DD2459 Software Reliability Lecture 4

Requirements Testing and Requirements Modeling

(see Amman and Offut, Chapter 14)

Part 1. Oracles, Bugs and Requirements Models

Topics

- Why requirements testing?
- How to capture precise requirements
  1. Black-box function models
  2. Program contracts

Part 1

Part 2

  3. Java Modeling Language (JML)
  4. Metamorphic equations
  5. Temporal logic

Why Requirements?

- So far we have looked at testing a program in systematic ways:
  - Structurally - searching through paths and control points
  - Black-box - searching through input data
- However, we have not considered the oracle problem.
- How do we deliver a verdict (pass/fail) on a test case?

    - manually?

    - automatically?

What is a bug?

- Consider the following error hierarchy:

  1. Syntax errors: (caught by compiler? Compile-time errors)
  2. Type errors, either caught by compiler or generate run-time errors
  3. Semantic errors, exceptions such as null pointers, divide by zero (generate *untrapped exceptions*)
  4. Behavioral errors: memory leakage, non-determinism, race conditions, unsynchronized threads, infinite loops.

     (may <u>not</u> generate untrapped exceptions)
  5. Requirements errors: code never crashes or hangs, but does the wrong thing.
  6. Performance errors: code does the right thing at the wrong time.

Mutation Theory
Level 2,3 Errors

- We mentioned medicine as an error model
- Would like something similar for testing
- Mutation theory provides a simple model of errors, that we will study in Lecture 6.
- Basic Idea: mutate (i.e. transform) the code to inject bugs
  - E.g. `x = (y + z )` becomes `x = (y – z)`
- See if test suite uncovers these bugs
- Mainly checks quality of a test suite rather than SUT

Static Checking
Level 4 errors

- The world of testing overlaps with other software quality assurance (SQA) methods
- E.g. static checking.
- Static checkers analyze source code looking for *specific kinds* of errors.
- Example: *Purify looks for memory leakage*
- Tools tend to be very efficient, but restricted to "*pre-defined*" errors.
- Can detect *liveness errors* e.g. non-termination

Requirements Testing Methods

Level 5 Errors

- Shouldn’t we should be testing the user requirements and not the code?
- Source: user requirements document
- Problems:
  - it may not exist!
  - undocumented legacy code?
  - user may have vague requirements!

Modeling
User Requirements

To make the oracle step clear, we must make user requirements clear. How?

1. Natural language?
2. Visual modeling languages e.g. UML?
3. Formal modeling languages e.g. JML, temporal logic?
4. Reference model e.g. TCP/IP protocol

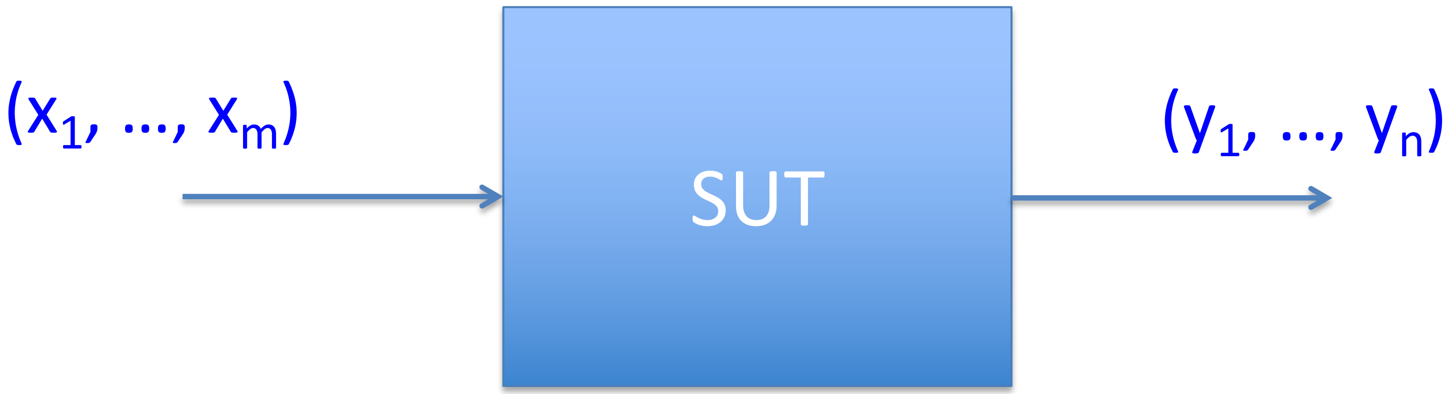
   model-based testing (Lecture 5)

Four Requirements Modeling Techniques

- We will consider four methods for modeling requirements precisely.
- No false positives or false negatives
- Accurate models lead to tools that can automate tasks
  - Test case generation
  - Test case execution
  - Verdict generation (test oracle) MAIN GOAL!
  - Measure coverage

Procedural Programs

A procedural ("C"-style) SUT takes in an input vector and produces an output vector. It may terminate, but maybe not always?



Method 1:
Black-box Function Model, Level 5

SUT described as a partial function

$$f_{SUT} : A_1 \times .. \times A_m \rightarrow B_1 \times .. \times B_n$$

A value $f(x_1 ,..., \ x_m)$ may be *undefined* if code does not terminate on $x_1 ,..., \ x_m$

Examples: Math Functions

- Sometimes we can describe $f(x)$ explicitly, e.g.
- $f(x) = \sqrt{x}$
- $f(x) = ax^2 + bx + c$
- $f(x) = \int_{x=i}^{x=j} g(x)$
- $f(\text{empty}) = \text{empty}$ & $f(x) = f(x).\ \text{head}(x)$
- $f(\text{empty}) = \text{empty}$ & $f(\text{push}(x, s)) = s$
- Mostly applicable to *mathematical problems*

Example:
Myers' Triangle Program

See Lab 1

Triangle : Int * Int * Int -> $\{$ scalene, isosceles, equilateral $\}$

```
Triangle( x, y, z ) = equilateral
          if x == y == z
Triangle( x, y, z ) = isosceles
          if x == y or y == z or x == z
Triangle( x, y, z ) = scalene
          if x != y & y !=z & x != z
```

Is this specification correct? If not fix it!

Method 2
Program Contracts, Level 5

- Try to model program behavior.
- Black-box requirements, *what and not how*?
- Define a contract between a component and its environment

  requires aka. precondition

  ensures aka postcondition
- *If environment guarantees precondition then component guarantees postcondition*
- Suitable for all levels of testing: unit, integration, system.

Programming Logic

- Use logic to build up pre and postconditions
  - Requires: P1 & P2 & …
  - Ensures: Q1 & Q2 & …
- P1 and P2 must hold <u>before</u> execution of SUT
- Then Q1 and Q2, … will hold <u>after</u> execution of SUT
- If P1 or P2 or … doesn't hold we know nothing!

Contract Examples for Components

- Requires: $x >= 0$
- Ensures: $|y * y - x| < \varepsilon$
- Component: Square root method

Suppose A: array(integer)

- Requires: $i < j \Rightarrow A[i] <= A[j]$
- Ensures: $y \in \{ A[1], \ldots, A[m] \}$
- Component: searching for y in an ordered array

- Requires: True (what does this mean?)
- Ensures: $A[1] <= A[2] <= \ldots <= A[m]$
- Component: sorting an unordered array

DD2459 Software Reliability
Lecture 4

Requirements Testing
and Requirements Modeling

(see Amman and Offut,
Chapter 14)

Part 2. JML, Metamorphic and Timing Models

Java Modeling Language
JML

- Modeling language for Java contracts
- What but not how!
- Therefore JML is not a programming language
- Java comments are interpreted as JML annotations when they begin with an @ sign

  //@ <JML specification> or

  /*@ <JML specification> @*/
- JMLUnitNG, JMLOK2, tools to generate test cases from JML annotated Java files
- ESC/Java2, an extended static checker which uses JML annotations to perform static checking

JML Markup

***Note**: No backslash needed*

requires Defines a precondition on the method that follows

ensures Defines a postcondition on the method that follows

signals Defines <u>at least</u> what exceptions can be thrown by the method that follows if precondition holds.

signals_only Defines <u>exactly</u> what exceptions can be thrown by the method that follows if precondition holds.

also Combines specification cases and can also declare that a method is inheriting specifications from its supertypes.

JML Basic Constructs

- An identifier for the anonymous return value of
  any method.
  \result

- A modifier to refer to the value of the <expression> at the time of entry into a method.
  \old(<expression>) e.g. `\old(x)`

- Includes all Java expressions, including array access and object dereferencing.
  e.g. A[i], A.length

JML First Order Logic

- All Java relations e.g. x == y,   x >= y,   x !=y

- All Java Boolean operators & (and), I (or) ! (not) **==>** (implies) and lazy operators &&, II.
- e.g. (x == y I x > y)

- The universal quantifier

  (\forall <decl>; <range-exp>; <body-exp>)

```
Example
(\forall int i; 0 <= i < A.length; A[i+1] > A[i])
```

- The existential quantifier

  (\exists <decl>; <range-exp>; <body-exp>)

Useful JML operators

Smallest solution to a constraint.

```
(\min <decl>; <range-exp>; <body-exp>)
```

Largest solution to a constraint.

```
(\max <decl>; <range-exp>; <body-exp>)
```

Sum of solutions to a constraint.

```
(\sum <decl>; <range-exp>; <body-exp>)
```

Product of solutions to a constraint.

```
(\product <decl>; <range-exp>; <body-exp>)
```

Number of solutions to a constraint.

```
(\num_of <decl>; <range-exp>; <body-exp>)
```

```
Example:
(\num_of int i; 0 <= i < A.length; A[i] >= 0)
```

Hint: You will need to use some of these in lab 2!

JML Example

```
public class BankingExample
{

//@ requires 0 < amount && amount + balance <MAX_BALANCE;
//@ ensures balance == \old(balance) + amount;
public void credit(final int amount)
   {
        this.balance += amount;
   }
//@ requires 0 < amount && amount <= balance;
//@ ensures balance == \old(balance) - amount;
public void debit(final int amount)
  {
        this.balance -= amount;
  }
```

```
    //@   requires !isLocked;
    //@   ensures \result == balance;
    //@ also
    //@   requires isLocked;
    //@   signals_only BankingException;
    public  int getBalance() throws BankingException
    {
        if (!this.isLocked)
        {
                return this.balance;
        }
        else
        {
                throw new BankingException();
        }
    }
}
```

Method 3:
Metamorphic Testing, Level 5

- Requirements testing with a pair of tests t_1, t_2
- Basic idea: mutate t_1 into t_2 and compare the outputs

- Three test principles at work:
  – Test very simple requirements to automate the oracle, equations and inequalities
  – Cross check outcomes of a pair of test cases
  – Automatically generate test cases by metamorphic transformation.

Simple Example

- Consider the trigonometric sine function
- $\mathrm{Sin}(x) : R \to R$

- Have some implementation under test
- `float mySineCode( float x ) { … }`
- How do we know `mySineCode(.)` is correct?

- Can test a specific value, e.g. $\sin(\pi / 4) \approx 0.7071$
- But what about other values?

Metamorphic Equations

(see Amman and Offut Section 14.2.4)

- Many trigonometric laws e.g: $\sin(x) = \sin(\pi - x)$
- So $\sin(\pi / 4) = \sin(3\pi / 4)$
- Now we can avoid predicting lhs or rhs (which assumes we have some other way of knowing the result)
- Compare this approach with contracts!

Basic Test Procedure

1. Execute `mySineCode(`$\pi / 4$`)` => y_1
2. Execute `mySineCode(`$3\pi / 4$`)` => y_2
3. If $y_1 = y_2$ Pass else Fail

More Generally: a Data Mutation

- Initial test value x (seed test case)
- Metamorphic equation: $f(x) = f(T(x))$
- T is the data transformation
- T generates a sequence of test cases

$$x, T(x), T^2(x) = T(T(x)), \ldots, T^n(x)$$

- n metamorphic test cases

- n-1 metamorphic equations give the test oracles

$$f(x) = f(T(x)), \quad \ldots \quad , f(T^n(x)) = f\,(T^{n+1}(x))$$

Application Areas, Timeline to 2015

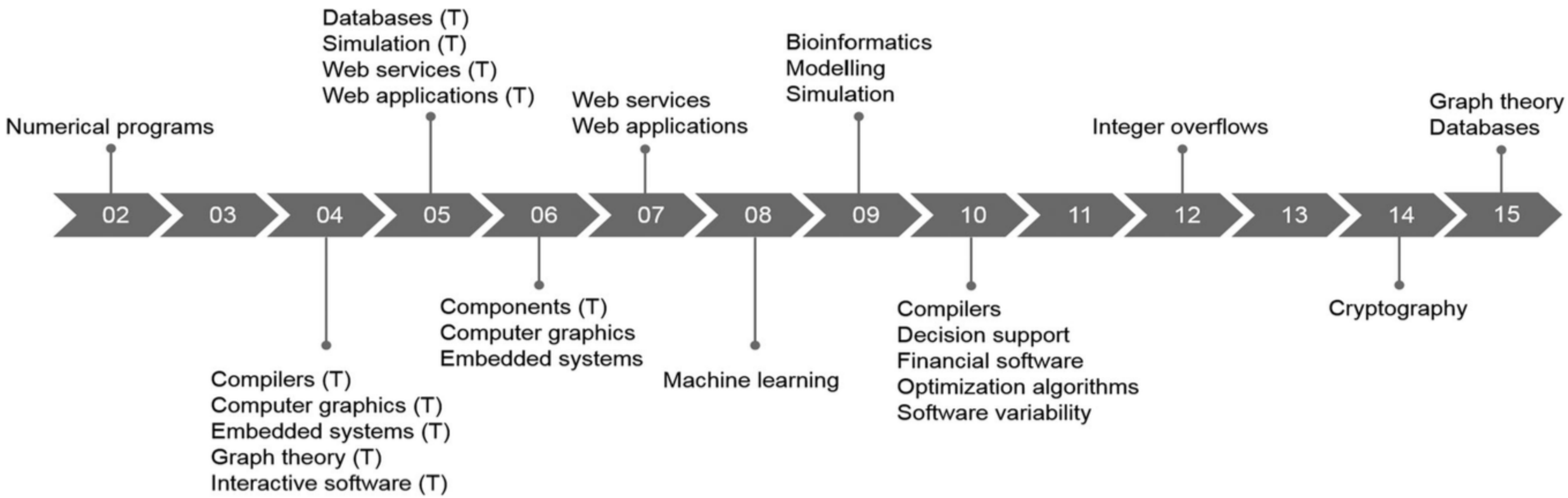
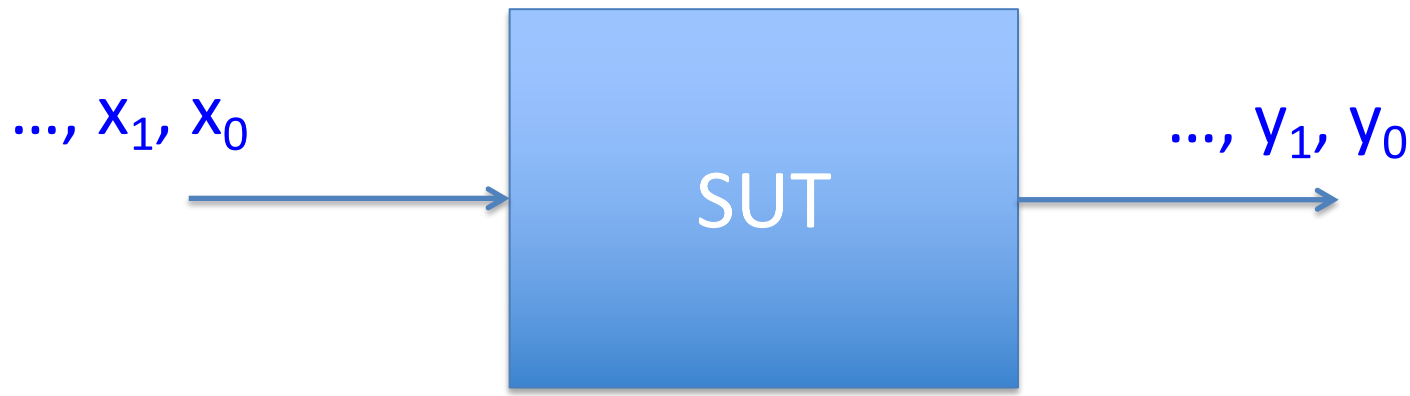


Fig. 5. Timeline of metamorphic testing applications. Domains marked with (T) were only explored theoretically.

Embedded and Reactive Systems

- *Reactive systems* respond continuously to environment events (stimuli) over time (e.g. servers)



Time may be *relative* or *absolute*, *discrete* or *continuous*

Method 4: Temporal Logic
Level 5, 6 Errors

- Embedded systems need to talk about the order of events in a physical model of time
- No end to time
- So pre/postconditions are no longer appropriate
- Temporal logic is one option
- Several kinds of temporal logic
  - Linear temporal logic
  - Computation tree logic (branching time for concurrency)

Propositional Linear Temporal Logic (PLTL)

- Basic propositions
  - buttonPressed, lightOn, switchOff ,...
- Boolean operators
  - F & G, F I G, !F, F$\Rightarrow$ G
- Temporal operators (modalities)
  - always F, sometime F, next F, (G until F)
- Time can be relative or absolute
- All depends on what next F means

PLTL Examples

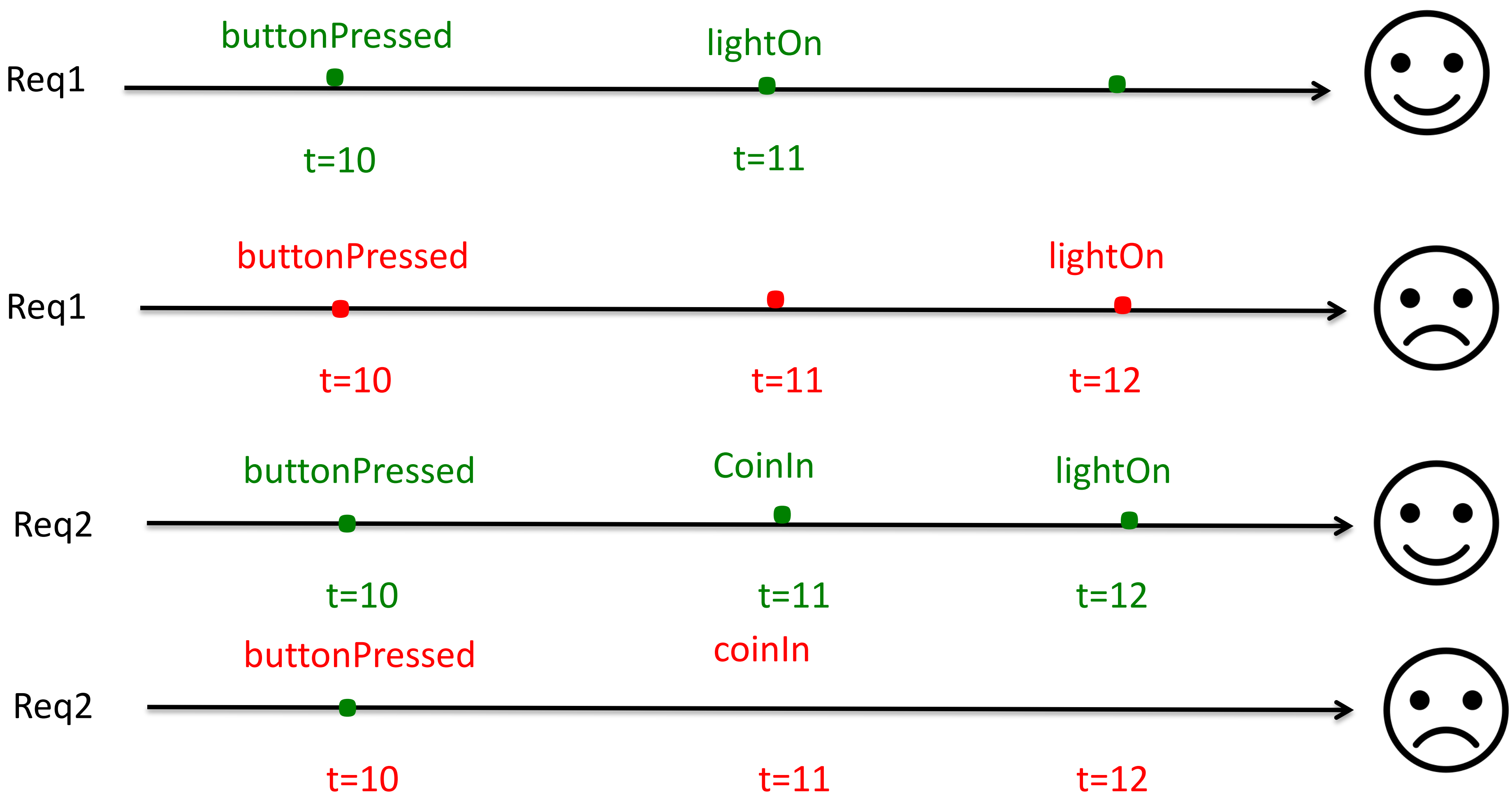
Req 1: Always (buttonPressed $\Rightarrow$ next(lightOn))

It always holds that if the button is pressed now then in the next time (from now) the light is on.

Req 2: Always (buttonPressed & next(coinIn)
$\Rightarrow$ next^2 (lightOn))

It always holds that if the button is pressed now and then a coin is fed in then in the second time (from now) the light is on.

Traces and Counterexamples



Black-box testing of Reactive Systems

- Given a user requirement as an LTL formula F
- Try to stimulate a behaviour that violates F
- Such an example is a counterexample trace
- Use input data to define test case.
- Observe output data
- Combine the two into a single trace t and evaluate the formula F on t.

Lecture 4 Summary

- Seen a hierarchy of types of software error
- Seen requirements modeling techniques mainly for Levels 5 and 6
- Four techniques for Level 5-6 errors
  - Function models
  - Contracts
  - Metamorphic testing
  - Temporal logic