On the Predictability of Random Tests for Object-Oriented Software

Ilinca Ciupa Alexander Pretschner Andreas Leitner Manuel Oriol Bertrand Meyer

Department of Computer Science, ETH Zurich, Switzerland

Group 15:
Allan Fernandes - 97281
João Salgueiro - 100740
Miguel Prazeres - 95649

Software Testing and Validation

Prof: João Pereira

Summary

Abstract

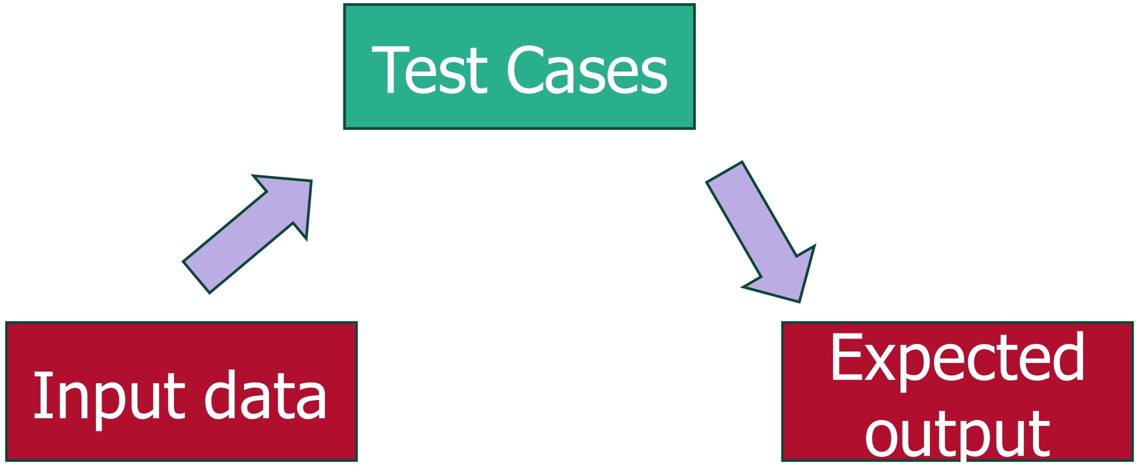
- Intuition, common sense

  C1 – first run of random testing duration x' – number of detected faults: 1000
  C1 – 2nd run of random testing duration x' – number of detected faults: ?

- Empirical study
- 1215 hours of randomly testing
- 27 Eiffel classes
- 30 seeds of the random number generator
- Over 6 million failures triggered
- Relative number of faults detected
- How quickly does random testing find faults?

- Predictability
- Random
- Tests
- Object-Oriented Software

1 - Introduction



- Expected output depends on a specific input, it cannot be generated at random
- Provide expected output at different abstraction levels

- “no exception is thrown.”
- solves the oracle problem

- For trivial problems, **what becomes of the random test case generation ?**
  - A simple correspondence between elements of the input domain and “no exception” as expected output

- **what about more complex objects?**
  - generating objects to use as input to a method is a non-trivial task

1 - Introduction

Eiffel Programs

What makes this class of object-oriented software interesting to the random testing problem?

- **DbC –** Contracts
  - Assertions
  - Preconditions
  - Postconditions
  - Class Invariants

application.e

```
class
    BAKERY

feature

    number_of_cakes : INTEGER
        -- A variable containing an integer

    buy_cakes (amount : INTEGER)
        require
            positive_amount: amount > 0 -- Check that amount is a positive number

        do
            number_of_cakes = number_of_cakes - amount

        ensure
            amount_reduced: number_of_cakes = old number_of_cakes - amount
            -- Check that the number of available cakes has decreased correctly
    end
end
```

- employed to help ensure program correctness without sacrificing efficiency

NOTE:

- Randomly generating test cases for Eiffel programs:
  - generating input objects for a method to be tested
  - adding the postcondition as the expected output

1 - Introduction

Problem:

- How predictable is random testing?
- What are the consequences?
- how would this technique be best used?
- Should testers constantly run it in the background?

Solution:

- AutoTest
- 27 classes from a widely used Eiffel library
- 90 minutes each class
- Repeat process 30 times, different seeds
- Testing time: 1215 hours
- Over 6 million triggered failures

2 - Random Tests for OO Programs

Goal: describe the conceptual framework for randomly testing OO programs and justify the choice of parameters measured in the experiments.

Relevant notions:

- Test Cases
- Faults
- Failure

- Test case generation algorithm – Example
- Stopping criterion for testing

2 - Random Tests for OO Programs

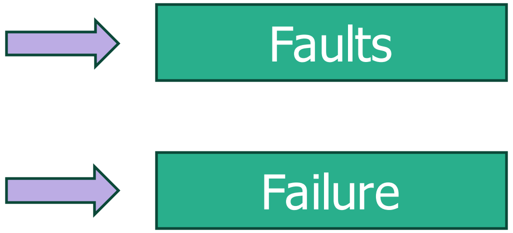
Relevant notions:

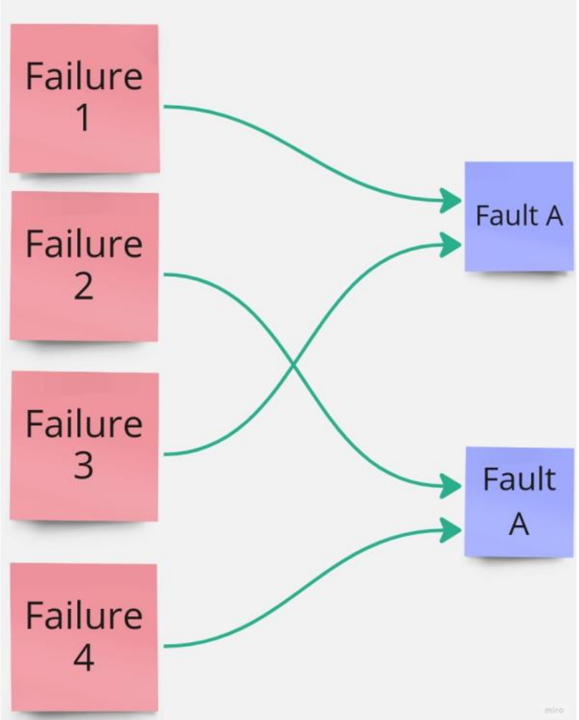
Test Cases

- Sub-method invocations
  - Strict sense
- Inputs
  - Bit field
  - Constructor Invocation
  - Mock objects with predefined behaviors
- Oracle
  - Different levels of abstraction

2 - Random Tests for OO Programs

Relevant notions:





- Failure = Actual Behavior – Intended Behavior
- Fault: wrong piece of code that triggers failure
- The same fault may trigger arbitrarily many failures
- Contract violations, what about them ?
  - Immediate precondition violation
- Mapping failures to faults

2 - Random Tests for OO Programs

Test case generation algorithm

Main loop of the testing strategy:

```
write_tests (timeout):
  from
      initialize_pool
  until timeout
  loop
     m := choose (methods_under_test ())
     write_test_for_method (m)
  end
```

Example test case generated by AutoTest:

```
1 create {STRING} v1.make_empty
2 create {BANK_ACCOUNT} v2.make (v1)
3 v3 := 452719
4 v2.deposit (v3)
5 v4 := Void
6 v2.transfer (v4, v3)
  ...
```

2 - Random Tests for OO Programs

Test case generation algorithm

write_test_for_method (m):
  ops := <>
  foreach *ot* **from** (<*type* (m)> ... *param_types* (m)) **do**
    if P (*gen_new*) **then**
      write_creation (*ot*)
    end
    ops := *ops* ... *choose* (*conforming_objects* (*ot*))
  end
  write_invoke_instruction (*m*, *ops*)
end

2 - Random Tests for OO Programs

Test case generation algorithm

```
write_creation (t):
  if is_basic_type (t) then
    if P (gen_basic_rand) then
      write_assignment (random_basic_object (t))
    else
      write_assignment (choose ( predefined_objects (t))
    end
  else
    c := choose (cons (t))
    ops := <>
    foreach ot from (param_types (c)) do
      if P(gen_new) then
          write_creation (ot)
      end
      ops := ops ... choose (conforming_objects (ot))
    end
    write_creation_instruction (c, ops)
  end
end
```

- The input generation algorithm uses two parameters:
  - *P (gen new)*
  - *P (gen basic rand)*

- *P (gen new)* = 0.25 and *P (gen basic rand)* = 0.25

2 - Random Tests for OO Programs

Stopping criterion

- Time

- Conceptual problem with the number of executed test cases

application.e

```
class
    EXAMPLE_CLASS
create
    make

feature
    make
        local
            value: INTEGER
        do
            value := calculate_value -- Invoking the sub-method "calculate_value"
            io.put_string("The calculated value is: ")
            io.put_integer(value)
        end

    calculate_value: INTEGER
        do
            -- Perform some calculations
            Result := 42
        end
end
```

3 - Experimental Setup

Input Generation Algorithm
Testing time
Classes to Test

IN

out

Minimized failure-reproducing examples
Statistics about session

Consists of

Driver:

Object creation;

Method invocation;

Tests all class methods, keeping statistics and fairness

Restarts Interpreter in case of failure (rebuilding new object pool)

Interpreter:

Executes tests

3 - Experimental Setup

27 classes x 30 sessions x 90 minutes = 50.6 days
Each session takes a different seed for RNG

Classes from EiffelBase 5.6
(some classes inherit from other classes)

AutoTest can report faults on other classes than CUT because:

[1]CUT calls a faulty **method** from other class

OR

[2]CUT calls a faulty **constructor** from other class

The researchers opted to count [1] fails, but to leave [2] out.
Considering that even though [1] faults are not CUT responsibility, but the supplier's, the user of CUT should be warned.
[2] is considered out of the scope of CUT

Table 1. Metrics of the tested classes

	Average	Median	Minimum	Maximum
LoCs	477.67	366	62	2600
Methods	108.37	111	37	171
Attributes	6.26	6	1	16
Contracts	111.07	98	53	296
Faults	39.52	38	0	94

4- Discussion

Analysis

How predictable is random testing? *(influence of the seed that generates pseudo-random numbers)*

Two kinds of predictability:
-Predictability of the **number** of detected distinct faults
AND
-Predictability of the different **kinds** of detected faults

4- Discussion

Observations

Range of detected faults over ALL classes: 0-94
Median=38
Std Dev=28
Two classes with no faults CHARACTER REF and STRING SEARCHER

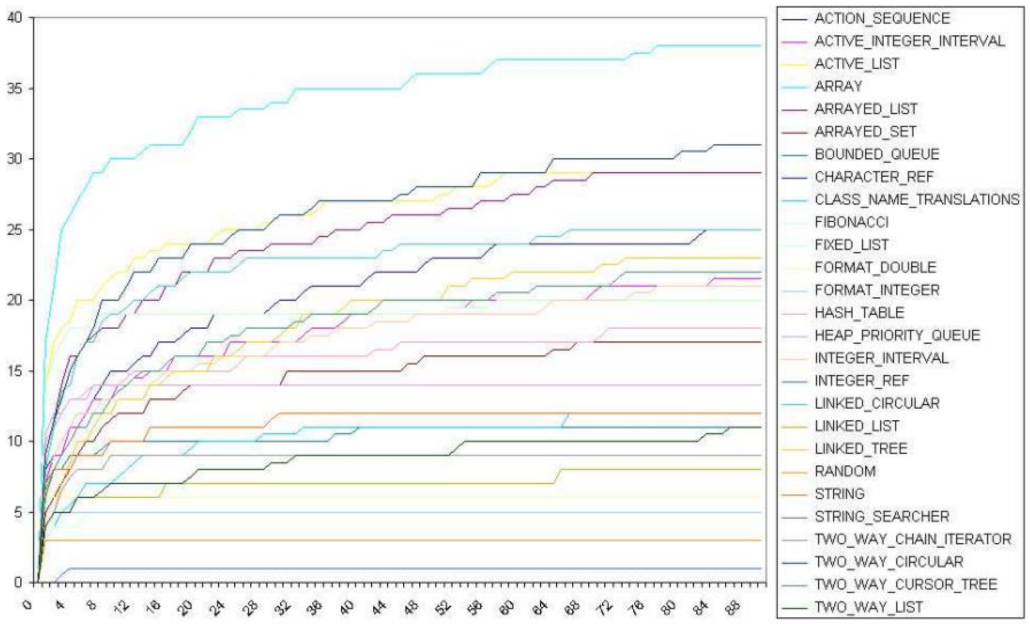


Figure 4. Medians of the absolute numbers of faults found in each class

Nº of faults by run /
Total number of faults for that class

30% after first 10 min
38% in the end, 90 min

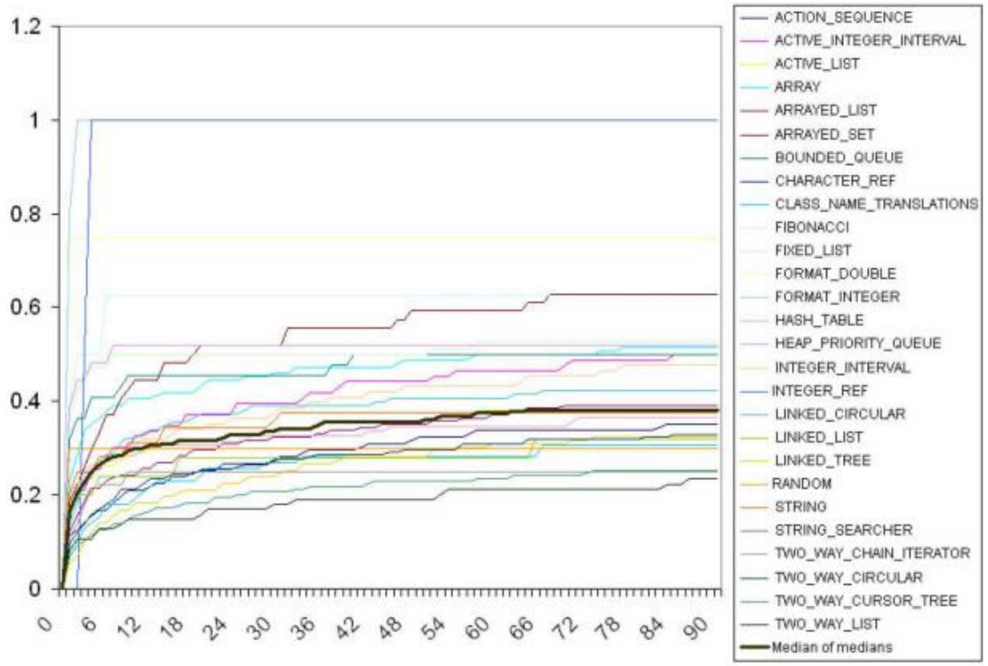


Figure 5. Medians of the normalized numbers of faults found in each class; their median

4- Discussion

Predictability of the **number** of detected distinct faults

Std Dev between 2%-6%

For the aggregate results, in **black**:
Stdev of stdevs: 4% -> 2% in the first 15 min
Med of stdevs: 3% -> 1.5% in the first 10 min

Rather predictable for t < 15m
Very predictable for t > 15m

Somewhat counter-intuitively:
*In terms of the relative number of detected faults, random testing In OO programs is **indeed predictable**.*

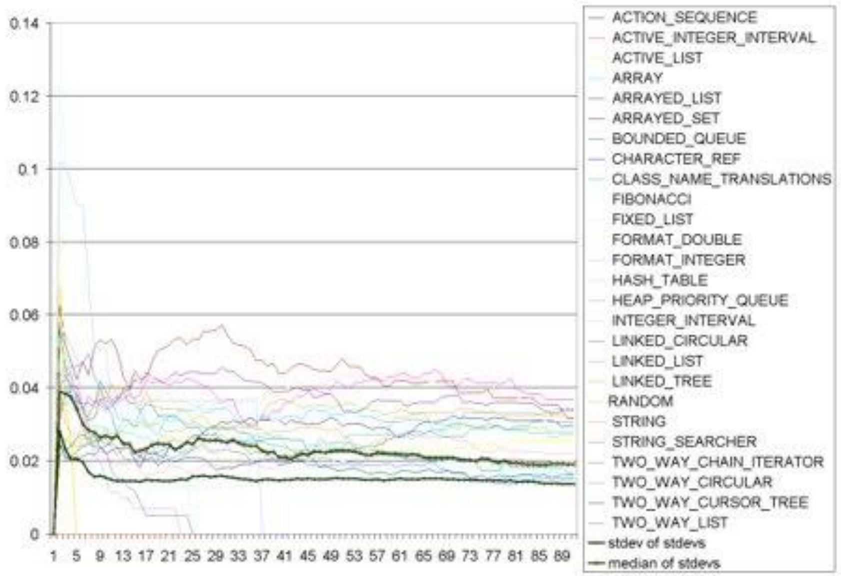


Figure 6. Standard deviations of the normalized numbers of faults found in each class; their median and standard deviation

4- Discussion

Predictability of the **kind** of detected distinct faults

An **identical relative number** of faults **doesn't necessarily mean** the **same faults** are detected.

If that was the case, then the normalized number should get close to 1, but **median is 38%**.

*In terms of the kind of detected faults, random testing In OO programs is **rather unpredictable**.*

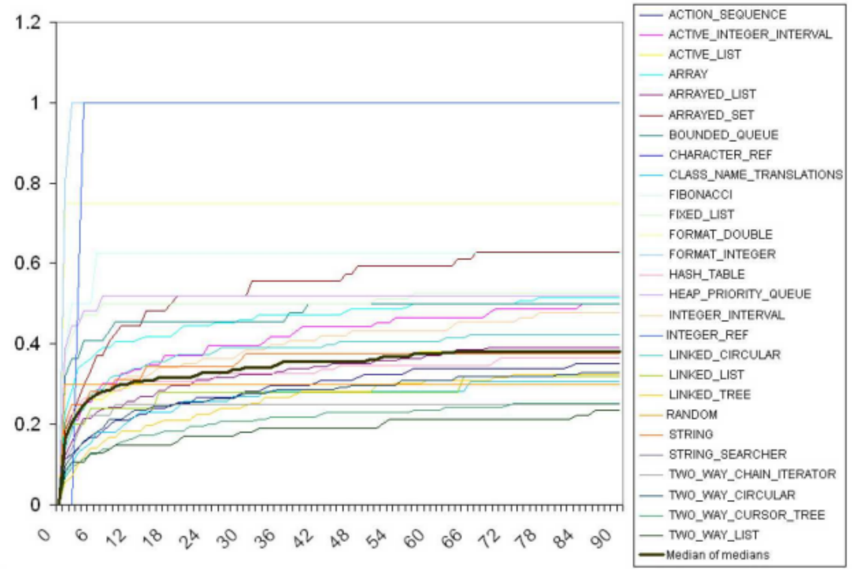


Figure 5. Medians of the normalized numbers of faults found in each class; their median

4- Discussion

BONUS DISCUSSION:

In 24 out of 25 CUT at least **1 session** found **1 fault** in the **1st second**.

New Question:
In terms of the efficiency of our technology, is there a difference between long and short tests?
Does it make a difference if we test one class once for ninety minutes or thirty times for three minutes?

This led to a change of perspective, testing **any class**:

-**90 min, never change seed**
-**90 min**, **change** seed every **3 min (30x)**
-**30 min**, **change** seed every **1 min (30x)**

Obtained the results...

4- Discussion

BONUS DISCUSSION:

-90m of testing changing the seed every 3m yields considerably **better results than keeping** the same **seed**

-30*3 detects more faults than 30*1
However, **takes three times longer**, and the normalized **number of faults is not 3 times bigger**

Apparently, short tests are more effective than longer Tests.*

**Because of interpreter restarts we cannot directly hypothesize on length of test cases –future work–*

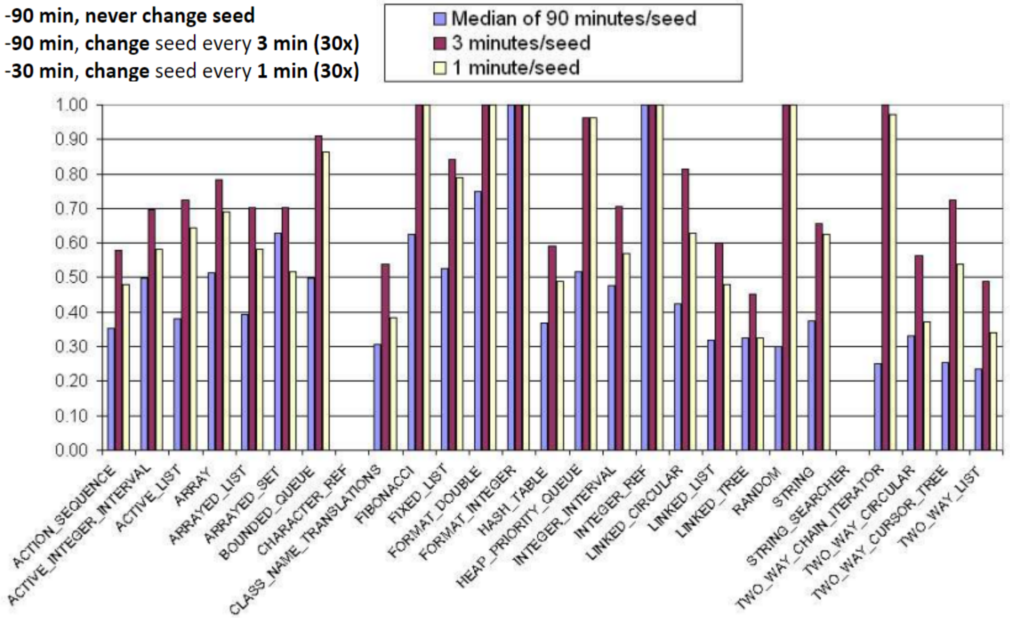


Figure 7. Cumulated normalized numbers of faults after 30*3 and 30*1 minutes; median normalized number of faults after 90 minutes

4- Discussion

Threats to the validity of generalization for these results:

-The representativeness of isolated classes in EiffelBase 5.6 in relation to the **whole universe of OO software is limited**;
-Interpreter restarts reinitialize the object pool, the restarts occur at intervals from 1m to >1h. **Some sessions reach rather complex object structures, others don't**;
-Random testing combined with automated Oracle can **miss many more faults**, and these numbers were not compared with faults detected manually or using other strategies;

and more:
-AutoTest implements only one of several possible algorithms for generating objects randomly;
-Most of the 1st second faults in the Bonus Discussion were found in constructors while using extreme values for Integers or Void for reference-type arguments. It isn't correct to generalize to classes without these arguments.

5. Related Work

Theoretical studies:

On Random and Partition Testing

Simeon Ntafos

University of Texas at Dallas
Computer Science Program
Richardson, TX 75083, USA
ntafos@utdallas.edu

Abstract
There have been many comparisons of random and partition testing. Proportional partition testing has been suggested as the optimum way to perform partition testing. In this paper we show that this might not be so and discuss some of the problems with previous studies. We look at the expected cost of failures as a way to evaluate the effectiveness of testing strategies and use it to compare random testing, uniform partition testing and proportional partition testing. Also, we introduce partition testing strategies that try to take the cost of failures into account and present some results on their effectiveness.

Keywords
Program Testing, Random Testing, Partition Testing.

likely to encounter a certain type of error. This allows partition testing to model a variety of knowledge-based strategies (although it is not, by any means, a perfect model).

In random testing, test cases are selected randomly from the input domain of the program. There is a body of research that appears to suggest that knowledge-based testing strategies do not perform significantly better than an equal number of randomly selected test cases unless they test low probability subdomains that have high failure rates [10,13]. An analytical comparison of random and partition testing [21] reports conditions under which random and partition testing outperform each other. One of the observations in [21] was that partition testing is certain to perform better than random testing if all the subdomains have equal size. This was followed up by others [2-6] who developed more conditions that make partition testing better than random

Found that partition testing can perform better but is more expensive.

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 17, NO. 7, JULY 1991

Analyzing Partition Testing Strategies

Elaine J. Weyuker and Bingchiang Jeng

Abstract—**In this paper, partition testing strategies are assessed analytically. An investigation of what conditions affect the efficacy of partition testing is performed, and comparisons of the fault detection capabilities of partition testing and random testing are made. The effects of subdomain modifications on partition testing's ability to detect faults are also studied.**

Index Terms—**Partition testing, random testing, software testing.**

I. Partition Testing

THE term "partition testing," in its broadest sense, refers to a very general family of testing strategies. The primary characteristic is that the program's input domain is divided into subsets, with the tester selecting one or more element

a given test case will generally cause many statements to be executed, it is a member of the subdomain determined by each such statement. *Branch testing* also divides the domain into non-disjoint subdomains. *Path testing* requires that sufficient test data be selected so that every path from the program's entry statement to the program's exit statement is traversed at least once. This strategy *does* divide the input domain into disjoint classes since a given test case causes exactly one path to be traversed.

Rapps and Weyuker [9], [10] introduced a family of testing strategies, known as the data flow testing criteria, which require the exercising of path segments determined by combinations of variable definitions and variable uses. For each of these criteria, the input domain is divided so that there is

Found that the more random tests you perform, the less effective they get, and the opposite for partition testing.

5. Related Work

Practical application:

Eclat 1.1

Downloading and Installing Eclat

Eclat requires Java 1.5.

Eclat: testing tool

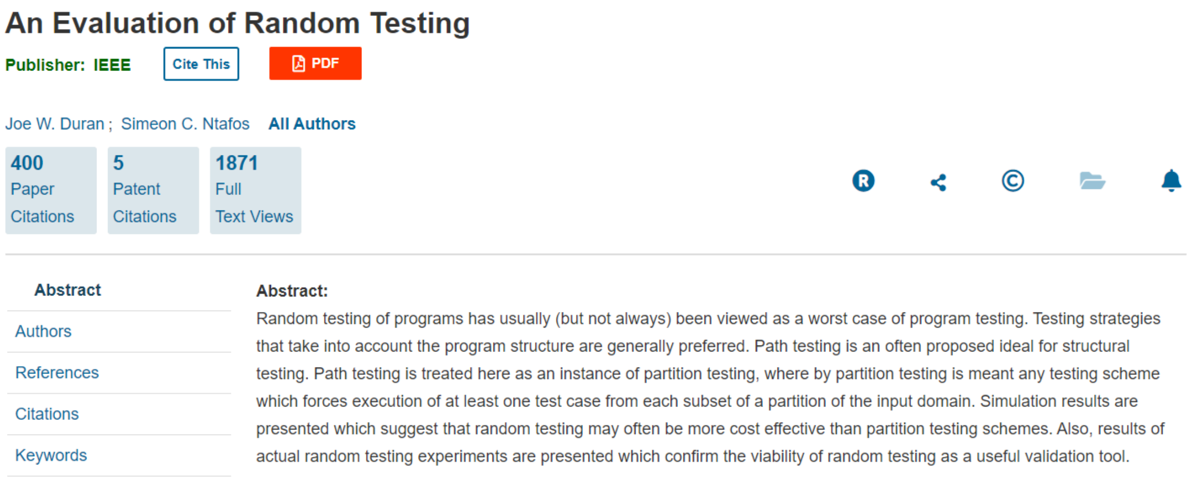
Welcome to JCrasher

An automatic robustness tester for Java

JCrasher: testing tool

5. Related Work

Empirical Studies:



An Evaluation of Random Testing

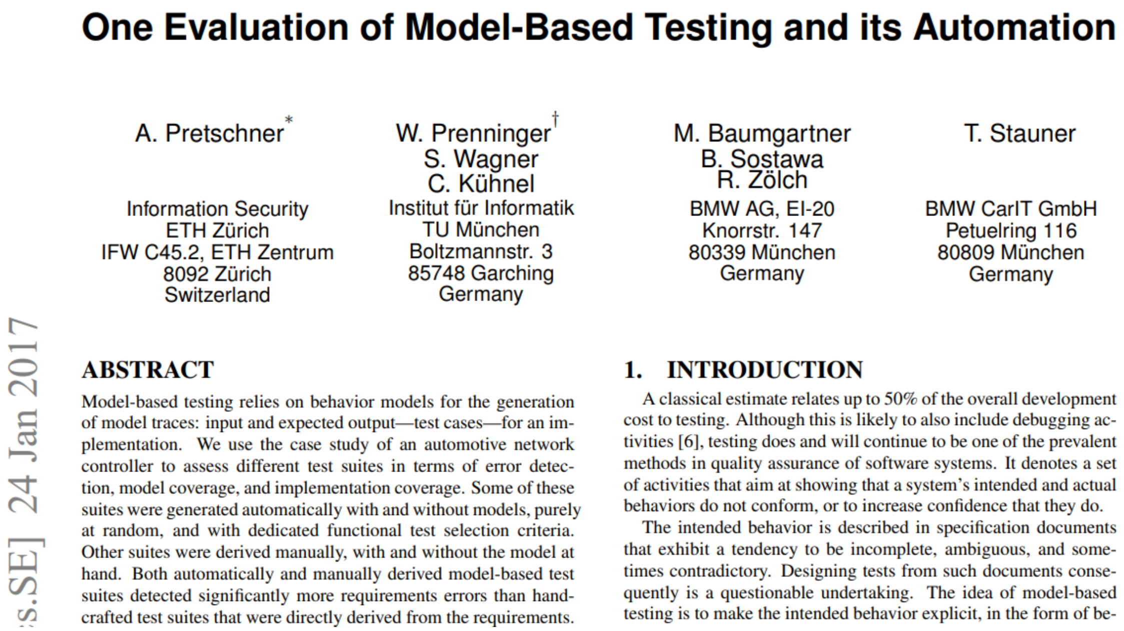
Publisher: IEEE Cite This PDF

Joe W. Duran ; Simeon C. Ntafos All Authors

400 Paper Citations | 5 Patent Citations | 1871 Full Text Views

Abstract
Authors
References
Citations
Keywords

Abstract:
Random testing of programs has usually (but not always) been viewed as a worst case of program testing. Testing strategies that take into account the program structure are generally preferred. Path testing is an often proposed ideal for structural testing. Path testing is treated here as an instance of partition testing, where by partition testing is meant any testing scheme which forces execution of at least one test case from each subset of a partition of the input domain. Simulation results are presented which suggest that random testing may often be more cost effective than partition testing schemes. Also, results of actual random testing experiments are presented which confirm the viability of random testing as a useful validation tool.



One Evaluation of Model-Based Testing and its Automation

A. Pretschner*
Information Security
ETH Zürich
IFW C45.2, ETH Zentrum
8092 Zürich
Switzerland

W. Prenninger†
S. Wagner
C. Kühnel
Institut für Informatik
TU München
Boltzmannstr. 3
85748 Garching
Germany

M. Baumgartner
B. Sostawa
R. Zölch
BMW AG, EI-20
Knorrstr. 147
80339 München
Germany

T. Stauner
BMW CarIT GmbH
Petuelring 116
80809 München
Germany

s.SE] 24 Jan 2017

ABSTRACT

Model-based testing relies on behavior models for the generation of model traces: input and expected output—test cases—for an implementation. We use the case study of an automotive network controller to assess different test suites in terms of error detection, model coverage, and implementation coverage. Some of these suites were generated automatically with and without models, purely at random, and with dedicated functional test selection criteria. Other suites were derived manually, with and without the model at hand. Both automatically and manually derived model-based test suites detected significantly more requirements errors than hand-crafted test suites that were directly derived from the requirements.

1. INTRODUCTION

A classical estimate relates up to 50% of the overall development cost to testing. Although this is likely to also include debugging activities [6], testing does and will continue to be one of the prevalent methods in quality assurance of software systems. It denotes a set of activities that aim at showing that a system's intended and actual behaviors do not conform, or to increase confidence that they do.

The intended behavior is described in specification documents that exhibit a tendency to be incomplete, ambiguous, and sometimes contradictory. Designing tests from such documents consequently is a questionable undertaking. The idea of model-based testing is to make the intended behavior explicit, in the form of be-

Found that random tests perform worse than both model-based and manually derived tests.

Found that partition and random testing are comparable in efficiency.

6. Conclusions and Future Work

- Eiffel chosen because of contracts.
- Predictability of random testing was never studied.
- Main results:
  - Random testing detects a defect within 30 seconds (almost) regardless of the class under test and regardless of the seed
  - Random testing is not very predictable in terms of the kind of defects that are detected but is predictable in terms of the relative number of defects.
  - The results are better if the first three-minute chunk of all thirty experiments are taken and put together than the median of all 90-minute-runs for that class.

6. Conclusions and Future Work

Examples of future and related work:

- Short vs long test runs.
- Complex vs simple structure.
- Efficiency of random tests regarding coupling and cohesion: