Verification and Test with Model-Based Design

Flight Software Workshop 2015

Jay Abraham
The software development process

1. Requirements
   - Develop, iterate and specify requirements

2. Design
   - Create high level and low level design components

3. Coding
   - Convert design to source code (C/C++ software)

4. Integration
   - Integrate software with complete system, commence system level tests
Phase where errors are found

Errors are typically found late in the process!

Latent errors may remain in the software!

Source: How good is the software: A review of defect prevention techniques; Brad Clark, David Zubrow, Carnegie Mellon Software Engineering Institute; Software Engineering Symposium 2001
Model-Based Design

- RESEARCH
  - DESIGN
    - Environmental Models
    - Mechanical
    - Electrical
    - Control Algorithms
    - Supervisory Logic

- REQUIREMENTS

- IMPLEMENTATION (Coding)
  - VHDL
  - Structured Text

- Integration (Test)

Models that describe your design
- Executable specification
- Evaluate and test in simulation

Start finding problems early
- Before getting to code
- Before hardware is ready

Continuous Verification and Validation

Requirements → Design → Coding → Integration

- Continuous Verification and Validation

- Research:
  - Environmental Models
  - Mechanical
  - Electrical
  - Control Algorithms
  - Supervisory Logic

- Requirements

- Implementation (Coding):
  - VHDL
  - Structured Text

- Integration (Test)
Modeling and simulation with Model-Based Design

Reference: http://www.flightgear.org/
Modeling and simulation with Model-Based Design

Reference: http://www.flightgear.org/
Simulation and modeling with Model-Based Design

- Now that you are done simulating, are you done?
- Can you generate code for your controller and go directly to hardware?
- Could there be other bugs in your design that could be detected earlier?
- Could there be hidden bugs lurking in your design?
Requirement errors and their consequences

- Missing or incomplete requirements
  - Design may not meet customer needs

- Design lacks sufficient requirements
  - Design may not work as expected

- Inconsistent requirements
  - Design may exhibit unintended behavior
Finding and fixing *requirement errors* early

- Bi-directionally trace requirements
  - From requirements to design and visa-versa
  - Check that all requirements map to design components

- Confirm expected behavior with simulation-based tests
  - Execute simulation-based tests that map to requirements
  - Identify unintended behavior or deviations

- Complement testing with formal methods
  - Prove that design meets safety requirements
  - Use counter examples to debug design
Confirm requirements map to design
Produce requirements trace reports

- **Identify missing requirements**!
- Design components that do not map to a requirement
- Indicative of incorrect design, or misinterpreting requirements
- Avoid finding unintended behaviors late in project phases

Tool: Simulink Verification and Validation
Finding and fixing *requirement errors* early

- **Bi-directionally trace requirements**
  - From requirements to design and visa-versa
  - Check that all requirements map to design components

- **Confirm expected behavior with simulation-based tests**
  - Execute simulation-based tests that map to requirements
  - Identify unintended behavior or deviations

- **Complement testing with formal methods**
  - Prove that design meets safety requirements
  - Use counter examples to debug design
Execute functional tests in simulation

Time series test vectors

Pass / fail results

Tool: Simulink Test
Execute functional tests in simulation
Sequence based tests and assessments

Requirement → In the event of an failure condition of the position sensor or low hydraulic pressure, the Fault Detection Isolation and Recovery (FDIR) application shall select the primary actuator to isolated mode.
Finding and fixing *requirement errors* early

- Bi-directionally trace requirements
  - From requirements to design and visa-versa
  - Check that all requirements map to design components

- Confirm expected behavior with simulation-based tests
  - Execute simulation-based tests that map to requirements
  - Identify unintended behavior or deviations

- Complement testing with formal methods
  - Prove that design meets safety requirements
  - Use counter examples to debug design
Complement testing with formal methods

Graphical representation of safety requirements for thrust reverser

If average airspeed > 150 knots, deploy cannot be true

If two WOW sensors are false, deploy cannot be true

If either wheelspeed sensor < 10 knots, deploy cannot be true

Tool: Simulink Design Verifier
Counter example test generation

Tool: Simulink Design Verifier
Implementation errors and their consequences

- Example implementation errors
  - Dead logic or unreachable states
  - Overflow, divide by zero, other mathematical errors

- Consequences
  - Incorrect operation when presented with abnormal inputs
  - Design may fail catastrophically (design is not robust)
Finding and fixing *implementation errors* early

- With formal methods, prove absence of
  - Dead logic or unreachable states (with model checking)
  - Overflow, divide by zero, out of bound arrays (with abstract interpretation)
Confirm design is free of implementation errors
Prove absence of dead logic, unreachable states

Tool: Simulink Design Verifier
Confirm design is free of implementation errors
Prove absence of overflow, divide by zero, and other design errors

Overflow proven *not* to occur (colored GREEN)

Overflow and divide by zero proven *not* to occur (colored GREEN)

Overflow proven to occur (colored RED)

Tool: Simulink Design Verifier
Finding and fixing *implementation errors* early

- Test completeness of design
  - Simulation-based tests to confirm design testability
  - All execution paths and conditions have been exercised
  - Use coverage metrics to confirm design has been fully tested

User defined input vectors → **System Under Test** → Coverage Report
Confirm design is fully tested

Tool: Simulink Verification and Validation, Simulink Test
Generate additional tests with formal methods

- Top off functional tests with generated tests
  - Use model checking to explore state space
  - Generate missing test-cases to execute all paths
  - Tests generated for decision, condition, MC/DC, table

Tool: Simulink Design Verifier, Simulink Test
Consistency issues and their consequences

- Design fails to meet standards compliance
  - Failure to certify development process and/or software

- Design model and code do not match
  - Functional behavior is different (e.g. timing, logic, etc.)
  - Differences between floating point simulation vs. fixed point
Addressing *consistency* issues

- Check design compliance to standards
  - E.g. do input signals specify min/max?
  - Certification standard: ISO 26262, DO-178C
  - Industry standards: MISRA®, MAAB

- Equivalence testing of model and code
  - Software In the Loop (SIL): test that *code compiled on host computer* matches simulation results
  - Processor In the Loop (PIL): test that *code executing on target processor* matches simulation results
Check model complies with standards

Tool: Simulink Verification and Validation
Confirm design and code consistency

Execute all tests to check equivalence of model to code

Normal mode simulation model

Test Vectors

Code generated from model, compiled on host

Compare

Tool: Simulink Test
Integrated code concerns

- Code generation automates coding process
  - But generated code will be integrated with other code
  - Examples: driver code, RTOS services, etc.

- Deployed code
  - Consists of handwritten and generated code
  - Important to consider the interfaces to avoid functional and robustness problems

- Potential issues to consider
  - Runtime errors; non initialized variables, bad pointers, etc.
Finding *runtime errors* before production

- Formal methods based static code analysis
  - Gain insight into the runtime behavior of the software
  - Verify code without executing the software or running tests

- Confirm integrated code is robust
  - Prove software free of critical bugs (with abstract interpretation)
  - Find critical runtime errors
Verify robustness of integrated code

Formal methods with Abstract Interpretation can prove:

\[ c = a + b; \quad \text{will never overflow} \]

\[ j = \text{arr}[i]; \quad i \text{ will always be within array bounds} \]

\[ \text{*ptr} = 12; \quad \text{will never be an illegal dereference} \]

\[ w = x / (y + z); \quad y \text{ never equal to } -z \text{ or visa versa (divide by zero)} \]

And many more …
static void pointer_arithmetic (void) {
    int array[100];
    int *p = array;
    int i;

    for (i = 0; i < 100; i++) {
        *p = 0;
        p ++;
    }

    if (get_bus_status () > 0) {
        if (get_oil_pressure () > 0) {
            *p = 5;
        } else {
            i ++;
        }
    }

    i = get_bus_status ();

    if (i >= 0) {
        *(p - i) = 10;
    }
}

Verify robustness of integrated code
Polyspace formal methods static analysis

Green: reliable
safe pointer access

Red: faulty
out of bounds error

Gray: dead
unreachable code

Orange: unproven
may be unsafe for some conditions

Purple: violation
code rules

Tool: Polyspace Code Prover
Summary of V&V with Model-Based Design

- Confirm requirements map to design
- Execute functional tests, perform formal verification, generate tests
- With testing, confirm model and design are consistent
- Verify robustness of integrated code