# TAV 21

## A Data Flow Approach to Testing Object-Oriented Java-Programs

**Norbert Oster**
University of Erlangen-Nuremberg (Germany)
Department of Software Engineering (Informatik 11)

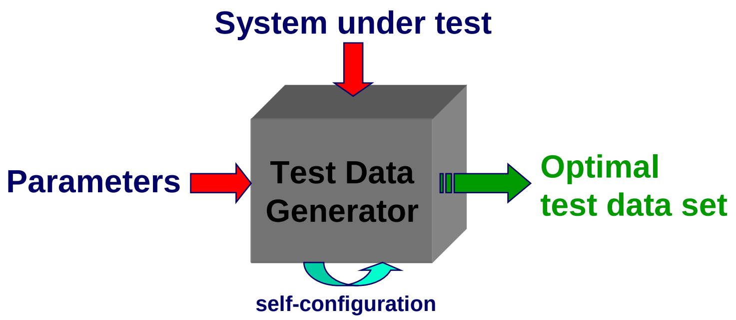


# Functional vs. structural testing

- **Functional testing**
  - test cases derived from specification (code seen as black-box)
  - focuses on expected/specified behaviour only
- **Structural testing**
  - considers unexpected functionality as a result of combinations of possible intended operations
    (based on code structure: code seen as white-box)
- **Effort**
  - existing tools usually just measure the coverage achieved
  - very few tools support tester with hints on how to increase coverage
  - fully automated test case generation based on deterministic static analysis is in general impossible
  - the result of each test run must be checked against specification

# Vision

- **Tester's desire:**




**optimal:**
high coverage with
small test set size



# Class structure of testing techniques

- **dynamic techniques**
  - **functional**
    - equivalence partitioning
    - cause effect graphing
  - **structural**
    - **control flow**
      - statement coverage
      - branch coverage
      - condition coverage
        - simple
        - condition/decision
        - minimal multiple
        - modified cond./decision
        - multiple
      - LCSAJ test
      - boundary interior test
      - structured path test
    - **data flow**
      - DU criteria
        - all defs
        - all p-uses
        - all c-uses
        - all p-uses/some c-uses
        - all c-uses/some p-uses
        - all uses
        - all DU-paths
      - required k-tuples
      - data context coverage
  - diversified
    - mutation test
    - back to back test
    - regression test

according to Liggesmeyer:
class structure of dynamic test techniques

# Original dataflow criteria by Rapps/Weyuker

- **Motivation**

  *Just as one would not feel confident about the correctness of a portion of a program which has never been executed, we believe that if the result of some computation has never been used, one has no reason to believe that the correct computation has been performed*

  Sandra Rapps / Elaine J. Weyuker (1982/1985)

- **Basis of Dataflow – Oriented Testing**
  - extended variant of control flow graph, annotated with data attributes
  - so-called data flow attributed control flow graph

- **Usage of Variables**
  - after memory allocation
  - until deletion

  three different types of operations can be carried out



# Dataflow relevant events

- **def** *definition*
  - associated to corresponding nodes of control flow graph containing variable defining (**not** declaring!) instruction
  - e.g. `x = f();`

- **c-use** *computational use*
  - associated to corresponding nodes of control flow graph containing computing instruction
  - e.g. `f(x + y);`

- **p-use** *predicative use*
  - associated to all edges of control flow graph going out from node containing predicate expression in order for branch coverage to be subsumed by most data-flow testing criteria
  - e.g. `if(x < y);`

# DU-Criteria

- **"all-defs" – criterion requires to execute**
  - **at least one** def-clear sub-path from **each** def to **at least one** reachable use

- **"all-p-uses" – criterion requires to execute**
  - **at least one** def-clear sub-path from **each** def to **each** reachable p-use
- **"all-c-uses" – criterion requires to execute**
  - **at least one** def-clear sub-path from **each** def to **each** reachable c-use

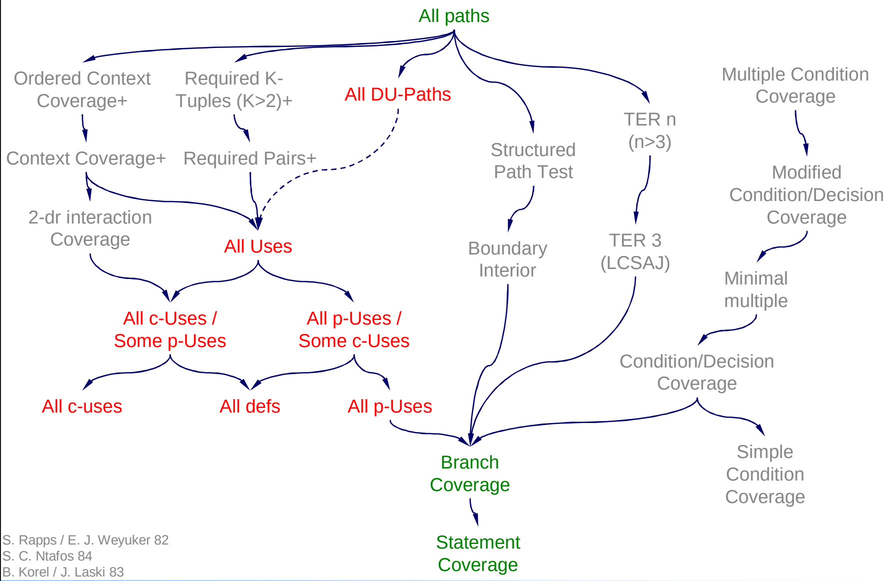
- **"all-p-uses/some-c-uses" – criterion requires to execute**
  - **at least one** def-clear sub-path from **each** def to **each** reachable p-use if a def does not reach a p-use, then to **at least one** reachable c-use
- **"all-c-uses/some-p-uses" – criterion requires to execute**
  - **at least one** def-clear sub-path from **each** def to **each** reachable c-use if a def does not reach a c-use, then to **at least one** reachable p-use

- **"all-uses" – criterion requires to execute**
  - **at least one** def-clear sub-path from **each** def to **each** reachable use

- **"all-du-paths" – criterion requires to execute**
  - **all** (feasible) **loop-free** def-clear sub-paths from **each** def to **each** reachable use



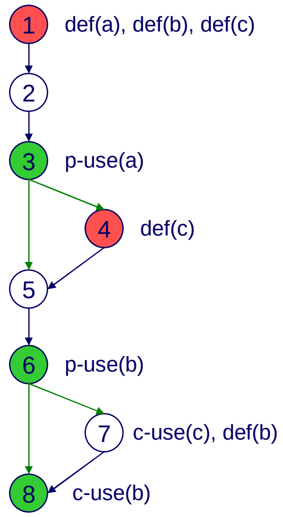
# Subsumption hierarchy




S. Rapps / E. J. Weyuker 82
S. C. Ntafos 84
B. Korel / J. Laski 83

# Why dataflow? – an example

```
public int f(int a, int b, String c) {
        …
        if (a > 0) {
                c = null;
        }
        …
        if (b < 0) {
                b = c.length();
        }
        return b;
}
```




statement-coverage:
**1-2-3-4-5-6-8** + 1-2-3-5-6-**7**-8 ☑PASS

branch-coverage:
1-2-**3-4**-5-**6-8** + 1-2-**3-5**-**6-7**-8 ☑PASS

e.g. all-uses (**requires pair 4/7**):
1-2-3-**4**-5-6-**7**-8 ☒FAIL



# Faults revealed by dataflow testing

- During static analysis phase:
  - dead code
  - syntactically endless loops
  - uses statically reachable without prior definition
- During dynamic execution phase
  - all-p-uses vs. branch coverage: not only each decision once, but additionally all possible data flows the decision might rely upon
  - definitions with unreachable uses (even if syntactically reachable): possible hint on logical program fault
  - different kinds of data-processing faults (e.g. anomalous conversion or type-inconsistent use) since all def/use-combinations must be exercised
  - in object-oriented software: state of an object and its change in terms of definitions and uses of variables representing the state

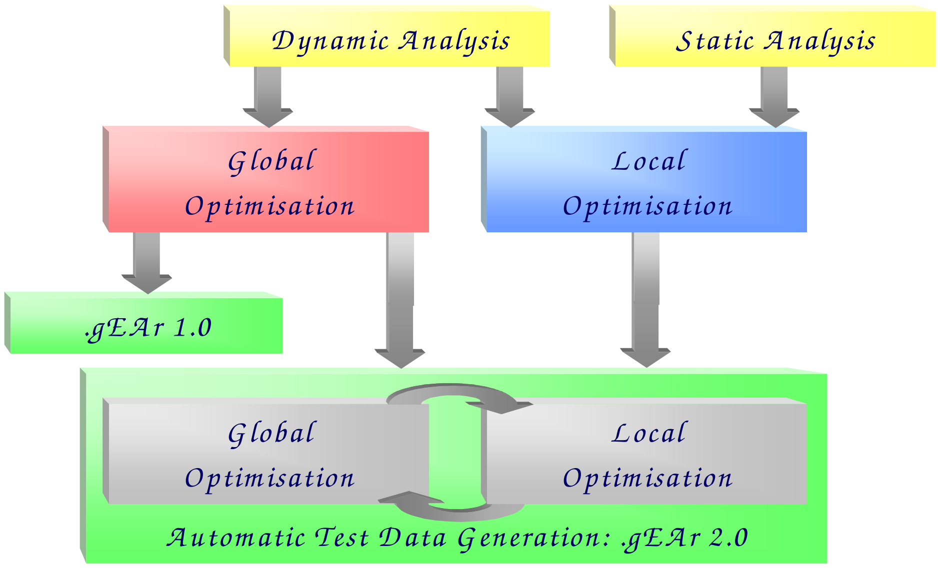
# Specifics of object-oriented Java Software

- “variables” must be distinguished:
  - static fields
  - local variables
  - (object) fields: same name in each instance
  - arrays: special “objects”
- multi-threading
- “pointer-aliasing” - equivalent
  - different variables might denote the same instance
- multiple hidden def/use-associations
  - due to field access through methods
- p-uses and c-uses hardly distinguishable
  - because predicates may contain method calls



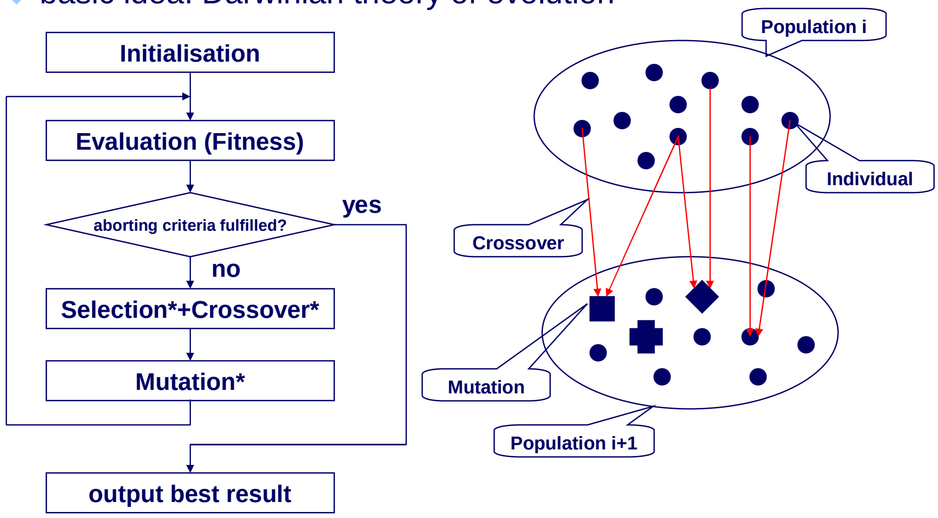
# .gEAr - Project

**D**ataflow **o**riented **t**est-case **g**eneration with **E**volutionary **A**lgo**r**ithms



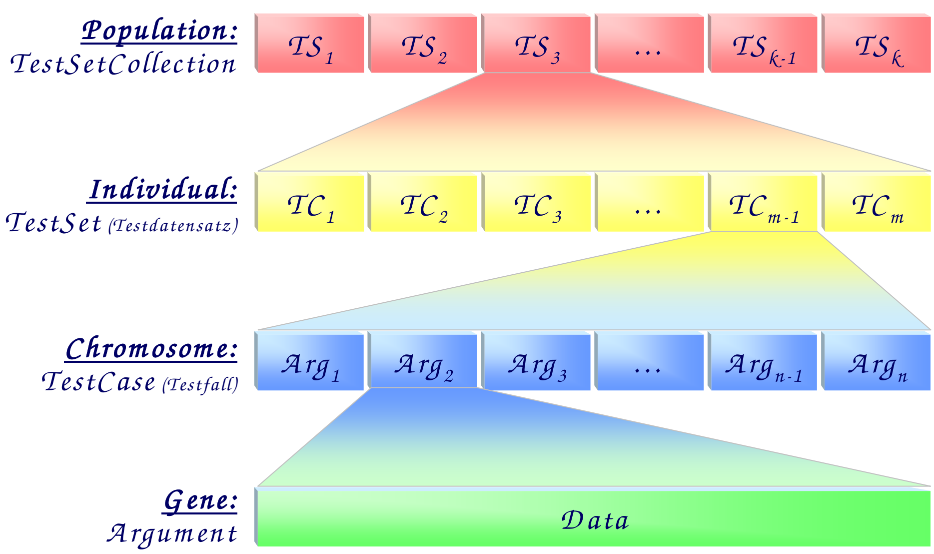
# Evolutionary Algorithms

- basic idea: Darwinian theory of evolution



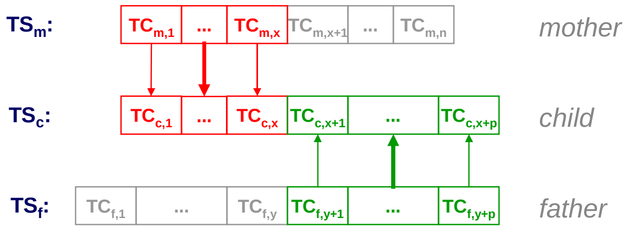



# Data Structure (global optimisation)



# Examples: Crossover, Mutation

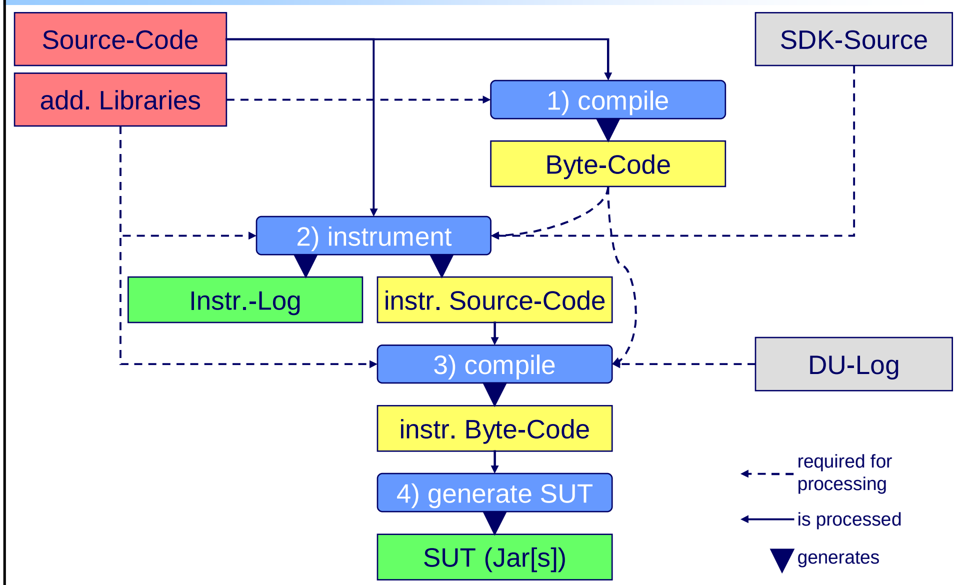
- **Crossover (example: single point)**



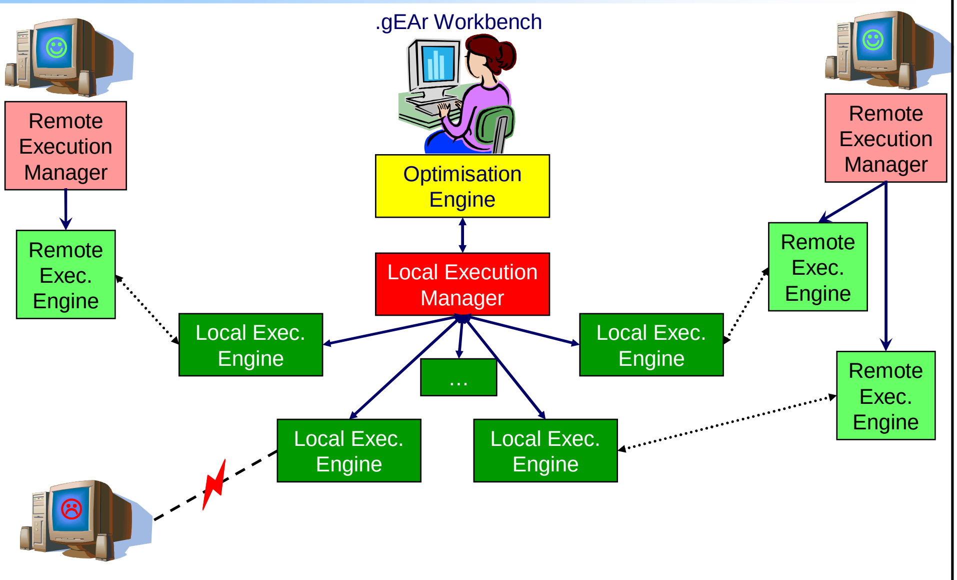

- **Mutation of a test set**
  - add a test case
  - remove a test case
  - mutate a test case:
    - add an argument
    - remove an argument
    - mutate an argument



# Processing of Source-Code

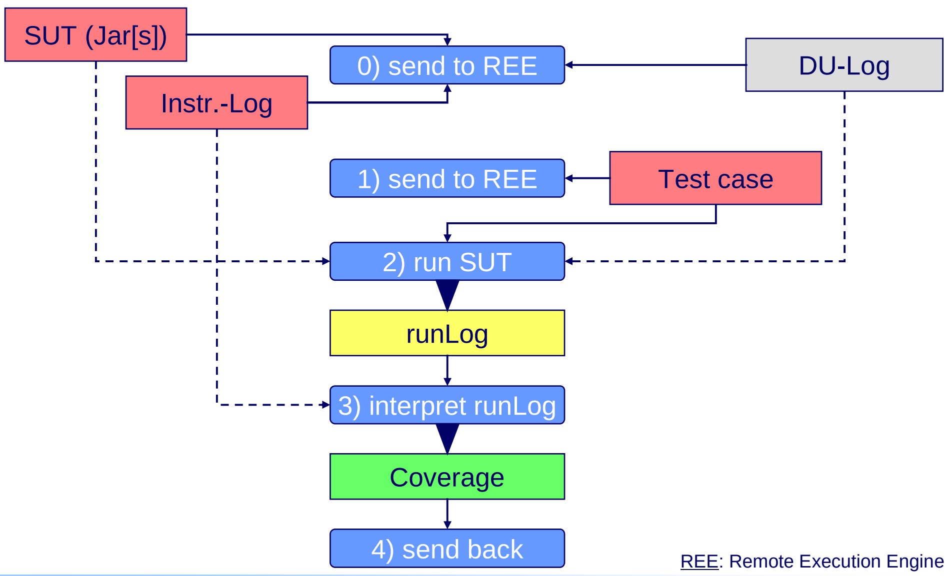


# Test Case Execution






# Execution of Test Cases




REE: Remote Execution Engine

# SUT - Interface

- Test case execution corresponds to running an “application” with test parameters (test case is therefore „String[] args“)
  - thus calling: public static void main(String[] args)
- Internal data types in .gEAr:
  - enumeration
  - string (any char or from a given set)
  - integer (long with adjustable range; covering byte, char, int, long)
  - floating point (double with adjustable range; covering float, double)
- Tester must specify in .gEAr:
  - the arguments in terms of the types above



# Example „OutputParameters“: Source Code

```
class OutputParameters {
        public static void main(String[] args) {
                try {
                        System.out.println("Parameters:");
                        for (int i=0; i< args.length; i++) {
                                System.out.println(" - <"+args[i]+">");
                        }
                        System.exit(0);
                } catch (Exception e) {
                        System.exit(1);
                }
        }
}
```

# Example: Instrumented Source Code

```
class OutputParameters implements InstanceId {
    public int ___instanceId = DULog.getNewInstanceId(0);
    public final synchronized int ___getInstanceId(){return ___instanceId;}
    public static void main(String[] args){
        DULog.enter(19);
        try{
            try{
                ((java.io.PrintStream)DULog.useStatic(1,System.out)).println
                    ((java.lang.String)DULog.cp(2,"Parameters:"));
                for(int i=(int)DULog.defLocal(3,0);
                        DULog.predResult(8,DULog.newPredicate(7),
                            (int)DULog.useLocal(4,i)
                            < DULog.useArrayLength(6,(java.lang.String[])DULog.useLocal(5,args)));
                        DULog.useDefLocal(9,i++))
                {((java.io.PrintStream)DULog.useStatic(10,System.out)).println
                        ((java.lang.String)DULog.cp(14," - <"+(java.lang.String)DULog.useArray(13,
                        (java.lang.String[])DULog.useLocal(11,args),DULog.useLocal(12,i))+">"));
                }
                System.exit((int)DULog.cp(15,0));
            } catch(Exception e){DULog.exceptHandlerCall(18);DULog.defLocal(16);
                System.exit((int)DULog.cp(17,1));
            }
        } finally{DULog.leave(20);}
    }
}
```

„DULog" short for „de.fau.cs.swe.sa.dynamicdataflowanalysis.rt.DULog"



# Example: Instrumentation Log

| | | | | |
|---|---|---|---|---|
| 1 | useStatic | public static final java.io.PrintStream java.lang.System.out | 4 | 31 |
| 2 | cp | public void java.io.PrintStream.println(java.lang.String) | 4 | 43 |
| 3 | defLocal | int OutputParameters.main([Ljava.lang.String;).i | 5 | 0 |
| 4 | useLocal | int OutputParameters.main([Ljava.lang.String;).i | 5 | 39 |
| 5 | useLocal | [Ljava.lang.String; OutputParameters.main([Ljava.lang.String;).args | 5 | 42 |
| 6 | useArrayLength | [Ljava.lang.String; OutputParameters.main([Ljava.lang.String;).args | 5 | 42 |
| 7 | newPredicate | - | 5 | 25 |
| 8 | predResult | - | 5 | 25 |
| 9 | useDefLocal | int OutputParameters.main([Ljava.lang.String;).i | 5 | 55 |
| a | useStatic | public static final java.io.PrintStream java.lang.System.out | 6 | 39 |
| b | useLocal | [Ljava.lang.String; OutputParameters.main([Ljava.lang.String;).args | 6 | 59 |
| c | useLocal | int OutputParameters.main([Ljava.lang.String;).i | 6 | 64 |
| d | useArray | [Ljava.lang.String; OutputParameters.main([Ljava.lang.String;).args | 6 | 59 |
| e | cp | public void java.io.PrintStream.println(java.lang.String) | 6 | 51 |
| f | cp | public static void java.lang.System.exit(int) | 8 | 36 |
| 10 | defLocal | java.lang.Exception e | 9 | 0 |
| 11 | cp | public static void java.lang.System.exit(int) | 10 | 36 |
| 12 | exceptHandlerCall | - | 9 | 19 |
| 13 | enter | public static void OutputParameters.main(java.lang.String[]) <br> **PARA:** [Ljava.lang.String; OutputParameters.main([Ljava.lang.String;).args | 2 | 0 |
| 14 | leave | public static void OutputParameters.main(java.lang.String[]) | 2 | 0 |

# Log-Events

CallPoint
**DefineArray**
**DefineField**
**DefineLocalVariable**
**DefineStaticVariable**
EarlyConstructorEnter
EnterClassInitialisation
EnterConstructor
EnterInstanceInitialisation
EnterMethod
ExceptionHandlerCall
LeaveClassInitialisation
LeaveConstructor
LeaveInstanceInitialisation
LeaveMethod
NewArray
NewCall
NewCallCompleted
NewPredicate
NewSwitchPredicate
PredicateResult
SwitchPredicateEquivalent
SwitchPredicateResult
**UseArray**
**UseArrayLength**
**UseField**
**UseLocalVariable**
**UseStaticVariable**
**UseDefineArray**
**UseDefineField**
**UseDefineLocalVariable**
**UseDefineStaticVariable**



# Example: Run-Log (application executed with 2 parameters)

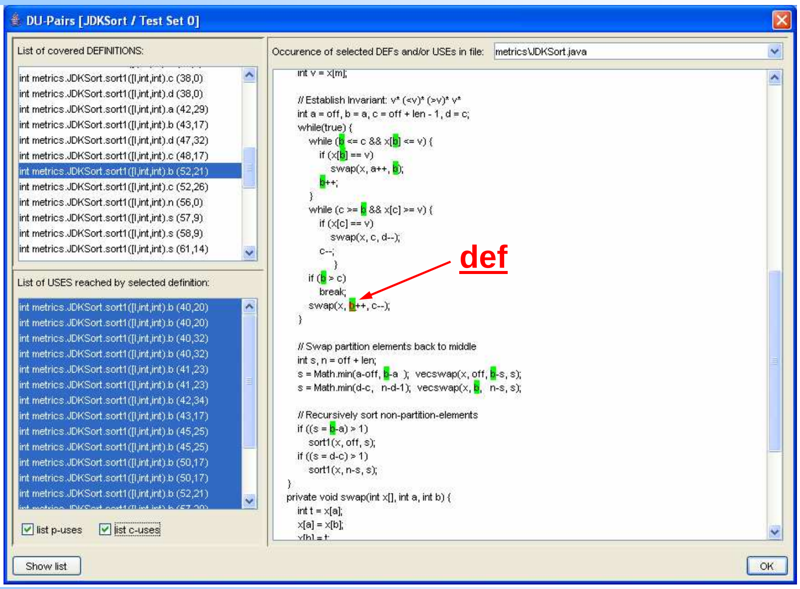
0-NewThread
1-EnterMethod: "OutputParameters.main(java.lang.String[])"
2-DefineLocalVariable: "OutputParameters.main([Ljava.lang.String;).args"
3-UseStaticVariable: "java.lang.System.out"
4-CallPoint: "java.io.PrintStream.println(java.lang.String)" (virtual)
**5-DefineLocalVariable: "OutputParameters.main([Ljava.lang.String;).i"**
**6-NewPredicate**
**7-UseLocalVariable: "OutputParameters.main([Ljava.lang.String;).i"**
8-UseLocalVariable: "OutputParameters.main([Ljava.lang.String;).args"
9-NewInstance
10-UseArrayLength: "OutputParameters.main([Ljava.lang.String;).args.length"
**11-PredicateResult [true]**
[...]
**17-UseDefineLocalVariable: "OutputParameters.main([Ljava.lang.String;).i"**
[...]
**29-NewPredicate**
30-UseLocalVariable: "OutputParameters.main([Ljava.lang.String;).i"
31-UseLocalVariable: "OutputParameters.main([Ljava.lang.String;).args"
32-UseArrayLength: "OutputParameters.main([Ljava.lang.String;).args.length"
**33-PredicateResult [false]**
34-CallPoint: "java.lang.System.exit(int)" (virtual)
35-EndOfLog

# Example: DU-pairs

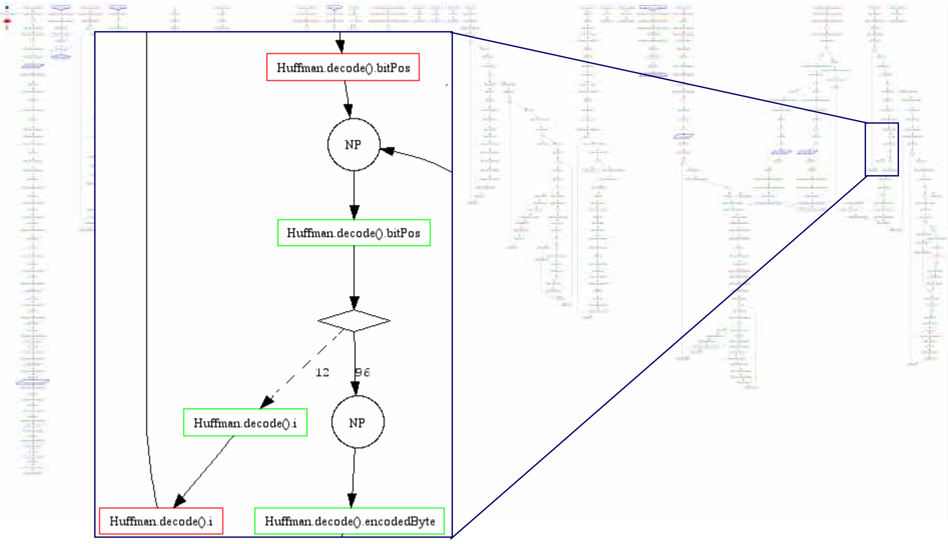
[DefineLocalVariable]{3}{@(5;0)} :int OutputParameters.main([Ljava.lang.String;).i
=> [UseLocalVariable]{4}{@(5;39)} :int OutputParameters.main([Ljava.lang.String;).i
=> [UseLocalVariable]{12}{@(6;64)} :int OutputParameters.main([Ljava.lang.String;).i
=> [UseDefineLocalVariable]{9}{@(5;55)} :int OutputParameters.main([Ljava.lang.String;).i
#3
[Define]{-10}{<IMPLICIT>@(0;0)} :public static final java.io.PrintStream java.lang.System.out
=> [UseStaticVariable]{1}{@(4;31)} :public static final java.io.PrintStream java.lang.System.out
=> [UseStaticVariable]{10}{@(6;39)} :public static final java.io.PrintStream java.lang.System.out
#2
[DefineLocalVariable]{19P0}{@(2;0)} :[Ljava.lang.String; OutputParameters.main([Ljava.lang.String;).args
=> [UseArray]{13}{@(6;59)} :[Ljava.lang.String; OutputParameters.main([Ljava.lang.String;).args
=> [UseLocalVariable]{11}{@(6;59)} :[Ljava.lang.String; OutputParameters.main([Ljava.lang.String;).args
=> [UseArrayLength]{6}{@(5;42)} :[Ljava.lang.String; OutputParameters.main([Ljava.lang.String;).args.length
=> [UseLocalVariable]{5}{@(5;42)} :[Ljava.lang.String; OutputParameters.main([Ljava.lang.String;).args
#4
[UseDefineLocalVariable]{9}{@(5;55)} :int OutputParameters.main([Ljava.lang.String;).i
=> [UseLocalVariable]{4}{@(5;39)} :int OutputParameters.main([Ljava.lang.String;).i
=> [UseLocalVariable]{12}{@(6;64)} :int OutputParameters.main([Ljava.lang.String;).i
=> [UseDefineLocalVariable]{9}{@(5;55)} :int OutputParameters.main([Ljava.lang.String;).i
#3
TOTAL:12



# Covered DU-pair browser



# Covered dataflow-annotated CFG






# First Experimental Results

| **Project** | **Size** | **Max. number of identified context-free DU-Pairs** |
|---|---|---|
| The Towers of Hanoi | 38 LOC<br>1 class (1.272 bytes) | 42 |
| Dijkstra's shortest path | 102 + 57 LOC<br>2 classes (4.589 bytes) | 213 |
| JDK integer-array sort | 82 LOC<br>1 class (2.639 bytes) | 315 |
| Huffman encoding | 240 + 78 LOC<br>2 classes (10.089 bytes) | 368 |

# First Experimental Results* (all-uses)

| Project | Coverage<br>Average<br>Min / Max | Test set size<br>Average<br>Min / Max | Generation<br>Average<br>Min / Max |
|---|---|---|---|
| The Towers of Hanoi | 42<br>42 / 42 | 2<br>2 / 2 | 9.2<br>5 / 15 |
| Dijkstra's shortest path | 213<br>213 / 213 | 2<br>2 / 2 | 29<br>10 / 89 |
| JDK integer-array sort | 315<br>315 / 315 | 2<br>2 / 2 | 117<br>5 / 348 |
| Huffman encoding | 367.6<br>366 / 368 | 3.6<br>3 / 5 | 155.8<br>10 / 540 |

* average over 5 runs, 1000 generations each, with default setup (self-adaptive multi-objective aggregation)
coverage weight: 1 vs. test set size weight: 0.05



# Experimental Results*

Comparison for project "JDK integer-array sort"

| Population size | Coverage<br>Average<br>Min / Max | Test set size<br>Average<br>Min / Max | Generation<br>Average<br>Min / Max |
|---|---|---|---|
| 40 | 315<br>315 / 315 | 2<br>2 / 2 | 117<br>5 / 348 |
| 30 | 315<br>315 / 315 | 2.6<br>2 / 4 | 57.8<br>24 / 104 |

* average over 5 runs, 1000 generations each, with self-adaptive multi-objective aggregation
coverage weight: 1 vs. test set size weight: 0.05

# Experimental Results*

| **Project** CPU-time** | **Coverage** Average Min / Max | **Test set size** Average Min / Max | **Generation** Average Min / Max |
|---|---|---|---|
| The Towers of Hanoi ~ 1:20 | 42<br>42 / 42 | 2<br>2 / 2 | 10.4<br>3 / 20 |
| Dijkstra's shortest path ~ 5:20 | 213<br>213 / 213 | 2<br>2 / 2 | 63.2<br>25 / 165 |
| JDK integer-array sort ~ 6:58 | 315<br>315 / 315 | 2<br>2 / 2 | 79.6<br>15 / 264 |
| Huffman encoding ~ 9:14 | 368<br>368 / 368 | 3<br>3 / 3 | 64.2<br>39 / 96 |

* average over 5 runs: multi-objective aggregation (mutation rate: 25%)
coverage weight: 1 vs. test set size weight: 0.05
** resources on workbench host in min:sec (for 200 generations; test case execution parallelized on 6 PCs)



# Summary

- Motivation:
  - functional testing covers only a subset of the "true functionality" provided by a given code (neglecting Trojan horse behaviour)
  - structural (especially dataflow) testing increases the chance of finding abovementioned faults
- State-of-the-art in practice
  - expensive test data generation
  - expensive check of test results because of large test sets
- Proposed solution by means of .gEAr:
  - maximise the coverage according to a given testing strategy
  - minimise the number of test cases (=> reduced effort)
  - achieve both goals by fully automated test set generation