Sebastian Bauersfeld, Stefan Wappler, and Joachim Wegener

Berner und Mattner Systemtechnik GmbH,
Gutenbergstr. 15, 10587 Berlin, Germany
{sebastian.bauersfeld,stefan.wappler,joachim.wegener}@berner-matt
www.berner-mattner.com

**Abstract.** As the majority of today's software applications employ graphical user interface (GUI), it is an important though challeng task to thoroughly test those interfaces. Unfortunately few tools ex to help automating the process of testing. Despite of their well-kno deficits, scripting- and capture and replay applications remain amo the most common tools in the industry. In this paper we will pres an approach where we treat the problem of generating test sequen to GUIs as an optimization problem. We employ ant colony optimi tion and a relatively new metric called MCT (Maximum Call Tree) search fault-sensitive test cases. We therefore implemented a test en ronment for Java SWT applications and will present first results of experiments with a graphical editor as our main application under te

**Keywords:** gui testing, search-based software testing, ant colo optimization.

## 1 Introduction

One reason why the test of applications with GUIs is often neglected, that this kind of testing is labour and resource intensive [14]. Capture a tools help the tester with recording input sequences that consist of mou ments, clicks on widgets and keystrokes. These sequences can then be on the software under test (SUT) to serve as regression tests. Unfo there are a few limitations to this approach:

1. It is difficult to find input sequences that are likely to expose erro SUT. The actions often need to be in a specific order, or have to the context of certain other actions to provoke faults.
2. This kind of testing is laborious and takes a lot of time. One oft several testers to compile an entire test suite.
3. Slight changes to the GUI of the SUT will break tests. For example a button that appears in a sequence as part of a click action, will sequence to not be replayed properly on the updated application.

M.B. Cohen and M. Ó Cinnéide (Eds.): SSBSE 2011, LNCS 6956, pp. 173–187, 2011.
© Springer-Verlag Berlin Heidelberg 2011

**Fig. 1.** Input sequence that causes Microsoft Word to print pages 22 to
current document

Considering these difficulties, techniques for automatic test case g
are quite desirable. One way to deal with the task of finding test data, i
it as an optimization problem. This means that one tries to find solut
the highest quality with respect to the chosen criteria. Since the inp
is large and has a complex structure, one could try to exploit meta
techniques. There has been a lot of research about this in a field c
known as Search-based Software Engineering [12,17,1]. Recently, some
techniques have also been applied to GUI testing [7,6,16,9]. The pr
finding input sequences or test suites is difficult. Some of the challeng

*GUI Model.* Throughout the optimization process it is necessary to gen
sequences which have to be assessed. An input sequence of length $n$ is a
$(a_1, a_2, \ldots, a_n) \in A^n$ where $A$ denotes the set of all actions that are e
on the SUT. Some actions are only available in certain states, so not
sequences are *feasible*. Since many sequences are infeasible, it can be
employ a model of the GUI. Many of the current approaches use an Ev
Graph (EFG) which is a directed graph whose nodes are the actions th
can perform. A transition between action $x$ and action $y$ means: *y is*
*after execution of x*. By traversing the edges of this graph one can
sequences offline, i.e. without starting the SUT. It is possible to auto
obtain an EFG by employing a *GUI-Ripper* [13]. Unfortunately the
EFG is not guaranteed to be complete and needs manual verification.

Since the model is only an approximation, it is still possible to gener
sible sequences. E.g. in Figure 2 we could generate $s = (Edit, Paste)$.
since in most applications the *Paste* menu entry is disabled or invisib
*Copy* action has occurred, the execution of $s$ is likely to fail.

*Appropriate Adequacy Criteria.* Before one can apply optimization te
it has to be defined what constitutes a good test sequence. Several crit
been proposed for GUI testing. In addition to classic ones like code
covering arrays [3] and criteria based on the EFG (e.g. all nodes / all e
have been employed. Choosing the right criteria is criticial to finding

*Exercising the GUI.* Modern GUIs are quite complex, have lots of differ
of widgets and allow various types of actions to be performed by the use
a lot of effort to implement a tool that is able to obtain the state of
and can derive a set of sensible actions. One first has to determine the
of all visible widgets and the state they are in. E.g. if a button is di
would not make sense to perform a click, since the event handler wou

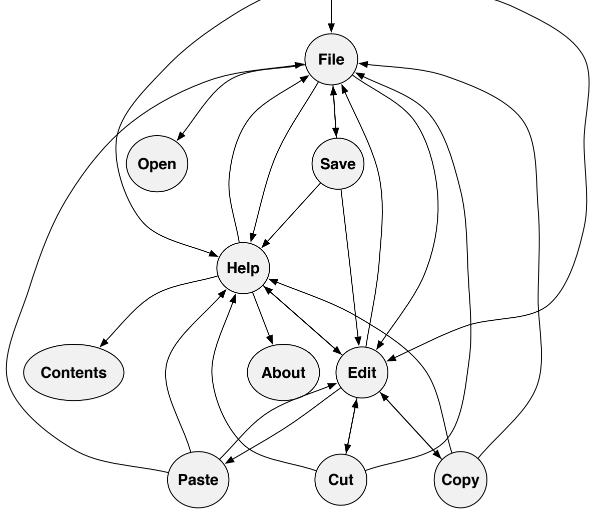


**Fig. 2.** Part of an Event Flow Graph of a typical GUI based application. T
correspond to clicks on menu items.

invoked. Likewise if a message box is on top of all windows and blocks
them, it would not be effective to perform actions on controls other
ones within the message box.

This paper proposes a new approach to input sequence generation
testing, based on ant colony optimization. Our work differs from the
approaches in that we use a relatively new criterion to direct our opt
process. We try to generate sequences that induce a large call tree w
SUT. The maximum call tree criterion (MCT) has been used by McMa
[11] to minimize existing test suites. Similar to Kasik et al. [7] we gen
sequences online, i.e. by executing the SUT. Thus we do not need a mod
not have to deal with infeasibility. We developed a test execution envi
which allows a rich set of action types to be performed (clicks, drag
operations, input via keyboard). This way we can exercise even very
SUTs like graphical editors.

In the following section we will look at related work. Section 3 disc
adequacy criterion, presents our test execution environment and the s
gorithm. In section 4 we present first results by comparing our tech
random sequence generation. Section 5 concludes the paper and rev
approach.

Their implementation captures the widget tree of the GUI at any gi They consider only actions that are executable on the current widget can thus generate arbitrary feasible input sequences. They reward act cause the GUI to stay on the same dialog based on the observation th users learn the behaviour of a GUI's functions through experimenta different parameters within the same dialog. Their program usually st an existing input sequence, where the tester may insert a *DEVIATE* con then deviates this part of the sequence and tries to make it look like it wa by a novice user. By supplying just a single *DEVIATE* command, the generates an entire sequence from scratch. However, according to the this gives quite random results which do not resemble novice user s There are two possible modes: meander and pullback. Meander moc turns over control to the genetic algorithm whenever it encounters a *L* command. It does not return to the remainder of the sequence, that fo *DEVIATE* command. In Pullback mode the authors give reward for to the rest of the sequence, e.g. when an action reaches the same wind action that the program tries to return to (the first action of the re sequence). It is not mentioned how the crossover and mutation opera and what type of subject applications have been used. Thus it is ha how well their implementation will perform on real world subject appl

Memon et al. [6] use genetic algorithms to fix broken test suites. Th an EFG for the GUI and try to find a *covering array* to sample from the space. A covering array $CA(N, t, k, v)$ is an $N \times k$ array ($N$ sequences $k$ over $v$ symbols). In such an array all $t$-tuples over the $v$ symbols are within every $N \times t$ sub-array. A covering array makes it possible t from the sequence space. Instead of trying all permuations of actions ( exponentially many) only the set of sequences that contains all $t$-leng in every position are considered. The parameter $t$ determines the streng sampling process.

Their array is constrained by the EFG (certain combinations of ac not permitted). Since it is hard to find such a constrained covering ar employ a special metaheuristic based on simulated annealing [5]. This get their initial test suite which, due to the fact that the EFG is on proximation of the GUI, contains infeasible input sequences. Their ne to identify these sequences and drop them. By doing that, they lose regarding the coverage array. Thus they use a genetic algorithm whic the EFG to generate new sequences offline, which will then be exec rewarded depending on how many of their actions are executable an much coverage they restore. Infeasible sequences are penalized by addi static value. They pair the individuals in descending order to proce point-crossover, mutate them and use elitism selection. Their stoppin are: maximum number of generations and maximum number of bad mov best individual of the current population is worse than the best of the *click"* actions.

Rauf et al. [16] use a similar technique, also employing an EFG. T a set of short handcrafted input sequences, from which they want to g longer one that contains the short ones as subsequences. A possible us this approach is not mentioned within the paper.

Yongzhong et al. [9] seem to take a similar approach, except that the colony optimization instead. Their work is hard to evaluate since the provide information on their fitness function.

## 3 Our Approach

In this section we will first explain the employed MCT metric and the m behind it. We will proceed with a quick insight into our execution env and conclude with a pseudocode listing of our optimization algorit listing will also present our fitness function.

### 3.1 Adequacy Criterion

For this work, we adopt a relatively new criterion that McMaster et al. to reduce the size of existing test suites. They instrumented the Jav machine of an application to obtain method call trees (see Figure 4) of their SUT. They started with an existing test suite which they exe obtain the method call tree for each sequence. They merged all these tr single large tree and determined the number of its leaves. Then they w remove those test cases which did not cause the tree to shrink significa means that they kept only the sequences which contributed the major leaves. After this process, the reduced versions of the test suites still most of the known faults. Thus we think that this strategy could be su sequence generation. Our goal will be to find sequences that generate a with a maximal number of leaves upon execution on the SUT. Throu rest of this paper, we will refer to this metric as MCT.

Figures 3 and 4 show a Java program and its corresponding me tree. The tree is just a simplification of the much larger original versi would also contain the methods of classloaders and Java library cod there would be several thread call trees for the given program, since threads are used for virtual machine initialization, cleanup, the garbage etc. In order to obtain the MCT metric, we merge all these thread tr single program call tree and count its leaves. Figure 5 illustrates thi We introduce a new root node and merge threads with the same `run(` into the same subtrees.

The idea behind the MCT metric is as follows: The larger the pro tree, the more contexts the methods of the SUT are tested in. For ex

```java
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();
  }
}
```

**Fig. 3.** Java program

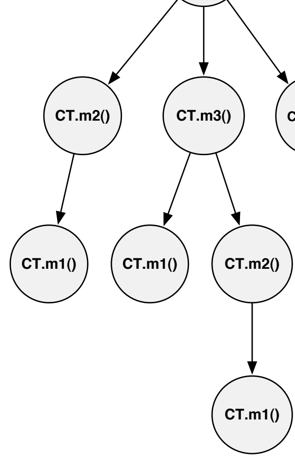


**Fig. 4.** Simplified call tre
main thread of the program
3 (library code partly omitt

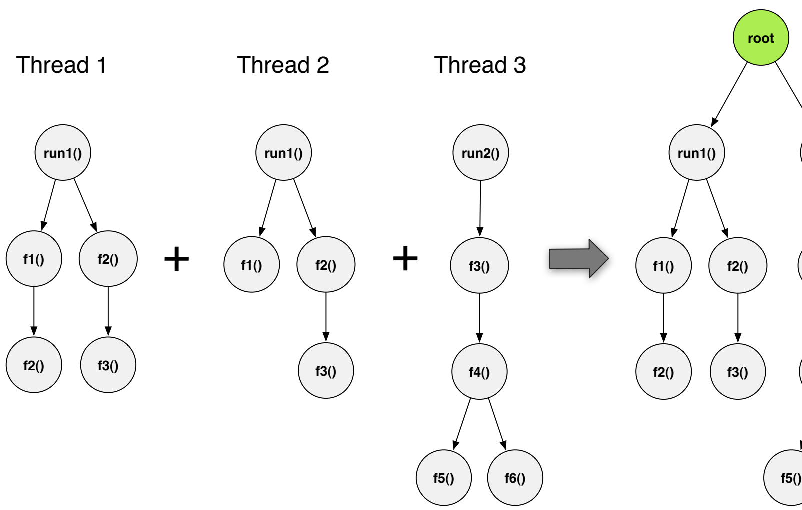


**Fig. 5.** Merging thread call trees into a single program call tree

m1() might depend on directly or indirectly and thus affect its beha
the premise is: The larger the call tree, the more aspects of the SUT a
The experiment of McMaster et al. gave first evidence of this.

One of the advantages of the MCT metric is, that no source code is
to obtain it. In addition the metric tracks activities within third-party
and thus is suitable for testing the SUT as a whole. McMaster et al. [11
an implementation that is able to obtain the call tree generated by
sequence. They developed solutions for Java- and C- based application
Java version employs the *Java Virtual Machine Tool Interface* (JVMT
provides callbacks for various events, like Thread-Start / -End, Meth
/ -Exit. This way a call tree can be generated for every thread. Table
the results of applying their implementation to three simple programs
see that the MCT metric captures activities within third-party modu
the Java library). Depending on the parameters supplied to println()
methods are invoked and hence different call trees are generated.

Unfortunately the JVMTI prevents the virtual machine from doin
tant optimizations [8]. This not only degrades runtime performance, b
destabilize the SUT and thus introduce artificial faults. Since we enc
slowdowns of up to factor 30 and crashes with our main SUT, we devel
own solution using byte code instrumentation. We insert static metho
the beginning and end of each method to obtain the call tree. This tec
frequently used by Java profilers [2].

**Table 1.** Three simple programs (main class and main method omitted
corresponding MCT metric. The programs have been executed on a Sun
Windows XP.

| Code | # Call Tree Leaves |
|---|---|
| `System.out.println("");` | 716 |
| `System.out.println("Hello World!");` | 748 |
| `System.out.println("Hello\nWorld!");` | 750 |

### 3.2 Test Environment

This section gives a short introduction to our test environment, whic
us to operate the SUT, i.e. click on controls, type in text and perform
drop operations. In order to do this, it needs to be able to

1. scan the GUI of the SUT to obtain all visible widgets and their p
(size, position, focus etc.),
2. derive a set of interesting actions (e.g. a visible, enabled button o
ground window, is clickable),

[1] http://sourceforge.net/projects/javacctagent/

One could try to take advantage of the various commercial and op
scripting and capture and replay tools. We tested *TestComplete*[2], *SWT-*
*WindowTester*[4]. Unfortunately all of these tools lack the capabilities
in 2. and 3.[5] They are good at recording and replaying, but expect th
supply the right actions. Thus we implemented our own tool, which
exercise nearly all types of SWT widgets.

Figure 6 outlines the process of generating a feasible input sequen
start the SUT and 2. perform the byte code instrumentation, which is
to obtain the method call tree at the end of the execution cycle. 3. The
the GUI to obtain all widgets and their properties. That means we dete
bounding rectangles of buttons, menus and other widgets and detect
they are enabled, have the focus etc. From this information we are a
compile a set of possible actions. In Figure 8 we can see a selection o
that can be performed within our main SUT, the *Classification Tree*
We only consider "interesting actions". For example: A click on a gr
i.e. disabled, menu item would not make sense since no event handler
invoked. 5. Our optimization algorithm then selects an appropriate ac
6. executes it. We repeat steps 3 to 6 until we generated a sequence of a
length. 7. Then we stop the SUT and count the number of leaves of our

Figure 7 shows the components of our test environment. All of t
functionality is packaged in a so called JavaAgent, which is attache
virtual machine of the SUT. Thus it has access to each loaded class an
including the widget objects. It obtains the necessary information and s
the optimization component which selects the actions that are to be pe
At the end of a sequence the optimization component retrieves the MC
from the agent.

Our implementation is also able to obtain the thrown exceptions dur
If these exceptions are "suspicious" or caused the SUT to crash, the co
ing sequence will be stored in a special file for later inspection.

### 3.3 The Algorithm

We will now describe our search-algorithm. We consider fixed-length
quences of the form $s = (a_1, a_2, a_3, \ldots, a_n) \in A^n$ where $A$ denotes the
actions within the SUT. We are trying to find a sequence $s^* \in A^n$

[2] `http://smartbear.com/products/qa-tools/automated-testing/`
[3] `http://www.eclipse.org/swtbot/`
[4] `http://code.google.com/javadevtools/wintester/html/index.html`
[5] TestComplete has a naming scheme, but it doesn't work for all SWT wid
[6] `http://www.berner-mattner.com/en/berner-mattner-home/products/c`
`index-cte-ueberblick.html`

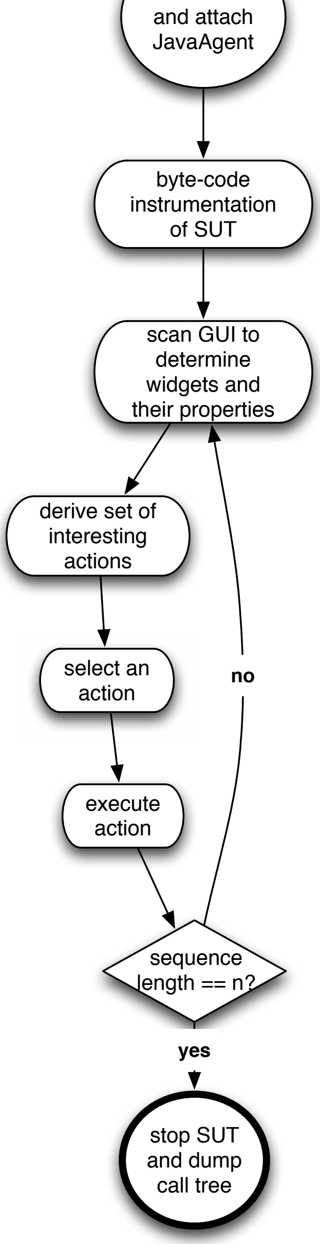


**Fig. 6.** Sequence generation

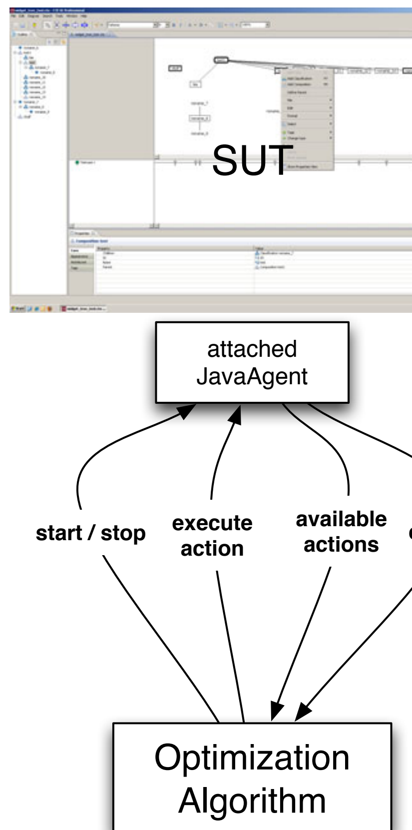


**Fig. 7.** Components of our framew

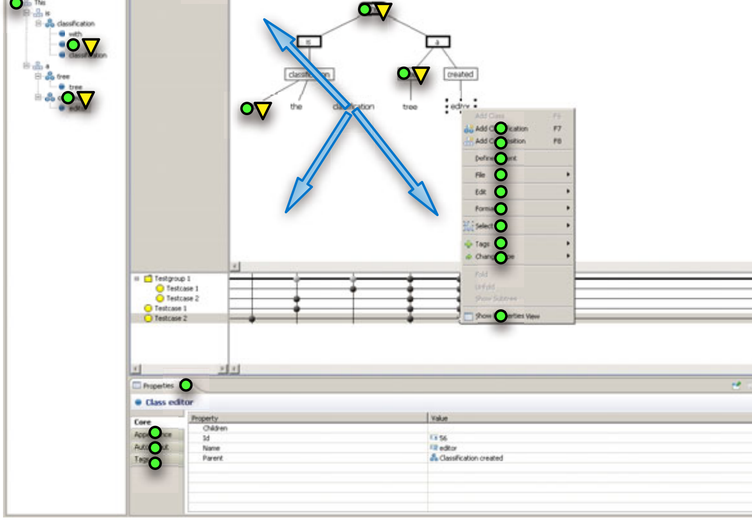

**Fig. 8.** Possible actions that can be performed on our SUT (green circles:
yellow triangles: right clicks, blue arrows: drag and drop operations, green sta
clicks). These are not all possible actions, but only a selection, to preserve c

$q^* = fitness(s^*) = max\{fitness(s)|s \in A^n\}$, where $fitness(s)$ return
of the method call tree generated by $s$, i.e. the MCT metric.

Due to the fact that not every action $a$ is available at all states of
we cannot arbitrarily combine actions, but have to make sure that our a
produces feasible permutations. Classical metaheuristics like simulate
ing or genetic algorithms make use of a mutation operator which, depe
certain parameters, makes small or large changes to candidate soluti
first issue here is, that in our case the operator has to maintain closu
application should not affect the feasibility of sequences. Due to the
dependencies among the actions, it is hard to implement such an opera
thermore it is unclear what constitutes a *small* or *large* change of a
If we substitute an action at position $i$, the rest of the sequence migh
undergo a complete change too, in order to maintain feasibility.

Thus we would like to bypass the implementation of such an oper
rather use metaheuristics where it is not necessary. This led us to con
colony optimization (ACO), which is an approach to combinatorial o
tion [10]. Since ACO does not make use of a mutation operator, we t
more suitable for the generation of input sequences than for exampl
algorithms.

ACO is a population-oriented metaheuristic, that means in contrast
ods such as hill climbing or simulated annealing, which constantly twea

component $c \in C$ being chosen, is determined by its pheromone value
assessing the fitness of each trail, the pheromones of the components
within that trail are updated according to the fitness value. Thus the ph
tell us about the history of a component and how often it participated
quality solutions.

**Algorithm 1.** *maximizeSeq*(*popsize*, *seqlength*)

**Output**: Sequence that generates a large call tree upon execution.
**begin**
  $\boldsymbol{p} \leftarrow \langle p_1, \ldots, p_l \rangle$ ; `/* initialize pherom`
  $best \leftarrow \square$ ; `/* best trail discovered so`
  $bestValue \leftarrow 0$ ; `/* fitness of best t`
  **while** $\neg stoppingCriteria()$ **do**
    **for** $i = 1$ *to popsize* **do**
      startSUT()
      **for** $j = 1$ *to seqlength* **do**
        $E \leftarrow$ scanGuiForActions()
        $t_{ij} \leftarrow pseudoProportionateActionSelection(\boldsymbol{p}, E)$
      shutdownSUT()
      $q_i \leftarrow fitness(t_i)$
      **if** $bestValue < q_i$ **then**
        $bestValue \leftarrow q_i$
        $best \leftarrow t_i$
    $\boldsymbol{p} \leftarrow adjustPheromones(\boldsymbol{t}, \boldsymbol{q}, \boldsymbol{p})$
  **return** best
**end**

**Algorithm 2.** fitness(*seq*)

**Input**: sequence *seq* to evaluate
**Output**: fitness value of the sequence *seq*.
**begin**
  **return** number of leaves of the call tree generated by *seq*
**end**

Algorithm 1 outlines our overall strategy. It does the optimization
tries to find a sequence that generates a large call tree. At the en
generation we pick the $k$ best rated trails and use them to update the ph
of the contained actions. Our pheromone update rule for an action $a_i$ is
$p_i = (1-\alpha) \cdot p_i + \alpha \cdot r_i$, where $r_i$ is the average fitness of all trails that $a_i$
in and $\alpha$ the evaporation rate, as described in [4].

During the construction of a trail, we select the components (the
actions) according to the pseudo random proportional rule described

ations. In the future we will employ more sophisticated criteria like for
the number of bad moves.

## 4 Experiment

To get a first impression of how well the optimization algorithm perf
compared it to a random generation strategy. The *Classification Tree*
graphical editor for classification trees, served as our SUT. Table 2 s
parameters and results of the random and ACO runs. $k$ is the number
sequences in every generation, which were used for the pheromone u
is the probability parameter for the pseudo proportional random selec
and $\alpha$ is the pheromone evaporation rate. Both runs generated 6000 s
Figures 9 and 10 show the course of the optimization processes. The A
rithm eventually found a sequence with $MCT_{ACO} = 144082$, whereas
sequence found by the random algorithm has $MCT_{Random} = 91587$.
the quality of the candidate solutions significantly improved towards t

**Table 2.** Parameters and results of the runs

| desc | $k$ | $\alpha$ | $\rho$ | popsize | generations | seqlength | pheromone default | best MCT |
|---|---|---|---|---|---|---|---|---|
| ACO | 15 | 0.3 | 0.7 | 20 | 300 | 10 | 30000 | 144082 |
| Random | all | 0.0 | 0.0 | 20 | 300 | 10 | 30000 | 91587 |

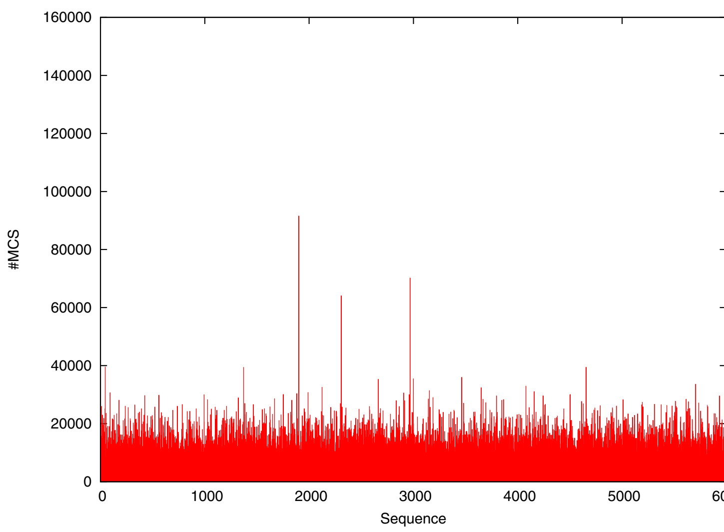


**Fig. 9.** Random run

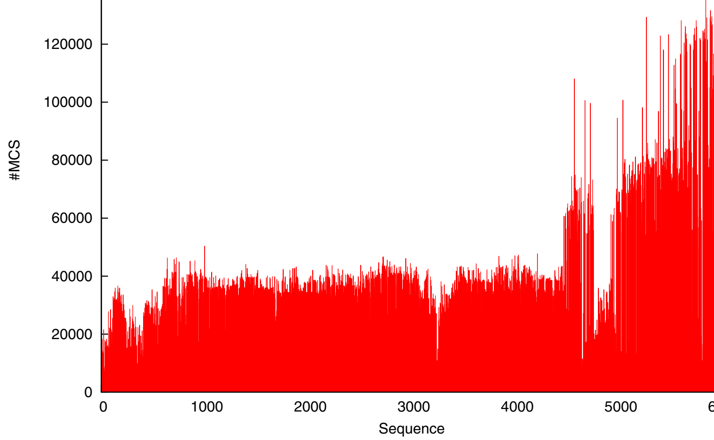


**Fig. 10.** ACO run

the ACO run, the algorithm might have performed better, but due to o
simple stopping criterion (fixed number of generations) the optimizatio
probably terminated prematurely. In future experiments we will emp
sophisticated criteria to determine when to stop.

## 5 Conclusion

In this paper we proposed an approach to automatic generation of
quences for applications with a GUI. Our approach differs from earl
in the way we tackle the optimization problem. We use dynamic feedb
the SUT in the form of the MCT metric to direct the search process.
forgo the application of a GUI model, we do not have the problem of g
infeasible sequences. We implemented a test environment which enab
generate arbitrary input sequences for Java SWT applications. Our opt
algorithm employs ant colony optimization with the pseudo proporti
dom selection rule. A first experiment showed that the implementation
that the algorithm continuously improved the candidate solutions and e
found a better sequence than the random strategy. In future works we
out additional experiments to analyze the fault revealing capabilities o
erated sequences. Our goal is to take a set of different applications wi
faults and generate test suites for them. We will then determine the a
faults discovered by these test suites.

## References


1. Baresel, A., Sthamer, H., Schmidt, M.: Fitness function design to impr tionary structural testing. In: Langdon, W.B., Cantú-Paz, E., Mathias, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., W Bull, L., Potter, M.A., Schultz, A.C., Miller, J.F., Burke, E.K., Jonoska GECCO, pp. 1329–1336. Morgan Kaufmann, San Francisco (2002)
2. Binder, W., Hulaas, J., Moret, P., Villazón, A.: Platform-independent p a virtual execution environment. Software: Practice and Experience (20
3. Cohen, M.B., Gibbons, P.B., Mugridge, W.B., Colbourn, C.J.: Constru suites for interaction testing. In: Proceedings of the 25th International C on Software Engineering, ICSE 2003, pp. 38–48. IEEE Computer Soci Washington, DC, USA (2003)
4. Dorigo, M., Blum, C.: Ant colony optimization theory: a survey. Theor. Sci. 344, 243–278 (2005)
5. Garvin, B.J., Cohen, M.B., Dwyer, M.B.: An improved meta-heuristic constrained interaction testing. In: Proceedings of the 2009 1st Internati posium on Search Based Software Engineering, SSBSE 2009, pp. 13– Computer Society Press, Washington, DC, USA (2009)
6. Huang, S., Cohen, M.B., Memon, A.M.: Repairing gui test suites using algorithm. In: ICST 2010: Proceedings of the 2010 Third International C on Software Testing, Verification and Validation, pp. 245–254. IEEE Society Press, Washington, DC, USA (2010)
7. Kasik, D.J., George, H.G.: Toward automatic generation of novice user te In: CHI 1996: Proceedings of the SIGCHI conference on Human factor puting systems, pp. 244–251. ACM, New York (1996)
8. Kurzyniec, D., Sunderam, V.: Efficient cooperation between java and na – jni performance benchmark. In: The 2001 International Conference o and Distributed Processing Techniques and Applications (2001)
9. Lu, Y., Yan, D., Nie, S., Wang, C.: Development of an improved gui a test system based on event-flow graph. In: International Conference on Science and Software Engineering, vol. 2, pp. 712–715 (2008)
10. Luke, S.: Essentials of Metaheuristics. Lulu (2009), `http://cs.gmu.e` `book/metaheuristics/`
11. McMaster, S., Memon, A.: Call-stack coverage for gui test suite reduct Transactions on Software Engineering 34, 99–115 (2008)
12. McMinn, P.: Search-based software test data generation: a survey: Res cles. Softw. Test. Verif. Reliab. 14, 105–156 (2004)
13. Memon, A., Banerjee, I., Nagarajan, A.: Gui ripping: Reverse engineerin ical user interfaces for testing. In: Proceedings of the 10th Working Con Reverse Engineering, WCRE 2003, IEEE Computer Society Press, W DC, USA (2003)
14. Memon, A.M.: A comprehensive framework for testing graphical user Ph.D, Advisors: Mary Lou Soffa and Martha Pollack; Committee Prof. Rajiv Gupta (University of Arizona), Prof. Adele E. Howe (Colo University), Prof. Lori Pollock (University of Delaware) (2001)

of Software Engineering, pp. 256–267. ACM, New York (2001)
16. Rauf, A., Anwar, S., Jaffer, M.A., Shahid, A.A.: Automated gui test
analysis using ga. In: Third International Conference on Information Te
New Generations, pp. 1057–1062 (2010)
17. Wappler, S., Wegener, J.: Evolutionary unit testing of object-orientec
using strongly-typed genetic programming. In: Cattolico, M. (ed.) GE
1925–1932. ACM, New York (2006)