exploratory study (Section 2), and again, we did not encounter any App issue.

The failures found through GUI testing via GUITAR and EXSYST are all true, though. In *Addressbook*, both tools found that pressing the *Apply* button without any contacts caused a crash (Section 3); in *TerpSpreadSheet,* both detected that pressing the *Paste* button with an empty clipboard would raise a `NullPointerException`. EXSYST also was able to detect issues related to opening and saving files with unspecified format and/or extension—issues that were not found by GUITAR.

> *Issues detected at the GUI level are real.*

5.3 Code Coverage

In terms of bug detection, GUITAR and EXSYST have been superior to Randoop. But how do these four tools compare in terms of coverage? For each of the five programs, we identified *non-dead classes*—that is, classes with at least one reference from within the application—and computed the instruction coverage achieved for each of the tools. Table 5 lists the results per project, and Figure 7 shows in detail how the different tools perform at covering different types of code.

[8]On *TerpPresent*, GUITAR crashed in an endless recursion while extracting a GUI model, and Randoop repeatedly crashed the X-Server.

Coverage. The results are clear: In four out of five programs (Table 5), EXSYST achieves the highest coverage. The only exception is *TerpPresent*, where EXSYST is (yet) unable to synthesize some of the complex input sequences required to reach specific functionality.[9]

> *In four out of five subjects,*
> *EXSYST achieves the highest code coverage.*

Influence of system level testing. Interestingly, this result is not achieved via search-based testing alone: EvoSuite, which applies search-based testing at the unit level, also achieves very high coverage, but is still dominated by EXSYST in all the above cases.

> *Testing at the system level can achieve higher coverage than testing at the unit level.*

Influence of search-based testing. The good EXSYST results, on the other hand, cannot be achieved by testing through the system level alone, as shown in the comparison with GUITAR. Indeed, EXSYST dominates GUITAR in all four code categories, as shown in Figure 7, just as EvoSuite dominates Randoop except for GUI-code.

> *The highest code coverage is achieved*
> *by testing techniques optimized towards this goal.*

Code categories. Across all code categories, EXSYST achieves the highest coverage; the only exception being non-GUI code, where EvoSuite fares slightly better.

> *Only in non-GUI code does testing at the unit level excel over EXSYST.*

Dead code. EvoSuite fares better in non-GUI code because it can directly invoke methods. On the other hand, this also allows EvoSuite and Randoop to exercise *dead code*—code otherwise unreachable from the application. On average, Randoop exercised 52.6% of the instructions in dead classes, and EvoSuite 47.3% of the instructions,[10] while GUITAR and EXSYST by construction never reach dead code. For an application, reporting a failure in unreachable code, as Randoop and EvoSuite can do, is unlikely to gain attention by programmers. On the other hand, tools like GUITAR and EXSYST could be helpful in identifying such code.

> *Unit test case generators may report failures*
> *in unreachable code.*

[9]To place an object, *TerpPresent* requires that the user first selects the type of object to be placed, and then a rectangular target area.

[10]If we omit *TerpPresent,* as for Randoop, the EvoSuite average becomes 42.3%. The difference is due to Randoop achieving a much higher coverage on *TerpWord* than EvoSuite, which can also be seen in Table 6.

Table 5: **Instruction coverage for non-dead classes** (avg. of 5 runs)

Subject	Randoop	EvoSuite	GUITAR	EXSYST
Addressbook	80.7%	86.7%	79.6%	90.7%
Calculator	65.8%	88.7%	88.7%	92.0%
TerpPresent	n/a[8]	49.8%	n/a[8]	25.8%
TerpSpreadSheet	13.5%	33.9%	28.2%	39.0%
TerpWord	43.1%	48.4%	27.0%	53.7%
Average	**50.8%**	**61.5%**	**55.9%**	**60.2%**

Statistical significance. To evaluate the significance of the statistical differences among the different tools, we followed the guidelines in [2]. In particular, we evaluated the statistical difference with a two-tailed Mann-Whitney U-test, whereas the magnitude of improvement is quantified with the Vargha-Delaney standardized effect size $\hat{A}_{12}$. Table 6 lists the average $\hat{A}_{12}$ values for the relevant comparisons. In our context, the $\hat{A}_{12}$ is an estimation of the probability that, if we run tool 1, we will obtain better coverage than running tool 2. When two randomized algorithms are equivalent, then $\hat{A}_{12} = 0.5$. A high value $\hat{A}_{12} = 1$ means that, in all of the five runs of the first tool in the comparison, we obtained coverage values higher than the ones obtained in all of the five runs of the second tool. All our findings above are significant at the 0.05 level for non-dead code, except for the advantage of EXSYST over GUITAR in *TerpWord*.

5.4 Long-Running Tests

In addition to our 15-minute runs, we also conducted a setup with two-hour runs, investigating if and how coverage would improve over time. Unfortunately, Randoop crashed for all five programs after ∼20 minutes, so we cannot compare it against GUITAR and EXSYST, whose coverage improved by 1–2 base points on each program (with the exception of *TerpWord*, where GUITAR would now achieve a coverage of 39.8% on non-dead classes). No additional issues were found.

5.5 Threats to Validity

Like any empirical study, our evaluation faces threats to validity. The most serious threat concerns *external validity*—the ability to generalize from our results. We only looked at five small-to-medium-sized programs with low GUI complexity; full-fledged office applications or 3-D games would impose a far higher challenge for any test generator. Likewise, the good coverage obtained for GUIs does not necessarily generalize to other system interfaces. Also, unit-level testing tools bear no special support for testing applications with graphical user interfaces, which may adversely affect their efficiency.

There is no fundamental reason why our approach would not scale, though; all it takes is sufficient time to explore and exercise the user interface. The fact that our approach produces no false alarms means that the cost of additional testing is limited to burning excess cycles. Regarding the tools used in our study, it is likely that search-based or constraint-based unit test generators would achieve higher coverage than Randoop—a comparison which we could not undertake due to issues of getting these programs to work (Section 5.1). Our point that search-based system testing can achieve a coverage that is as high as unit test generators remains unchallenged, though; and even with more sophisticated unit test generators, the fundamental problem of false failures remains.

Threats to *internal validity* concern our ability to draw conclusions about the connections between our independent and dependent variables. To counter such threats, we generally kept human influence to a minimum. All tools were used "out of the box" with their default settings, and, as always in tool comparisons, tuning parameters such as the test length might change the results. In our classification of Randoop issues (Section 2), we did our best to identify issues in the applications (but failed). To counter such threats, all programs and data used for this study are made publicly available (Section 7).

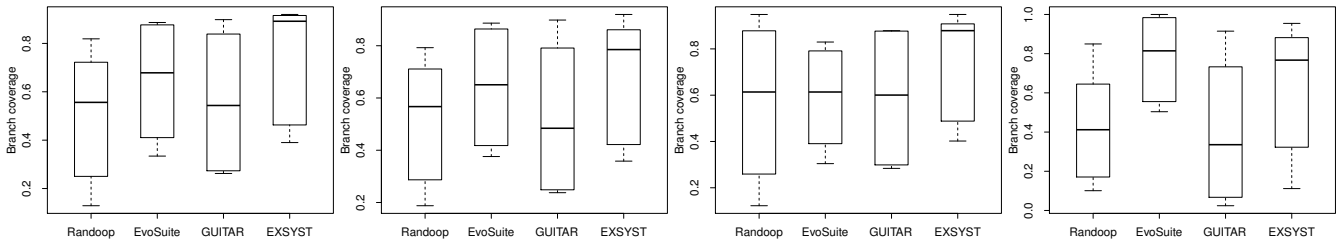
Threats to *construct validity* concern the adequacy of our measures for capturing dependent variables. Using coverage to assess the quality of test suites is common; yet, one may also wish to assess test suites by their ability to find defects. Again, we make our programs and data available to enable the exploration of additional measures and comparisons.

Table 6: Statistical significance. The table lists the $\hat{A}_{12}$ measures from the coverage comparisons, indicating the effect size: $\hat{A}_{12} < 0.5$ means tool 1 resulted in lower, $\hat{A}_{12} = 0.5$ equal, and $\hat{A}_{12} > 0.5$ higher coverage than tool 2. Statistical significance at 0.05 level is highlighted with bold fonts.

Tool 1 vs. Tool 2	EXSYST vs. EvoSuite	EXSYST vs. GUITAR	EXSYST vs. Randoop	EvoSuite vs. Randoop
Addressbook				
Non-Dead	**1.00**	**1.00**	**1.00**	**1.00**
Total	**0.00**	**1.00**	0.88	**1.00**
GUI	**1.00**	**1.00**	**1.00**	**1.00**
Non-GUI	**0.00**	**1.00**	**0.00**	**1.00**
Dead	**0.00**	0.50	**0.00**	**1.00**
Calculator				
Non-Dead	**1.00**	**1.00**	**1.00**	**1.00**
Total	**1.00**	**1.00**	**1.00**	**1.00**
GUI	**1.00**	0.50	**0.00**	**0.00**
Non-GUI	**0.00**	**1.00**	**1.00**	**1.00**
Dead	0.50	0.50	0.50	0.50
TerpWord				
Non-Dead	**1.00**	1.00	**1.00**	**1.00**
Total	**1.00**	1.00	**1.00**	0.28
GUI	**1.00**	1.00	**1.00**	**1.00**
Non-GUI	**0.00**	**1.00**	0.20	**1.00**
Dead	**0.00**	0.50	**0.00**	**0.00**
TerpSpreadSheet				
Non-Dead	**1.00**	**1.00**	**1.00**	**1.00**
Total	**0.00**	**1.00**	**1.00**	**1.00**
GUI	**1.00**	**1.00**	**1.00**	**1.00**
Non-GUI	**0.00**	**1.00**	**1.00**	**1.00**
Dead	**0.00**	0.50	**0.00**	0.56
TerpPresent				
Non-Dead	**0.00**	—	—	—
Total	**0.00**	—	—	—
GUI	**0.00**	—	—	—
Non-GUI	**0.00**	—	—	—
Dead	**0.00**	—	—	—

6. RELATED WORK

Software testing is a commonly applied technique to detect defects in software. When no specification to test against is available, testing usually focuses on exploring the program behavior as thoroughly as possible. A simple yet effective technique to do so is random testing. Although tools such as Randoop [33] can generate large numbers of test cases in short time, there are two important drawbacks: First, some program states are difficult to reach and re-



(a) Total Non-Dead Code (b) Total Code (c) GUI Code (d) Non-GUI Code

Figure 7: Branch coverage achieved on the four case study applications where all tools succeeded for different types of code.

quire particular values or sequences of actions which are unlikely to occur during random testing. The second problem is that random testing tools commonly assume the availability of an automated test oracle that can detect failures—such oracles are available only for a very limited set of defects (e.g., program crashes), while for example normal functional errors can only be detected by verifying that the program behavior matches the expected behavior. This leads to the oracle problem, which requires the tester to understand a generated test case in order to come up with a test verdict for its execution. It is unrealistic to assume that the tester will manually generate oracles for large sets of randomly generated test cases.

6.1 Structural Test Generation

The problem of generating test cases for difficult to reach execution paths has been addressed by several techniques, of which constraint-based and search-based approaches have lead to the most prominent solutions recently. As both constraint-based testing as well as search-based testing have specific drawbacks, a promising avenue seems to be the combination of evolutionary methods with dynamic symbolic execution (e.g., [23, 21]). Constraint-based testing interprets program paths as constraint systems using dynamic symbolic execution, and applies constraint solvers to derive new program inputs, and has been successfully implemented in tools like DART [15] and Microsoft's parameterized unit testing tool PEX [35]. Although EXSYST uses a search-based approach, in principle also approaches based on dynamic symbolic execution can be used for GUI testing [14].

Search-based testing [24] aims to produce test inputs using metaheuristic search algorithms. *Local* search techniques consider the immediate neighborhood of candidate solutions, and have been successfully applied for generating test data at the method level; there is evidence that local search algorithms can perform more efficiently than *global* search algorithms like Genetic Algorithms; yet they are less effective [18]. As the search spaces differ considerably between EXSYST (GUI interactions) and traditional work on testing procedural code (e.g., [18], it is not clear how local search would be applied to the individuals, and whether the results on local versus global search in procedural code would carry over to EXSYST.

However, GUI test cases are very similar in principle to the sequences of method calls that are used in object-oriented unit testing. In this context, Genetic Algorithms (GA) have been successfully used [36, 9], and so EXSYST follows the search based approach applied in EVOSUITE and also uses a GA. EXSYST extends our EVOSUITE tool [9] which performs search at the API level; EXSYST shares the fitness function with EVOSUITE, but uses its own representation and search operators.

Most systematic test generation techniques try to maximize code coverage by targeting one coverage goal at a time. However, optimizing a test case for one specific branch may be very difficult, while at the same time many other branches are accidentally covered during the search. This insight has lead to techniques that try to include all branches in the search, even when targeting one specific branch (e.g., [32]). EXSYST follows the whole test suite generation approach introduced by EVOSUITE [9]: The search aims to optimize a test suite with respect to *all* code branches, thus exploiting the effects of collateral coverage and avoiding the necessity to decide on the order in which to address individual testing goals.

Traditionally, search-based test generation with respect to code coverage uses a test driver (which can be automatically generated) for a particular function, and then the search optimizes the input values to this function [24]. A kind of system testing may be performed at the API level when this function is the `main` function. For example, Jia and Harman [17] determined that the problem of test generation becomes significantly more difficult when testing the entire program, rather than individual functions, which underlines the challenges that EXSYST faces. Although structural test data generation has been the primary focus of search-based testing, it has also been directly applied for testing at the system level, for example to real-time [4] or embedded [20] systems, or Simulink models [40]. In contrast to previous work, EXSYST does not generate traditional test data but GUI event sequences.

6.2 GUI Testing

Usually, automated techniques (including GUITAR) to derive test cases for graphical user interfaces (GUIs) first derive graph models that approximate the possible sequences of events of the GUI [37], and then use these models to derive representative test sets [30, 38], for example guided by coverage criteria defined on the GUI [31]. As this does not correctly account for interdependencies and interactions between different actions (an action could only become available after successfully performing another action) the models are only approximations of real application behaviour. In general, the longer a test case is the more likely it is to be infeasible, and so effort has been put into repairing infeasible GUI test cases [28, 19]. Alternatively, the GUI model itself can be updated during the testing process [39]. 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. [3], who link GUI tests to the code by optimizing individual test cases to achieve a maximum size of the call tree. In contrast, EXSYST continually refines its models from run-time feedback attained via evolution, makes use of newly discovered application behaviour in subsequent iterations of the evolution, explicitly tries to maximize the code coverage of the whole application, and uses guidance based on the branch distance measurement, while at the same time aiming to produce small test suites to counteract the oracle problem.

6.3 Understanding Tests

To ease the task of oracle generation, a common step is to aim to produce as few as possible test cases, which nevertheless exercise the system under test as thoroughly as possible. Usually, this is guided by coverage criteria. When test cases are sequences (e.g., when testing object-oriented software), it is helpful to minimize these sequences [22]. Such minimization could in principle also be applied to the GUI sequences produced by EXSYST. Other work considers the task of producing human readable test cases, for example by including results from web queries [25]. In previous work, we addressed the problem of *implicit preconditions* in the code by learning from user code and trying to imitate it, thus increasing readability and reducing the probability of uninteresting faults [12]. However, exploration of new behavior might still lead to detection of false failures. In contrast, when testing at the system level by definition there is no non-sensical or uninteresting behavior: *Everything* that can be done with the system as a whole must be valid behavior, as the system cannot restrict the environment in which it is used.

7. CONCLUSION AND CONSEQUENCES

Test case generation is a promising approach in theory, but with a number of limitations in practice. One such limitation is *false failures:* Failures of infeasible tests which point to errors in the tests rather than in the code. As we demonstrate in this paper, it can happen that *all* failures reported in a program are false failures.

The problem of false failures can be easily addressed by testing through a system interface. As we show in this paper, well-designed system test case generators can be just as effective as unit test case generators, yet ensure by construction that every failure reported is a true failure. This allows developers to reduce information overload—and focus on the problems that are *real*, and on problems that *matter*. As it comes to test case generation, it is time to reconsider system layers as low-risk, high-benefit alternative to method and unit layers.

Besides focusing on general issues such as generality and packaging for end users, our future work will focus on the following:

Test carving [8] is a technique to record and extract unit tests from system tests. Applying test carving to the executions generated by EXSYST would allow to combine the efficiency of unit testing while still retaining true failures only.

Alternate GUIs and platforms may prove a far more valuable application ground than Java/Swing. Mobile phone and touch-screen devices provide a far richer input vocabulary, yet have very well-defined input capabilities.

Alternate system layers besides GUIs may provide alternate handles for providing system inputs. Structured data inputs, network communication—anything that is under control by a third party could serve as a vector for generating realistic executions. The question is whether search-based techniques could again discover the relations between input features and code features.

Dynamic specification mining infers program models from executions. The more executions observed, the more accurate the mined models become; infeasible executions, on the other hand, spoil the resulting models. Techniques that leverage test case generation for specification mining [7] thus greatly benefit from realistic executions.

Feature location attempts to locate features in code that serve a given concern. With generated realistic executions, one could use dynamic techniques to accurately locate code fragments tied to a particular concept in the user interface or otherwise structured input.

EXSYST is implemented as an extension of the EVOSUITE test generation framework, which will be released as open source in Summer of 2012. Live demos of EVOSUITE as well as additional data on the experiments described in this paper are available at

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

Acknowledgments. Atif Memon provided invaluable support for this work by assisting us in the GUITAR experiments and by making his tool and test subjects available. The work presented in this paper 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". The authors assume responsibility for the content. EXSYST is additionally funded by the Google Focused Research Award "Test Amplification". We thank Eva May, Nikolai Knopp, Kim Herzig, Valentin Dallmeier, David Schuler, Matthias Höschele as well as the anonymous reviewers for providing valuable feedback on earlier revisions of this paper.

8. REFERENCES

[1] "GUITAR — a GUI Testing frAmewoRk", 2009. Website.

[2] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 1–10, 2011.

[3] S. Bauersfeld, S. Wappler, and J. Wegener. A metaheuristic approach to test sequence generation for applications with a GUI. In M. B. 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.

[4] L. C. Briand, Y. Labiche, and M. Shousha. Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. *Genetic Programming and Evolvable Machines*, 7:145–170, June 2006.

[5] M. Burger and A. Zeller. Minimizing reproduction of software failures. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 221–231, New York, NY, USA, July 2011. ACM.

[6] I. Ciupa, A. Leitner, and L. L. Liu. Automatic testing of object-oriented software. In *In Proceedings of SOFSEM 2007 (Current Trends in Theory and Practice of Computer Science)*. Springer-Verlag, 2007.

[7] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 85–96. ACM, 2010.

[8] S. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde. Carving and replaying differential unit test cases from system test cases. *IEEE Transactions on Software Engineering*, 35:29–45, 2009.

[9] 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.

[10] G. Fraser and A. Arcuri. It is not the length that matters, it is how you control it. In *ICST 2011: Proceedings of the International Conference on Software Testing, Verification, and Validation*, pages 150–159, Los Alamitos, CA, USA, 2011. IEEE Computer Society.

[11] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *ISSTA'10: Proceedings of the ACM International Symposium on Software Testing and Analysis*, pages 147–158. ACM, 2010.

[12] G. Fraser and A. Zeller. Exploiting common object usage in test case generation. In *ICST 2011: Proceedings of the International Conference on Software Testing, Verification, and Validation*, pages 80–89, Los Alamitos, CA, USA, 2011. IEEE Computer Society.

[13] G. Fraser and A. Zeller. Generating parameterized unit tests. In *ISSTA'11: Proceedings of the ACM International Symposium on Software Testing and Analysis*. ACM, 2011.

[14] S. R. Ganov, C. Killmar, S. Khurshid, and D. E. Perry. Test generation for graphical user interfaces based on symbolic execution. In *Proceedings of the 3rd international workshop on Automation of software test*, AST '08, pages 33–40, New York, NY, USA, 2008. ACM.

[15] 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.

[16] F. Gross, G. Fraser, and A. Zeller. EXSYST: Search-based gui testing. In *ICSE 2012 Demonstration Track: Proceedings of the 34th International Conference on Software Engineering*, 2012.

[17] M. Harman, Y. Jia, and W. B. Langdon. Strong higher order mutation-based test data generation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 212–222, New York, 2011. ACM.

[18] M. Harman and P. McMinn. A theoretical and empirical study of search based testing: Local, global and hybrid search. *IEEE Trans. Softw. Eng.*, 36(2):226–247, 2010.

[19] S. Huang, M. B. Cohen, and A. M. Memon. Repairing GUI test suites using a genetic algorithm. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 245–254, Washington, DC, USA, 2010. IEEE Computer Society.

[20] P. M. Kruse, J. Wegener, and S. Wappler. A highly configurable test system for evolutionary black-box testing of embedded systems. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO '09, pages 1545–1552, New York, 2009. ACM.

[21] K. Lakhotia, P. McMinn, and M. Harman. Automated test data generation for coverage: Haven't we solved this problem yet? In *TAIC-PART '09: Proceedings of Testing: Academic & Industrial Conference - Practice And Research Techniques*, pages 95–104, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[22] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient unit test case minimization. In *ASE '07: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 417–420, New York, NY, USA, 2007. ACM.

[23] J. Malburg and G. Fraser. Combining search-based and constraint-based testing. In *IEEE/ACM International Conference on Automated Software Engineering (ASE'2011)*, 2011.

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

[25] P. McMinn, M. Shahbaz, and M. Stevenson. Search-based test input generation for string data types using the results of web queries. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2012.

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

[27] A. M. Memon. Dart: A framework for regression testing nightly/daily builds of GUI applications. In *In Proc. of ICSM*, pages 410–419. BibTeX, 2003.

[28] A. M. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Trans. Softw. Eng. Methodol.*, 18:4:1–4:36, November 2008.

[29] 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.

[30] A. M. Memon, M. E. Pollack, and M. L. Soffa. Hierarchical GUI test case generation using automated planning. *IEEE Trans. Softw. Eng.*, 27:144–155, February 2001.

[31] A. M. Memon, M. L. Soffa, and M. E. Pollack. Coverage criteria for GUI testing. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-9, pages 256–267, New York, NY, USA, 2001. ACM.

[32] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *IEEE Trans. Softw. Eng.*, 27:1085–1110, December 2001.

[33] 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.

[34] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 263–272, New York, NY, USA, 2005. ACM.

[35] N. Tillmann and J. N. de Halleux. Pex — white box test generation for .NET. In *TAP 2008: International Conference on Tests And Proofs*, volume 4966 of *LNCS*, pages 134 – 253. Springer, 2008.

[36] P. Tonella. Evolutionary testing of classes. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 119–128, New York, NY, USA, 2004. ACM.

[37] Q. Xie and A. M. Memon. Using a pilot study to derive a GUI model for automated testing. *ACM Trans. Softw. Eng. Methodol.*, 18:7:1–7:35, November 2008.

[38] 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.

[39] X. Yuan and A. M. Memon. Iterative execution-feedback model-directed GUI testing. *Information and Software Technology*, 52(5):559 – 575, 2010. TAIC-PART 2008.

[40] Y. Zhan and J. A. Clark. The state problem for test generation in Simulink. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, GECCO '06, pages 1941–1948, New York, NY, USA, 2006. ACM.