SYSC 4101 / SYSC5105

Definitions—Part I

What are we looking for ???

- Fault: A fault (which is usually referred to as a bug for historic reasons) is a defect in a system.
  - Humans commit faults
  - E.g., a faulty statement
- Error: Errors occur at runtime when some part of the system enters an unexpected state due to the activation of a fault.
  - E.g., an erroneous state
- Failure: A failure of the system occurs when the delivered service deviates from what the system is intended for.
- The word “bug” is not part of the vocabulary in SYSC4101.

What are we looking for ???

- Fault → Error → Failure

  Three conditions necessary for a failure to be observed

  – Reachability: the location(s) in the program that contain the fault must be reached

  – Infection: the state of the program must be incorrect

  – Propagation: the infected state must propagate to cause some output of the program to be observed as incorrect.

- All three are important concepts to understand and keep in mind
- All three are necessary
  – It is not sufficient to reach the fault
    - There may not be any infection
    - Delivered service may not be affected

  – It is not sufficient to “infect” the state
    - Delivered service may not be affected

Observability vs. Controllability

- Software Observability:

  How easy it is to observe the behavior of a program in terms of its outputs, effects on the environment and other hardware and software components

  - Software that affects hardware devices, databases, or remote files have low observability

  Observability relates to Propagation

- Software Controllability:

  How easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviors

  - Easy to control software with inputs from keyboards
  - Inputs from hardware sensors or distributed software is harder
  - Data abstraction reduces controllability and observability

  Controllability relates to Reachability and Infection

Reachability

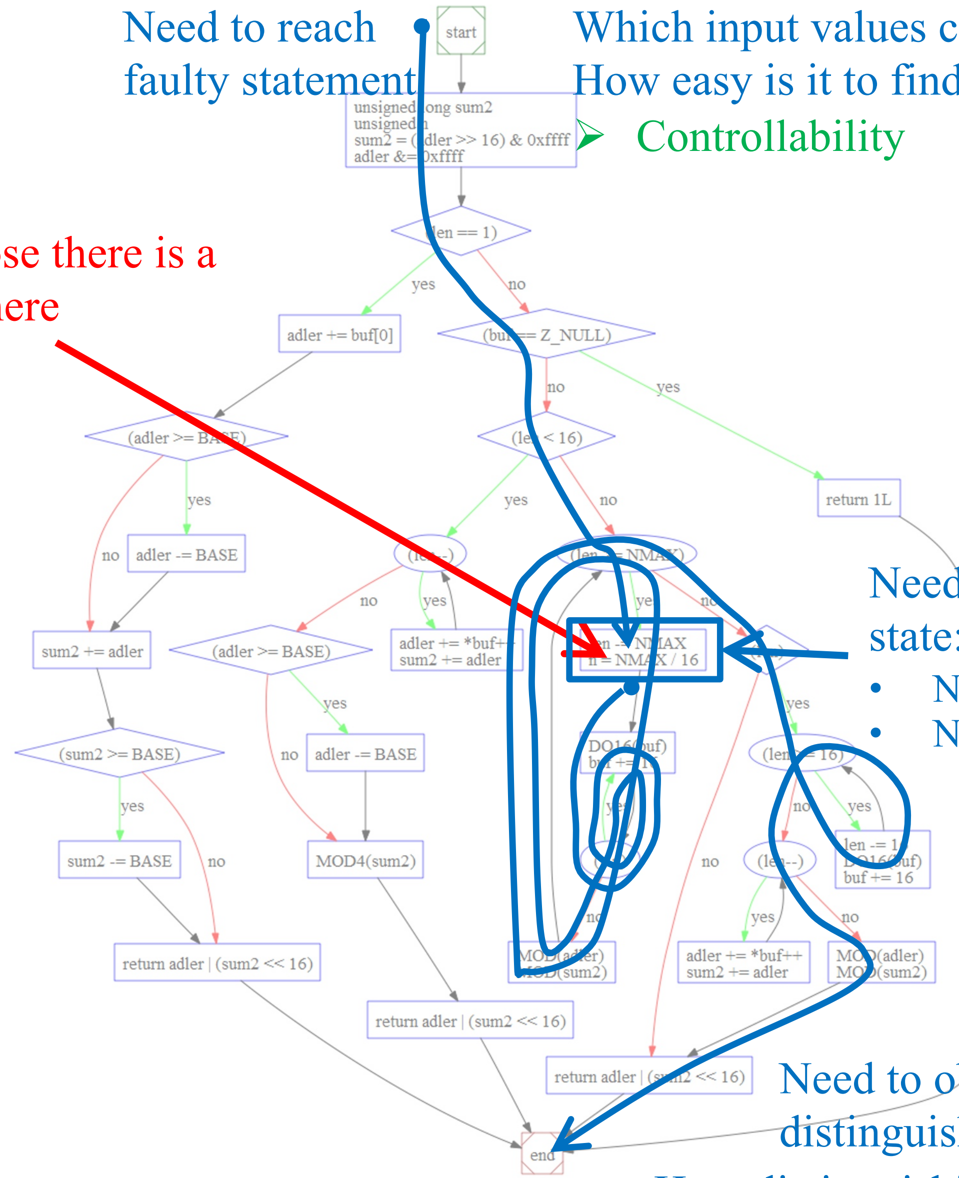
Propagation

Need to reach faulty statement

Which input values can do that?
How easy is it to find them?

➢ Controllability

Suppose there is a fault here



Need to result in erroneous state:

- Not all computations do
- Not all input values allow this

Need to obtain an outcome that we can distinguish from what is expected.

How distinguishing is the outcome?

➢ Observability

Control flow graph picture from https://scitools.com/confusing/

Test scaffolding / Test harness

Infrastructure (software and/or hardware) we put in place to execute test cases

- Test driver
- Test stub (a.k.a., mock)
- Test oracle

Test scaffolding / Test harness

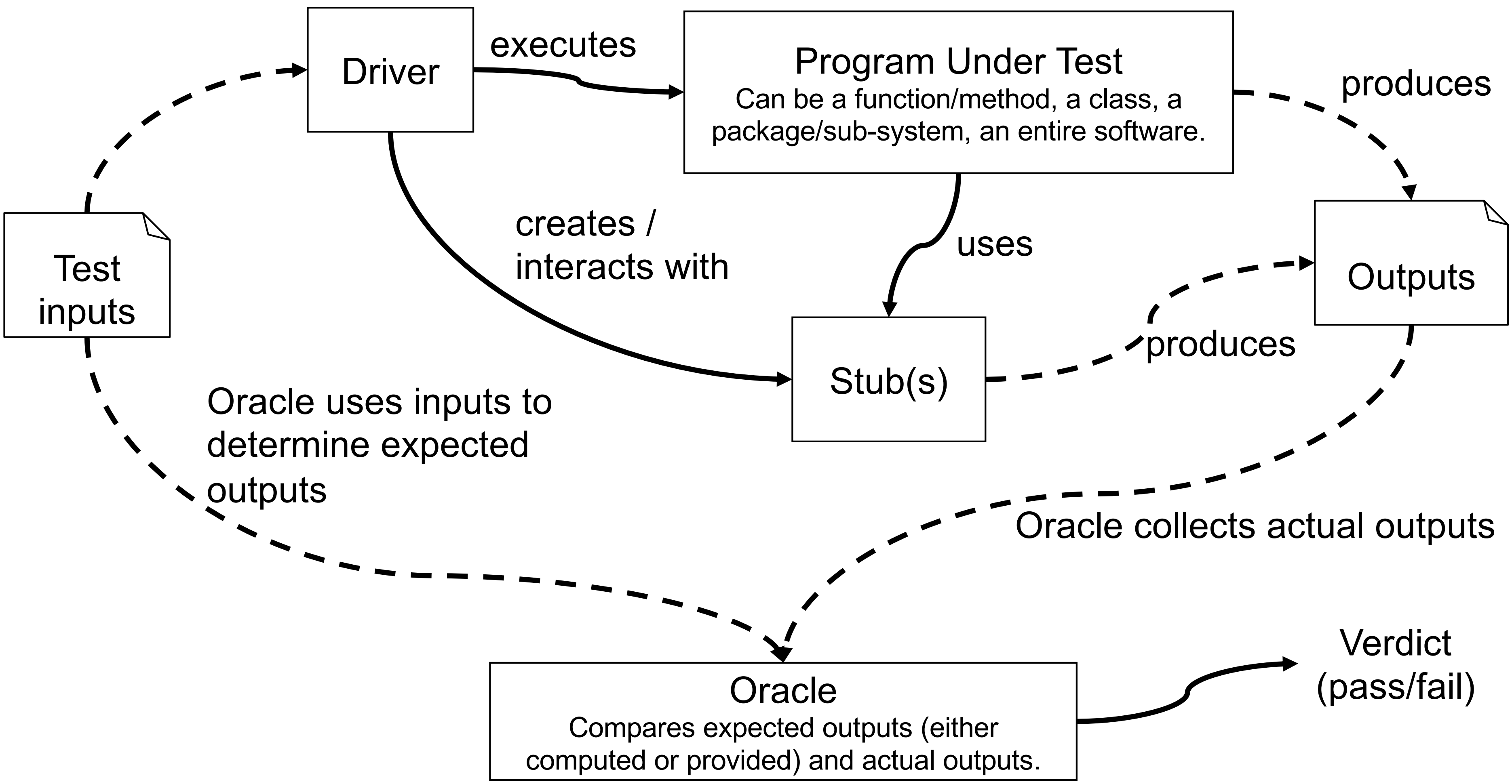
- Test driver:
  - A software component or test tool that replaces a component that takes care of the control and/or the calling of a software component. [Ammann & Offutt]
  - The test driver executes a test case (one execution of system under test with input values) or a test suite/set (a set of test cases)
- Test stub:
  - A skeletal or special-purpose implementation of a software module, used to develop or test a component that calls the stub or otherwise depends on it. It replaces a called component. [Ammann & Offutt]
  - E.g., simulates a piece of code not yet ready.

Stubs and drivers enable components to be isolated from the rest of the system for testing purposes

Test scaffolding / Test harness

- Oracle:
  - Assist in deciding whether a test outcome is successful or not
  - Outcome (verdict) of the oracle evaluation: pass / fail
  - Has **two** important tasks:
    1. Comparing the actual output against the expected output.
       - E.g., using an assert() statement
       - Very often done by software
    2. Deciding what the expected output is, given the test inputs of a test case.
       - E.g., deciding what to check in an assert() statement
       - Very often done by human

Putting things together



A Simple Example

```
#include "mySquareRoot.h"

int root(double a, double b, double c, double *root1, double *root2) {
  double determinant = b*b-4*a*c;
  if (determinant > 0) {
    *root1 = (-b+mySquareRoot(determinant))/(2*a);
    *root2 = (-b-mySquareRoot(determinant))/(2*a);
    return 1;
  } else if (determinant == 0) {
    *root1 = *root2 = -b/(2*a);
    return 1;
  } else {
    return 0;
  }
}
```

Suppose

- I want to test function root()
- Function mySquareRoot() is not yet available

➢ We stub mySquareRoot()

A Simple Example (cont.)

```
// file mySquareRoot.h
double mySquareRoot(double num);
```

Provides what function root() expects:
A function called mySquareRoot().

```
// file mySquareRoot-Stub.c
#include "mySquareRoot.h"
#include "stubFormySquareRoot.h"
static double valueToReturn;
double mySquareRoot(double n) {
 return valueToReturn;
}
void setReturnValue(double r) {
  valueToReturn = r;
}
```

Provides what the driver expects:
A function called setReturnValue().

```
// file stubFormySquareRoot.h
void setReturnValue(double r);
```

This is the stub, a simple simulation of
the behavior of function mySqureRoot().

A Simple Example (cont.)

```
#include "root.h"
#include "stubFormySquareRoot.h"

int main() {
  double a, b, c, root1, root2;
  int result;
  double expectedRoot1, expectedRoot2;
  double epsilon = 0.000001;

  // test case 1
  setReturnValue(5); //instructing the stub what to respond to root()
  a = -2; b = 1; c = 3;
  expectedRoot1 = -1; expectedRoot2 = 1.5;
  result = root(a, b, c, &root1, &root2);
  if ( (result==1) && (fabs(expectedRoot1-root1)<epsilon) &&
(fabs(expectedRoot2-root2)<epsilon) ) printf("test case 1 passes.\n");
  else printf("test case 1 fails.\n");
```

This is the driver.

Setting environment, including stub

Executing test

Oracle

A Simple Example (cont.)

- Compiling and executing the test (with stub)

```
cc -c mySquareRoot-Stub.c
cc -c root.c
cc -c root-UnitTestWithStub.c
cc mySquareRoot-Stub.o root.o root-
UnitTestWithStub.o -o root-UnitTestWithStubs
./root-UnitTestWithStubs
```

- Compiling and running the program with all components

```
cc -c mySquareRoot.c
cc -c root.c
cc -c quadratic.c
cc mySquareRoot.o root.o quadratic.o -o quadratic
./quadratic
```

Automated Test Infrastructure

- You may be familiar with
  - JUnit
  - Google's C++ xUnit
  - … (put your favorite programming language)
- Less familiar with
  - TTCN-3
  - … (organization-specific set up)

What changes ?

- the syntax

What does not change ?

- we need to decide what inputs/outputs to choose !!!

```
public void evaluatesExpression()  {
  Calculator calculator = new Calculator();
  int sum = calculator.evaluate("1+2+3");
  assertEquals(6, sum);
}
```

```
TEST(CalculatorTest, sumOneTwoThree) {
  Calculator calculator;
  int sum = calculator.evaluate("1+2+3");
  EXPECT_EQ(6, sum);
}
```

```
template calculatorRequest request1 := {
  input := "1+2+3"
}
template calculatorResponse response1 := {
  output := 6
}
testcase test1() runs on MTCType {
  calculator.send(request1);
  alt {
    [] calculator.receive(response1) {
         setverdict(pass)
    }
    [] calculator.receive {
         setverdict(fail)
    }
  }
}
```