MathWorks
AUTOMOTIVE CONFERENCE 2016

9월 2일 - 서울
A Unified Approach to Model and Code Verification

Young Joon Lee
Principal Application Engineer
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?
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
Case Study: Cruise Control Architecture

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

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

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.
Case Study: Workflow

S-function Code

Pedal Command Control Module

Target Speed Control Module

Software requirements → Executable specification → Model used for production code generation → Generated C/C++ code

MBD Generated Code

Architectural design → Unit design → (Manual) implementation

Hand Code

Read Inputs

Fault Logging

Write Outputs

Cruise Control Application

Integrated code → Object code

Compilation and linking
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.
Simulink Design Verifier identifies design errors on the model
These “dead logic” errors would prevent successful functional testing.
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*
Legacy Code Tool imports the hand code into an s-function block and creates a tlc file for code generation.
Customer Lookup Table: Checking the S-Function Code for Runtime Errors

![Diagram of the S-Function code and analysis results]

- **Source code snippet:**
  ```c
  y = Y[0];
  
  else
  
  { 
  while((u > [x[index]]) && (index < (mySize)))
  
  index++;
  ```

- **Analysis results:**
  - **Pointer access out of bounds** (Impact: High)
  - Attempt to dereference pointer at index 11.
  - Valid range: [0 .. 10]

- **Pedal Command Control Module**
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, 1.5F, 2.0F, 2.5F,
                          3.0F, 3.5F, 4.0F, 4.5F, 5.0F } ;
int8_T SpeedDelX[11] = { -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5 } ;

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]),
                            X[index], temp, y);
    }
    else {
    }

    while((u >= X[index]) && (index < mySize))
    {
        if (index > 0)
        {
            index++;
        }
    }
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*

[Image of Model Advisor interface with selected guidelines and checks for MISRA C:2012]
Checking Model for MISRA compliance with Model Advisor

Target Speed Control Module
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

Target Speed Control Module
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.

Polyspace Bug Finder - CruiseControl_PS C:\Demo\VnV\VnV_WkshpResults Summary
- All results

MISRA AGC 18
- 1 Environment 4
- 8 Declarations and definitions 14

Category: Obligatory CruiseControl_PS.c Global
- boolean_T AccelResSw;
- boolean_T Brake;
- boolean_T CoastSetSw;
Reduce MISRA violations with “Code Placement” setting

<table>
<thead>
<tr>
<th>Global data placement (custom storage classes only)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data definition: Data defined in a single separate source file</td>
</tr>
<tr>
<td>Data definition filename: cruise_control_global.c</td>
</tr>
<tr>
<td>Data declaration: Data declared in a single separate header file</td>
</tr>
<tr>
<td>Data declaration filename: cruise_control_global.h</td>
</tr>
<tr>
<td>#include file delimiter: Auto</td>
</tr>
</tbody>
</table>

Global data placement (MPT data objects only)

Signal display level: 10

Code Packaging

File packaging format: Modular
Justify other violations by adding annotation.

8 Declarations and definitions

8.10 All declarations and definitions of objects or functions

Result Details
- Severity: Not a defect
- Status: Justify with annotations

Add justification here

CruiseControl_PSE

29 /* Block states (auto storage) */
30 DW_CruiseControl_PSE T DW;
31
32 /* Real-time model */
33 RT_MODEL_CruiseControl_PSE T M_;
34 RT_MODEL_CruiseControl_PSE T *const M = &M_;
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.
- Unable to achieve full coverage due to “defensive” programming.

Uncovered Links:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decision (D1)</td>
<td>100% (2/2) decision outcomes</td>
</tr>
<tr>
<td>Condition (C1)</td>
<td>75% (3/4) condition outcomes</td>
</tr>
</tbody>
</table>

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; (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
Code Integration Check with Polyspace: Non-terminating loop in Hand Code

SysTick_Handler.c  schedulerExecutive.c  monitorBuiltInTestFaults.c  fault_log.c
SysTick_Handler  schedulerExecutive  monitorBuiltInTestFaults  fault_log

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

! 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)
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;
}
```

### Fault Logging

<table>
<thead>
<tr>
<th>ix</th>
<th>pFlt</th>
<th>typedef members</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x40000000</td>
<td>Expected_Value</td>
</tr>
<tr>
<td>1</td>
<td>0x40000004</td>
<td>Received_Value</td>
</tr>
<tr>
<td>2</td>
<td>0x40000008</td>
<td>Fault_ID</td>
</tr>
<tr>
<td>3</td>
<td>0x40000000c</td>
<td>Fault_Type</td>
</tr>
<tr>
<td>4</td>
<td>0x40000010</td>
<td>Time_mSec</td>
</tr>
<tr>
<td>5</td>
<td>0x40000014</td>
<td>Time_Sec</td>
</tr>
<tr>
<td>6</td>
<td>0x40000018</td>
<td>Time_Min</td>
</tr>
<tr>
<td>7</td>
<td>0x4000001c</td>
<td>Time_Hr</td>
</tr>
<tr>
<td>8</td>
<td>0x40000020</td>
<td>Additional_Flt_Spec01</td>
</tr>
<tr>
<td>9</td>
<td>0x40000024</td>
<td>Additional_Flt_Spec02</td>
</tr>
<tr>
<td>10</td>
<td>0x40000028</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0x4000002c</td>
<td>CruiseOnOff</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Brake</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CostalSetSw</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AccellResSw</td>
</tr>
<tr>
<td>0x40000030</td>
<td>Speed</td>
<td></td>
</tr>
</tbody>
</table>
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

```c
/* Conversion factors of speed */
#define NEW_ECU
#elif NEW_ECU
    #define SPEED_MASK 0xFFF
#else
    #define SPEED_MASK 0x3FFF /* Original design specification */
#endif

/* Scaling for conversion factor for translating sensor input to miles/hr */
#define CONV_FACTOR 0.01
/* FAILS */

#define MAX_AT_RAW_COUNTS_BUFSIZE 10

/* Convert raw counts to speed */
AI_Speed.Speed = ((AI_Speed.Average & SPEED_MASK) * CONV_FACTOR);

/* Updated analog inputs */
MDB_Shared_Data.Speed = AI_Speed.Speed;
```

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**

- Check model early for design errors
- Check MISRA compliance (Mdl Advisor)

- Functional testing (simulation)

- (Hand-written) C/C++ code

- Check integrated code for run-time errors

- Cruise Control Application

- Code coverage (SIL)

- SIL mode support

- Check s-function code for run-time errors

- Check MISRA compliance (Polyspace)
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!