Università degli Studi di Salerno
Dipartimento di Informatica

Tesi di Laurea di I livello in
Informatica

Automatic Test Case Generation for CyberPhysical Systems

Relatore
Giuseppe Scanniello
Correlatore
Dott. Nome Cognome

Candidato
Michelangelo Esposito

Academic Year 2021-2022

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Contents

List of Figures

List of Acronyms and Abbreviations

Chapter 1

Introduction

Things to add:

- where different types of coverage are useful
- Add formulas for coverage criteria

Chapter 2

Problem formulation

2.1 An overview on search problems

As humans, everyday we perform search: from looking for our car keys in the house to searching for a new book to purchase, this action is engraved in our daily life. In the same way, search is one of the most fundamental operations on which computer science has focused since its early days: performing efficient search is the core operation of many classes of algorithms, such as the ones responsible for route planning, computer vision, robotics, automated software testing, puzzle solving, and many others.

Any search problem is typically defined by:

- A search space: the set of elements in which we search for our solution. Examples include paths in a graph, the numbers to insert in a sorted list, or the web pages accesses by a web index.
- A condition that defines the characteristics of candidate solutions.

formal problem definition with graphs, distance functions, ...

A search algorithm will therefore examine the search space according to some criteria and attempt to find a suitable solution. Finding any candidate solution is just one of the objective of search problems however, often times we are interested in finding the best possible solution, the optimum; such problems are referred to as optimization problems.

The exploration of the search space can happen in many different ways. The most basic approach consists of exploring the elements one by one, till a solution/the optimum is found, in a brute-force manner. Applying this simple solution for problems whose search space is somewhat limited can be done without too many repercussions on execution time, however, given the exponential size of the search spaces of most practical problems, a brute-force approach is infeasible.

For problems in which we have no choice but to employ brute-force search, there may be some room for improvement by using heuristics. Heuristics are estimates of the distance function to reach a goal state from the current node [1].

Therefore, what it is preferred to obtain a solution that may not be the optimum, but rather it is "good enough" for our objective. This approach is called local search.

Hill climbing, simulated annealing, GAs... Genetic Algorithms (GAs) are an example of an evolutionary search approach for test case generation; starting from an initial, often randomly generated, population of test cases, the algorithm keeps evolving the individuals according to simulated

natural evolution theory principles. In this context, a typical fitness function of a GA would measure the distance between the execution trace of the generated test cases and the coverage targets.

2.2 Driving simulators for self-driving vehicles

- BeamNG
- CARLA [2] is another open source simulator built specifically for self-driving vehicles in mind. Similarly to BeamNG, it features an API written in Python to actively interact with the simulation any manipulate traffic, pedestrians, weather conditions and more parameters. Furthermore, an interesting feature of CARLA is its runtime texture streaming; this features allows developers to change the texture of every object in the scene during runtime. Such a feature can be particularly useful to tune a driving model to continuously deal with adversarial attacks. Finally, CARLA simulations can run without rendering any asset, thus allowing runs to be executed much quicker.
- Mathematical models:

Chapter 3

Literature

In this chapter we provide a general overview of the literature concerning evolutionary test case generation, in both the Object-Oriented and procedural contexts. The main techniques and algorithms employed for the problem will be examined and compared, to act as the base for our study.

3.1 EvoSuite

EvoSuite is an example of an evolutionary algorithm that optimizes the whole test suite towards just one coverage criterion, rather than generating test cases directed towards multiple coverage criteria. With EvoSuite, any collateral coverage isn't a concern since all coverage is intentional, given that the ultimate goal is to generate the whole test suite. The algorithm starts with a randomly generated population of test suites. The fitness function rewards better coverage of the source code; if two suites have the same coverage, the one with fewer statements is chosen. For each test suite, its fitness is measured by executing all of its test cases and keeping track of the executed methods and of the minimal branch distance for each branch.

- Expand on bloat in EvoSuite
- algorithm pseudocode

3.2 NSGA-II

A popular algorithm for many-objective search problems is the Non-dominated Sorting Genetic Algorithm II (NSGA-II). This algorithm is based on three principles:

- It uses elitism when evolving the population: the most fit individuals are carried over along the offsprings.
- It uses an explicit diversity-preserving mechanism, the Crowding distance.
- It emphasizes the non-dominated solutions, as its name suggests.

In the context of test cases, domination can be expressed by the following relation:

Definition 1. A test case x dominates another test case y, also written $x \prec y$ if and only if ...

Algorithm 1: NSGA-II

input : $U = \{u_1, ..., u_m\}$ the set of coverage targets of a program
Population size M
output : A test suite T

```
1  begin
2      t ← 0
3      P_t ← RANDOM-POPULATION(M)
4      while not(search_budget_consumed) do
5          Q_t ← GENERATE-OFFSPRING(P_t)
6          R_t ← P_t ∪ Q_t
7          F ← FAST-NONDOMINATED-SORT(R_t)
8          P_{t+1} ← 0
9          d ← 1
10         while (|P_{t+1}| + |F_d| ⩽ M) do
11             CROWDING-DISTANCE-ASSIGNMENT(F_d)
               P_{t+1} ← P_{t+1} ∪ F_d
12             d ← d + 1
13         Sort(F_d) // according to the crowding distance
14         P_{t+1} ← P_{t+1} ∪ F_d[1 : (M − |P_{t+1}|)]
15         t ← t + 1
16     S ← t + 1
```

The NSGA-II algorithm works as follows:

- Starting from an initial population of individuals Pt, generate an offspring population Qt of equal size and merge the two together, obtaining the population Rt.
- Perform non-dominated sorting of the individuals in Rt based on target indicators and classify them by fronts, i.e. they are sorted according to an ascending level of non-domination. This ensures that the top Pareto-optimal individuals will survive to the next generation.
- If one of the fronts in the sorted sequence doesn't fit in terms of population size, crowding distance sorting is performed.
- Create the new population based on crowded tournament selection, then perform crossover and mutation.

Figure 3.1 summarizes the main loop of the algorithm:

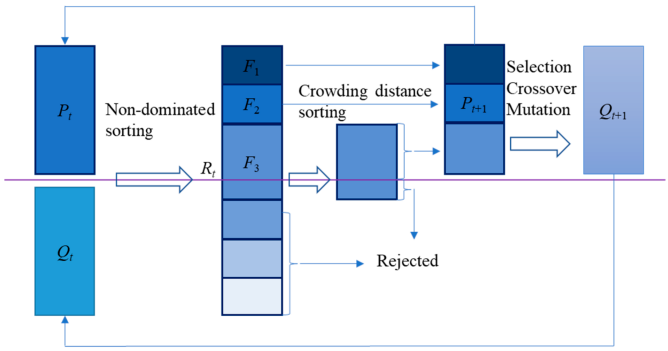

Figure 3.1: NSGA-II algorithm main loop.

In the context of software engineering, NSGA-II has been applied to problems such as software refactoring and test case prioritization, with two or three objectives. If the number of objectives begins to grow, however, the performance of the algorithm doesn't scale up well [3]. To overcome these limitations, there have been various adjustments and optimization of this algorithm...

3.3 SPEA2

Strength Pareto Evolutionary Algorithm (SPEA) is another multi-objective evolutionary search algorithm based on the concept of Pareto domination for fitness and selection, originally proposed by Zitzler and Thiele in 1999 [4]. SPEA2 was presented as an evolution of the algorithm by the same authors [5]. A peculiarity of SPEA2 is that it uses an archive to store non-dominated solutions generated in each iteration; as we will see later, the archive approach is also employed by more modern solutions.

During the evolution, a strength value is calculated for each individual in each generation and is used for the selection of the fittest individuals. For any solution i, this strength value is measured based on the numbers of j individuals, belonging to the archive and the population, dominated by i.

Algorithm 2: SPEA2

```
   input : U = {u_1, ..., u_m} the set of coverage targets of a program
           An archive A
           Population size M
1  begin
2      P_t ← RANDOM-POPULATION(M)
3      while not(search_budget_consumed) do
4          CALCULATE-STRENGTH(P_t, A)
5          UPDATE-ARCHIVE(P_t, A)
6          BINARY-TOURNAMENT-SELECTION(P_t)
7          CROSOVER(P_t)
8          MUTATION (P_t)
```

If the size of the archive doesn't reach the population size, dominated individuals are used to fill up the remaining spaces. Similarly to NSGA-II isn't suitable for problems with more than three objectives[6].

3.4 MOSA

The Multi-Objective Sorting Algorithm, MOSA [7] is a GA proposed as an improvement over NSGA-II and SPEA2 for many-objective test case generation, with the main goal of allowing for a higher number of objectives to optimize. With this algorithm, the coverage of the different branches makes up the set of objectives to be optimized. The main characteristic about MOSA is that instead of ranking candidates for selection based on their Pareto optimality, it uses a preference criterion; this criterion selects the test case with the lowest objective score for each uncovered target; these selected individuals are given a higher chance of survival, while other test cases are ranked with the traditional NSGA-II approach.

As the first step, MOSA starts from a randomly generated initial population of test cases. To create the next generation, MOSA generates offsprings by implementing the classic operations of selection, crossover and mutation. Before selection occurs however, for each uncovered branch, the test case with the lowest objective score(branch distance + approach level) is determined; these test cases will have assigned the rank 0 and will make up the first front. The remainder of the test cases will be sorted according to the traditional NSGA-II approach.

After the rank assignment step is complete, the crowding distance is calculated to determine which individuals to select: the individuals with higher distance from the rest are given a higher chance of being selected. The algorithm attempts to select as many test cases as possible, starting from front 0, until the population size is reached.

Algorithm 3: MOSA

input : $U = \{u_1, ..., u_m\}$ the set of coverage targets of a program
Population size M
output : A test suite T

```
1  begin
2      t ← 0
3      P_t ← RANDOM-POPULATION(M)
4      archive ← UPDATE-ARCHIVE(P_t, ∅)
5      while not(search_budget_consumed) do
6          Q_t ← GENERATE-OFFSPRING(P_t)
7          archive ← UPDATE-ARCHIVE(Q_t, archive)
8          R_t ← P_t ∪ Q_t
9          F ← PREFERENCE-SORTING(R_t)
10         P_{t+1} ← 0
11         d ← 0
12         while (|P_{t+1}| + |F_d| ⩽ M) do
13             CROWDING-DISTANCE-ASSIGNMENT(F_d)
               P_{t+1} ← P_{t+1} ∪ F_d
14             d ← d + 1
15         Sort(F_d) // according to the crowding distance
16         P_{t+1} ← P_{t+1} ∪ F_d[1 : (M − |P_{t+1}|)]
17         t ← t + 1
18     T ← archive
```

Algorithm 4: PREFERENCE-SORTING

input : $T = \{t_1, ..., t_n\}$, a set of candidate test cases
Population size M
output : A non-dominated ranking assignment F

```
1  begin
2      F_0 // first front
3      for u_i ∈ U and u_i is uncovered do
4          // for each uncovered test target, the best test case according
             to the preference criterion is selected
5          t_best ← test case in T with minimum objective score for u_i
6          F_0 ← F_0 ∪ {t_best}
7      T ← T − F_0
8      if |F_0| > M then
9          F_1 ← T
10     else
11         U* ← {u ∈ U : u is uncovered}
12         E ← FAST-NONDOMINATED-SORT(T , {u ∈ U*})
13         d ← 0
14         for All non-dominated fronts in E do
15             F_{d+1} ← E_d
```

Algorithm 5: DOMINANCE-COMPARATOR

input : Two test cases to compare, $t1$ and $t2$
$U = \{u_1, ..., u_m\}$ the set of coverage targets of a program

```
1  begin
2      dominates1 ← false
3      dominates2 ← false
4      for u_i ∈ U and u_i is uncovered do
5          f_i^1 ← values of u_i for t_1
6          f_i^2 ← values of u_i for t_2
7          if f_i^1 < f_i^2 then
8              dominates1 ← true
9          if f_i^2 < f_i^1 then
10             dominates2 ← true
11         if dominates1 == true and dominates2 == true then
12             break
13     if dominates1 == dominates2 then
14         // t1 and t2 don't dominate each other
15     else
16         if dominates1 == true then
17             // t1 dominates t2
18         else
19             // t2 dominates t1
```

Finally, MOSA uses an archived population that keeps track of the best performing test cases, in order to form the final test suite. The archive is continuously updated as follows:

Algorithm 6: UPDATE-ARCHIVE

input : A set of candidate test cases T
An archive A
output : An updated archive A

```
1  begin
2      for u_i ∈ U do
3          t_best ← ∅
4          best_length ← ∞
5          if u_i already covered then
6              t_best ← test case in A covering u_i
7              best_length ← number of statements in t_best
8          for t_j ∈ T do
9              score ← objective score of t_j for target u_i
10             length ← number of statements in t_j
11             if score == 0 and length ≤ best_length then
12                 replace t_best with t_j in A
13                 t_best ← t_j
14                 best_length ← length
15     return A
```

3.5 DynaMOSA

One of the limitations of MOSA is its inability to consider the implicit dependencies between targets, which simply appear as independent objectives to optimize. Such dependencies can be quite common in practice: most commonly, there exist branches which may only be satisfied if and only if other branches in the outer scope have been already covered.

DynaMOSA, Dynamic Many-Objective Sorting Algorithm [8] is an algorithm that focuses on these dynamic dependencies, and has been proposed as an evolution of MOSA. Before introducing the algorithm, a few definitions are needed [8]:

Definition 2. (Dominator): A statement $s1$ dominates another statement $s2$ if every execution path to $s2$ passes through $s1$.

Definition 3. (Post-dominator): A statement $s1$ post-dominates another statement $s2$ if every execution path from $s2$ to the exit point passes through $s1$.

Definition 4. (Control dependency): There is a control dependency between program statement $s1$ and program statement $s2$ iff: (1)$s2$ is not a postdominator of $s1$, and (2) there exist a path in the control flow graph between $s1$ and $s2$ whose nodes are postdominated by $s2$.

Definition 5. (Control dependency graph): The graph $G = \langle N, E, s \rangle$, consisting of nodes $n \in N$ that represent program statements, and edges $e \in E \subseteq NXN$ that represent control dependencies between program statements, is called control dependency graph. Node $s \in N$ represents the entry node, which is connected to all nodes that are not under the control dependency of another node.

This definition can be extended to other coverage criteria. For example, given two branches $b1$ and $b2$, we say that $b1$ holds a control dependency on $b2$ if $b1$ is postdominated by a statement $s1$ which holds a control dependency on a node $s2$ that postdominates $b2$.

DynaMOSA uses the control dependency graph to identify which targets are independent from each other and which ones can be covered only after satisfying previous targets in the graph. Algorithm 6 highlights the test case evolution in DynaMOSA.

Algorithm 7: DynaMOSA

input : $U = \{u_1, ..., u_m\}$ the set of coverage targets of a program
Population size M
$G = \langle N, E, s \rangle$: control dependency graph of the program
$\phi : E \to U$: partial mapping between edges and targets
output : A test suite T

```
1  begin
2      U* ← targets in U with not control dependencies
3      t ← 0
4      P_t ← RANDOM-POPULATION(M)
5      archive ← UPDATE-ARCHIVE(P_t, ∅)
6      U* ← UPDATE-TARGETS(U*, G, φ)
7      while not(search_budget_consumed) do
8          Q_t ← GENERATE-OFFSPRING(P_t)
9          archive ← UPDATE-ARCHIVE(Q_t, archive)
10         U* ← UPDATE-TARGETS(U*, G, φ)
11         R_t ← P_t ∪ Q_t
12         F ← PREFERENCE-SORTING(R_t, U*)
13         P_{t+1} ← 0
14         d ← 0
15         while (|P_{t+1}| + |F_d| ⩽ M) do
16             CROWDING-DISTANCE-ASSIGNMENT(F_d, U*)
               P_{t+1} ← P_{t+1} ∪ F_d
17             d ← d + 1
18         Sort(F_d) // according to the crowding distance
19         P_{t+1} ← P_{t+1} ∪ F_d[1 : (M − |P_{t+1}|)]
20         t ← t + 1
21     T ← archive
```

The main difference with MOSA is how the initial target population is selected; with DynaMOSA, only the targets that are free from dependencies are part of this set. Then, in each iteration, the set of non-dependent targets is updated by using the following procedure:

Algorithm 8: UPDATE-TARGETS

input : $G = \langle N, E, s \rangle$: control dependency graph of the program
$U^* \subseteq U$: current set of targets
$\phi : E \to U$: partial mapping between edges and targets
output : U^*: updated set of targets to optimize

```
1  begin
2      for u ∈ U do
3          if u is covered then
4              U* ← U* − {u}
5              e_u ← edge in G for the target u
6              visit(e_u)
7  function visit(e_u ∈ E):
8      for each unvisited e_n ∈ E control depenmdent on e_u do
9          if φ(e_n) is not covered then
10             U* ← U* ∪ {φ(e_n)}
11             set e_n as visited
12         else
13             visit(e_m)
```

This routine update the sets of selected targets U^* in order to include any uncovered targets that are control dependent on the newly covered target. In the case of newly covered targets, the procedure iterates over the control graph to find all control dependent targets.

3.6 OCELOT and object-oriented versus procedural test case generation

Generally speaking, both EvoSuite and MOSA/DynaMOSA have been designed with the OO paradigm in mind, as well as most empirical software engineering tools for testing.

...expand on automated testing for procedural programs...

Optimal Coverage sEarch-based tooL for sOftware Testing, OCELOT [9] is a test case generation tool for C programs that implements both a multi-objective approach based on MOSA, and a new iterative single-target approach named LIPS, Linear Independent Path-based Search. OCELOT can automatically detect the input types of a C function, without requiring any specification of parameters; Additionally it can handle the different data types of C, including pointers and structs, and produces test cases based on the Check framework [10]. Similarly to other tools, OCELOT is not able to generate oracles. From an implementation point of view, the GA implementation in OCELOT was realized with the JMetal, a Java framework for multi-objective optimization with meta-heuristics [11]. Furthermore, JNI is used as interface with the target program to try different combinations of test data. SBX Crossover, Polynomial Mutation and Binary Tournament Selection, for both the multi-objective and iterative approaches.

The iterative approach used in OCELOT, LIPS

Running OCELOT consists of two main phases: build and run [12], as highlighted in figure 3.2:

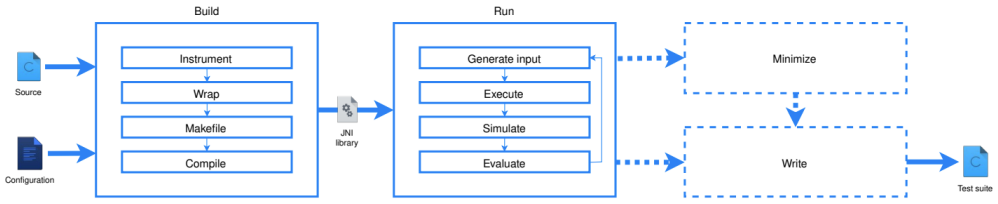

Figure 3.2: OCELOT components and workflow.

3.7 Test case generation for CyberPhysical Systems

Typically, in the development stages of a CPS, validation and verification happen according to the V-model approach.

AmbieGen based on NSGA-II. Markov chains used to assign values to different attributes

[13] [14]

Chapter 4

Conclusions

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Chapter 5

Math reference

1. $$\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n = \lim_{n\to\infty}\frac{n}{\sqrt[n]{n!}}$$

2. $$e = \sum_{n=0}^{\infty}\frac{1}{n!}$$

3. $$e = 2 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{2}{3 + \cfrac{3}{4 + \ddots}}}}$$

4. $$\int_a^b f(x)dx$$

5. $$x_1, x_2, ..., x_n$$

Bibliography

[1] Stefan Edelkamp and Stefan Schrodl. *Heuristic Search: Theory and Applications*. Morgan Kaufmann, 2008, pp. 403–449.

[2] Alexey Dosovitskiy et al. "CARLA: An Open Urban Driving Simulator". In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 1–16.

[3] Bingdong Li et al. "Many-Objective Evolutionary Algorithms: A Survey". In: vol. 48. 1. 2015, 13:1–13:35. DOI: 10.1145/2792984. URL: https://doi.org/10.1145/2792984.

[4] Eckart Zitzler and Lothar Thiele. "Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach". In: vol. 3. 4. 1999, pp. 257–271. DOI: 10.1109/4235.797969. URL: https://doi.org/10.1109/4235.797969.

[5] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. "SPEA2: Improving the Strength Pareto Evolutionary Algorithm". In: (2001).

[6] Xi Liu and Dan Zhang. "Multi-objective Investment Decision Making Based on an Improved SPEA2 Algorithm". In: *Artificial Intelligence and Security - 5th International Conference, ICAIS 2019, New York, NY, USA, July 26-28, 2019, Proceedings, Part I*. Ed. by Xingming Sun, Zhaoqing Pan, and Elisa Bertino. Vol. 11632. Lecture Notes in Computer Science. Springer, 2019, pp. 434–443. DOI: 10.1007/978-3-030-24274-9_39. URL: https://doi.org/10.1007/978-3-030-24274-9%5C_39.

[7] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. "Reformulating Branch Coverage as a Many-Objective Optimization Problem". In: *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 2015, pp. 1–10. DOI: 10.1109/ICST.2015.7102604. URL: https://doi.org/10.1109/ICST.2015.7102604.

[8] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. "Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets". In: vol. 44. 2. 2018, pp. 122–158. DOI: 10.1109/TSE.2017.2663435. URL: https://doi.org/10.1109/TSE.2017.2663435.

[9] Simone Scalabrino et al. "Search-Based Testing of Procedural Programs: Iterative Single-Target or Multi-target Approach?" In: *Search Based Software Engineering - 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings*. Ed. by Federica Sarro and Kalyanmoy Deb. Vol. 9962. Lecture Notes in Computer Science. 2016, pp. 64–79. DOI: 10.1007/978-3-319-47106-8_5. URL: https://doi.org/10.1007/978-3-319-47106-8%5C_5.

[10] *Check: framework for unit testing in C*. URL: https://libcheck.github.io/check/.

[11] *JMetal Java framework*. URL: http://jmetal.sourceforge.net/.

[12] Simone Scalabrino et al. "OCELOT: a search-based test-data generation tool for C". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. Ed. by Marianne Huchard, Christian Kästner, and Gordon Fraser. ACM, 2018, pp. 868–871. DOI: 10.1145/3238147.3240477. URL: https://doi.org/10.1145/3238147.3240477.

[13] Dmytro Humeniuk, Foutse Khomh, and Giuliano Antoniol. "A search-based framework for automatic generation of testing environments for cyber-physical systems". In: *Inf. Softw. Technol.* 149 (2022), p. 106936. DOI: 10.1016/j.infsof.2022.106936. URL: https://doi.org/10.1016/j.infsof.2022.106936.

[14] Walid M. Taha, Abd-Elhamid M. Taha, and Johan Thunberg. *Cyber-Physical Systems: A Model-Based Approach*. Springer US, 2021.

[15] Vincenzo Riccio et al. "Testing machine learning based systems: a systematic mapping". In: vol. 25. 6. 2020, pp. 5193–5254. DOI: 10.1007/s10664-020-09881-0. URL: https://doi.org/10.1007/s10664-020-09881-0.

[16] Gordon Fraser and Andrea Arcuri. "Whole Test Suite Generation". In: vol. 39. 2. 2013, pp. 276–291. DOI: 10.1109/TSE.2012.14. URL: https://doi.org/10.1109/TSE.2012.14.

[17] Gordon Fraser and Andrea Arcuri. "Evolutionary Generation of Whole Test Suites". In: *International Conference On Quality Software (QSIC)*. Los Alamitos, CA, USA: IEEE Computer Society, 2011, pp. 31–40. DOI: http://doi.ieeecomputersociety.org/10.1109/QSIC.2011.19.

[18] Andrea Arcuri. "It Does Matter How You Normalise the Branch Distance in Search Based Software Testing". In: *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*. IEEE Computer Society, 2010, pp. 205–214. DOI: 10.1109/ICST.2010.17. URL: https://doi.org/10.1109/ICST.2010.17.

[19] Julia Handl, Simon C. Lovell, and Joshua D. Knowles. "Multiobjectivization by Decomposition of Scalar Cost Functions". In: *Parallel Problem Solving from Nature - PPSN X, 10th International Conference Dortmund, Germany, September 13-17, 2008, Proceedings*. Ed. by Günter Rudolph et al. Vol. 5199. Lecture Notes in Computer Science. Springer, 2008, pp. 31–40. DOI: 10.1007/978-3-540-87700-4_4. URL: https://doi.org/10.1007/978-3-540-87700-4%5C_4.

[20] Derek F. Yates and Nicos Malevris. "Reducing the Effects of Infeasible Paths in Branch Testing". In: *Proceedings of the ACM SIGSOFT '89 Third Symposium on Testing, Analysis, and Verification, TAV 1989, Key West, Florida, USA, December 13-15, 1989*. Ed. by Richard A. Kemmerer. ACM, 1989, pp. 48–54. DOI: 10.1145/75308.75315. URL: https://doi.org/10.1145/75308.75315.

[21] Edmund K. Burke and Graham Kendall. *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Springer US, 2014, pp. 403–449.

[22] A. Author and A. Author. *Book reference example*. Publisher, 2099.

[23] A. Author. "Article title". In: *Journal name* (2099).

[24] *Example*. URL: https://www.isislab.it.

[25] A. Author. "Tesi di esempio ISISLab". 2099.