Unit Testing

Rob Bocchino
NASA Jet Propulsion Laboratory
October 23, 2024

Jet Propulsion Laboratory
California Institute of Technology

Introduction

- Testing is an important part of FSW development
- We usually divide testing into at least two phases
  - **Unit testing:** Test individual units (e.g., F Prime components)
  - **Integration testing:** Test the integrated system
- This section covers unit testing
- Thorough unit testing is critical
  - It makes integration easier
    - Localized errors are caught early
    - When you get to integration, issues you see are system-level issues
  - It provides unit-level regression tests

Goals of Unit Testing

- Cover all component-level requirements
- Achieve high **code coverage**
  - 100%, or close to it
  - This is usually feasible, but it requires effort
  - Time and budget constraints may limit the amount of coverage
- Achieve a reasonable amount of **state and path coverage**
  - This is harder to measure
  - It is usually satisfied by writing tests that cover desired behaviors

Mapping Tests to Requirements

- If you have component-level requirements, then
  - The requirements should drive the tests
  - It's good to maintain a record of how the tests cover the requirements
- The mapping can be recorded
  - In a separate table or spreadsheet; or
  - In the tests themselves (e.g., in comments or in console output)
- See examples in the unit tests for the F Prime Svc components

Writing Unit Tests

- A standard approach
  - Write a first complete test that covers a requirement
  - Write a second complete test, etc.
  - Where there is overlap, copy-paste (bad!) or refactor into functions (better)
- A more disciplined approach
  - Write functions that test individual behaviors
  - Write tests by composing the functions
  - We have developed support for this approach (rule-based testing)
- First approach is “easier” but can lead down a bad path
  - If you aren’t disciplined about refactoring, code duplication gets out of hand
  - For large test sets, this becomes a maintenance problem

Writing Test Code

- Treat unit tests as a programming problem
  - Apply the same style guidelines as for flight code
    - Pay attention to code structure and clarity
    - Avoid code duplication
    - Avoid undefined behavior, dangerous C and C++ code
  - Apply (almost) the same level of rigor as for flight code
    - Some FSW rules don't apply to test code
    - E.g., *malloc*/*new* and *free*/*delete* are allowed
- Don't throw down messy code to get coverage
  - Requires a bit more up-front work
  - But will pay off with more readable, maintainable, and modifiable tests

Picking Inputs

- Good tests require good inputs
  - “Good” means “exercising enough behavior”
  - “Enough” can be made precise on some dimensions
    - For example, code coverage
    - In practice it can be a qualitative judgment
- Techniques for picking inputs
  - Use arbitrary values, e.g., 42: “Easy,” but not robust
  - Use significant values, e.g., boundary values
  - Use random values
  - Use an analysis tool to pick values
- In F Prime, the STest framework has a random value picker

Modeling External Behavior

- A unit under test is part of a complete system
- When unit testing a component, you must model external behavior
  - For example, Command Dispatcher sends commands, expects responses
  - Test code must model behavior “receive command, send response”
- To do this, write a test harness or “mock system”
  - Think about relevant system behavior
  - Model it at the appropriate level of abstraction
- This approach fosters modularity of tests
  - Write one abstracted system model
  - Use it in many tests

Testing Against the Interface

- When writing unit tests for a component, test against the interface
  - For example, send commands, send data on ports
  - Avoid directly updating the state (member variables) of the component
    - You can **read** internal component state to verify it is OK
    - But try not to **modify** the state except through the interface
  - This approach leads to more structured, better tests
- Sometimes you want to test a function in a component implementation
  - E.g., if it implements a complex algorithm
  - In this case, write a test against the function interface

Testing Components That Use Libraries

- Sometimes components call into external libraries
- You can link against the library in the test
- However, it's usually better to link against a mock or stub library
  - Avoids complex library behavior
  - Makes it easier to induce behaviors for testing (e.g., inject faults)
  - May be the only option on some platforms

Checking Code Coverage

- **Code coverage** means
  - Of all lines of code in the source program
  - Which lines were run at least once in some test?
- Standard tools such as *gcov* can do this analysis
  - Compile tests with coverage flags
  - Run tests to generate .gcov files
  - Run *gcov* to analyze .gcov files and produce a report
- Report consists of source files with coverage annotations for each line
  - You can convert this into a percentage (covered / total)
  - Or use a wrapper tool such as *gcovr* to produce pretty output

Achieving Code Coverage

- It's usually easy to get to about 80% code coverage
  - Think about "ordinary" behavior of the code
  - Write tests that exercise it
- The cases that remain are usually rare or off-nominal behaviors
  - Covering these cases may require more thought
  - You may have to
    - Reason backwards from the desired behavior to synthesize inputs
    - Inject faults into library behaviors
- At some point you may hit diminishing returns
  - One common case is *assert(0)* in code that should never be reached
  - Use judgment

Limitations of Code Coverage Analysis

- 100% code coverage is not a panacea!
- It says nothing about
  - State coverage: Which system states were tested?
  - Path coverage: Which paths through the code were tested?
- In general, full checking of state and path coverage is not possible

Exercise: State Coverage

- Find two sets of values of *x:*
  - S1: *f* runs successfully with 100% code coverage over all runs
  - S2: *f* fails with an assertion
- Explain why this is possible

```
void f(U32 x) {
  assert(x != 0);
  if (x == 5) {
    …
  }
  else {
    …
  }
}
```

Exercise: Path Coverage

- Find two sets of values of *x*
  - S1: *f* runs successfully with 100% code coverage over all runs
  - S2: *f* fails with an assertion
- Explain why this is possible

```
void f(U32 x) {
  for (U32 i = 0; i < x; ++i) {
    assert(x < 10);
  }
}
```