A Unified Approach to Model and Code Verification

12 May 2016

Chuck Olosky
Anthony Abraham

Application Engineering
Application Engineering
Motivation

- Most controls applications are a combination of model-based generated code and hand code
- How do I efficiently test this mix of hand code and generated code?
- MathWorks has tools for testing models and tools for testing code
- Is there a workflow for me to use these tools in a complementary, optimum way?
Agenda

- Static analysis of the model and code before functional testing
- Dynamic, functional testing of the model, s-function and generated code
- Static analysis of the integrated code: hand code, s-function code and generated code
- A unified, complementary model and code verification workflow to continually increase design confidence
Case Study: Cruise Control Application

Objective: set cruise control target speed and pedal position based on driver & vehicle inputs

Cruise Control Application (C code)
- Hand code components
- Model-based Stateflow component
- Model-based S-function component

65 mph
Case Study: Cruise Control Architecture

**Cruise Control Application**

- **Function Scheduler**
  - **Read Inputs**
  - **Fault Logging**
  - **Target Speed Control Module**
  - **Pedal Command Control Module**
  - **Write Outputs**

**System Inputs**
- Cruise Power
- Brake
- Vehicle Speed
- Coast/Set
- Accel/Resume

**System Outputs**
- Target Speed
- Engaged
- Pedal Position

---

- Hand Code
- MBD Gen Code
- S-function Code
Case Study: Roles & Workflow

- **MBD Controls Guy: Chuck**
  - Develops modules using Simulink models
  - Integrates C code with models via s-functions
  - Generates the code
  - Relies on model-based testing methods

- **Integration & Build Guy: Anthony**
  - Develop C code modules by hand
  - Integrates hand code and generated code
  - Creates the ECU build
  - Relies on the HiL bench for testing
Case Study: Deliver First Production Release to Customer

To deliver our first production release we will need the following new features/changes:

• Move signals/cals from floats to integers in Target Speed Module
• Include customer lookup table code in Pedal Command to support calibration
• Demonstrate generated code is MISRA compliant
• Remove unused fault record
• Migrate the code to run on customer’s ECU (14-bit to 12-bit ADC)

In addition to the changes we will need to provide functional test results for the model-based modules and the integrated code.
Model-based Design Tasks

First let’s focus on the model-based design tasks and what checks are available:

- Convert signals/cals from floats to integers in Target Speed Module
- Include the customer lookup table in the Pedal Cmd to support calibration
- Demonstrate generated code is MISRA compliant

Our approach will be to do checks before functional testing, early in the development to minimize re-work.
Floats to Integers: Checking the Model for Design Errors

Simulink Design Verifier identifies model design errors that may result in “dead logic” that would prevent successful functional testing.
Root Cause Analysis/Fix of Dead Logic

- Dead logic due to “uint8” operation on incdec/holdrate*10
- Fix change the order of operation 10*incdec/holdrate
Model-based Design Tasks

First let’s focus on the model-based design tasks and what checks are available:

- Convert signals/cals from floats to integers in Target Speed Module
- Include the customer lookup table in the Pedal Cmd to support calibration
- Demonstrate generated code is MISRA compliant

Our approach will be to do checks before functional testing, early in the development to minimize re-work
Customer Lookup Table: Checking the S-Function Code for Runtime Errors

```
2
4
1

S

single y1 = Lookup1D_C(mtB u1,mt8 p1[] single p2[]) PedalCmdLookup_C
```
Root Cause Analysis/Fix of S-Function Run-time Errors

/* Definition for custom storage class: Global */

real32_T PedalCmdY[11] = { 0.0f, 0.5f, 1.0f, 1.0f, 2.0f, 2.0f, 3.0f, 3.0f, 3.0f, 3.0f, 3.0f, 3.0f, 3.0f};

int8_T SpeedDelX[11] = { -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20};

float Lookup1D_C(char u, char const X[], float const Y[])
{
  float y = 0.0f;
  unsigned char index = 0;
  float temp = 0.0f;

  unsigned char mySize = 11;

  if (engaged)
  {
    PedalPos = Lookup1D_C( (int8_T)rtb_Sum, (int8_T*)(&SpeedDelX[0]), (real32_T*)(&PedalCmdY[0]));
  }
  else
  {
    PedalPos = PedalPosRaw;
  }

  while((u == X[index]) && (index < mySize))
  {
    y += Y[index];
    index++;
  }

  while((u >= X[index]) && (index < (mySize-1)))
  {
    index++;
  }

  if (index > 0)
Model-based Design Tasks

First let’s focus on the model-based design tasks and what checks are available:

- Convert signals/cals from floats to integers in Target Speed Module
- Include the customer lookup table in the Pedal Cmd to support calibration
- Demonstrate generated code is MISRA compliant

Our approach will be to do checks before functional testing, early in the development to minimize re-work
Checking Model for MISRA compliance with Model Advisor
Checking Model for MISRA compliance with Model Advisor

Model Advisor - CruiseControl_PS

Check configuration parameters for MISRA C:2012

- By Product
- By Task
- Code Generation Efficiency
- Data Transfer Efficiency
- Frequency Response Estimation
- Managing Data Store Memory Blocks
- Managing Library Links And Variants
- Migrating to Simplified Initialization mode
- Model Metrics
- Model Referencing
- Modeling Guidelines for MISRA C:2012
  - Check configuration parameters for MISRA C:2012
  - Check for blocks not recommended for MISRA C:2012
  - Check for unsupported block names
  - Check usage of Assignment blocks
  - Check for bitwise operations on signed integers
  - Check for recursive function calls
  - Check for equality and inequality operations on floating-point values
  - Check for switch case expressions without a default case

- Modeling Physical Systems
- Modeling Signals and Parameters using Buses
- Modeling Single-Precision Systems
- Modeling Standards for DO-178C/DO-331
- Modeling Standards for EN 50128

Result: Warning

Identify configuration parameters that might impact MISRA C:2012 compliant code generation.

Warning
The model configuration parameters are not set to the recommended values specified in the data file.

<table>
<thead>
<tr>
<th>Status</th>
<th>Parameter</th>
<th>Current Value</th>
<th>Recommended Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warning</td>
<td>Model Verification block enabling (AssertControl)</td>
<td>UseLocalSettings</td>
<td>DisableAll</td>
</tr>
<tr>
<td>D - Warning</td>
<td>Shared code placement</td>
<td>Auto</td>
<td>Shared location</td>
</tr>
<tr>
<td>Warning</td>
<td>GenerateSharedConstants</td>
<td>Prerequisite constraint not met</td>
<td>off</td>
</tr>
</tbody>
</table>

Action
Modify the configuration parameters listed above to the recommended values.

Run This Check
Modify All
Checking Model for MISRA compliance with Model Advisor

- Checks model design and code configuration settings
- Increases likelihood of generating MISRA C:2012 compliant code
Configuring Polyspace from the Model
Launching Polyspace from the Model
Review Bug Finder MISRA results

**MISRA AC AGC 8.10 (Obligatory)**

All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.

Variable 'AccelResSw' should have internal linkage.
Reduce MISRA violations with “Code Placement” setting
Justify other violations by adding annotation
Model-based Design Tasks

First let’s focus on the model-based design tasks and what checks are available:

- Convert signals/cals from floats to integers in Target Speed Module
- Include the customer lookup table in the Pedal Cmd to support calibration
- Demonstrate generated code is MISRA compliant

Our approach will be to do checks before functional testing, early in the development to minimize re-work
Model-based Design Tests

All checks are complete, we will need to provide test results for the model-based modules:

- Functional testing of s-function based Pedal Command module
- Equivalence (model-to-code) testing of the Target Speed module
Functional Testing of Pedal Command (S-Function)

Coverage analysis for the model and the s-function code.

Uncovered Links:

Metric | Coverage
--- | ---
Decision (D1) | 100% (2/2) decision outcomes
Condition (C1) | 75% (3/4) condition outcomes

Conditions analyzed:

<table>
<thead>
<tr>
<th>Description</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u \geq X[index]$</td>
<td>105</td>
<td>21</td>
</tr>
<tr>
<td>$index &lt; (\text{mySize}-1)$</td>
<td>105</td>
<td>0</td>
</tr>
</tbody>
</table>
Model-based Design Tests

All checks are complete, we will need to provide test results for the model-based modules:

- Functional testing of s-function based Pedal Command module
- Equivalence (model-to-code) testing of the Target Speed module
Check the Generated Code for Equivalent Model Behavior

- Integrated SIL mode support for model-to-code equivalence testing
- Coverage report for generated code for a detailed equivalence analysis
Model-based Design Tests

*All checks are complete, we will need to provide test results for the model-based modules:*

- Functional testing of s-function based Pedal Command module
- Equivalence (model-to-code) testing of the Target Speed module
Integrated Code Testing

The hand code design tasks:

- Remove unused fault record
- Migrate the code run on customer’s ECU (14-bit to 12-bit ADC)

The minor hand code changes have been made.

An ECU build was created based on the integration of hand code and generated code.

We now need to provide functional test results for the integrated code on the HiL bench.

Find issues that result from the integration of tested modules from hand code, s-function code and model-based generated code.
Issues Found on HIL Bench…

- The Cruise Control powered off during fault testing
- And, the Target Speed never exceeded 40 mph
Creating a Code Prover project to check the Integrated Code

- Read Inputs
- Write Outputs
- Scheduler
- Fault Logging

Target Speed Control Module

Pedal Command Control Module
Code Integration Check with Polyspace: Non-terminating loop in Hand Code

```c
void fault_log(FAULT_LOG_INFO_T *pFaultInfo)
{
    uint32_t ix;

    uint32_t *pFlt = (uint32_t *)pFaultInfo;

    /* Validate current fault index counter */
    if (FaultRecIndexCntr >= (MAX_FAULT_LOG_INFO_SIZE - 12u))
    {
        FaultRecIndexCntr = 0x0u;
    }

    /* Store the current fault into circular buffer */
    for (ix = 0; ix < 12u; ix++)
    {
        FaultInfoElement[FaultRecIndexCntr + ix] = *pFlt;
        *pFlt = 0x0u;
        pFlt++;
    }

    /* Update the circular buffer fault index counter */
    FaultRecIndexCntr += ix-1;
}
```

- Non-terminating loop in Hand Code
- Fault Logging

Non-terminating loop

The loop is infinite or contains a run-time error. Loop may fail due to a run-time error (maximum number of iterations: 11)

Illegally dereferenced pointer

Warning: pointer may be outside its bounds.
Dereference of local pointer 'pFlt' (pointer to unsignd int 32, size: 32 bits):
Pointer is not null. Points to 4 bytes at offset multiple of 4 in [0 .. 40] in buffer of 40 bytes, so may be outside bounds.
Cause of Cruise Control Powering off during fault testing

```c
void fault_log(FAULT_LOG_INFO_T *pFaultInfo)
{
    uint32_t ix;

    uint32_t *pFlt = (uint32_t *)pFaultInfo;

    /* Validate current fault index counter */
    if (FaultRecIndexCntr >= (MAX_FAULT_LOG_INFO_SIZE - 12u))
    {
        FaultRecIndexCntr = 0x0u;
    }

    /* Store the current fault into circular buffer */
    for (ix = 0; ix < 12u; ix++)
    {
       FaultInfoElement[FaultRecIndexCntr + ix] = *pFlt;
        *pFlt = 0x0u;
        pFlt++;
    }

    /* Update the circular buffer fault index counter */
    FaultRecIndexCntr += ix-1;
}
```

<table>
<thead>
<tr>
<th>ix</th>
<th>pFlt</th>
<th>typedef members</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0x40000000</td>
<td>Expected_Value</td>
<td></td>
</tr>
<tr>
<td>1 0x40000004</td>
<td>Received_Value</td>
<td></td>
</tr>
<tr>
<td>2 0x40000008</td>
<td>Fault_ID</td>
<td></td>
</tr>
<tr>
<td>3 0x4000000c</td>
<td>Fault_Type</td>
<td></td>
</tr>
<tr>
<td>4 0x40000010</td>
<td>Time_mSec</td>
<td></td>
</tr>
<tr>
<td>5 0x40000014</td>
<td>Time_Sec</td>
<td></td>
</tr>
<tr>
<td>6 0x40000018</td>
<td>Time_Min</td>
<td></td>
</tr>
<tr>
<td>7 0x4000001c</td>
<td>Time_Hr</td>
<td></td>
</tr>
<tr>
<td>8 0x40000020</td>
<td>Additional_Flt_Spec01</td>
<td></td>
</tr>
<tr>
<td>9 0x40000024</td>
<td>Additional_Flt_Spec02</td>
<td></td>
</tr>
<tr>
<td>10 0x40000028</td>
<td>CruiseOnOff</td>
<td></td>
</tr>
<tr>
<td>11 0x4000002c</td>
<td>Brake</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CostalSetSw</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AccellResSw</td>
<td></td>
</tr>
<tr>
<td>0x40000030</td>
<td>Speed</td>
<td></td>
</tr>
</tbody>
</table>
Root cause of Cruise Control Powering off
Fix and verify the hand code is free of Runtime Errors

```c
void fault_log(FAULT_LOG_INFO_T *pfaultInfo)
{
    uint32_t ix;
    uint32_t *pFlt = (uint32_t *)pfaultInfo;

    /* Validate current fault index counter */
    if (FaultRecIndexCntr >= (MAX_FAULT_LOG_INFO_SIZE - 12u))
    {
        FaultRecIndexCntr = 0x0u;
    }

    /* Store the current fault into circular buffer */
    for (ix = 0; ix < 10u; ix++)
    {
        FaultInfoElement[(FaultRecIndexCntr + ix) % MAX_FAULT_LOG_INFO_SIZE] = *pFlt;
        *pFlt = 0x0u;
        pFlt++;
    }

    /* Update the circular buffer fault index counter */
    FaultRecIndexCntr += ix-1;
}
```
Vehicle speed signal propagated to “CruiseControl_PS.c” [0 … 40]

```c
} else if (Speed > maxtspeed) {
    /* Transition: '<S1>:114' */
    /* Exit Internal 'ON': '<S1>:54' */
    DW.is_ON = CruiseControl_IN_NO_ACTIVE_CHILD;
    DW.is_CRUISE = CruiseControl_PS_IN_STANDBY;
    /* Entry 'STANDBY': '<S1>:52' */
    engaged = false;
```

Maximum target speed = 90

Unreachable/Dead code
Root Cause for Dead Code: Speed Sensor Input Hand Code

Changing analog-to-digital converter from 14 to 12-bit results in dead code

MASK – accounts for scaling down for new ADC from 14-bit to 12-bit

Overlooked changing CONV_FACTOR for new ADC
Workflow Summary: Complementary Model & Code Verification

Hand Code → (Hand-written) C/C++ code

- Check model early for design errors
- Check MISRA compliance (Mdl Advisor)
- Functional testing (simulation)
- Code coverage (SIL)
- SIL mode support
- Check integrated code for run-time errors

MBD Generated Code

- Textual requirements
- Executable specification
- Model used for production code generation
- Generated C/C++ code
- Object code

Cruise Control Application

- Check s-function code for run-time errors
- Check MISRA compliance (Polyspace)

Code analysis

Compilation and linking
A Complementary Model and Code Verification Process …

✓ Model and code checks before functional testing to minimize rework
✓ Perform functional, dynamic testing with model and code structural analysis with automation, and reuse of test assets
✓ Analyze the code to find issues resulting from the integration of
  o hand code
  o s-function code
  o model-based generated code
✓ Includes formal methods analysis to go beyond functional testing
✓ Enables more, early testing of the model and code
✓ Continual increase in design confidence
Thank You!