Simulink를 이용한 효율적인 레거시 코드 검증 방안

류성연
Agenda

- Overview to V&V in Model-Based Design
- Legacy code integration using Simulink
- Workflow for legacy code verification
Model-Based Design With Legacy C/C++ Code?

Legacy code verification using Simulink?

MBD with C/C++ code?

Model-Based Design
Why Using Simulink for Legacy Code Testing?
ISO26262 “Road Vehicles - Functional Safety”

- **Functional safety standard** for passenger cars
  - Concerned with avoidance of unreasonable risks due to hazards caused by E/E systems
  - Recommends tool certification, but offers little guidance

- Serves as an umbrella standard for industry specific adaptions including:
  - ISO 26262 - Automotive
  - EN 50128 - Rail
  - IEC 62304 - Medical
  - IEC 61511 - Process Control
Software Development Workflow for Embedded Applications

- **Requirements Authoring**
  - Textual Requirements
  - Executable Specification
  - Model used for production code generation
  - Generated C/C++ code
  - Object code

- **Modelling**
  - Software architecture and design

- **Code Generation**
  - Handwritten C/C++ code

- **Compilation and Linking**

**Requirements Trace**
- Documentation
- Version Control
- Tool Qualification
Legacy Code Verification Overview

Textual Requirements

Executable Specification

Model used for production code generation

Generated C/C++ code

Object code

Requirements Authoring

Modelling

Code Generation

Compilation and Linking

For Model-Based Design

Software architecture and unit design

S-functions from Handed C/C++ code

Coverage analysis
/
Test case generation

Unit(Integration) testing

Code review and Static analysis

For legacy code development
Agenda

- Overview to V&V in Model-Based Design
- Legacy code integration using Simulink
- Workflow for legacy code verification
How to Import Legacy Code

- Legacy Code Tool
- C Caller Block
- Legacy code integration in Stateflow
What legacy C code integration in Simulink means?

- Legacy Code Tool enables existing C code to be used in Simulink models
How to use Legacy Code Tool?

- General procedure for using Legacy Code Tool

Modeling with S-Function

S-function block

Legacy C code

Legacy Code Tool (LCT)

1. Initialize LCT data structure
2. Populate LCT data structure
3. Generate S-function source file
4. Compile S-function source file
5. Create masked S-Function block

C MEX S-function

int16 y1 = fcn(int16 u1)
Prerequisite to use Legacy Code Tool

- What is wrapper code?
  - Root-level C function having in/output variables for S-Function block’s in/out ports

```c
double doubleit(double u1)
{
    MainInp = u1;
    double_main();
    y1 = MainOutp;
    return y1;
}
```

```c
void doubleit(double u1, double *y1)
{
    MainInp = u1;
    double_main();
    *y1 = MainOutp;
}
```
MATLAB Script to Build and Generate S-Function Block

- **m-script file**: compiling C files and generate a S-Function Block

```matlab
Simulink.importExternalCTypes('ex_myTypes_LCT.h');

def = legacy_code('initialize');

def.SFunctionName = 'sfun_ex_mySrc_LCT';
def.SourceFiles = {'ex_mySrc_LCT.c'};
def.HeaderFiles = {'ex_myTypes_LCT.h'};
def.OutputFcnSpec = 'void myFcn(sigStructType u1[1], paramStructType p1[1], sigStructType y1[1])';
def.IncPaths = {'rtwdemo_lct_src'};
def.SrcPaths = {'rtwdemo_lct_src'};

legacy_code('sfcn_cmex_generate', def);

legacy_code('compile', def);

legacy_code('slblock_generate', def);
```

1. **C files to integrate in Simulink**
2. **S-Function block specification**
3. **Include folders**
4. **Compile and s-function generation**
Generate Simulink Representations from C or C++ Code

- Import external C header file and generate available Simulink data types

```matlab
Simulink.importExternalCTypes('ex_myTypes_LCT.h');
```

Automatically generating to Simulink Bus

Selecting generated Simulink Bus
Issues for Legacy Code Tool

- There are still technical challenges to make S-Function Block

Conventional C code verification
  - Difficult to build test cases
  - Limited input variation
  - Long lead time from development to test
  - Hard work to improve test results

Legacy Code Tool

C code verification with Simulink
  - No unified interfaces to interact with legacy code
  - Hard to build S-Function Block
  - No auto sync with custom C code change
  - Still maintenance problem
Example Issue: Too Many Function Arguments

### Wrapper code

```c
#include "CrsCntrl_Wrapper.h"

void CrsCntrl(boolean_t u1, boolean_t u2, boolean_t u3, boolean_t u4, boolean_t u5, boolean_t u6,
              int16_t u7, uint8_t u8, uint8_t u9, Int32_t u10, Int32_t u11,
              uint8_t *y1, boolean_t *y2, uint8_t *y3, int32_t *y4, int32_t *y5)
{
    // Code...
}
```

### Legacy code

```c
#ifndef CRUISECNTRLR_H
#define CRUISECNTRLR_H

#include "DataTypes.h"
#include "CruiseCtrlrTypes.h"

/* Cruise controller input */
extern int16_t s16BrakeP;
extern boolean_t u3CncSw;
extern boolean_t u1DecSw;
extern boolean_t u1LeftSw;
extern boolean_t u1ResumeSw;
extern boolean_t u1DrnkSw;
extern boolean_t u1StopSw;
extern boolean_t u1EscSw;
extern boolean_t u1AckSw;

define CRUISECNTRLR
#define DEFINE_CRUISECNTRLR

#define Keyon 2
#define ShiftDrive 2
#define BrakeOnThrs# 5
#endif /* CRUISECNTRLR_H */
```

### Script file

```c
#include "CrsCntrl_Wrapper.c"
#include "CruiseCtrlr.c"

// Code...
```

- Too many interface variables
- Nested structure
- Bitfield
- etc.
Maintenance Problem...

Legacy code

```
void CrCtrlMainMain(void)
{
  boolean_t u1UlIncSelLong;
  boolean_t u1UlDecSelLong;

  // setting increase button is pressed for long time
  if (u1IncWw == TRUE)
  {
    u1UlIncSelLong = TRUE;
  }
  else
  {
    u1IncWw = u1IncWw + 1;
    u1UlIncSelLong = FALSE;
  }

  else
  {
    u1IncWw = u1IncWw - 1;
    u1UlIncSelLong = FALSE;
  }

  // setting decrease button is pressed
  if (u1DecWw == TRUE)
  {
    u1UlDecSelLong = TRUE;
  }
  else
  {
    u1DecWw = u1DecWw + 1;
    u1UlDecSelLong = FALSE;
  }
}
```

Wrapper code

```
#include "Control_Draper.h"

void CrCtrlDraperMain(void)
{
  void CrCtrlMainMain(void);
  ...}
```

Modeling

![Simulation Model](image)

Script file

```
def SourceFiles = ['C:\Code\File1.c', 'C:\Code\File2.c'];
def HeaderFiles = ['C:\Code\Header1.h', 'C:\Code\Header2.h'];
def IncludePaths = [(defaultDir, 'C:\Code\IncludePath1'),
  (defaultDir, 'C:\Code\IncludePath2')];
def SearchPaths = [(defaultDir, 'C:\Code\SearchPath1'),
  (defaultDir, 'C:\Code\SearchPath2')];
def Target = 'C:\Code\Target.exe';
def StartFlags = ['void init();'];
def DataFiles = ['C:\Code\Data1.dat', 'C:\Code\Data2.dat'];
def Options = {coverage: true, ...};
debugger = 'C:\Debugger.exe';
```
Introducing C Caller Block

C Caller Block makes it easier to call C Functions in Simulink
→ It works for simulation and Code Generation
Key Features

- Automate the process
  - Define Block Interface
  - Build Simulation MEX
  - Write Codegen TLC
  - Automate
  - Tedious
  - Error prone
  - Hard to maintain

- Synchronize with custom code changes

C/C++ Code
Using C Caller Block

1. Specify Custom Code in the Configuration Parameters

- Custom code is specified on the Configuration Parameters.
  - **The Header file section**: Any code that needs to be inserted into the header file
  - **The Source files section**: List of source files that needs to be compiled
Using C Caller Block

2. Select the function that you want to call
Using C Caller Block

3. Customize the function that you want to call

- Mapping inputs, outputs or parameters to C Caller Block

1) Change argument scope to “Output”

2) (Optional) Override with a better output name

3) Complete the test model with connecting signal ports
Demo: Simple C Caller
Library Workflow

- C Caller block can be configured as a library model
  - Custom Code Settings can be accessed from View Menu → Library Custom Code Settings

- Include custom header file: `#include "opencv_lib.hpp"
- Add custom source file: `opencv_lib.cpp"
Demo: Reusable Library Workflow with OpenCV
Legacy Code Evaluation in Stateflow

- Using legacy code in Stateflow chart

```c
#include "custom_code.h"

double c_multiple = 0.8;

double c_fcn(double in)
{
    return in * c_multiple;
}

void set_c_multiple(double in)
{
    c_multiple = in;
}
```

Step 1: Have C code
Step 2: Put on Config. Set
Step 3: Use in Stateflow
Agenda

- Overview to V&V in Model-Based Design
- Legacy code integration using Simulink
- Workflow for legacy code verification
Legacy Code Verification using Simulink V&V

- Test Cases
- Test harness model
- S-Function or C Caller
- Test case generation
- Code coverage analysis

SIL Coverage Report by Model

- Top Model: cry_controller_harness_SIL
- Complexity: 81%
- Decision: 83%
- Condition: 88%
- Statement: 86%
- Function: 100%
- Function call: 100%
Demo: Legacy C Code Verification
Needs for Test Automation

- Test automation/management
- Code coverage analysis
- Function/Function call coverage
- Report generation
Test Automation with Test Manager

- Test automation
- Test case creation from template
- Customization
- View, share, report results
Static Code Analysis

- Run time error / MISRA rule check
- Polyspace report from Simulink
- Reducing Polyspace set-up efforts
Key Takeaways

- **Requirements Authoring**
  - Textual Requirements
  - Executable Specification
  - Modelling
  - Model used for production code generation
  - Generated C/C++ code
  - Object code

- **Modelling**
  - Software architecture and unit design
  - C Caller for legacy code

- **Code Generation**
  - Code Generation
  - Compilation and Linking

- **Review and Static analysis**
  - Coverage analysis / Test case generation

- **Unit/Integration) testing**
  - SL Requirement
  - Simulink Requirement
  - Simulink Test
  - Simulink Coverage
  - Simulink Design Verifier

- **Polyspace**
  - Simulink
  - Simulink Coder
  - Embedded Coder

- **Simulink**
  - Simulink Requirement
  - Simulink Test
  - Simulink Coverage

- **SL Requirement**
  - SL Requirement

- **Simulink Test**
  - Simulink Test

- **Simulink Coverage**
  - Simulink Coverage

- **Simulink Design Verifier**
  - Simulink Design Verifier
Thank You!