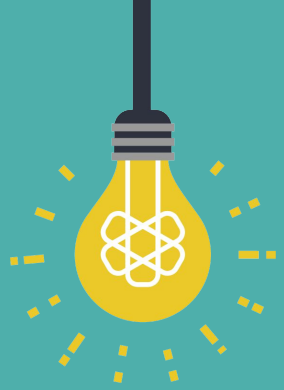


Magic-like Unit Testing

How to increase your code coverage without effort?

symflower

AUTOMATING QUALITY ASSURANCE

Agenda

1. Introduction (5 minutes)
2. Generating Unit Tests (20 minutes)
3. Q&A (5 minutes)

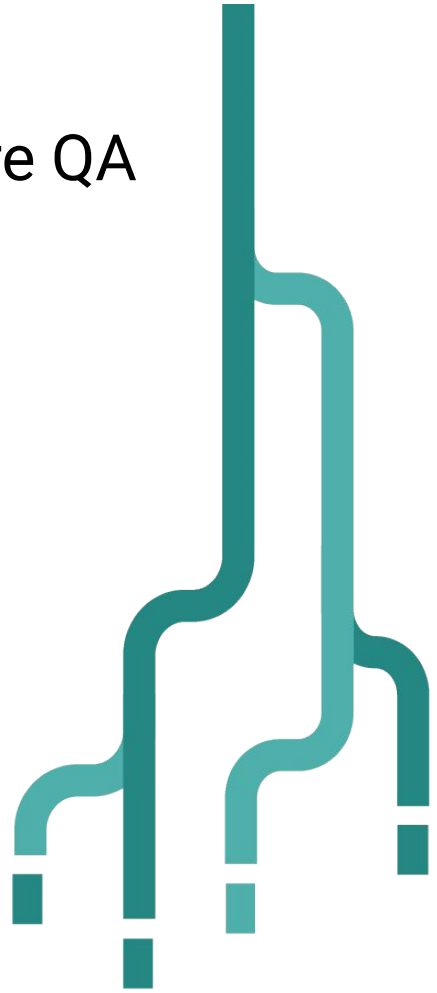
1 Introduction

The story behind Symflower

Symflower the Company

symflower

- Startup based in Linz
- Vision: Complete automation of software QA
- Founders:
  - → Evelyn Haslinger
  - → Markus Zimmermann
- Team: Currently a team of 10
- Top tech investor: eQventure
  - → Lead investor: Franz Fuchsberger (Founder of Tricentis)



2

Generating Unit Tests

How to ease development and testing

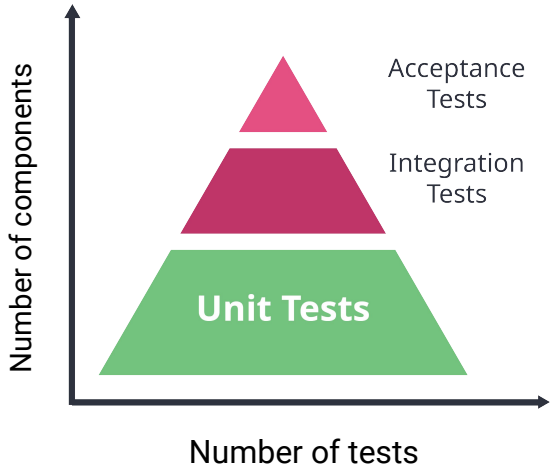
symflower

AUTOMATING QUALITY ASSURANCE

Testing Types

symflower

- Acceptance Testing
  - Does my software do what it is supposed to do?
- Integration Testing
  - Do my components work together?
- Unit Testing
  - Have I built the right foundation?
  - Majority of bugs can be discovered by unit tests
  - The majority of testing should be done on this layer



Fundamental Testing Problems

symflower

- Am I testing the “right things”?
  - Specification-based testing?
    Not enough: e.g. implementation diverges
- What is “thorough testing” anyway?
  - Should we look at the code coverage?
    Which one? e.g. line/MCDC/...? mutation coverage?
    Not enough: e.g. problems are missing
    Not enough: e.g. coverage often just execution
- When do I know that I have tested enough?

Suggestions

symflower

- Mold “the whole” implementation into test cases
  - One test case for every “interesting” path
  - Specification can then be checked with all cases
- “Mutation Testing” for existing tests
  - Check if current test cases are actually useful at all
  - Check if the whole implementation is **really** covered
- Generate test cases automatically
  - Writing and maintaining tests costs time, money, NERVES
  - No overlooked cases
  - Apply “knowledge” and specification as rules
  - Rules can be applied to everything, not by chance

How to Ease Unit Testing

symflower

- Step 1: Templating Tools
  - Many IDEs offer the capability to create test methods that are prefilled
  - Object creation and asserts usually need to be filled out by the developer

  - **Let's take a look**

  - Templating tools already save time
  - But, do not help with finding test cases
  - **Are there options to create test values as well?**

How to Ease Unit Testing

symflower

- Step 2: Generate Test Values

Boundary value analysis (Tessy)

- Limited predefined values, **very low coverage**

Fuzzing (AFL=American Fuzzy Lop)

- Random values, coverage depends on **pure luck**

Symbolic execution (SE) (Symflower)

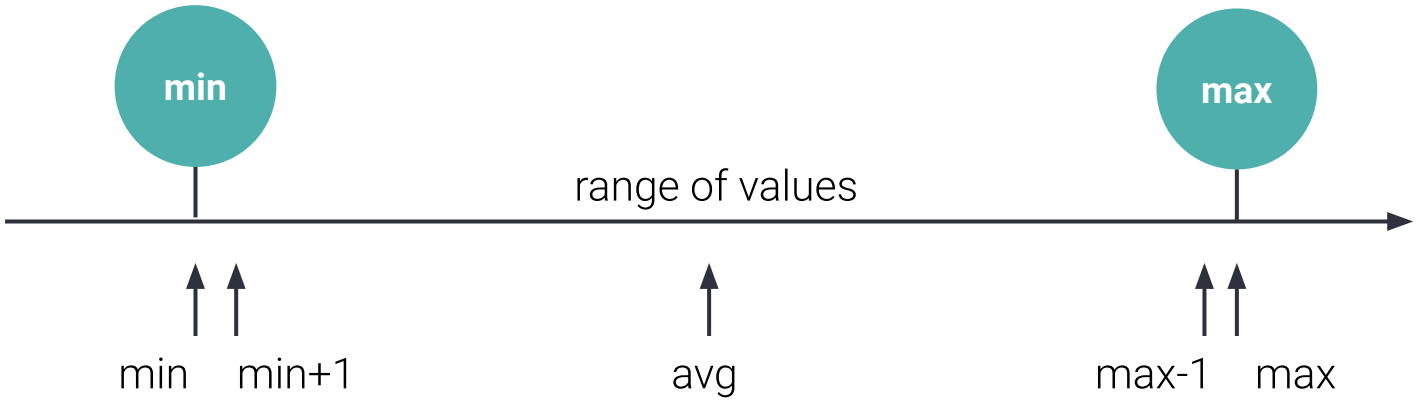
- Targeted values, theoretically **unlimited** depth and complexity

Additional options: **log values**, **search based techniques** and **bounded model checking**.
Or a good semi-automatic option: **model-based testing**

Boundary Value Analysis

symflower

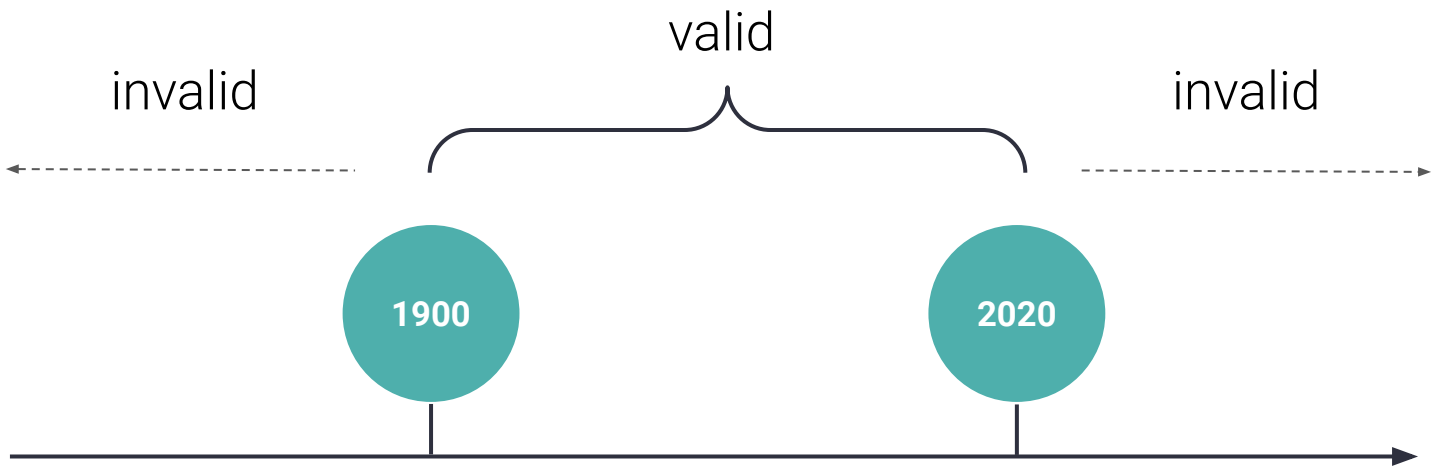
- Boundary Testing
  - Black box testing technique
  - Idea: catch likely corner cases



Boundary Value Analysis

symflower

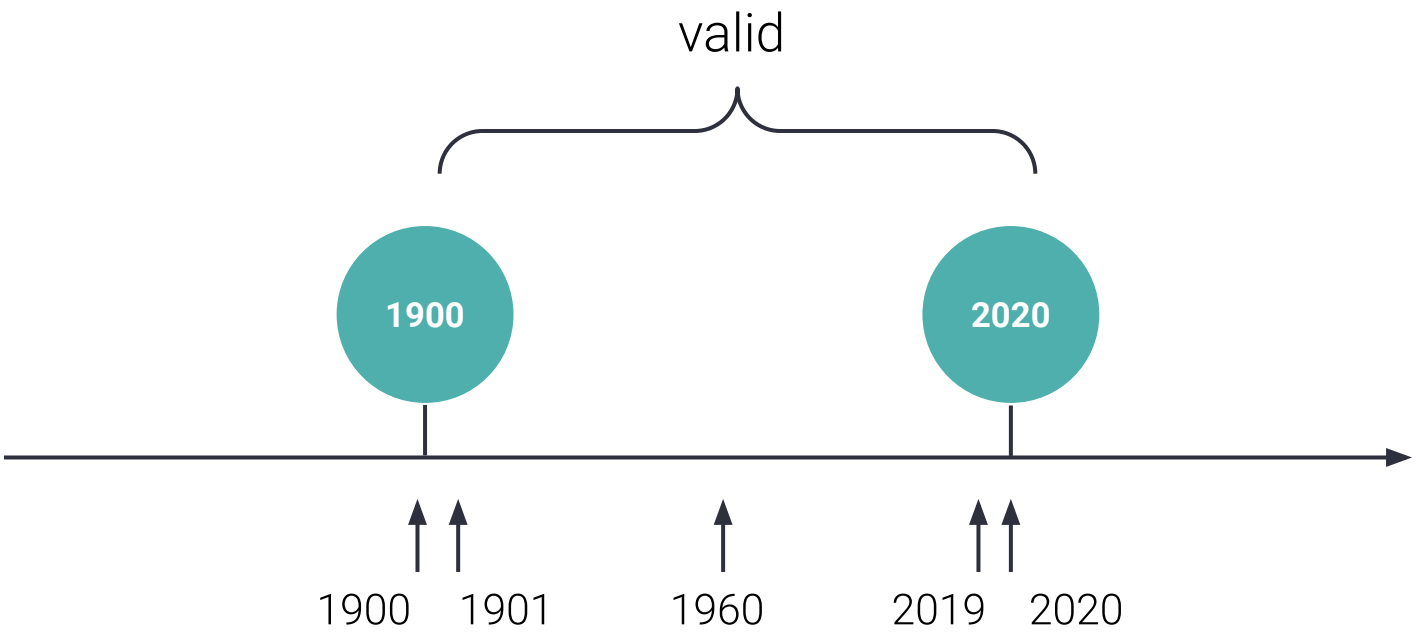
- Equivalence class partitioning
  - Divides input range into partitions
  - **E.g. year of birth**



Boundary Value Analysis

symflower

- Boundary values for partitions
  - Values for “valid” partition



Boundary Value Analysis

symflower

- Boundary values for partitions
  - Values for “invalid” partitions

Boundary Value Analysis

symflower

- Black box testing technique
- No guarantee to reach full test coverage
- Corner Cases are most likely missed
- Does not really work with strings, objects, ...

```
public int Compute(int i){
  if(i < 1900) {
    throw new ...;
  }
  ...
  if(isLeapYear(i) && i%10!=0){
    ...
  }
  ...
}
```

Fuzzing

symflower

- Feed (invalid) unexpected or random data to a program
- Monitor for:
  - Crashes
  - Failed built-in assertions
  - Memory leaks
  - ....
  - New coverage

→ With each input there is a chance to detect a bug

Types of Fuzzing

symflower

- Awareness of input structure
  - Dedicated Format Fuzzer
    - Understanding of input format
    - Generates valid/invalid inputs

- Universal Fuzzer
  - Fuzzer generates purely random inputs
  - Can be used to fuzz any program
  - Might not yield as good results as a dedicated fuzzer

Types of Fuzzing

symflower

- Awareness of program structure
  - Black-box fuzzer are completely unaware of program structure
    - May only scratch the surface and only discover shallow bugs
- Grey-box fuzzer
  - Use instrumentation of the code to increase coverage
- White-box fuzzer
  - Uses programm analysis to reach bugs deep in the program

Symbolic Execution

symflower

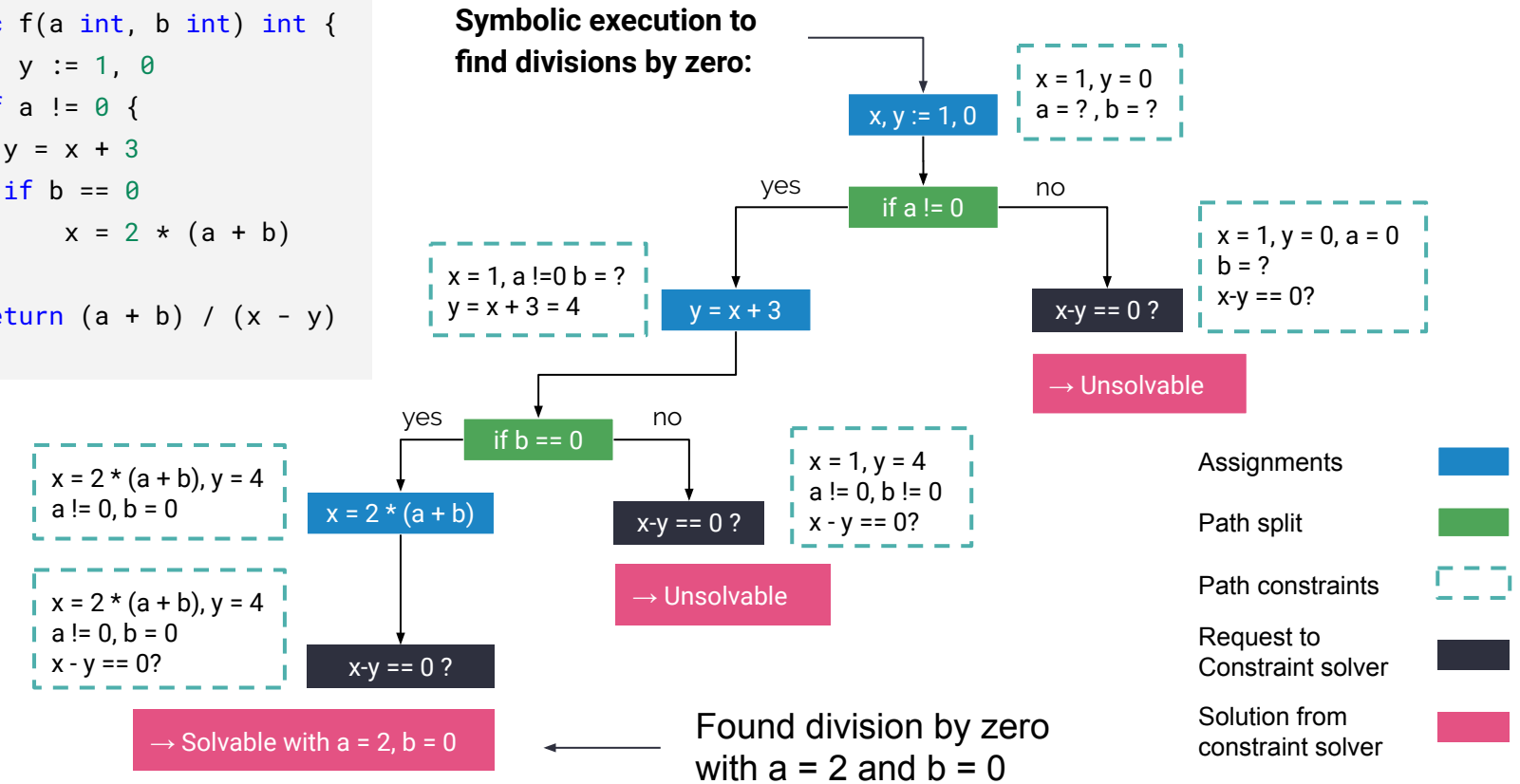
- Program analysis technique
  - Research topic since the mid 70s
- Operate on symbolic rather than concrete values
  - Reason on classes of inputs
- Technique to find the inputs to execute all paths of a function
  - Full MC/DC coverage becomes doable

Symbolic Execution

symflower

Executing a program symbolically means, that rather than operating on concrete values, one is **operating on symbolic values** considering **all possible execution paths at once**. Constraint solvers are used to get concrete values fulfilling the constraints that describe a path.

```
func f(a int, b int) int {
  x, y := 1, 0
  if a != 0 {
    y = x + 3
    if b == 0
        x = 2 * (a + b)
  }
  return (a + b) / (x - y)
}
```



How to Ease Unit Testing

symflower

- Step 2: Generate Test Values
  - Allows to check for obvious problems
    - Program Crashes
    - Memory Leaks
    - Security Issues
  - But are there options to verify for correct behavior?

How to Ease Unit Testing

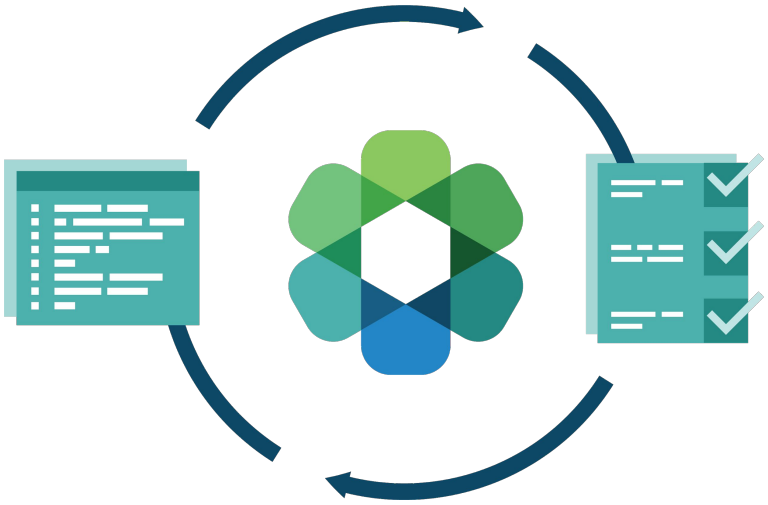
symflower

- Step 3: Check behavior
  - Yes, but we need help from the user
  - Pre-, postconditions and invariants allow us to check whether they hold on all paths

Put It All Together: Symflower

symflower

➔ Let’s take a look



3

Q&A

Let’s discuss

symflower

AUTOMATING QUALITY ASSURANCE

symflower

AUTOMATING QUALITY ASSURANCE

Evelyn Haslinger eh@symflower.com

Markus Zimmermann mz@symflower.com