MATLAB EXPO

# DO Qualification Kit와 모델기반 설계를 이용한 항공 Software 개발

*이승현, 현대자동차*

# Introduction to Organization and Business

## Hyundai Motors Company AAM Division

- Advanced Air Mobility is new concept in aviation
- AAM ⊂ UAM(Urban) + RAM(Regional)
- Benefits of AAM
  - **Reduced commute times:** significantly reduce commute times, especially in congested urban areas
  - **Increased access to rural areas**: make it easier for people to access rural areas, which are often underserved by traditional transportation options
  - **Improved air quality:** electric propulsion system, so they produce zero emissions
  - **Increased safety:** expected to be safer than traditional aircraft, thanks to their advanced safety features and autonomous flight systems

# Overview

## Development of Avionics SW using DO Qualification Kit and Model Based Design

- What DO-178C/331 Standards
- Why Model Based Design
- DO Qualification Kit
- SW Development Infrastructure

- Process with DO Qualification Kit and MBD
  - Requirement
  - Model Architecture
  - Model Static Analysis
  - Model Dynamics Analysis
  - Code Generation
  - Code Verification
  - SIL/PIL Test
  - Tool Qualification
  - Strategy of Continuous Integration/Deployment

# DO-178C/331 Standards

DO-178C : Software Considerations Airborne Systems and Equipment Certifications

DO-331 : Model-Based Development and Verification Supplement to DO-178C

- RTCA published : RTCA(Radio Technical Commission for Aeronautics) is a United States non-profit organization that develops technical guidance for use by government regulatory authorities and by industry
- Primary document by which the certification authorities such as FAA, EASA to approve all commercial software-based aerospace systems
- Guideline = Process + Objective + Activity + Output
    - each process presents the objective to be achieved. as the process, it presents proper activities to achieve this objective

Federal Aviation Administration

**5.2 Software Design Process** — Process

The high-level requirements are refined through one or more iterations in the software design process to develop the software architecture and the low-level requirements that can be used to implement Source Code.

**5.2.1 Software Design Process Objectives** — Objective

The objectives of the software design process are:

a. The software architecture and low-level requirements are developed from the high-level requirements.

b. Derived low-level requirements are defined and provided to the system processes, including the system safety assessment process.

**5.2.2 Software Design Process Activities** — Activity

The software design process inputs are the Software Requirements Data, the Software Development Plan, and the Software Design Standards. When the planned transition criteria have been satisfied, the high-level requirements are used in the design process to develop software architecture and low-level requirements. This may involve one or more lower levels of requirements.

Output — The primary output of the process is the Design Description (see 11.10) which includes the software architecture and the low-level requirements.

# DO-178C/331 Standards

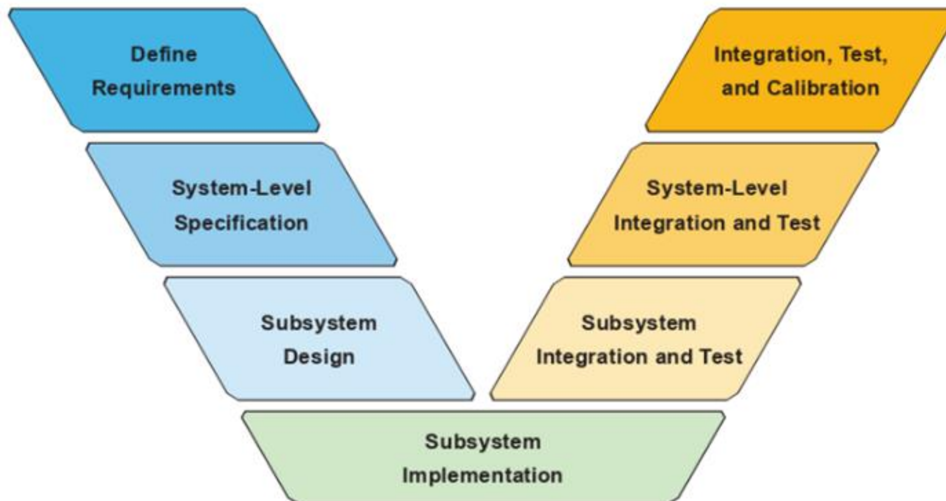
- The software level of a software component, also known as the Design Assurance Level(DAL), is based upon the contribution of software to potential failure conditions as determined by the system safety assessment process
- As the software level, Objective to accomplish is different ( Level A : MC/DC 100% )

| Software Level | Failure condition |
| --- | --- |
| Level A | **Catastrophic -** Failure may cause deaths, usually with loss of the airplane |
| Level B | **Hazardous -** Failure has a large negative impact on safety or performance, or reduces the ability of the crew to operate the aircraft due to physical distress or a higher workload, or causes serious or fatal injuries among the passengers. |
| Level C | **Major -** Failure significantly reduces the safety margin or significantly increases crew workload. May result in passenger discomfort (or even minor injuries). |
| Level D | **Minor -** Failure slightly reduces the safety margin or slightly increases crew workload. Examples might include causing passenger inconvenience or a routine flight plan change. |

**Table A-7 Verification of Verification Process Results**

| | Objective | | Activity | Applicability by Software Level | | | | Output | | Control Category by Software Level | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | Description | Ref | Ref | A | B | C | D | Data Item | Ref | A | B | C | D |
| 1 | Test procedures are correct. | 6.4.5.b | 6.4.5 | ● | ○ | ○ | | Software Verification Results | 11.14 | ② | ② | ② | |
| 2 | Test results are correct and discrepancies explained. | 6.4.5.c | 6.4.5 | ● | ○ | ○ | | Software Verification Results | 11.14 | ② | ② | ② | |
| 3 | Test coverage of high-level requirements is achieved. | 6.4.4.a | 6.4.4.1 | ● | ○ | ○ | ○ | Software Verification Results | 11.14 | ② | ② | ② | ② |
| 4 | Test coverage of low-level requirements is achieved. | 6.4.4.b | 6.4.4.1 | ● | ○ | ○ | | Software Verification Results | 11.14 | ② | ② | ② | |
| 5 | Test coverage of software structure (modified condition/decision coverage) is achieved. | 6.4.4.c | 6.4.4.2.a<br>6.4.4.2.b<br>6.4.4.2.d<br>6.4.4.3 | ● | | | | Software Verification Results | 11.14 | ② | | | |
| 6 | Test coverage of software structure (decision coverage) is achieved. | 6.4.4.c | 6.4.4.2.a<br>6.4.4.2.b<br>6.4.4.2.d<br>6.4.4.3 | ● | ● | | | Software Verification Results | 11.14 | ② | ② | | |
| 7 | Test coverage of software structure (statement coverage) is achieved. | 6.4.4.c | 6.4.4.2.a<br>6.4.4.2.b<br>6.4.4.2.d<br>6.4.4.3 | ● | ● | ○ | | Software Verification Results | 11.14 | ② | ② | ② | |

# Why Model Based Design - Traditional design flow




Traditional design flow

**Traditional design flow**

1. Requirement and Specifications – Systems engineers define functional requirements and interface specifications on different components of the design.

2. Design - Design engineers, sometimes working in many separate teams, create models of the system components

3. Implementation - The models created by the design engineers are realized in the physical world. This includes software engineers writing embeddable code for algorithms created by the design engineers

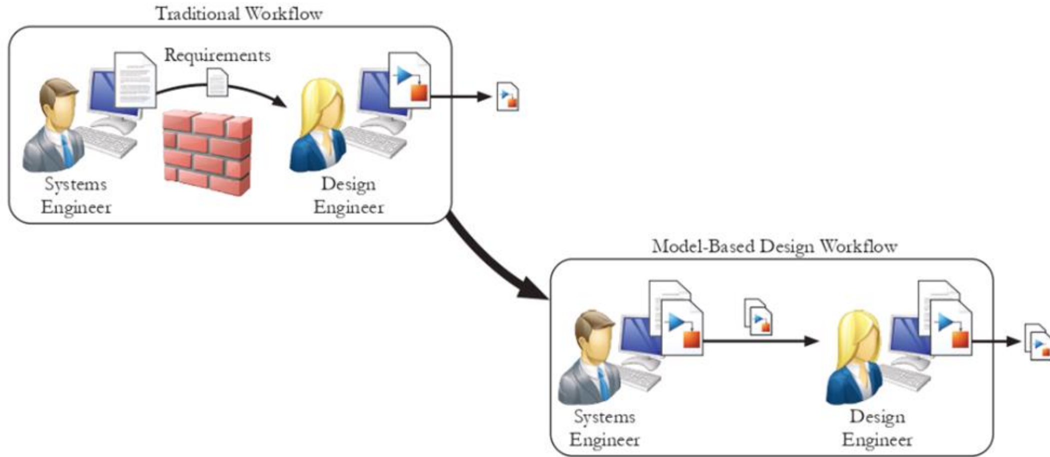
4. Test and Verification - The different system components are tested, as is the overall design, to ensure their function as expected

**Some common issues**

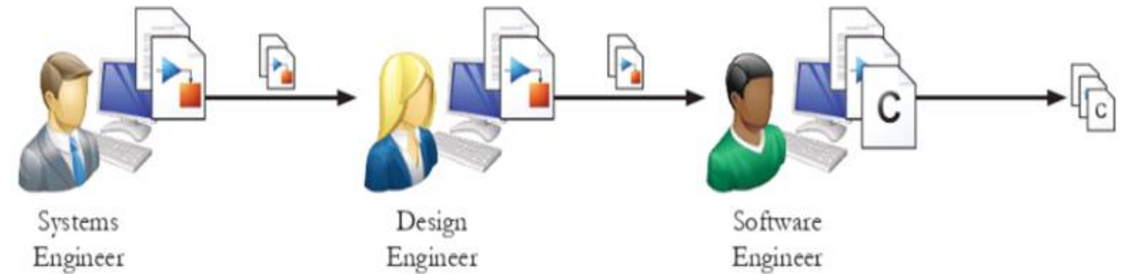
- Communication between teams
  - Design and interface specifications are commonly in a written document that must be read and understood by design engineers
- Recreation of work
  - Software engineers might have to rewrite an algorithm that a design engineer has been using
- Problems found late in design process
  - If a problem exists due to integration of components, it is not discovered until the testing phase of the design cycle

# Why Model Based Design - 4 key features

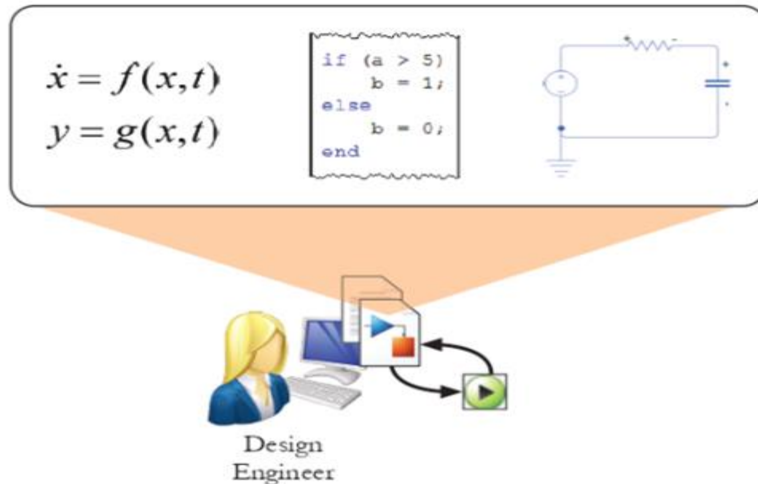
## Executable Specification from Models



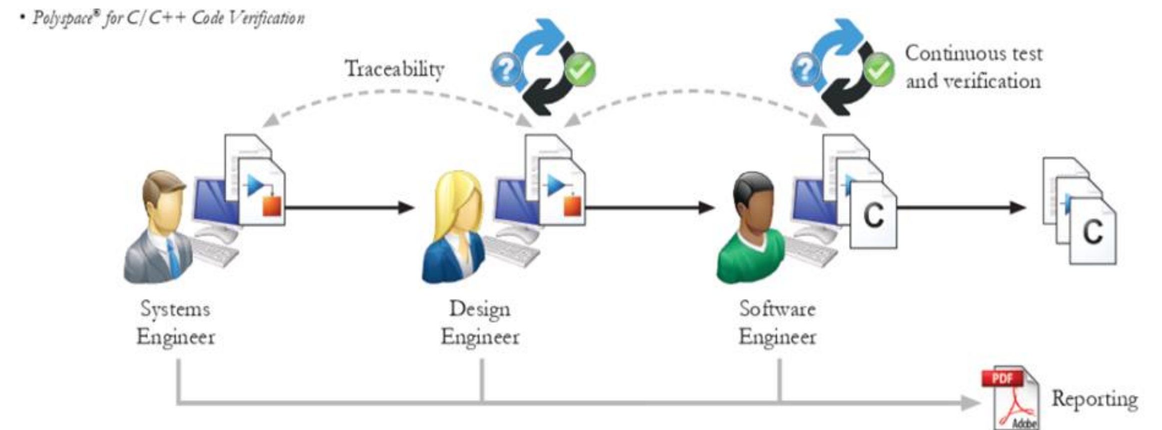

## Implementation with Automatic Code Generation




## Design with Simulation

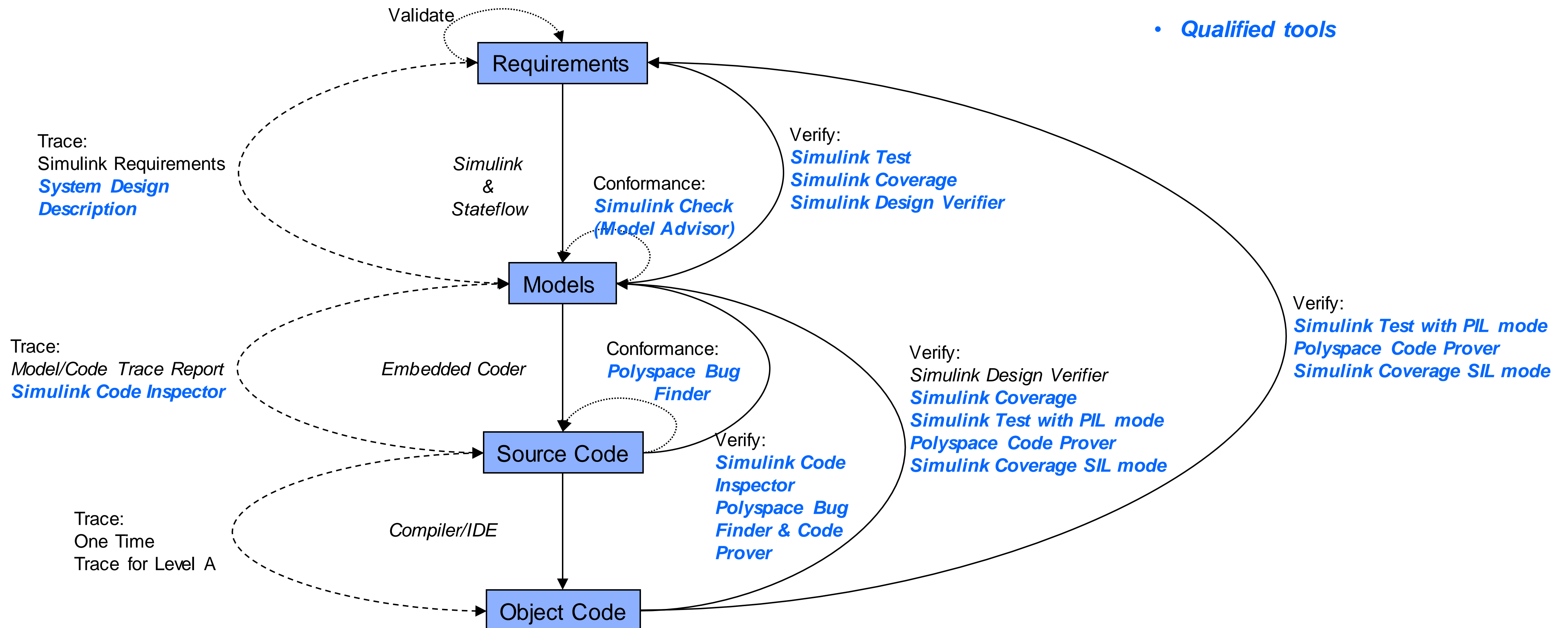



## Continuous Test and Verification



# DO Qualification Kit

- DO Qualification Kit = Qualified tools + 3 elements for Tool Qualification(documentation, test cases, procedures)
- Tool qualification have to be done every project, and an error in the tool may have a negative impact on software functionality if the tool inadequately performs its intended functions. In order to avoid this risk and to ensure the integrity of the tool functionality. the tool should be developed and verified using adequate processed



# SW Development Infrastructure

implemented
planned

**ALM**

codebeamer

Requirement Management

Issue & Risk Management

Development Support

Test Management

Change Management

Build & Release Management

**link via ReqIF**

Requirement Editor

Jira

Confluence

MATLAB & SIMULINK

source control

GitLab

**trace to design**

**trace to test case**

model static/dynamics analysis

Jenkins

code verification

request to verification

request to release

POLYSPACE

# Requirement - Adequate High-Level Requirement

- In DO-331 Model Based Design, definition about two type requirements

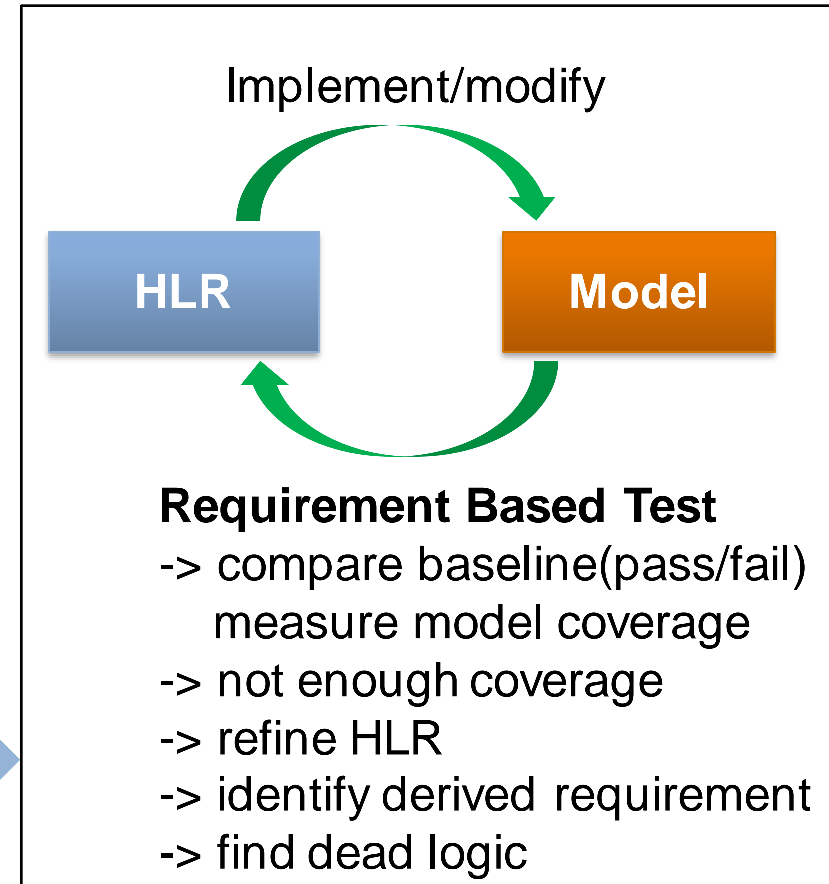
For the purpose of this supplement, the term "high-level requirement" refers to either of the following:

- Any requirement contained in a Specification Model.
- Any requirement from which a Design Model is developed. → **HLR**

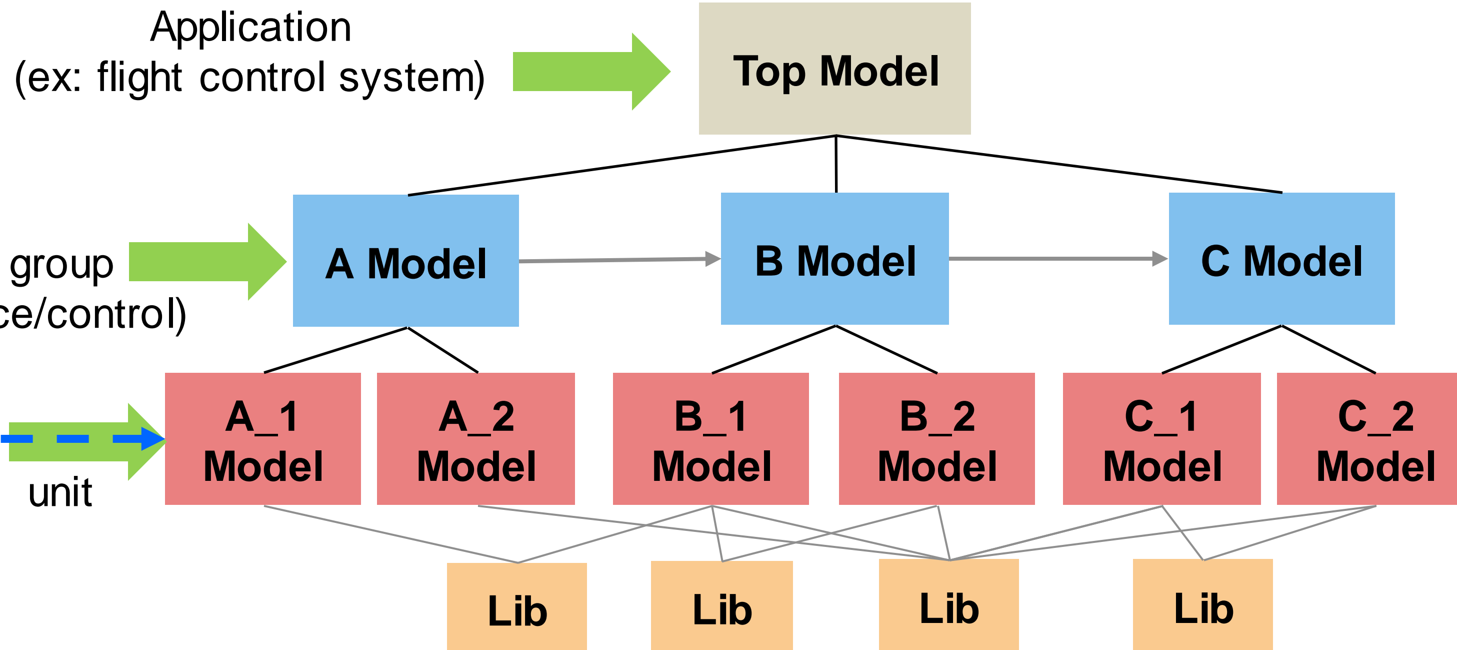
For the purpose of this supplement, the term "low-level requirement" refers to:

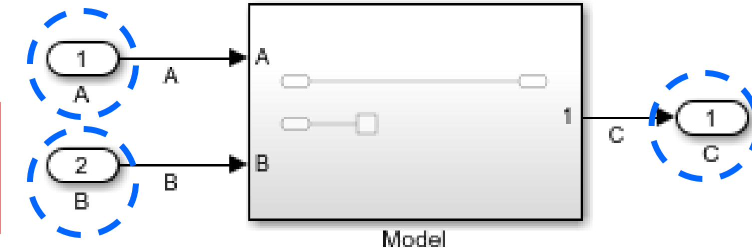
- Any requirement contained in a Design Model → **LLR = Design Model**

- Condition of ideal HLR
  - Aspect of developer : is this HLR adequate to implement model?
  - Aspect of verifier : is this HLR adequate to extract test vector?
- It is difficult to write adequate HLR at once
  - It need to be completed through the iteration
- DO-331 standard require to make the document
  - SRStd(Software Requirement Standard)



# Requirement – Model Hierarchy and Requirement Format



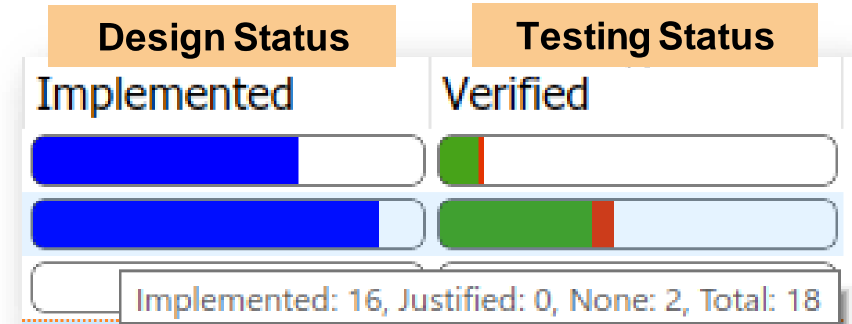




Port name correspond to requirement

- Format : (function group) (shall) (process) (object) (details) (condition)
- Example : (A Model) (shall) (change) (the operation mode) (according to the system mode and the previous operation mode) (If the system mode input is not defined, the operation mode shall maintain the previous mode)
- Requirements should be written about the functions that the function group should have. and then developer have to Implement unit models based on these requirements from its function group
- It should be clearly written to identify input and output name and the unit model should be designed using same name
- It should be written about interface requirement including min/max range, data type, unit and resolution

# Requirement - Link design and tests

- Analyzes the traceability to identify gaps in implementation or testing
- Author requirements in CodeBeamer(ALM)
- Export via ReqIF file
- Simulink Requirements Import from ReqIF file
- Link requirements to design and tests



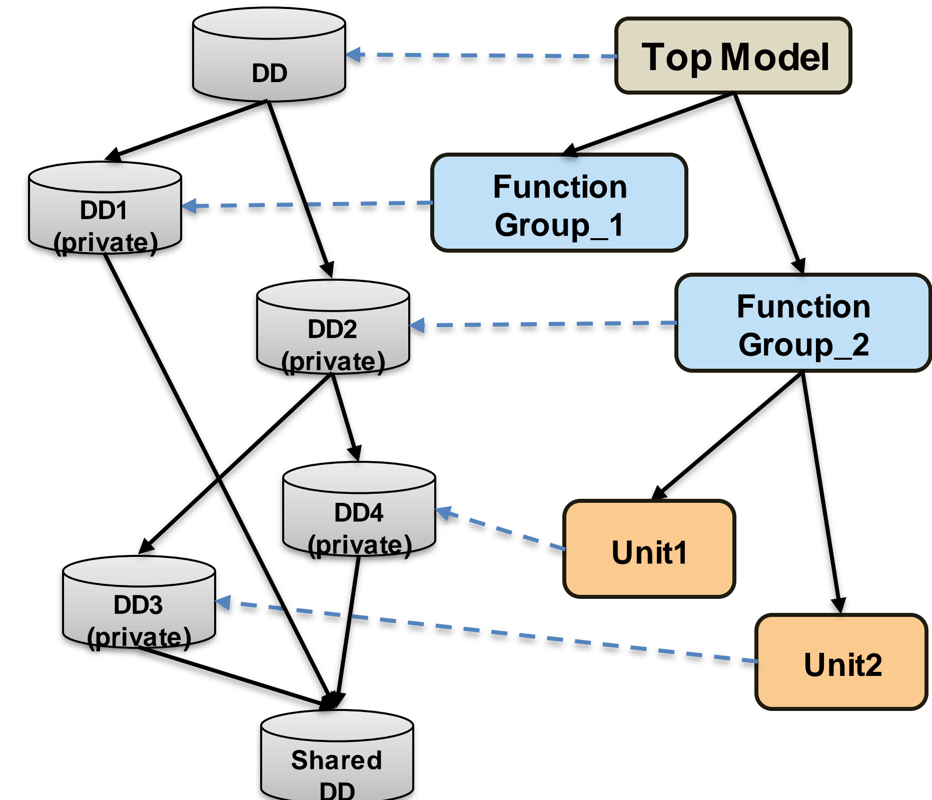
# Model Architecture – Concurrent Development within team

- For Concurrent Development individually within the team

**Simulink Project** + **Data Dictionary**

**key Considerations**

- Componentization
  - Model (function group model, unit model)
  - Data Dictionary (private, shared)
  - Requirement
- Commonization
  - Which model to make Library
  - Shared interfaces (bus signal)
- Common environment management
  - How to organize Simulink Projects with referencing projects
  - How to manage code-gen and cache folders
  - How to make collaborative workflow on a project



# Model Architecture – Simulink Project + Data Dictionary

- Why use Simulink Project
  - Projects can promote more efficient team work and individual productivity by helping you
  - Find all the files that belong with your project
  - Create standard ways to initialize and shut down a project (*.mat, DB )
  - Create, store, and easily access common operations
  - View and label modified files for peer review workflows
  - Share projects using built-in integration with Git™, external source control tools
  - Important file and folder organization

- Why use Data Dictionary
  - Data organization
    - Tight Connection to Simulink model
    - Separate model's global design data from other data
    - Partition and share data via referenced dictionaries
  - Change tracking workflow
    - Change detection, last modified
    - Integration with file-based CM system and Simulink Projects

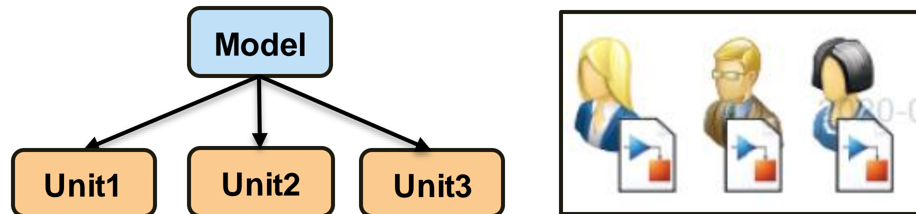
| Name | Label |
|---|---|
| DO_01_Planning | |
| checklists | MB.11.3 SVP |
| plans | |
| standards | |
| DO_02_Requirements | |
| specification | MB.11.21 TD, MB.11.9 S... |
| verification_results | MB.11.14 SVR, MB.11.2... |
| DO_03_Design | |
| Actuator_Control | |
| specification | MB.11.10 SDD, MB.11.2... |
| data | |
| DD_Actuator_Control.sldd | |
| localDD_Actuator_Control.m | |
| documents | |
| Actuator_Control.slmx | |
| Actuator_Control.slx | |
| open_Actuator_Control.m | |
| test_cases | MB.11.13 SVCP, MB.11.... |
| HLR | |
| LLR | |
| verification_results | MB.11.14 SVR, MB.11.2... |
| AHRS_Voter | |
| common | |
| Flight_Control | |
| InnerLoop_Control | |
| OuterLoop_Control | |
| sample_model | |
| DO_04_Code | |
| specification | MB.11.11 SC, MB.11.12 ... |
| verification_results | MB.11.14 SVR, MB.11.2... |
| DO_05_Artifacts | |
| MB_11_15_SECI | MB.11.15 SECI |
| MB_11_16_SCI | MB.11.16 SCI |
| MB_11_17_PR | MB.11.17 PR |
| MB_11_18_SCMR | MB.11.18 SCMR |
| MB_11_19_SQAR | MB.11.19 SQAR |
| MB_11_20_SAS | MB.11.20 SAS |
| MB_11_22_PDIF | MB.11.22 PDIF |
| DO_06_ToolQualification | |

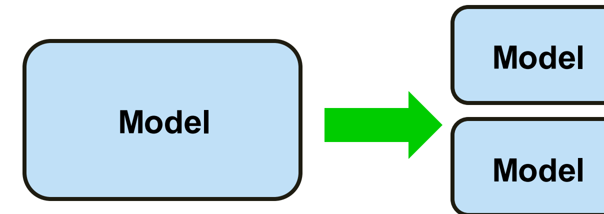
Best practice file/folder Organization
(Helicopter Flight Control MBD Example)

# Model Architecture – Considerations of Unit-level Model

## Team collaboration

If the model have to be designed from 3 engineers
It should be partitioned to Unit



## Testability

The smaller the model, the more easy the test
but if the model is more separated, model files and test case will be increased

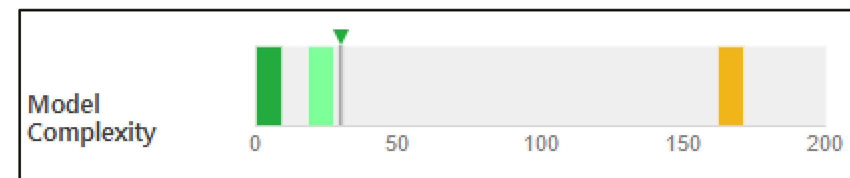
Model → Model, Model

## Reusability

The model that can be used in multiple places throughout an integration model should be Unit

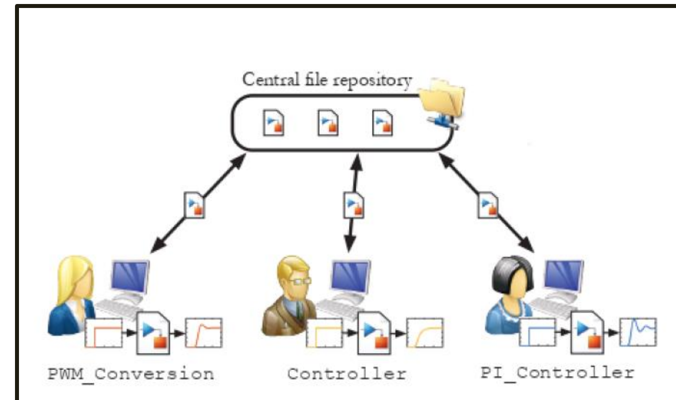
## Model Complexity

If complexity of model is higher than the criteria, model needs to be partitioned to Unit



# Model Architecture - Model Reference Features

## Why Unit-Level Models should be Model Reference




- **Traceability** : model references are atomic, meaning they can provide traceability between the model and the generated code
- **Reusability** : referenced model is forced to be completely self-reliant so it is easy to reuse
- **Unit Testing** : referenced model behaves the same way during a standalone simulation as it does when referenced from a parent simulation
- **Source Control** : model is stored in a separate file, this allows for concurrent development, as well as the ability to independently keep separate versions of each component

  -> as the model has changed, it need less regression test.
- **Model Guideline about interface**: an important aspect of model reference is that they cannot propagate signal properties across their boundaries. This is different from subsystems, which do allow signal property propagation.

  -> As DO-331 standard, interface of unit models must be defined by specific properties. If the wrong data type is transmitted through signal propagation in the previous model, it can be found. but subsystem can’t

MATLAB EXPO

# Model Architecture - Modeling Standard

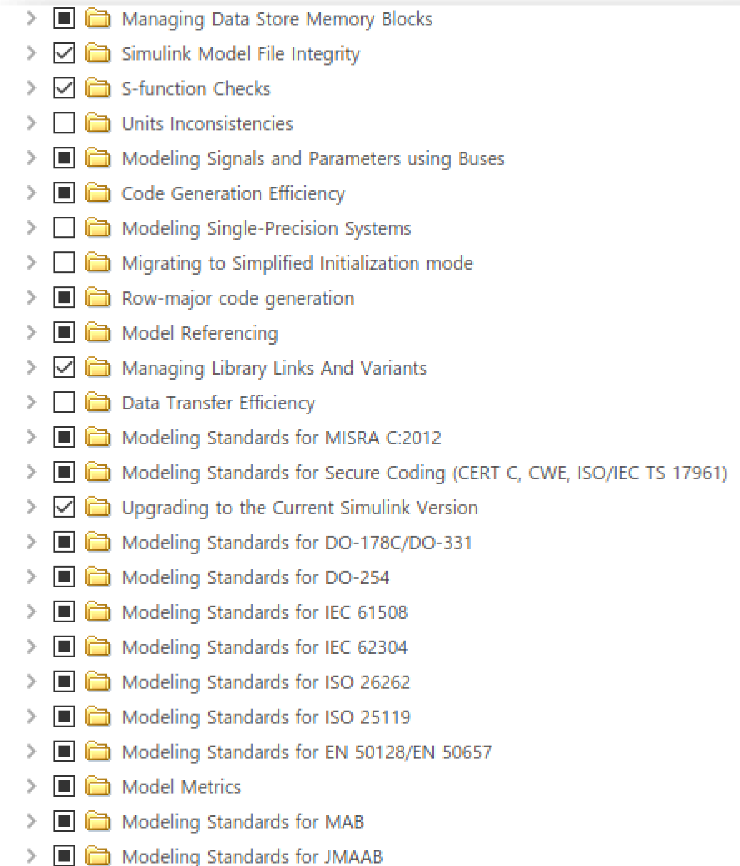
- Why need Modeling Standard : when multiple people works on the same project, you may find inconsistencies among their modeling styles. For increased consistency, you may find it helpful to enforce model standard on all models within a project

- **Performance**
Check under Code Generation Efficiency can help identify modeling constructs that decrease the efficiency of the generated code.

- **Accuracy**
There is a check under Simulation Runtime Accuracy Diagnostics that ensures data store memory read/write diagnostics are enabled

- **Predictability**
The Managing Library Links And Variants checks can find disabled library links in your model, thus helping to ensure that your library reference blocks behave as you expect

- **Consistency**
The Requirements Consistency Checking task helps ensure requirement links match up with the requirement documents



MATLAB EXPO

# Model Architecture – Example of Model Standard

## hisl_0025: Design min/max specification of input interfaces

| ID: Title | hisl_0025: Design min/max specification of input interfaces |
|---|---|
| Description | Provide design min/max information for root-level Inport blocks to specify the input interface ranges. |
| Notes | • Specifying the range of Inport blocks on the root level enables additional capabilities[a]Examples include:<br>- Detection of overflows through simulation range checking.<br>- Code optimizations using Embedded Coder®.<br>- Design model verification using Simulink Design Verifier™.<br>- Fixed-point autoscaling using Fixed-Point Designer™.<br>• Specified design ranges are used by Embedded Coder to optimize the generated code. To use these design ranges for optimization, select configuration parameter **Optimize using the specified minimum and maximum values**. This configuration parameter is applicable only when the ***System target file*** is an ERT-based target.<br>• Ranges for bus-type Inport blocks are specified with the bus elements of the defining bus object. Simulink ignores range specifications provided directly at Inport blocks that are bus-type. |
| Rationale | Support precise specification of the input interface. |
| Model Advisor Checks | Check for root Inports with missing range definitions (Simulink Check) |
| References | • DO-331 Section MB.6.3.1.b 'High-level requirements are accurate and consistent'<br>DO-331 Section MB.6.3.2.b 'Low-level requirements are accurate and consistent' |

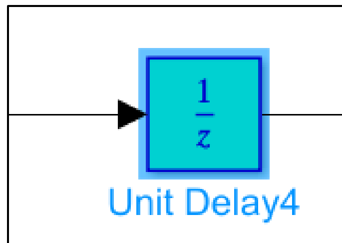
Referencing from DO-331 Standard

# Model Static Analysis - Simulink Checks(Model Advisor)

Conform

Model

- Improve the consistency, clarity, and readability of your models
- Identify model settings, blocks, and block parameters that affect simulation behavior or code generation




link

**Check usage of tunable parameters in blocks**

Identify tunable parameters used to specify expressions, data type conversions, or indexing operations.

Warning

The following blocks have tunable parameters that specify expressions, data type conversions, or indexing operations:

- flight_mode/Unit Delay4

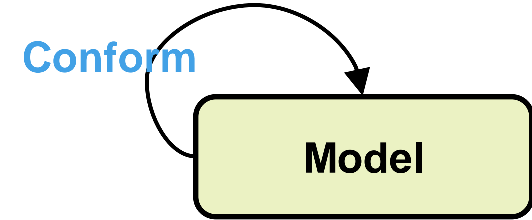
**Recommended Action**

Modify the use of the tunable parameters in one of the following methods:

- Perform the calculations using Simulink basic blocks or precompute the values.
- Use the Selector block to extract array entries.
- Use the Data Type Conversion block to change data types.

MATLAB EXPO

# Model Static Analysis – Example of MAB Guideline



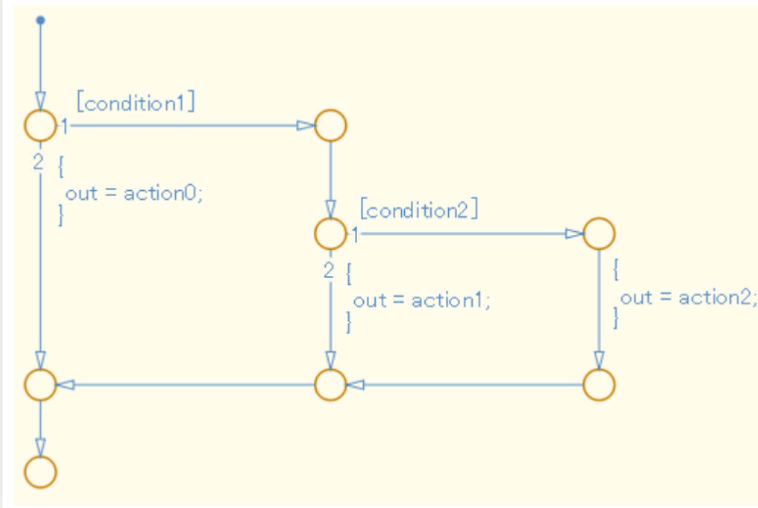

- Example of MAB guideline : db_0132: Transitions in flow charts

**Sub ID b**

In a flow chart, the condition shall be positioned on a horizontal transition line and the condition action shall be positioned on a vertical transition line.

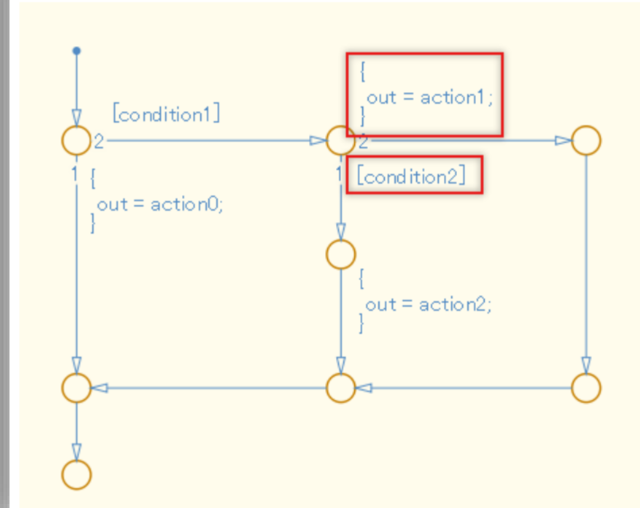
**Example — Correct**

The condition is positioned on a horizontal transition line and the condition action is on a vertical transition line.




**Example — Incorrect**

The condition is positioned on a vertical transition line and the condition action is on a horizontal transition line.




flow chart
→ Condition: Horizontal transition
→ Action: Vertical transition

MATLAB EXPO

# Model Static Analysis – Simulink Design Verifier

Conform

Model

- Objective : Design error detection
  - Dead logic
  - Integer or fixed-point data overflow
  - Division by zero
  - Out of bound array access
  - Data store access violations
  - Specified block input range violations

Web Browser - Simulink Design Verifier Report

Save Report: off

### Unsupported Blocks

The following blocks are not supported by Simulink Design Verifier. They were abstracted during the analysis. This can lead Simulink Design Verifier to produce only partial results for parts of the model that depends on the output values of these blocks.

| Block | Type |
|---|---|
| Sin | Trigonometry |

### Chapter 3. Dead Logic

Simulink Design Verifier proved these decisions and conditions to be unreachable or dead logic. This can be a side effect of parameter configurations or minimum and maximum constraints specified on inputs. Simulink Design Verifier ran a partial check for dead logic. Consider enabling the 'Dead logic > Run exhaustive analysis' configuration option in order to perform an exhaustive analysis.

| # | Type | Model Item | Description |
|---|---|---|---|
| 1 | Decision | hdot_cmd_slc/alt2hdot_cmd/-500~6000 | input > lower limit **can only be true** |

### Chapter 4. Design Error Detection Objectives Status

**Table of Contents**

Objectives Valid

### Objectives Valid

| # | Type | Model Item | Description | Analysis Time (sec) |
|---|---|---|---|---|
| 123 | Division by zero | hdot_cmd_slc/alt2hdot_cmd/SUMP_m7/Divide | Division by zero | 10 |

link

Simulink Design Verifier Results Summary: autopilot_mode_slc

| | |
|---|---|
| Progress | |
| Objectives processed | 87/87 |
| Valid | 1 |
| Falsified | 1 |
| Elapsed time | 0:19 |

Design error detection completed normally.

Simulink Design Verifier ran a partial check for dead logic. Consider enabling the 'Dead logic > Run exhaustive analysis' configuration option in order to perform an exhaustive analysis.

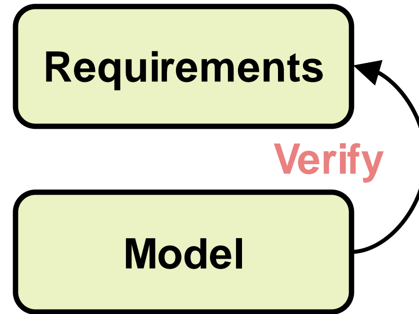
1/87 objective is valid
1/87 objective is dead logic

Results:

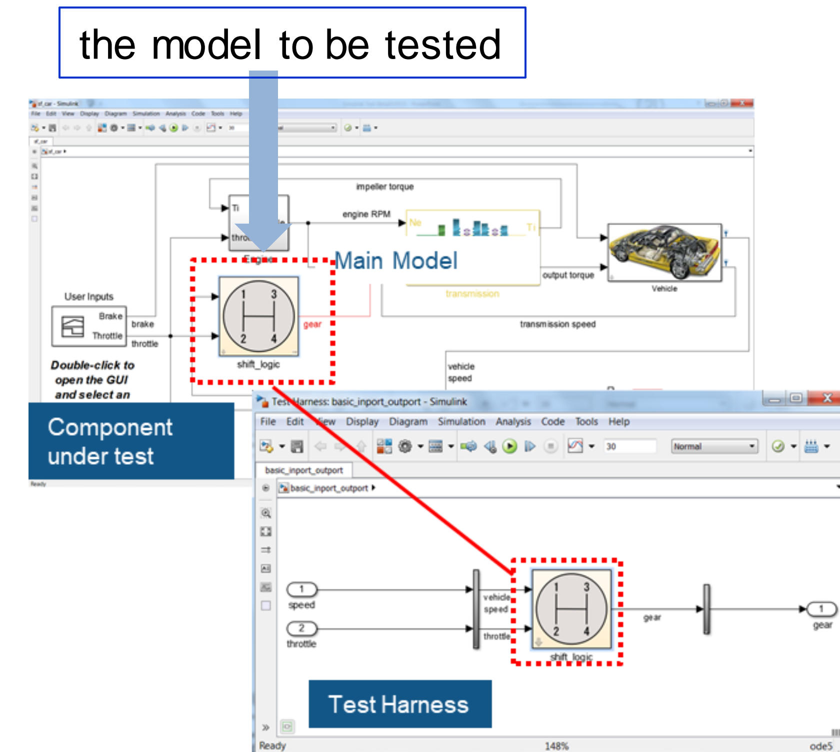
- Open filter viewer
- Highlight analysis results on model
- Detailed analysis report: (HTML) (PDF)

Data saved in: autopilot_mode_slc_sldvdata.mat
in folder: D:₩002. MBD(Mathworks)₩00. eCody_Baseline₩Design₩Modeling₩claws₩sldv_output₩autopilot_mode_slc

# Model Dynamics Analysis – Requirement Based Test




- Workflow of Requirement Based Test
  1. Analysis requirement
     - Identify requirement type ( related to time or logical)
     - Find input and output correspond to model (if enumeration type, find mapped numerical variables)
     - Estimate expected output and set the baseline (pass/fail)
     - if not enough, find addition documents(ex: SDD)
  2. Make test harness model
     - Choose & make adequate test input to get (test sequence, excel, etc.)
     - Define inputs and assessments based on logical, temporal conditions
  3. Run and evaluate test result via Test Manager
     - Author, execute, manage test suites(test cases)
     - Review, export, report
     - Collect model coverage



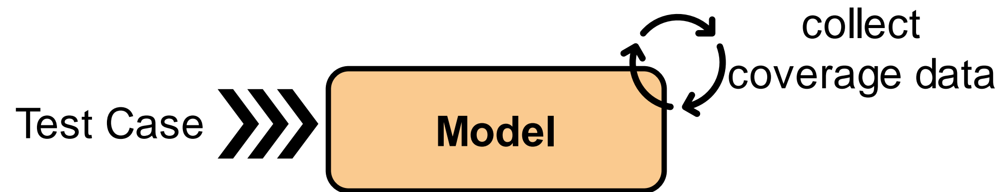
MATLAB EXPO

# Model Dynamics Analysis – Model Coverage

Requirements

Verify

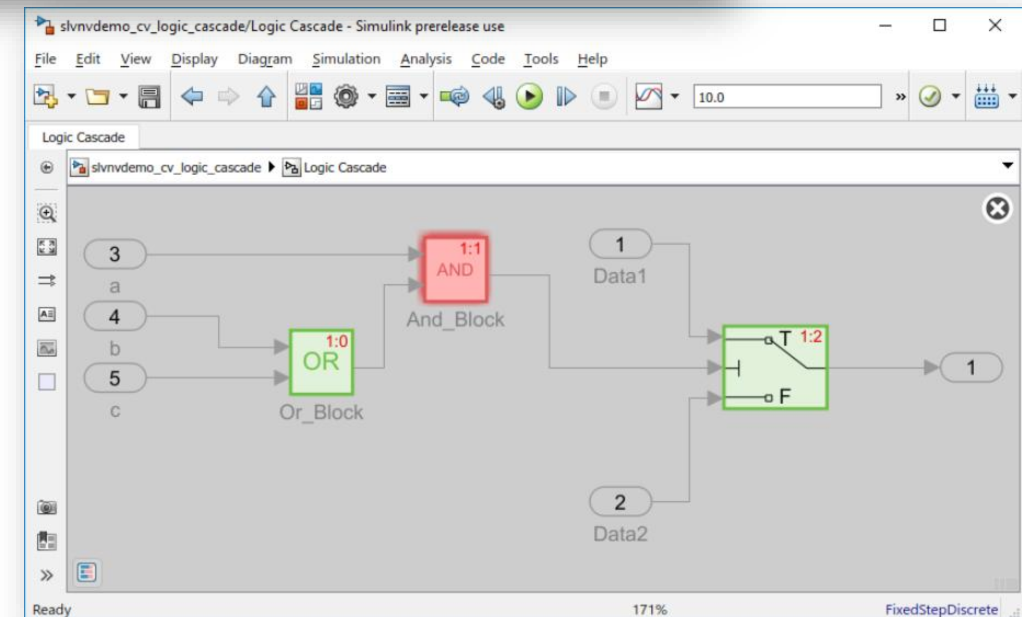
Model




**Summary**

| Model Hierarchy/Complexity | | Test 1 Decision | Condition | MCDC | Execution |
|---|---|---|---|---|---|
| 1. sldemo_clutch | 16 | 75% | 100% | 100% | 100% |
| 2. . . . Friction Model | | NA | NA | NA | 100% |
| 3. . . . Friction Mode Logic | 11 | 67% | 100% | 100% | 100% |
| 4. . . . . . . Break Apart Detection | 1 | 100% | 100% | NA | 100% |
| 5. . . . . . . Lockup Detection | 1 | 100% | 100% | 100% | 100% |
| 6. . . . . . . . . . Friction Calc | | NA | NA | NA | 100% |
| 7. . . . . . . . . . Required Friction for Lockup | 1 | 100% | 100% | NA | 100% |
| 8. . . . . . . Lockup FSM | 9 | 50% | 100% | NA | 100% |
| 9. . . . . . . Requisite Friction | | NA | NA | NA | 100% |
| 10. . . . Locked | 2 | 100% | NA | NA | 100% |
| 11. . . . Unlocked | 2 | 100% | NA | NA | 100% |

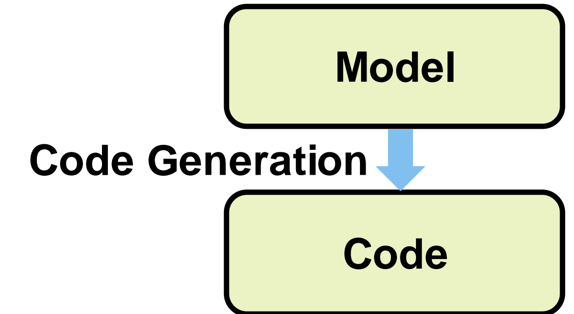
- Model coverage is indicator to confirm that sufficient tests have been performed
- DO178C/331 require to specific model coverage according to software level
- Measure model coverage, if not enough coverage
  - Reanalysis the requirement and extract more test vector
  - Identify derived requirement and extract test vector
  - Check dead logic and modify the model

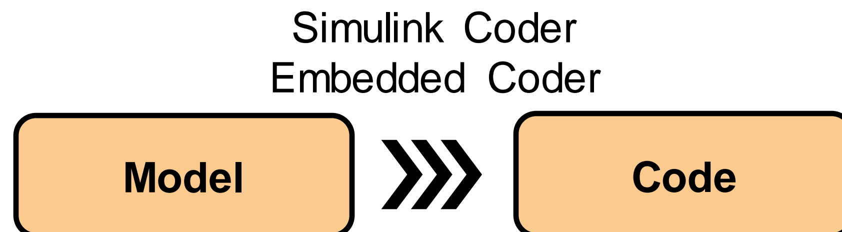


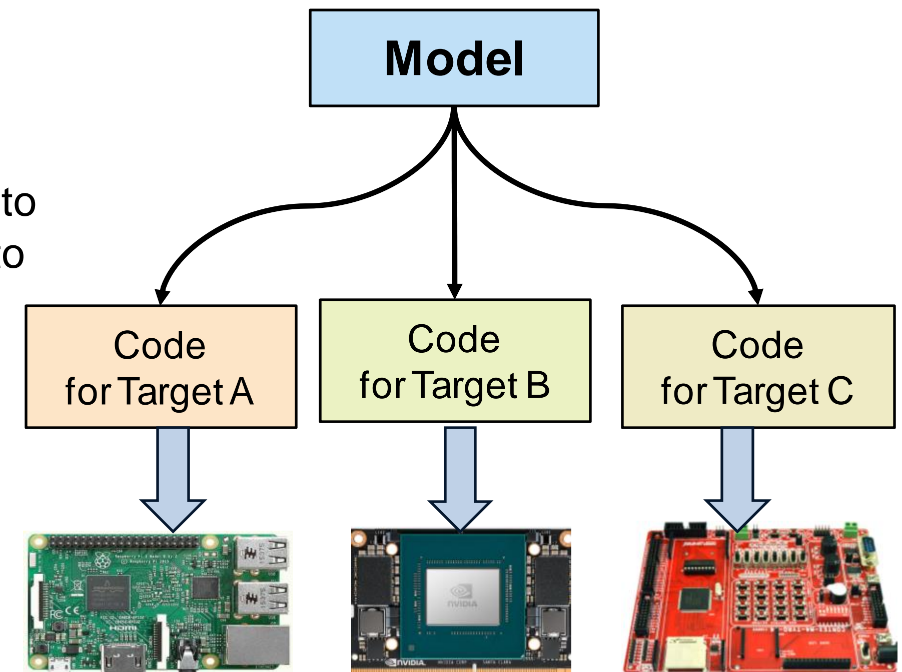

- Execution Coverage (EC)
- Decision Coverage (DC)
- Condition Coverage (CC)
- Modified Condition/Decision Coverage (MCDC)

# Code Generation

- Consideration of code generation
  - Process and Scripts need to allow models developed through multiple developers to generate code with the same code generation configuration
  - Process need to apply the suitable code configuration when the target has changed - multiple target
  - Even if the config file is changed by the developer's mistake, you need to keep the reference file separately and keep the code-gen consistency to connect the config file when executing the code generation script



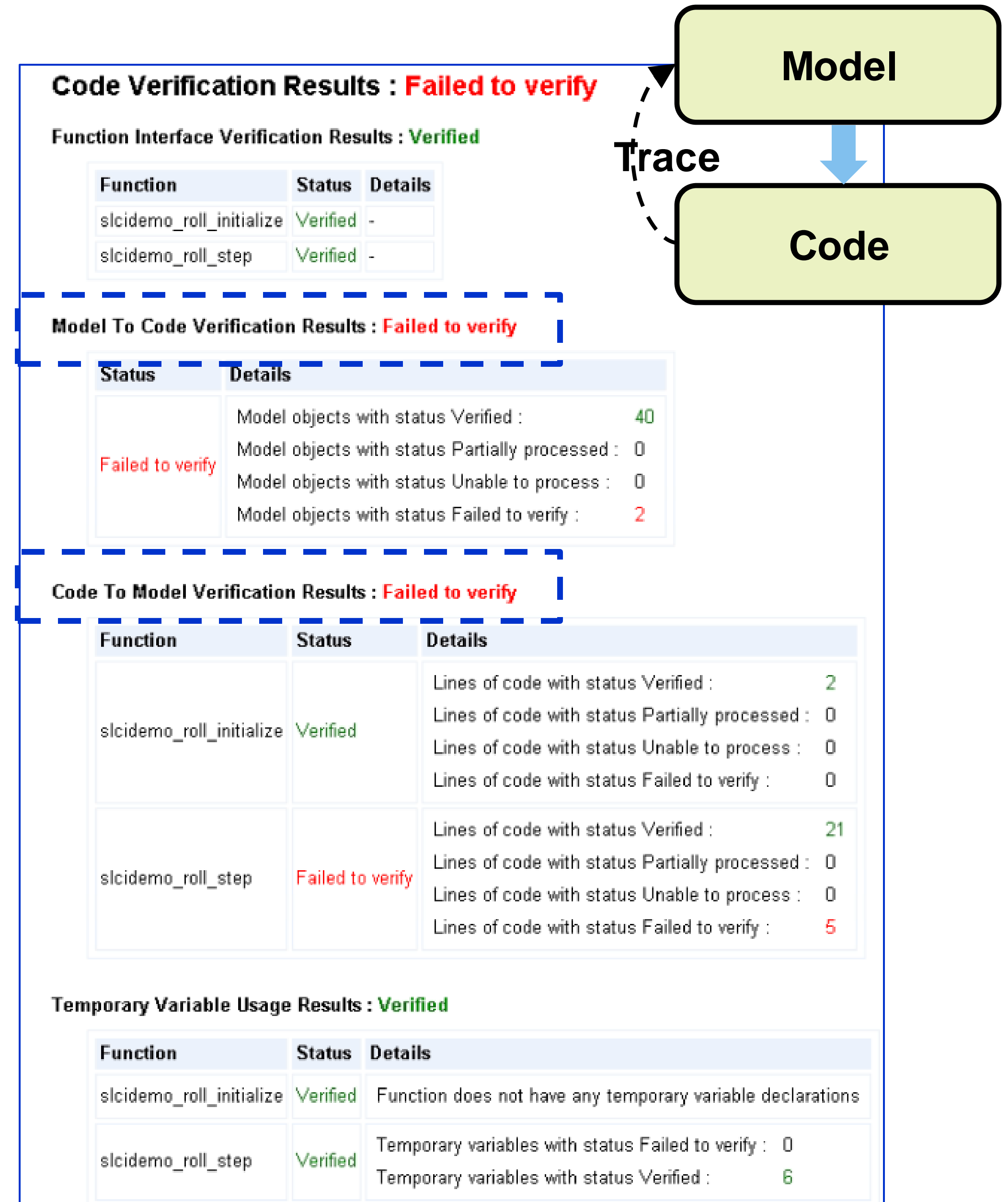




Multiple target

# Code Verification – Code Inspector

- Simulink Code Inspector provides detailed model-to-code and code-to-model traceability analysis. It generates functional equivalence and traceability reports that you can submit to certification authorities to satisfy DO-178 software coding verification objectives
- Code Inspection workflow
  1. Code inspection compatibility check via model advisor
  2. Generate code from model
  3. Inspect code and review inspection results
  4. Generate code inspection and traceability matrix
  5. If failed, check model pattern and code generation config




**Code Verification Results : Failed to verify**

**Function Interface Verification Results : Verified**

| Function | Status | Details |
|---|---|---|
| slcidemo_roll_initialize | Verified | - |
| slcidemo_roll_step | Verified | - |

**Model To Code Verification Results : Failed to verify**

| Status | Details |
|---|---|
| Failed to verify | Model objects with status Verified : 40<br>Model objects with status Partially processed : 0<br>Model objects with status Unable to process : 0<br>Model objects with status Failed to verify : 2 |

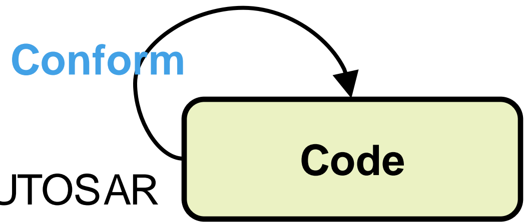
**Code To Model Verification Results : Failed to verify**

| Function | Status | Details |
|---|---|---|
| slcidemo_roll_initialize | Verified | Lines of code with status Verified : 2<br>Lines of code with status Partially processed : 0<br>Lines of code with status Unable to process : 0<br>Lines of code with status Failed to verify : 0 |
| slcidemo_roll_step | Failed to verify | Lines of code with status Verified : 21<br>Lines of code with status Partially processed : 0<br>Lines of code with status Unable to process : 0<br>Lines of code with status Failed to verify : 5 |

**Temporary Variable Usage Results : Verified**

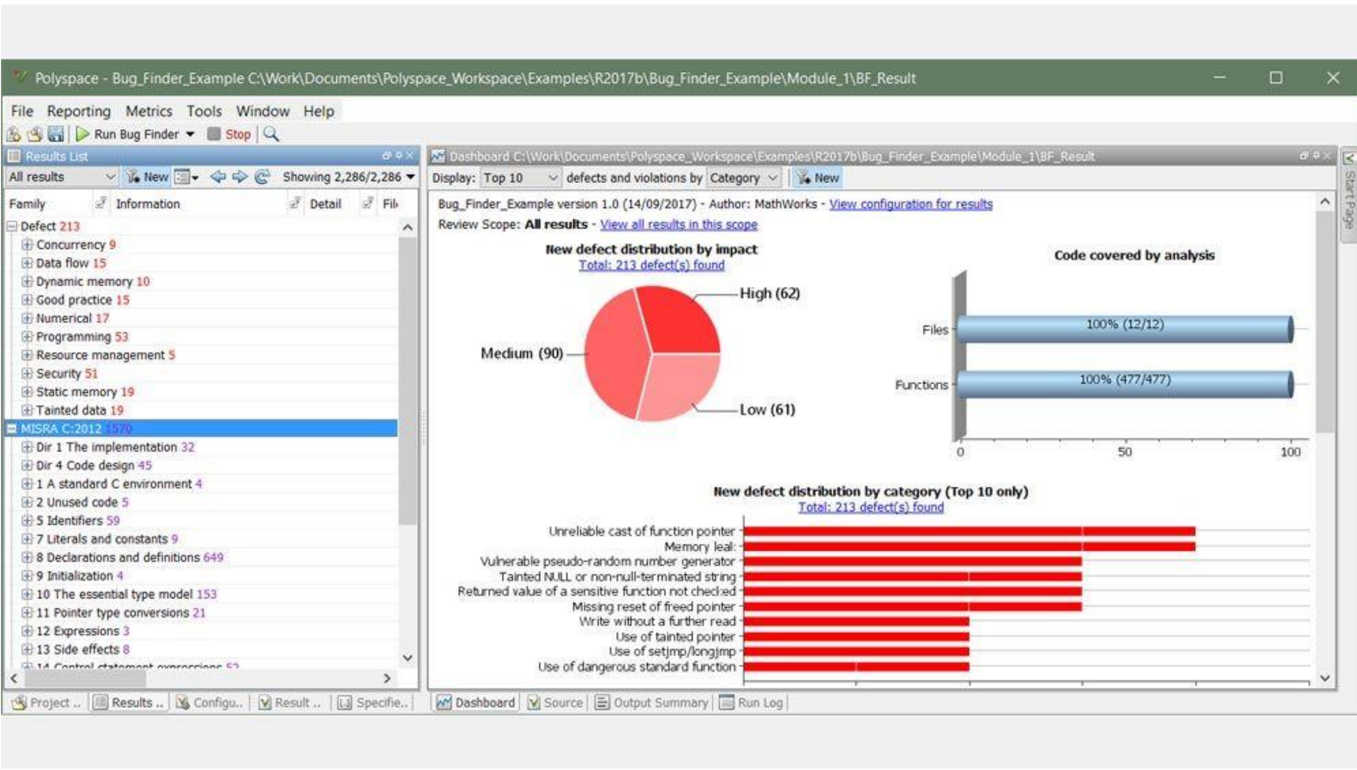
| Function | Status | Details |
|---|---|---|
| slcidemo_roll_initialize | Verified | Function does not have any temporary variable declarations |
| slcidemo_roll_step | Verified | Temporary variables with status Failed to verify : 0<br>Temporary variables with status Verified : 6 |

# Code Verification – Bug Finder and Code Prover




- Bug Finder checks compliance with coding rule standards such as MISRA C®, MISRA C++, AUTOSAR C++14, CERT® C, CERT C++, and custom naming conventions. It generates reports consisting of bugs found, code-rule violations, and code quality metrics, including cyclomatic complexity.

- Polyspace Code Prover™ proves the absence of overflow, divide-by-zero, out-of-bounds array access, and certain other run-time errors in C and C++ source code. It produces results without requiring program execution, code instrumentation, or test cases




**Green: reliable**
safe pointer access

**Red: faulty**
out of bounds error

**Gray: dead**
unreachable code

**Orange: unproven**
may be unsafe for some conditions

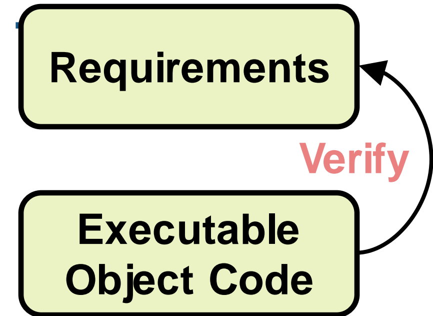
**Purple: violation**
MISRA-C/C++ or JSF++ code rules

***Range data***
*tool tip*

```
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 - i) = 10;
    }
}
```

variable 'I' (int32): [0 .. 99]
assignment of 'I' (int32): [1 .. 100]

# SIL(Software In-the Loop) & PIL(Processor In-the Loop)

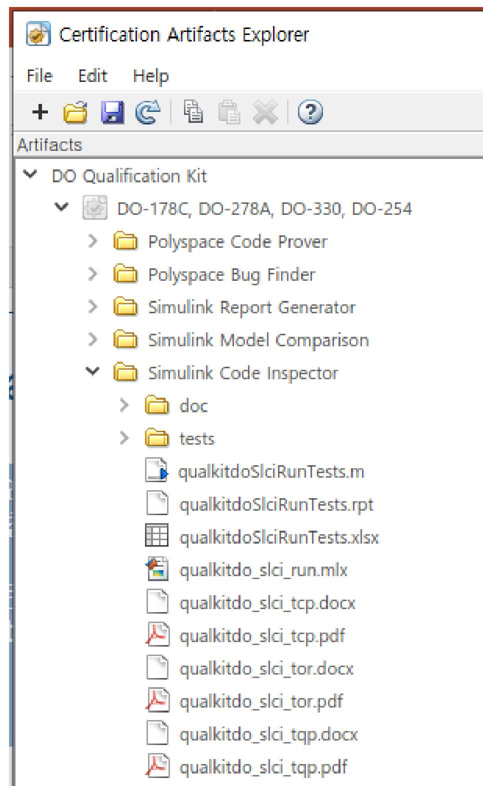



- Objective : equivalence tests(back-to-back tests) in different environments
- SIL : test generation production code with your environment or plant model to verify a successful conversion of the model to code
- PIL : evaluate the behavior of a candidate algorithm on the target processor

| Method | Execution Environments | Feature |
|---|---|---|
| MILS(Model In the Loop Simulation) | Test Cases n → Flight Control → Aircraft Model | - Verify the processing output about software |
| SILS(Software In the Loop Simulation) | Test Cases n → Flight Control (Code, C/C++) → Aircraft Model | - Verify the Auto-Generated Code(C/C++) from model<br>- Compare between Model and Code(Equivalence) |
| PILS(Processor In the Loop Simulation) | Test Cases n → Flight Control → Aircraft Model | - Actual CPU operation of the code, and the cross-compiler/linker setting on target CPU<br>- Processing time, memory usage evaluation |

# Tool Qualifications

- Tools can assist software development to analyze and potentially improve system safety by the automation of the activities performed and by predictably performing functions that may be prone to human error.
- However, an error in the tool may have a negative impact on software functionality if the tool inadequately performs its intended functions. In order to avoid this risk and to ensure the integrity of the tool functionality. the tool should be developed and verified using adequate processed.




**Tool Qualification Plan (TQP)**
**Tool Operational Requirement (TOR)**
**Test Procedure and Test Cases (TCP)**

- Tool qualification workflow

1. Provide certification authorities with a Tool Qualification Plan
2. Document Tool Operational Requirements
3. Verify that the tool satisfies Tool Operational Requirements using Test Procedure and Test Cases
4. Provide traceability between model objects, generated code, and model requirements
5. Provide certification authorities with Tool Qualification Results

# Relation between DO Qualification Kit and Life Cycle Data

- To comply standard, need to prepare 23 artifacts
  - The blue artifacts directly related in Mathworks DO Qualification Kit with MBD process

- Planning Process
  - MB11.1 Plan for Software Aspects of Certification(PSAC)
  - MB11.2 Software Development Plan
  - MB11.3 Software Verification Plan
  - MB11.4 Software Configuration Management Plan
  - MB11.5 Software Quality Assurance Plan
  - MB11.7,8 Software Requirements, Design and Code Standards
  - MB11.23 Software Model Standard(Only MBD)

- Development Process
  - MB11.9 Software Requirements Data
  - MB11.10 Software Design Description
  - MB11.11 Source Code
  - MB11.12 Executable Object Code
  - MB11.22 Parameter Data Item File
  - MB11.21 Trace Data

- Verification Process
  - MB11.13 Software Verification Cases and Procedures
  - MB11.14 Software Verification Results
  - MB11.17 Problem Reports
  - MB11.21 Trace Data

- Software Quality Assurance Process
  - MB11.19 SQA Records

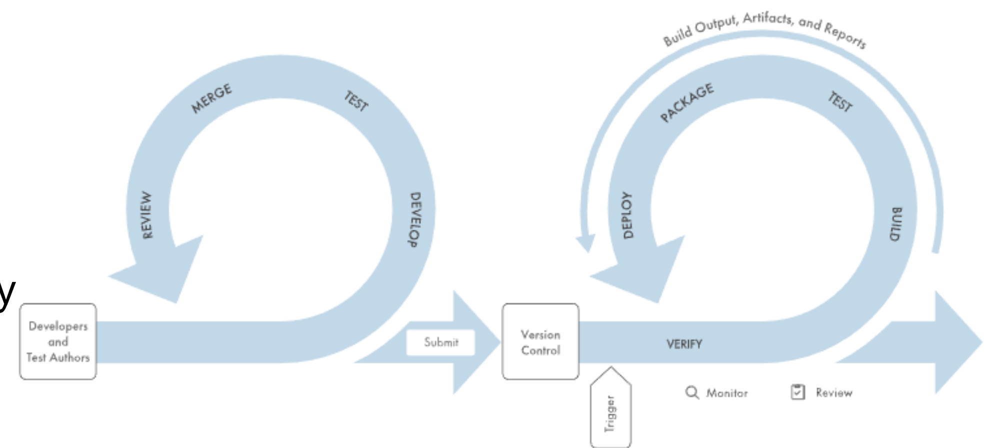
- SW Configuration Management Process
  - MB11.18 Software Configuration Management Records
  - MB11.17 Problem Reports
  - MB11.16 Software Configuration Index
  - MB11.15 Software Life Cycle Environment Configuration Index

- Certification Liaison Process
  - Software Accomplishments Summary
  - MB11.1 PSAC
  - MB11.16 Software Configuration Index

# Strategy of Continuous Integration/Deployment

Jenkins

- Need to setup strategy about applying CI/CD
  - all process are not able to be accomplished at once
  - considering team organization situation (ex: after code gen.)
  - how much automate and which process have to review manually
- Jenkins call MATLAB script to execute each process




| PROCESS | CODE | DESCRIPTION | TRIGGERD BY | ARTIFACTS | DEPENDENCY/ERROR COUNTEF | RETRUN TO | | | | | | | | | REMARKS |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| REQ DEV | A0 | | | | | | | | | | | | | | |
| SL MODEL DEV | B0 | Model Reference | | | | | | | | | | | | | |
| | B1 | Harness Model, Test Vector | | | | | | | | | | | | | |
| STATIC ANALYSIS | C0 | dead logic, run-time error | Model pushed to git repository | analysis report (derived requirements) | | A0 | B0 | | | | | | | | |
| | C1 | model guidelines (include SLCI compatibility) | C0-PASS | analysis report (model modification) | guidelines rule configuration | | B0 | | C0 | | | | | | |
| DYNAMIC ANALYSIS | D0 | Requirement based Test | C1-PASS | analysis report (pass/fail, coverage) | model, harness and test vector | A0 | B0 | B1* | C0 | C1 | | | | | B1 Only, if missing coverage |
| CODE GENERATION | E0 | Code Generation | D0-PASS | generated code | model configuration and parameters | | B0 | | | C1 | | | | | |
| CODE VERIFICATION | F0 | Code Inspection | E0-PASS | inspection report | | | B0 | | C0 | C1 | D0 | E0 | | | |
| | F1 | Equivalance Test (SIL) | F0-PASS | analysis report (code coverage) | | | B0 | | C0* | C1* | D0* | E0* | | | (*): Not required, but recommended |
| | - | Equivalance Test (PIL) | | | | | | | | | | | | | |
| | F2 | Polyspace Bug Finder | F1-PASS | analysis report (MISRA, defects, CERT-C) | | | B0* | | C0 | C1 | D0 | E0 | | | (*): justification: comments at models |
| | F3 | Polyspace Code Prover | F2-PASS | analysis report (run time errors: g/o/r/gr) | | | B0* | | C0 | C1 | D0 | E0 | F0 | F1 | red, green: model modification<br>(*): orange, green: justification |

# MATLAB EXPO

## Thank you

© 2023 The MathWorks, Inc. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See *mathworks.com/trademarks* for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.