POLITECNICO
MILANO 1863

Software Engineering 2

Search-based Software Testing

M Camilli, E Di Nitto, M Rossi

Verification & Validation

Search-Based Software Testing

Search-Based Software Testing (SBST)

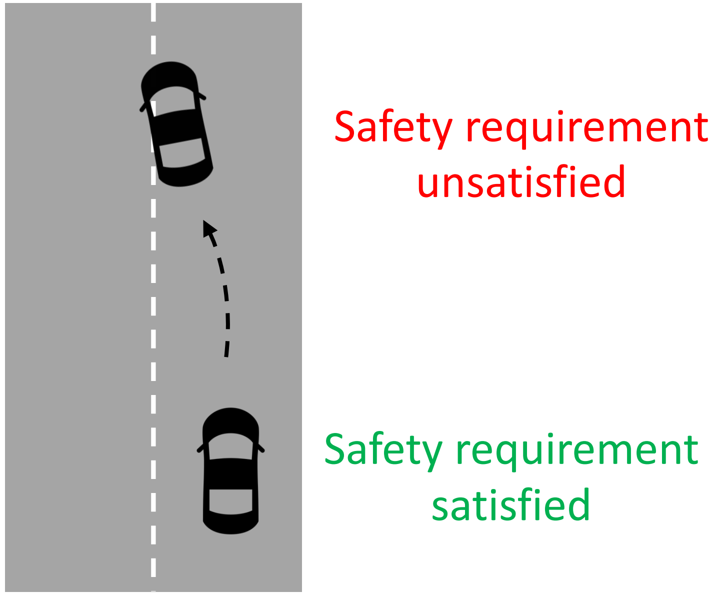
- Complements test case generation techniques seen so far
  - Works at **component** or **system** level
  - Guides the generation toward a **specific testing objective**
  - Compared to fuzzing, typically incorporates **domain-specific knowledge** to generate more **meaningful** test cases
  - Can deal with **functional** and **non-functional** aspects (e.g., reliability, safety)

SBST: the very idea

- Generate **specific test cases** that achieve some test objective
  - Examples of common test objectives
    - Observing wrong/undesired outputs
    - Breaking given requirements
    - Reaching specific source code locations
    - Executing some given paths
- Essence of SBT
  - Recast testing as an optimization problem
    - **Search space** + **fitness** to guide the exploration
  - Generate better and better tests to achieve the objective

SBST: motivating (toy) example

- Autonomous driving
  - Assume you want to test the subsystem that controls the **steering angle** of an autonomous vehicle
  - **Safety requirement**: the vehicle shall always maintain a given safety distance from the centre of the road

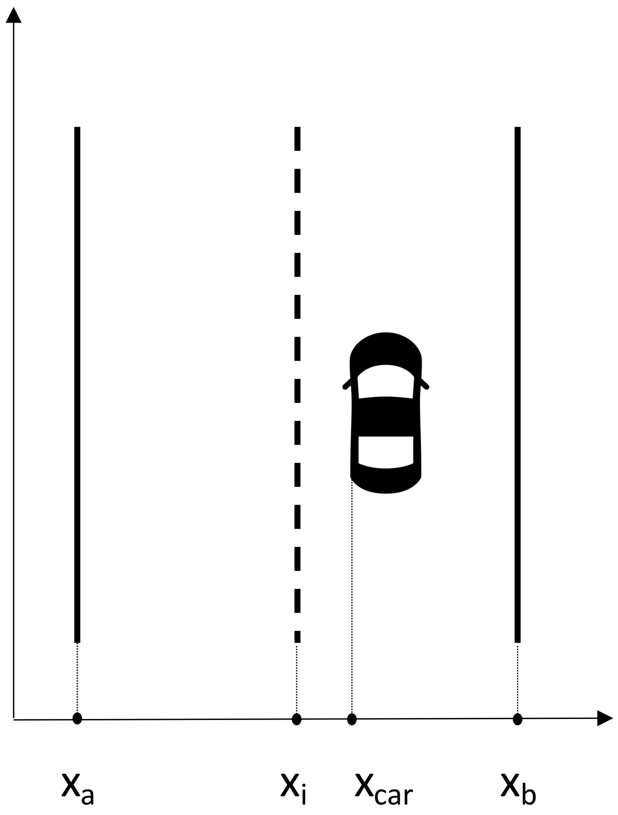


SBST: steps

POLITECNICO
MILANO 1863

1. Identify the objective
2. Define how to measure the distance of the current execution from the objective
    - The distance measures the "*fitness*" of the current execution with respect to the objective
    - The "*current execution*" is identified by the inputs (i.e., the test case) provided to the program
3. Instrument the code to compute the fitness (i.e., the distance) of the current execution (i.e., of the test case) to the objective
4. Pick (randomly) some inputs to run the program (i.e., identify a test cases)
5. Execute the test case, and compute its fitness with respect to the objective
6. If fitness is not sufficient, go to step 4
7. else, we are satisfied (e.g., found a test case that achieves the objective)

Toy example

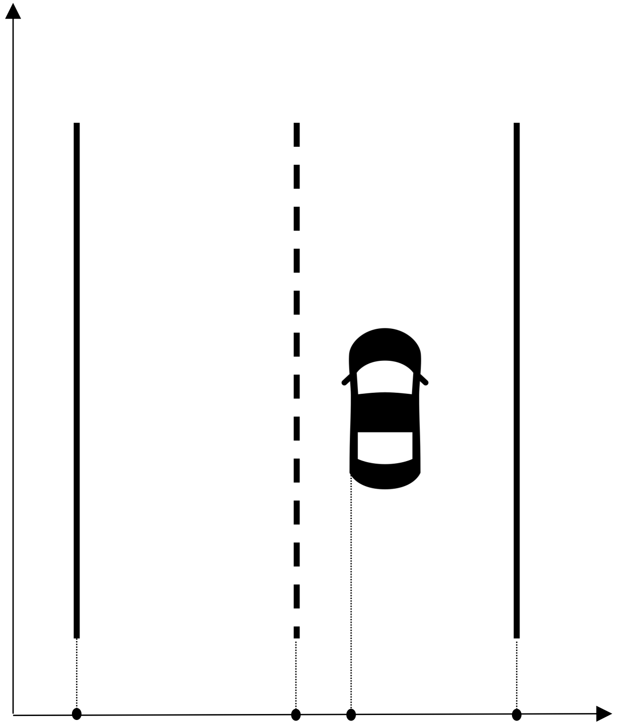



```python
def get_dist_from_middle(xa: int, xb: int, xcar: int):
    xi = (xa+xb)/2
    return xcar-xi
```

- **SBST objective**: identify test cases that make `get_dist_from_middle` return a value ≤ 0
- How to measure the distance from the objective

```python
def calculate_fitness(xa: int, xb: int, xcar: int):
    r = get_dist_from_middle(xa, xb, xcar)
    if r<=0:
        return 0
    return r
```

Toy example




```
def get_dist_from_middle(xa: int, xb: int, xcar: int):
    xi = (xa+xb)/2
    return xcar-xi
```

- **SBST objective**: identify test cases that make `get_dist_from_middle` return a value ≤ 0
- Let's search the input space. We choose a first test case
  - xa: 3, xb: 10, xcar: 7 → return 0.5, distance from goal: 0.5
- We search in the neighborhood decreasing and increasing each parameter
  - xa: 2, xb: 10, xcar: 7 → return 1, distance from goal: 1 → no good
  - xa: 3, xb: 9, xcar: 7 → return 1, distance from goal: 1 → no good
  - xa: 3, xb: 10, xcar: 6 → return -0.5, distance from goal: 0 → great! We have been lucky…

SBST: key points

- How do we identify/define the **objective**?
- How do we measure the **distance** of the current test case to the objective?
- How do we **instrument** the code to retrieve the necessary information to compute the fitness?
- How do we **pick the next test case**, if the current one is not good enough?
  - i.e., how do we search in the space of test cases?

Test Generation as a Search Problem

POLITECNICO MILANO 1863

- Search space
  - Defines **what we are looking for** (test cases)
  - Depends on the testing problem
    - Single integer values, tuples of values, objects, XML documents, …

space of all possible test cases

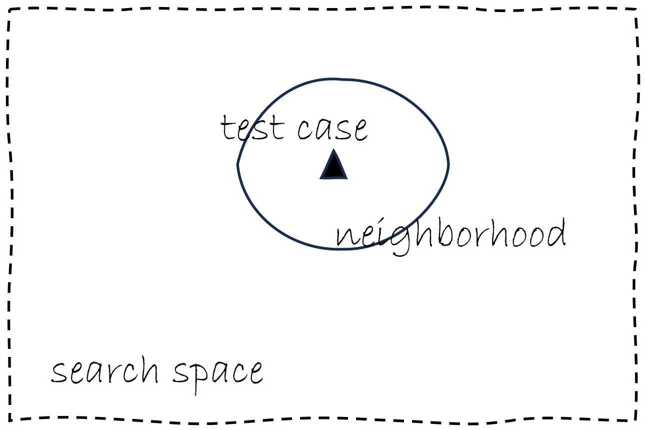
search space

- Example: function `get_dist_from_middle`
  - 3 input parameters, returns a float
  - Search space (representation for test cases) = input tuples (xa, xb, xcar)

```
get_dist_from_middle(3,10,7)   =  0.5
get_dist_from_middle(5,15,8)   = -2.0
get_dist_from_middle(10,22,19) =  3.0
```

Test Generation as a Search Problem

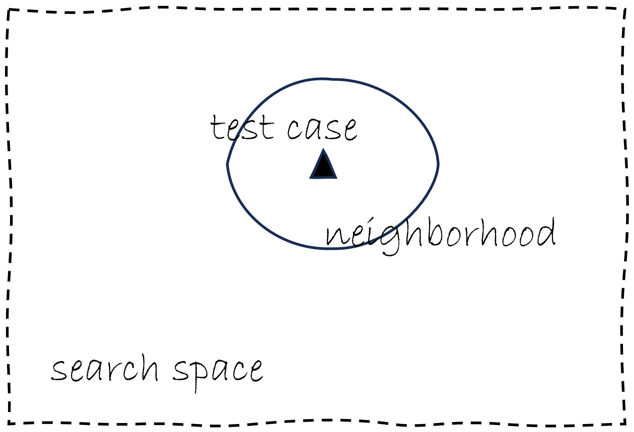
- Search space: neighborhood
  - Each point in the search space has its **neighbors**
  - Neighborhood includes inputs that are related to a given one (e.g., “close” according to some distance function)
- Foo example
  - Test case = input tuple (xa, xb, xcar)
  - Each tuple has 26 neighbors (3*3*3 - 1)
    - xa-1, xb-1, xcar-1
    - xa-1, xb-1, xcar
    - xa-1, xb-1, xcar+1
    - xa-1, xb, xcar-1
    - xa-1, xb+1, xcar-1
    - xa, xb-1, xcar-1
    - xa+1, xb-1, xcar-1
    - …



Test Generation as a Search Problem

POLITECNICO MILANO 1863

- Given **search space** + **neighborhood relation** we can define
  - Fitness function → defines the "goodness" of a given test case (candidate solution)
  - Algorithm → explores the neighborhood using the heuristic to steer the search



Test Generation as a Search Problem

- Fitness function
  - “goodness” of an individual = **fitness**
    - Maps a test case to a **numeric value** (the better the candidate the better the value)
  - Depends on the **objective** of testing!

- Example: function `get_dist_from_middle`

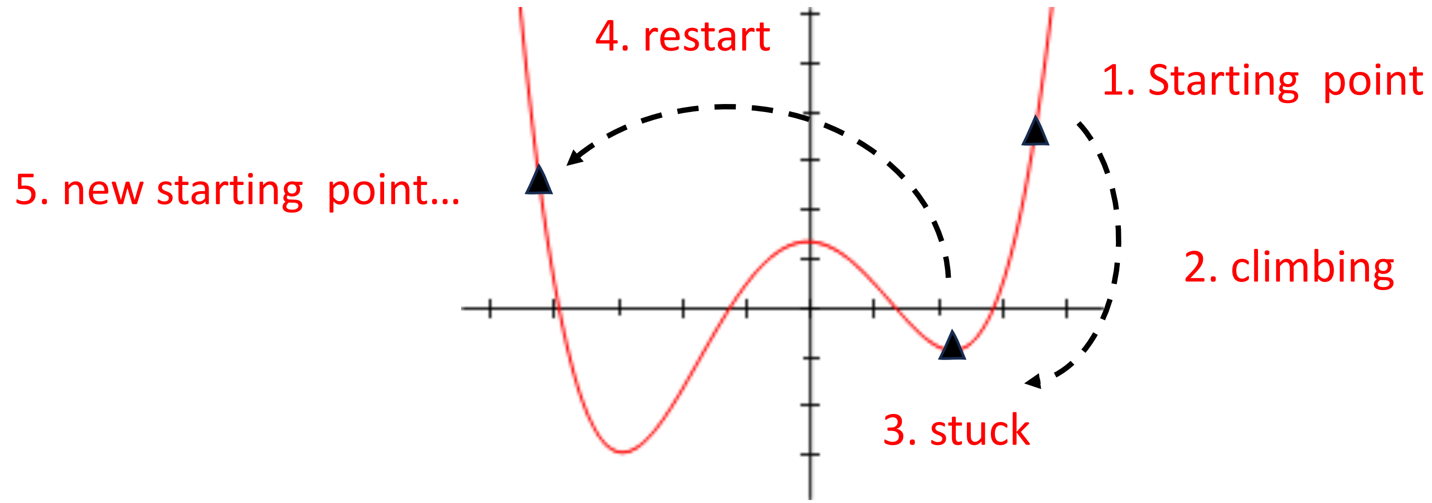
```python
def calculate_fitness(xa: int, xb: int, xcar: int):
    r = get_dist_from_middle(xa, xb, xcar)
    if r <= 0:
        return 0
    return r
```

Test Generation: Hillclimbing

- Search algorithm - Hillclimbing (simple meta-heuristic algorithm)
  1. Take a random starting point
  2. Determine fitness value of all neighbors
  3. Move to neighbor with the best fitness value
  4. If solution is not found, continue with step 2

Test Generation: Hillclimbing

- Escaping local optima
  - The easiest way is to give up and restart from a new random point



Test Generation: Evolutionary Search

POLITECNICO
MILANO 1863

- Hillclimbing works well if
  - Reasonably **small** neighborhood and search space
- Assume we have a program that receives UTF-16 strings (max len 10) as input:
  - Each char is encoded with 16 bits = 65536 possible encodings
  - What is the size of the search space?
  - Hillclimbing would take **unreasonably long time**! Why?
- Global search
  - Hillclimbing searches locally only!
  - We should make larger steps to extend the search "more globally"

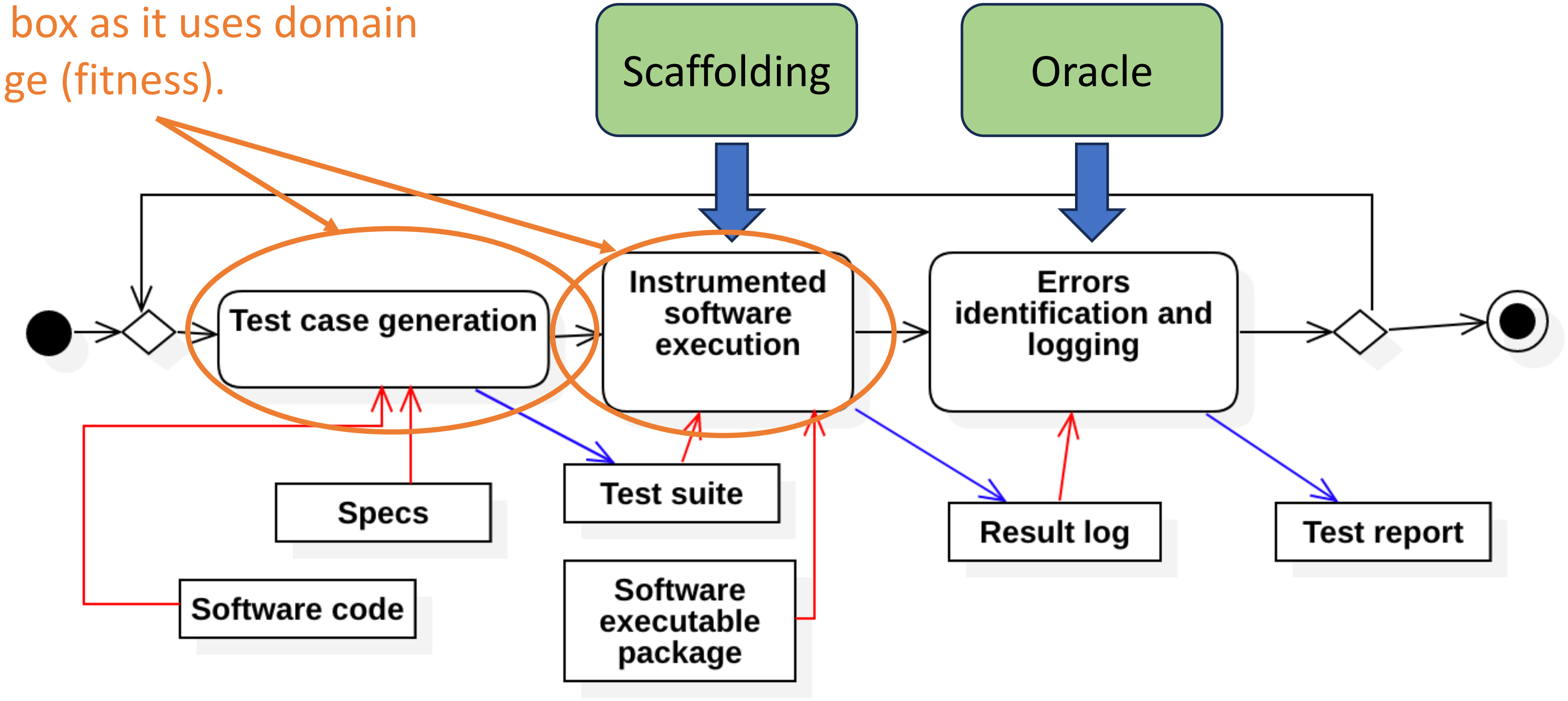
Test Generation: Evolutionary Search

- Global search
  - We can use (again) the notion of **mutation**
  - It can be used to carry out larger steps
    - Shall not completely change an individual
    - Shall keep most of its “traits”
  - **Examples**
    - Mutation for **strings**: flip random char, add/remove a random char, …
    - Mutation for **integers**: sum a small random number, flip some random bits in the binary encoding, …

Positioning SBST in the testing workflow

POLITECNICO MILANO 1863

SBT introduces automation here.
It is gray box as it uses domain knowledge (fitness).



SBT: Summary

- Strengths
  - Compared to concolic guides the generation toward a **specific testing objective** (e.g., wrong outputs, coverage of given branches)
  - Compared to fuzzing typically generate more **meaningful** test cases
  - Eventually reaches the objective with enough budget
- Weaknesses
  - Requires domain-knowledge to define the notion of fitness (nontrivial)
  - Heavily relies on the quality of the heuristics to explore the neighborhood
  - Computationally expensive and time-consuming

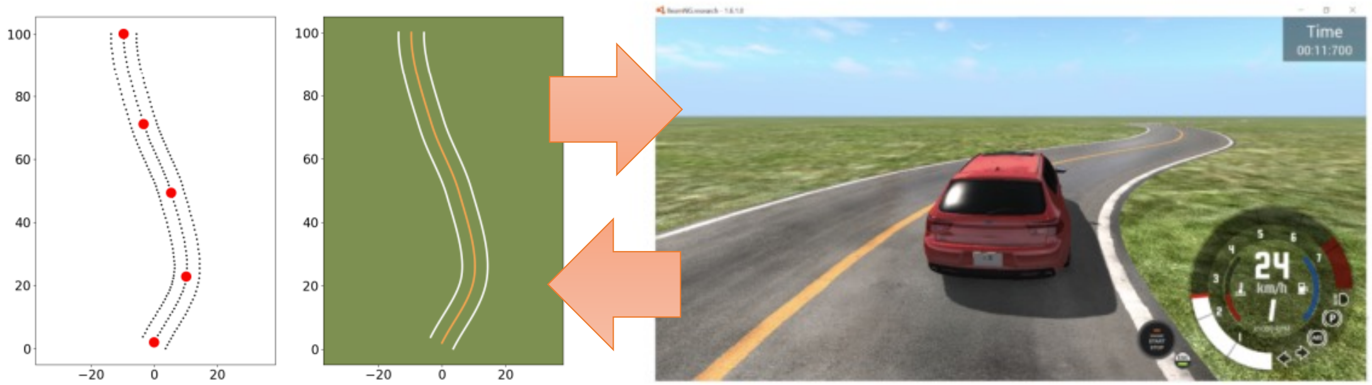
SBST: tools

- EvoSuite
  - Tool that automatically generates test cases for Java code
  - Automatically instruments the sources
  - Generates test suites towards satisfying a given objective (i.e., coverage criterion)
  - https://www.evosuite.org/

EVOSUITE
Automatic Test Suite Generation for Java

SBST: latest trends

- Automated testing of autonomous systems
  - **Example**: search for road shapes where the car cannot keep the lane (testing objective = invalidate safety requirements)



Test case generation
road shape

Scenario simulation
fitness calculation given the generated road

https://dl.acm.org/doi/10.1145/3368089.3409730

References

- Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler: "The Fuzzing Book". Retrieved 2023-11. https://www.fuzzingbook.org/
- Fraser, Gordon, and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. Proceedings of the 19th ACM SIGSOFT symposium and the 13th FSE. 2011. https://dl.acm.org/doi/abs/10.1145/2025113.2025179
- Evo Suite tool: https://www.evosuite.org/
- Vincenzo Riccio and Paolo Tonella. 2020. Model-based exploration of the frontier of behaviours for deep learning system testing. In Proceedings of the 28th ACM ESEC/FSE 2020. ACM, New York, NY, USA, 876–888. https://doi.org/10.1145/3368089.3409730