Testing of processes

Topics:
- Testing systems using an operational specification
- Traces, and coverage criteria for trace-based testing
- The problem of non-determinism
- Use of analysis rather than testing to verify these systems

Traditional testing methodology

Unit testing:
- Inputs: module and its functional specification
- Task: develop test suite that covers either the structure of the module or its specification

Integration testing:
- Inputs: assembly of modules and functional specification of the behavior of the assembly
- Task: develop test suite that covers the specification

Note: Tests assumed to comprise set of inputs and expected outputs; unit/program expected to terminate

Interactive/concurrent systems

A large and interesting class of systems cannot be tested in the traditional manner

E.g., systems that do not terminate, such as interactive systems
- behave like a process, not a function
- so can’t easily test against functional specs

E.g., systems that employ concurrency
- may behave like a non-deterministic process
- even if functional spec exists, executions that demonstrate failures difficult to reproduce

View of a system under test as a process

Observation “ports” over which data may be exchanged (input or output)

Testing environment

Customized testing environment
- exercises the system
- observes inputs that the system observed over ports
- observes outputs that the system offers over ports
- may serve as an oracle

Specification of a process

Expresses temporal behavior of observable actions or events

Notations:
- State-event diagrams
- Process algebras

Typically represents non-terminating behavior
May be deterministic or non-deterministic
- To begin, we limit discussion to deterministic processes
Example: Vending machine controller

Consider the controller for a vending machine
- allows users to insert quarters
- enables user to select to dispense candy or coffee
  - candy costs 25¢
  - coffee costs 50¢

In process terms:
- ports include "quarter", "candy", and "coffee"
- controller offers (to the environment) the opportunity to interact over these ports
- controller observes interactions that are offered by the environment over these same ports

Vending machine specification

Operational specifications

Exhibits behaviors observable over its ports:
- each distinct behavior is called a trace
- infinite number of traces in this system
- each trace is infinitely long!

Simplified model of testing:
- Test "input" designed to exercise a trace
- A test suite contains tests for some finite subset of the possible traces

Example: Vending machine
Example: Vending machine

- start
- quarter 50Cents
- candy 25Cents

Execution history: quarter → quarter

Example: Vending machine

- start
- quarter 50Cents
- candy 25Cents

Execution history: quarter → quarter

Example: Vending machine

- start
- quarter 50Cents
- candy 25Cents

Execution history: quarter → quarter → coffee

Example: Vending machine

- start
- quarter 50Cents
- candy 25Cents

Execution history: quarter → quarter → coffee

Example: Vending machine

- start
- quarter 50Cents
- candy 25Cents

Execution history: quarter → quarter → coffee

traces = { quarter → candy → quarter → candy → ..., quarter → quarter → candy → quarter → ..., quarter → quarter → candy → candy → ... }
Testing such systems

To test such a system means to exercise one or more of its traces.

**Problem:** How do we exercise a trace in a finite amount of time?

Trace checking

Traces are infinite, but we can test finite prefixes of any given trace:
- E.g., to test that our system supports the trace `quarter → candy → quarter → candy → ...`
- We could design a test for the prefix `<quarter, candy, quarter, candy, quarter candy>`

Remains to:
- Select a set of traces based on some coverage criterion:
  - E.g., event coverage, event interaction coverage, length-n event sequence coverage
- Design each test

Effectiveness of trace-based testing

Problems with testing systems with process (rather than functional) specifications:
- Real test exercises only a finite prefix of a trace
- Effectiveness assumes reproducibility of tests

Second assumption is a big one and is not reasonable for a large class of systems

Non-deterministic vending machine

**Question:** How does this vending machine differ from our previous example?

**Answer**

Unlike previous machine, after accepting two quarters, this machine may refuse to allow the user to select coffee (or candy).

This occurs because, from within state 25Cents, a quarter event can transition the system into either state 50C_1 or 50C_2, and the choice of which is up to the system (not the user):
- In state 50C_1, system refuses quarter and candy events
- In state 50C_2, system refuses quarter and coffee events

Problem

Suppose we build a system with the behavior of the non-deterministic VM spec and test it against the deterministic VM spec.

The traces of both specs are completely indistinguishable.

Thus, exercising traces may fail to discover the refusal properties!
Expected reaction of system...

During testing, difficult to determine the "state" of the system being tested
- E.g., after inserting a quarter, how would you know you’re in “25Cents” state?

Answer

Trace through the model to determine state the system should be in after executing an event sequence $\sigma$
Figure out the set $\text{ready}(\sigma)$ of actions that should be “enabled” after simulating $\sigma$, according to the model
Then verify that these and only these actions are enabled

Example ready sets

$\text{ready}(\text{<>}) = \{ \text{quarter} \}$

$\text{ready}(\text{<quarter>}) = \{ \text{quarter, candy} \}$

$\text{ready}(\text{<quarter, quarter>}) = \{ \text{candy, coffee} \}$

Example ready sets

$\text{ready}(\text{<>}) = \{ \text{quarter} \}$
$\text{ready}(\text{<quarter>}) = \{ \text{quarter, candy} \}$
$\text{ready}(\text{<quarter, quarter>}) = \{ \text{candy, coffee} \}$
$\text{ready}(\text{<quarter, quarter, candy>}) = \{ \text{quarter, candy} \}$

...
Test using ready sets

ready<> = { quarter } ???

ready<quarter> = { quarter, candy } ???

ready<quarter, quarter> = { candy, coffee } ???

Test using ready sets

ready<> = { quarter, candy } VERIFIED!!!
Support for checking ready sets

Verifying program is receptive to actions in the ready set requires some system-level support
- E.g., instrumentation to check status of key program objects
- E.g., screen-scraping technology in GUIs

Verifying correctness of non-deterministic processes

Non-deterministic processes

Significantly more difficult to verify than a deterministic process
- New kinds of errors, which may not manifest when exercising traces
- E.g., concurrent system with multiple threads accessing shared data
  - Non-deterministic execution ordering can mean a test that passes during one trial will fail during another

Need to verify process behavior of a system often dominates other design concerns

Two-tiered approach

Much work put into designing and verifying the specification of such a system
- Analysis effort to remove, bound, or verify properties about the non-determinism in the spec
- Refinement techniques used to connect the (verified) specification to the code

Examples:
- Use of specification-based code generators
- Design methods that refine model structures into code structures (e.g., Magee and Kramer).

Analytical approach

Begin with an operational model, SPEC1, of the intended behavior
Construct an operational model, SPEC2, that accurately reflects the real design
- Typically much more complex (less direct) than SPEC1
Verify that SPEC1 and SPEC2 are testing equivalent,
  - tr(SPEC1) = tr(SPEC2)
  - Refusals of SPEC2 conform to those of SPEC1

Example: Vending machine
Example: Vending machine

Start quarter

50 cents

Quarter

Coffee

25 cents

Quarter

Candy

CSE 435: Software Engineering
K. Stirewalt
Example: Vending machine

- start
- quarter 50Cents
- candy
- quarter 25Cents
- coffee
- quarter candy
- quarter candy
- quarter coffee