Chapter 6

Symbolic execution

Course “Model checking”
Volker Stolz, Martin Steffen
Autumn 2019

Section

Targets

Chapter 6 “Symbolic execution”
Course “Model checking”
Volker Stolz, Martin Steffen
Autumn 2019

Chapter 6

Learning Targets of Chapter “Symbolic execution”.

The chapter gives an not too deep introduction to *symbolic* execution and *concolic* execution.

Chapter 6

Outline of Chapter “Symbolic execution”.

Section

Introduction

Testing and path coverage
Symbolic execution
Concolic testing

Chapter 6 “Symbolic execution”
Course “Model checking”
Volker Stolz, Martin Steffen
Autumn 2019

Introduction

- symbolic execution: "old" technique [3]
- natural also in the context of testing
- concolic execution: extension
- used also in compiler
  - code generation
  - optimization

Code example

```
f(int x, int y) {
  if (x*x*x > 0) {
    if (x > 0 && y == 10) {
      fail();
    }
  } else {
    if (x > 0 && y == 20) {
      fail();
    }
  }

  complete();
}
```

How to analyse a (simple) program like that? :

- testing
- "verification" (whatever that means)
  - could include code review
- model-checking? Hm?
- symbolic and concolic execution (see later)

Testing

- maybe the most used method for ensuring software (and system) "quality"
- broad field
  - many different testing goals, techniques
  - also used in combination, in different phases of software engineering cycle
- here: focus on

"white-box" testing

- AKA structural testing
- program code available (resp. CFG)

- also focus: *unit* testing

Goals

- detect errors
- check corner cases
- provide high (code) coverage

(Code) coverage

- note: typically a non-concurrent setting (unit testing)
- different coverage criteria
  - nodes
  - edges, conditions
  - combinations thereof
  - path coverage
- defined to answer the question

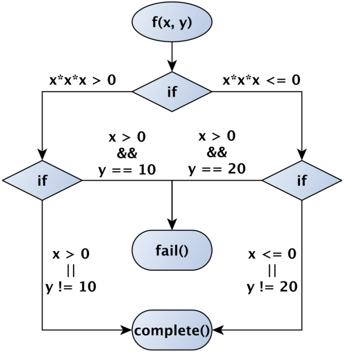
When have I tested "enough"?

path coverage

- ambitious to impossible (loops)
- note: still not *all reachable states*, i.e., not verified yet

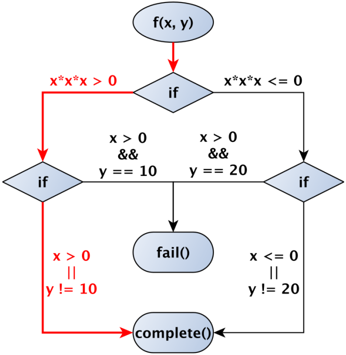
Path coverage

```
f(int x, int y) {
  if (x*x*x > 0) {
    if (x > 0 && y == 10) {
      fail();
    }
  } else {
    if (x > 0 && y == 20) {
      fail();
    }
  }

  complete();
}
```



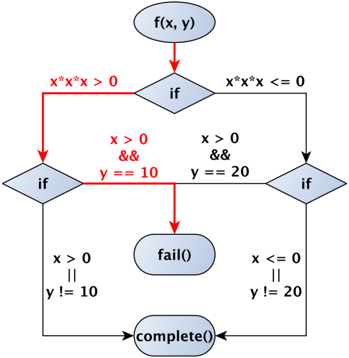
Path coverage

```
f(int x, int y) {
  if (x*x*x > 0) {
    if (x > 0 && y == 10) {
      fail();
    }
  } else {
    if (x > 0 && y == 20) {
      fail();
    }
  }

  complete();
}
```



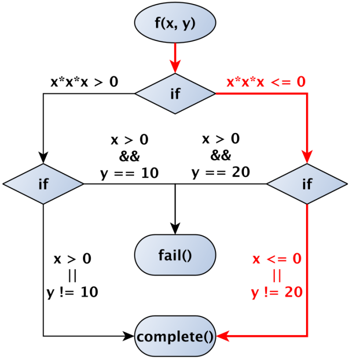
Path coverage

```
f(int x, int y) {
  if (x*x*x > 0) {
    if (x > 0 && y == 10) {
      fail();
    }
  } else {
    if (x > 0 && y == 20) {
      fail();
    }
  }

  complete();
}
```



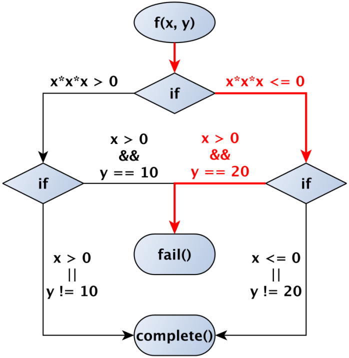
Path coverage

```
f(int x, int y) {
  if (x*x*x > 0) {
    if (x > 0 && y == 10) {
      fail();
    }
  } else {
    if (x > 0 && y == 20) {
      fail();
    }
  }

  complete();
}
```

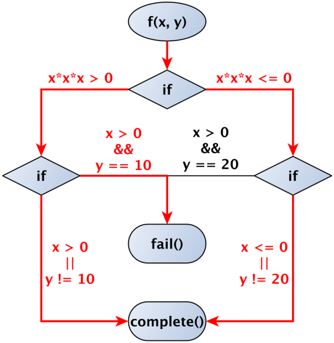


Path coverage

```
f(int x, int y) {
  if (x*x*x > 0) {
    if (x > 0 && y == 10) {
      fail();
    }
  } else {
    if (x > 0 && y == 20) {
      fail();
    }
  }

  complete();
}
```



Path coverage



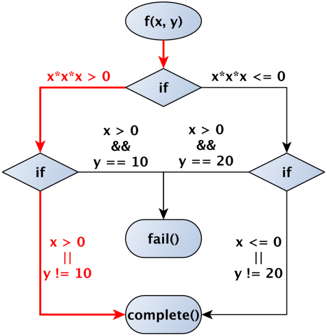
- 3 possible exec. path
- corresponding path conditions
- "optimal": cover all path
- find input set to run program covering all those paths

Random testing

- most naive way of testing
- generating random inputs
- concrete input values
- dynamic executions of programs
- *observe actual* behavior and
- compare it agains *expected behavior*

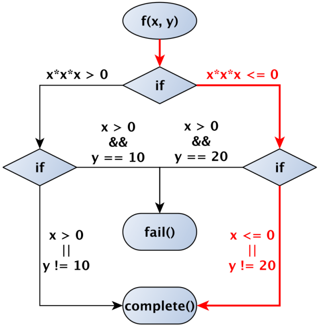
Random testing

- different inputs, different paths
- maybe
  - $(x, y) = (700, 500)$
  - $(x, y) = (-700, 500)$
  - ...

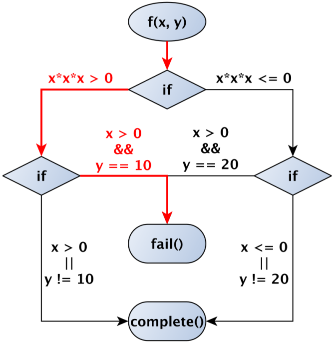


Random testing

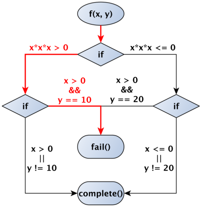
- different inputs, different paths
- maybe
  - $(x, y) = (700, 500)$
  - $(x, y) = (-700, 500)$
  - ...



One path so far missed



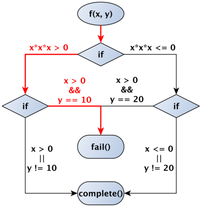
How to get that path (or others)?



- maybe: $(x, y) = (145, 10)$
- by chance: very low probability to randomly get $y = 10$

- path condition

How to get that path (or others)?



- maybe: $(x, y) = (145, 10)$
- by chance: very low probability to randomly get $y = 10$

Symbolic representation

$$x > 0 \land y = 10$$

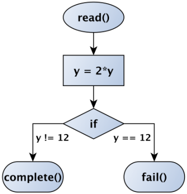
- path condition

Symbolic execution

- symbols instead of concrete value
- use if path conditions, aka path constraints
- cf. connection to SAT and SMT
- constraint solver computes real values

Simple example

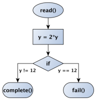
```
1 y = read();
2 y = 2 * y;
3
4 if (y == 12) {
5   fail();
6 }
7
8 complete();
```



- in the code: *assignments* not equations (`y := read()`)
- introduce variable `s` for `read()`
- assignments
  - `y := read()` $\Rightarrow y = s$
  - `y := 2*y` $\Rightarrow y = 2s$
- branching point in line 4
  - right: $2s = 12$
  - left: $2s \neq 12$

Which input leads to the error?

```
y = read();
y = 2 * y;

if (y == 12) {
  fail();
}

complete();
```



Constraint solver

Solve the path constraint $2s = 12$

- child's play: the solution is $s = 6$
- but: requires solver that can do "arithmetic", including multiplication

In summary

Symbolic execution for dummies

- take the code (resp. the CFG of the code)
- collect all paths into path conditions
  - big conjunctions of all conditions along each the path
  - each condition b will have
    - one positive mention b in one continuation of the path
    - one negated mention $\neg b$ in the other continuation
- solve the constraints for paths leading to errors with an approriate SMT solver

- works best for loop-free program
- cf. also SSA
- but there is another problem as well (see next)

How about the program we started with?

```
f(int x, int y) {
  if (x*x*x > 0) {
    if (x > 0 && y == 10) {
      fail();
    }
  } else {
    if (x > 0 && y == 20) {
      fail();
    }
  }

  complete();
}
```

Complex condition x^3

```
f(int x, int y) {
  if (x*x*x > 0) {
    if (x > 0 && y == 10) {
      fail();
    }
  } else {
    if (x > 0 && y == 20) {
      fail();
    }
  }

  complete();
}
```

- non-linear constraint
- in general undecidable
- most constraint solvers throw the towel
- for instance: execution stops, no path covered

What can one do?

what can one do (beyond accepting the SE won't cover all path)?

- "static analysis": abstracting
  - cover both path approximately
- theorem proving? one cannot sell that to testers

Concolic testing

Concrete & Symbolic = "concolic"

Concolic testing

- here following *DART*
- combination of two techniques

Random testing

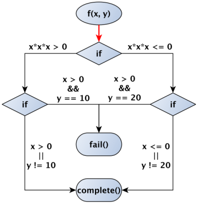
- concrete values
- dynamic execution

Symbolic execution

- symbols, variables
- static analysis

- other name: Dynamic symbolic execution (DSE)

Dart (1)

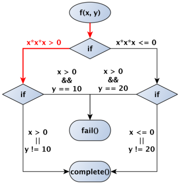


Dynamic execution

- random input: as in random testing
- concrete $(x, y) = 700, 500)$

Symbolic execution

Dart (1)



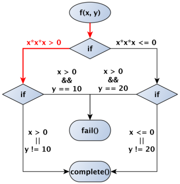
Dynamic execution

- random input: as in random testing
- concrete $(x, y) = 700, 500)$
- `x * x * x > 0`

Symbolic execution

- introduce symbols $x_1 = x, y_1 = y$
- constraint $x^3 \leq 0$

Dart (1)



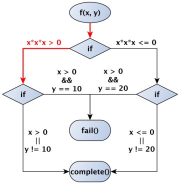
Dynamic execution

- random input: as in random testing
- concrete $(x, y) = 700, 500)$
- `x * x * x > 0`

Symbolic execution

- introduce symbols $x_1 = x, y_1 = y$
- constraint $x^3 \leq 0$
- non-linear: fail

Dart (1)



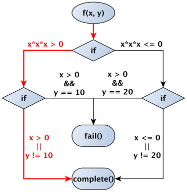
Dynamic execution

- random input: as in random testing
- concrete $(x, y) = 700, 500)$
- `x * x * x > 0`

Symbolic execution

- introduce symbols $x_1 = x, y_1 = y$
- constraint $x^3 \leq 0$
- non-linear: fail
- concrete fall-back: $x_1 = 700$

Dart (1)



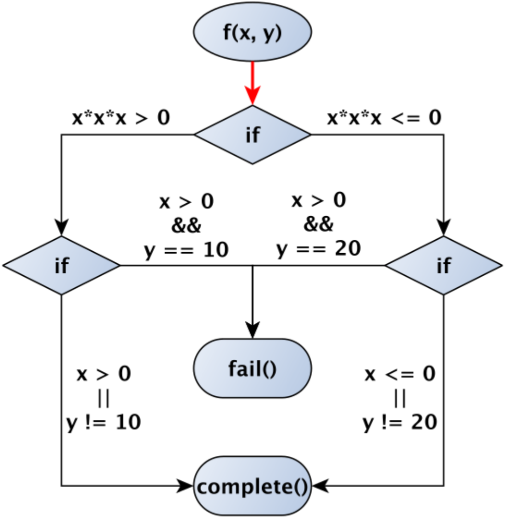
Dynamic execution

- random input: as in random testing
- concrete $(x, y) = 700, 500)$
- `x * x * x > 0`
- `y !=10`

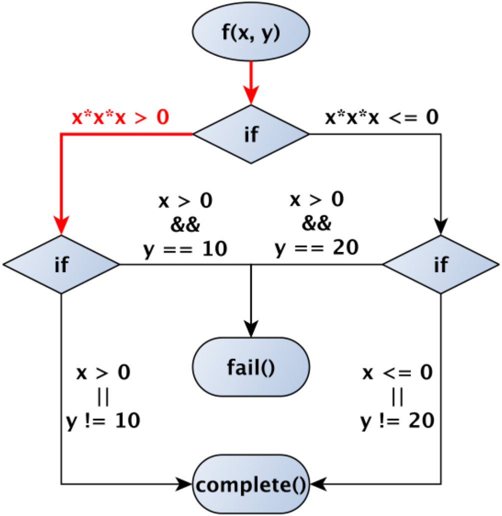
Symbolic execution

- introduce symbols $x_1 = x, y_1 = y$
- constraint $x^3 \leq 0$
- non-linear: fail
- constraint $y_1 = 10$
- solve the constraint:

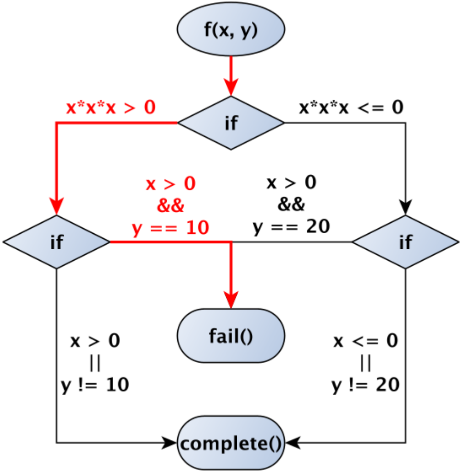
Dart 2



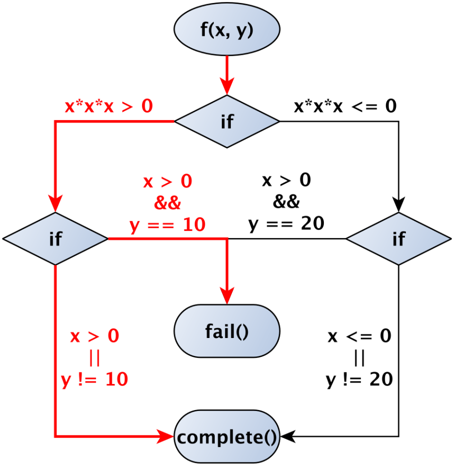
Dart 2



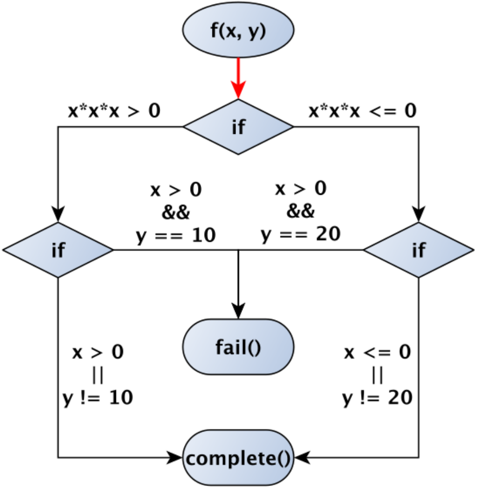
Dart 2



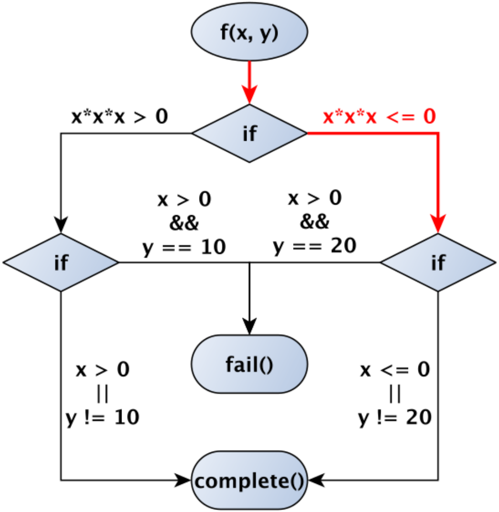
Dart 3



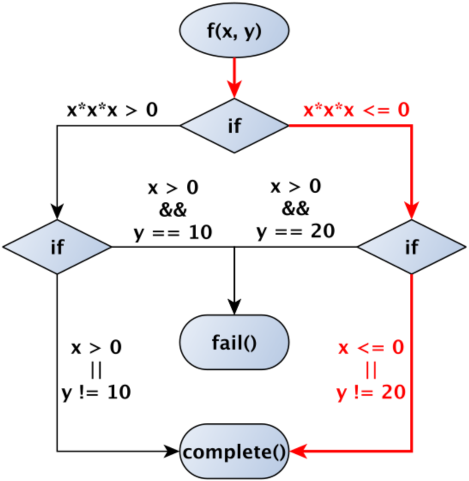
Dart n



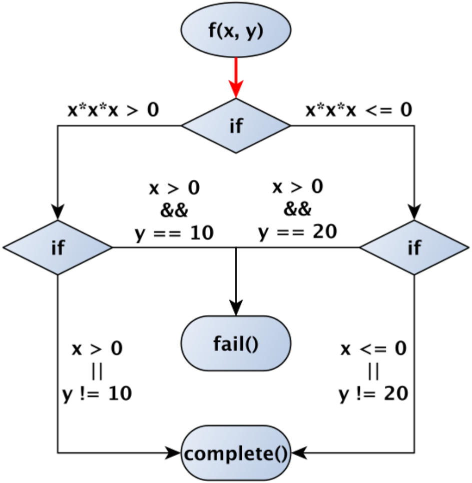
Dart n



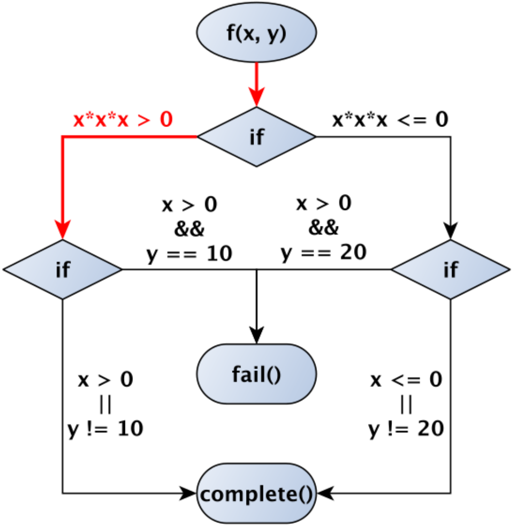
Dart n



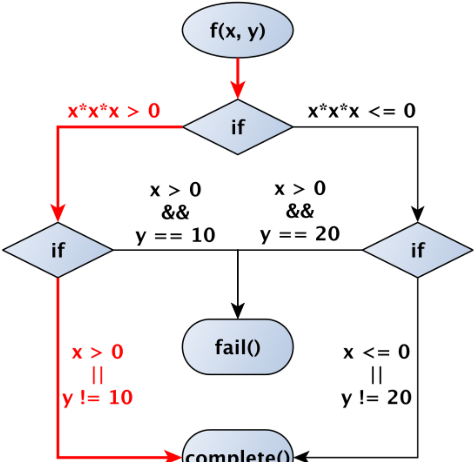
Dart $n + 1$



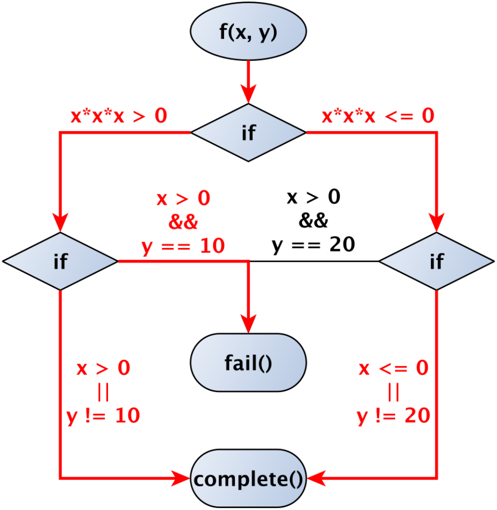
Dart $n + 1$



Dart $n + 1$



Dart completed



References I

Bibliography

[1] Baldoni, R., Coppa, E., D'Ella, D. C., Demetrescu, C., and Finocchi, I. (2018). A survey of symbolic execution techniques. *ACM Computing Survey*, 51(3).

[2] Godefroid, P., Klarlund, N., and Sen, K. (2005). Dart: Directed automated runtime testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM.

[3] King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394.