11
Verification, Validation & Certification

Distributed Embedded Systems
Philip Koopman
October 5, 2015
Where Are We Now?

◆ Where we’ve been:
  • How to build and analyze things
  • Testing – but that is only one way to evaluate how well something is built

◆ Where we’re going today:
  • Validation, Verification & Certification – making sure it works
    – Largely this focuses on the design correctness part of dependability
    – It should also deal with failure modes and safety

◆ Where we’re going next:
  • Economics
  • Test #1
  • Embedded networks
  • Mid-semester presentations
  • Dependable/safe/secure systems
Three related concepts:

- Verification: making sure each design step does what it was supposed to do
- Validation: making sure the end result satisfies requirements
- Certification: a written guarantee that a system is acceptable for operational use

General Approaches

- Testing
- Analysis
- Certification strategies

Areas of concern:

- Hardware correctness
- Software correctness
Why Is Time To Market Important?

- Profit window for consumer/commodity electronics may be 3 months
  - Moral: Get it right the first time; use good process to improve your odds
  - Sometimes – make profits on services/software, not hardware items
What’s The Cost Of A Finding & Fixing A Defect?

- Get it right the first time if you can
- If you get it wrong, find out about it as soon as possible
Is Speed The Key To Success?

- A fast design process only helps if you get it right
  - If you get it wrong, you get to spend more money fixing problems because you move further into the design before you find them!
Traceability

- Traceability is checking to ensure steps fit together
  - Starting point for most V&V

- **Forward Traceability:**
  - Next step in process has everything in current step
  - “Nothing got left out”

- **Backward Traceability**
  - Previous step in process provoked current step
  - “Nothing spurious included/no gold plating”

- **Traceability is an audit**
  - Doesn’t prove correctness if tracing is OK
  - But, problems are there if tracing fails
Definitions

◆ Verification:
  • The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
  • Loosely: forward traceability as design progresses
  • “Did we build the product correctly?”

◆ Validation:
  • The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.
  • Loosely: backward traceability to requirements
  • “Does the product do what it should?”

◆ Certification:
  • A written guarantee that a system or component complies with its specified requirements and is acceptable for operational use.
  • “Is an 3rd party happy enough with the product to stake his/her reputation on it?”

◆ Degree of required V/V/C often set by regulators (e.g., FAA)
General Approaches To V/V/C

◆ Testing
  • Execute system to determine behavior
  • Inject intentional faults to determine system response

◆ Analysis
  • Determine if desirable properties hold true; if undesirable properties exist
  • Find inconsistencies among design phases
  • Determine if design rules have been followed
  • Scrutinize design and documents (reviews)

◆ Process inspection
  • Determine if process is appropriate for desired end result
  • Determine if process was adequately followed

◆ Many techniques can be used for any of Verif., Valid., Cert.
  • But, some techniques are better fits for a particular use
Testing Review/Summary

- **White-box testing ("structural testing")**
  - Look at program structure and design tests
    - e.g., 100% of branch path coverage (both sides of each branch)

- **Black-box testing ("functional testing")**
  - Test every item on the functional specification
  - Also, test for robustness/error handling

- **Levels of test**
  - Unit test – testing small pieces of code; done by programmer
  - Module/functional test – testing at API level; done by testing group
  - Integration test – testing pieces working together; done by testing group
  - Acceptance test – testing whole system; done by customer (or surrogate)
  - Beta test – letting a few customers use product before full production
  - Regression test – make sure changes haven’t re-activated old bugs
Starting Points For Embedded Test Coverage

- Below are example useful coverage metrics
  - But remember from testing lectures – 100% coverage is not “100% tested”

- Requirements coverage
  - All requirements tested in every major operating mode

- Scenario coverage
  - All sequence diagrams tested (this is a form of system integration testing)

- Statechart coverage
  - All states visited
  - All arcs exercised

- Code coverage
  - Every statement in program executed (100% branch coverage)
  - Every exception handler exercised; every fault handler exercised

- FMEA coverage
  - FMEA = Failure Mode Effect Analysis (table predicting results of component faults)
  - Inject faults to see if FMEA correctly predicts system response
Things Other Than SW Get Tested Too!

◆ **Hardware testing**
  - “Shake & Bake” testing – temperature and vibration
  - STRIFE testing – stress + life – run just beyond hardware limits
    - E.g., 5% over-voltage and 5% over temperature
    - Components that fail are “weak”, and likely to be field failure sources
  - Margin testing
    - E.g., increase clock speed until something breaks
    - See if there is enough design margin to account for component variation & aging

◆ **System-level testing**  (“execution” of human use of system)
  - Usability tests
  - Check that maintenance can be performed within required time limits
  - Ensure that install & maintenance procedures work

◆ **Software gets stress tested … but nobody really knows what that means (in any rigorous way)!**
# Cost To Certify An IEEE 802.16e Aircraft Radio

<table>
<thead>
<tr>
<th>Environmental Conditions</th>
<th>Category and Requirement</th>
<th>Cost (Euro)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temperature / Altitude</td>
<td>DO-160E, Section 4, cat.A2</td>
<td>1800</td>
</tr>
<tr>
<td>Temperature variation</td>
<td>DO-160E, Section 5, cat. B</td>
<td>1200</td>
</tr>
<tr>
<td>Humidity</td>
<td>DO-160E, Section 6, cat. B</td>
<td>5500</td>
</tr>
<tr>
<td>Operational shock and crash safety</td>
<td>DO-160E, Section 7, cat B</td>
<td>1700</td>
</tr>
<tr>
<td>Vibration</td>
<td>DO-160E, Section 8, curve R</td>
<td>5000</td>
</tr>
<tr>
<td>Explosion proofness</td>
<td>DO-160E, Section 9, cat. X</td>
<td>n/a</td>
</tr>
<tr>
<td>Waterproofness</td>
<td>DO-160E, Section 10, cat. Y (condensation)</td>
<td>600</td>
</tr>
<tr>
<td>Fluid susceptibility</td>
<td>DO-160E, Section 11, cat. F</td>
<td>1700</td>
</tr>
<tr>
<td>Sand and dust</td>
<td>DO-160E, Section 12, cat. X (cat X means n/a)</td>
<td>n/a</td>
</tr>
<tr>
<td>Fungus resistance</td>
<td>DO-160E, Section 13, cat. F</td>
<td>2000</td>
</tr>
<tr>
<td>Salt spray</td>
<td>DO-160E, Section 14, cat. X</td>
<td>n/a</td>
</tr>
<tr>
<td>Fire, Flammability</td>
<td>DO-160E, Section 26, Category C</td>
<td>2000</td>
</tr>
<tr>
<td>Magnetic effect</td>
<td>DO-160E, Section 15, cat. Z</td>
<td>1000</td>
</tr>
<tr>
<td>Power supply</td>
<td>DO-160E, Section 16, cat. [BZ]</td>
<td>3000</td>
</tr>
<tr>
<td>Voltage spike</td>
<td>DO-160E, Section 17, cat. A</td>
<td>1000</td>
</tr>
<tr>
<td>Audio frequency conducted susceptibility</td>
<td>DO-160E, Section 18, cat. Z</td>
<td>1000</td>
</tr>
<tr>
<td>Induced signal susceptibility</td>
<td>DO-160E, Section 19, cat. Z</td>
<td>2000</td>
</tr>
<tr>
<td>Conducted susceptibility</td>
<td>DO-160E, Section 20, cat. W</td>
<td>3000</td>
</tr>
<tr>
<td>Radiated susceptibility</td>
<td>DO-160E, Section 20, cat. G 100 to 200 V/m SW/ICW and HIRF 700 to 3500 V/m PM</td>
<td>6000</td>
</tr>
<tr>
<td>Conducted emission of radio frequency energy</td>
<td>DO-160E, Section 21, cat. H</td>
<td>1000</td>
</tr>
<tr>
<td>Radiated emission of radio frequency energy</td>
<td>DO-160E, Section 21, cat. H</td>
<td>2000</td>
</tr>
<tr>
<td>Lightning induced transient susceptibility</td>
<td>DO-160E, Section 22 cat A3G33J33</td>
<td>6000</td>
</tr>
<tr>
<td>Lightning direct effects</td>
<td>DO-160E, Section 23, cat. X</td>
<td>n/a</td>
</tr>
<tr>
<td>Icing</td>
<td>DO-160E, Section 24, cat. X</td>
<td>n/a</td>
</tr>
<tr>
<td>Electrostatic discharge</td>
<td>DO-160E, Section 25, cat. A</td>
<td>2000</td>
</tr>
<tr>
<td>Bonding</td>
<td>MIL-STD 464, parag. 5</td>
<td>1000</td>
</tr>
<tr>
<td>Manufacturer extra effort</td>
<td>(person.day)</td>
<td></td>
</tr>
<tr>
<td>Planning, follow-up, investigations, result analysis</td>
<td></td>
<td>140</td>
</tr>
</tbody>
</table>

| Table 3: DO-160E Qualification Costs |

Source: Rockwell Collins
1/29/2009
Role Of Testing

- **Mostly for Validation:**
  - Unit test – does the unit behave as it should?
  - Acceptance test – if customer accepts product, that validates system is OK

- **Certification**
  - For narrow certification, can test a specific property
    - FCC certification that system does not emit too much RF interference
  - For broader certification, may need tests to give credibility to analyses
    - “Wing snap” test on Boeing 777 was used to demonstrate stress model accuracy
    - For X-by-Wire, might need tests to demonstrate models represent actual vehicle
Run-Time Instrumentation

- Related idea is to perform some “tests” all the time
  - Even in production units!
  - Everyday system usage forms the “workload”
  - Use a data recorder to catch and report problems for later analysis

- Selected run-time “test” techniques
  - Log actions and analyze logs
  - Assertions
    - e.g., `#assert RPM < RedLineLimit`
    - Doesn’t enforce this – just checks for when it happens
    - Throws an exception if assertion fails at run-time; good for monitoring invariants
  - Monitor system resources, e.g., memory exhaustion
  - Log all exceptions that occur
  - Detect loss of control loop closure
    - Commanded position too far from actual position for an actuator
Error Logs

◆ Keep a run-time log of errors you encounter

◆ Helps detect bugs that escape into fielded products
  • A robustly designed system will hide many bugs from the user…
    … so how to you know problems are happening?
    – For example, watchdog timer resets
    – For example, running control loops fast to tolerate occasional missed deadline
  • Permits early detection of problems that haven’t been seen by customer
    – If a run-time error occurs, something is wrong with your design
  • What to log: system resets, run-time errors, assertion violations, hardware failures, non-computer failures (problems with the plant), operating conditions, time stamps

◆ Protects software developers from blame
  • “Product is acting weird; must be software”…
    … “Our error logs say it is a hardware problem; go harass them instead”
X-by-Wire Fault Injection

◆ Assume that the safety case’s fault hypothesis is:
  “Continues to operate despite an arbitrary single point fault”
  • Then, it makes sense to test “arbitrary” faults
  • Hardware or software-based fault injection makes sense

◆ Potential approaches to X-by-Wire fault injection:
  • Test software that corrupts bits in memory
    – Used successfully in many areas
  • Radiation chamber
    – Used successfully to find problems with TTP
  • Network message fault injection
    – Corrupt or drop messages on network
  • Pin-level fault injection
    – Disturb electrical signals on circuit boards
Analysis

◆ Examination of software & documentation
  • No actual execution of real software
  • Very effective at finding defects in requirements, design, and software

◆ Includes varying levels of tool / human involvement
  • Ranges from complete static analysis by a compiler-like tool…
  • … to humans sitting in a conference room looking at requirements documents

◆ Primary techniques we’ll discuss:
  • Traceability
  • Reviews
  • Static analysis
  • Model checking

  • Safety analysis (FTA/FMEA/etc.) –discussed in separate lecture
Refresher On Design Reviews

◆ **Design reviews are the most cost-effective way of preventing defects**
  • Think of it as V&V during design instead of after the fact

◆ **Simple version:**
  • Explain your software to someone else, going through it line by line
    – Explaining it out loud to yourself is helpful, but not good enough
    – Doing it via e-mail generally isn’t good enough –
      too easy to sweep things under rug or miss subtleties

◆ **More industrial-strength design reviews**
  • Get a book on how to run design reviews
  • Convene a set of people do to a review in a fixed length of time
  • Have people study the code before the review; assign roles to reviewers
  • Have the presenter go through it and answer questions
  • Take corrective action; iterate reviews if necessary
  • Part of this is knowing what to review (checklist is recommended reading);
    part of it is having someone who knows how to run an effective review
Static Checking & Compiler Warnings

• Static analysis looks at design or code to find problems
  • E.g., look at statechart for states not connected to any other states
  • E.g., look at software for “dead code” – code that can’t be reached by any possible execution path
  • Can be done manually, but better to use tools if available

• Example static analysis approaches:
  • “Lint” / C compiler warning messages (and MISRA C style checkers)
    – Questionable syntax
    – Type checking errors
    – Bad practices
  • Tools to compute McCabe Cyclomatic Complexity
    – Simplistically, Cyclomatic Complexity is number of branches in a code module
    – High complexity means code is more failure prone and more difficult to test
  • More complex tools, such as finding possible memory leaks and unhandled exceptions
  • Always leave warnings turned on and ensure code compiles “clean”
    – This is basically a “free” design review – why would you ignore it???
2012 Coverity scan of open source software results:

- **Sample size:** 68 million lines of open source code
  - Control flow issues: 3,464 errors
  - Null pointer dereferences: 2,724
  - Resource leaks: 2,544
  - Integer handling issues: 2,512
  - Memory – corruptions: 2,264
  - Memory – illegal accesses: 1,693
  - Error handling issues: 1,432
  - Uninitialized variables: 1,374
  - Uninitialized members: 918

- **Notes:**
  - Warning density 0.69 per 1,000 lines of code
  - Most open source tends to be non-critical code
  - Many of these projects have previously fixed bugs from previous scans

Model Checking

- Model checking is a formal method for verifying finite-state concurrent systems

- Intuitive explanation:
  - Start with a model of a system. Might be something like a statechart.
  - State an invariant that should apply:
    - E.g., “All network nodes eventually belong to a single group after a single error”
    - E.g., “Motor will not be commanded to run if any elevator doors are open”
  - Run a model checker, which explores all possible transitions through statechart
    - There are, in general, many transitions.
  - Model checker says one of two things:
    1. “I’ve looked at all possible execution paths, and what you say is guaranteed true”
    2. “I found a counter-example: here it is…”

(For more explanation, see: http://www.cs.cmu.edu/~modelcheck/tour.htm)
Applying Model Checking

- **Model checking is very good for proving pieces of systems correct**
  - Complexity is exponential with number of states
    - So it doesn’t work with arbitrarily large systems; but technology improves yearly
  - OK for aspects of network protocols and small pieces of software

- **But, there are some cautions:**
  - Tests a model of design, not actual code. Software defects can still occur. Models might have errors, etc.
  - Requires specialized skills; not accessible to everyday engineers yet.
  - Model has underlying assumptions!
    - Assumptions are usually not true in all cases
    - Arguing that an assumption is “reasonable” is insufficient for $10^{-9}$ failure rates!
  - Scalability is always an issue – can’t model check a whole car

  - The tricky part is knowing what properties to check!
Some Certifying Authority says that it is “good enough”

- Certification of individuals – licensed PE
- Certification of organizations – ISO 9000; CMM Level 3
- Certification of tools/methods – certified Ada compiler
- Certified systems or products – UL-listed

May be process- or product-based

- UL labs – based on standardized tests of products
- ISO 9000 – audit of process
- … and lots of places in between

Certification may not be a warranty

- Warranty gives legal remedies; certification means it is up to some standard level of “goodness”
- Certification simply places the reputation of the certifier at stake
Example: FAA Software Certification

- **Based on RTCA/DO-178B**
  - Demonstrate that it satisfies requirements
  - Demonstrate there are no errors leading to catastrophic failure
  - (Newer version RTCS/DO-178C is recently out)

- **Verification:**
  - HW/SW integration testing
  - SW integration testing
  - Low-level testing
  - Requirements-based test coverage analysis
  - Structural coverage analysis

- **Alternate verification methods**
  - Formal methods
  - Exhaustive input testing
Example: UL 1998 for Software Components

- **Consumer electronics certification addresses software**
  - For software that replaces functions that previously had hardware protection
  - They want to see the software! Testing alone just isn’t good enough

- **Requirements:**
  - Design for safety
  - Verification, Validation & Test
  - Change management
  - Software Risk Analysis

- **Risk traceability matrix**
  - FMEA-like table

- **Certification components:**
  - Electrical safety reviews + tests
  - Environmental stress tests
  - Software review – source code & some process documents
1994 Pentium FDIV bug:
• “We’ll replace the chip if you can prove the problem affects you”
• Eventually replaced chip for everyone who asked at cost of ~$500M
Hardware Correctness

- **Hardware testing is more manufacturing-centric**
  - Scan approaches
    - Scan paths to test flip-flops
    - Boundary scan to test chip-to-chip interconnects
  - Automatic test generation

- **But, what if the design is incorrect?**
  - The Pentium FDIV bug was a rude awakening
    - Math error in floating point division that affected only a few input values
    - Poor public relations resulted in demand for replacement chips ➔ cost ~$500M
    - But almost every CPU has bugs in it somewhere!
  - Isolate subsystems and test in isolation
  - Incorrect hardware design in many ways “feels” like a software problem…
  - And, in Jan 2011 … Intel found a bug with the Cougar Point support chip
    - Estimated $700M total cost
Software Correctness

◆ **This is a big can of worms**
  - In general, we can’t prove software is correct (i.e., exactly meets the spec.)
  - Even if we can prove it’s correct, we don’t know if the specification is correct
  - So what we do is also include process (lectures on that coming up)

◆ **Software reliability – how many defects when it ships?**
  - Can be inferred by tracking bug detection rates (ship it when you stop finding bugs in testing)
  - Can be improved by better process
  - In general, current state of knowledge is:
    “keep testing until it takes a long time to find the next defect, then ship”

◆ **Software “field reliability” – does it fail in the field?**
  - Difficult problem; not a lot to say about this yet except that it is an issue
  - Components to software field reliability
    – Exposing design defects due to randomly occurring unusual events
    – Failures due to “code rot” and “resource leaks”
Configuration Management

◆ **Make sure that the hardware and software is actually the right stuff**
  • For example, compute a checksum or secure hash of the binary image
  • Make sure before you ship that you are shipping the right software version
  • Have a formal build process to make sure you ship a clean build

◆ **How can this go wrong?**
  • Someone leaves debug code in the final build
    – Watchdog timer accidentally turned off from single-step debugging
    – Back door factory access code left in (security problem)
  • Someone compiles with wrong version of libraries, source code, etc.
  • A virus gets into the build system and infects the built image …

◆ **Applies to hardware as well**
  • Want to make sure you know version and source for critical system components

Airbus A-380 bolt with part tracking information.
Size: 2 cm x 1 cm
What V&V Approaches Are We Using?

- For the course elevator project, list V&V techniques:
Challenge: Ultradependability

- Ultradependable systems “never” fail
  - But if they never fail, how can you know what the failure rate really is?

- Can you test for ultradependability?
  1. How many tests to check all possible behaviors for this function:
     ```
     int32 MyProc(int32 A, int32 B, int32 C)
     ```
     - (who remembers this from last lecture?)
  
  2. How long do you have to test to verify MTBF of $10^{-9}$/hr?
     (This is a typical aircraft failure rate target)

     - Need to test longer than $10^9$ hours and even then you didn’t test enough
     - In fact, need to test between $3^*$ and $10^*$ MTBF to verify MTBF
       - $10^9$ hours = 1,141,000 years of testing
Ultradependability Approaches

- **Good process and lots of V&V**
  - Use good design methodology to reduce design defect rate
  - Use proven, “mature” components
    - But, be careful not to expose hidden limitations with new conditions
  - Test a large number of systems for a long time
    - Need completely failure-free operation during testing – even one failure can be too many for ultradependability
    - There is NO SUCH THING as a one-time failure… there are just situations waiting to re-occur in a different context
  - Use formal methods on tractable, high-risk pieces
  - Use fault injection to assess resiliency to problems that do happen

- **This is the approach taken by safety standards**
And Now, The Problem Gets Harder

◆ **Embedded Internet** –
  - Can someone hack into your car?
    …into your house?
    …into your digital wallet?
    …into your medication pump?
    …into your pacemaker?
    …into your train?

  - Security is now becoming part of the validation/verification/certification picture
    - Static checkers are the first line of defense, but much more is required
    - Penetration testing helps, but much more is required.
Review

◆ The Big Problem
  • We need to ensure systems will really work, and we’re on a tight deadline
  • BUT, there are proven techniques that can help!

◆ Approaches:
  • Design reviews to ensure designs are good before implementation
  • Verification: making sure each design step does what it was supposed to do
  • Validation: making sure the end result satisfies requirements
  • Certification: a written guarantee that a system is acceptable for operational use

◆ In most embedded system companies, testing is the only real V&V
  • BUT, testing isn’t good enough for high-dependability / safe systems!
  • Need as many V&V techniques as possible
  • Ultimately, need a dependability case or safety case to be sure things are OK

  • Later lectures will describe more rigorous processes for critical systems