JUnit
assertEquals(x,y) : test if x=y

Lecture 4

Requirements Testing
& Requirements Modeling

Topics

- Why requirements testing? Point of it is to achieve automation
- How to capture precise requirements
  - Reference models
  - Pre/postconditions
  - JML
  - OCL
  - Temporal logic
  - Timed automata

Why Requirements?

- So far we have looked at exercising a program in systematic ways. Either:
  - White-box
  - Structurally, by searching its paths and control points
  - Black-box, by searching through collections of behaviours
- However, we have not considered the oracle problem. How do we deliver a verdict on a test case?

  - manually?

  - automatically? Preferred

What are we searching for?

- We need to consider what kind of errors we are interested in:
  - Syntax errors: (caught by compiler?)
  - Type errors, either caught by compiler or generate a run-time error
  - Semantic errors, exceptions such as null pointers, divide by zero (untrapped exceptions)
  - Behavioral errors: memory leakage, non-determinism, race conditions, unsynchronised threads, infinite loops.
  - Requirements errors: code never crashes, but does the wrong thing.
  - Performance errors: code does the right thing at the wrong time.

Static Checking

- The world of testing overlaps with other QA methods such as static checking.
- Static checkers analyse 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.
- Focuses on liveness issues e.g. termination

Requirements Testing

- Surely we should be testing the requirements and not the code?
- Source: user requirements document
- Problem:
  - may not exist!
  - undocumented legacy code
  - user may have vague requirements

How to model user requirements?

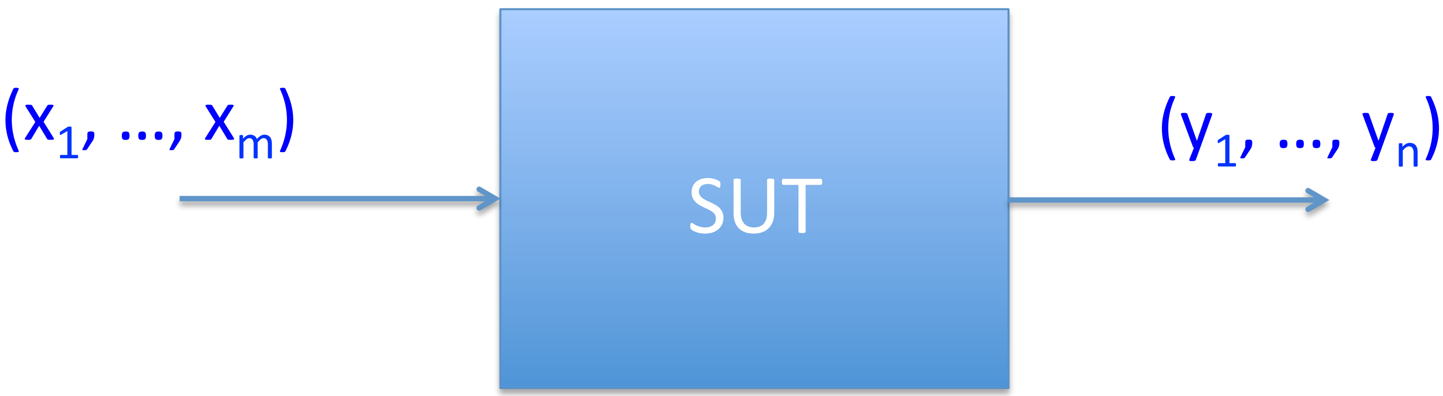
- To make the oracle step clear, we must make user requirements clear. But how?
- Natural language?
- Visual modeling languages e.g. UML?
- Formal modeling languages e.g. JML, temporal logic?
- Reference model e.g. TCP/IP protocol

Requirements Modeling Techniques

- We will consider some methods for modeling requirements accurately.
- Accurate models lead to tools that can automate tasks
  - Test case generation
  - Test case execution
  - Verdict generation (the oracle step)
  - 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.



Maps vector of x to vector of y.

We will assume variables x_i and y_j are <u>disjoint</u>.

Partial Function Model

- If $x_1{:}A_1 ,\dots,\ x_m{:}A_m$ and $y_1{:}B_1 ,\dots, y_n{:}B_n$ then SUT can be described as a partial function

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

i.e. some values of $f(x_1 ,\dots,\ x_m)$ may be undefined

Examples

- Perhaps 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(empty) = empty & f(x) = f(x). head(x)
- f(empty) = empty & f(push(x, s)) = s

Specify the Triangle program

- 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!

Data Types or Reference Models

- This might be termed the specification method of abstract or concrete data types.
- Effectively sets up a reference model for behaviour.
- Can use reference model to predict outputs.
- Can use equations to define reference model, or borrow any existing reference model.
- Need to be able to execute reference model itself.

Problems

- This method can be useful if we have a clear idea what we want, and the system is not too large to write down.
- Reference model becomes a system prototype
- Reference model is correct?
- Tends to <u>overspecify</u> system
  - What if a range of outputs is acceptable?
  - Can’t pick one unique reference output!

Specifying system properties

- More economic and practical to define key behavioral properties.
- Black-box requirements
  - Requires: P1, P2, …
  - Ensures: Q1, Q2, …
- P1, P2 must hold <u>before</u> execution of SUT
- Then Q1, Q2, … will hold <u>after</u> execution of SUT
- All bets are off if P1 or P2 or … doesn't hold
- Contract between component and environment

  Requires = precondition

  Ensures = postcondition
- Can use logic to build up pre and postconditions

Examples

- Requires: x >= 0
- Ensures: $|\, y * y - x \,| < \varepsilon$ epsilon is a very small number

Take any 2 index i and j, if i < j then, array value i must be smaller or equal to array value j.

- Requires: $i < j \Rightarrow A[i] <= A[j]$
- Ensures: $y \in \{ A[1], \ldots, A[m] \}$ y is a member of the array.

- Requires: True True: for any input
- Ensures: $A[1] <= A[2] <= \ldots <= A[m]$

What familiar computations do these contracts express informally?

Java Modeling Language (JML)

- A first-order language for talking about black-box requirements on Java programs.
- Java comments are interpreted as JML annotations when they begin with an @ sign

  //@ <JML specification> or

  /*@ <JML specification> @*/
- JMLUnit, a tool to generate files for running JUnit tests on JML annotated Java files
- ESC/Java2, an extended static checker which uses JML annotations to perform static checking

JML First-order language

- **\result**
- An identifier for the return value of the method that follows.
- **\old(<expression>)**
- A modifier to refer to the value of the <expression> at the time of entry into a method.
- **(\forall <decl>; <range-exp>; <body-exp>)**
- The universal quantifier
- **(\exists <decl>; <range-exp>; <body-exp>)**
- The existential quantifier
- **a ==> b**
- a implies b
- includes Java Boolean operators & (and), | (or) ! (not) and lazy versions &&, ||.
- Includes all Java expressions, including array access and object dereferencing.

JML meta-notation

requires Defines a precondition on the method that follows

ensures Defines a postcondition on the method that follows

signals Defines a postcondition for when a given exception is thrown by the method that follows.

signals_only Defines what exceptions may be thrown when the given precondition holds.

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

```
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;
 //@ assignable balance;
//@ ensures balance == \old(balance) - amount;
public void debit(final int amount)
 {
      this.balance -= amount;
 }
```

```java
  //@   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();
     }
  }
}
```

Embedded and Reactive Systems

- Reactive systems continuously respond to environment events (stimuli) over time

Always online, never terminate.
Will run indefinitely.
Have to keep feeding it data all the time.
Have to manage this pipeline.

Time may be relative or absolute, discrete or continuous

Temporal Logic

- No fixed termination point No end point --> cannot use post condition
- So clearly pre/postconditions are no longer appropriate
- Temporal logic is one option
- Many types of temporal logic
  - Linear temporal logic
  - Computation tree logic
  - CTL*

Propositional Linear Temporal Logic (PLTL)

- Basic propositions
  - buttonPressed, lightOn, switchOff ,...
- Boolean operators
  - F & G, F I G, !F, F$\Rightarrow$ G
- Temporal operators (modalities) will bring about the concept of time
  - always F, sometime F, next F , (G until F)
    e.g. lightOn until 7pm
- Relative time
- Independent of absolute (wall-clock) time

Example

next always reference time

$$\text{Always (buttonPressed} \Rightarrow \text{next(lightOn))}$$

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

$$\text{Always (buttonPressed \& next(coinIn)}$$

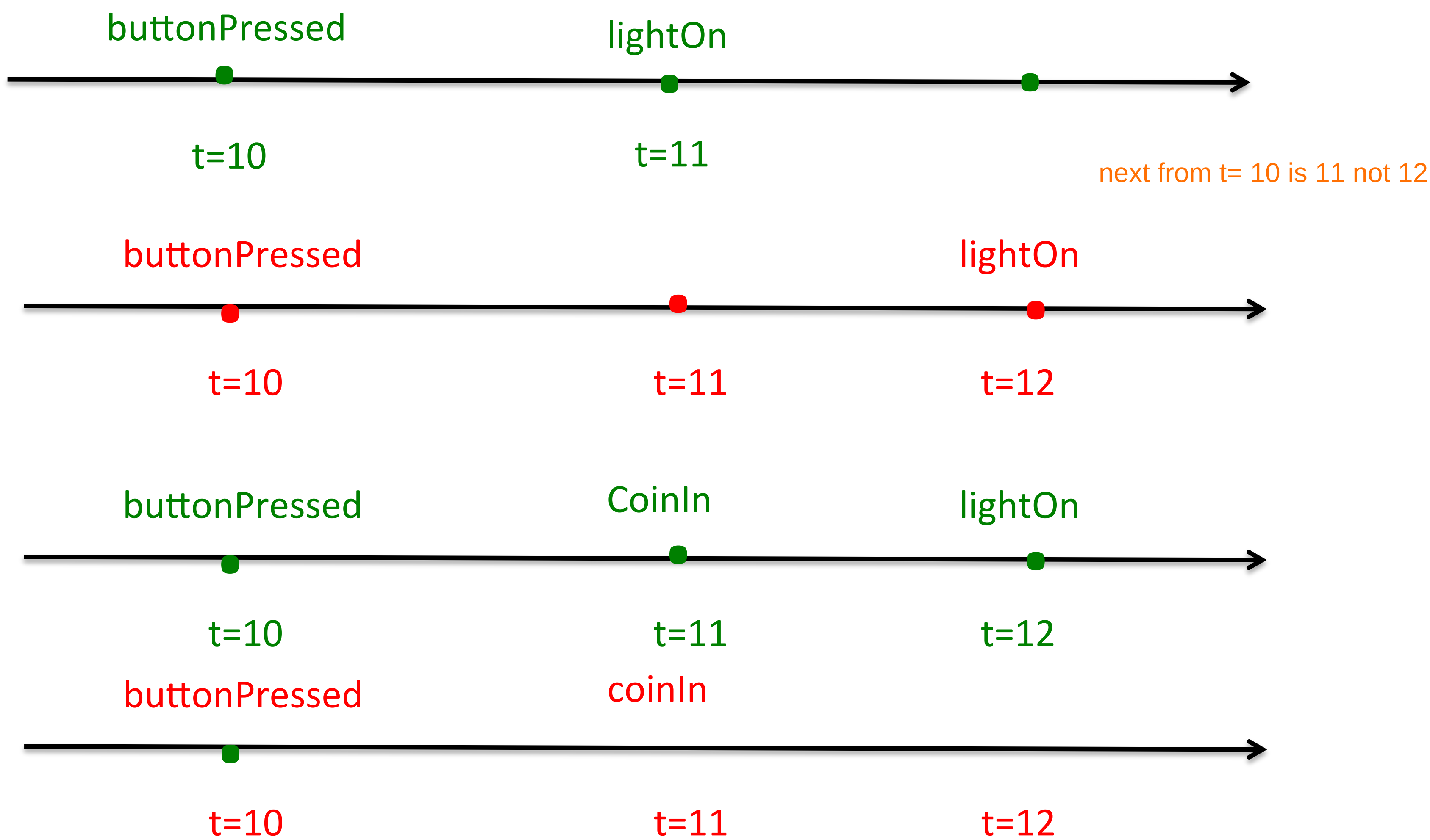
$$\Rightarrow \text{next}^2 \text{(lightOn))}$$

means next-next

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.

Trace Examples & Counterexamples

green: time-line makes requirement true
red: time-line makes requirement false



Black-box testing of reactive systems

linear temporal logic

- Given an LTL formula F, try to stimulate a behaviour that violates it.
- Use input data to define test case.
- Store observed output data
- Combine the two into a single trace t and evaluate the formula F on t.

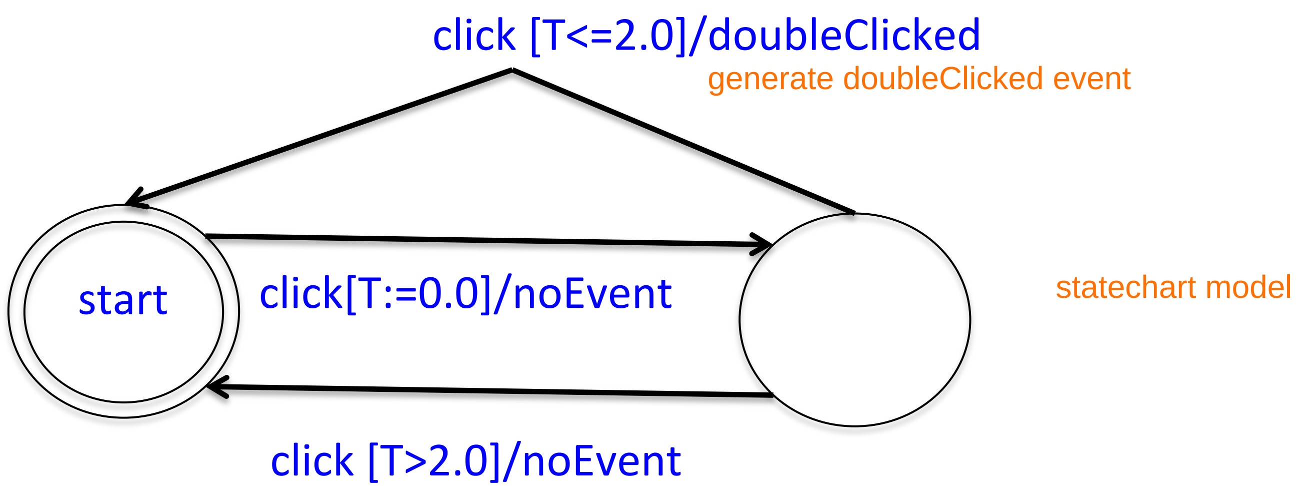
Here, doing right thing on the wrong time is still a failure. It is not the case in procedural programming.

Hard Real-time Systems

- Hard real-time systems have strict time constraints.
- Usually absolute, real or wall-clock time
- Relative to a local clock
- Use timed traces
- $(x_1, t_1), (x_2, t_2), \ldots., (x_n, t_n)$
- $x_1, \ldots, x_n \in$ Prop are events (in or out)
- $t_1, \ldots, t_n \in \Re$ are real-valued times

Real-time Automata

- Automata with one (or more) clocks and timed transitions



Double click detection for a mouse

Black-box testing of timed systems

- Given a hard real-time SUT and a reference timed automaton, try to stimulate a timed behaviour that violates the reference automaton.
- Use input data to define test case.
- Store time stamped output data
- Combine the two into a timed trace and evaluate on the reference automaton.

Mutation Theory

- Mutation theory provides a model of errors, that we can study later.
- Basic Idea: mutate (transform) the code to introduce bugs.
- See if test suite uncovers these bugs
- Checks robustness of test suite?

Compare JML with UMLs OCL
Object Constraint Language

context Account::withdraw (amount : Real)

pre: amount <= balance

post: balance = balance@pre - amount

context Account::getBalance() : Real

post : result = balance

- OCL Evaluator a tool for editing, syntax checking & evaluating OCL
- Octopus OCL 2.0 Plug-in for Eclipse

OCL collection language

size The number of elements in the collection

count(object) The number of occurences of object in the collection.

includes(object) True if the object is an element of the collection.

isEmpty True if the collection contains no elements.

iterate(expression) Expression is evaluated for every element in the collection.

sum(collection) The addition of all elements in the collection.

exists (expression) True if expression is true for at least one element in the collection.

forAll (expression) True if expression is true for all elements.

one(expression) Returns true if exactly one element satisfies the expression