FONDAZIONE BRUNO KESSLER

SE SOFTWARE ENGINEERING

Search Based Test-case Generation

Paolo Tonella

Software Engineering Research Unit

Fondazione Bruno Kessler

Trento, Italy

http://se.fbk.eu/tonella

FONDAZIONE BRUNO KESSLER

SE SOFTWARE ENGINEERING

Outline

- Search based algorithms
- Automated input data generation
- Object oriented testing
- Dynamic symbolic execution

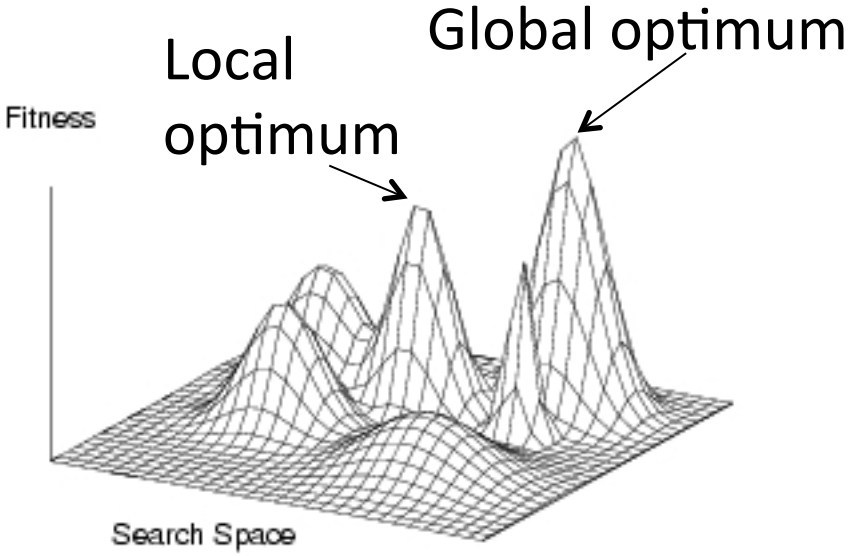
Search based algorithms

The search problem

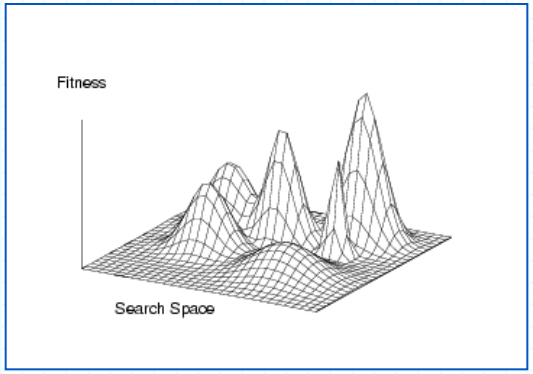
Find a value x^* which maximizes (minimizes) the objective (fitness) function f over the search space X:

$f: X \to \mathbb{R}$

$x^*: \forall x \in X, f(x^*) \geq f(x)$



Search algorithms



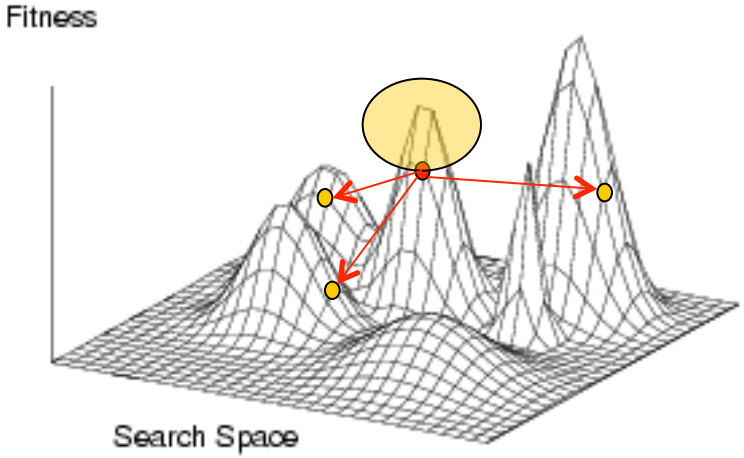
- Exhaustive search
- Random search
- Metaheuristic search

Metaheuristic algorithm: iterative optimization of candidate solutions for arbitrary problem instances. It is not ensured to find a global optimum, it finds a so-called *near-optimal solution*.

E.g.: find input that crashes `myFunc(int a, int b, int c){…}`

$$|X| = 2^{32} \times 2^{32} \times 2^{32} = 10^{28}$$

Exploration vs. exploitation



Since the search budget is finite, metaheuristic algorithms aim for a balance between:

- **local search**: exploitation (intensification);
- **global search**: exploration (diversification).

Search budget: max number of fitness evaluations (number of samples *x* taken) compatible with the max algorithm execution time.

No free lunch theorems

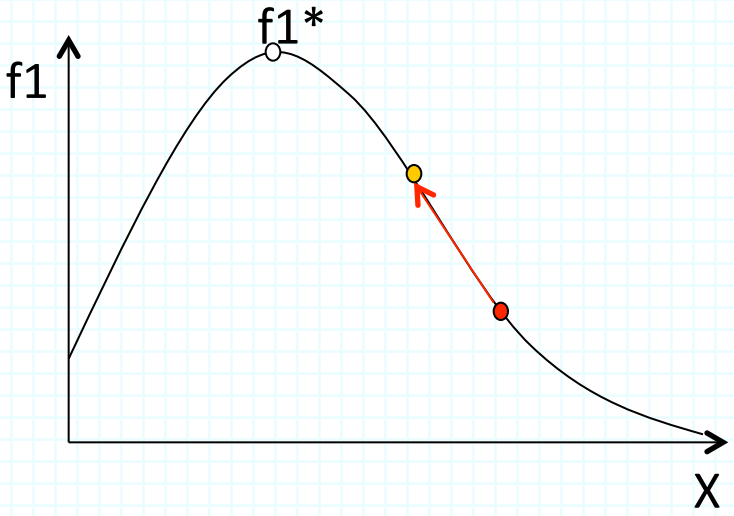
$$\sum_f P(f^*|f,m,A) = \sum_f P(f^*|f,m,B)$$

NFL theorem: given a search budget *m*, the average probability of obtaining the near-optimal value *f** using algorithm *A* is the same as the probability of obtaining the same near-optimal value using another, arbitrarily chosen algorithm *B*.

Hence, if metaheuristic *A* performs better than random search on problem instance *f1*, there will be another problem instance *f2* on which random performs better than *A*.

David H. Wolpert, William G. Macready: *No Free Lunch Theorems for Optimization*, IEEE Transactions on Evolutionary Computation, vol. 1, n. 1, April 1997.

Deceptive fitness function



$P(f1^*|f1, m, HC) = 1$

$P(f1^*|f1, m, RND) = |m| / |X|$

$P(f2^*|f2, m, HC) = |\varepsilon| / |X|$

$P(f2^*|f2, m, RND) = |m| / |X|$

HC = hill climbing (steepest ascent)

RND = random search

FONDAZIONE BRUNO KESSLER

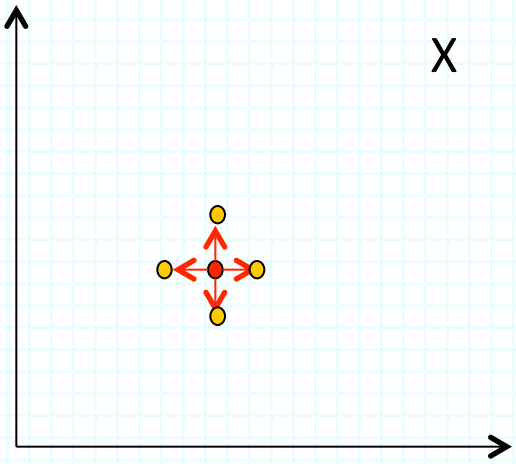
SE SOFTWARE ENGINEERING

Metaheuristic algorithms

- Hill climbing
- Simulated annealing
- Tabu search
- Genetic algorithms
- Ant colony optimization
- Particle swarm optimization

Hill climbing

```
1.   x_opt = x0 // random
2.   max = f(x_opt)
3.   LOOP // with search budget m
4.     improved = FALSE
5.     FOR x IN Neighbors(x_opt)
6.       IF f(x) > max
7.         max = f(x)
8.         x_opt = x
9.         improved = TRUE
10.      END IF
11.    END FOR
12.    IF NOT improved
13.      RETURN x_opt
14.    END IF
15.  END LOOP
16.  RETURN x_opt
```



E.g.: find input that crashes

`myFunc(int a, int b, int c){…}`

E.g.: (a±1, b±1, c±1)

Equivalent to greedy or steepest ascent: exploitation, no exploration.

Hill climbing variants

- **Stochastic hill climbing**: selects randomly among the improving solutions in the neighborhood.
- **Random-restart (shotgun) hill climbing**: randomly restarts when no improving solution is found (or the improvement is small).

They attempt to add more exploration, when local search is ineffective.

Simulated annealing

```
1.   x_opt = x0 // random
2.   k = 1       // time
3.   T = T_max   // temperature
4.   x = x_opt
5.   LOOP // with search budget m
6.     x_new = RNDNeighbor(x)
7.     Df = f(x_new) - f(x)
8.     IF Df > 0
9.        x_opt = x_new
10.    END IF
11.    IF Df > 0 OR
12.       P(Df, T) > rand(0, 1)
13.       x = x_new
14.    END IF
15.    T = cool(T, k)
16.    k = k + 1
17.  END LOOP
18.  RETURN x_opt
```

Parameters:

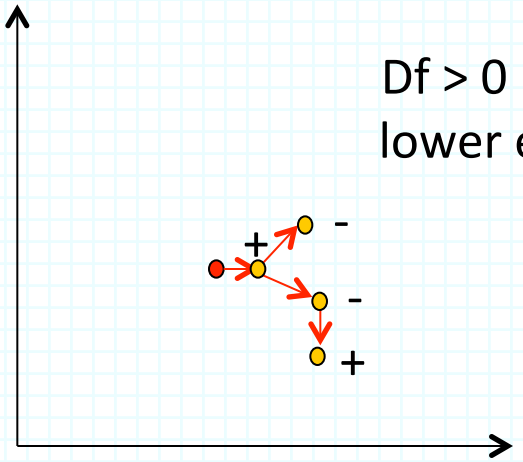
E.g.:

P(Df, T) = exp(Df / T)

if Df < 0; 1 otherwise

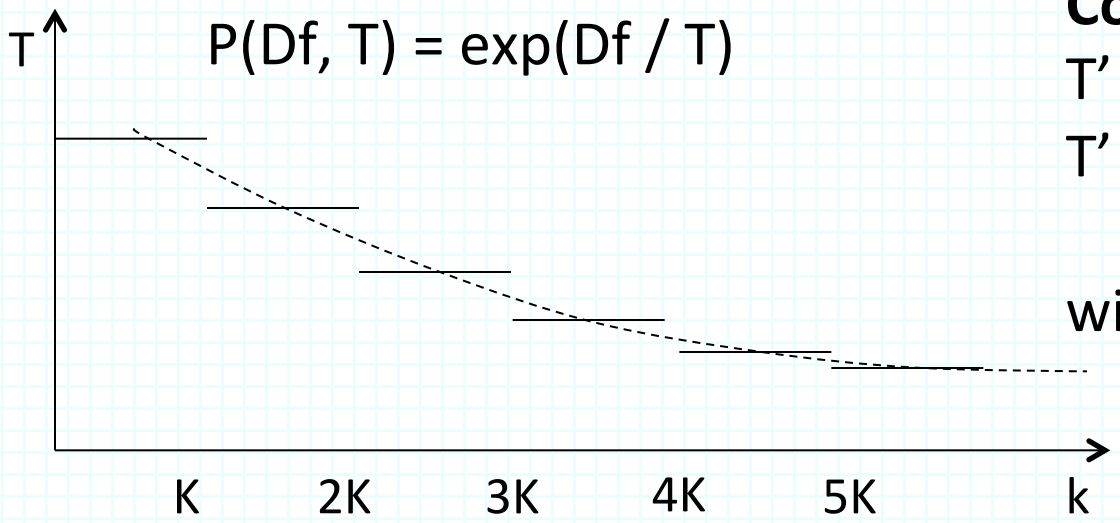
cool(T, k) = α T

if k % K = 0; T otherwise



Simulated annealing

- **Initial high temperature**: initially, exploration is privileged over exploitation: worsening solutions are initially accepted with high probability
- **Low temperature at the end**: only improving solutions are accepted at the end of the search (similar to hill climbing).



Cooling schedule:

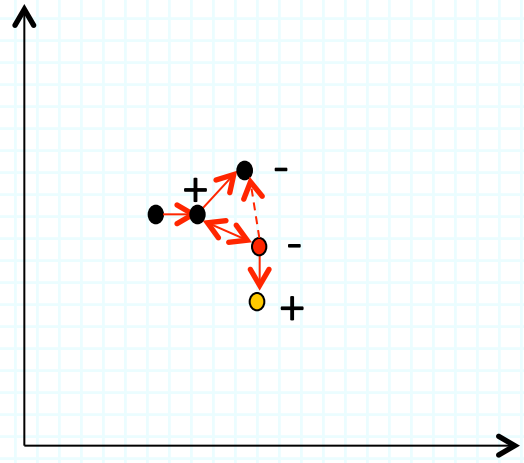
$T' = \alpha T$ if $k \% K = 0$

$T' = T$ otherwise

with $0 \leq \alpha \leq 1$

Tabu search

```
1.   x_opt = x0 // random
2.   add(tabuFIFOList, x_opt)
3.   LOOP // with search budget m
4.     x_new = RNDMutate(x_opt)
5.     IF x_new NOT IN tabuFIFOList
6.        add(tabuFIFOList, x_new)
7.        IF f(x_new) > f(x_opt)
8.           x_opt = x_new
9.        END IF
10.    END IF
11.  END LOOP
12.  RETURN x_opt
```



Local search (exploitation), optimized to avoid revisiting previously examined solutions

Tabu search variants

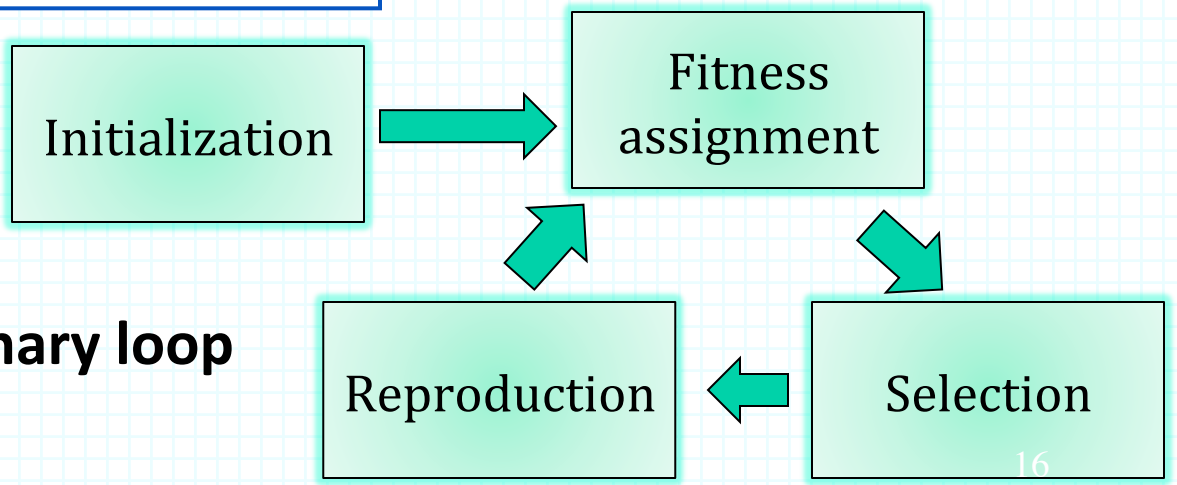
- **Neighborhood tabu**: the entire neighborhood of an already visited solution is declared tabu; mutations jump beyond such neighborhood.
- **Property based tabu list**: tabu solutions are characterized by their properties, such that any (even non visited) solution with the properties of a tabu solution is considered tabu as well.
- **Reactive tabu search**: when the same (tabu) solutions reappear frequently during the search, a random walk is made to explore a different portion of the search space.

Genetic algorithms

```
1.   Pop = {x0, …, xN} // random
2.   x_opt = NULL
3.   LOOP // with search budget m
4.     assignFitness(Pop)
5.     x_opt = best(Pop, x_opt)
6.     Pop = select(Pop)
7.     Pop = reproduce(Pop)
8.   END LOOP
9.   RETURN x_opt
```

Key decisions:

1. Representation
2. Fitness function
3. Selection
4. Reproduction (crossover and mutation)



Evolutionary loop

Representation

Fixed length chromosome:

y_1	y_2	y_3	y_4	y_5	y_6	…	y_k

y_i may be a single bit, a number, a character, etc.

E.g.: find input that crashes `myFunc(int a, int b, int c){…}`
Chromosomes = (2, 0, 3), (0, 0, -1)

Intron: portion of the chromosome that does not contribute to the phenotype (i.e., the fitness value)

E.g.: find input that crashes `myFunc(char* s){…}`
Chromosomes = (‘a’, ‘b’, ‘c’, ‘\0’, ‘t’), (‘z’, ‘\0’, ‘a’, ‘g’, ‘h’)

Variable length chromosome: *k* varies across individuals

Chromosomes = (‘a’, ‘b’, ‘c’, ‘\0’), (‘z’, ‘\0’)

Selection

Pop = select(Pop)

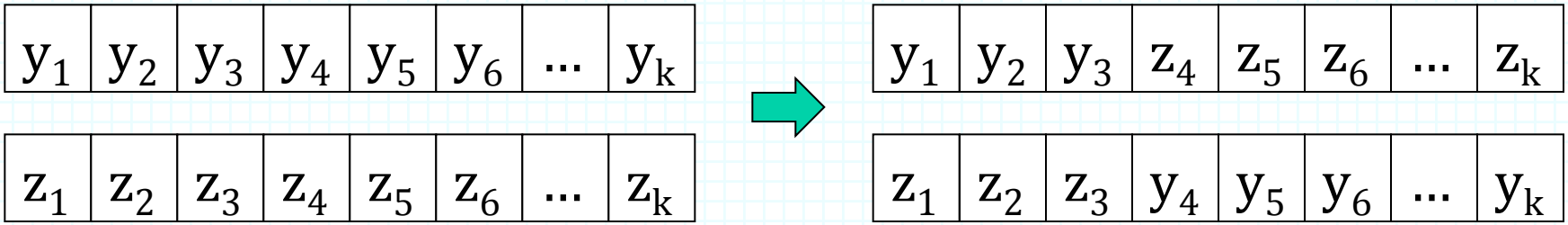
- **Truncation selection**: the best *M* individuals are selected; to obtain the number of individuals required to fill-in *Pop*, the selected individuals are replicated multiple times.
- **Roulette wheel selection**: the probability of selecting an individual is proportional to its fitness; individuals can be selected multiple times.
- **Tournament selection**: *T* individuals (e.g., 2) are randomly selected and compared (according to their fitness); the winner is inserted into the new population.
- **Ordered selection**: the probability of selecting an individual is proportional to its position in the list of all individuals ranked by increasing fitness.
- **Elitism**: at least one copy of the best individual(s) is propagated to the next generation.

Reproduction

Pop = reproduce(Pop)

Crossover: $(y'_i, y'_j) = y_i \otimes y_j$

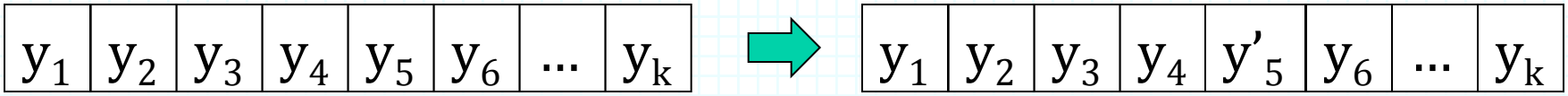
E.g., P = 80%



Single point crossover

Mutation: $y'_i = \mu(y_i)$

E.g., P = 10%



Genetic algorithms

The parameters of these algorithms (population size, kind of selection, crossover and mutation rates, etc.) and the design of the mutation/crossover operators determine the trade-off between exploration and exploitation. E.g.:

- Neighborhood mutation / two-parent crossover / low mutation and crossover rate / elitism => **local search**.
- Disruptive mutation / multi-parent crossover / high mutation and crossover rate / ordered selection => **global search**.

FONDAZIONE BRUNO KESSLER

SE SOFTWARE ENGINEERING

Automated input data generation

Phil McMinn, *Search-based software test data generation: a survey*. Journal of Software Testing, Verification and Reliability, vol. 14, n. 2, pp. 105-156, June 2004.

FONDAZIONE BRUNO KESSLER

SE SOFTWARE ENGINEERING

Running example

```
class Triangle {
  int a, b, c; // sides
  int type = NOT_A_TRIANGLE;

  Triangle(int a, int b, int c) {…}
  void checkRightAngle(){…}
  void computeTriangleType() {…}
  boolean isTriangle() {…}
  public static void main(String args[]) {
    int a = Integer.parseInt(args[0]);
    int b = Integer.parseInt(args[1]);
    int c = Integer.parseInt(args[2]);
    Triangle t = new Triangle(a, b, c);
    if (t.isTriangle())
       t.computeTriangleType();
    System.out.println(typeToString(t.type));
  }
}
```

Running example

```
class Triangle {
  int a, b, c; // sides
  int type = NOT_A_TRIANGLE;

  Triangle(int a, int b, int c) {…}
  void checkRightAngle() {…}
  void computeTriangleType() {…}
  boolean isTriangle() {
    if (a <= 0 || b <= 0 || c <= 0)
        return false;
    if (a + b <= c || a + c <= b || b + c <= a)
        return false;
    return true;
  }
  public static void main(String args[]) {…}
}
```

Running example

```
class Triangle {
  int a, b, c; // sides
  int type = NOT_A_TRIANGLE;

  Triangle(int a, int b, int c) {…}
  void checkRightAngle() {…}
  void computeTriangleType() {
    if (a == b)
        if (b == c) type = EQUILATERAL;
        else type = ISOSCELE;
    else if (a == c) type = ISOSCELE;
    else if (b == c) type = ISOSCELE;
    else checkRightAngle();
  }
  boolean isTriangle() {…}
  public static void main(String args[]) {…}
}
```

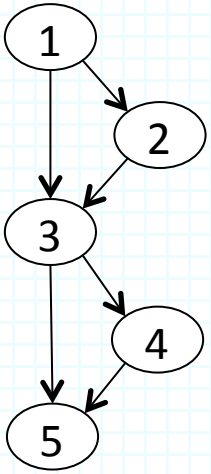
Running example

```
class Triangle {
  int a, b, c; // sides
  int type = NOT_A_TRIANGLE;

  Triangle(int a, int b, int c) {…}
  void checkRightAngle() {
    if (a*a + b*b == c*c)
        type = RIGHT_ANGLE;
    else if (b*b + c*c == a*a)
        type = RIGHT_ANGLE;
    else if (a*a + c*c == b*b)
        type = RIGHT_ANGLE;
    else type = SCALENE;
  }
  void computeTriangleType() {…}
  boolean isTriangle() {…}
  public static void main(String args[]) {…}
}
```

Coverage testing

```
void f(int a, int b) {
1    if (a < 0)
2        print("a is negative");
3    if (b < 0)
4        print("b is negative");
5    return;
}
```

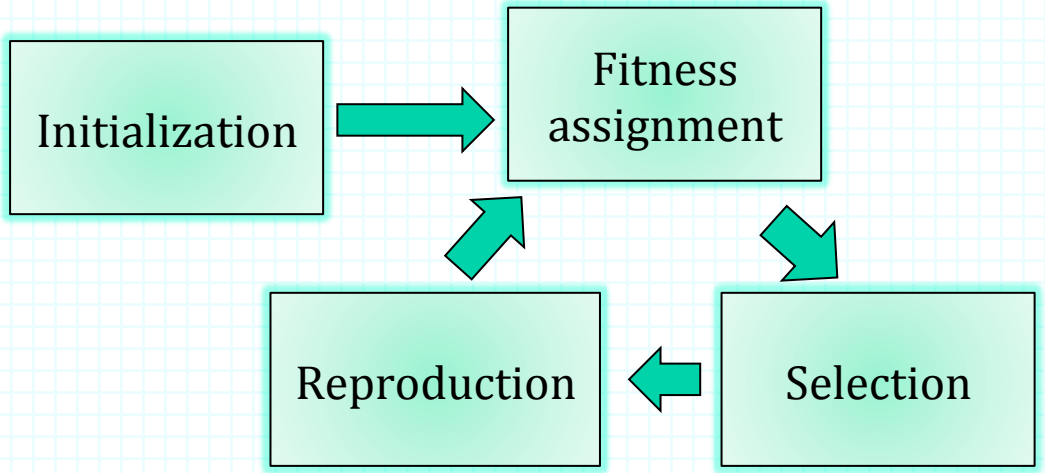


	Coverage targets	Test cases
Path coverage	{<1, 3, 5>, <1, 2, 3, 5>, <1, 3, 4, 5>, <1, 2, 3, 4, 5>}	f(-1, -1), f(1, -1) f(-1, 1), f(1, 1)
Branch coverage	{<1, 2>, <2, 3>, <1, 3>, <3, 4>, <4, 5>, <3, 5>}	f(-1, -1), f(1, 1)
Statement coverage	{1, 2, 3, 4, 5}	f(-1, -1)

Running example

```
class Triangle {
  int a, b, c; // sides
  int type = NOT_A_TRIANGLE;
  Triangle(int a, int b, int c) {…}
  void checkRightAngle(){…}
  void computeTriangleType() {…}
  boolean isTriangle() {…}
  public static void main(String args[]){…}
}
```

Goal:
Automatic generation of test cases using genetic algorithms so as to achieve statement coverage.



Key decisions:

1. Representation
2. Fitness function
3. Selection
4. Reproduction (crossover and mutation)

Representation

The chromosome used for test case generation is the input vector, (sequence of input values to be provided by the test case running the program) which may be fixed length or variable length.

E.g., in our running example:

Fixed length chromosome

a	b	c

Example of (randomly generated) initial population:
Pop = {(2, 2, -2), (2, 1, 1), (2, 1, 2), (3, 4, 2), (4, 3, 3), (4, -5, 6), (3, 5, 2) (-3, 0, -2)}

Fitness function

For statement and branch coverage, given a specific coverage target *t*, a widely used fitness function (to be minimized) is:

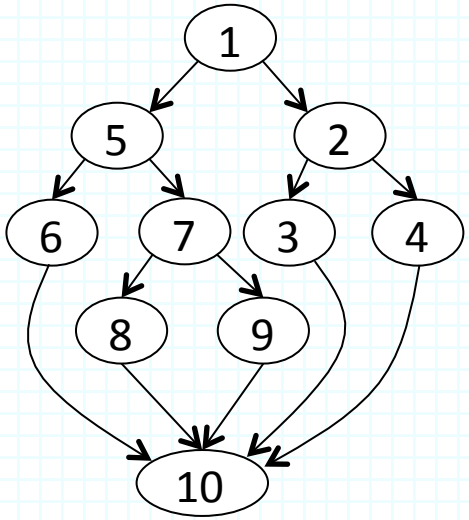
f(x) = approach_level(P(x), t) + branch_distance(P(x), t)

approach_level(P(x), t) :
Given the execution trace obtained by running program *P* with input vector *x*, the approach level is the minimum number of control nodes between an executed statement and the coverage target *t*.

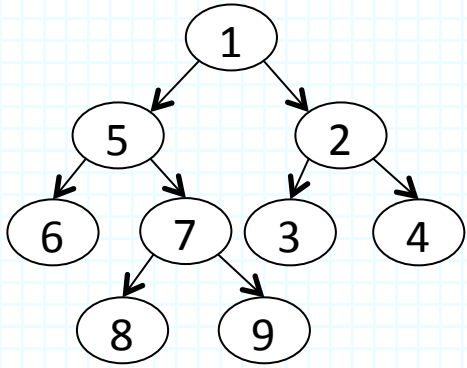
branch_distance(P(x), t)
Given the first control node where the execution diverges from the target *t*, the predicate at such node is converted to a distance (from taking the desired branch), normalized between 0 and 1.

Approach level

```
class Triangle {
  void computeTriangleType() {
1     if (a == b)
2        if (b == c)
3           type = EQUILATERAL;
4        else type = ISOSCELE;
5     else if (a == c)
6           type = ISOSCELE;
7     else if (b == c)
8           type = ISOSCELE;
9     else checkRightAngle();
10    return;
  }
}
```



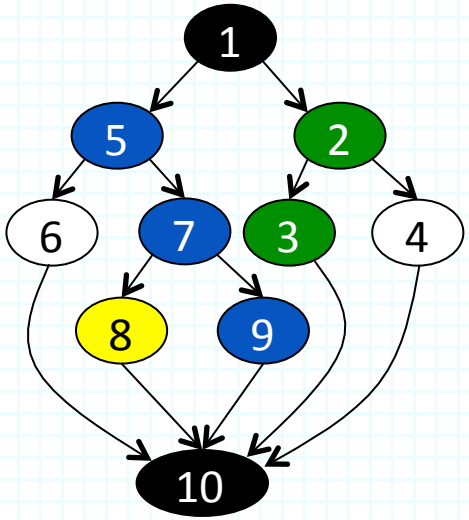
Control flow graph



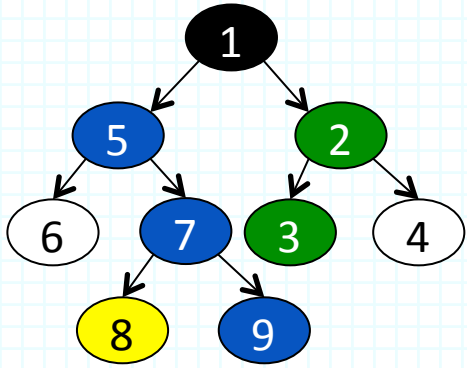
Control dependency graph

Approach level

```
class Triangle {
  void computeTriangleType() {
1     if (a == b)
2        if (b == c)
3           type = EQUILATERAL;
4        else type = ISOSCELE;
5     else if (a == c)
6           type = ISOSCELE;
7     else if (b == c)
8           type = ISOSCELE;
9     else checkRightAngle();
10    return;
  }
}
```



Control flow graph



Control dependency graph

Ch1 = (2, 2, 2) P(Ch1) = <1, 2, 3, 10> AL = 2

Ch2 = (2, 3, 4) P(Ch2) = <1, 5, 7, 9, 10> AL = 0

Branch distance

The predicate of the control node which is closest to the target is converted to a distance, which measures how far the test case is from taking the desired branch. For boolean/numeric variables a, b:

Condition c = atomic predicate	Distance BD(c) = d / (d + 1)
a	d = {0 if a == true; K otherwise}
!a	d = {K if a == true; 0 otherwise}
a == b	d = {0 if a == b; abs(a - b) + K otherwise}
a != b	d = {0 if a != b; K otherwise}
a < b	d = {0 if a < b; a - b + K otherwise}
a <= b	d = {0 if a <= b; a - b + K otherwise}
a > b	d = {0 if a > b; b - a + K otherwise}
a >= b	d = {0 if a >= b; b - a + K otherwise}

Branch distance

For string variables a, b:

Condition c = atomic predicate	Distance BD(c) = d / (d + 1)
a == b	d = {0 if a == b; edit_dist(a, b) + K otherwise}
a != b	d = {0 if a != b; K otherwise}
a < b	d = {0 if a < b; a[j] - b[j] + K otherwise}
a <= b	d = {0 if a <= b; a[j] - b[j] + K otherwise}
a > b	d = {0 if a > b; b[j] - a[j] + K otherwise}
a >= b	d = {0 if a >= b; b[j] - a[j] + K otherwise}

where j is the position of the first different character: *a[j] != b[j],* while *a[i] == b[i]* for *i < j* (*a[j] –b[j]* is set to zero if *a == b*).

Example of edit distance: edit_dist(“strqqvt”, “trwwv”) = 6

Branch distance

Condition c = composite predicate	Distance BD(c) = d / (d + 1)
!p	Negation is propagated inside p
p & q	d = d(p) + d(q)
p | q	d = min(d(p), d(q))
p XOR q = p & !q | !p & q	d = min(d(p)+d(!q), d(!p)+d(q))

Alternative normalizations of *d*:

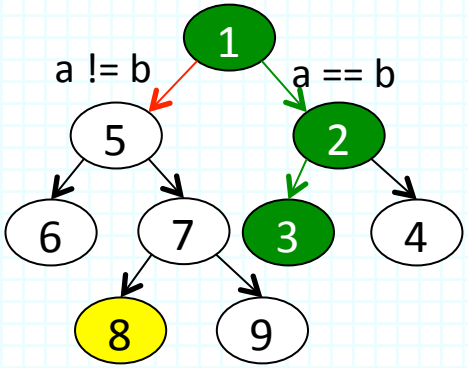
$BD(c) = 1 - \alpha^{-d}$

$BD(c) = d / (d + \beta)$

with $\alpha > 1$ and $\beta > 0$

Branch distance

```
class Triangle {
  void computeTriangleType() {
1     if (a == b)
2        if (b == c)
3           type = EQUILATERAL;
4        else type = ISOSCELE;
5     else if (a == c)
6           type = ISOSCELE;
7     else if (b == c)
8           type = ISOSCELE;
9     else checkRightAngle();
10    return;
  }
}
```



d(a != b) = K = 1

BD(a != b) = 1 / (1 + 1) = 0.5

f(Ch1) = 2 + 0.5 = 2.5

Ch1 = (2, 2, 2)

Ch2 = (2, 3, 4)

P(Ch1) = <1, 2, 3, 10>

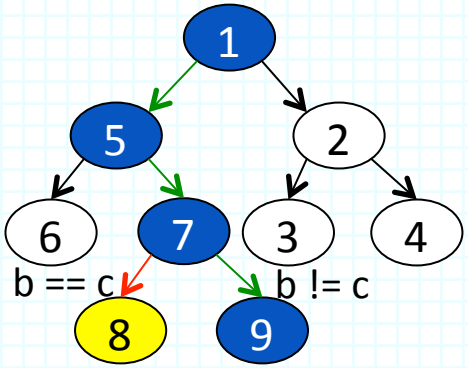
P(Ch2) = <1, 5, 7, 9, 10>

AL = 2 f = 2.5

AL = 0

Branch distance

```
class Triangle {
  void computeTriangleType() {
1     if (a == b)
2        if (b == c)
3           type = EQUILATERAL;
4        else type = ISOSCELE;
5     else if (a == c)
6           type = ISOSCELE;
7     else if (b == c)
8           type = ISOSCELE;
9     else checkRightAngle();
10    return;
  }
}
```



d(b == c) = abs(b - c) + K = 2

BD(b == c) = 2 / (2 + 1) = 0.66

f(Ch2) = 0 + 0.66 = 0.66

Ch1 = (2, 2, 2)

Ch2 = (2, 3, 4)

P(Ch1) = <1, 2, 3, 10>

P(Ch2) = <1, 5, 7, 9, 10>

AL = 2 f = 2.5

AL = 0 f = 0.66

Selection

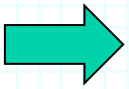
Ch1 = (2, 2, 2) f = 2.5
Ch2 = (2, 3, 4) f = 0.66
Ch3 = (-2, 3, 6) f = ∞
Ch4 = (2, 3, 7) f = ∞
Ch5 = (2, 2, 3) f = 2.5
Ch6 = (3, 4, 5) f = 0.66
Ch7 = (3, 5, 7) f = 0.75
Ch8 = (6, 8, 4) f = 0.83

```
class Triangle {
  void computeTriangleType() {
1     if (a == b)
2        if (b == c)
3           type = EQUILATERAL;
4        else type = ISOSCELE;
5     else if (a == c)
6           type = ISOSCELE;
7     else if (b == c)
8           type = ISOSCELE;
9     else checkRightAngle();
10    return;
  }
}
```

Selection

Truncation selection

Ch6 = (3, 4, 5) f = 0.66
Ch2 = (2, 3, 4) f = 0.66
Ch7 = (3, 5, 7) f = 0.75
Ch8 = (6, 8, 4) f = 0.83
Ch1 = (2, 2, 2) f = 2.5
Ch5 = (2, 2, 3) f = 2.5
Ch3 = (-2, 3, 6) f = ∞
Ch4 = (2, 3, 7) f = ∞



Ch6 = (3, 4, 5)
Ch2 = (2, 3, 4)
Ch7 = (3, 5, 7)
Ch8 = (6, 8, 4)
Ch6 = (3, 4, 5)
Ch2 = (2, 3, 4)
Ch7 = (3, 5, 7)
Ch8 = (6, 8, 4)

Selection

Roulette wheel selection

Ch6 = (3, 4, 5) P ≈ 1/f = 0.23
Ch2 = (2, 3, 4) P ≈ 1/f = 0.23
Ch7 = (3, 5, 7) P ≈ 1/f = 0.20
Ch8 = (6, 8, 4) P ≈ 1/f = 0.18
Ch1 = (2, 2, 2) P ≈ 1/f = 0.06
Ch5 = (2, 2, 3) P ≈ 1/f = 0.06
Ch3 = (-2, 3, 6) P ≈ 1/f = 0
Ch4 = (2, 3, 7) P ≈ 1/f = 0

Ch6 = (3, 4, 5)
Ch2 = (2, 3, 4)
Ch6 = (3, 4, 5)
Ch1 = (2, 2, 2)
Ch8 = (6, 8, 4)
Ch2 = (2, 3, 4)
Ch7 = (3, 5, 7)
Ch6 = (3, 4, 5)

Selection

Roulette wheel selection with elitism

Ch6 = (3, 4, 5) P ≈ 1/f = 0.23
Ch2 = (2, 3, 4) P ≈ 1/f = 0.23
Ch7 = (3, 5, 7) P ≈ 1/f = 0.20
Ch8 = (6, 8, 4) P ≈ 1/f = 0.18
Ch1 = (2, 2, 2) P ≈ 1/f = 0.06
Ch5 = (2, 2, 3) P ≈ 1/f = 0.06
Ch3 = (-2, 3, 6) P ≈ 1/f = 0
Ch4 = (2, 3, 7) P ≈ 1/f = 0

Ch6 = (3, 4, 5)
Ch1 = (2, 2, 2)
Ch8 = (6, 8, 4)
Ch2 = (2, 3, 4)
Ch7 = (3, 5, 7)
Ch6 = (3, 4, 5)

Elite

Ch6 = (3, 4, 5)
Ch2 = (2, 3, 4)

Selection

Tournament selection

Ch1 = (2, 2, 2) f = 2.5	<Ch2, Ch7>	Ch2 = (2, 3, 4)
Ch2 = (2, 3, 4) f = 0.66	<Ch1, Ch5>	Ch5 = (2, 2, 3)
Ch3 = (-2, 3, 6) f = ∞	<Ch3, Ch8>	Ch8 = (6, 8, 4)
Ch4 = (2, 3, 7) f = ∞	<Ch3, Ch2>	Ch2 = (2, 3, 4)
Ch5 = (2, 2, 3) f = 2.5	<Ch6, Ch5>	Ch6 = (3, 4, 5)
Ch6 = (3, 4, 5) f = 0.66	<Ch6, Ch4>	Ch6 = (3, 4, 5)
Ch7 = (3, 5, 7) f = 0.75	<Ch8, Ch3>	Ch8 = (6, 8, 4)
Ch8 = (6, 8, 4) f = 0.83	<Ch8, Ch1>	Ch8 = (6, 8, 4)

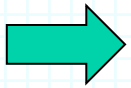
Selection

				Ordered selection
1	Ch4 = (2, 3, 7)	f = ∞	P = 1/33 = 0.03	Ch2 = (2, 3, 4)
1	Ch3 = (-2, 3, 6)	f = ∞	P = 1/33 = 0.03	Ch2 = (2, 3, 4)
3	Ch5 = (2, 2, 3)	f = 2.5	P = 3/33 = 0.09	Ch6 = (3, 4, 5)
3	Ch1 = (2, 2, 2)	f = 2.5	P = 3/33 = 0.09	Ch4 = (2, 3, 7)
5	Ch8 = (6, 8, 4)	f = 0.83	P = 5/33 = 0.15	Ch8 = (6, 8, 4)
6	Ch7 = (3, 5, 7)	f = 0.75	P = 6/33 = 0.18	Ch2 = (2, 3, 4)
7	Ch6 = (3, 4, 5)	f = 0.66	P = 7/33 = 0.21	Ch7 = (3, 5, 7)
7	Ch2 = (2, 3, 4)	f = 0.66	P = 7/33 = 0.21	Ch6 = (3, 4, 5)

Rank sum = 33

Reproduction: crossover

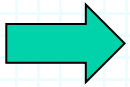
Ch1 = (2, 3, 4)	-	Ch1 = (2, 3, 4)
Ch2 = (2, 2, 3)	<Ch2, Ch5>	Ch2 = (2, 4, 5)
Ch3 = (6, 8, 4)	<Ch3, Ch8>	Ch3 = (6, 8, 4)
Ch4 = (2, 3, 4)	<Ch4, Ch6>	Ch4 = (2, 3, 5)
Ch5 = (3, 4, 5)	SEL	Ch5 = (3, 2, 3)
Ch6 = (3, 4, 5)	SEL	Ch6 = (3, 4, 4)
Ch7 = (6, 8, 4)	-	Ch7 = (6, 8, 4)
Ch8 = (6, 8, 4)	SEL	Ch8 = (6, 8, 4)



One-point crossover, with P = 0.8

Reproduction: mutation

Ch1 = (2, 3, 4)	Ch1 = (2, 3, 4)
Ch2 = (2, 4, 5)	Ch2 = (2, 5, 5)
Ch3 = (6, 8, 4)	Ch3 = (6, 8, 4)
Ch4 = (2, 3, 5)	Ch4 = (2, 3, 5)
Ch5 = (3, 2, 3)	Ch5 = (2, 2, 3)
Ch6 = (3, 4, 4)	Ch6 = (3, 4, 4)
Ch7 = (6, 8, 4)	Ch7 = (6, 8, 4)
Ch8 = (6, 8, 4)	Ch8 = (6, 8, 4)

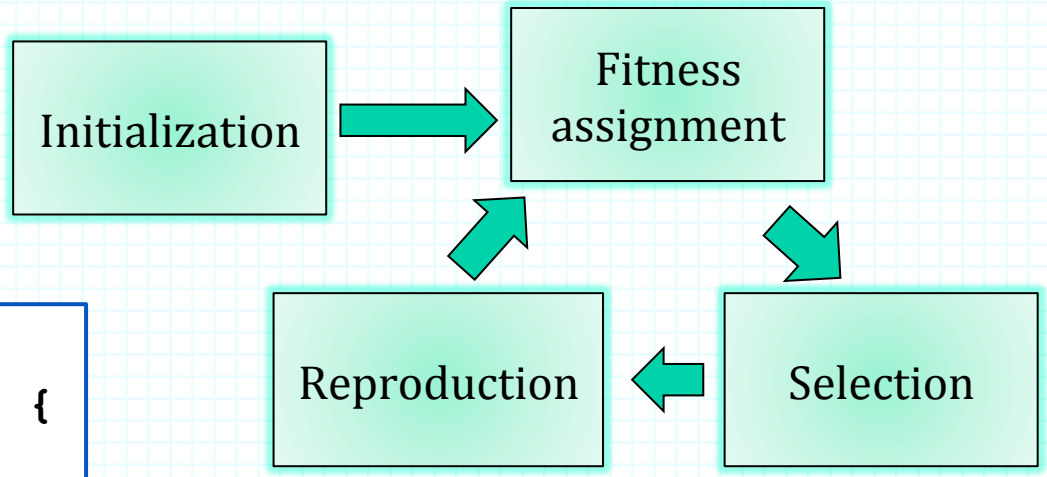


Mutation probability: P = 0.2

Mutation operators: increment/decrement (P = 50%)

Iterating

Until search budget (number of fitness evaluations) is over or target is covered.




```
class Triangle {
  void computeTriangleType() {
1     if (a == b)
2        if (b == c)
3           type = EQUILATERAL;
4        else type = ISOSCELE;
5     else if (a == c)
6              type = ISOSCELE;
7     else if (b == c)
8              type = ISOSCELE;
9     else checkRightAngle();
10    return;
  }
}
```

Then, select the next target (yet to cover statement) and repeat the test case generation cycle.

Collateral coverage

```
class Triangle {
  void computeTriangleType() {
1    if (a == b)
2       if (b == c)
3          type = EQUILATERAL;
4       else type = ISOSCELE;
5    else if (a == c)
6          type = ISOSCELE;
7    else if (b == c)
8          type = ISOSCELE;
9    else checkRightAngle();
10   return;
  }
}
```

When a target is covered accidentally (e.g., (2, 2, 3)) in a test case generation cycle, it is removed from the list of yet to cover targets and the related test case is stored, to be included in the final test suite.

Collateral (serendipitous) coverage: if test cases involved in collateral coverage are not detected and stored, it can be shown that random testing performs asymptotically better than search based testing.

Andrea Arcuri, Muhammad Zohaib Z. Iqbal, Lionel C. Briand: *Formal analysis of the effectiveness and predictability of random testing*. ISSTA, pp. 219-230, 2010.

Final test suite

```
class Triangle {
  void computeTriangleType() {
1    if (a == b)
2       if (b == c)
3          type = EQUILATERAL;
4       else type = ISOSCELE;
5    else if (a == c)
6          type = ISOSCELE;
7    else if (b == c)
8          type = ISOSCELE;
9    else checkRightAngle();
10   return;
  }
}
```

Ch1 = (2, 2, 3)
Ch2 = (2, 5, 5)
Ch3 = (2, 2, 2)
Ch4 = (4, 3, 4)
Ch5 = (3, 4, 5)

The final test suite consists of all chromosomes that have been found to cover (even accidentally) one or more yet to cover statements.

The final test suite might contain redundancies that can be eliminated in a post-processing (by keeping only the test cases that cover at least one target uniquely).

FONDAZIONE BRUNO KESSLER

SE SOFTWARE ENGINEERING

Object oriented testing

Paolo Tonella, *Evolutionary testing of classes*. Proc. of the International Symposium on Software Testing and Analysis (ISSTA), pp. 119-128, Boston, USA, July 2004.
Gordon Fraser, Andrea Arcuri, *Whole Test Suite Generation.* IEEE Transactions on Software Engineering, vol. 39, n. 2, pp. 276-291, 2013.

Unit testing of classes

1. An object of the class under test is created using one of the available constructors.
2. A sequence of zero or more methods is invoked on it.
3. The method currently under test is executed.
4. The final state of the object is examined to produce the pass/fail result.

Drivers/stubs are created whenever necessary.
Steps 1, 2 are repeated for each parameter of object type.

Example of test case

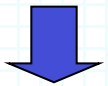
```
1    A a = new A();
2    B b = new B();
3    b.f(2);
4    a.m(5, b);
```

Let us assume that **m** is the method under test.

- Object **b** is created because it is required to call **m**.
- Its state is changed by calling **f** on it.
- Sequence of required input values: <2, 5>.

Features of object-oriented unit testing

- The number and order of method invocations is variable.
- The number of input values is also variable.
- Some parameters in method calls are objects themselves, thus requiring further object constructions.
- The state of the object under test and of the object parameters affects the result.



Chromosomes are not just sequences of input values

Chromosomes

Procedural code: (v1, v2, ..., vN)

Object-Oriented code: $x0=A(): $x0.f(): $x0.g() @ v1, v2, ..., vN

$a=A(int):$b=B():$b.f(int):$a.m(int, $b) @ -1, 2, 5

chromosome variables

input values

```
class TestA extends TestCase {
    public void testCase1() {
        A a = new A(-1);
        B b = new B();
        b.f(2);
        a.m(5, b);
    }
}
```

Random chromosome construction

1. A constructor for the object under test is randomly selected: `$a=A(int)@-1`
2. The invocation of the method under test is appended: `$a=A(int):$a.m(int,$b)@-1,5`
3. All required object constructions are inserted: `$a=A(int):$b=B():$a.m(int,$b)@-1,5`
4. Method invocations to change the state of the created objects are randomly inserted: `$a=A(int):$b=B():$b.f(int):$a.m(int,$b)@-1,2,5`

Steps 3 and 4 are repeated until all chromosome variables used as method or constructor parameters are properly initialized (*well-formedness* of the resulting chromosome).

Random input data generation

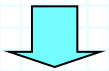
Default, parameterized, pooled and customized input generators:

`A.m(int)`	Default integer generator: uniform selection in [0, 100]
`A.m(int[-2;2])`	Parameterized integer generator: uniform selection in [-2, 2]
`A.m(int<pool>)`	Randomly selected from the program's pool of integer constants
`A.m(int[MyIntGenerator])`	Customized integer generator: method newIntValue() from class MyIntGenerator is called to obtain the value
`A.m(boolean)`	Default boolean generator: **true** and **false** are equally likely
`A.m(String)`	Default string generator: characters are uniformly chosen from [a-zA-Z0-9], with the string length decaying exponentially
`A.m(String<pool>)`	Randomly selected from the program's pool of string constants
`A.m(String[DateGenerator])`	Customized string generator: only strings representing legal dates are produced (e.g., "3/3/2003")

Mutation operators

Change input value:

```
$a=A(int):$b=B():$b.f(int):$a.m(int, $b) @ -1, 2, 5
```

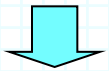


```
$a=A(int):$b=B():$b.f(int):$a.m(int, $b) @ -1, 4, 5
```

Mutation operators

Change constructor:

$a=A(int):$b=B():$b.f(int):$a.m(int, $b) @ -1, 2, 5



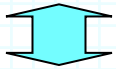
$a=A():$b=B():$b.f(int):$a.m(int, $b) @ 2, 5

The alternative is to use **null**

Mutation operators

Insert/remove method call:

```
$a=A(int):$b=B():$b.f(int):$a.m(int, $b) @ -1, 2, 5
```



```
$a=A(int):$b=B():$a.m(int, $b) @ -1, 5
```

Crossover

$a=A(int):$b=B():$b.g():$a.m(int, $b) @ 0, -3

$a=A():$b=B():$b.f(int):$a.m(int, $b) @ -1, 2

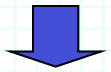
Well-formedness must be maintained

$a=A(int):$b=B():$b.g():$b.f(int):$a.m(int, $b) @ 0, -1, 2

$a=A():$b=B():$a.m(int, $b) @ -3

$a=A():$b=B(int):$c=C(int):$b.h($c):$b.f():$a.m(int, $b) @ 1, 4, 5

$a=A(int, int):$b=B():$a.m(int, $b) @ 0, 3, 6



The alternative is to use **null**

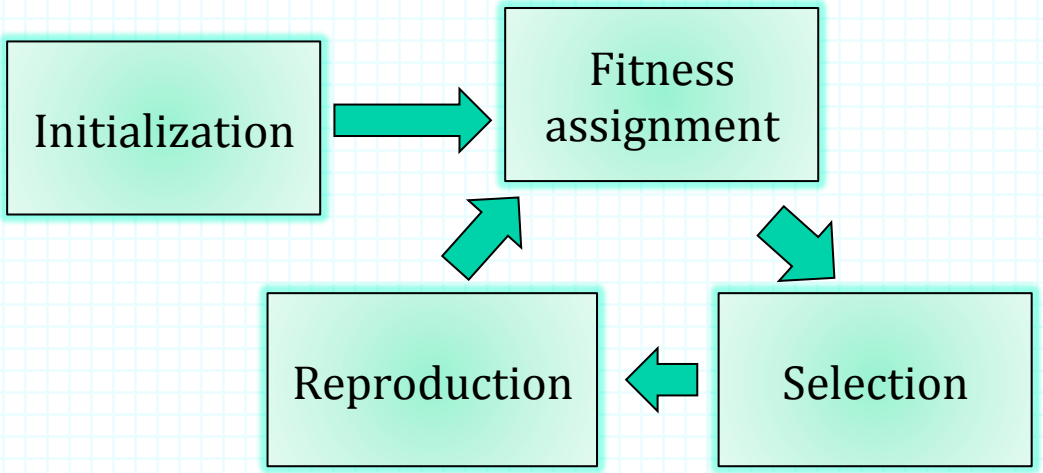
$a=A():$b=B(int):$a.m(int, $b) @ 1, 6

$a=A(int, int):$b=B():$c=C():$b.h($c):$b.f():$a.m(int, $b) @ 0, 3, 5

Fitness

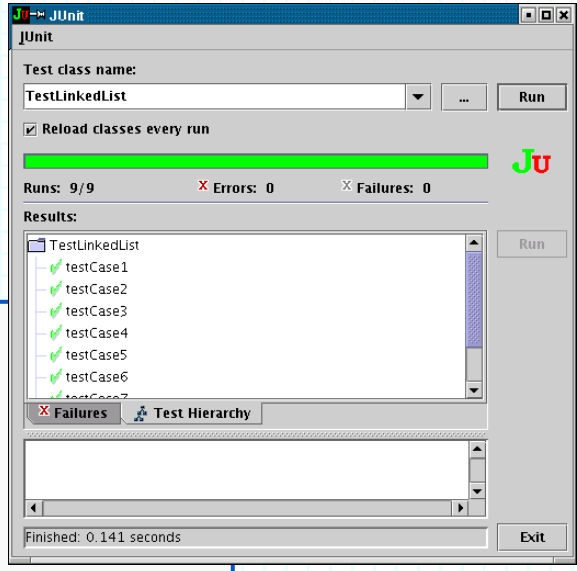
f(x) = approach_level(P(x), t) + branch_distance(P(x), t)

- Truncation selection
- Roulette wheel selection
- Tournament selection
- Ordered selection
- Elitism



Test case generator

$x0=BinaryTree(): $x1=Integer(int): $x2=BinaryTreeNode($x1): $x0.insert($x2): $x3=Integer(int): $x0.search($x3) @ 5, 5




```
class TestBinaryTree extends TestCase {
    ...
    public void testCase2() {
        BinaryTree x0 = new BinaryTree();
        Integer x1 = new Integer(5);
        BinaryTreeNode x2 = new BinaryTreeNode(x1);
        x0.insert(x2);
        Integer x3 = new Integer(5);
        assertTrue(x0.search(x3));
    }
}
```

Assertion added/validated manually

FONDAZIONE BRUNO KESSLER

SE SOFTWARE ENGINEERING

Assertions

Assertions can be inserted as follows:

- The **value returned** from each method call is compared with the expected one.
- If a method call can raise an **exception**, an assertion on its actual occurrence is added.
- At the end of each test case, the **final state** of the object under test is compared with the expected one.

Assertions

Assertions are added at method calls to check the actually **returned values** vs. the expected ones.

```
class TestLinkedList extends TestCase {
    public void testCase1() {
        LinkedList x0 = new LinkedList();
        …
        Integer x1 = null;
        x0.addFirst(x1);
        Integer x2 = null;
        assertTrue(x0.remove(x2));
        …
    }
}
```

Assertions

Assertions are added at method calls possibly raising exceptions.

```
class TestLinkedList extends TestCase {
    public void testCase2() {
        LinkedList x0 = new LinkedList();
        …
        try {
            x0.listIterator(78);
            fail();
        } catch(IndexOutOfBoundException e) {}
}
```

Assertions

Assertions are added at to check the **final object state**.

```
class TestLinkedList extends TestCase {
    public void testCase3() {
        LinkedList x0 = new LinkedList();
        …
        Integer x1 = new Integer(2);
        x0.addLast(x1);
        assertTrue(x0.toString().equals("[2]"));
    }
}
```

FONDAZIONE BRUNO KESSLER

SE SOFTWARE ENGINEERING

Whole test suite generation

Since a limited search budget is available for test case generation:

- **Infeasible targets** may use the entire search budget without achieving any target.
- **Difficult targets** may use most of the search budget, leaving lots of easier coverage targets uncovered.
- The **order** in which targets are considered affects the final result.

Whole test suite generation:

- A set of test cases is generated to cover all targets at the same time.
- The result is independent of the order of the targets and the presence of infeasible or difficult targets.

Example

```
class Stack {
    int[] values = new int[INIT_SIZE];
    int size = 0;

    void push(int x) {
      if (size >= values.length)
          resize();
      if (size < values.length)
          values[size++] = x;
    }
    int pop() {…}
    private void resize() {…}
}
```

Difficult to cover

Else branch is infeasible

Whole test suite generation

Gordon Fraser, Andrea Arcuri, *Whole Test Suite Generation.* IEEE Transactions on Software Engineering, vol. 39, n. 2, pp. 276-291, 2013.

Chromosome

An individual is an entire test suite, i.e. a set of test cases:

Ch1 = {TC1, TC2, …} =

{<$a=A(int):$b=B():$b.f(int):$a.m(int, $b) @ -1, 2, 5>,
<$a=A(int):$b=B():$a.m(int, $b) @ 3, 7>, …}

Ch1: cov = 30
Ch2: cov = 15
Ch3: cov = 17
Ch4: cov = 34
Ch5: cov = 21
Ch6: cov = 11
Ch7: cov = 13
Ch8: cov = 45

Individuals (test suites) with higher coverage are better, so fitness is based on (the complement of) coverage.

Fitness

Sum of the branch distances for the branches that lead to uncovered test targets (to be minimized):

$$f(ch) = |M| - |exec(M)| + \sum_{c \in Uncov} MinBD(c)$$

where *c* is any predicate condition leading to an uncovered target; *minBD* gives the minimum normalized branch distance over all executions of the predicate containing *c* (if the predicate is executed less than two times, BD=1).
The number of non executed methods accounts for the entry branch.

Ties are resolved by rewarding the chromosomes representing smaller test suites (i.e., test suites with less statements).

Mutation

In addition to the mutation operators for the individual test cases:

- Remove method/constructor call;
- Insert method/constructor call;
- Change input value.

Mutation operators that work on whole test suites include:

- **Add test case**: Ch = {TC1, …} → Ch' = {TC_new, TC1, …}
- **Remove test case**: Ch = {TC_k, TC1, …} → Ch' = {TC1, …} if TC_k becomes empty after (repeatedly) applying remove method/ constructor call to it.

Crossover

Each test suite is partitioned into two non empty subsets, which are exchanged between the two individuals.

Ch1 = {TC1, TC2, TC3, TC4} Ch2 = {TC5, TC6, TC7}

Ch1’ = {TC1, TC2, TC7} Ch2’ = {TC5, TC6, TC3, TC4}

FONDAZIONE BRUNO KESSLER

SE SOFTWARE ENGINEERING

Tool

EvoSuite: http://www.evosuite.org

by Gordon Fraser & Andrea Arcuri.

- Implements the genetic algorithm for whole test suite generation
- Instruments the Java bytecode under test for coverage analysis.
- Uses Java reflection to execute test cases encoded as chromosomes and to determine the execution traces.
- Only mutated test cases are re-executed from one iteration to the next one.
- Test case execution is timed out to prevent non-termination.
- Produces a JUnit test class as output.

Dynamic symbolic execution

Static symbolic execution

Path coverage:

1. Select a path *p* to be covered.
2. Determine the path condition (path constraint), by propagation of symbolic input values along the path (expressions are evaluated symbolically). The path condition is a boolean expression containing only input variables.
3. Solve the path condition using an SMT solver.

FONDAZIONE BRUNO KESSLER

SE SOFTWARE ENGINEERING

Static symbolic execution

```
class Triangle {
  int a, b, c; // sides
  int type = NOT_A_TRIANGLE;

  Triangle(int a, int b, int c) {…}
  void checkRightAngle() {…}
  void computeTriangleType() {
    if (a == b)
        if (b == c) type = EQUILATERAL; // target
        else type = ISOSCELE;
    else if (a == c) type = ISOSCELE;
    else if (b == c) type = ISOSCELE;
    else checkRightAngle();
  }
  boolean isTriangle() {…}
  public static void main(String args[]) {…}
}
```

Static symbolic execution

```
class Triangle {
  int a, b, c; // sides
  int type = NOT_A_TRIANGLE;

  Triangle(int a, int b, int c) {…}
  void checkRightAngle(){…}
  void computeTriangleType() {…}
  boolean isTriangle() {…}
  public static void main(String args[]) {
    int a = Integer.parseInt(args[0]);
    int b = Integer.parseInt(args[1]);
    int c = Integer.parseInt(args[2]);
    Triangle t = new Triangle(a, b, c);
    if (t.isTriangle())
       t.computeTriangleType();
    System.out.println(typeToString(t.type));
  }
}
```

Static symbolic execution

```
class Triangle {
  int a, b, c; // sides
  int type = NOT_A_TRIANGLE;

  Triangle(int a, int b, int c) {…}
  void checkRightAngle() {…}
  void computeTriangleType() {…}
  boolean isTriangle() {
    if (a <= 0 || b <= 0 || c <= 0)
        return false;
    if (a + b <= c || a + c <= b || b + c <= a)
        return false;
    return true;
  }
  public static void main(String args[]) {…}
}
```

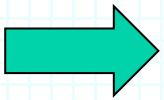
Path condition = ! (a <= 0 || b <= 0 || c <= 0) && !(a + b <= c || a + c <= b || b + c <= a) && (a == b) && (b == c)

Yices SMT solver

Path condition = ! (a <= 0 || b <= 0 || c <= 0) && !(a + b <= c || a + c <= b || b + c <= a) && (a == b) && (b == c)

= (a > 0) && (b > 0) && (c > 0) && (a + b > c) && (a + c > b) && (b + c > a) && (a == b) && (b == c)

```
(define a::int)
(define b::int)
(define c::int)
(assert (> a 0))
(assert (> b 0))
(assert (> c 0))
(assert (> (+ a b) c))
(assert (> (+ a c) b))
(assert (> (+ b c) a))
(assert (= a b))
(assert (= b c))
(check)
```



```
sat
(= a 1)
(= b 1)
(= c 1)
```

Problems of static symbolic execution

In practice, **static** symbolic execution has several limitations that make it inapplicable for test case generation for not trivial programs:

- Selected paths may be infeasible and there are exponentially many paths
- Loops must be unrolled a fixed number of times.
- Static symbolic execution may require a lot of (SMT-solving) time and does not scale to large programs.
- SMT solvers usually do not handle non linear constraints.
- SMT solvers cannot deal with black box functions.

Dynamic symbolic execution:
Start from an available (random), concrete execution and use symbolic execution to explore alternative paths.

FONDAZIONE BRUNO KESSLER

SE SOFTWARE ENGINEERING

Dynamic symbolic execution

1. Execute a random test case.
2. Collect symbolic constraints along the concretely executed path.
3. Negate one branch condition in the path constraint.
4. Use constraint solvers to generate a new test input for the negated constraint.
5. Execute the new test case, covering a new path; iterate from 2.

Dynamic symbolic execution

Problem: *the constraint solver may not be powerful enough to determine concrete values that satisfy the negated path constraint.*

```
4. Use constraint solvers to generate a new
   test input for the negated constraint.
4' If necessary, simplify the path
   constraint by replacing some of the
   symbolic values with concrete values.
```

In particular, simplification may be necessary whenever non linear constraints are involved or black-box, external functions are called.

Implementation

Instrumentation can be used to perform concrete and symbolic execution at the same time.

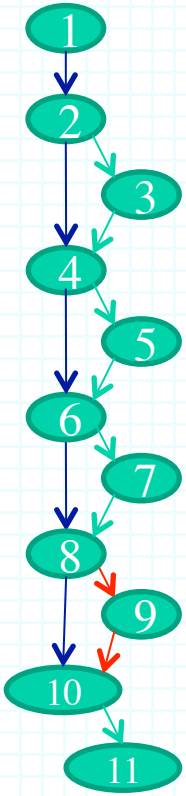
```
x = y + 2;
Symb_exec("x=y+2");
…
if (x > z) {
   Symb_exec("x>z");
```

1. Update symbolic state
2. Accumulate path condition

When a function is called for which the code is not available for symbolic execution, the outcome of its concrete execution is used as an approximation of its symbolic execution.

Example 1

```
void testme(char x[4]) {
1   int n = 0;
2   if (x[0] == 'b')
3     n++;
4   if (x[1] == 'a')
5     n++;
6   if (x[2] == 'd')
7     n++;
8   if (x[3] == '!')
9     n++;
10  if (n >= 4)
11    abort();
}
```

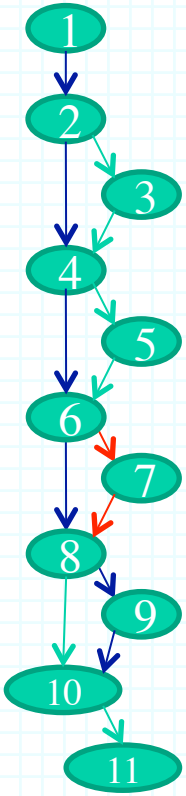


Concrete Execution		Symbolic Execution	
concrete state		symbolic state	constraints
x = "good"		x = {x_0, x_1, x_2, x_3}	(x_0 ≠ 'b') &
n = 0		n = 0	(x_1 ≠ 'a') &
			(x_2 ≠ 'd') &
			(x_3 ≠ '!') &
			(0 < 4)

(x_0 ≠ 'b') & (x_1 ≠ 'a') & (x_2 ≠ 'd') & (x_3 = '!') ➔ <x = "goo!">

Example 1

```
void testme(char x[4]) {
1   int n = 0;
2   if (x[0] == 'b')
3     n++;
4   if (x[1] == 'a')
5     n++;
6   if (x[2] == 'd')
7     n++;
8   if (x[3] == '!')
9     n++;
10  if (n >= 4)
11    abort();
}
```

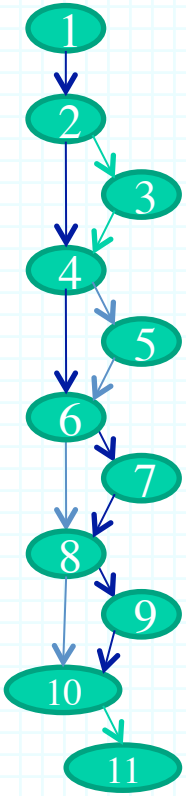


Concrete Execution		Symbolic Execution	
concrete state		symbolic state	constraints
x = "goo!"		x = {x_0, x_1, x_2, x_3}	(x_0 ≠ 'b') &
n = 1		n = 1	(x_1 ≠ 'a') &
			(x_2 ≠ 'd') &
			(x_3 = '!') &
			(1 < 4)

(x_0 ≠ 'b') & (x_1 ≠ 'a') & (x_2 = 'd') & (x_3 = '!') ➔ <x = "god!">

Example 1

```
void testme(char x[4]) {
1   int n = 0;
2   if (x[0] == 'b')
3     n++;
4   if (x[1] == 'a')
5     n++;
6   if (x[2] == 'd')
7     n++;
8   if (x[3] == '!')
9     n++;
10  if (n >= 4)
11    abort();
}
```

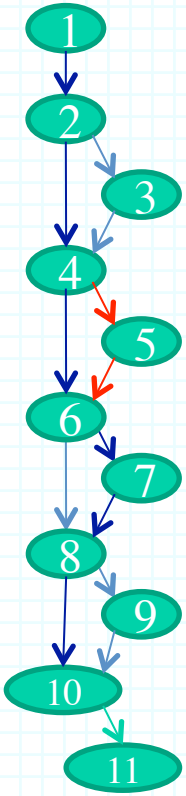


Concrete Execution		Symbolic Execution	
concrete state		symbolic state	constraints
x = "god!"		x = {x_0, x_1, x_2, x_3}	(x_0 ≠ 'b') &
n = 2		n = 2	(x_1 ≠ 'a') &
			(x_2 = 'd') &
			(x_3 = '!') &
			(2 < 4)

(x_0 ≠ 'b') & (x_1 ≠ 'a') & (x_2 = 'd') & (x_3 ≠ '!') ➔ <x = "godd">

Example 1

```
void testme(char x[4]) {
1   int n = 0;
2   if (x[0] == 'b')
3     n++;
4   if (x[1] == 'a')
5     n++;
6   if (x[2] == 'd')
7     n++;
8   if (x[3] == '!')
9     n++;
10  if (n >= 4)
11    abort();
}
```

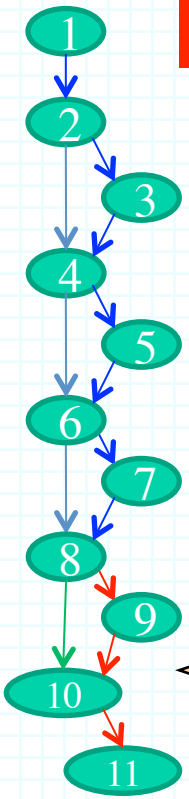


Concrete Execution		Symbolic Execution	
concrete state		symbolic state	constraints
x = "godd"		x = {x_0, x_1, x_2, x_3}	($x_0 \neq$ 'b') &
n = 1		n = 1	($x_1 \neq$ 'a') &
			(x_2 = 'd') &
			($x_3 \neq$ '!') &
			(1 < 4)

($x_0 \neq$ 'b') & (x_1 = 'a') & (x_2 = 'd') & ($x_3 \neq$ '!') ➔ <x = "gadd">

Example 1

```
void testme(char x[4]) {
1   int n = 0;
2   if (x[0] == 'b')
3     n++;
4   if (x[1] == 'a')
5     n++;
6   if (x[2] == 'd')
7     n++;
8   if (x[3] == '!')
9     n++;
10  if (n >= 4)
11    abort();
}
```



After 16 steps

Concrete Execution

Symbolic Execution

concrete state	symbolic state	constraints
x = "badd" n = 3	x = {x_0, x_1, x_2, x_3} n = 3	(x_0 = 'b') & (x_1 = 'a') & (x_2 = 'd') & (x_3 ≠ '!') & (3 < 4)

Target reached

(x_0 = 'b') & (x_1 = 'a') & (x_2 = 'd') & (x_3 = '!') ➔ <x = "bad!">

Example 2

```
void again_test_me(int x,int y){
    z = x*x*x + 3*x*x + 9;
    if(z != y){
        printf("Good branch");
    } else {
        printf("Bad branch");
        target();
    }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x+9
  - cannot handle symbolic value of z
  - make symbolic z = 9 and proceed
- take *then* branch with constraint
  - 9 != y
- solve 9 = y to take *else* branch
- execute next run with x = -3 and y = 9
  - Target reached

Example 2

```
void again_test_me(int x,int y){
    z = x*x*x + 3*x*x + 9;
    if(z != y){
        printf("Good branch");
    }
}
```

Replace symbolic expression with concrete value when symbolic expression becomes unmanageable (e. g. non-linear)

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x+9
  - cannot handle symbolic value of z
  - make symbolic z = 9 and proceed

take *then* branch with constraint

- 9 != y

solve 9 = y to take *else* branch

execute next run with x = -3 and y = 9

- Target reached

Example 2

```
void again_test_me(int x,int y){
    z = x*x*x + 3*x*x + 9;
    if(z != y){
        printf("Good branch");
    } else {
        printf("Bad branch");
        abort();
    }
}
```

```
void again_test_me(int x,int y){
    z = black_box_fun(x);
    if(z != y){
        printf("Good branch");
    } else {
        printf("Bad branch");
        abort();
    }
}
```

Properties

- **Incompleteness**: dynamic symbolic execution is complete only if the constraint solver can successfully solve all (i.e., exponentially many) constraints (without having to simplify them).
- **Approximation**: an input satisfying a simplified path condition, where some symbolic values are replaced by concrete ones, is not ensured to execute the path of interest.
- **Soundness**: dynamic symbolic execution is always sound, since generated test cases are executable, hence revealed faults are real faults.

Example of approximation

```
class Triangle {
  int a, b, c; // sides
  int type = NOT_A_TRIANGLE;

  Triangle(int a, int b, int c) {…}
  void checkRightAngle() {
    if (a*a + b*b == c*c)
        type = RIGHT_ANGLE;
    else if (b*b + c*c == a*a)
        type = RIGHT_ANGLE;
    else if (a*a + c*c == b*b)
        type = RIGHT_ANGLE;
    else type = SCALENE;
  }
  void computeTriangleType() {…}
  boolean isTriangle() {…}
  public static void main(String args[]) {…}
}
```

Concrete input = (2, 3, 4) [SCALENE]

Path condition = ! (a <= 0 || b <= 0 || c <= 0) && !(a + b <= c || a + c <= b || b + c <= a) && !(a*a + b*b == c*c) && !(b*b + c*c == a*a) && !(a*a + c*c == b*b)

Example of approximation

Concrete input = (2, 3, 4) [SCALENE]

SMT solver:

(a > 0 && b > 0 && c > 0) && (a + b > c && a + c > b && b + c > a) &&
(a*a + b*b != c*c) && (b*b + c*c != a*a) &&
(a*a + c*c == b*b)

Error: feature not supported: non linear problem.

Simplified constraint:

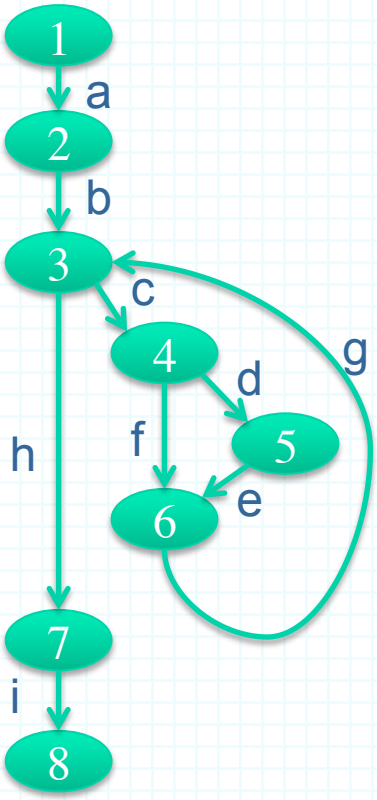
(a > 0 && b > 0 && c > 0) && (a + b > c && a + c > b && b + c > a) &&
(2*a + 3*b != 4*c) && (3*b + 4*c != 2*a) &&
(2*a + 4*c == 3*b)

```
sat
(= a 4)
(= b 4)
(= c 1)
```

Not a right angle triangle!

Target not covered.

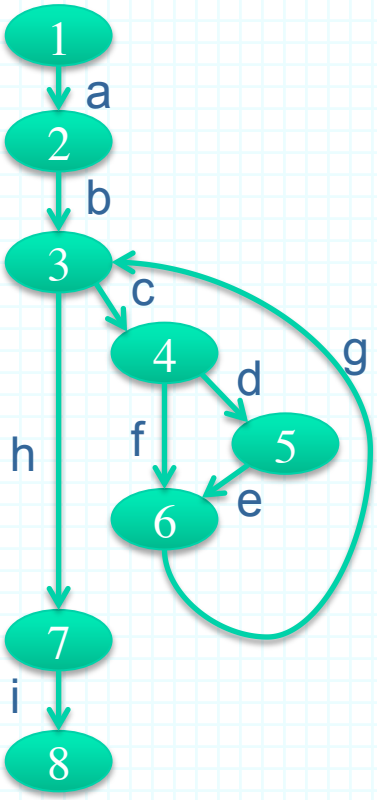
SBST vs DSE: Different goals



SBST = {a, b, c, d, e, f, g, h, i}

DSE= {<a, b, h, i>,
<a, b, c, f, g, h, i>,
<a, b, c, d, e, g, h, i>,
<a, b, c, d, e, g, c, f, g, h, i>,
<a, b, c, d, e, g, c, d, e, g, h, i>,
…}

SBST vs DSE: Different goals



SBST = {a, b, c, d, e, f, g, h, i}

DSE= {<a, b, h, i>,
<a, b, c, f, g, h, i>,
<a, b, c, f, g, c, f, g, h, i>,
<a, b, c, f, g, c, f, g, c, f, g, h, i>,
…}

Different weaknesses and strengths

	Weaknesses	Strengths
DSE	• Loops • Black box functions • Non linear constraints • Complex data structures • Divergences • Reflection	• Exploration strongly guided by path condition • Few executions (fitness evaluations) required
SBST	• Flat (non-guiding) fitness functions • Deceptive fitness functions • Many fitness evaluations (executions) required	Robust w.r.t complex/unknown program semantics (e.g., black box functions, non linear expressions, complex data structures, reflection)

Combining SBST & DSE

- [**I&X'08**] Kobi Inkumsah, Tao Xie. *Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution*. Proc. of Automated Software Engineering (ASE), pp. 297-306, 2008.
- [**XTH&S'09**] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, Wolfram Schulte. *Fitness-guided path exploration in dynamic symbolic execution*. Proc. of the International Conference on Dependable Systems and Networks (DSN), pp. 359-368, 2009.
- [**BHHLMT&V'11**] Arthur I. Baars, Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, Paolo Tonella, Tanja E. J. Vos. *Symbolic search-based testing*. Proc. of Automated Software Engineering (ASE), pp. 53-62, 2011.
- [**M&F'11**] Jan Malburg, Gordon Fraser. *Combining search-based and constraint-based testing*. Proc. of Automated Software Engineering (ASE), pp. 436-439, 2011.
- [**GF&A'13**] Juan Pablo Galeotti, Gordon Fraser, Andrea Arcuri. *Improving Search-based Test Suite Generation with Dynamic Symbolic Execution*. Proc. of the 24th Int. Symposium on Software Reliability Engineering (ISSRE), 2013.

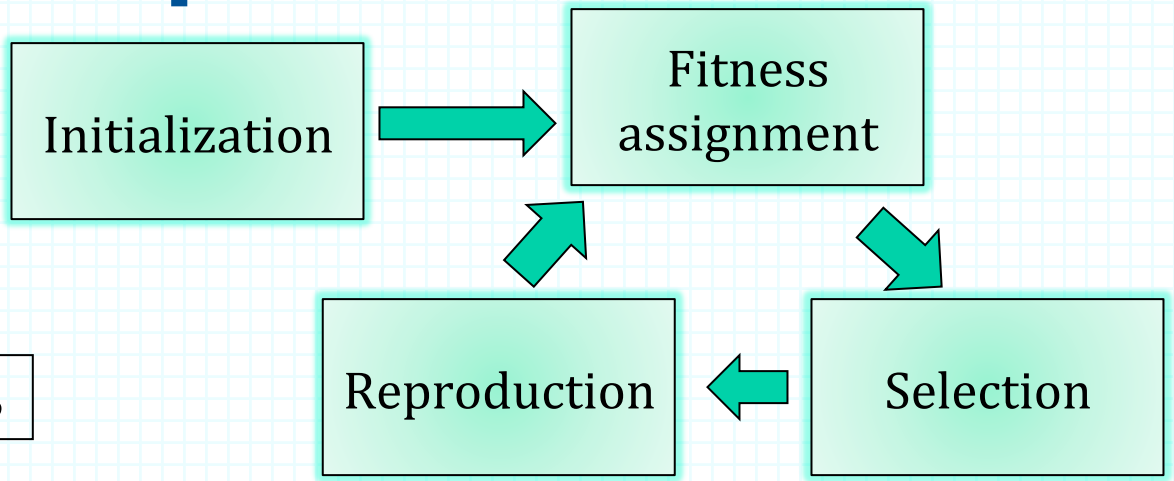
FONDAZIONE BRUNO KESSLER

SE SOFTWARE ENGINEERING

Existing combinations

- DSE as additional genetic operator [**M&F'11,GF&A'13**]
- Alternation between DSE and SBST [**I&X'08**]
- Fitness used to select which path to explore in DSE [**XTH&S'09**]
- Symbolic execution based fitness in SBST [**BHHLMT&V'11**]

Additional genetic operator



Ch1:

1	5	-1	0	3

pc1: C1 && C2 && **C3** && C4

pc1': C1 && C2 && !**C3**

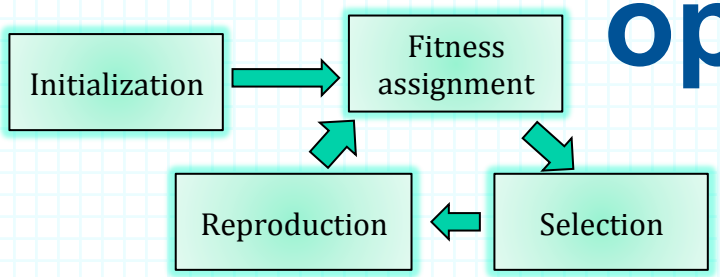
Ch1':

1	0	2	1	1

- Mutation
- Crossover
- **DSE based mutation**

[**M&F'11**] Jan Malburg, Gordon Fraser. *Combining search-based and constraint-based testing*. Proc. of Automated Software Engineering (ASE), pp. 436-439, 2011.

Additional genetic operator



1. On which individuals is DSE applied?
2. When is it applied?
3. How is it applied?

1. Individuals for which primitive mutation affects the fitness.
2. DSE is applied with probability P (suggested value = 100%).
3. Individuals produced by DSE are kept only if they improve the fitness

[**GF&A'13**] Juan Pablo Galeotti, Gordon Fraser, Andrea Arcuri. *Improving Search-based Test Suite Generation with Dynamic Symbolic Execution*. Proc. of the 24th International Symposium on Software Reliability Engineering (ISSRE), 2013.

Additional genetic operator

```
double testMe(int x, int y, double z) {
1   boolean flag = y > 1000;
2   //…
3   if (x + y == 1024)
4      if (flag)
5         if (Math.cos(z) - 0.95 < Math.exp(z))
6            abort();
7   return 0.0;
}
```

Ch1:	1024	0	0.0

Standard mutation may take a while to make y > 1000 and x + y = 1024

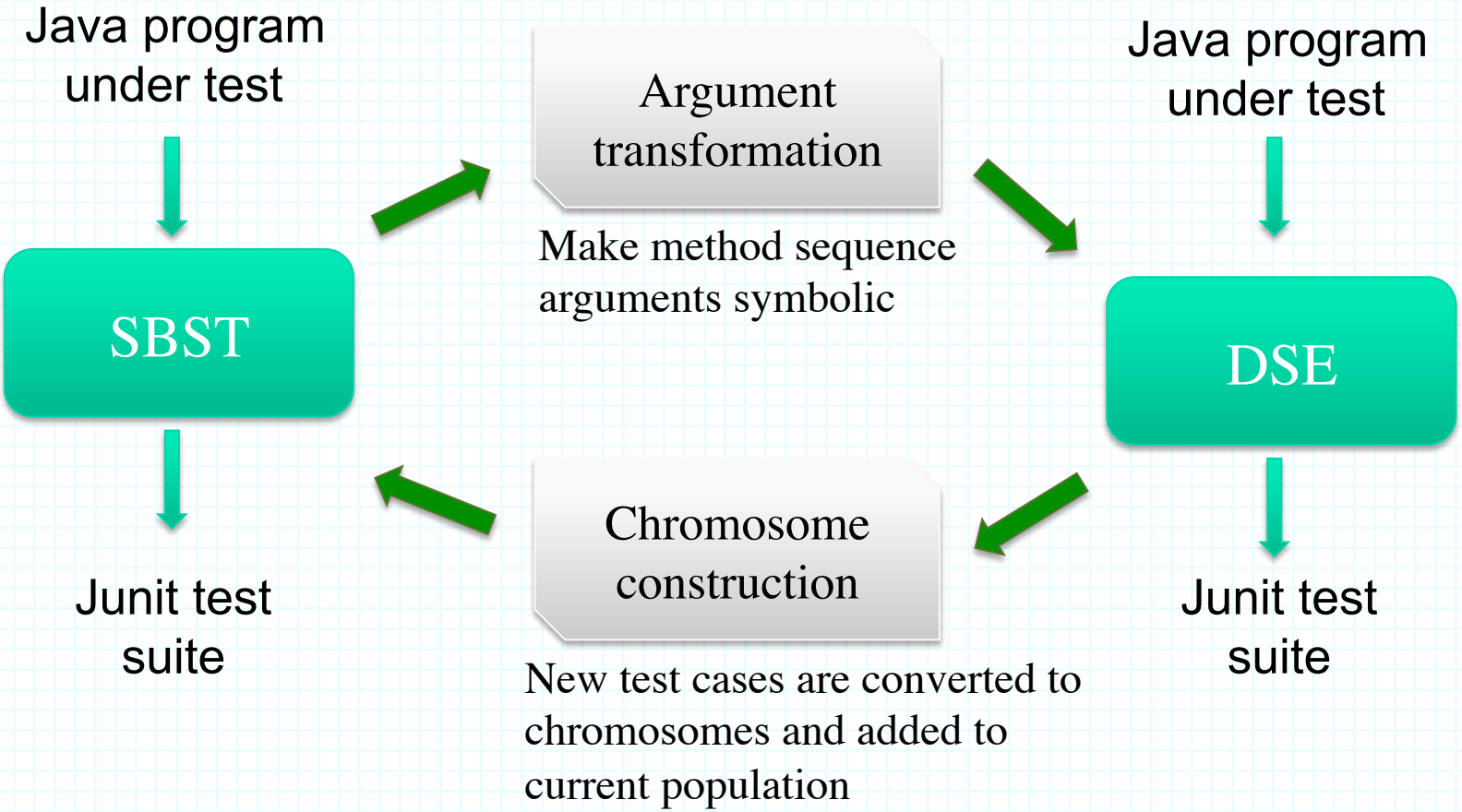
pc1: (x + y = 1024) && (y ≤ 1000)

pc1': (x + y = 1024) && (y > 1000)

Ch1':	23	1001	0.0

Normal mutation and crossover are still required for conditions such as 5.

Alternation



[**I&X'08**] Kobi Inkumsah, Tao Xie. *Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution*. Proc. of Automated Software Engineering (ASE), pp. 297-306, 2008.

Alternation

$x0=BankAccount(): $x0.deposit(int): $x0.withdraw(int) @ 1, 20

Primitive values are made symbolic: z0 [balance], z1 [amount]

Path condition (withdraw) = (z1 > z0)

Error handling branch:

```
void withdraw(int amount) {
  if (amount > balance) {
     printError();
     return;
  }
  balance = balance - amount;
  …
}
```

Alternation

$x0=BankAccount(): $x0.deposit(int): $x0.withdraw(int) @ 1, 20

Primitive values are made symbolic: z0, z1

Path condition (withdraw) = (z1 > z0)

Negated path condition = (z1 <= z0)

```
sat
(= z1 1)
(= z0 20)
```

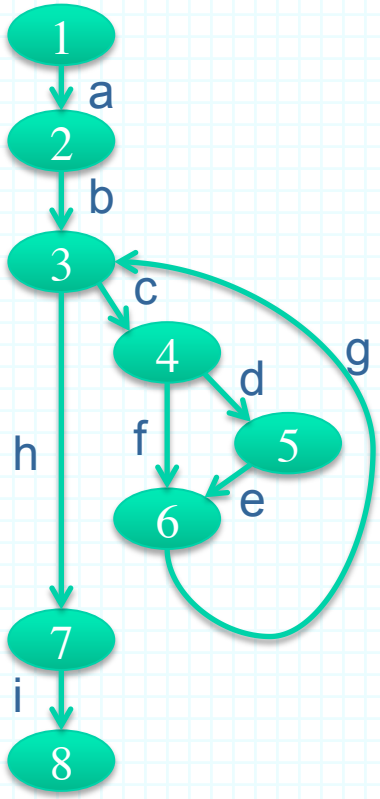
$x0=BankAccount(): $x0.deposit(int): $x0.withdraw(int) @ 20, 1

Added to current population

Non-error branch:

```
if (amount > balance) {
   printError();
   return;
} balance = balance - amount;
…
```

Path selection

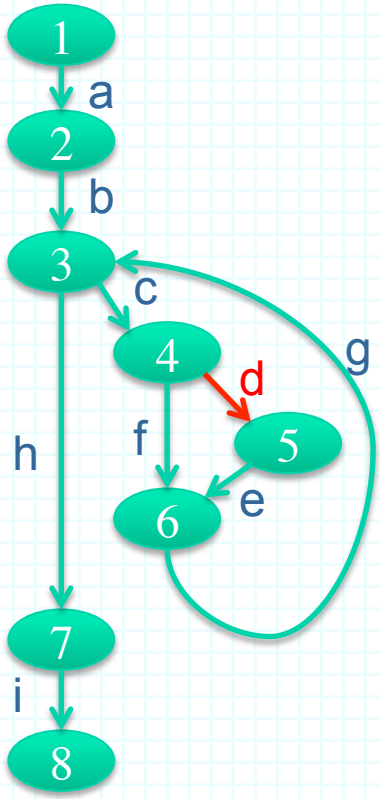


DSE= {<a, b, h, i>, // TC1
<a, b, c, f, g, h, i>, // TC2
<a, b, c, f, g, c, f, g, h, i>} // TC3

- Which test case shall be selected for branch flipping?
- Which branch shall be flipped?

[**XTH&S'09**] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, Wolfram Schulte. *Fitness-guided path exploration in dynamic symbolic execution*. Proc. of the International Conference on Dependable Systems and Networks (DSN), pp. 359-368, 2009.

Path selection

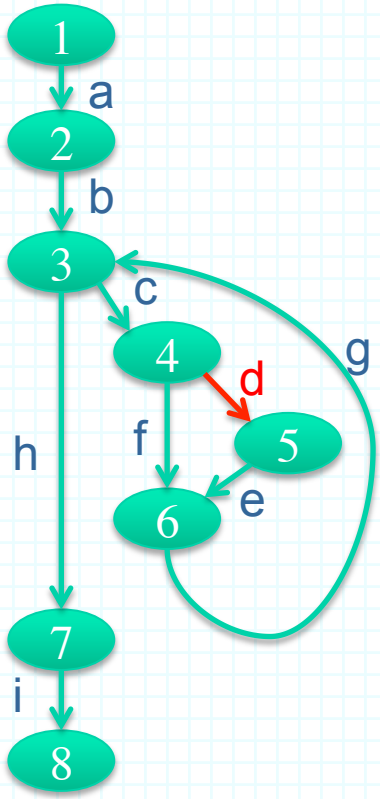


DSE= {<a, b, h, i>, // TC1
<a, b, c, f, g, h, i>, // TC2
<a, b, c, f, g, c, f, g, h, i>} // TC3

- Which test case shall be selected for branch flipping?

BranchDistance(TC2, d) = 0.8
BranchDistance(TC3, d) = min(0.8, 0.5) = 0.5

Path selection



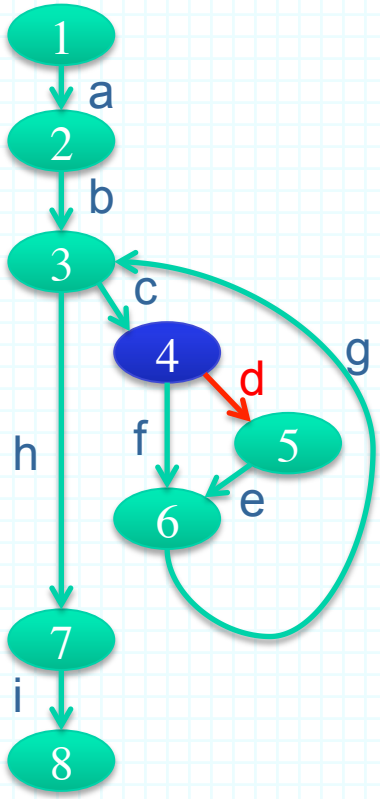
DSE= {<a, b, h, i>, // TC1
<a, b, c, f, g, h, i>, // TC2
<a, b, c, f, g, c, f, g, h, i>} // TC3

$3T_1$ $4F_1$ $3T_2$ $4F_2$ $3F_1$

- Which branch shall be flipped?

Solve(PC3[C$4F_1$ ➔ !C$4F_1$]) = *UNSAT*
Solve(PC3[C$4F_2$ ➔ !C$4F_2$]) = *UNSAT*
FitnessGain(3F) = (1+1+0.3)/3 = 0.76
FitnessGain(3T) = -0.76

New fitness function

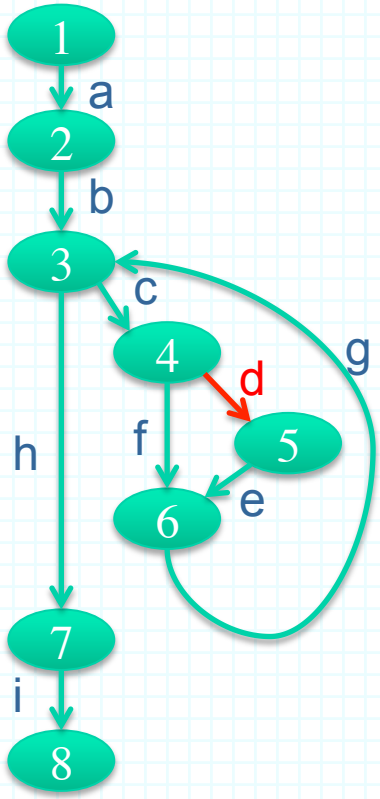


PathExpression(4, d) = (fgc)*d

P1 = d
P2 = fgcd
P3 = fgcfgcd
P4 = fgcfgc **D[fgc$^+$]** d

[**BHHLMT&V'11**] Arthur I. Baars, Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, Paolo Tonella, Tanja E. J. Vos. *Symbolic search-based testing*. Proc. of ASE, pp. 53-62, 2011.

New fitness function

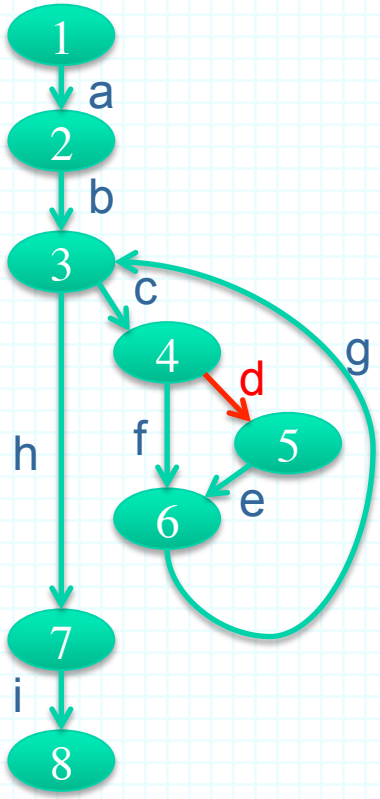


P1 = d
PC1 = C4
FF1 = BD(dist(C4))

P2 = fgcd
PC2 = C3 ∧ C4
FF2 = BD(dist (C3) + dist (C4))

P3 = fgcfgcd
PC3 = C3 ∧ C3' ∧ C4
FF3 = BD(dist (C3) + dist (C3') + dist (C4))

New fitness function



P4 = fgcfgc **D[cfg⁺]** d

PC4 = C3 ∧ C3' ∧ D[C3] ∧ C4

FF4 = BD(dist(C3) + dist(C3') + 1 + dist(C4))

Fitness(TC, 4, d) = *min*(FF1[TC], FF2[TC], FF3[TC], FF4[TC])

StdFitness(TC, 4, d) = BD(dist(C4))

Comparison of existing combinations

Paper	Switching vs. Enhancing	Overcome limitation	Algorithmic change
[M&F'11, GF&A'13]	S	Stagnation/slow convergence	New genetic operator
[I&X'08]	S	Long time required for parameter value generation	Meta-level algorithm for switching
[XTH&S'09]	E (DSE)	Path exploration preventing (branch) coverage	Test case and branch selection
[BHHLMT&V'11]	E (SBST)	Standard fitness, accounting only for shortest path to target	New fitness function

- DSE as additional genetic operator [**M&F'11, GF&A'13**]
- Alternation between DSE and SBST [**I&X'08**]
- Fitness used to select which path to explore in DSE [**XTH&S'09**]
- Symbolic execution based fitness in SBST [**BHHLMT&V'11**]

Conclusions and future work

FONDAZIONE BRUNO KESSLER

SE SOFTWARE ENGINEERING

Conclusions

- High coverage can be achieved automatically using search based test case generation.
- Functions and classes under test are exercised in a quite sophisticated way.
- Resulting test suites are generally quite compact and their size can be controlled as a further optimization objective.

FONDAZIONE BRUNO KESSLER

SE SOFTWARE ENGINEERING

Future work

- Investigation of alternative adequacy criteria and non-functional goals.
- Automating/supporting the generation of oracles (i.e., anomaly detectors), based on invariants and temporal properties of the functions and classes under test.
- Further empirical studies, especially on combined approaches.

References

- Shaukat Ali, Lionel C. Briand, Hadi Hemmati, Rajwinder Kaur Panesar-Walawege: *A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation*. IEEE Transactions on Software Engineering, vol. 36 n. 6, pp. 742-762, 2010.
- Andrea Arcuri, Muhammad Zohaib Z. Iqbal, Lionel C. Briand: *Formal analysis of the effectiveness and predictability of random testing*. ISSTA, pp. 219-230, 2010.
- Gordon Fraser, Andrea Arcuri: *It is Not the Length That Matters, It is How You Control It*. Proc. of the International Conference on Software Testing (ICST), pp. 150-159, 2011.
- Gordon Fraser, Andrea Arcuri, *Whole Test Suite Generation.* IEEE Transactions on Software Engineering, vol. 39, n. 2, pp. 276-291, 2013.
- Arthur I. Baars, Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, Paolo Tonella, Tanja E. J. Vos. *Symbolic search-based testing*. Proc. of Automated Software Engineering (ASE), pp. 53-62, 2011.
- Juan Pablo Galeotti, Gordon Fraser, Andrea Arcuri. *Improving Search-based Test Suite Generation with Dynamic Symbolic Execution*. Proc. of the 24th International Symposium on Software Reliability Engineering (ISSRE), 2013.
- Patrice Godefroid, Nils Klarlund, and Koushik Sen, DART: Directed Automated Random Testing, PLDI'05.
- Mark Harman and Phil McMinn. *A Theoretical and Empirical Study of Search Based Testing: Local, Global and Hybrid Search*. IEEE Trans. on Soft. Eng., vol. 36, n. 2, pp. 226-247, 2010.

References

- Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, Andre Baresel and Marc Roper. *Testability Transformation*. IEEE Transactions on Software Engineering, vol. 30, n. 1, pp. 3-16, 2004.
- Kobi Inkumsah, Tao Xie. *Evacon: a framework for integrating evolutionary and concolic testing for object-oriented programs*. Proc. of Automated Software Engineering (ASE), pp. 425-428, 2007.
- Kobi Inkumsah, Tao Xie. *Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution*. Proc. of Automated Software Engineering (ASE), pp. 297-306, 2008.
- Bogdan Korel: *Automated Software Test Data Generation*, IEEE Transactions on Software Engineering, vol. 16, n. 8, pp. 870-879, 1990.
- Jan Malburg, Gordon Fraser. *Combining search-based and constraint-based testing*. Proc. of Automated Software Engineering (ASE), pp. 436-439, 2011.
- Phil McMinn, *Search-based software test data generation: a survey*. Journal of Software Testing, Verification and Reliability, vol. 14, n. 2, pp. 105-156, June 2004.
- Roy P. Pargas, Mary Jean Harrold and Robert R. Peck: *Test-data generation using genetic algorithms*, Software Testing, Verification and Reliability, vol. 9, n. 4, pp. 263-282, 1999.
- Koushik Sen, Darko Marinov, Cul Agha, CUTE: A Concolic Unit Testing Engine for C, ESEC-FSE'05.

FONDAZIONE BRUNO KESSLER

SE SOFTWARE ENGINEERING

References

- Paolo Tonella, *Evolutionary testing of classes*. Proc. of the International Symposium on Software Testing and Analysis (ISSTA), pp. 119-128, Boston, USA, July 2004.
- David H. Wolpert, William G. Macready: *No Free Lunch Theorems for Optimization*, IEEE Transactions on Evolutionary Computation, vol. 1, n. 1, April 1997.
- Tao Xie, Nikolai Tillmann, Jonathan de Halleux, Wolfram Schulte. *Fitness-guided path exploration in dynamic symbolic execution*. Proc. of the International Conference on Dependable Systems and Networks (DSN), pp. 359-368, 2009.