A Unified Approach to Model and Code Verification

12 May 2016

Chuck Olosky, Application Engineering
Anthony Abrham, Application Engineering

Motivation

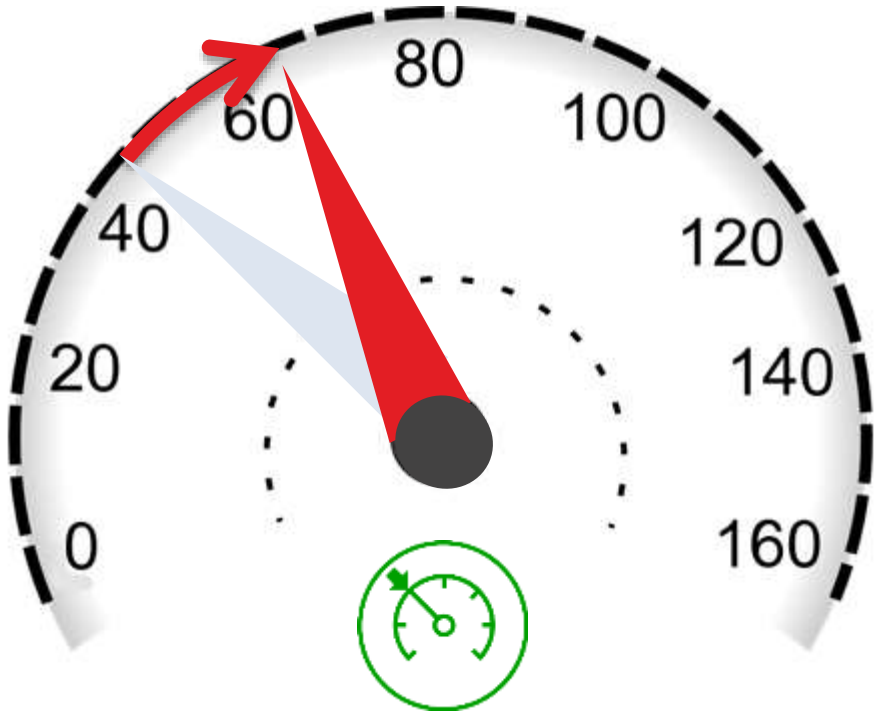
- Most controls applications are a combination of model-based generated code and hand code
- How do I efficiently test this mix of hand code and generated code?
- MathWorks has tools for testing models and tools for testing code
- Is there a workflow for me to use these tools in a complementary, optimum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

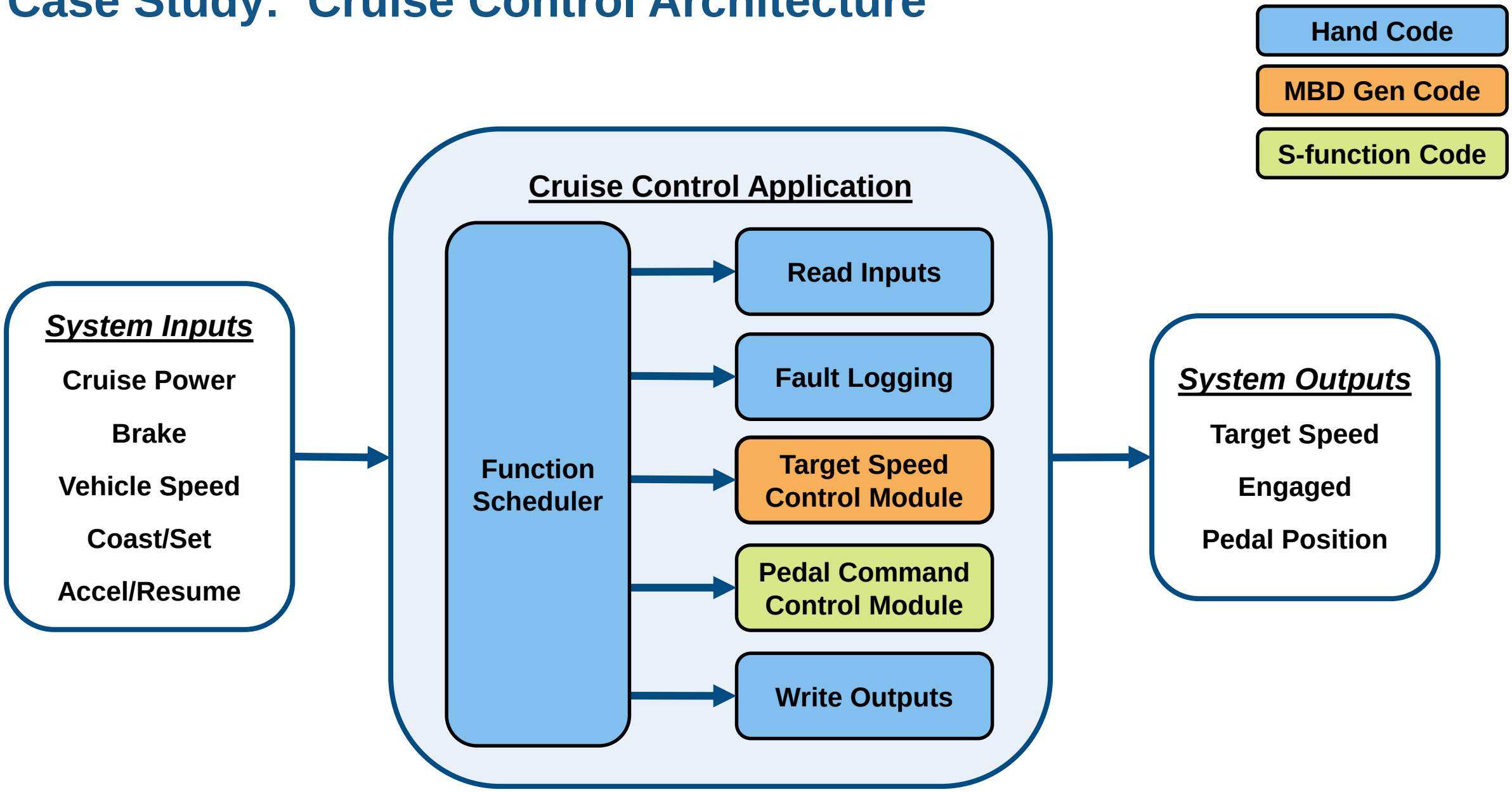
Objective: set cruise control target speed and pedal position based on driver & vehicle inputs



Cruise Control Application (C code)

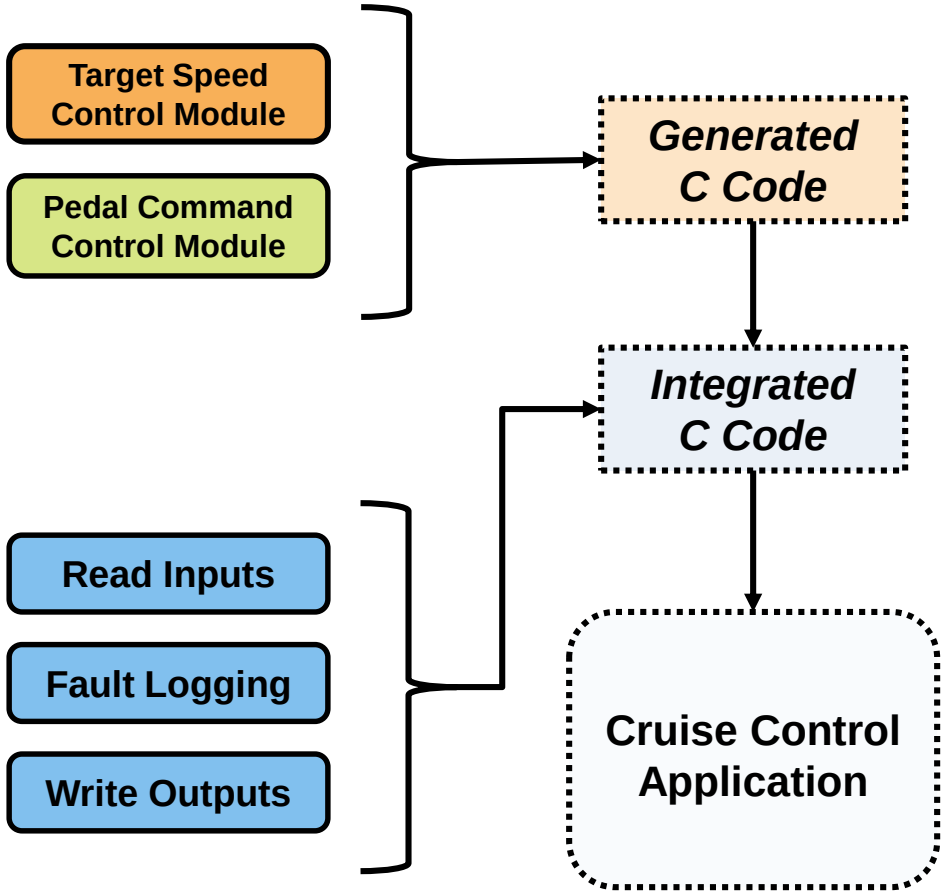
- *Hand code components*
- *Model-based Stateflow component*
- *Model-based S-function component*

Case Study: Cruise Control Architecture



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:

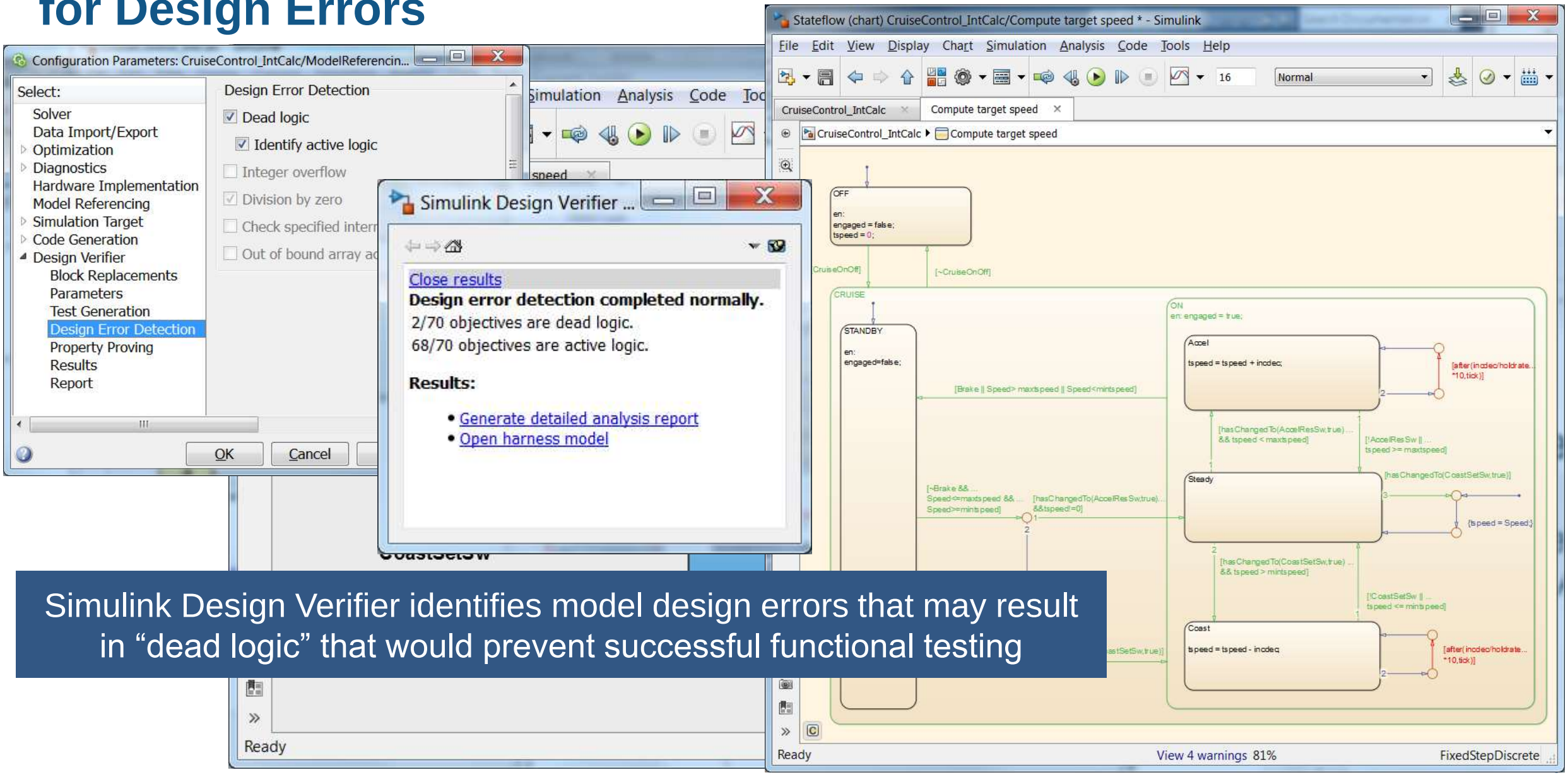
- **Convert signals/cals from floats to integers in Target Speed Module**
- **Include the customer lookup table in the Pedal Cmd to support calibration**
- **Demonstrate generated code is MISRA compliant**

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

MathWorks

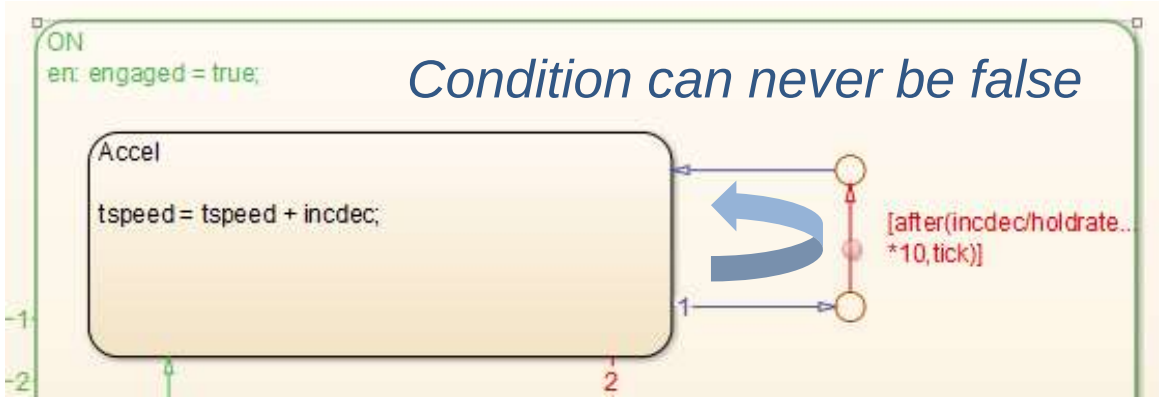
Target Speed Control Module



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

```
Command Window
debug>> incdec
incdec =
     1

debug>> holdrate
holdrate =
     5

debug>> class(incdec)
ans =
uint8

debug>> class(holdrate)
ans =
uint8

debug>> incdec/holdrate*10
ans =
     0

debug>> 10*incdec/holdrate
ans =
     2

fx debug>>
```



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

Model-based Design Tasks

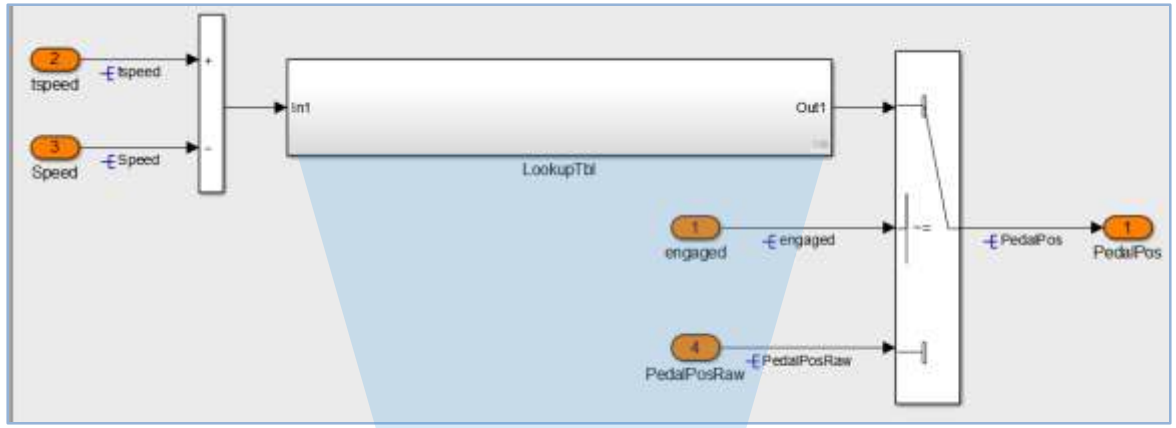
First let's focus on the model-based design tasks and what checks are available:

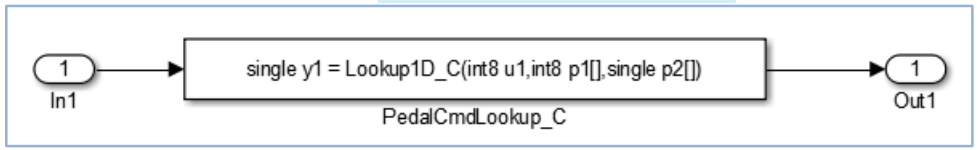
- **Convert signals/cals from floats to integers in Target Speed Module**
- **Include the customer lookup table in the Pedal Cmd to support calibration**
- **Demonstrate generated code is MISRA compliant**

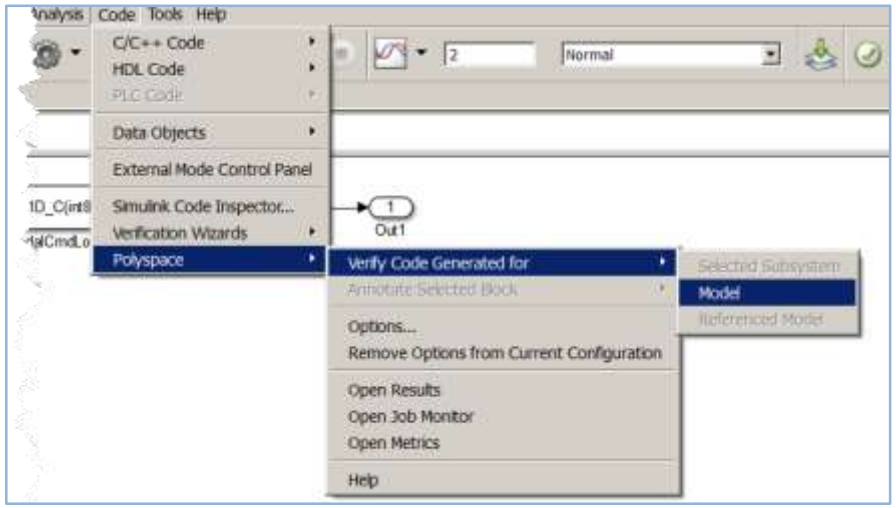
Our approach will be to do checks before functional testing, early in the development to minimize re-work

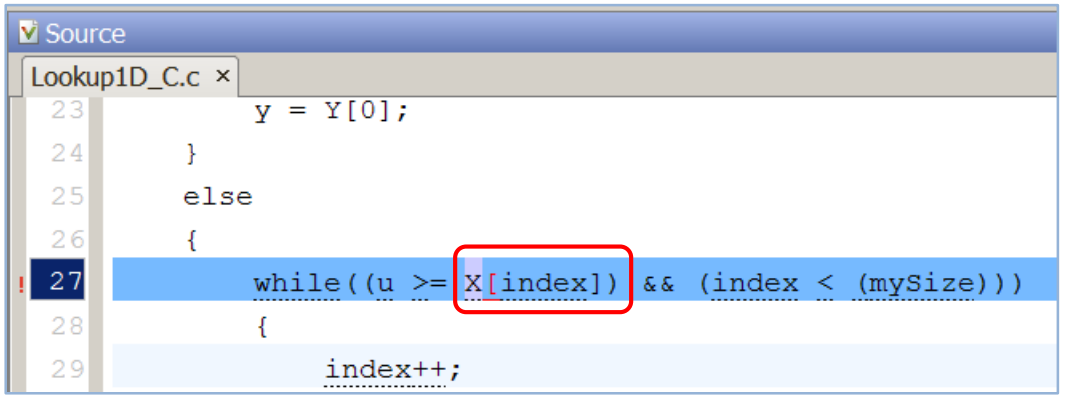
Customer Lookup Table: Checking the S-Function Code for Runtime Errors

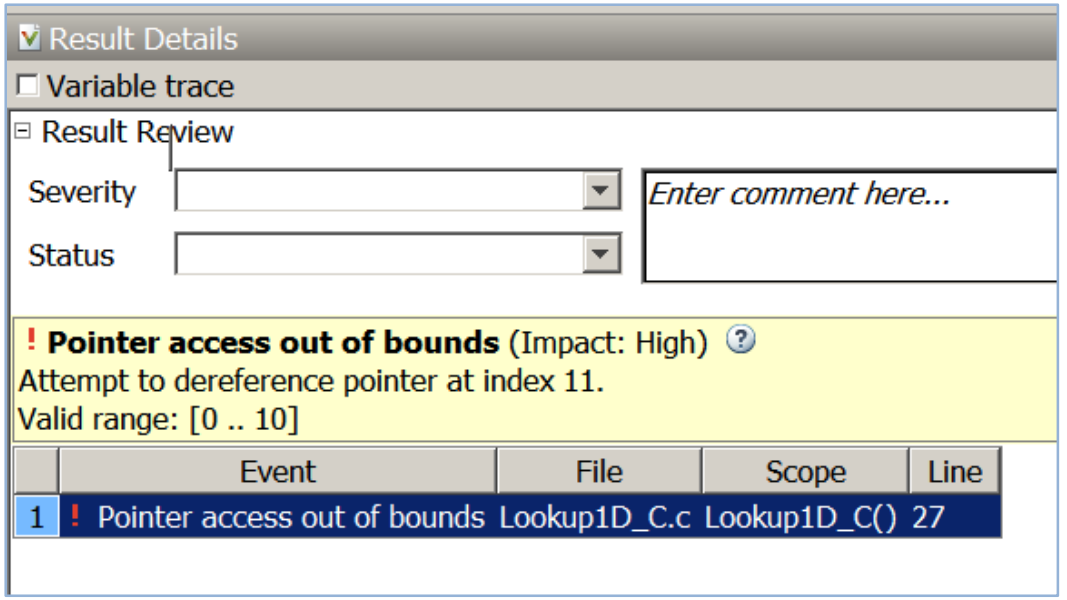
Pedal Command Control Module











Root Cause Analysis/Fix of S-Function Run-time Errors

MathWorks

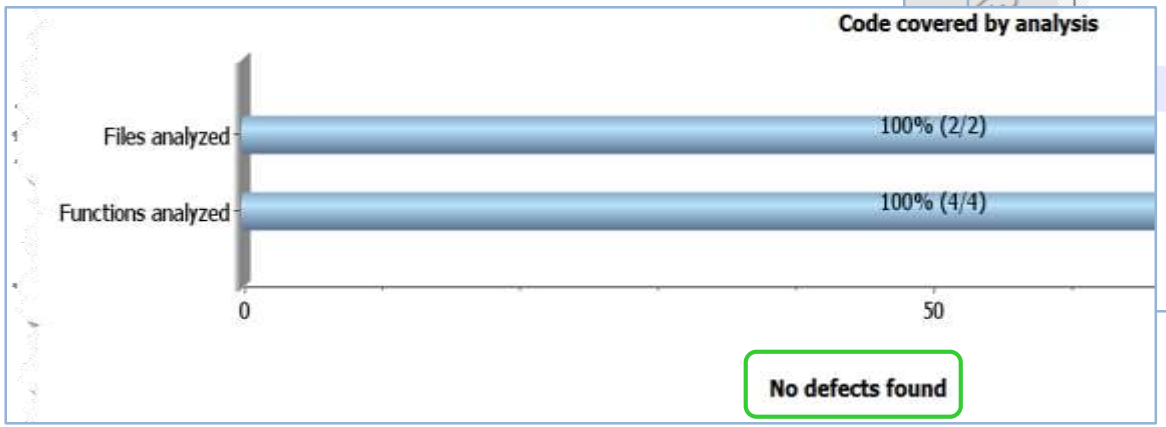
Pedal Command Control Module

```
25  /* Definition for custom storage class: Global */
26  real32_T PedalCmdY[11] = { 0.0F, 0.5F, 1.0F, 1.5F, 2.0F, 2.5F
27                             3.0F
28  int8_T SpeedDelX[11] = { -5, -
29
```

```
54      *   S-Function (PedalCmdLooku
55      *   Sum: '<Root>/Sum'
56      */
57    if (engaged) {
58      PedalPos = Lookup1D_C( (int8_T)rtb_Sum, (int8_T*)(&(SpeedDelX[0])),
59                             (real32_T*)(&(PedalCmdY[0])));
60    } else {
61      PedalPos = PedalPosRaw;
62    }
```

```
9   float Lookup1D_C(char u, char const X[], float const Y[])
10  {
11      float          y     = 0.0f;
12      unsigned char index = 0;
13      float          temp  = 0.0f;
14
15      unsigned char mySize = 11;
```

```
26          while((u >= X[index]) && (index < mySize))
27          {
28              index++;
25      else
        {
            while((u >= X[index]) && (index < (mySize-1)))
            {
                index++;
            }
            if (index > 0)
```



Model-based Design Tasks

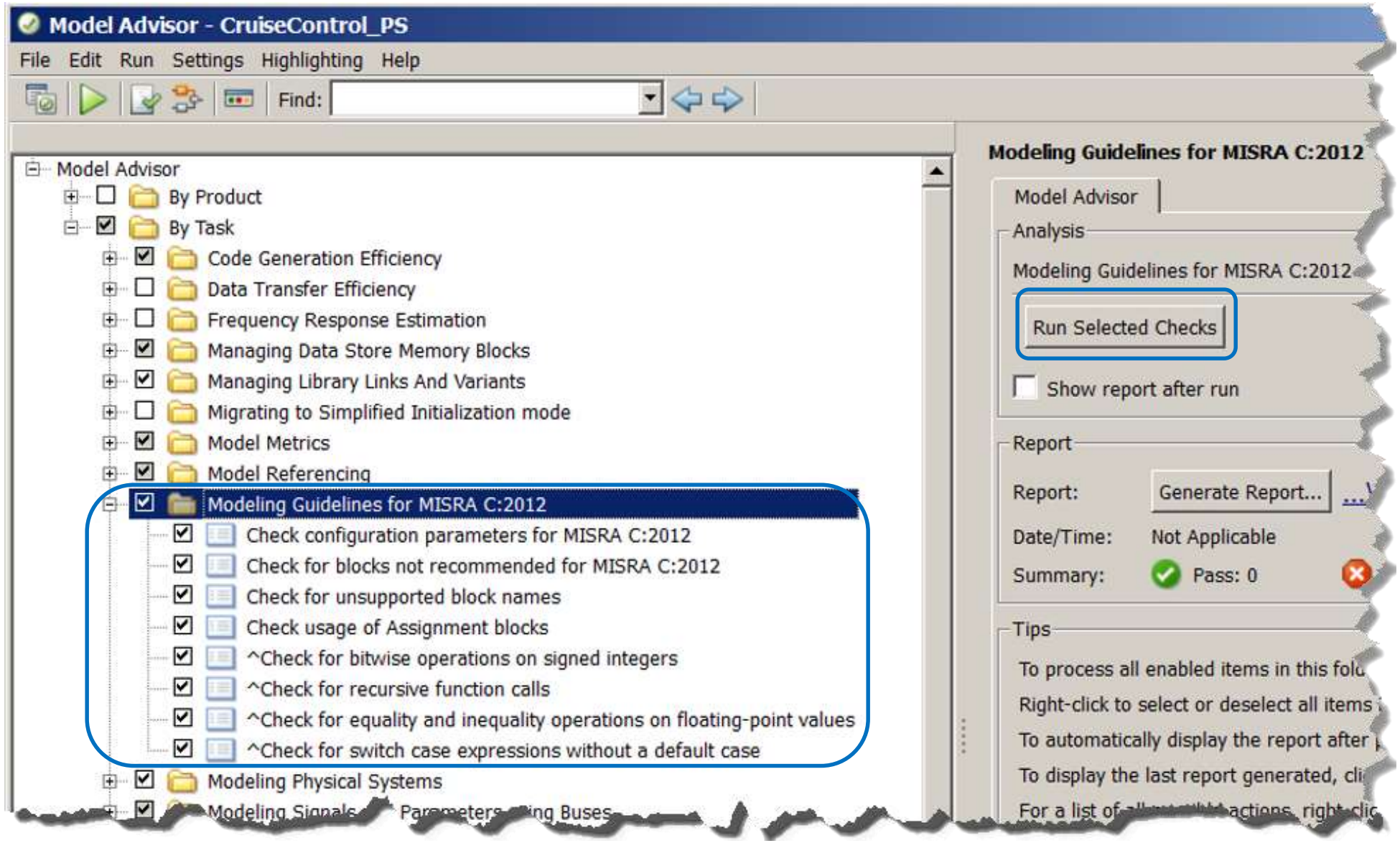
First let's focus on the model-based design tasks and what checks are available:

- **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

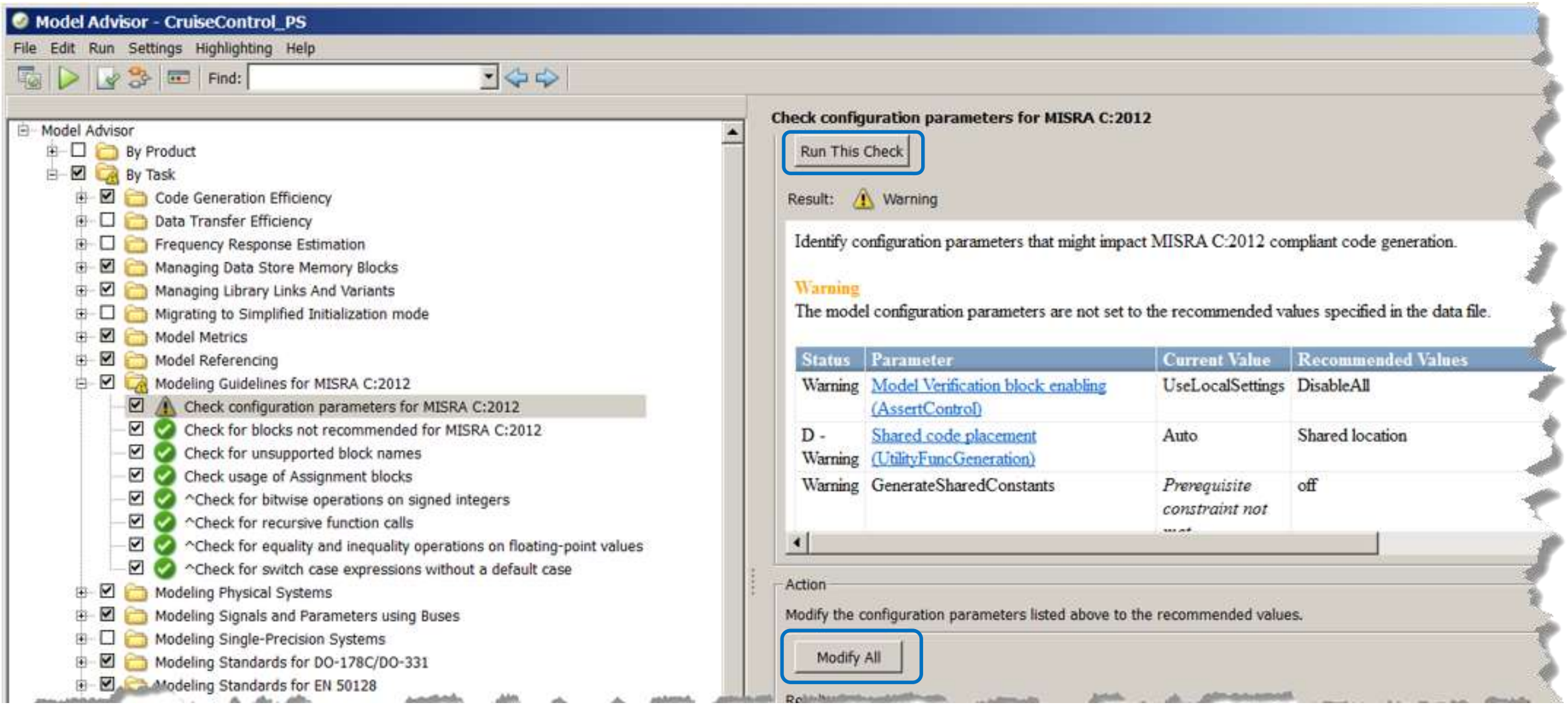
Checking Model for MISRA compliance with Model Advisor

Target Speed Control Module



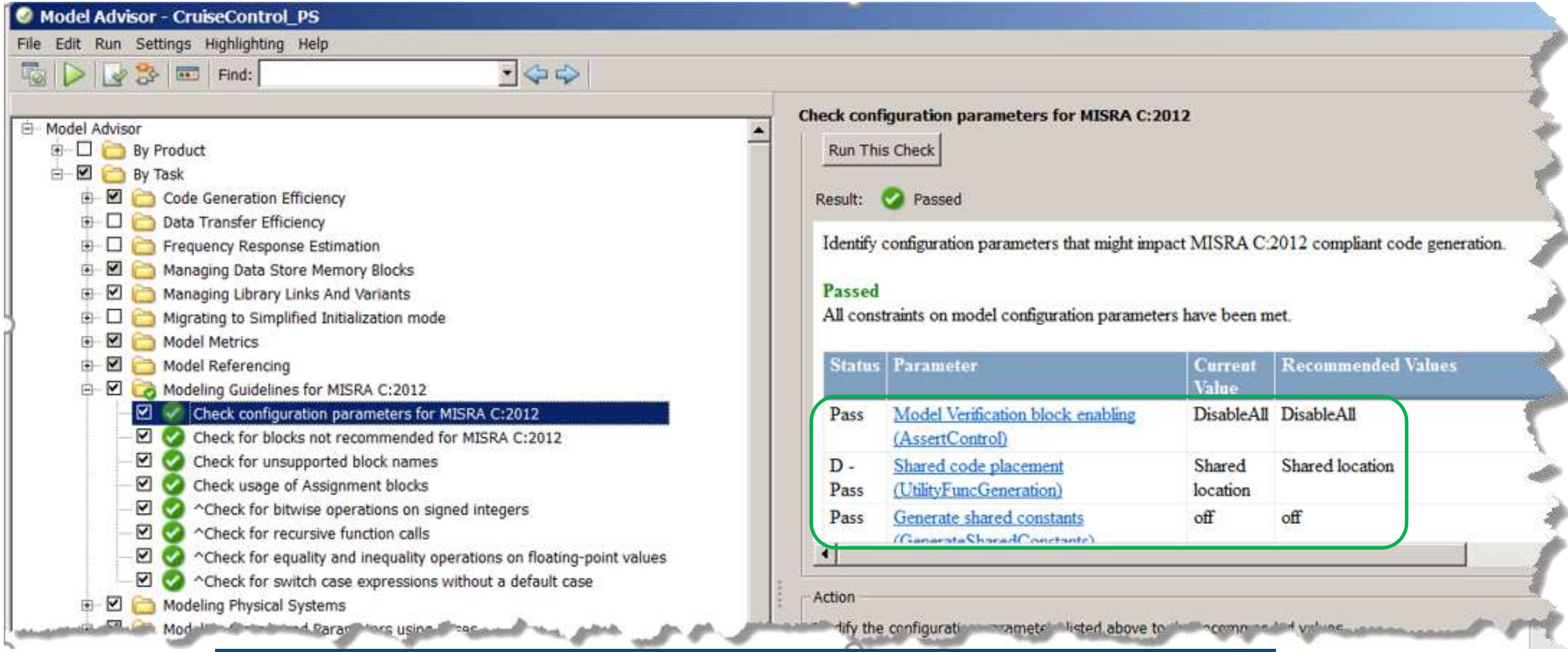
Checking Model for MISRA compliance with Model Advisor

Target Speed Control Module



Checking Model for MISRA compliance with Model Advisor

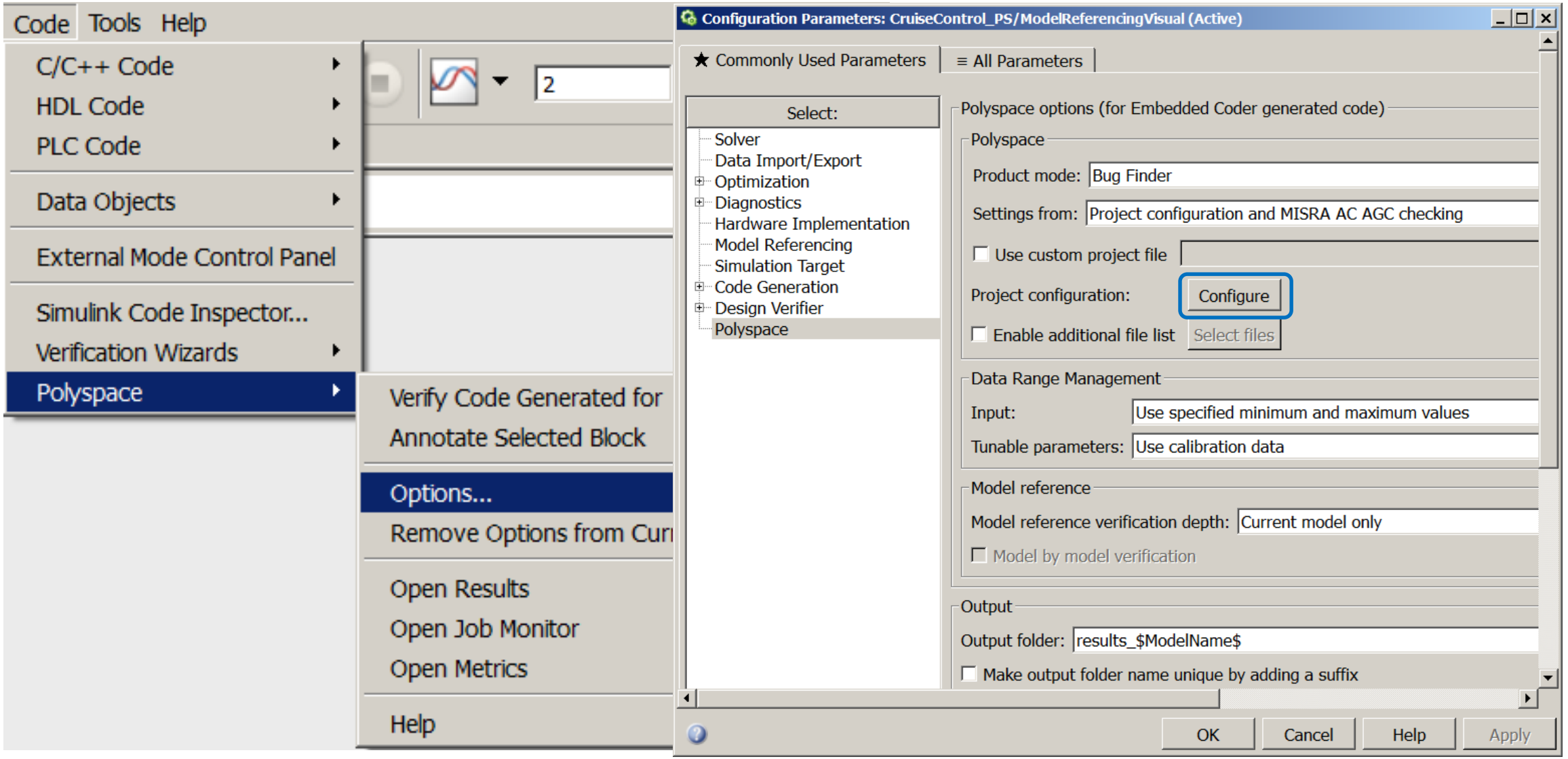
Target Speed Control Module



- ☑ *Checks model design and code configuration settings*
- ☑ *Increases likelihood of generating MISRA C:2012 compliant code*

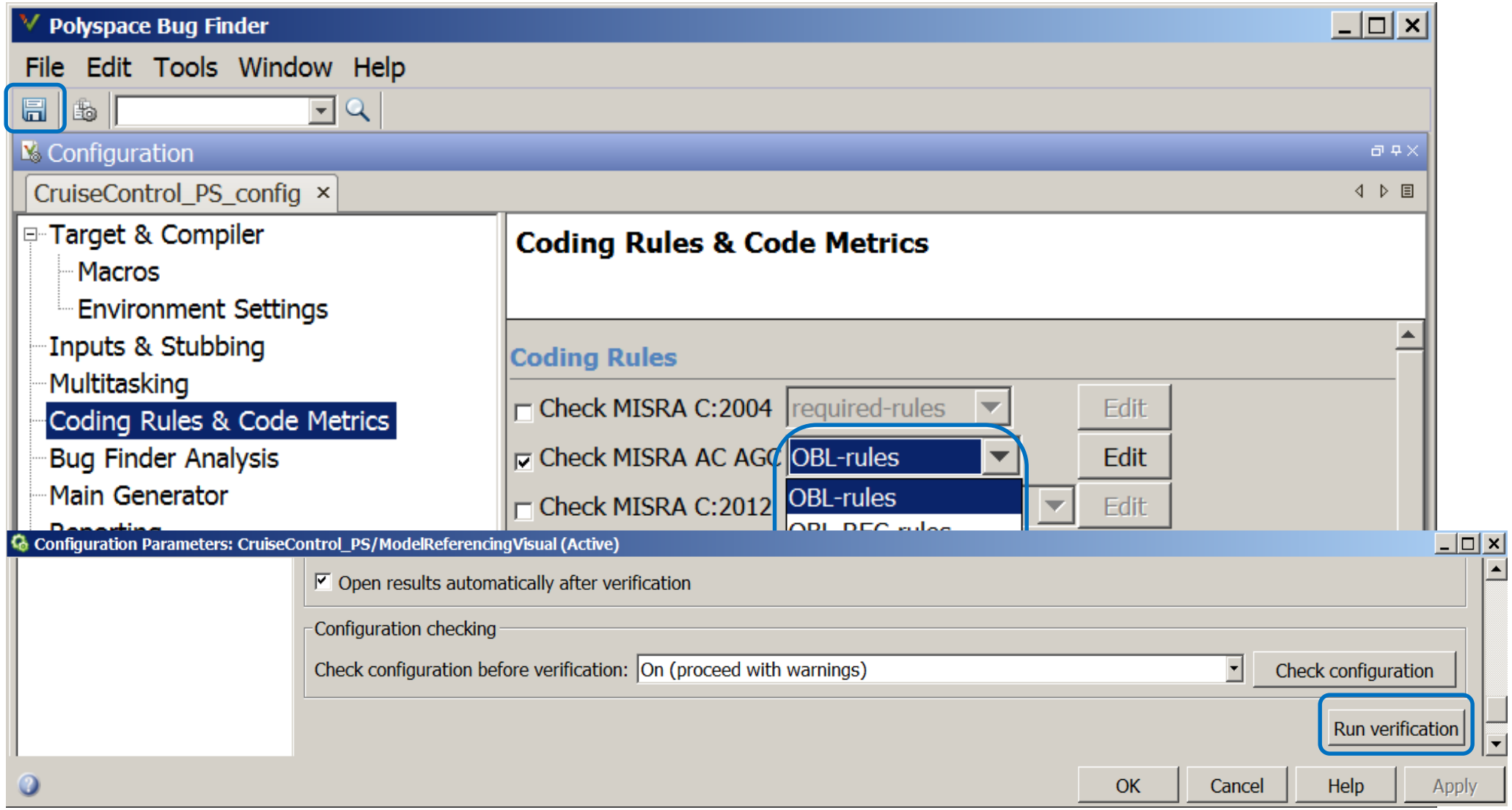
Configuring Polyspace from the Model

Target Speed Control Module



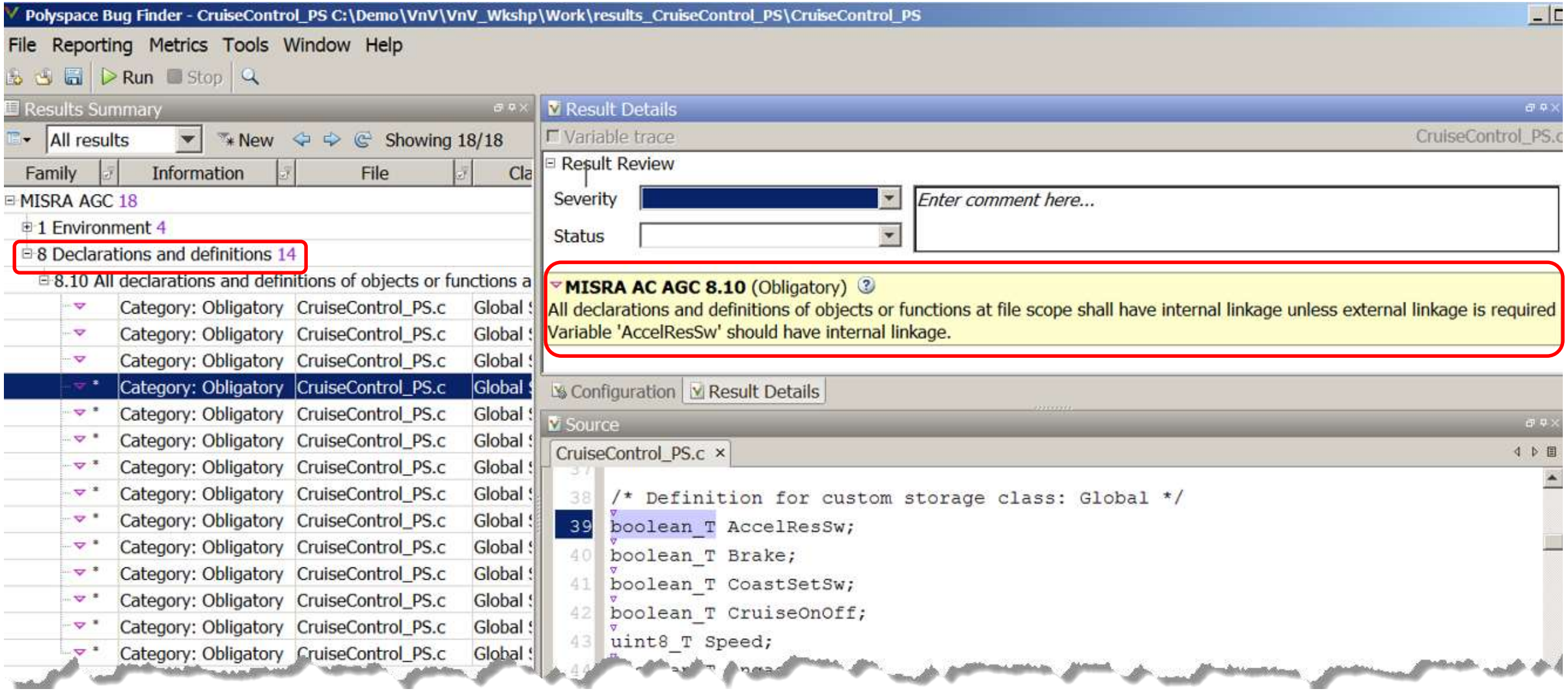
Launching Polyspace from the Model

Target Speed Control Module



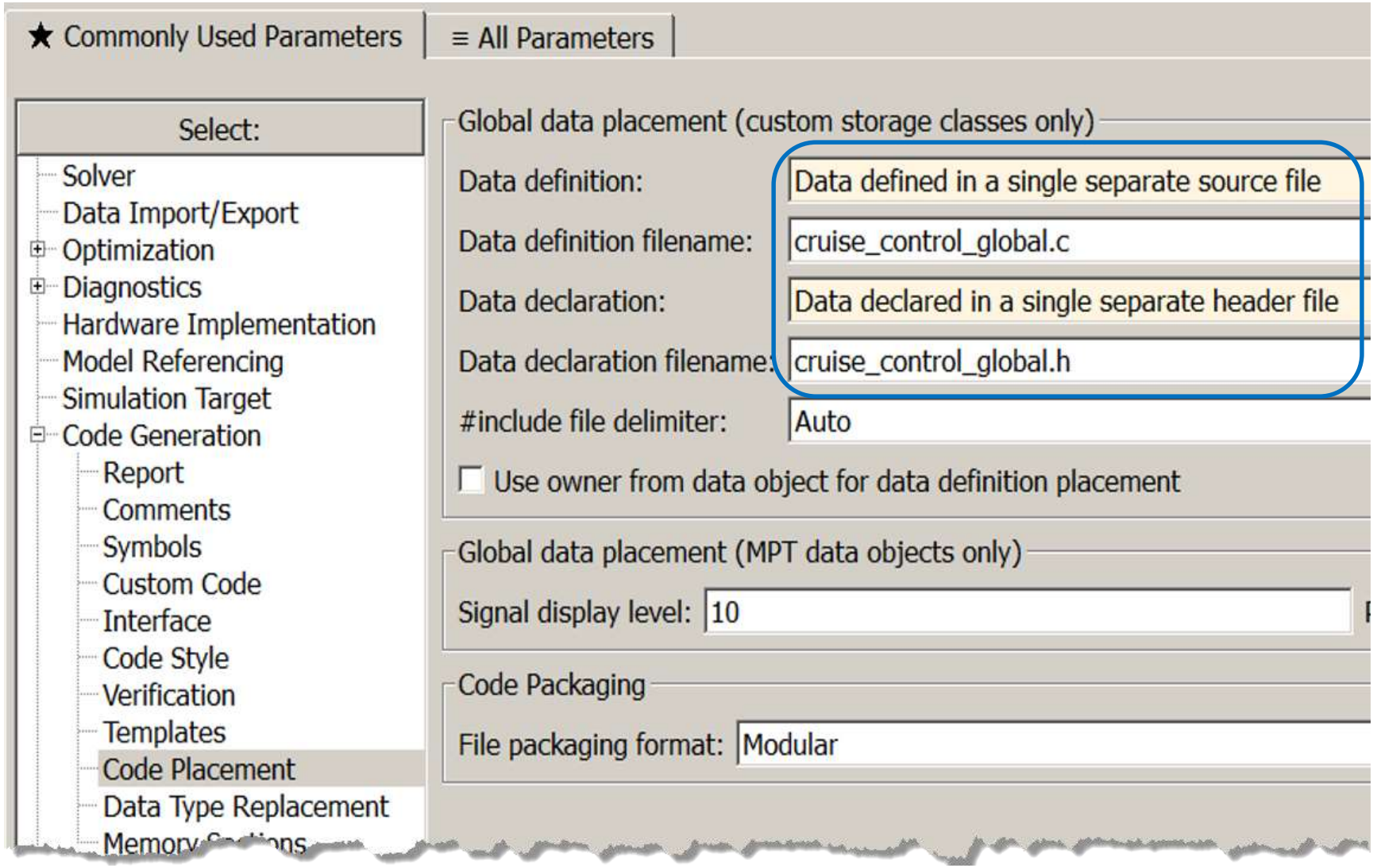
Review Bug Finder MISRA results

Target Speed Control Module



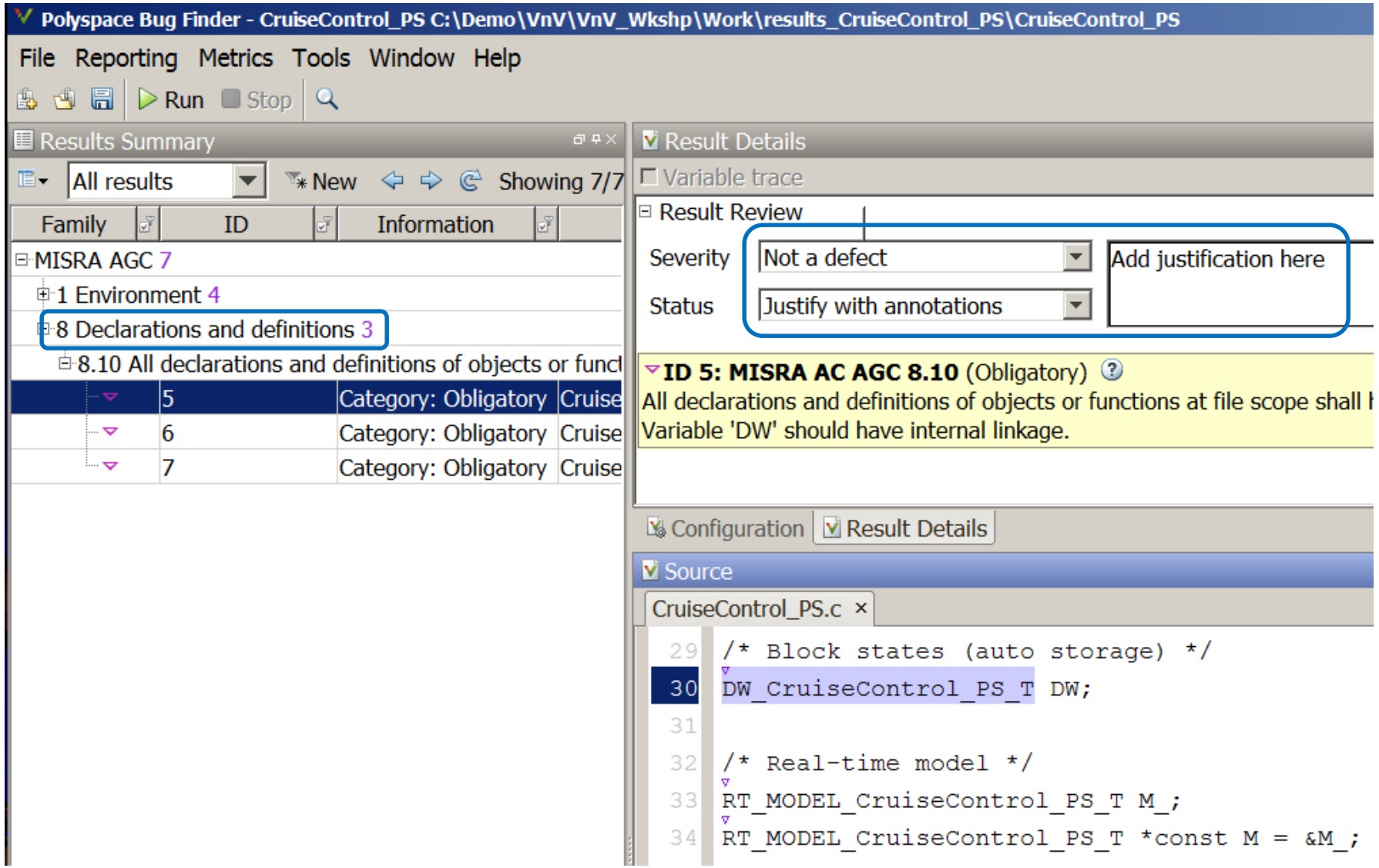
Reduce MISRA violations with “Code Placement” setting

Target Speed Control Module



Justify other violations by adding annotation

Target Speed Control Module



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

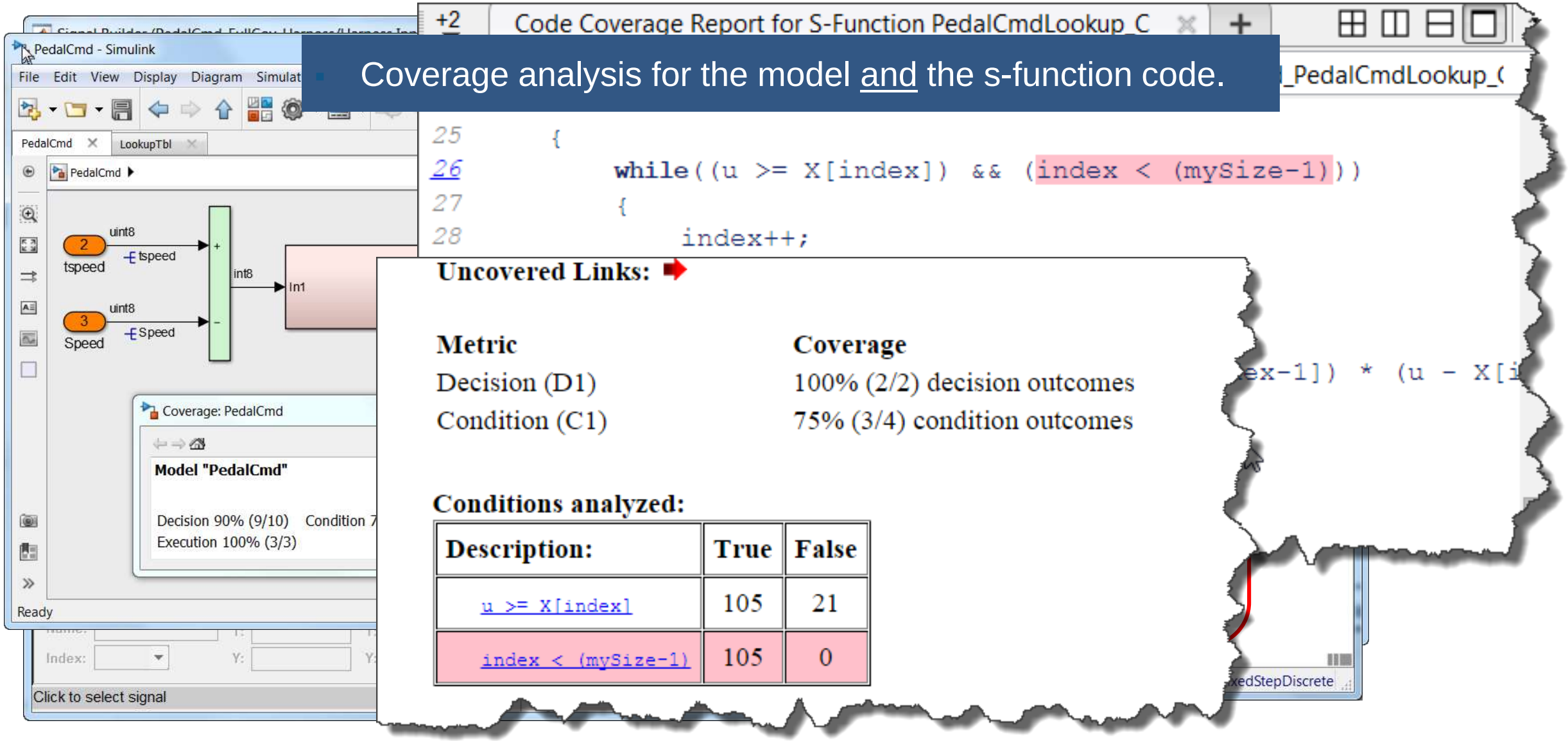
All checks are complete, we will need to provide test results for the model-based modules:

- **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)

Pedal Command Control Module

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

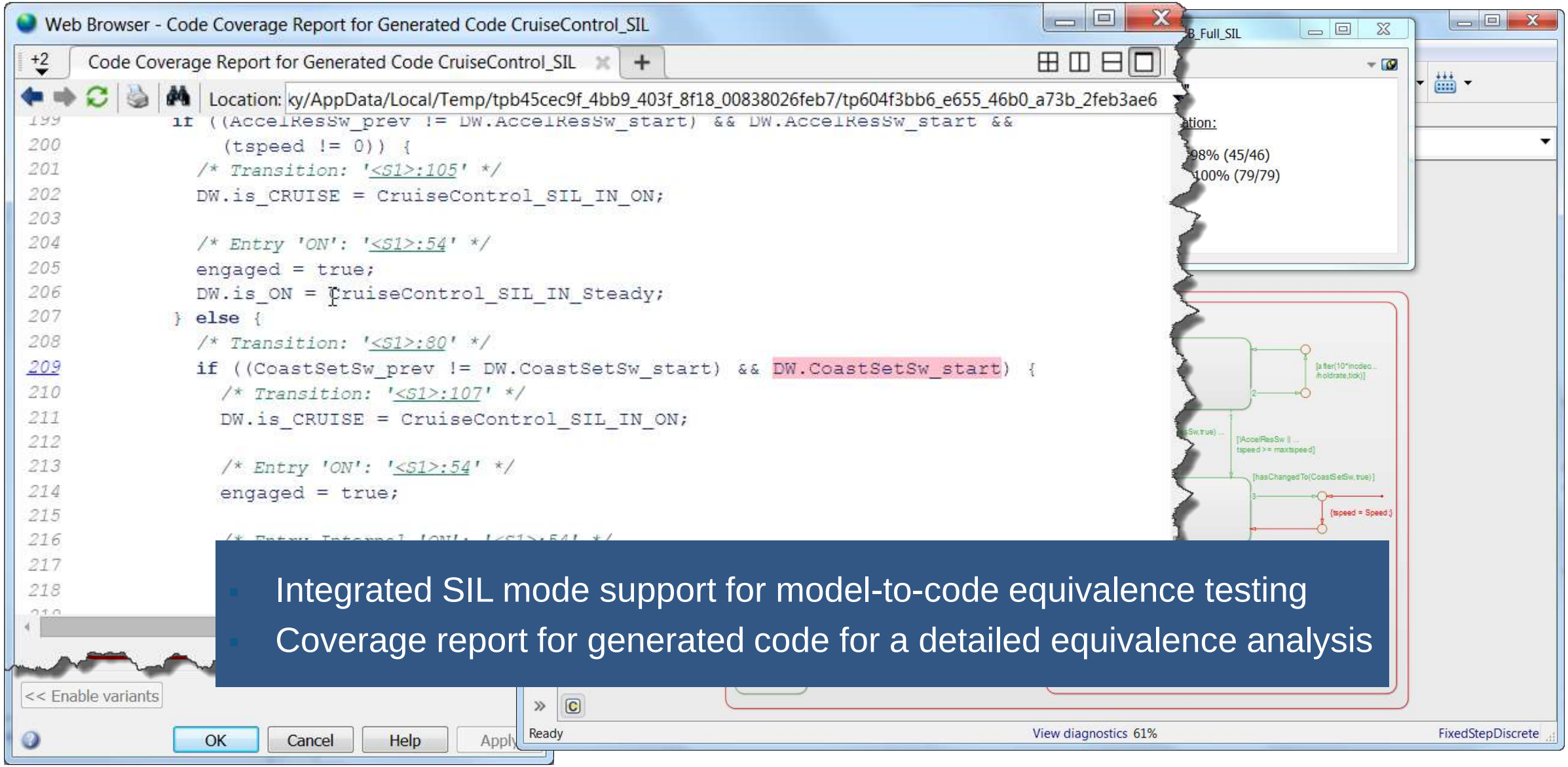


Model-based Design Tests

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.

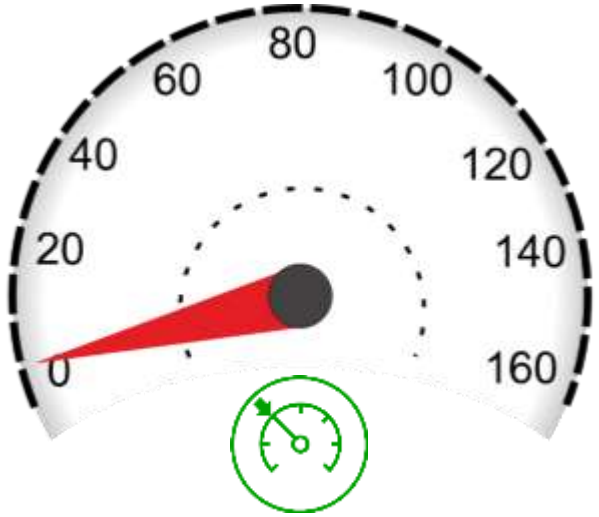
An ECU build was created based on the integration of hand code and generated code

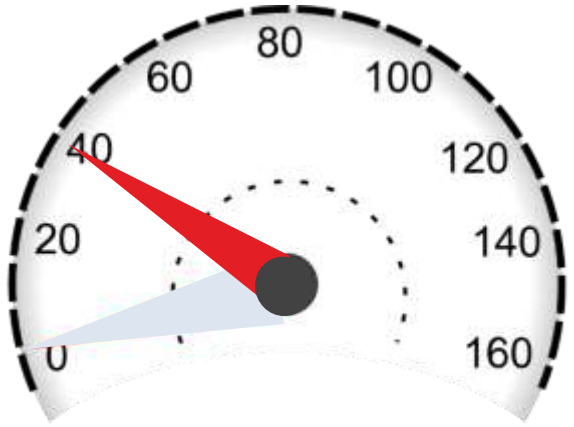
We now need to provide functional test results for the integrated code on the HiL bench

Find issues that result from the integration of tested modules from hand code, s-function code and model-based generated code.

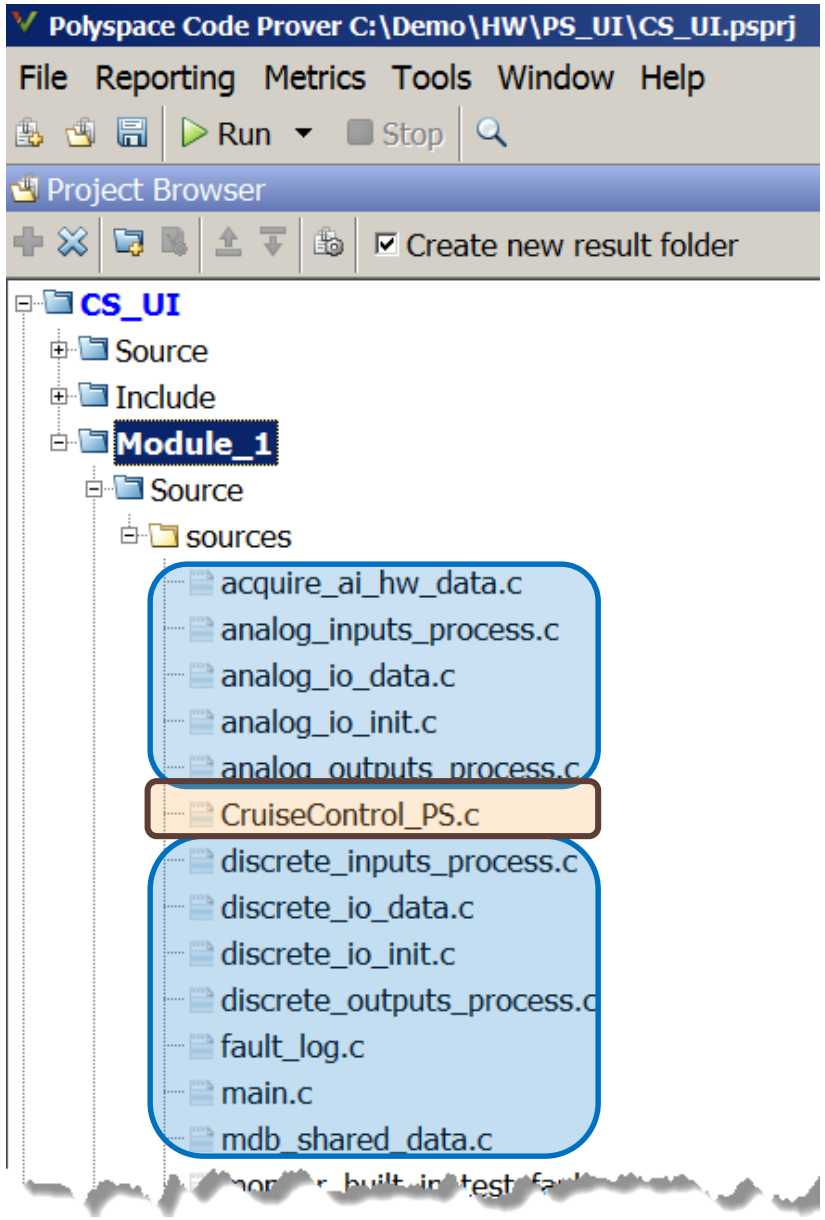
Issues Found on HIL Bench…

- *The Cruise Control powered off during fault testing*
- *And, the Target Speed never exceeded 40 mph*





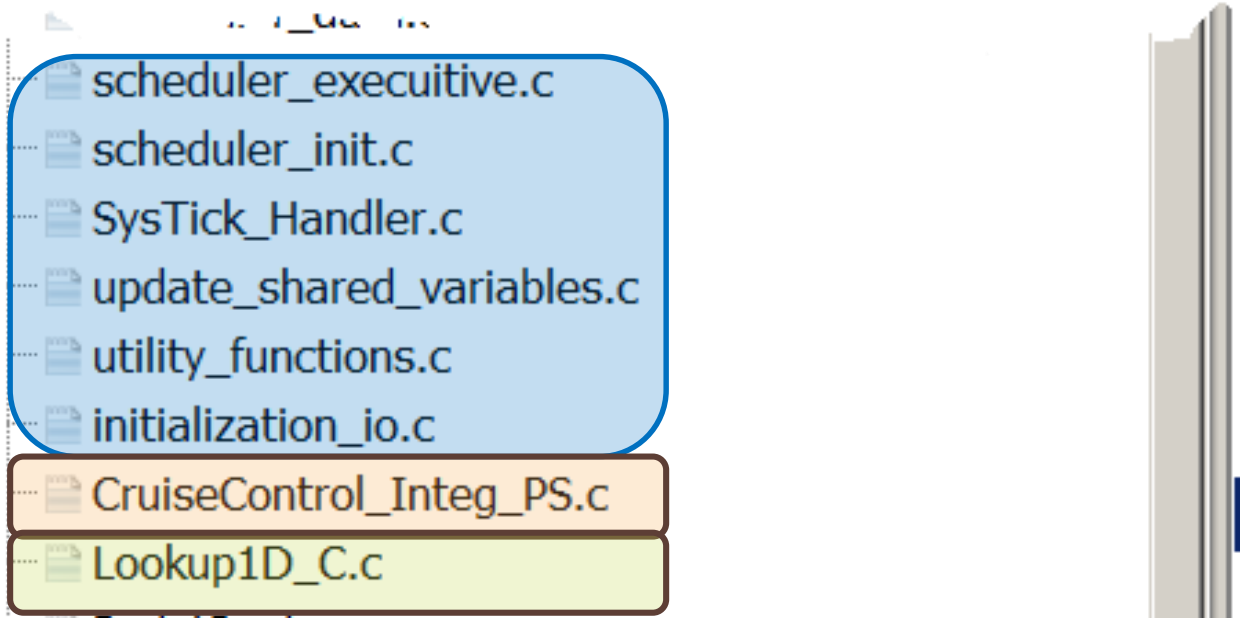
Creating a Code Prover project to check the Integrated Code



- Read Inputs
- Write Outputs
- Scheduler
- Fault Logging

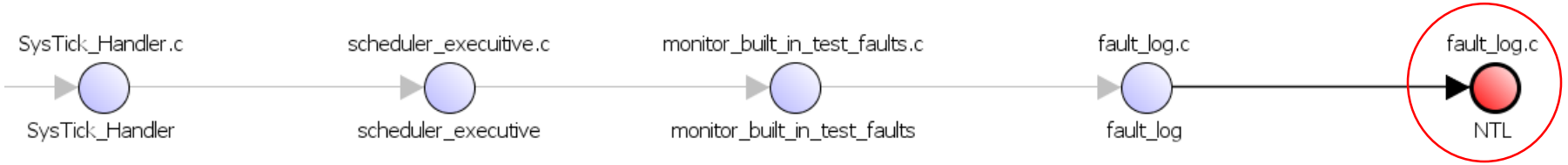
Target Speed Control Module

Pedal Command Control Module

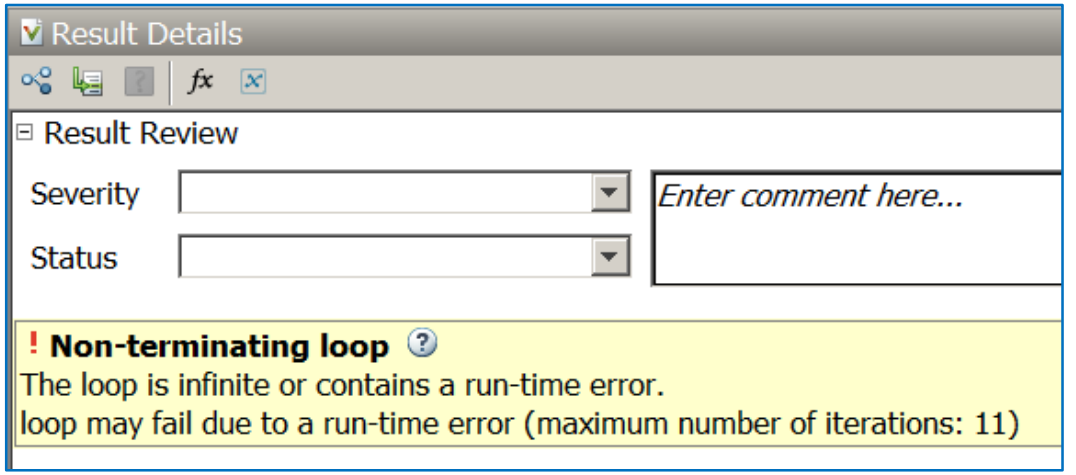


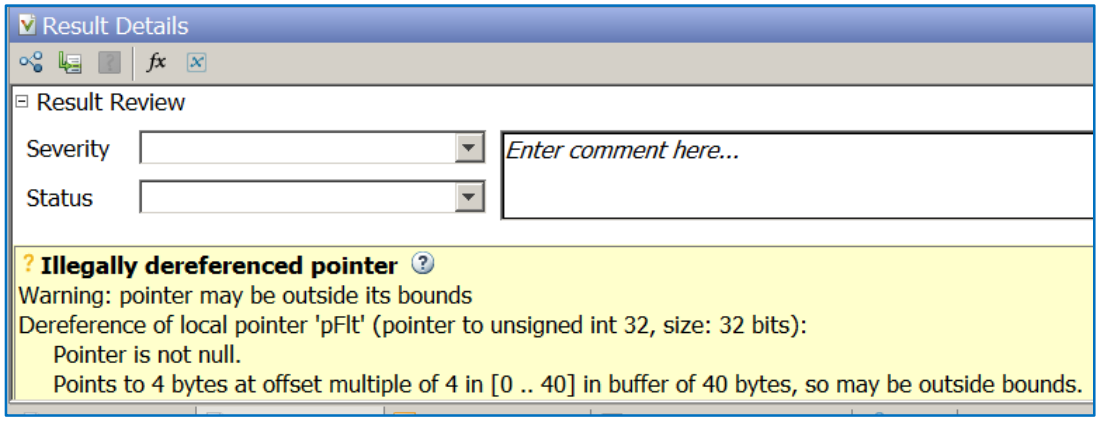
Code Integration Check with Polyspace: Non-terminating loop in Hand Code

Fault Logging




```
18  void fault_log(FAULT_LOG_INFO_T *pFaultInfo)
19  {
20      uint32_t ix;
21
22      uint32_t *pFlt = (uint32_t *)pFaultInfo;
23
24      /* Validate current fault index counter */
25      if (FaultRecIndexCntr >= (MAX_FAULT_LOG_INFO_SIZE - 12u))
26      {
27          FaultRecIndexCntr = 0x0u;
28      }
29
30      /* Store the current fault into circular buffer */
31      for (ix = 0; ix < 12u; ix++)
32      {
33          FaultInfoElement[FaultRecIndexCntr + ix] = *pFlt;
34          *pFlt = 0x0u;
35          pFlt++;
36      }
37      /* Update the circular buffer fault index counter */
38      FaultRecIndexCntr += ix-1;
39  }
```





Cause of Cruise Control Powering off during fault testing

```
18  void fault_log(FAULT_LOG_INFO_T *pFaultInfo)
19  {
20      uint32_t ix;
21
22      uint32_t *pFlt = (uint32_t *)pFaultInfo;
23
24      /* Validate current fault index counter */
25      if (FaultRecIndexCntr >= (MAX_FAULT_LOG_INFO_SIZE - 12u))
26      {
27          FaultRecIndexCntr = 0x0u;
28      }
29
30      /* Store the current fault into circular buffer */
31      for (ix = 0; ix < 12u; ix++)
32      {
33          FaultInfoElement[FaultRecIndexCntr + ix] = *pFlt;
34          *pFlt = 0x0u;
35          pFlt++;
36      }
37      /* Update the circular buffer fault index counter */
38      FaultRecIndexCntr += ix-1;
39  }
```

Fault Logging

ix	pFlt	typedef members
0	0x40000000	Expected_Value
1	0x40000004	Received_Value
2	0x40000008	Fault_ID
3	0x4000000c	Fault_Type
4	0x40000010	Time_mSec
5	0x40000014	Time_Sec
6	0x40000018	Time_Min
7	0x4000001c	Time_Hr
8	0x40000020	Additional_Flt_Spec01
9	0x40000024	Additional_Flt_Spec02
10	0x40000028	
11	0x4000002c	CruiseOnOff
		Brake
		CostalSetSw
		AccellResSw
	0x40000030	Speed

Root cause of Cruise Control Powering off

Fault Logging

```
30    /* Store the current fault into circular buffer */
31    for (ix = 0; ix < 12u; ix++)
32    {
33        FaultInfoElement[FaultRecIndexCntr + ix] = *pFlt;
34        *pFlt = 0x0u;
35        pFlt++;
36    }
41  typedef struct FAULT_LOG_INFO_TAG
42  {
43      uint32_t Expected_Value;
44      uint32_t Received_Value;
45      uint32_t Fault_ID;
46      uint32_t Fault_Type;
47      uint32_t Time_mSec;
48      uint32_t Time_Sec;
49      uint32_t Time_Min;
50      uint32_t Time_Hr;
51      uint32_t Additional_Flt_Spec01;
52      uint32_t Additional_Flt_Spec02;
53      uint32_t Additional_Flt_Spec03;
54      uint32_t Additional_Flt_Spec04;
55  }FAULT_LOG_INFO_T;
56
```

12

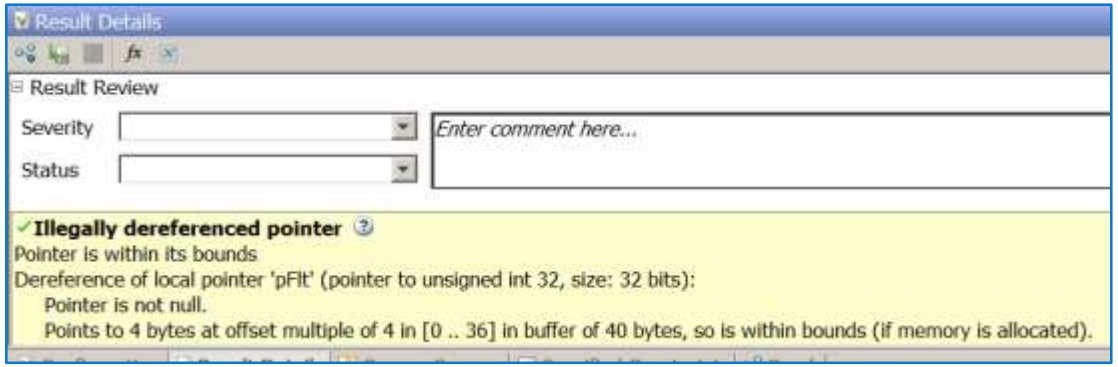
```
30    /* Store the current fault into circular buffer */
31    for (ix = 0; ix < 10u; ix++)
32    {
33        FaultInfoElement[FaultRecIndexCntr + ix] = *pFlt;
34        *pFlt = 0x0u;
35        pFlt++;
36    }
24  /* NEW: Fault log record information */
25  typedef struct FAULT_LOG_INFO_TAG
26  {
27      uint32_t Expected_Value;
28      uint32_t Received_Value;
29      uint32_t Fault_ID;
30      uint32_t Fault_Type;
31      uint32_t Time_mSec;
32      uint32_t Time_Sec;
33      uint32_t Time_Min;
34      uint32_t Time_Hr;
35      uint32_t Additional_Flt_Spec01;
36      uint32_t Additional_Flt_Spec02;
37  }FAULT_LOG_INFO_T;
```

10

Fix and verify the hand code is free of Runtime Errors

Fault Logging

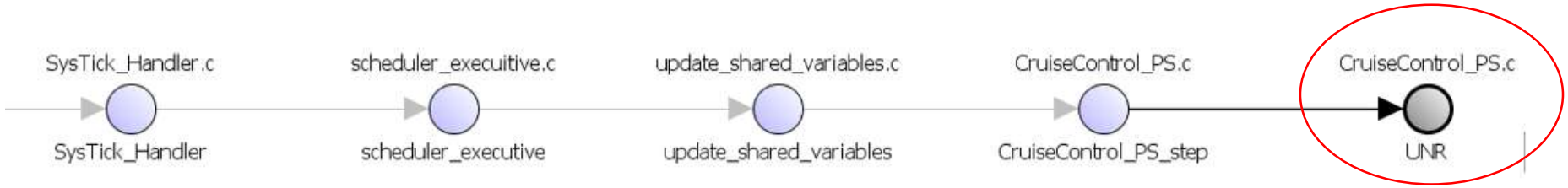
```
18  void fault_log(FAULT_LOG_INFO_T *pFaultInfo)
19  {
20      uint32_t ix;
21
22      uint32_t *pFlt = (uint32_t *)pFaultInfo;
23
24      /* Validate current fault index counter */
25      if (FaultRecIndexCntr >= (MAX_FAULT_LOG_INFO_SIZE - 12u))
26      {
27          FaultRecIndexCntr = 0x0u;
28      }
29
30      /* Store the current fault into circular buffer */
31      for (ix = 0; ix < 10u; ix++)
32      {
33          FaultInfoElement[FaultRecIndexCntr + ix] = *pFlt;
34          *pFlt = 0x0u;
35          pFlt++;
36      }
37      /* Update the circular buffer fault index counter */
38      FaultRecIndexCntr += ix-1;
39  }
```



ix	pFlt	typedef members
0	0x40000000	Expected_Value
1	0x40000004	Received_Value
2	0x40000008	Fault_ID
3	0x4000000c	Fault_Type
4	0x40000010	Time_mSec
5	0x40000014	Time_Sec
6	0x40000018	Time_Min
7	0x4000001c	Time_Hr
8	0x40000020	Additional_Flt_Spec01
9	0x40000024	Additional_Flt_Spec02
	0x4000002c	CruiseOnOff
		Brake
		CostalSetSw
		AccellResSw
	0x40000030	Speed

Code Integration Check with Polyspace: Dead Code Found in Generated Code

Target Speed Control Module



Vehicle speed signal propagated to "CruiseControl_PS.c" [0 … 40]

Maximum target speed = 90

```
111   } else if (Speed > maxtspeed) {
112     /* Transition: '<S1>:114' */
113     /* Exit Internal 'ON': '<S1>:54' */
114     DW.is_ON = CruiseContro_IN_NO_ACTIVE_CHILD;
115     DW.is_CRUISE = CruiseControl_PS_IN_STANDBY;
116
117     /* Entry 'STANDBY': '<S1>:52' */
118     engaged = false;
```

Unreachable/Dead code

Root Cause for Dead Code: Speed Sensor Input Hand Code

Read Inputs

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

analog.h

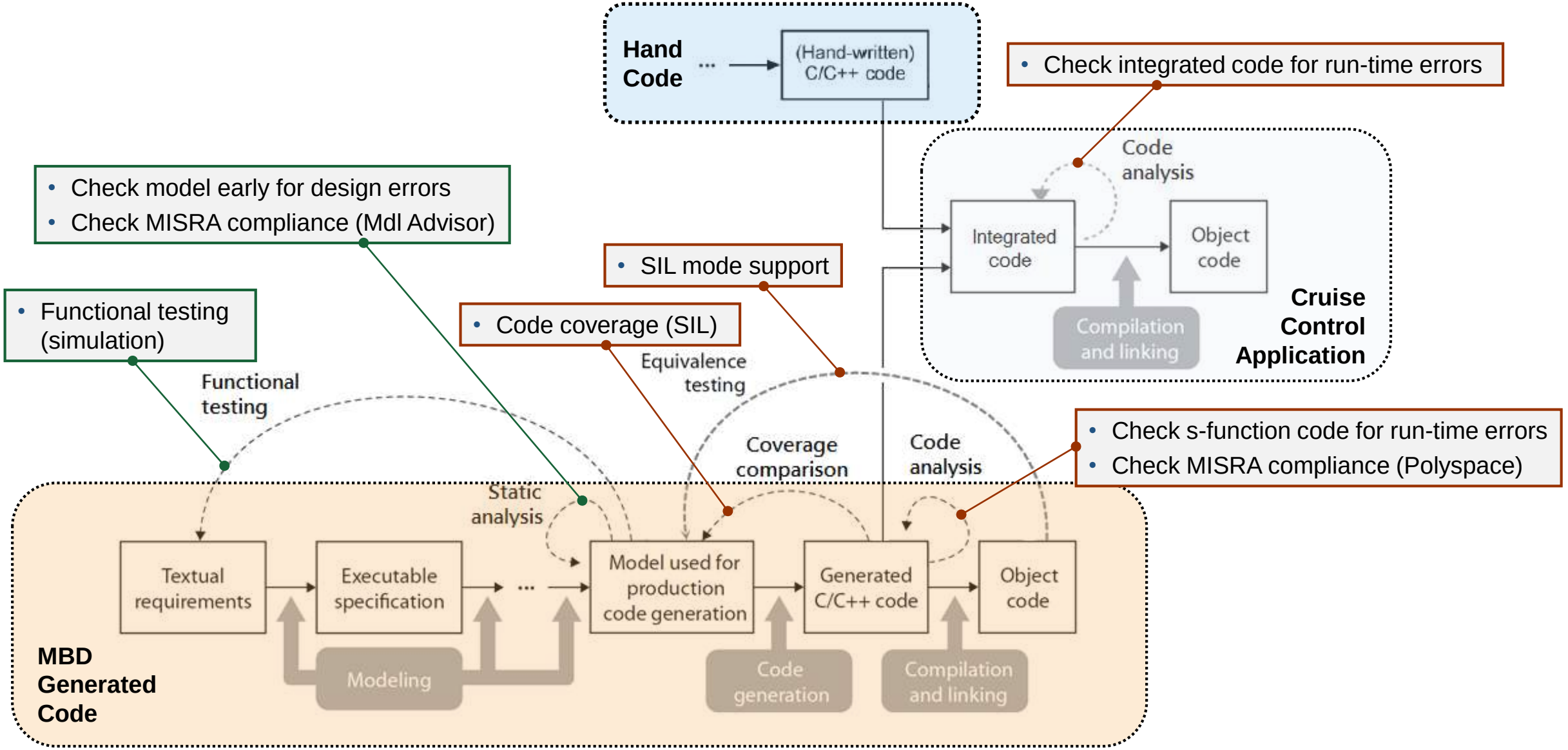
```
19 /* Conversion factors of speed */
20 #define NEW_ECU
21 #ifdef NEW_ECU
22     #define SPEED_MASK 0xFFF /* New ECU */
23 #else
24     #define SPEED_MASK 0x3FFF /* Original design specification */
25 #endif
26
27 /* Scaling for conversion factor for translating sensor input to miles/hr */
28 #define CONV_FACTOR 0.01 /* FAILS */
29
```

```
33
34     /* Convert raw counts to speed */
35     AI_Speed.Speed = ((AI_Speed.Average & SPEED_MASK) * CONV_FACTOR);
36
37     /* Updated analog inputs */
38     MDB_Shared_Data.Speed = AI_Speed.Speed;
39 }
```

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



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
  - hand code
  - s-function code
  - 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!