Unifying Model and Code Verification
Motivation

- Most controls applications are a combination of model-based generated code and hand code
- How do I efficiently verify this mix of hand code and generated code?
- MathWorks has tools for verifying models and tools for verifying 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

**Code Generation**
- Hand Code
- MBD Gen Code
- S-function Code
Workflow for integrated Model-based and Hand Code Components
Workflow for integrated Model-based and Hand Code Components

Hand Code

- Read Inputs
- Fault Logging
- Write Outputs

S-function Code

- Pedal Command Control Module

MBD Generated Code

- Target Speed Control Module

Architectural design
- Unit design
- (Manual) implementation

Software requirements
- Architectural design specification
- Unit design specification
- (Hand-written) C/C++ code

Target Speed

Cruise Control Application

- Integrated code
- Object code

Compilation and linking
Check MISRA compliance (Mdl Advisor)
Checking Model for MISRA compliance with Model Advisor
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
• Check MISRA compliance (Mdl Advisor)
• Check model early for design errors
Floats to Integers: Checking the Model for Design Errors

- Simulink Design Verifier identifies design errors on the model.
- These “dead logic” errors 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`

*Design Error Detection reveals this condition can never be false!*
Simulation and Test

- Functional testing (simulation)
Functional Testing Workflow

Requirements

Did we meet our requirements?

Review functional behavior

Tests

Did we completely test our model?

Structural coverage report
Simulink Test™

- Authoring Tests
- Managing Tests
- Execution of Tests

- Key Features
  - Test harness
  - Test sequence block for running tests and assessments
  - Pass-fail criteria, including tolerances, limits, and temporal conditions
  - Baseline, equivalence, and back-to-back testing
  - Automatic report generation to document test outcomes
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>$index &lt; (mysize-1)$</td>
<td>105</td>
<td>0</td>
</tr>
</tbody>
</table>
Back-to-Back Comparison Test (ISO 26262)

Table 10 — Methods for software unit testing

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL A</th>
<th>ASIL B</th>
<th>ASIL C</th>
<th>ASIL D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>Requirements-based test(^a)</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1b</td>
<td>Interface test(^b)</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1c</td>
<td>Fault injection test(^c)</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1e</td>
<td>Back-to-back comparison test between model and code, if applicable(^e)</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
</tbody>
</table>

\(^a\) The software requirements of the unit level are the basis for this requirements-based test.
\(^b\) This includes injection of arbitrary faults (e.g., by corrupting values of variables, by introducing code mutations, or by corrupting values of CPU registers).
\(^c\) Some aspects of the resource usage test can only be evaluated properly when the software unit tests are executed on the target hardware or if the simulator for the target processor supports resource usage tests.
\(^d\) This method requires a model that can simulate the functionality of the software units. Here, the model and code are stimulated in the same way and results compared with each other.

Table 13 — Methods for software integration testing

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL A</th>
<th>ASIL B</th>
<th>ASIL C</th>
<th>ASIL D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>Requirements-based test(^d)</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1b</td>
<td>Interface test(^e)</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1c</td>
<td>Fault injection test(^f)</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
</tbody>
</table>

\(^d\) This method requires a model that can simulate the functionality of the software components. Here, the model and code are stimulated in the same way and results compared with each other.
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
Optional: Automatic Code Reviews
Automatic Code Reviews

Model and code development

Simulink Code Inspector ®

Independent code inspection

- Static verification tool.
- Checks the generated code against model
Simulink Design Verifier – Automatic Test Generation

- Functional testing (simulation)
- Code coverage (SIL)

(Hand-written) C/C++ code

Integrated code

Object code

Compilation and linking

Code analysis

Static analysis

Model used for production code generation

Generated C/C++ code

Object code

Code generation

Compilation and linking
SL Design Verifier created test cases to cover all possible “Decision/Condition” objectives in the model and s-function code.
Simulink Design Verifier – Test Generation for Modules

Overview

- **Input**
  - Model
  - Coverage metric
    - Decision coverage
    - Condition coverage
    - MC/DC
    - Custom Objectives (requirements-based)

- **Optional Input**
  - Modifiable parameter sets
  - Existing coverage data

- **Results**
  - Harness model
  - Input test signals
  - Unreachable objects
  - Detailed reports
Simulink Design Verifier – Property Proving

Overview

- Inputs
  - Model
  - Requirements
- Optional Inputs
  - Assumptions
- Proof satisfied or Proof falsified
- Counterexample
- Detailed reports
Simulink Design Verifier – Use Cases

Model Slicing

Test Generation

Design Error Detection

Property Proving
Issues Found on HIL Bench…

- The Cruise Control powered off during fault testing
- And, the Target Speed never exceeded 40 mph
Polyspace product family for C/C++

- Polyspace Code Prover
  - Proves code to be safe and dependable
  - Deep verification of software components
  - Perform QA signoff for production ready code

- Polyspace Bug Finder
  - Quickly find bugs in embedded software
  - Check code compliance for MISRA and JSF
  - Intended for every day use by software engineers

Ada language also supported for proving code
Customer Lookup Table: Checking the S-Function Code for Runtime Errors

- Pointer access out of bounds (Impact: High)
  Attempt to dereference pointer at index 11.
  Valid range: [0 .. 10]
Root Cause Analysis/Fix of S-Function Run-time Errors

```c
/* 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])),
    }
    else {
        while((u >= X[index]) && (index < mySize))
        {
    }
    else
    {
        while((u >= X[index]) && (index < (mySize-1)))
        {
            index++;
        }
        if (index > 0)
        {
```

Pedal Command Control Module
Configuring Polyspace from the Model
Launching Polyspace from the Model

CruiseControl_PS_config

Coding Rules & Code Metrics

Coding Rules
- Check MISRA C:2004
- Check MISRA AC AGO
- Check MISRA C:2012
- Check custom rules
- Effective boolean types

OBL-rules
- OBL-REC-rules
- all-rules
- SQO-subset1
- SQO-subset2
- custom

Run verification
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.

8 Declarations and definitions 14

/* Definition for custom s */

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>
</tbody>
</table>

Target Speed Control Module
Justify other violations by adding annotation

```
/* Block states (auto storage) */

30
DW_CruiseControl_PS_T DW;

/* Real-time model */

32
RT_MODEL_CruiseControl_PS_T M_

33
RT_MODEL_CruiseControl_PS_T *const M = &M_;
```
What support is there for analyzing the integrated code?

- Check integrated code for run-time errors
  - Polyspace
Why code verification? Code verification is exhaustive!

Statically and semantically verifies all possible executions of your code (considering all possible inputs, paths, variable values)

- **Proves** when code **will not fail** under any runtime conditions, reducing robustness testing efforts
- Finds **unreachable code**, **runtime errors**, **boundary conditions** and without testing
- Gives insight into runtime behavior and data ranges

```c
static void pointer_arithmetic (void) {
    int array[100];
    int *p = array;
    int i;
    for (i = 0; i < 100; i++) {
        *p = 0;
        p++;
    }
    if (get_bus_status() > 0) {
        if (get_oil_pressure() > 0) {
            *p = 5;
        } else {
            i++;
        }
    }
    i = get_bus_status();
    if (i > 0) {
        *p = 10;
    }
}
```
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

SysTick_Handler.c  scheduler_executive.c  monitor_builtin_test_faults.c  fault_log.c

SysTick_Handler  scheduler_executive  monitor_builtin_test_faults  fault_log

/* Store the current fault into circular buffer */
for (ix = 0; ix < 12u; ix++)
{
    FaultInfoElement[FaultRecIndexCnter + ix] = *pFlt;
    *pFlt = 0x00u;
    pFlt++;
}

? Illegally dereferenced pointer ?
Warning: pointer may be outside its bounds
Dereference of local pointer 'pFlt' (pointer to unsigned 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.

! 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>idx</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>0x4000000c</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></td>
<td></td>
<td>Speed</td>
</tr>
</tbody>
</table>
Code Integration Check with Polyspace: Dead Code Found in Generated Code

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

Maximum target speed = 90

Unreachable/Dead code

```
} else if (Speed > maxtspeed) {
    /* Transition: '<S1>:114' */
    /* Exit Internal 'ON': '<S1>:54' */
    DW.is_ON = CruiseContro_IN_NO_ACTIVE_CHILD;
    DW.is_CRUISE = CruiseControl_PS_IN_STANDBY;
    /* Entry 'STANDBY': '<S1>:52' */
    engaged = false;
```
Root Cause for Dead Code: Speed Sensor Input Hand Code

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

```
/* Conversion factors of speed */
#define NEW_ECU
#ifdef NEW_ECU
  #define SPEED_MASK 0xFFF /* New ECU */
#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_BUFFER_SIZE 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;
```
Workflow Summary: Complementary Model & Code Verification

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

MBD Generated Code

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

Modeling
Code generation
Compilation and linking

Hand Code → (Hand-written) C/C++ code

Code analysis
Compilatation and linking

Static analysis
ISO 26262 Structure

ISO 26262-1 • Vocabulary
ISO 26262-2 • Management of functional safety
ISO 26262-3 • Concept phase
ISO 26262-4 • Product development: system level
ISO 26262-5 • Product development: hardware level
ISO 26262-6 • Product development: software level
ISO 26262-7 • Production and operation
ISO 26262-8 • Supporting processes
ISO 26262-9 • ASIL-oriented and safety-oriented analyses
ISO 26262-10 • Guideline

- Model-Based Design
- Early verification and validation
- Code generation
- Tool classification and qualification
ISO 26262 Tool Qualification Approach - 

**I. Tool Classification**

<table>
<thead>
<tr>
<th>Tool use cases</th>
<th>Tool impact</th>
<th>Tool error detection</th>
<th>Tool confidence level</th>
</tr>
</thead>
<tbody>
<tr>
<td>UC 1..n</td>
<td>TI 2</td>
<td>TD 3</td>
<td>TCL 3</td>
</tr>
<tr>
<td></td>
<td>TI 1</td>
<td>TD 2</td>
<td>TCL 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TD 1</td>
<td>TCL 1</td>
</tr>
</tbody>
</table>

**II. Tool Qualification**

- Increasing qualification requirements
- Qualification methods for TCL1
- Qualification methods for TCL2
- Qualification not required
Qualification of MathWorks Tools

Tool qualification may involve multiple parties

- **Tool user**
  - Responsible for final tool qualification in the context of the application

- **Tool vendor**
  - Conducts generic pre-classification and pre-qualification based on reference use cases / reference workflow

- **3rd party assessor** (optional)
  - Provides independent assessment of reference workflow and pre-qualification artifacts

ISO 26262 Tool Qualification Kit

TÜV SÜD Independent Assessment

MathWorks Pre-classification / qualification based on typical use cases (reference workflows)

Tool User
Project-specific adaptation
Independent Assessment by TÜV SÜD

Example

Certificate

No. Z10 11 01 67052 008

Holders of Certificate:
The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760
USA

Factory(ies):

Certification Mark:

Product:
Software Tools for Safety Related Development

Model(s):
Simulink® Verification and Validation™ Simulink® Design Verifier™

Parameters:
The certification marks are valid for software-related development of safety-related software according to IEC 61508, EN 50128, EN 50129, and derivative standards. The certification marks are valid for software-related development of safety-related software according to ISO 26262. MN66534C is a variant of the certificate.

Tasked according to:
IEC 61508-3:2011 (Category B to C)
EN 50129:2006 (Category 4 to 5)
EN 50128:2011

The product is tested on a voluntary basis and complies with the essential requirements. The certification marks are valid only on condition that the products is not marketed in an incorrect manner. The certification mark is a voluntary quality mark for the products and serves to inform and to assure the addressee of the certificate holder that the products meet the essential requirements.

The product is tested on a voluntary basis and complies with the essential requirements. The certification marks are valid only on condition that the products is not marketed in an incorrect manner. The certification mark is a voluntary quality mark for the products and serves to inform and to assure the addressee of the certificate holder that the products meet the essential requirements.

Assessment Report

Report to the Certificate

Z10 11 01 67052 008

Software Tools for Safety Related Development

Simulink® Verification and Validation™ Simulink® Design Verifier™

Manufacturer:
The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760
USA

Report No.: MN66534C
Revision 1.1 dated Jan. 26, 2011

5.2 Usage considerations for development processes which need to comply with IEC 61508, ISO 26262, EN 50128, or derivative standards

The capabilities of Simulink® Verification and Validation™ and Simulink® Design Verifier™ listed in sections 2.1.2 and 2.2.2 respectively are certified for use in development processes which need to comply with IEC 61508, ISO 26262, EN 50128, or derivative standards. The two verification tools allow the automation of core verification and validation activities for Simulink models and generated code.
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!
# Process Compliance Demonstration

Annotated method tables with suggestions on how to use Model-Based Design processes and tools to apply the methods listed in ISO 26262-6

## Table 9 - Methods for Verification of Software Unit Design and Implementation

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>Applicable Model-Based Design Tools and Processes</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a Walkthrough</td>
<td>++</td>
<td>+</td>
<td>0</td>
<td>0</td>
<td>Simulink, Simulink Report Generator – Web View, System Design Description (SDD) report</td>
<td>Unit design walkthroughs can be based on a model, a generated Web View, or an SDD report.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Embedded Coder – Code generation report</td>
<td>Code walkthroughs can be based on HTML code generation reports or code generation reports with an integrated Web View of the model.</td>
</tr>
<tr>
<td>1b Inspection</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>Simulink, Simulink Report Generator – Web View, System Design Description (SDD) report</td>
<td>Unit design inspections can be based on a model, a generated Web View, or an SDD report.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Simulink Verification and Validation – Model Advisor checks</td>
<td>Unit design inspections can be supported by ISO 26262.</td>
</tr>
</tbody>
</table>

## Table 10 - Methods for Software Unit Testing

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>Applicable Model-Based Design Tools and Processes</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a Requirements-based test</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>Simulink Verification and Validation – Requirements Management Interface (RMI)</td>
<td>RMI can be used to establish bidirectional links between textual requirements and models.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IEC Certification Kit – Traceability matrix</td>
<td>Generated traceability matrices can be used to document and review existing links between textual requirements, models, and code.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Simulink – Signal Builder block</td>
<td>Signal Builder blocks can be used to create open-loop model tests.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Stateflow – Dynamic test vector charts</td>
<td>Dynamic test vector charts can be used to create closed-loop, reactive model tests.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Simulink Verification and Validation – Component testing capabilities</td>
<td>Component testing capabilities can be used to create model test harnesses. They also enable a requirements pane in the Signal Builder that can be used to link tests with textual requirements.</td>
</tr>
</tbody>
</table>
ISO 26262 – Walk-through and Inspection

ISO 26262 (2011)

Table 9 — Methods for the verification of software unit design and implementation

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1a Walk-through&lt;sup&gt;a&lt;/sup&gt;</td>
<td>++</td>
</tr>
<tr>
<td>1b Inspection&lt;sup&gt;a&lt;/sup&gt;</td>
<td>+</td>
</tr>
<tr>
<td>1c Semi-formal verification</td>
<td>+</td>
</tr>
<tr>
<td>1d Formal verification</td>
<td>o</td>
</tr>
<tr>
<td>1e Control flow analysis&lt;sup&gt;b,c&lt;/sup&gt;</td>
<td>o</td>
</tr>
<tr>
<td>1f Data flow analysis&lt;sup&gt;b,c&lt;/sup&gt;</td>
<td>+</td>
</tr>
<tr>
<td>1g Static code analysis</td>
<td>+</td>
</tr>
<tr>
<td>1h Semantic code analysis&lt;sup&gt;d&lt;/sup&gt;</td>
<td>+</td>
</tr>
</tbody>
</table>

<sup>a</sup> In the case of model-based software development the software unit specification design and implementation can be verified at the model level.

<sup>b</sup> Methods 1e and 1f can be applied at the source code level. These methods are applicable both to manual code development and to model-based development.

<sup>c</sup> Methods 1e and 1f can be part of methods 1d, 1g or 1h.

<sup>d</sup> Method 1h is used for mathematical analysis of source code by use of an abstract representation of possible values for the variables. For this it is not necessary to translate and execute the source code.

NOTE Table 9 lists only static verification techniques. Dynamic verification techniques (e.g. testing techniques) are covered in Tables 10, 11 and 12.

"[..] unit specification design and implementation can be verified at the model level."
Qualification of Model-Based Design Tools

☑ pre-qualified for all ASILs according to ISO 26262
## Defects detected by Polyspace Bug Finder

### Numerical
- Integer division by zero
- Float division by zero
- Integer conversion overflow
- Unsigned integer conversion overflow
- Sign change integer conversion overflow
- Float conversion integer overflow
- Float conversion overflow
- Integer overflow
- Unsigned integer overflow
- Float overflow
- Invalid use of std. library integer routine
- Invalid use of std. library float routine
- Shift of a negative value
- Shift operation overflow

### Static memory
- Array access out of bounds
- Null pointer
- Pointer access out of bounds
- Uninitialized cast of function pointer
- Uninitialized cast of pointer
- Invalid use of std. library memory routine
- Invalid use of std. library string routine
- Arithmetic operation with NULL pointer
- Wrong allocated object size for cast
- Local address escape
- Buffer overflow from incorrect string format specifier

### Dynamic memory
- Use of previously freed pointer
- Unprotected dynamic memory allocation
- Release of previously deallocated pointer
- Invalid free of pointer
- Memory leak

### Programming
- Invalid use of = operator
- Invalid use of == operator
- Declaration mismatch
- Invalid use of floating point operation
- Wrap-around on unsigned integer
- Assertion
- Invalid use of other standard lib. routine
- Missing null in string array
- Qualifier removed in conversion
- Wrong type used in sizeof
- Format string specifiers and arguments mismatch
- Writing to const qualified object

### Dataflow
- Write without further read
- Non-initialized variable
- Non-initialized pointer
- Variable shadowing
- Missing or invalid return statement
- Dead code
- Partially access array
- Uncalled function
- Pointer to non initialized value converted to const pointer
- Code deactivated by constant false condition
- Unreachable code
- Useless if

### Concurrency
- Race condition (atomic / non atomic)
- Missing lock/unlock
- Deadlock
- Double lock/unlock

### Resources management
- Use of previously closed resource
- Resource leak
- Writing to read-only resource

### Security / tainted data (38 checkers)
- chroot misuse
- Use of dangerous standard function
- Mismatch between data length and size
- Use of non-secure temporary file
- Sensitive heap not cleared
- Tainted array index
- Tainted external command
- Tainted integer division

### Object oriented (12 checkers)
- Object slicing
- Copy constructor not called
- Member not initialized in constructor
- ...

## Good practice
- Large pass-by-value argument
- More than one statement
- Delete of void pointer
- Hard-coded buffer size / loop boundary
- Unused parameter
## Defects detected by Polyspace Bug Finder

### Numerical
- Integer division by zero
- Float division by zero
- Integer conversion overflow
- Unsigned integer conversion overflow
- Sign change integer conversion overflow
- Float conversion overflow
- Integer overflow
- Unsigned integer overflow
- Float overflow
- Invalid use of std. library integer routine
- Invalid use of std. library float routine
- Shift of a negative value
- Shift operation overflow

### Static memory
- Array access out of bounds
- Null pointer
- Pointer access out of bounds
- Unreliable cast of function pointer
- Unreliable cast of pointer
- Invalid use of std. library memory routine
- Invalid use of std. library string routine
- Arithmetic operation with NULL pointer
- Wrong allocated object size for cast
- Local address escape
- Buffer overflow from incorrect string format/specifier

### Dynamic memory
- Use of previously freed pointer
- Unprotected dynamic memory allocation
- Release of previously deallocated pointer
- Invalid free of pointer
- Memory leak

### Programming
- Invalid use of = operator
- Invalid use of == operator
- Declaration mismatch
- Invalid use of floating point operation
- Wrap-around on unsigned integer
- Assertion
- Invalid use of other standard lib. routine
- Missing null in string array
- Qualifier removed in conversion
- Wrong type used in sizeof
- Format string specifiers and arguments mismatch
- Writing to const qualified object

### Dataflow
- Write without further read
- Uninitialized variable
- Non-initialized pointer
- Variable shadowing
- Missing or invalid return statement
- Dead code
- Partially access array
- Uncalled function
- Pointer to non initialized value converted to const pointer
- Code deactivated by constant false condition
- Unreachable code
- Useless if

### Concurrency
- Race condition (atomic / non atomic)
- Missing lock/unlock
- Deadlock
- Double lock/unlock

### Resources management
- Use of previously closed resource
- Resource leak
- Writing to read-only resource

### Security / tainted data (38 checkers)
- chroot misuse
- Use of dangerous standard function
- Mismatch between data length and size
- Use of non-secure temporary file
- Sensitive heap not cleared
- Tainted array index
- Tainted external command
- Tainted integer division

### Object oriented (12 checkers)
- Object slicing
- Copy constructor not called
- Member not initialized in constructor
- ...

### Good practice
- Large pass-by-value argument
- More than one statement
- Delete of void pointer
- Hard-coded buffer size / loop boundary
- Unused parameter

### Checkers inherited from Code Prover
Full list of run-time error checks in Polyspace Code Prover

**C run-time checks**
- Unreachable Code
- Out of Bounds Array Index
- Division by Zero
- Non-Initialized Variable
- Scalar and Float Overflow (left shift on signed variables, float underflow versus values near zero)
- Initialized Return Value
- Shift Operations (shift amount in 0..31/0..63, left operand of left shift is negative)
- Illegal Dereferenced Pointer (illegal pointer access to variable of structure field, pointer within bounds)
- Correctness Condition (array conversion must not extend range, function pointer does not point to a valid function)
- Non-Initialized Pointer
- User Assertion
- Non-Termination of Call (non-termination of calls and loops, arithmetic expressions)
- Known Non-Termination of Call
- Non-Termination of Loop
- Standard Library Function Call
- Absolute Address
- Inspection Points

**C++ run-time checks**
- Unreachable Code
- Out of Bounds Array Index
- Division by Zero
- Non-Initialized Variable
- Scalar and Float Overflow
- Shift Operations
- Pointer of function Not Null
- Function Returns a Value
- Illegal Dereferenced Pointer
- Correctness Condition
- Non-Initialized Pointer
- Exception Handling (calls to throws, destructor or delete throws, main/tasks/C_lib_func throws, exception raised is not specified in the throw list, throw during catch parameter construction, continue execution in __except)
- User Assertion
- Object Oriented Programming (invalid pointer to member, call of pure virtual function, incorrect type for this-pointer)
- Non-Termination of Call
- Non Termination of Loop
- Absolute Address
- Potential Call
- C++ Specific Checks (positive array size, incorrect typeid argument, incorrect dynamic_cast on reference)
Examples of software safety requirements

<table>
<thead>
<tr>
<th>Topics</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1a Enforcement of low complexity</td>
<td>++</td>
</tr>
<tr>
<td>1b Use of language subsets</td>
<td>++</td>
</tr>
<tr>
<td>1c Enforcement of strong typing</td>
<td>++</td>
</tr>
<tr>
<td>1d Use of defensive implementation techniques</td>
<td>+</td>
</tr>
<tr>
<td>1e Use of established design principles</td>
<td>+</td>
</tr>
<tr>
<td>1f Use of unambiguous graphical representation</td>
<td>+</td>
</tr>
<tr>
<td>1g Use of style guides</td>
<td>+</td>
</tr>
<tr>
<td>1h Use of naming conventions</td>
<td>++</td>
</tr>
</tbody>
</table>

Table 1 – Topics to be covered by modelling and coding guidelines

Detect and correct areas with high complexity

Polyspace Metrics
Examples of software safety requirements

<table>
<thead>
<tr>
<th>Topics</th>
<th>ASIL</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initiation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Architecture</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design/Implementation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unit Testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integration Testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Topics</th>
<th>ASIL</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a Enforcement of low complexity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1b Use of language subsets</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1c Enforcement of strong typing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1d Use of defensive implementation techniques</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1e Use of established design principles</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1f Use of unambiguous graphical representation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1g Use of style guides</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1h Use of naming conventions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1 – Topics to be covered by modelling and coding guidelines

Enforcement of coding guidelines

<table>
<thead>
<tr>
<th>Verification</th>
<th>Coding Rules</th>
<th>Software Quality Objectives</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Confirmed Defects</td>
<td>Justified</td>
</tr>
<tr>
<td>concurrency.c</td>
<td>0.0%</td>
<td>464</td>
</tr>
<tr>
<td>datatow.c</td>
<td>0.0%</td>
<td>58</td>
</tr>
<tr>
<td>dynamicmemory.c</td>
<td>0.0%</td>
<td>101</td>
</tr>
<tr>
<td>numperio.c</td>
<td>0.0%</td>
<td>99</td>
</tr>
<tr>
<td>other.c</td>
<td>0.0%</td>
<td>27</td>
</tr>
<tr>
<td>programming.c</td>
<td>0.0%</td>
<td>80</td>
</tr>
<tr>
<td>stationemory.c</td>
<td>0.0%</td>
<td>48</td>
</tr>
</tbody>
</table>

Polyspace Metrics
Reference Workflow

**Model-Based Design for ISO 26262**

- **Modeling standards checking** *(Simulink V&V)*
- **Simulation / model testing** *(Simulink, Simulink Test)*
  - Model coverage
  - Req. Mgmt. Int. *(Simulink V&V)*
- **Module and integration testing at the model level**
- **Review and static analysis at the model level**
- **Model used for production code generation**
  - Code generation
  - Compilation and linking
  - Embedded Coder
  - Third-party tool
- **Generated C code**
- **Object code**
- **PIL test (Embedded Coder)**
- **Traceability matrix analysis** *(IEC Certification Kit)* or model vs. code coverage *(Simulink V&V)*
- **Equivalence testing**
- **Prevention of unintended functionality**

**Tools**
- Simulink/Stateflow
- Embedded Coder
- Third-party tool
### Advanced Reference Workflow

#### Additional Best Practices

- **Modeling standards checking** (Simulink V&V)
- **Property Proving, Design Error Detection** (Simulink Design Verifier)
- **Simulation / model testing** (Simulink, Simulink Test)
- **Model coverage** (Simulink V&V)
- **Equivalence testing**
- **Prevention of unintended functionality**
- **Traceability matrix analysis** (IEC Certification Kit) or model vs. code coverage (Simulink V&V)
- **MISRA-C checking** (Polyspace products)
- **Review and static analysis at the model level**
- **Module and integration testing at the model level**
- **Run-time error detection** (Polyspace products)
- **Property Proving, Design Error Detection** (Simulink Design Verifier)
- **Test generation** (Simulink Design Verifier)
- **PIL test** (Embedded Coder)
- **Test generation** (Simulink Design Verifier)
- **Model used for production code generation**
- **Generated C code**
- **Object code**
- **Textual requirements**
- **Executable specification**
- **Modeling** (Simulink/Stateflow)
- **Code generation**
- **Compilation and linking**
- **Third-party tool**
- **Embedded Coder**
Workflow Summary: Complementary Model & Code Verification

- Check s-function code for run-time errors
- Check MISRA compliance (Polyspace)
- Check integrated code for run-time errors
Model-Based Design
Verification and Validation

Requirements Traceability
• Navigation and Reports

Modeling Standards Checking
• Automatic Review

Testing and Test Automation
• Test Authoring
• Coverage Analysis
• Automatic Test Case Generation

Property Proving
• Quickly find incorrect behavior
• Prove correctness of behavior

Static Analysis on Model and Code
• Identify defects
• Prove absence of runtime errors
• Compliance to coding standards

Compliance to ISO 26262
• Workflow and tool qualification
Verification on Code Level – Polyspace

- Finds bugs
- Checks coding rule conformance (MISRA/JSF/Custom)
- Provides metrics (Cyclomatic complexity etc.)
- Proves the existence and absence of errors
- Indicates when you’ve reached the desired quality level