A Metaheuristic Approach to Test Sequence Generation for GUI-based Applications

Sebastian Bauersfeld, Dr. Joachim Wegener
Berner & Mattner Systemtechnik GmbH

Overview

- Motivation
- Classification-Tree Editor CTE XL Professional
- Objectives for SBST application
- Application of ACO
- Objective Function
- Test Environment
- Fully Automatic Testing of CTE XL Professional
- Conclusion, Outlook
Motivation

- Many GUI based applications in all application domains
- Tester’s task to define, execute and evaluate most interesting input sequences
- Input sequences are sequences of user actions (mouse events, keyboard events etc. such as clicks, drag and drop, keystrokes)
- Existing tools:
  - Many Capture + Replay Tools available, but limited applicability (e.g. B&M uses TestComplete and QF Test)
    - Definition of test sequences
      - by capturing user actions
      - developing test scripts
    - Only replay part is “automatic”
- Test suites require constant maintenance
- Labor intensive
- Automatic generation of input sequences is quite desirable

Typical Berner & Mattner Products

**CTE XL Prof.**
Systematic test case design for specification-based testing

**MESSINA**
Virtual Integration and Testing of AUTOSAR-SWC

**MERAN**
Variant management and model-based development for specifications in DOORS

Java applications based on Eclipse RCP and SWT
Objectives for Application of Search-Based Testing

- search for interesting test sequences
- fullest possible execution of the program functions in different contexts
- in our case:
  - find sequences that generate large amounts of different call stacks (the more CSs a sequence generates, the more aspects of the SUT are tested (McMaster et al.) ⇒ call trees with many leaves most interesting for fault detection)
  - check for exceptions occurring during the execution
- Alternatively:
  - Check for memory leaks,
  - Check for code coverage,
  - Check for performance bottlenecks,
  - Check for assertion violations
  - …
Application of Ant Colony Optimization

Reasons for using ACO

• ACO usually applied for sequence generation problems, e.g. TSP. Independent of mutation and crossover.

• Mutation operator problems for sequence generation
  – Easy generation of infeasible sequences (not all actions are available in all contexts)
  – Neighbourhood of a sequence leads to artificial definitions

• Crossover operators introduce similar problems (exchanging sequence parts will lead to infeasible sequences)

Ant Colony Optimization

• Idea:
  – C = component set (here: C = set of feasible actions)
  – Generate trails (sequences of user actions) by selecting components \( c_i \in C \) considering pheromone values \( p_i \)
    • Pseudo random proportionate selection
  – Assess trails (# Call Tree Leaves)
  – Reward components \( c_i \) that appear in “good” trails by increasing their pheromones \( p_i \)
  • After each generation
    – Only top k trails are considered
    – \( p_i := p_i \cdot (1 - \alpha) + \alpha \cdot r_i \) where \( \alpha \) is the evaporation / learning rate and \( r_i \) the average reward of the trails that \( c_i \) appeared in
Objective Function

For each generated sequence of user actions the size of its call tree is calculated by the number of leaves: # CT leaves.

Call Trees for multiple threads are combined into one call tree. Redundancies are eliminated.

Test Environment
Test Environment

Optimization Component

- independent of source code
- attaches to SUT
- instruments bytecode to obtain call tree (includes third party modules)
- scans the GUI to create a widget tree for each execution state
- defines unique identifiers for each action
- executes selected actions
- returns overall call tree
- monitors exceptions

Java Agent

- implements the search (ACO)
- maintains the pheromones for each named action according to call trees
- selects most promising actions
- analyse exceptions

Test Environment – Sequence Generation
Active Widget Tree

Form

Main Menu

(dropdown)
Search
Experiment

ACO Run

Random Run

<table>
<thead>
<tr>
<th>desc</th>
<th>k</th>
<th>α</th>
<th>ρ</th>
<th>popsize</th>
<th>generations</th>
<th>seqlength</th>
<th>pheromone</th>
<th>max</th>
<th>#CS</th>
<th>duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>aco</td>
<td>15</td>
<td>0.3</td>
<td>0.7</td>
<td>20</td>
<td>300</td>
<td>10</td>
<td>30000</td>
<td>144082</td>
<td>12.5 h</td>
<td></td>
</tr>
<tr>
<td>(random)</td>
<td>all</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>300</td>
<td>10</td>
<td>1</td>
<td>91587</td>
<td>12.5 h</td>
<td></td>
</tr>
</tbody>
</table>

Conclusion

- High demand for automatic GUI testing in industrial practice
- Typical application: CTE XL Professional (Eclipse RCP, SWT)
- Functional testing for logical errors difficult, because guidance to unknown logical errors hard to formalize
- Functional testing for exceptions, memory leaks, … possible

- Test environment allows to
  - determine all possible user actions in each execution state
  - selects most interesting actions
  - assesses overall quality of test sequences by analyzing the call tree
  - generates long test sequences with most highest variety

- Evaluation
  - Application of search successfull
  - Initial experiments confirm better performance than random testing
Outlook

- Experiments on generating entire test suites to be performed
- Possible improvement of algorithm to be more explorative

- Evaluate more advanced objective functions (not only number of call tree leaves)

- Increase efficiency
  - Sequence generation is expensive ➔ parallelization of search and test
  - ACO good choice? ➔ disregards linkage among actions (context of actions not considered for pheromone value calculation)

- Fault sensitivity of generated sequences ➔ empirical evaluation