An Approach to Automatic Input Sequence Generation for GUI Testing using Ant Colony Optimization

Sebastian Bauersfeld*
Berner & Mattner GmbH
Gutenbergstr. 15
10587 Berlin, Germany

Stefan Wappler†
Berner & Mattner GmbH
Gutenbergstr. 15
10587 Berlin, Germany

Joachim Wegener‡
Berner & Mattner GmbH
Gutenbergstr. 15
10587 Berlin, Germany

ABSTRACT

Testing applications with a graphical user interface (GUI) is an important, though challenging and time consuming task. The state of the art in the industry are still capture and replay tools, which greatly simplify the recording and execution of input sequences, but do not support the tester in finding fault-sensitive test cases. While search-based test case generation strategies, such as evolutionary testing, are well researched for various areas of testing, relatively little work has been done on applying these techniques to an entire GUI of an application. This paper presents an approach to finding input sequences for GUIs using ant colony optimization and a relatively new metric called maximum call stacks for use within the fitness function.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Test coverage of code, Testing tools*

General Terms

Verification

Keywords

GUI testing, metaheuristics, ant colony optimization, automated test case generation

1. INTRODUCTION

Despite of their known deficits, scripting- and capture and replay tools are frequently used in the industry for testing applications with GUIs. They help with recording and replaying input sequences, but the tester still has to come up with appropriate test cases. The resulting test suites require constant maintenance, since changes to the SUT will cause some of the sequences to not be replayed properly. Considering these difficulties, techniques for automatic test case generation are quite desirable.

One way to deal with the task of finding test data, is to treat it as an optimization problem. There has been a lot of research about this in a field commonly known as search-based software engineering [5, 6]. Some of these techniques have also been applied to GUI testing [2, 3]. Often these works use an approximation of the GUI in the form of an event flow graph (EFG). An EFG is a directed graph whose nodes are the actions that a user can perform. A transition between action x and action y means: *y is available after execution of x*. By traversing the edges of this graph one can generate sequences offline. A length-n input sequence is a tuple $s = (a_1, a_2, \ldots, a_n) \in A^n$ where A denotes the set of all actions that are executable on the SUT. Since the EFG model is only an approximation, and due to the fact that some actions are only executable in certain states of the SUT, not all generated sequences are guaranteed to be feasible.

For example Memon et al. [2] use genetic algorithms to fix broken test suites. They try to find a *covering array* to sample from the sequence space. The array is constrained by the EFG of the GUI (certain combinations of actions are not permitted). Since it is hard to find such a constrained covering array, they employ a special metaheuristic based on simulated annealing. This way they get their initial test suite which, due to the fact that the EFG is only an approximation of the GUI, contains infeasible input sequences. By dropping these, they lose coverage regarding the coverage array. Thus they use a genetic algorithm which utilizes the EFG to generate new sequences offline, which will then be executed and rewarded depending on how many of their actions are executable and on how much coverage they restore.

2. OUR APPROACH

This paper proposes a new approach to input sequence generation, based on ant colony optimization. We use a relatively new criterion called maximum call stacks (MCS) to direct our optimization process. We generate our sequences online, which means that we execute the SUT and repeatedly choose from a set of possible actions. Thus we do not need a model of the GUI and do not have to deal with infeasibility. We developed our own test execution environment, which is able to scan Java SWT applications and obtain all visible widgets and their properties. From these we are able

```
click("File"), click("Print"), press(Tab),
type("22"), press(Tab), type("44"), clickBtn("OK")
```

Figure 1: Input sequence to Microsoft Word.

*sebastian.bauersfeld@berner-mattner.com
†stefan.wappler@berner-mattner.com
‡joachim.wegener@berner-mattner.com

GECCO'11, July 12–16, 2011, Dublin, Ireland.
ACM 978-1-4503-0690-4/11/07.

```
public class CT{
  public static void main(String[] args){
    CT ct = new CT();
    ct.m2();
    ct.m3();
  }
  public CT(){System.out.println("ctor");}
  public void m1(){}
  public void m2(){ m1(); }
  public void m3(){
    m1();
    for(int i = 0; i < 100; i++)
      m2();
  }
}
```

Figure 2: Simple Java program

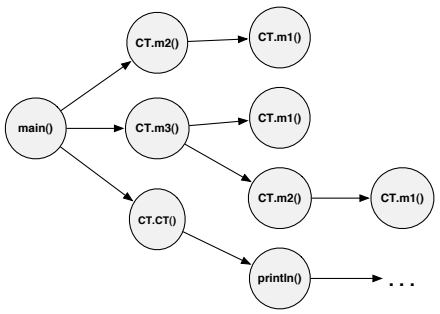

Figure 3: Call tree for the program in Figure 2.

to derive a set of possible actions (clicks, drag and drop actions, keystrokes). Our main SUT is the *Classification Tree Editor*[1], a graphical editor for classification trees. Our general strategy for sequence generation looks as follows:

generateSequence()

```
(1)  do while ¬stoppingCriteria()
(2)      S ← {}
(3)      for i = 1 to populationSize
(4)          startSUT()
(5)          for j = 1 to sequenceLength
(6)              W ← getWidgetsAndProperties()
(7)              A ← derivePossibleSetOfActions(W)
(8)              a ← fitnessProportionateSelection(A)
(9)              execute(a)
(10)             s_ij ← a
(11)         end
(12)         shutdownSUT()
(13)         S ← S ∪ {s_i}
(14)         if mcs(maxSequence) < mcs(s_i)
(15)           maxSequence ← s_i
(16)         end
(17)     end
(18)     adjustPheromones(S)
(19) end
(20) return maxSequence
```

We start the SUT and perform a fitness proportionate selection among the available actions. The fitness of an action is a combination of its pheromone and value: $fitness(a) = pheromone(a)^{\delta} + value(a)^{\epsilon}$ were δ and ϵ are tuning parameters used to adjust the importance of pheromones and values. After generating a certain amount of sequences, we rate these (according to the number of generated MCSs) and adjust the pheromones of the contained actions. We are currently in the process of assessing appropriate stopping criteria and pheromone update algorithms.

We adopted a relatively new criterion that McMaster et al. [4] used to reduce the size of existing test suites. The idea is to instrument the Java virtual machine of an application to obtain a method call tree for every thread. From these trees one is able to generate the set of maximum call stacks (MCS). An MCS is a path through a tree starting at the root node and ending at a leaf. In figures 2 and 3 we see a Java program and its corresponding method call tree.

The tree is just a simplification of the much larger original version, which would also contain the methods of classloaders and Java library code. In order to obtain the number of MCSs, we just count the trees' leaves. The sum of all leaves of all call trees will be our metric. Threads with the same *run()* method will be merged into a single tree.

McMaster et al. provide an implementation for Java applications which we use to find test sequences (of a given length), which generate as many different MCSs as possible when executed on the SUT. The idea behind this is: The more MCSs are generated, the more aspects of the SUT are tested and the better the quality of the sequence.

The above functionality is packaged in a so called JavaAgent, which can be attached to a virtual machine via command line. The approach neither requires modifications to the SUT nor access to its source code.

We are currently in the process of implementing our ideas and are looking forward to presenting a working framework and experimental results soon.

3. ACKNOWLEDGEMENTS

This work is supported by EU grant ICT-257574 (FITTEST).

4. REFERENCES

[1] http://www.berner-mattner.com/en/berner-mattner-home/products/cte/index-cte-ueberblick.html.

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

[3] D. J. Kasik and H. G. George. Toward automatic generation of novice user test scripts. In *CHI '96: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 244–251, New York, NY, USA, 1996. ACM.

[4] S. McMaster and A. Memon. Call-stack coverage for gui test suite reduction. *IEEE Transactions on Software Engineering*, 34:99–115, 2008.

[5] P. McMinn. Search-based software test data generation: a survey: Research articles. *Softw. Test. Verif. Reliab.*, 14:105–156, June 2004.

[6] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In M. Cattolico, editor, *GECCO*, pages 1925–1932. ACM, 2006.