Compliance-based Testing Workshop

Testing safety-critical software

Jim Thomas
TVS, Director of Software Testing
Content

1. What is Safety-critical Software?
2. Verification and Testing Requirements
3. The Cost of High Integrity Software
4. Software Requirements
5. Unit Testing
Safety-critical Software

• “A safety critical system is a system where human safety is dependent upon the correct operation of the system”

• Elements of safety critical systems:
  – Computer hardware
  – Other electronic and electrical hardware
  – Mechanical hardware
  – Operators or users
  – Software

• Traditionally associated with embedded control systems
Safety Standards

- **DO178**: Software considerations in airborne systems and equipment certification
- **EN50128**: Software for railway control and protection systems
- **IEC60880**: Software aspects for computer-based systems performing category A functions
- **IEC61508**: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems
- **IEC62304**: Medical device software -- Software life cycle processes
- **ISO26262**: Road vehicles – Functional safety
Quality Characteristics

- Defensive
- Resilient to errors
- Fail safe?
- Necessary functionality only
- Extensible and maintainable – long lifetimes
- Reliable
- Very low residual fault rates
Verification and Testing Requirements

- Depend on integrity level/class
- Software Verification Plan
- Specifications and Code Reviews
- Testing (against specifications)
  - Unit
  - Software Integration
  - Software System
- Test Coverage Criteria
- Requirements and Test Traceability
- Independence
• Define and document software system requirements (all safety classes)
• Verify each software unit and document results (classes B and C)
• Integration planning (classes B and C)
• Document results of integration tests and identify testers
• Software system testing linked to requirements, all requirements tested or otherwise verified
• Software verification planning – deliverables, acceptance criteria, milestones
### Dynamic analysis and testing

<table>
<thead>
<tr>
<th>Technique</th>
<th>SIL 1</th>
<th>SIL 2</th>
<th>SIL 3</th>
<th>SIL 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural test coverage (entry points) 100%</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>Structural test coverage (statements) 100%</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>Structural test coverage (branches) 100%</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>Structural test coverage (conditions, MC/DC) 100%</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>Test case execution from boundary value analysis</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>Test case execution from error guessing</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Test case execution from error seeding</td>
<td>-</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Test case execution from model-based test case generation</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>Performance modelling</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>Equivalence classes and input partition testing</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
</tbody>
</table>
Is High Integrity Software Expensive?

• It can be but ....

- 40-50% project effort on avoidable rework
- Extra 50% cost to develop high integrity software
- Extra 75% cost to maintain low integrity software

Source: Boehm – Software Management Article 2001
And …

Programmers <50% efficient at finding own defects

Reviews/Inspections often >65% efficient

Static analysis often >65% efficient

Inspections, Static analysis, Testing:
>97% efficient
Lower costs & schedules by >20%
Lower TCO by >45%

Source: Capers Jones – Software Quality in 2012: A Survey of the State of the Art
All the Right Stages but Not Necessarily in the Right Order

Software Requirements

High Level Design

Unit Level Design

Coding

Unit Testing

Integration Testing

Software System Testing
Agile Development

Sprint 1
Sprint 2
Sprint 3
Sprint 4
Sprint 5

Sprint 1 Dev
Sprint 2 Dev
Rwk
Sprint 3 Dev
Rwk
Sprint 3 Dev
Rwk
When done well ....
System v Software Requirements

- Documented for all integrity levels/classes
- Traceable to System Requirements
- Documented and auditable process to produce
- Without implementation bias
- All requirements to be tested or otherwise verified
1. When the receiver aircraft has broken away the ram air turbine blades shall be feathered, the green indicator light shall be switched off and the amber indicator light shall be switched on to indicate the pod is available for refuelling.
Deriving Software Requirements

- Scope: Boundaries and interfaces
- Defined H/W interfaces and system interfaces
- Specific behaviour software must exhibit
- Specific qualities software must have
- Specific constraints software must comply with
- Basis for software design
- Basis for software system testing
Software Context

Software system

- fuel connector state
- hose length
- fuel temp
- Turbine blade angle
- Red indicator
- Amber indicator
- Green indicator
- Refuelling console interface
In-Flight Refuelling System – Software Requirements

1. The fuel connector status will be on or off
2. The fuel connector status shall change from on to off only after the fuel connector state is read as off for five successive reads, with each read 1ms apart
3. When the fuel connector status changes from open to closed the turbine blade angle shall be driven to 0 at maximum rate
4. Etc.
5. If the fuel connector state read fails …..
What Testers Want ....

• **Requirements that are:**
  – Unambiguous
  – Consistent
  – Complete
  – Atomic
  – Uniquely identified
  – Traceable
  – Testable

• **Part of the review process**
Some Techniques to Use During Software System Test Design

- Requirements-based
- Analysis of inputs, outputs and state
- Black box techniques - equivalence partitioning, boundary value analysis, use case analysis, decision tables, all-pairs testing, state transition tables
- Error scenarios and negative testing
Unit Testing

- Verification mandatory for higher integrity levels
- Needs unit specifications
- Typically dynamic testing with coverage analysis
- Coverage criteria increases with integrity level:
  - Statement
  - Decision
  - Condition
  - MC/DC
The Unit Test Foundation

The Pyramid of Proof

Integration of Pre-tested Components
Unit Testing

- Provides confidence in units
- Provides integration predictability
- Provides basis for regression testing
- Provides visibility of progress
- Can be expensive so:
  - Efficient/fast
  - Repeatable
  - Automated
Unit Test Environment

- Input parameters:
  - Called Functions
  - Static Functions
  - Static data

- Output parameters:
  - Return
  - Global data

SUT (system under test)
Unit Testing Example

• Fuel Connector Specification
• Input:
  – connector_value
• Global data:
  – connector_status
  – prev_connector_value
  – connector_change_count
• Logic:
  • If the connector_value is the same as the connector_status there is no change
  • If they differ, then:
    – if the connector_value has changed since the last check (prev_connector_value <> connector_value) start the connector change count (set connector_change_count to 1)
    – Otherwise increment the connector change count, then check if the count has reached 5. If it has then change the connector_status to the connector_value
• Set prev_connector_value to connector_value
Unit Testing Example

• Logic:

• If the connector_value is the same as the connector_status there is no change

• If they differ, then:
  – if the connector_value has changed since the last check (prev_connector_value <> connector_value) start the connector change count (set connector_change_count to 1)
  – Otherwise increment the connector change count, then check if the count has reached 5. If it has then change the connector_status to the connector_value

• Set prev_connector_value to connector_value
Static Analysis

- IEC 61508
- “defensive programming, modular approach, coding standards, structured programming” would all be prudent
- How to show these have been considered?

- IEC 60880
- More specific. Examples:
  - GOTO statements as well as label variables should be avoided
  - Data structures and naming conventions shall be used uniformly throughout the system
  - Subroutines should have only a predefined maximum number of parameters
  - Subroutines should be organised as simply as possible

- IEC 62304, ISO 26262, DO-178 etc, etc
- Similar again, with varying levels of the specifics
What They Have in Common

• Define some rules that can be checked e.g.:
  – McCabe Cyclomatic Complexity < 10
  – Max nesting level < 4
  – No GOTO statements
  – All global variables prefixed with glb_
  – No dead code
  – No unused variables
  – etc

• Justify why these are sufficient

• Provide evidence that your development meets these rules
MISRA

• Coding standard, not a safety standard
  – MISRA C 2004
  – MISRA C++ 2008
  – MISRA C 3 / 2012

• Approach is to write safer software from the beginning by using a very restrictive subset of the language

• Using a pre-defined and well known coding standard reduces the amount of justification required for auditors
  – We’re using MISRA, except for... – great! 😊
  – We’re using our own awesome coding standard – justify yourself! 😞
Static Analysis Tools

- If you are using tool, rather than code inspections to verify your coding rules for a functional safety standard certified project then:
  - You just need to record and justify any deviations to the planned rules
  - Generate a report with any ignored defects and the justifications why each is accepted and with what controls
And importantly ...

More advanced static analysis tools also weed out critical bugs, early!

Unit/integration testing then focuses on finding design issues.
Summary

• Software Requirements Specification essential
• Specifying Software Requirements correctly provides the foundation for requirements testing
• Unit tests are required for higher integrity levels
• Unit tests provide the foundation for more predictable higher level testing
• Automated unit testing essential
• Review and static analysis complement testing
• Plan for test and verification