Simulink Verification and Validation

Mark Walker
MathWorks
7th October 2014
When to Stop?

- A perfectly tested design would never be released
  - Time spent on V&V is finite
  - May be small or large

- How can I make the best use of this time?

- Early
- Effective
Early V&V

- The earlier a defect is discovered, the cheaper it is to fix

Effective V&V

- My design is well tested because I have run lots of test cases
  - The good test cases are a small subset of the possible ones
- Development processes, such as DO-178C, deliver reliability
  - Isn't this expensive to do?
  - Bugs are expensive on all projects
- Simulink Verification and Validation helps automate your V&V
  - Automated techniques are efficient, not expensive to apply
- They are accessible to all projects
  …not just people making planes

- How can V&V be applied effectively to models
3 Key DO-178 Concepts

- Requirements traceability
- Testing or inspecting a design in the most important places
- Using analysis to show a design is testable
DO-178 Principle: Requirements Traceability

- Requirements are shown to trace completely
- A complete trace between higher level designs and lower level implementation is independent evidence.
- Every part of a design or implementation must trace
  - Granularity of this traceability can vary
- “A provenance for every line of code”
- Simulink Verification and Validation helps you establish, report on and measure this linkage
Requirements Linking

- When using models, traceability = linking
- Requirement:
  - Increment a clock function by 1ms
  - Must handle years, months, days, hours, seconds and milliseconds
  - Does not need to handle leap seconds (non-requirements)

- Show that the implementation is complete and error free
- MATLAB’s addtodate can be used as a reference
- Alternatively, we would be reviewing results for correct answer.
- <Demo: addtodate_early_example>
Design Evolution

- **System Requirements Allocated to Software**
  - Add 1 millisecond to the time and return the result, matching the behaviour of MATLAB’s `addtodate` function.
  - The inputs and outputs will be in calendar form, i.e. `yy:mm:dd:hr:ss:ms`
  - The resolution in each field shall be 1 unit, i.e. 1 year or 1 millisecond.
  - Invalid dates, such as 2014:09:31:24:60:999 shall be rejected by the function.
  - The function shall always return a valid calendar form.

- **Additional Requirements**
  - The function will return all zeros if presented with an invalid date.
  - January is represented by `mm = 1`.
  - The first day of the month is represented by `dd = 1`.
  - Midnight is represented by `hr = 0`.
  - Year zero is a leap year.
  - Years 9999 and below are valid.
- `<demo: add_to_date_early_example - Establishing links>`
Q: What should happen if the input is invalid?

A: Derived requirement
The function will return all zeros if presented with an invalid date.
- <demo: Model requirements highlight and traceability report>
Reporting Traceability

- Generate reports from your traceability
  - Identify gaps
  - Quickly inspect for correct linkage

Table 3.2. Requirements Traceability Data for code in add_to_date/MATLAB Function

<table>
<thead>
<tr>
<th>Linked Code</th>
<th>Requirements Traceability Data</th>
</tr>
</thead>
</table>
| `if (timsec.milliseconds > uint16(999)) || ...` | "Invalid dates, such as 1/2014:09:31 24:60:999 shall be rejected by the function."
| `timsec.seconds > uint8(59) || ...` | Requirements.doc at "Simulink_requirement_item_7"
| `timsec.minutes > uint8(59) || ...` | "Only years below 12000 are valid."
| `timsec.hours > uint8(23) || ...` | Requirements.doc at "Simulink_requirement_item_8"
| `timsec.days < uint8(31) || ...` | "The function will return all zeros if presented with an invalid date."
| `timsec.days > uint8(31) || ...` | Requirements.doc at "Simulink_requirement_item_7"
| `timsec.months < uint8(13) || ...` | Requirements.doc at "Simulink_requirement_item_7"
| `timsec.months > uint8(12) || ...` | "This limit is fixed by our reference behaviour"
DO-178 Principle: Testing in the Right Place

- Every possible input:
  - Total word width = 72
  - Input combinations = $2^{72} = 2 \times 10^{21}$
  - Testing all at 1000/second = 150 billion years

- Much of the input space is invalid
  - Test out of range behaviour once and focus on in-range only
  - Input combinations reduced to $3.2 \times 10^{14}$ (approx: $10000 \times 365.25 \times 24 \times 60 \times 60 \times 1000$)
  - Testing time reduced to 10,000 years
Model Coverage

- Most of the time this function is very boring
- 99.9% of valid inputs
  - Just add 1 to the millisecond value
- 0.1% of valid inputs
  - Add 1 to the millisecond value and 1 to the second value
- 0.00167% of valid inputs
  - Add 1 to the millisecond, second and minute values
- ...
- As with out of range, only need to test these once
- How do I find the interesting values?
Model Coverage

- Definitions of coverage:
  - Cyclomatic Complexity
  - Condition Coverage (CC)
  - Decision Coverage (DC)
  - Lookup Table Coverage
  - Modified Condition/Decision Coverage (MCDC)
  - Relational Boundary Coverage
  - Saturate on Integer Overflow Coverage
  - Signal Range Coverage
  - Signal Size Coverage
  - Simulink Design Verifier Coverage
Model Coverage

- Definitions of coverage:
  - Cyclomatic Complexity
  - Condition Coverage (CC)
  - Decision Coverage (DC)
  - Lookup Table Coverage
  - Modified Condition/Decision Coverage (MCDC)
  - **Relational Boundary Coverage – New in R2014b**
  - Saturate on Integer Overflow Coverage
  - Signal Range Coverage
  - Signal Size Coverage
  - Simulink Design Verifier Coverage
Boundary Values

- Test for “not a leap year”
  - if \( \text{mod} \left( \text{timevec\_plus\_one\_ms\_years, 4} \right) \geq \text{uint16(1)} \)
    - \( \text{mod} \left( \text{timevec\_plus\_one\_ms\_years, 4} \right) \) could be wrong
    - \( \geq \) could be wrong
    - 1 could be wrong

- Interesting places:
  - 0 \( \geq \) 1 FALSE (Leap year)
  - 1 \( \geq \) 1 TRUE (Not a leap year)
  - 2 \( \geq \) 1 TRUE (Not a leap year)

- We should inspect for correct behaviour at all these conditions
  - 0 \( \geq \) 1 and 1 \( \geq \) 1 sufficient for Condition Coverage

- Tests achieve 100% Boundary Value Coverage
Creating Tests

- Where can “Boundary Coverage Complete” tests come from?
  - Write tests from the requirements
    + Fully independent
    - Needs very complete requirements
  - Write tests from the design
    + Easier to work out the test cases
    - Need an independent way to show requirements are satisfied

- Tests can also be *generated* from requirements (if executable) and designs
- Be careful about independence…
Testing for Expected Behaviour

- addtodate.m is our independent reference
- Model uses a verification subsystem to assert this behaviour
- Reviewing results manually against requirements would also be valid
- Key: Each requirement traces to a test
Testing for Expected Behaviour

- Tests link to requirements in the same way as the design

Chapter 3. System - add_to_date

Chapter 4. System - Verification Subsystem

Table 4.1. add_to_date/Verification Subsystem Requirements Traceability Data

<table>
<thead>
<tr>
<th>Link#</th>
<th>Link Description</th>
<th>Link Target (document name and location ID)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;Add 1 millisecond to the time and return the result, matching the behaviour of MATLAB’s add2date function.&quot;</td>
<td>Requirements: doc at &quot;Simulink_requirement_item_4&quot;</td>
</tr>
</tbody>
</table>
% Call out to MATLAB to test against its reference implementation
verify_in_MATLAB(in, out);

% Test for same answer
if (uint16(r(1)) == out.years) && ...
   (uint8(r(2)) == out.months) && ...
   (uint8(r(3)) == out.days) && ...
   (uint8(r(4)) == out.hours) && ...
   (uint8(r(5)) == out.minutes) && ...
   (uint16(floor(r(6))) == out.seconds) && ...
   (uint16((mod(r(6),1) * 1000)) == out.milliseconds)
   % Match
   result = true;
end
DO-178 Principle: Using analysis to show a design is testable

- Simulink Design Verifier
- What is it?
  - Design Error Detection
  - Test Generation
  - Property Proving

- If something is testable, I get a test case.
- If not, SLDV proves that no test case exists
- The objective is unreachable
- <demo: test gen on add_to_date_early_example>
Generating Tests (Early)

Simulink Design Verifier Results Summary: add_to_date_early_example

- Progress: 85/88
- Satisfied: 85
- Unsatisfiable: 3

Results:
- Highlight analysis results on model
- View detailed analysis report
- Create harness model
- Simulate tests and produce a model coverage report

Data saved in: add_to_date_early_example.slddata.mat
in folder: C:\Projects\expoly\cugu\add_to_date_early_example

3 unsatisfiable
Impossible to test
Generating Tests (Early)

- Fail on Boundary Coverage and MC/DC (redundant logic)

2. MATLAB Function "MATLAB Function"

Parent: /add_to_date_early_example

<table>
<thead>
<tr>
<th>Metric</th>
<th>Coverage (this object)</th>
<th>Coverage (inc. descendants)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclomatic Complexity</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>Condition (C1)</td>
<td>NA</td>
<td>100% (18/18) condition outcomes</td>
</tr>
<tr>
<td>Decision (D1)</td>
<td>NA</td>
<td>100% (13/13) decision outcomes</td>
</tr>
<tr>
<td>MCDC (C1)</td>
<td>NA</td>
<td>75% (7/9) conditions reversed the outcome</td>
</tr>
<tr>
<td>Relational Boundary</td>
<td>NA</td>
<td>95% (37/39) objective outcomes</td>
</tr>
</tbody>
</table>

```matlab
28 % Month cases
29 if (timevec_plus_one_ms.months == uint8(2)) && mod(timevec_plus_one_ms.years,4) > uint8(0) && timevec_plus_one_ms.days > uint8(30)) || ...
30 (timevec_plus_one_ms.months == uint8(2)) && mod(timevec_plus_one_ms.years,4) <= uint8(0) && timevec_plus_one_ms.days > uint8(29)) || ...
31 (timevec_plus_one_ms.days > uint8(30)) && any(timevec_plus_one_ms.months == uint8([4 6 9 11])) || ...
32 (timevec_plus_one_ms.days > uint8(31))
33 timevec_plus_one_ms.days = uint8(1);
34 timevec_plus_one_ms.months = timevec_plus_one_ms.months + uint8(1);
35 end
```
Q: How should I refactor this code to make it more testable?

A: Replace repeated leap year test with a single if.
Generating Tests (Early)

Q: How should I refactor this code to make it more testable?

A: Make the boundaries explicitly testable
Generating Tests (Later)

2 \times 10^{21} \text{ down to 121}
Running the Tests

- We now have:
  - A set of effective tests
  - A means of automatically checking if they pass
- <demo: add_to_date_mid_example test gen and harness run>
Running the Tests

```
out.minutes == uint8(0) && ...
out.seconds == uint8(0) && ...
out.milliseconds == uint16(0)

% Match result
end
end

if ~result
    why
end

% Assertion
assert(result == true, 'Design failed to match MATLAB reference');
```

```
function verify_in_MATLAB(in, out)

% Host only
result = false;

% Create a MATLAB serial date number
n = max([double(in.years),
        double(in.months),
        double(in.days),
        double(in.hours),
        double(in.minutes),
        double(in.seconds) + (double(in.milliseconds) / 1000)], 0);
```

```
in:
years: 0
months: 0
days: 31
hours: 0
minutes: 0
seconds: 0
milliseconds: 0
```
Invalid dates, such as 2014:09:31:24:60:999 shall be rejected by the function.

Feb cases handled for rollover, but not input.
- <demo: add_to_date test gen and coverage run>
Running the Tests

- 100% Boundary Coverage and no assertions

Summary

<table>
<thead>
<tr>
<th>Model Hierarchy/Complexity</th>
<th>D1</th>
<th>CI</th>
<th>MCDC</th>
<th>Relational Boundary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 add_to_date</td>
<td>24</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>2 . . MATLAB Function</td>
<td>23</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>
What did we find?

- Edge cases not handled correctly
  - How: establishing traceability between models and requirements

- The implementation was not fully testable
  - How: test generation with Simulink Design Verifier

- The implementation contained mistakes
  - How: simulating at boundary coverage points and checking for expected behaviour
Summary

- Early: Good V&V helps discover defects earlier
  - Invest in testing right from the start of development

- Effective: Good tool support makes advanced techniques more cost-effective
  - All projects can benefit, not just High Integrity applications