Software Testing, Verification and Validation

December 6, 2022
Week #13— Lecture #10

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Last week, we revisited the testing pyramid and the remaining testing levels: integration testing and system testing. We also introduced acceptance testing and regression testing. This week we will talk about automated test case generation.

Demo

(Backup demo https://www.evosuite.org/documentation/tutorial-part-1)

Test Creation / Generation

Software testing has become such an important piece of software development process, that it is commonly estimated that **half of the total cost/time to develop a software program** is dedicated to testing & debugging. The reason is that, although it is very common to use automated tools to execute test cases, such **test cases are commonly hand-written which is a tedious and error prone task**.

Automating the creation of such test cases offers several benefits, however it also raises some issues that would have to be addressed in order to be useful to use those tests.

1. Automation could reduce the cost/time of the testing process, and it could also create a much more complete set of test cases (as they would be systematically generated).
2. There are two main issues that need to be considered when generating test cases automatically: 1) test data (which inputs should be used to exercise the software under test?), and 2) test oracle (does the execution of the test reveal any fault?).

Test Creation / Generation

Several techniques for test generation have been proposed in the literature, including **random testing**, in which a software is executed with randomly generated inputs, **symbolic-execution** which explores control/data-paths of the software, and **search-based testing** in which efficient meta-heuristic search algorithms are used to generate test cases that resemble manually written tests (i.e., few short tests that exercise most of the code under test) are the most popular ones.

Test Creation as a Search Problem

- General **goals** while testing: make the program crash, achieve some code coverage, kill all mutants, …
- We have been **searching** for a test suite that achieves those goals.

- "I want to find all faults" cannot be measured. But code coverage, number of crashes can. If a goal can be measured, search can be automated.

Random Testing

The most **naïve test generation technique is Random Testing** (RT). In RT, the software under test is exercised with randomly generated inputs from the whole input domain of the software, and its observed output. Due to its simplistic nature, RT can be applied in practice with litter overhead and it has been widely used to, for example, exercise generic object contracts, unexpected security problems, and to reveal failures in several systems.

👎 However, there are some disagreements between researchers and practitioners on the coverage and effectiveness achieved by RT techniques on test generation. The **main point of criticism** among researchers is the **lack of a strategy to generate inputs**, as RT techniques do not take into account any information about the software under test, i.e., in theory, every test input in the input domain has the same probability of being selected.

Random Testing

```
       public String returnTen(int x) {
1          if (x == 10)
2            return “Six”; /∗ FAULT ∗/
3          else
4            return “Other number”;
       }
```

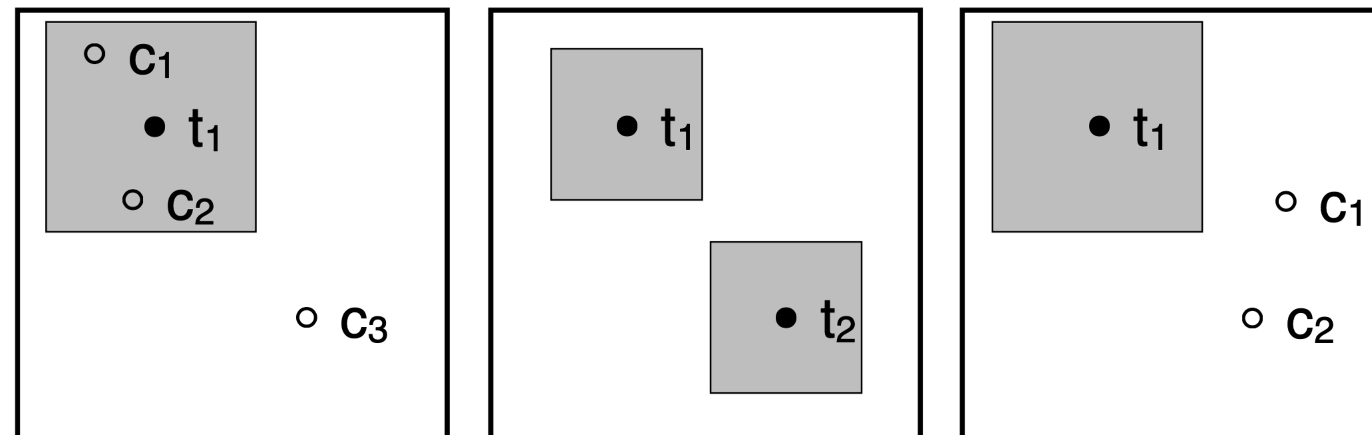
For example, considering this code under test, the probability of the conditional statement if (x == 10) being satisfied is 1 in 2^32 (assuming x is a 32-bits value), which illustrates the limitation of RT approaches.

Randoop

Randoop is a feedback-oriented technique which explores the execution of tests as they are created to avoid generating invalid inputs.
1. it generates a sequence of methods calls (each one selected at random), and methods arguments from previously created sequences.
2. it executes a sequence in order to provide feedback to the test generator, e.g., to avoid generation of tests that lead to runtime exceptions or to generate assertions that could trigger future changes.

It has been shown that Randoop is able to generate tests that are able to detect previously-unknown errors (not found by pure random techniques) in widely used Java libraries. However, the **large number of test cases generated by random testing techniques (including Randoop) may limit their adoption in practice**. As executing, evaluating, and maintaining such tests can become impractical over time.

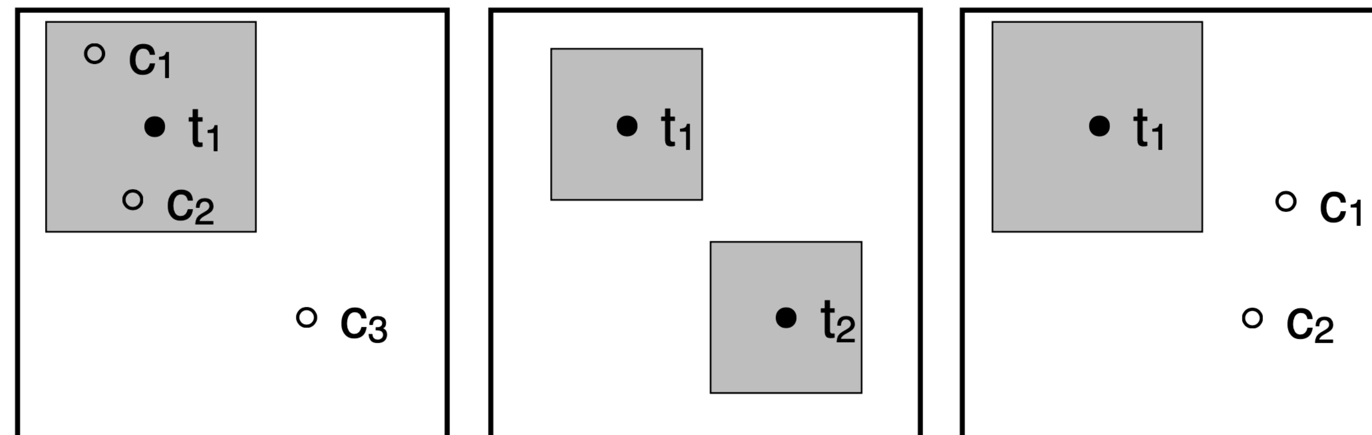
Adaptive/Restricted Random Testing



Restricted Random Testing (RRT) is an Adaptive Random Testing (ART) approach which excludes areas of the input domain. RRT randomly generates a test input from the entire input domain (for example, test input t1) and creates an exclusion region around t1. Then, new test candidates are generated, for example, c1 and c2. However, as they are in an exclusion region, both are discarded. If a test candidate is successfully generated out of an exclusion region (e.g., c3), it becomes a valid test input (e.g., t2) and a new exclusion region around it is created.

- If an exclusion region is to small, similar test inputs could be generated.
- If an exclusion region is too large, similar inputs would never be generated and the total number of inputs that could be explored would be limited. (Note that outside of exclusion regions candidates are selected with the same probability.)

Adaptive/Restricted Random Testing



To verify whether a new candidate is inside/outside of an exclusion region, RRT approach measures the euclidean distance between the new candidate and all test inputs previously selected, which could be very time consuming for a large number of test inputs.

Euclidean distance is a measure of the straight-line distance between two points in a Euclidean space. It is calculated by taking the square root of the sum of the squares of the differences between the coordinates of the two points. For example, in a two-dimensional space, the Euclidean distance between the points (x1, y1) and (x2, y2) would be calculated as: sqrt((x1 - x2)^2 + (y1 - y2)^2)

Euclidean distance is commonly used in geometry, computer vision, and machine learning. It is a useful metric for comparing the similarity of two points or vectors, and it is often used in algorithms that involve clustering or classification.

Effectiveness of Random Testing

👍 👎 - Thayer et al. argued that RT should be recommended as a fundamental step of the testing process, however Myers et al. stated that RT is the poorest test input methodology.
👍 - Mak compared RT and ART in terms of number of test inputs required to detect the first failure, and concluded that ART is able to detect the first failure with 30% (occasionally 50%) less test inputs. Although ART may be quicker or require less test inputs to detect the first failure than RT, ART requires more computational time and memory because the additional task of generating test inputs evenly spread across the input domain.
👎 - An empirically study conducted by Mayer et al. confirmed that although RRT is one of the most effective ART approaches, their runtime may become extremely long.
👎 - More recently, Arcuri et al. reported that **although ART could perform better than RT, the chance of finding faults with ART is less than 1%**.

Symbolic Execution

Symbolic Execution (SE) is a program analysis approach that executes a software program with symbolic values instead of concrete inputs, and represents the values of program variables as symbolic expressions. SE approaches proposed in the literature have been successful at finding subtle faults in several NASA's projects, at testing newly-modified source code, at automated debugging, and in many other areas.

Symbolic Execution

```
    public void foo(int x) {
1       int y = x * 3;
2       if (y == 42)
3         print("Good");
4       else
5         print("Bad");
    }
```

In an execution with **concrete inputs**, `foo` would be called with a concrete value (e.g., 7). Then, `y` would get the result of multiplying 7 by 3, i.e., 21. As 21 is not equal to 42, the condition on line 2 would be evaluated as `false`, and therefore the execution would print the word “Bad”.

Symbolic Execution

```
   public void foo(int x) {
1      int y = x * 3;
2      if (y == 42)
3        print("Good");
4      else
5        print("Bad");
   }
```

In a **symbolic execution**, `foo` would be called with a symbolic value (e.g., β). The execution then proceed with the multiplication and assigns β × 3 to y. Therefore, the condition to be evaluated on line 2 is no longer `if (y == 42)` but `if (β × 3 == 42)`. At this point in the execution, β could take any value. To solve the constraint `β × 3 == 42`, i.e., to generate two values such that each one could satisfy each outcome of the expression (i.e., `true` and `false`), constraint solvers such as Z3 are usually used. For this particular example, the value 14 would make the condition to be evaluated as `true`, and any other value would make the condition to be evaluated as `false`. Therefore, SE has explored all feasible paths of this toy example.

Symbolic Execution

👎 The number of paths in a program can grow exponentially with respect to the size of the program — a problem known as path explosion — or with the presence of loops (where the number of possible iterations could make the number of paths infinite). Therefore, applying traditional SE approaches to real and large software programs can become impractical. Nevertheless, several approaches have been proposed to address this issue (check the references slide).

Search-based Software Testing

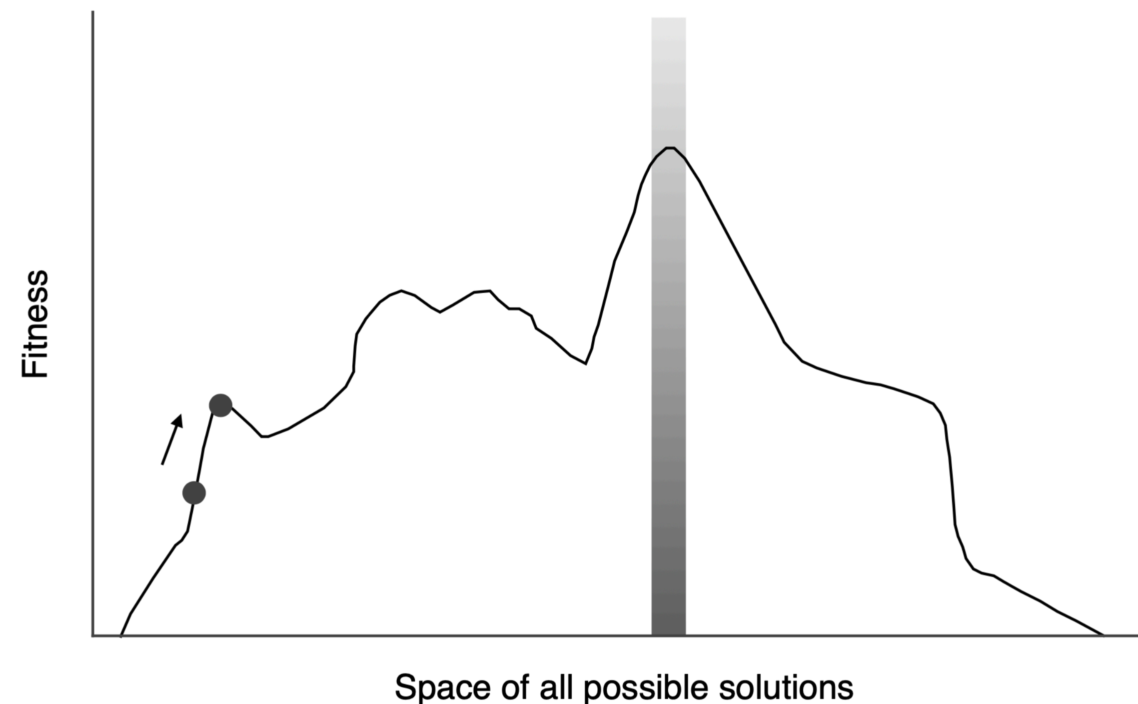
The application of **meta-heuristic search algorithms** (e.g., evolutionary algorithms) to software testing is known as Search-Based Software Testing (SBST). In SBST, test cases (or only test inputs) represent the search space of a meta-heuristic search algorithm and they are typically optimised for structural criteria (line coverage). However, other criteria such as functional and non-functional requirements, mutation, and exceptions have been also explored.

SBST — Representation

Evolutionary Algorithms (EAs) are inspired by natural evolution, and have been successfully used to address many kinds of optimisation problems. In the context of EAs, a solution is encoded "genetically" as an individual ("chromosome"), and a set of individuals is called a population.

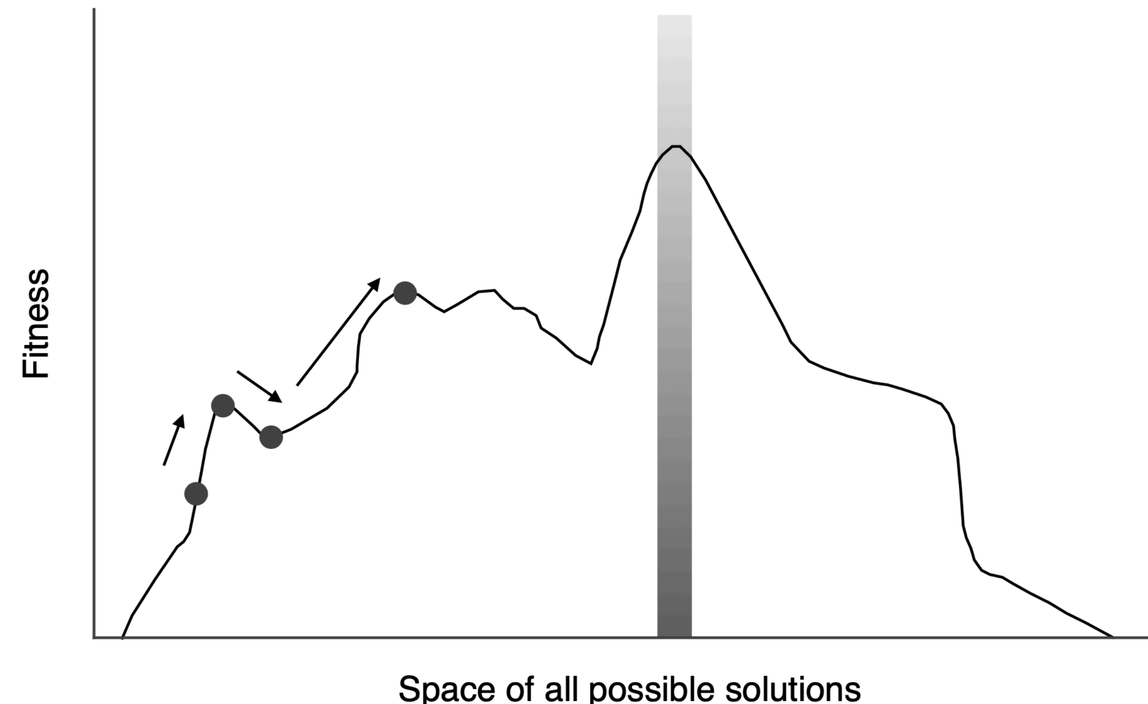
For **test suite generation**, the individuals of a **population** are **sets of test cases** (test suites); each **test case is a sequence of calls**. The population is gradually optimised using genetic-inspired operations such as
- **Crossover**, which merges genetic material from at least two individuals to yield new offspring, and
- **Mutation**, which independently changes the elements of an individual with a low probability.

Local search — Hill Climbing



Hill Climbing is a local search algorithm which evaluates solutions according to a fitness function. It starts with a random solution and in an, e.g., 1-dimensional problem, evaluates two neighbours (one to the right and one to the left). The solution with the best score value, i.e., fitness value, replaces the current one. However, due to lack of search power, the Hill Climbing algorithm does not make any assumptions about the landscape (a plot of the fitness) of the problem. Therefore, it only performs movements in the landscape if the next individual is better than the current, which could lead to be trapped in a local optimum solution.

Local search — Simulated Annealing



Simulated Annealing is a meta-heuristic algorithm similar to Hill Climbing, however, movements through the search space are not so restricted. To explore a large portion of the search-space, it uses a control parameter called temperature as the probability of accepting worse solutions, i.e., solutions with a lower fitness value. It starts with a high temperature value, but as the search evolves, the temperature decreases until it reaches zero, in which the search would work similar to the Hill Climbing algorithm. As the Hill Climbing algorithm, Simulated Annealing only considers one solution at time and it does not make any assumption about the landscape. If the temperature cools down to quickly, it might get stuck in a local optimum as the Hill Climbing algorithm.

Global search — Genetic Algorithm

The Genetic Algorithm (GA) is one of the most widely-used EAs in many domains because it can be easily implemented and obtains good results on average. Algorithm 1 illustrates a Standard GA. It starts by creating an initial random population of size ps (Line 1). Then, a pair of individuals is selected from the population using a strategy sf, such as rank-based, elitism or tournament selection (Line 6). Next, both selected individuals are recombined using crossover cf (e.g., single point, multiple-point) with a probability of cp to produce two new offspring o1, o2 (Line 7). Afterwards, mutation is applied on both offspring (Lines 8–9), independently changing the genes with a probability of mp, which usually is equal to n1, where n is the number of genes in a chromosome. The two mutated offspring are then included in the next population (Line 10). At the end of each iteration the fitness value of all individuals is computed (Line 13).

Algorithm 1 Standard Genetic Algorithm

Input: Stopping condition C, Fitness function δ, Population size p_s, Selection function s_f, Crossover function c_f, Crossover probability c_p, Mutation function m_f, Mutation probability m_p

Output: Population of optimised individuals P

```
1:  P ← GenerateRandomPopulation(p_s)
2:  PerformFitnessEvaluation(δ, P)
3:  while ¬C do
4:      N_P ← {}
5:      while |N_P| < p_s do
6:          p1, p2 ← Selection(s_f, P)
7:          o1, o2 ← Crossover(c_f, c_p, p1, p2)
8:          Mutation(m_f, m_p, o1)
9:          Mutation(m_f, m_p, o2)
10:         N_P ← N_P ∪ {o1, o2}
11:     end while
12:     P ← N_P
13:     PerformFitnessEvaluation(δ, P)
14: end while
15: return P
```

Fitness Function

In search-based test generation, the selection of individuals is guided by **fitness functions (which measure how good a test case or test suite is with respect to the search optimisation objective)**, such that individuals with good fitness values are more likely to survive and be involved in reproduction. **Fitness functions are usually based on metrics such as structural coverage**, functional and non-functional requirements, or mutation. Importantly, a fitness function usually also **provides additional search guidance leading to satisfaction of the goals**. For example, just checking in the fitness function whether a coverage target is achieved would not give any guidance to help covering it.

Fitness Function

Although structural coverage criteria are well established in order to evaluate existing test cases, they may be less suitable in order to guide test generation. As with any optimisation problem, an imprecise formulation of the optimisation goal could lead to unexpected results: for example, *although it is generally desirable that a reasonable test suite covers all statements of a software under test, the reverse may not hold – not every test that executes all statements is reasonable.*

Fitness Function, e.g., branch coverage

The concept of covering branches is also well understood in practice and implemented in popular tools, even though the practical definition of branch coverage may not always match the more theoretical definition of covering all edges of a program's control flow. Branch coverage is often interpreted as maximising the number of branches of conditional statements that are covered by a test suite. Hence, a test suite is said to satisfy the Branch Coverage criterion if and only if for every branch statement in the software under test, it contains at least one test whose execution evaluates the branch predicate to `true`, and at least one test whose execution evaluates the branch predicate to `false`.

Branch coverage, attempt 1

- **Goal:** Tests reach branching point (i.e., if-statement) and execute all possible outcomes.

- **Fitness function:** Measure coverage and try to maximize % covered.

👍 Measurable indicator of progress.
👎 No information on how to improve coverage.

Branch coverage, attempt 2

- **Goal:** Tests reach branching point (i.e., if-statement) and execute all possible outcomes.

- **Fitness function:** Branch Distance + Approach Level
    - Branch distance
      If other outcome is taken, how “close” was the target outcome?
      How much do we need to change program values to get the outcome we wanted?
    - Approach level
      Number of branching points we need to execute to get to the target branching point.

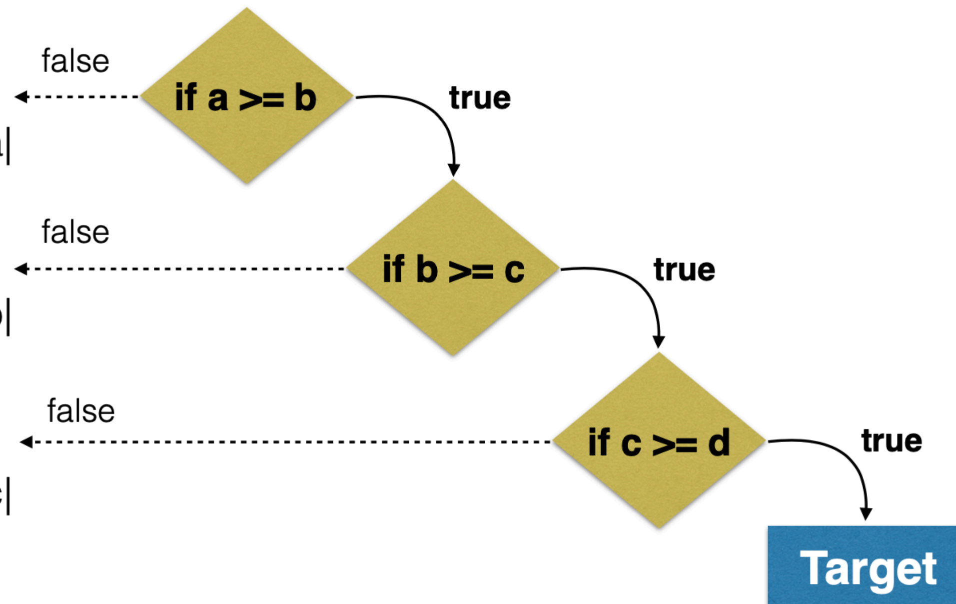
👍 Measurable indicator of progress.
👎 No information on how to improve coverage.

```java
public void foo(int a, int b, int c, int d) {
 if (a >= b) {
  if (b >= c) {
   if (c >= d) {
    // target
```

Target Missed
Approach level = 2
Branch distance = |b - a|

Target Missed
Approach level = 1
Branch distance = |c - b|

Target Missed
Approach level = 0
Branch distance = |d - c|



For example, given the first predicate `a >= b` and an execution with values `a=5` and `b=3`, the branch distance to the predicate evaluating to `true` would be `|3−5|=2`, whereas an execution with values `a=5` and `b=4` is closer to being `true` with a branch distance of `|4−5|=1`.

The execution with values `a=5` and `b=4` and `c=6` and `d=7` leads to a branch fitness function of `1+|4−6|=3`.

Tools

- EvoSuite, https://github.com/EvoSuite/evosuite
- Randoop, https://github.com/randoop/randoop
- KLEE, https://github.com/klee/klee
- Java PathFinder, https://github.com/SymbolicPathFinder/jpf-symbc

Many other @ https://github.com/ksluckow/awesome-symbolic-execution#tools

References

- I. Burnstein. Practical software testing a process-oriented approach.
- P. Ammann, J. Offutt. Introduction to Software Testing.
- P. Jorgensen. Software Testing A Craftsman's Approach.
- M. Pezze, M. Young. Software Testing and Analysis Process, Principles and Techniques.
- Gordon Fraser and José Miguel Rojas; Software Testing, 2019. ISBN 978-3-030-00262-6.
- R. Baldoni, E. Coppa, D. Cono D'elia, C. Demetrescu, I. Finocchi; A Survey of Symbolic Execution Techniques, 2019, https://doi.org/10.1145/3182657.

References

- Glenford J. Myers, Corey Sandler, and Tom Badgett. The Art of Software Testing. 3rd. Wiley Publishing, 2011. isbn: 1118031962, 9781118031964
- Gordon Fraser and Andreas Zeller. "Mutation-driven Generation of Unit Tests and Oracles". IEEE Transactions on Software Engineering 38.2 (2012), pp. 278–292. issn: 0098-5589
- Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. "Random Testing: Theoretical Results and Practical Implications". IEEE Transactions on Software Engineering 38.2 (Mar. 2012), pp. 258–277. issn: 0098-5589
- Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. "Feedback-Directed Random Test Generation". Proceedings of the 29th international conference on Software Engineering. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 75–84. isbn: 0-7695-2828-7
- Paolo Tonella. "Evolutionary Testing of Classes". Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA '04. Boston, Massachusetts, USA: ACM, 2004, pp. 119–128. isbn: 1-58113-820-2
- T. A. Thayer, M. Lipow, and E. C. Nelson. Software Readability. North Holland, Amsterdam, 1978
- T.Y. Chen, H. Leung, and I.K. Mak. "Adaptive Random Testing". Advances in Computer Science ASIAN 2004. Higher-Level Decision Making. Ed. by MichaelJ. Maher. Vol. 3321. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 320–329. isbn: 978-3-540-24087-7

References

- Kwok Ping Chan, Tsong Yueh Chen, and Dave Towey. "Restricted Random Testing". Proceedings of the 7th International Conference on Software Quality. ECSQ '02. London, UK, UK: Springer-Verlag, 2002, pp. 321–330. isbn: 3-540-43749-5
- Huai Liu, Xiaodong Xie, Jing Yang, Yansheng Lu, and Tsong Yueh Chen. "Adaptive Random Testing Through Test Profiles". Software-Practice & Experience 41.10 (Sept. 2011), pp. 1131–1154. issn: 0038-0644
- I. K. Mak. "On the Effectiveness of Random Testing". MA thesis. Australia: University of Melbourne, Department of Computer Science, 1997
- Johannes Mayer and Christoph Schneckenburger. "An Empirical Analysis and Comparison of Random Testing Techniques". Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering. ISESE '06. Rio de Janeiro, Brazil: ACM, 2006, pp. 105–114. isbn: 1-59593-218-6
- Andrea Arcuri and Lionel Briand. "Adaptive Random Testing: An Illusion of Effectiveness?" Proceedings of the 2011 International Symposium on Software Testing and Analysis. ISSTA '11. Toronto, Ontario, Canada: ACM, 2011, pp. 265–275. isbn: 9781-4503-0562-4
- L.A. Clarke. "A System to Generate Test Data and Symbolically Execute Programs". IEEE Transactions on Software Engineering SE-2.3 (Sept. 1976), pp. 215–222. issn: 0098-5589
- Corina S. Pasareanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. "Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing Nasa Software". Proceedings of the 2008 International Symposium on Software Testing and Analysis. ISSTA '08. Seattle, WA, USA: ACM, 2008, pp. 15–26. isbn: 978-1-60558-050-0

References

- Zhihong Xu, Yunho Kim, Moonzoo Kim, Gregg Rothermel, and Myra B. Cohen. "Directed Test Suite Augmentation: Techniques and Tradeoffs". ACM Symposium on the Foundations of Software Engineering (FSE). FSE '10. Santa Fe, New Mexico, USA: ACM, 2010, pp. 257–266. isbn: 978-1-60558-791-2
- Zhihong Xu, Yunho Kim, Moonzoo Kim, and Gregg Rothermel. "A Hybrid Directed Test Suite Augmentation Technique". IEEE International Symposium on Software Reliability Engineering (ISSRE). ISSRE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 150–159. isbn: 978-0-7695-4568-4
- Cristian Zamfir and George Candea. "Execution Synthesis: A Technique for Automated Software Debugging". Proceedings of the 5th European Conference on Computer Systems. EuroSys '10. Paris, France: ACM, 2010, pp. 321–334. isbn: 978-1-60558-5772
- Corina S. Pasareanu and Willem Visser. "A Survey of New Trends in Symbolic Execution for Software Testing and Analysis". International Journal on Software Tools for Technology Transfer (STTT) 11.4 (Oct. 2009), pp. 339–353. issn: 1433-2779
- Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. "Symbolic Execution for Software Testing in Practice: Preliminary Assessment". Proceedings of the 33rd International Conference on Software Engineering. ICSE '11. Honolulu, HI, USA: ACM, 2011, pp. 1066–1071. isbn: 978-1-4503-0445-0
- Phil McMinn. "Search-based Software Test Data Generation: A Survey". Software Testing, Verification and Reliability 14.2 (June 2004), pp. 105–156. issn: 0960-0833

References

- Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. "Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications". Department of Computer Science, King's College London, Tech. Rep. TR-09-03 (2009)
- Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. "Search-based Software Engineering: Trends, Techniques and Applications". ACM Comput. Surv. 45.1 (Dec. 2012), 11:1–11:61. issn: 0360-0300
- Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, and Shin Yoo. "Search Based Software Engineering: Techniques, Taxonomy, Tutorial". Ed. by Bertrand Meyer and Martin Nordio. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 1–59. isbn: 978-3-642-25230-3
- Phil McMinn. "Search-Based Software Testing: Past, Present and Future". Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops. ICSTW '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 153–163. isbn: 978-0-7695-4345-1
- S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gal, S. Katsikas, and K. Karapoulios. "Application of Genetic Algorithms to Software Testing". Proceedings of the 5th International Conference on Software Engineering and its Applications. Toulouse, France: 1992, pp. 625–636
- B. Korel. "Automated Software Test Data Generation". IEEE Transactions on Software Engineering 16.8 (Aug. 1990), pp. 870-879. issn: 0098-5589
- B.F. Jones, H.-H. Sthamer, and D.E. Eyres. "Automatic Structural Testing using Genetic Algorithms". Software Engineering Journal 11.5 (Sept. 1996), pp. 299–306. issn: 0268-6961

References

- R. P. Pargas, M. J. Harrold, and R. R. Peck. "Test-Data Generation Using Genetic Algorithms". Software Testing, Verification and Reliability 9 (4 Dec. 1999), pp. 263–282
- Joachim Wegener, Andre Baresel, and Harmen Sthamer. "Evolutionary Test Environment for Automatic Structural Testing". Information and Software Technology 43.14 (2001), pp. 841–854. issn: 0950-5849
- Oliver Bühler and Joachim Wegener. "Evolutionary Functional Testing". Computers and Operations Research 35.10 (Oct. 2008), pp. 3144–3160. issn: 0305-0548
- Wasif Afzal, Richard Torkar, and Robert Feldt. "A Systematic Review of Search-based Testing for Non-functional System Properties". Information and Software Technology 51.6 (June 2009), pp. 957–976. issn: 0950-5849
- Gordon Fraser and Andrea Arcuri. "Achieving Scalable Mutation-based Generation of Whole Test Suites". Empirical Software Engineering (2014), pp. 1–30. issn: 1382-3256
- Nigel Tracey, John Clark, and Keith Mander. "Automated Program Flaw Finding Using Simulated Annealing". SIGSOFT Software Engineering Notes 23.2 (Mar. 1998), pp. 73–81. issn: 0163-5948
- Nigel Tracey, John Clark, John McDermid, and Keith Mander. "Systems Engineering for Business Process Change". Ed. by Peter Henderson. New York, NY, USA: Springer-Verlag New York, Inc., 2002. Chap. A Search-based Automated Test-data

References

- Generation Framework for Safety-critical Systems, pp. 174–213. isbn: 1-85233-399-5
- Gordon Fraser and Andrea Arcuri. "Whole Test Suite Generation". IEEE Transactions on Software Engineering 39.2 (2013), pp. 276–291. issn: 0098-5589
- Mark Harman and Phil McMinn. "A Theoretical & Empirical Analysis of Evolutionary Testing and Hill Climbing for Structural Test Data Generation". Proceedings of the 2007 International Symposium on Software Testing and Analysis. ISSTA '07. London, United Kingdom: ACM, 2007, pp. 73–83. isbn: 978-159593-734-6
- J. H. Holland. "Genetic Algorithm and the Optimal Allocation of Trials". SiAM Journal of Computing 2.2 (June 1973), pp. 88-105