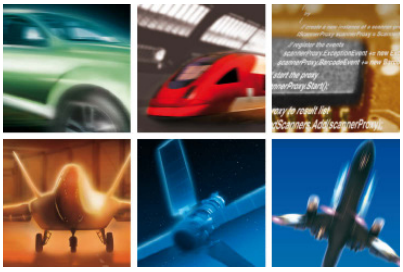
A Metaheuristic Approach to Test Sequence Generation for Applications with a GUI

Sebastian Bauersfeld, Stefan Wappler, Joachim Wegener
Berner & Mattner Systemtechnik GmbH



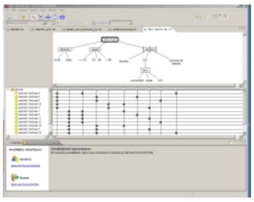
Overview

- Motivation
- Approach
- Objective Function
- Application of ACO
- Test Environment
- Experiments: Fully Automatic Testing of CTE XL Professional
- Conclusion + Outlook

Motivation

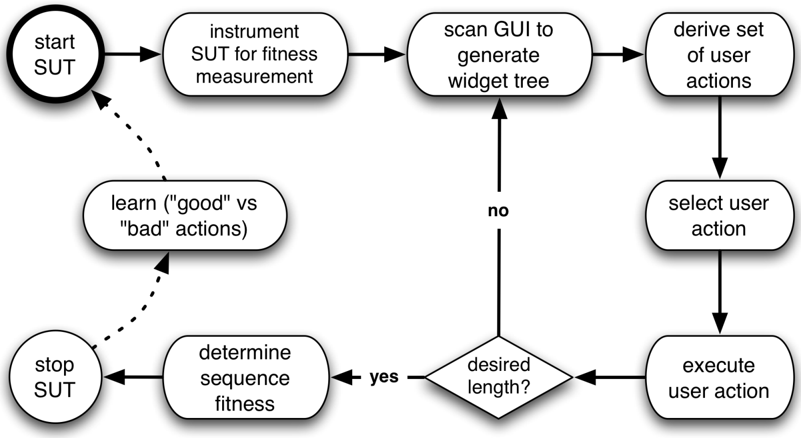
- Many GUI based applications in all application domains
- Tester’s task: finding, executing and evaluating most interesting input sequences
- Input sequences are sequences of user actions (mouse events, keyboard events etc., such as clicks, drag and drop, keystrokes)

- 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

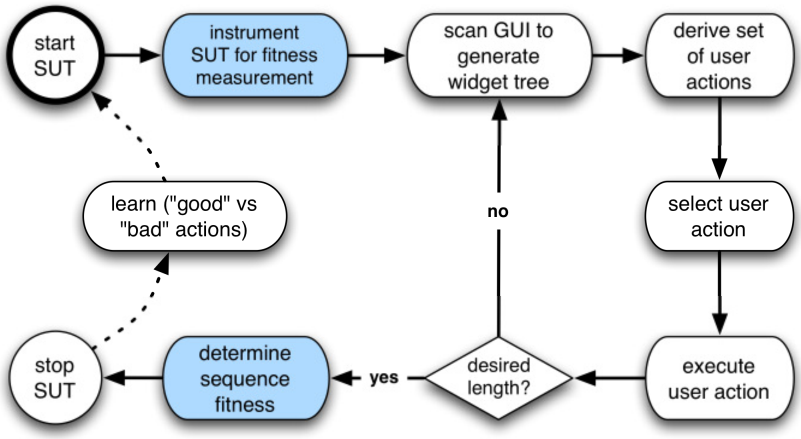


➢ **Automatic generation of input sequences is desirable**

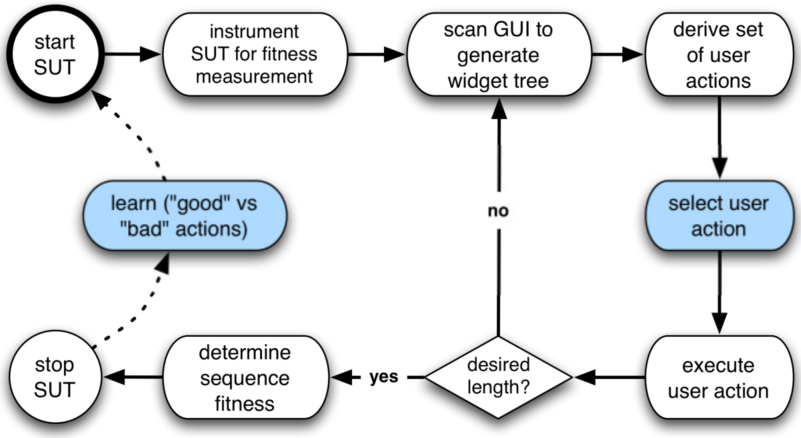
The Approach


start SUT
instrument SUT for fitness measurement
scan GUI to generate widget tree
derive set of user actions
select user action
execute user action
desired length?
no
yes
determine sequence fitness
stop SUT
learn ("good" vs "bad" actions)

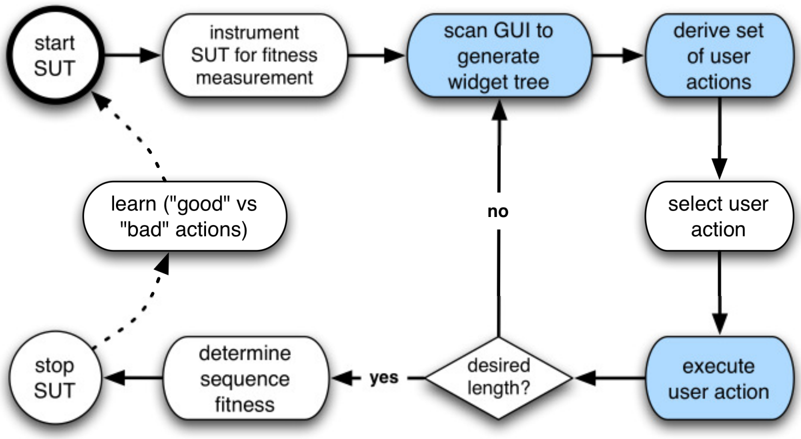
Objective Function


start SUT
instrument SUT for fitness measurement
scan GUI to generate widget tree
derive set of user actions
learn ("good" vs "bad" actions)
no
select user action
stop SUT
determine sequence fitness
yes
desired length?
execute user action

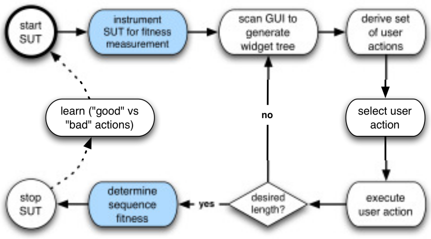
Optimization Algorithm


start SUT
instrument SUT for fitness measurement
scan GUI to generate widget tree
derive set of user actions
learn ("good" vs "bad" actions)
no
select user action
stop SUT
determine sequence fitness
yes
desired length?
execute user action

Test Environment


start SUT
instrument SUT for fitness measurement
scan GUI to generate widget tree
derive set of user actions
select user action
execute user action
desired length?
no
yes
determine sequence fitness
stop SUT
learn ("good" vs "bad" actions)

Objective Function


start SUT
instrument SUT for fitness measurement
scan GUI to generate widget tree
derive set of user actions
select user action
execute user action
desired length?
no
yes
determine sequence fitness
stop SUT
learn ("good" vs "bad" actions)

Search for interesting test sequences

The larger the Call Tree of a sequence, **the more aspects** of the SUT **are tested** (McMaster et al.).

Call tree: Structure that displays **calling relationships among methods** of an executed program. Each node represents a method. A directed edge between two nodes *f* and *g* means, that the method f called the method g.

Sequence fitness: **Number of call tree leaves** (call trees with many leaves most interesting for fault detection)

Implementation: **Bytecode instrumentation** of the SUT to obtain the call tree
➔ no source code needed

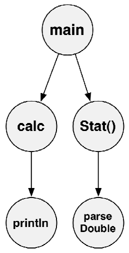
Objective Function: # Call Tree Leaves

```
public class Stat{
    double[] data;
    public static void main(String[] args){
        Stat s = new Stat(args);
        s.calc();
    }
    public Stat(String[] args){
        data = new double[args.length];
        for(int i = 0; i < args.length; i++)
            data[i] = Double.parseDouble(args[i]);
    }
    public void calc(){
        if(data.length > 1){
            System.out.println(mean() + ", " + var());
        }else{
            System.out.println(data[0] + ", " + 0.0);
        }
    }
    double mean(){
        double ret = 0.0;
        for(double d : data) ret += d;
        return ret / data.length;
    }
    double var(){
        double ret = 0.0;
        for(double d : data) ret += Math.pow(d - mean(), 2);
        return ret / data.length;
    }
}
```

Objective Function: # Call Tree Leaves

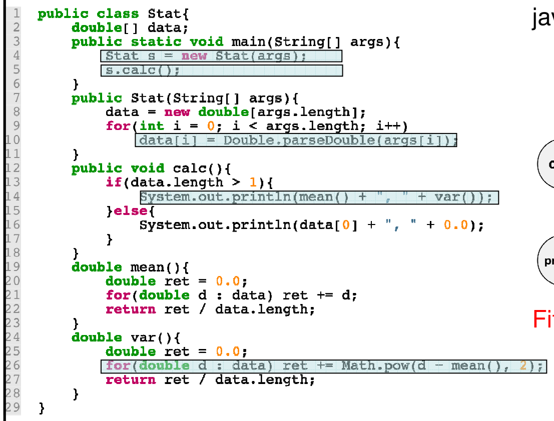
```
public class Stat{
    double[] data;
    public static void main(String[] args){
        Stat s = new Stat(args);
        s.calc();
    }
    public Stat(String[] args){
        data = new double[args.length];
        for(int i = 0; i < args.length; i++)
            data[i] = Double.parseDouble(args[i]);
    }
    public void calc(){
        if(data.length > 1){
            System.out.println(mean() + ", " + var());
        }else{
            System.out.println(data[0] + ", " + 0.0);
        }
    }
    double mean(){
        double ret = 0.0;
        for(double d : data) ret += d;
        return ret / data.length;
    }
    double var(){
        double ret = 0.0;
        for(double d : data) ret += Math.pow(d - mean(), 2);
        return ret / data.length;
    }
}
```

java Stat 1.0

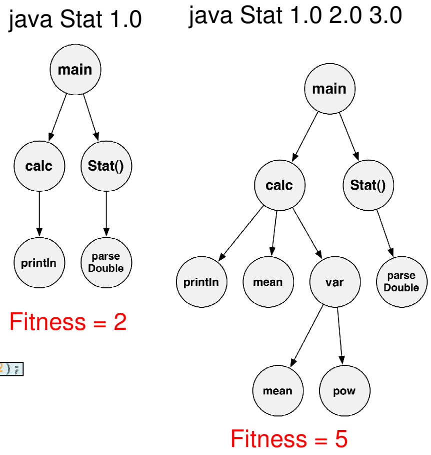


Fitness = 2

Objective Function: # Call Tree Leaves



```
public class Stat{
    double[] data;
    public static void main(String[] args){
        Stat s = new Stat(args);
        s.calc();
    }
    public Stat(String[] args){
        data = new double[args.length];
        for(int i = 0; i < args.length; i++)
            data[i] = Double.parseDouble(args[i]);
    }
    public void calc(){
        if(data.length > 1){
            System.out.println(mean() + ", " + var());
        }else{
            System.out.println(data[0] + ", " + 0.0);
        }
    }
    double mean(){
        double ret = 0.0;
        for(double d : data) ret += d;
        return ret / data.length;
    }
    double var(){
        double ret = 0.0;
        for(double d : data) ret += Math.pow(d - mean(), 2);
        return ret / data.length;
    }
}
```



Fitness = 2

Fitness = 5

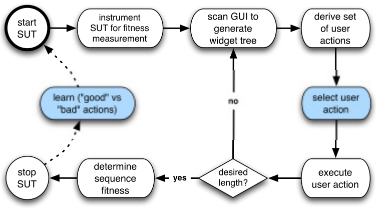
Optimization Algorithm: Ant Colony Optimization

- ACO has been successfully applied to sequence generation problems, e.g. TSP

- Seamless integration into the sequence generation process: sequences are constructed step by step

- Independent of mutation and crossover
  - Mutation and crossover are “destructive”
  - Mutation: may lead to infeasible sequences (not all actions are available in all contexts)
  - Crossover: difficult to define, because sequence parts cannot be arbitrarily exchanged, also leading to infeasible sequences

Optimization Algorithm: Ant Colony Optimization

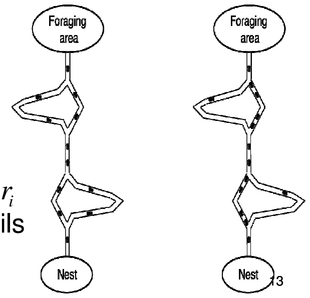


start SUT → instrument SUT for fitness measurement → scan GUI to generate widget tree → derive set of user actions → select user action → execute user action → desired length? → yes → determine sequence fitness → stop SUT → learn ("good" vs "bad" actions) → start SUT

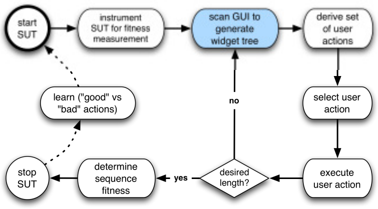
desired length? → no → scan GUI to generate widget tree

Idea:

- C = component set (here: set of actions)
- Each component c_i is associated with a pheromone value p_i
- Generate trails (sequences of user actions) by selecting components according to pheromone values p_i
- After each generation reward components that appear in “good” trails by increasing their pheromones
- Selection Rule: pseudo random proportional
- Pheromone Update Rule: $p_i' = p_i \cdot (1-\alpha) + \alpha \cdot r_i$ (α : learning rate, r_i : average fitness of the trails that c_i appeared in)



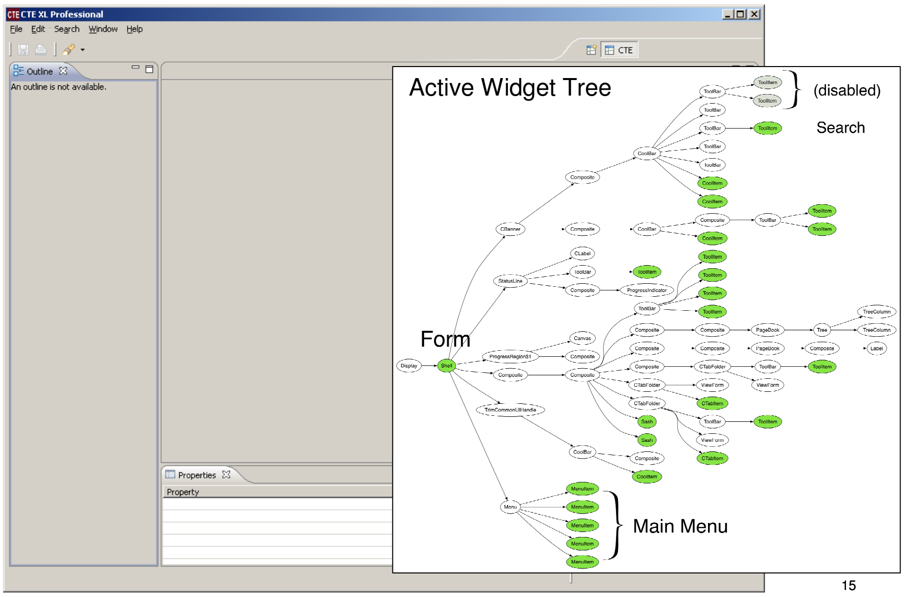
Test Environment: Scanning the GUI



start SUT → instrument SUT for fitness measurement → scan GUI to generate widget tree → derive set of user actions → select user action → execute user action → desired length? → yes → determine sequence fitness → stop SUT → learn ("good" vs "bad" actions) → start SUT

desired length? → no → scan GUI to generate widget tree

- In order to perform actions we first need to determine the visible control elements and their properties (e.g. to click a button: Is it enabled? Coordinates?)
- This information is saved in a *widget tree*, which is a hierarchical representation of the GUI and its control elements and properties
- State of the GUI changes ➔ widget tree needs to be constructed after each performed action



An outline is not available.

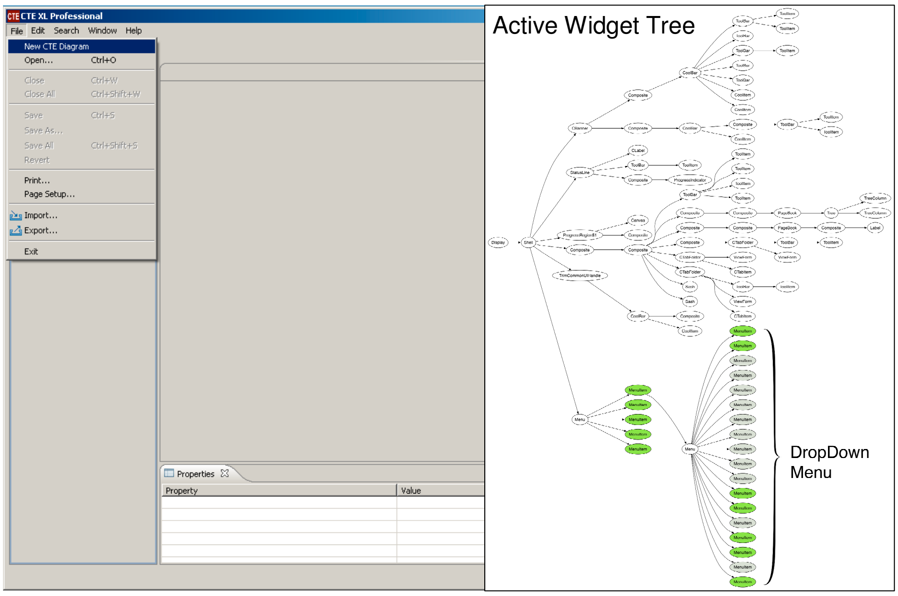
Active Widget Tree

(disabled)

Search

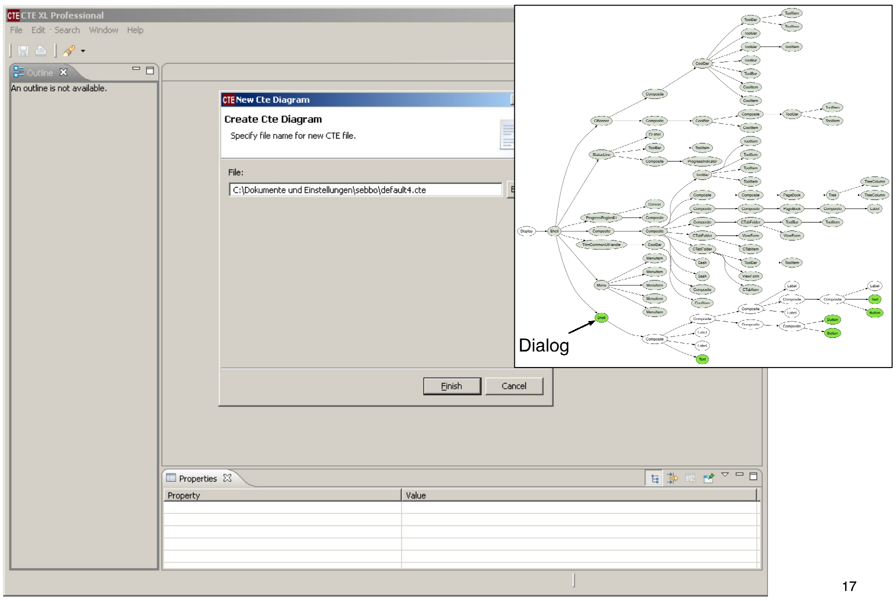
Form

Main Menu



Active Widget Tree

DropDown Menu



CTE XL Professional

An outline is not available.

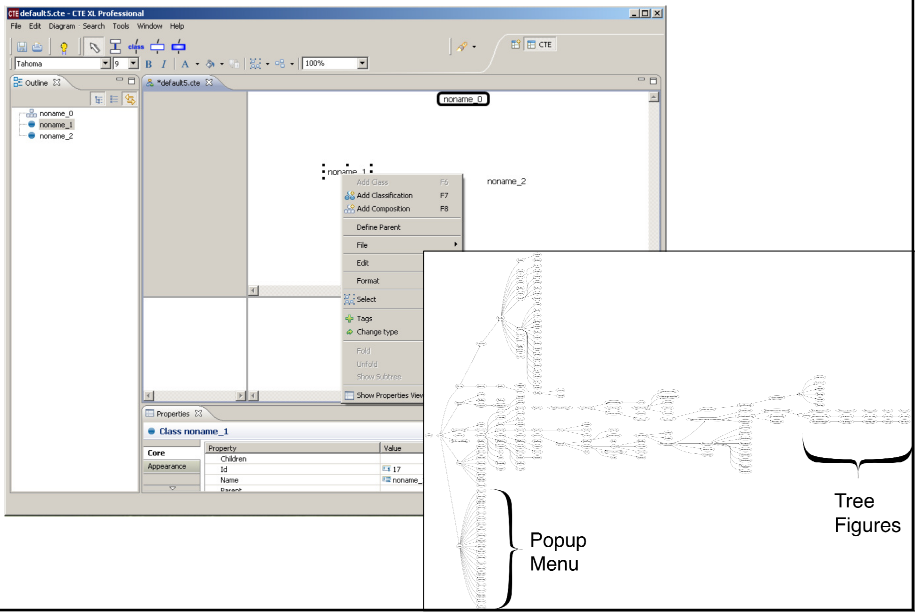
New Cte Diagram

Create Cte Diagram

Specify file name for new CTE file.

File: C:\Dokumente und Einstellungen\sebbo\default4.cte

Dialog



default5.cte - CTE XL Professional

Class noname_1

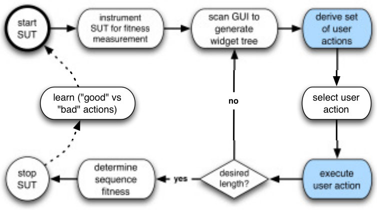
Property	Value
Children	
Id	17
Name	noname_
Parent	

Tree Figures

Popup Menu

Test Environment: Derive + Execute User Actions

- Based on the information in the widget tree, we can derive a set of "reasonable" actions



start SUT

instrument SUT for fitness measurement

scan GUI to generate widget tree

derive set of user actions

select user action

execute user action

desired length?

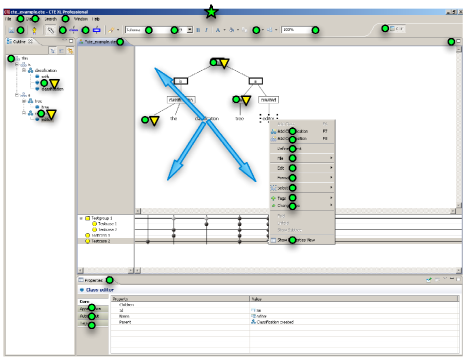
no

yes

determine sequence fitness

stop SUT

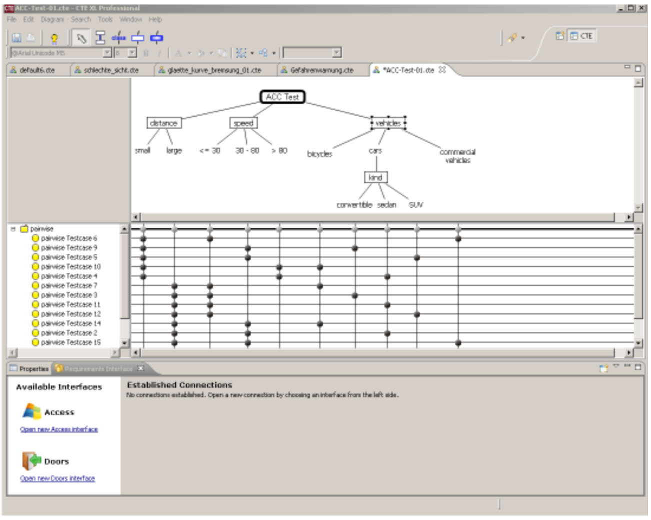
learn ("good" vs "bad" actions)



Simplified Set of Possible Actions

- After the optimization algorithm selected an action, it will be executed, e.g. click button, drag scrollbar, …

Experiments: Fully Automatic Testing of CTE XL Professional



No connections established. Open a new connection by choosing an interface from the left side.

Drawing area for classification trees

Combination table for test case specifications

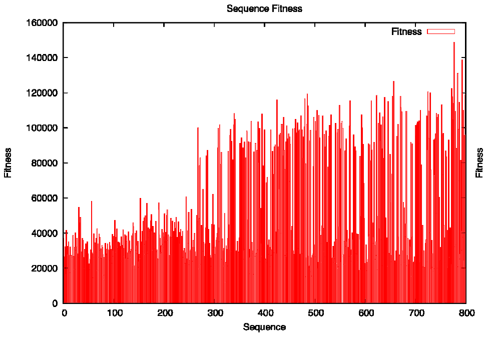
Panel for establishing RM / TM connections

Demo

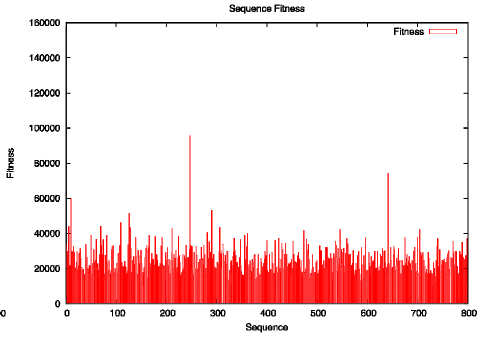
- Demo

Experiment: Results

ACO Run



Random Run



desc	length	popsize	generations	time per run	n	avg best	min	max
aco	30	10	80	~148 min	10	134729	113822	153978
rnd	30	10	80	~148 min	10	89670	71480	101861

Systematic vs. Generated Test Sequences

- Automatic Regression Test Suite for CTE XL Professional
  - 34 Sequences
  - average length: 14 actions (max: 64, min: 6)
  - average fitness: 61164 (max: 102031, min: 23466)

- Generated Sequences
  - 34 Sequences
  - length: 14
  - average fitness: 91369 (max: 111866, min: 58248)

Conclusion

- High demand for automatic GUI testing in industrial practice
- Typical B&M applications: CTE XL Professional, MESSINA (Eclipse RCP, SWT)

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

- Evaluation
  - Application of search successful
  - Initial experiments confirm better performance than random testing
  - First interesting results compared to functional testing

- Functional testing for logical errors difficult, because guidance to unknown logical errors hard to formalize
- Functional testing for exceptions, memory leaks, … possible

Outlook

- Generate entire test suites
- Possible improvement of algorithm to be more explorative
  - Prefer sequences with yet unexecuted actions
- Evaluate other objective functions
  - not only number of call tree leaves, but method diversity within call tree, or maximal call tree depth, etc.
  - Other criteria such as code coverage, temporal testing, ...
- Increase efficiency
  - Sequence generation is expensive ➔ parallelization of sequence execution
  - ACO good choice? ➔ disregards linkage among actions (context of actions not considered during pheromone update)
- Fault sensitivity of generated sequences ➔ empirical evaluation