**MBD Platform (ISO 26262)**

User Guide

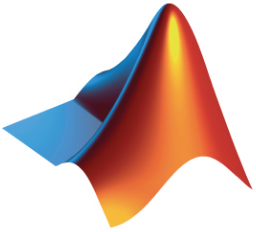


Table of Contents

[Introduction 3](#_Toc87568070)

[Product Description 3](#_Toc87568071)

[System Requirements 3](#_Toc87568072)

[Installation and Configuration 4](#_Toc87568073)

[Software Installation 4](#_Toc87568074)

[Installed Files and Folders 4](#_Toc87568075)

[Software Uninstallation 4](#_Toc87568076)

[Getting Started 5](#_Toc87568077)

[Creating a New ISO 26262 Project 5](#_Toc87568078)

[Navigating the ISO 26262 Project 5](#_Toc87568079)

[Reconfiguring the Standard Model Configurations 6](#_Toc87568080)

[Adding a New Model 8](#_Toc87568081)

[Example 10](#_Toc87568082)

[Creating the ISO 26262 Example Project 10](#_Toc87568083)

[Navigating the ISO 26262 Example Project (Before Exercising Workflow Demonstration) 10](#_Toc87568084)

[Exercising the Workflow Demonstration of the ISO 26262 Example Project 12](#_Toc87568085)

[Navigating the ISO 26262 Example Project (After Exercising Workflow Demonstration) 12](#_Toc87568086)

[Utility Functions 14](#_Toc87568087)

# Introduction

## Product Description

The MBD Platform is a template for use to streamline the adoption of Model-Based Design for high-integrity software development. This template is designed to create a new project that stores all documents, requirements, designs, code, tests, results, and tools that are relevant for ISO 26262 applications. This project is prepopulated with:

* Planning document templates
* Recommended modeling and coding standards
* Design constraints in the forms of model templates, model configurations, and checks
* Tooling for automation of vital development and verification tasks

Upon creating a new project using this template, add your software requirements, design models, and requirement-based tests, with the proper traceability. Then use the tools and utilities provided in the project to exercise the necessary development and verification tasks to meet the applicable ISO objectives. For an illustration of how to apply the MBD Platform to the Model-Based Design workflow for ISO 26262, see the “Example Project”.

## System Requirements

* Supported platforms:
* Windows
* Linux
* Minimum MATLAB software requirements:
  + MATLAB®
  + Simulink®
  + MATLAB Report Generator™
  + Simulink Report Generator™
  + MATLAB Coder™
  + Simulink Coder™
  + Embedded Coder®
  + Simulink Requirements™
  + Simulink Check™
  + Simulink Coverage™
  + Simulink Test™
  + Simulink Design Verifier™
  + Polyspace Bug Finder™
  + Polyspace Code Prover™
* External software requirements:
  + Supported C/C++ compiler per <https://www.mathworks.com/support/compilers.html>.

# Installation and Configuration

## Software Installation

To install the MBD Platform, run the app installer *ISO MBD Platform R2021b.mlappinstall*. This will create an *ISOMBDPlatformR2021b* folder under the MATLAB add-on installation folder, which is *C:\Users\username\AppData\Roaming\MathWorks\MATLAB* by default.

## Installed Files and Folders

When using the MBD Platform, you might find it helpful to know the related files as explained below:

* The MBD Platform provides a project template called *ISO\_Project\_R2021b.sltx* that streamlines the adoption of Model-Based Design for software development per ISO 26262. Use this template to create a new Simulink Project when starting a new development.
* The MBD Platform also provides an example template named *ISO\_Project\_Example\_R2021b.sltx*. Use this template to demonstrate features of the MBD Platform and the overall workflow.

## Software Uninstallation

To uninstall the MBD Platform,

1. Open the *Apps* tab from the MATLAB desktop.
2. In the *My Apps* section, right-click *ISO MBD Platform R2021b*, and then select *Uninstall*.

# Getting Started

## Creating a New ISO 26262 Project

The MBD Platform adds a template for managing/conducting software development process for ISO 26262 using MathWorks tools. To take advantage of the outline of this project when starting a new program, you can create a new project based on this template. To do this,

1. Open the *Apps* tab from the MATLAB desktop.
2. In the *My Apps* section, select *ISO MBD Platform R2021b* to load the app.
3. On the MATLAB *Home* tab, click *Simulink* to open the Simulink Start Page.
4. In the Simulink Start Page, hover over *ISOProject* under the *My Templates* panel, and then click *Create Project*. This opens the Create Project dialog.
5. In the Create Project dialog, enter the *Name* and *Folder*, and then click *OK*. This creates a new project based on the *ISOProject* template.

## Navigating the ISO 26262 Project

After the project is created, Simulink automatically launches the new project. In this new project, you will find the following folders:

* *ISO\_01\_Planning*
* In the *checklists* subfolder, you will find checklists that are useful for reviewing plans, requirements, models, code, test cases, and test results.
* In the *plans* subfolder, you will find a template for Documentation of the Software Development Environment.
* In the *standards* subfolder, you will find templates for Modeling Guidelines and Coding Guidelines.
* *ISO\_02\_Requirements*
* In the *specification* subfolder, you will store all documents that capture your software safety requirements and derived requirements.
* In the *verification\_results* subfolder, you will store checklists related to all review activities of the software safety requirements and derived requirements.
* *ISO\_03\_Architecture*
* In the *analysis* subfolder, you will store all reports from applicable safety analysis and dependent failures analysis, as well as all control flow analysis, data flow analysis, and scheduling analysis if performed.
* In the *specification* subfolder, you will store all documents and data that capture your software architecture design.
* In the *verification\_results* subfolder, you will store checklists related to all review activities of the software architecture design.
* *ISO\_04\_Design*
* In the *common* subfolder, you will create block libraries and data dictionaries that are shared across multiple models in the project.
* The *sample\_model* subfolder is a folder template for use to create actual model folders based on model names. Once a model folder is created, you will create the model and its data in the *specification* subfolder, develop test cases used for verifying the model against software requirements in the *test\_cases* subfolder, and store all verification results for the model in the *verification\_results* subfolder.
* *ISO\_05\_Code*
* In the *specification* subfolder, you will store all source code and executable object code generated from the models in the project.
* In the *verification\_results* subfolder, you will store all verification results for the source code and executable object code.
* *ISO\_06\_ToolQualification*

You will find placeholders for tool qualification life cycle data.

* *help*

You will find a user guide.

* *project\_configuration*
* In the *checks* subfolder, you will find the configuration files for Model Advisor checks and Polyspace Coding Standard checks designed to complement the Modeling Guidelines and Coding Guidelines provided in the project, respectively.
* In the *templates* subfolder, you will find model configurations that are compatible with the software design constraints and the Modeling Guidelines.
* *tools*
* In the *apps* subfolder, you will find visual layouts and callbacks of graphical user interfaces developed to streamline user operations.
* In the *icons* subfolder, you will find digital images used for rendering graphics.
* In the *utilities* subfolder, you will find utilities for project management, model query, report generation, code generation, and model/code verification.
* *work*

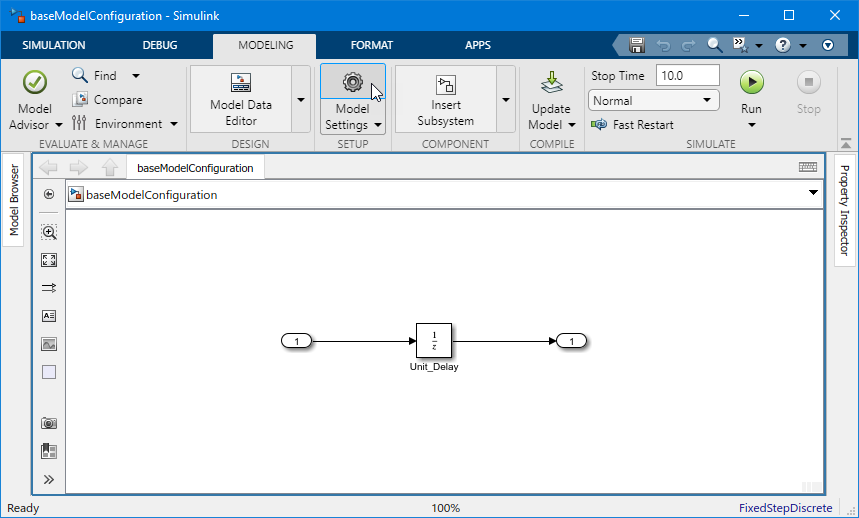
The default working folder. This folder also contains the *cache* folder in which model build artifacts used for simulation are stored.

## Reconfiguring the Standard Model Configurations

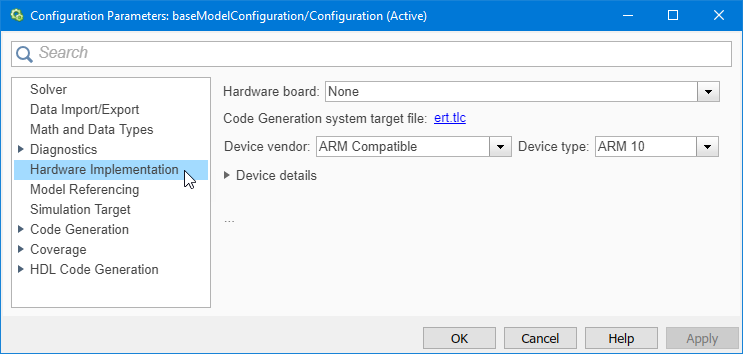
Every new project created from the template is equipped a *csMultiInstance.sldd* and a *csSingleInstance.sldd* in *ISO\_04\_Design\common\specification\configurations*. These standard data dictionaries each holds a model configuration meant to be used by all models you plan to develop in the project. For reusable (multi-instantiable) models, use the configuration in *csMultiInstance.sldd*. Otherwise, use the configuration in *csSingleInstance.sldd*. Except for the *ModelReferenceNumInstancesAllowed* property, the configurations in both *csMultiInstance.sldd* and *csSingleInstance.sldd* have the same settings. Created using *reusableModelConfig.m* and *nonreusableModelConfig.m*, these configurations derive their standardized settings from *iso26262Config.m*. The standardized settings, in turn, are exported from the model *baseModelConfiguration.slx*.

The model configurations must be reconfigured to account for the correct *Hardware Implementation* settings for your application. To do this,

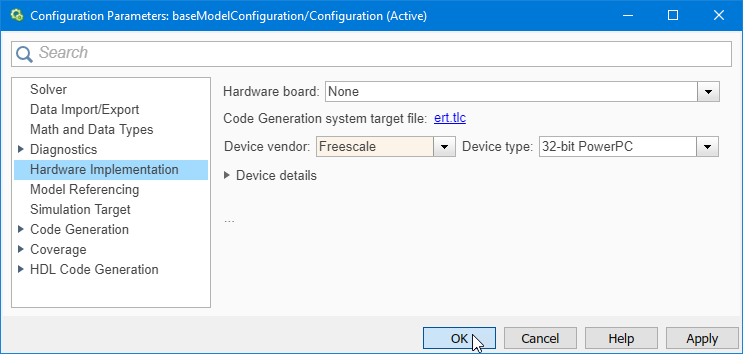
1. Enter *baseModelConfiguration* to open the model.
2. Select *MODELING > SETUP > Model Settings* to open the Configuration Parameters dialog.



1. In the Configuration Parameters dialog, click the *Hardware Implementation* item to open the *Hardware Implementation* pane.



1. Set the *Device vendor* and *Device type* options per your hardware specifications, and then click *OK*.



1. Save the model.
2. Use the following command to get the new *Simulink.ConfigSet* object from the model:

*>> configSet = getConfigSet('baseModelConfiguration', 'Configuration');*

1. Then use the following command to export the new *Simulink.ConfigSet* object to *iso26262Config.m*, overwriting the existing settings in the file:

*>> configSet.saveAs('..\project\_configuration\templates\model\_configurations  
\iso26262Config');*

1. Use the following commands to update the data dictionary that holds the model configuration for reusable (multi-instantiable) models:

*>> dd = Simulink.data.dictionary.open('csMultiInstance.sldd');  
>> cfg = dd.getSection('Configurations');  
>> cfg.importFromFile('reusableModelConfig.m', 'existingVarsAction', 'overwrite');  
>> dd.saveChanges();*

1. Use the following commands to update the data dictionary that holds the model configuration for nonreusable (not multi-instantiable) models:

*>> dd = Simulink.data.dictionary.open('csSingleInstance.sldd');  
>> cfg = dd.getSection('Configurations');  
>> cfg.importFromFile('nonreusableModelConfig.m', 'existingVarsAction', 'overwrite');  
>> dd.saveChanges();*

## Adding a New Model

Based on the software requirements, you create models that serve as the software unit design. This kit expects that each model has its own model folder under the *ISO\_04\_Design* folder based on the folder structure of *sample\_model*.

Before you create a new model, use the *addModelFolder* utility to create a new model folder based on the model name. For example, to create a new model folder for a new model named *controller*, use one of the following commands:

* *>> addModelFolder('controller')*
* *>> addModelFolder('controller', ' Nonreusable')*
* *>> addModelFolder('controller', ' Nonreusable', 'ASILD')*

This command creates a new *controller* folder under *ISO\_04\_Design*. Like *sample\_model*, this new folder contains a *specification*, a *test\_cases*, and a *verification\_results* subfolder. In the *specification* subfolder, you will find an empty model *controller.slx*. In addition, you will find a complementary data dictionary *DD\_controller.sldd* in the *specification\data* subfolder. Note that this data dictionary is automatically attached to the model and it includes either *csMultiInstance.sldd* or *csSingleInstance.sldd* as a referenced dictionary. You can find both *csMultiInstance.sldd* and *csSingleInstance.sldd* in *ISO\_04\_Design\common\specification\configurations*. If the command is called without a second argument, you will also find a *localDD\_controller.m* in the *specification\data* subfolder. When using this command, the third argument can be either *N/A*, *QM*, *ASILA*, *ASILB*, *ASILC*, or *ASILD*. This argument governs the naming rule of shared utilities to prevent software of different ASIL from accessing the same code. You are now ready to build the model using native Simulink blocks.

When developing a model, you often define data to help characterize its behavior. Keep in mind that the required data may be specific to the model or shared across multiple models. The collection of data that is consumed only by the model is referred to as its model specific data dictionary, whereas data that is shared forms a common data dictionary.

Every model you develop in the project is associated with a complementary Simulink data dictionary that is named after the model with a prefix of *DD\_* in the *specification\data* subfolder of the model folder. For a nonreusable model:

* The complementary data dictionary must include *csSingleInstance.sldd*, which holds the model configuration derived from *nonreusableModelConfig.m*.
* Use the complementary data dictionary to hold model specific data.

For a reusable model:

* The complementary data dictionary must include *csMultiInstance.sldd*, which holds the model configuration derived from *reusableModelConfig.m*.
* Use the complementary data dictionary to hold model specific data that is not instance specific.
* You may additionally need to create instance specific data, including model arguments, in the model workspace.
* Use a MATLAB script that is named after the model with a prefix of *localDD\_* in the *specification\data* subfolder of the model folder to hold instance specific data.
* Set up the model to automatically load instance specific data into its model workspace by running this script.

Using the example above, you would create a *DD\_controller.sldd* and a *localDD\_controller.m* (if reusable) in *ISO\_04\_Design\controller\specification\data*.

For shared data, you create Simulink data dictionaries in *ISO\_04\_Design\common\specification\data*. These data dictionaries must be referenced by the complementary data dictionary of every model that consumes the data.

You may need to create user-defined data types like aliases, buses, and enumerations to work with the model. For *Simulink.AliasType* and *Simulink.Bus* objects, create Simulink data dictionaries in *ISO\_04\_Design\common\specification\data\_types*, and then reference these data dictionaries as necessary. For enumeration classes, simply create the appropriate class definition files in *ISO\_04\_Design\common\specification\data\_types*.

Upon developing the model and its data dictionaries, you should create a MATLAB script for use to open the model. This script is required if you intend to leverage utilities in the ISO 26262 Project to perform downstream tasks. The script must be named after the model with a prefix of *open\_*. Using the example above, you would create an *open\_controller.m* to achieve this goal.

# Example

## Creating the ISO 26262 Example Project

The MBD Platform adds a project that illustrates the Model-Based Design workflow for ISO 26262 as a template to your MATLAB installation. To access the example in this project, you must create a new project based on this template. To do this,

1. Open the *Apps* tab from the MATLAB desktop.
2. In the *My Apps* section, select *ISO MBD Platform R2021b* to load the app.
3. On the MATLAB *Home* tab, click *Simulink* to open the Simulink Start Page.
4. In the Simulink Start Page, click *ISOProjectExample* under the *My Templates* panel, and then click *Create Project*. This opens the Create Project dialog.
5. In the Create Project dialog, enter the *Project Name* and *Project Folder*, and then click *OK*. This creates a new project based on the *ISOProjectExample* template.

## Navigating the ISO 26262 Example Project (Before Exercising Workflow Demonstration)

Simulink automatically launches the example project after creating it. Notice that the structure of this example project is identical to the new ISO 26262 Project. The example project does have the following additional contents used to explain the workflow and the available utilities:

* Under *ISO\_02\_Requirements*
* In the *specification* subfolder, you will find a document for the software safety requirements. Authored using the Requirement Editor, the software safety requirements are captured in the requirement set *CruiseControl\_SwReq.slreqx*.
* Under *ISO\_04\_Design*
* In the *common\specification\data\_types* subfolder, you will find *opMode.m* and *reqMode.m*. These files define the enumerations that are vital to the project.
* There is a model folder that is derived from the *sample\_model* folder template. This folder contains a Simulink model that implements part of the software safety requirements captured in *ISO\_02\_Requirements\specification\CruiseControl\_SwReq.slreqx*.
* In the *specification* subfolder of the model folder, you will find the model file itself. Along with the *.slx* file, you will also find a *.slmx* file that stores the trace data between the model and the associated software requirements. As part of the convention when using the ISO 26262 Project, notice that every model has a complementary MATLAB script for use to open the model.
* In the *specification\data* subfolder of the model folder, you will find a complementary Simulink data dictionary that is named after the model with a prefix of *DD\_* for holding its model specific data.
* The *specification\documents* subfolder of the model folder is used for holding a design description of the model. This folder is empty initially.
* In the *test\_cases* subfolder of the model folder, you will find an *RBT* and an *SBT* folder that are used for holding test cases of the model.
* The *RBT* folder is used for keeping test cases developed to verify the model against the software requirements it implements. In this folder, you create a Simulink Test file to manage all requirement-based test cases. This test file must be named after the model with a postfix of *\_REQ\_Based\_Test*. In this example, we have developed requirement-based test cases for *DriverRequest*. You will find them in *DriverRequest\_REQ\_Based\_Test.mldatx* under *DriverRequest\test\_cases\RBT*. These test cases make use of an externally stored test harness *DriverRequest\_Harness\_REQ.slx*. Along with *DriverRequest\_REQ\_Based\_Test.mldatx*, you will also find an *DriverRequest\_REQ\_Based\_Test.slmx* that stores the trace data between the test cases and the software safety requirements they are developed to verify.
* The *SBT* folder is used for keeping test cases derived from the model. These structure-based tests are often needed to supplement the requirement-based tests to achieve the necessary test coverage. This folder is empty initially.
* In the *verification\_results* subfolder of the model folder, you will find folders for the various model verification tasks. These tasks include design review of the model, verification of the model against requirements, coverage analysis of the model, examination of the model against design errors using static analysis, inspection of the model against design standards, and measurement of model metrics.
* The *verification\_results\model\_reviews* subfolder of the model folder is used for holding the design review checklist of the model. This folder is empty initially.
* The *verification\_results\simulation\_results* subfolder of the model folder contains an *RBT* and an *SBT* folder that are used for holding simulation results of the model. These folders are empty initially.
* The *verification\_results\model\_coverages* subfolder of the model folder contains an *RBT* and an *SBT* folder that are used for holding coverage results of the model. These folders are empty initially.
* The *verification\_results\design\_error\_detections* subfolder of the model folder is used for holding Simulink Design Verifier analysis results of the model. This folder is empty initially.
* The *verification\_results\design\_standard\_checks* subfolder of the model folder is used for holding Model Advisor check results of the model. This folder is empty initially.
* The *verification\_results\model\_metrics* subfolder of the model folder is used for holding design metrics of the model. This folder is empty initially.
* Under *ISO\_05\_Code*
* In the *verification\_results* subfolder, you will find folders for the various code verification tasks. These tasks include verification of code against requirements, code coverage analysis, finding of defects in code using static analysis, inspection of code against coding standards, and measurement of code metrics.
* The *verification\_results\eoc\_test\_results* subfolder holds software-in-the-loop and processor-in-the-loop test results of the generated code. The folder created for the model is empty initially.
* The *verification\_results\code\_coverages* subfolder holds coverage results of the generated code. The folder created for the model is empty initially.
* The *verification\_results\coding\_error\_detections* subfolder holds Bug Finder analysis results of the generated code. The folder created for the model is empty initially.
* The *verification\_results\code\_proving* subfolder holds Code Prover verification results of the generated code. The folder created for the model is empty initially.
* The *verification\_results\code\_standard\_checks* subfolder holds Bug Finder coding rule checker results of the generated code. The folder created for the model is empty initially.
* The *verification\_results\code\_metrics* subfolder holds code metrics of the generated code. The folder created for the model is empty initially.
* Under *work*
* The *demo* subfolder contains a MATLAB live script *runDemo.mlx* created for use in this example to illustrate the Model-Based Design workflow for ISO 26262. This live script contains instructions to the entire workflow demonstration.

## Exercising the Workflow Demonstration of the ISO 26262 Example Project

After launching the example project, you can simply follow the instructions given in *work\demo\runDemo.mlx* to run the workflow demonstration. Use the following command to open the live script if necessary:

*>> open('runDemo.mlx')*

Following the explanations given in the live script, use the hyperlinks to carry out the tasks called out in the workflow demonstration. You can hover over a hyperlink to view the MATLAB command it executes.

## Navigating the ISO 26262 Example Project (After Exercising Workflow Demonstration)

Upon the completion of the workflow demonstration, you should find the following generated artifacts in the example project:

* Under *ISO\_02\_Requirements*
* In the *specification\documents* subfolder, you will find the report of the software safety requirements. This report is generated from *CruiseControl\_SwReq.slreqx* using *genReqReport* (*tools\utilities\report\_generation\genReqReport.m*).
* Under *ISO\_04\_Design*
* In the *specification\documents* subfolder of each model folder, you will find the design description of the model. This report is generated from the model using *genSDD* (*tools\utilities  
  \report\_generation\genSDD.m*).
* In the *test\_cases\SBT* subfolder of *DriverRequest*, you will find the structure-based test generation results and report of *DriverRequest*, including a test file and a test harness. You will find the generated structure-based test cases in *DriverRequest\_SLDV\_Based\_Test.mldatx*. These test cases make use of an externally stored test harness *DriverRequest\_Harness\_SLDV.slx*. They are generated from *DriverRequest* to satisfy objectives that are not achieved in its requirement-based test coverage using *genLowLevelTests* (*tools  
  \utilities\verification\dynamic\_testing\genLowLevelTests.m*).
* In the *verification\_results\simulation\_results\RBT* subfolder of *DriverRequest*, you will find the requirement-based simulation results and report of *DriverRequest*. These simulation results are generated from running the test cases that verify *DriverRequest* against the software requirements it implements using *verifyModel2Reqs* (*tools\utilities\verification  
  \dynamic\_testing\verifyModel2Reqs.m*).
* In the *verification\_results\model\_coverages\RBT* subfolder of *DriverRequest*, you will find the requirement-based model coverage results and report of *DriverRequest*. These coverage results are collected from running the test cases that verify *DriverRequest* against the software requirements it implements using *verifyModel2Reqs* (*tools\utilities\verification  
  \dynamic\_testing\verifyModel2Reqs.m*).
* In the *verification\_results\design\_error\_detections* subfolder of each model folder, you will find the design error detection report of the model. This report is generated from examining the model against potential run-time errors and dead logic with Simulink Design Verifier using *detectDesignErrs* (*tools\utilities\verification\static\_analysis\detectDesignErrs.m*).
* In the *verification\_results\design\_standard\_checks* subfolder of each model folder, you will find the Model Advisor report of the model. This report is generated from examining the model against Model Advisor checks enabled by *project\_configuration\checks\iso26262Checks.json* using *checkModelStds* (*tools\utilities\verification\static\_analysis\checkModelStds.m*).
* Under *ISO\_05\_Code*
* In the *specification* subfolder, you will find the generated code of the model developed for use in this example. This source code is generated from the model using *genSrcCode* (*tools\utilities  
  \code\_generation\genSrcCode.m*). Note that the code is placed under *slprj\ert*.
* In the *verification\_results\eoc\_test\_results\DriverRequest\host\RBT*, you will find the requirement-based software-in-the-loop test results and report of the generated code for *DriverRequest*. These test results are generated from running the test cases that verify *DriverRequest* against the software requirements it implements in software-in-the-loop mode using *verifyObjCode2Reqs* (*tools\utilities\verification\dynamic\_testing\verifyObjCode2Reqs.m*).
* In the *verification\_results\eoc\_test\_results\DriverRequest\host\SBT*, you will find the structure-based software-in-the-loop test results and report of the generated code for *DriverRequest*. These test results are generated from running the test cases derived from *DriverRequest* to complement its requirement-based test coverage in software-in-the-loop mode using *verifyObjCode2LowLevelTests* (*tools\utilities\verification\dynamic\_testing  
  \verifyObjCode2LowLevelTests.m*).
* In the *verification\_results\code\_coverages\DriverRequest\host\RBT* subfolder, you will find the requirement-based code coverage results and report of the generated code for *DriverRequest*. These coverage results are collected from running the test cases that verify *DriverRequest* against the software requirements it implements in software-in-the-loop mode using *verifyObjCode2Reqs* (*tools\utilities\verification\dynamic\_testing\verifyObjCode2Reqs.m*).
* In the *verification\_results\code\_coverages\DriverRequest\host\SBT* subfolder, you will find the structure-based code coverage results and report of the generated code for *DriverRequest*. These coverage results are collected from running the test cases derived from *DriverRequest* to complement its requirement-based test coverage in software-in-the-loop mode using *verifyObjCode2LowLevelTests* (*tools\utilities\verification\dynamic\_testing  
  \verifyObjCode2LowLevelTests.m*).
* In the *verification\_results\code\_coverages\DriverRequest* subfolder, you will find the cumulative code coverage report of the generated code for *DriverRequest*. These coverage results are assembled from merging both the requirement-based and structure-based code coverage results using *mergeCodeCoverage* (*tools\utilities\verification\dynamic\_testing  
  \mergeCodeCoverage.m*).
* In the *verification\_results\code\_proving\DriverRequest* subfolder, you will find the Code Prover verification report of the generated code for *DriverRequest*. This report is generated from checking the source code of *DriverRequest* against potential software defects with Polyspace Code Prover using *proveCodeQuality* (*tools\utilities\verification\static\_analysis  
  \proveCodeQuality.m*).
* In the *verification\_results\code\_standard\_checks\DriverRequest* subfolder, you will find the Bug Finder coding rule analysis report of the generated code for *DriverRequest*. This report is generated from checking the source code of *DriverRequest* against Bug Finder coding rules using *checkCodeStds* (*tools\utilities\verification\static\_analysis\checkCodeStds.m*).

# Utility Functions

The following table summarizes the utilities available in the MBD Platform:

|  |  |
| --- | --- |
| **Utility Name** | **Description** |
| *addModelFolder* | Create a new model folder based on the folder template *ISO\_04\_Design\sample\_model*. This also creates model specific folders under the subfolders in *ISO\_05\_Code\verification\_results*.  Syntax:   * *addModelFolder('controller')* * *addModelFolder('controller', ' Nonreusable')* * *addModelFolder('controller', ' Nonreusable', 'ASILD')*   Use the *Nonreusable* option if you want the new model to be nonreusable, otherwise it is reusable. Use the third argument to specify ASIL of the model, i.e., *N/A*, *QM*, *ASILA*, *ASILB*, *ASILC*, or *ASILD*. |
| *clearTestManager* | Clear all test files and result sets loaded in the Test Manager.  Syntax:  *clearTestManager()* |
| *deleteModelFolder* | Delete the model folder. Use this to erase the model folder from the file system after removing it from the project.  Syntax:  *deleteModelFolder('MODEL')* |
| *removeModelFolder* | Remove the model folder from the current project. This only excludes the model folder from the project. It does not erase the model folder from the file system.  Syntax:  *removeModelFolder('MODEL')* |
| *resetDefaultMdlAdvCfg* | Reset the default Model Advisor configuration to use the factory default.  Syntax:  *resetDefaultMdlAdvCfg()* |
| *resetWorkspace* | Clear all variables in the workspace except the default model configurations. Use this in lieu of *clear*.  Syntax:  *resetWorkspace()* |
| *setDefaultMdlAdvCfg* | Set the default Model Advisor configuration to use the groups of checks based on *project\_configuration\checks \iso26262Checks.json*.  Syntax:  *setDefaultMdlAdvCfg()* |
| *viewFileDependency* | Launch the Model Dependency Viewer to display model and library file dependency.  Syntax:  *viewFileDependency('MODEL')* |
| *viewModelDependency* | Launch the Model Dependency Viewer to display model hierarchy.  Syntax:  *viewModelDependency('MODEL')* |
| *genReqReport* | Generate a requirement report from the requirement set.  Syntax:   * *genReqReport('REQSET')* * *genReqReport('REQSET', 'AUTHOR')* * *genReqReport('REQSET', [], 'CI')*   Use the second argument to optionally specify an author’s name. Use the *CI* option to operate within the context of continuous integration. This option suppresses launching of all reports upon completion.  This utility creates the following:  A document named after the requirement set with a postfix of *\_Requirements\_Report* under *ISO\_02\_Requirements \specification\documents* |
| *genSDD* | Generate a System Design Description (SDD) report from the model.  Syntax:   * *genSDD('MODEL')* * *genSDD('MODEL', 'AUTHOR')* * *genSDD('MODEL', [], 'CI')*   Use the second argument to optionally specify an author’s name. Use the *CI* option to operate within the context of continuous integration. This option suppresses launching of all reports upon completion.  This utility creates the following:  A document named after the model with a postfix of *\_SDD* under *ISO\_04\_Design\MODEL\specification\documents*  This utility depends on the following:  *open\_MODEL* |
| *genSrcCode* | Generate source code from the model using Embedded Coder.  Syntax:   * *genSrcCode('MODEL')* * *genSrcCode('MODEL', 'TreatAsTopMdl')*   Use the *TreatAsTopMdl* option to handle the model as a top-level model, otherwise it is regarded as a referenced model.  This utility creates the following:  Code under *MODEL\_ert\_rtw* or *slprj\ert\MODEL* in *ISO\_05\_Code\specification*  This utility depends on the following:  *open\_MODEL* |
| *genLowLevelTests* | Generate structure-based tests for the model based on existing coverage data using Simulink Design Verifier.  Syntax:   * *genLowLevelTests('MODEL')* * *genLowLevelTests('MODEL', 'ModelCoverageObjectives', 'MCDC', 'TestSuiteOptimization', 'IndividualObjectives' 'MaxProcessTime', 300, 'AbsTol', 1e-6, 'RelTol', 1e-3, 'CI', true)*   Use the *ModelCoverageObjectives* setting to decide which type of coverage analysis to perform, i.e., *Decision*, *ConditionDecision*, or *MCDC*. Use the *TestSuiteOptimization* setting to decide which test generation strategy to use, i.e., *Auto*, *IndividualObjectives*, *LongTestCases*, or *LargeModel (Nonlinear Extended)*. Use the *MaxProcessTime* setting to control the time limit of the analysis. Use the *AbsTol* and *RelTol* settings to control the baseline criteria tolerances of the generated test cases. Use the *CI* setting to control if the operation is within the context of continuous integration. Setting this option to *true* suppresses launching of all reports upon completion. If not specified, the default values of *ModelCoverageObjectives*, *TestSuiteOptimization*, *MaxProcessTime*, *AbsTol*, *RelTol*, and *CI* are *MCDC*, *IndividualObjectives*, *300*, *1e-6*, *1e-3*, and *false*, respectively.  This utility creates the following:   * A test generation report named after the model with a postfix of *\_Test\_Generation\_Report* under *ISO\_04\_Design\MODEL \test\_cases\SBT* * A test file named after the model with a postfix of *\_SLDV\_Based\_Test* under *ISO\_04\_Design\MODEL\test\_cases \SBT*.   This utility depends on the following:   * *open\_MODEL* * *ISO\_04\_Design\MODEL\verification\_results\model\_coverages \RBT\MODEL\_REQ\_Based\_Model\_Coverage.cvt* |
| *mergeCodeCoverage* | Merge coverage results from verifying the generated code of the model against the software requirements and structure-based tests generated by Simulink Design Verifier.  Syntax:   * *mergeCodeCoverage('MODEL', 'SIL')* * *mergeCodeCoverage('MODEL', 'PIL')* * *mergeCodeCoverage('MODEL', 'SIL', 'CI')*   Use the second argument to control if the coverage results are collected from exercising the compiled code on the host computer via SIL simulations or the target computer via PIL simulations. Use the *CI* option to operate within the context of continuous integration. This option suppresses launching of all reports upon completion. It also returns a result object that captures the operational outcome.  This utility creates the following (*SIL*):  A cumulative coverage report named after the model with a postfix of *\_Merged\_Code\_Coverage\_Report* under *ISO\_05\_Code\verification\_results\code\_coverages\MODEL \host*.  This utility depends on the following (*SIL*):   * *ISO\_05\_Code\verification\_results\eoc\_test\_results\MODEL \host\RBT\MODEL\_SIL\_REQ\_Based\_Test\_Results.mldatx* * *ISO\_05\_Code\verification\_results\eoc\_test\_results\MODEL \host\SBT\MODEL\_SIL\_SLDV\_Based\_Test\_Results.mldatx* |
| *mergeModelCoverage* | Merge coverage results from verifying the model against the software requirements and structure-based tests generated by Simulink Design Verifier.  Syntax:   * *mergeModelCoverage('MODEL')* * *mergeModelCoverage('MODEL', 'CI')*   Use the *CI* option to operate within the context of continuous integration. This option suppresses launching of all reports upon completion. It also returns a result object that captures the operational outcome.  This utility creates the following:  A cumulative coverage report named after the model with a postfix of *\_Merged\_Model\_Coverage\_Report* under *ISO\_04\_Design\MODEL\verification\_results\model\_coverages*.  This utility depends on the following:   * *ISO\_04\_Design\MODEL\verification\_results\simulation\_results \RBT\MODEL\_REQ\_Based\_Test\_Results.mldatx* * *ISO\_04\_Design\MODEL\verification\_results\simulation\_results \SBT\MODEL\_SLDV\_Based\_Test\_Results.mldatx* |
| *verifyModel2LowLevelTests* | Verify the model against structure-based tests generated by Simulink Design Verifier, and then perform model coverage analysis. All tests exercise the compiled model on the host computer via normal simulations.  Syntax:   * *verifyModel2LowLevelTests('MODEL')* * *verifyModel2LowLevelTests('MODEL', 'TreatAsTopMdl')* * *verifyModel2LowLevelTests('MODEL', [], 'AUTHOR')* * *verifyModel2LowLevelTests('MODEL', [], [], 'CI')*   Use the *TreatAsTopMdl* option to handle the model as a top-level model, otherwise it is regarded as a referenced model. Use the third argument to optionally specify an author’s name. Use the *CI* option to operate within the context of continuous integration. This option suppresses launching of all reports upon completion. It also returns a result object that captures the operational outcome.  This utility creates the following:   * A test report named after the model with a postfix of *\_SLDV\_Based\_Test\_Report* under *ISO\_04\_Design\MODEL \verification\_results\simulation\_results\SBT* * A coverage report named after the model with a postfix of *\_SLDV\_Based\_Model\_Coverage\_Report* under *ISO\_04\_Design \MODEL\verification\_results\model\_coverages\SBT*   This utility depends on the following:  *ISO\_04\_Design\MODEL\test\_cases\SBT \MODEL\_SLDV\_Based\_Test.mldatx* |
| *verifyModel2Reqs* | Verify if the model complies with the software requirements, and then perform model coverage analysis. All tests exercise the compiled model on the host computer via standard simulations.  Syntax:   * *verifyModel2Reqs('MODEL')* * *verifyModel2Reqs('MODEL', 'TreatAsTopMdl')* * *verifyModel2Reqs('MODEL', [], 'AUTHOR')* * *verifyModel2Reqs('MODEL', [], [], 'CI')*   Use the *TreatAsTopMdl* option to handle the model as a top-level model, otherwise it is regarded as a referenced model. Use the third argument to optionally specify an author’s name. Use the *CI* option to operate within the context of continuous integration. This option suppresses launching of all reports upon completion. It also returns a result object that captures the operational outcome.  This utility creates the following:   * A test report named after the model with a postfix of *\_REQ\_Based\_Test\_Report* under *ISO\_04\_Design\MODEL \verification\_results\simulation\_results\RBT* * A coverage report named after the model with a postfix of *\_REQ\_Based\_Model\_Coverage\_Report* under *ISO\_04\_Design \MODEL\verification\_results\model\_coverages\RBT*   This utility depends on the following:  *ISO\_04\_Design\MODEL\test\_cases\RBT \MODEL\_REQ\_Based\_Test.mldatx* |
| *verifyObjCode2LowLevelTests* | Verify the executable object code of the model against structure-based tests generated by Simulink Design Verifier. All tests exercise the compiled code on the target computer via PIL simulations.  Syntax:   * *verifyObjCode2LowLevelTests('MODEL', 'SIL')* * *verifyObjCode2LowLevelTests('MODEL', 'PIL')* * *verifyObjCode2LowLevelTests('MODEL', 'SIL', 'DisableCoverage')* * *verifyObjCode2LowLevelTests('MODEL', 'SIL', [], 'TreatAsTopMdl')* * *verifyObjCode2LowLevelTests('MODEL', 'SIL', [], [], 'AUTHOR')* * *verifyObjCode2LowLevelTests('MODEL', 'SIL', [], [], [], 'CI')*   Use the second argument to control if the tests exercise the compiled code on the host computer via SIL simulations or the target computer via PIL simulations. Use the *DisableCoverage* option to suppress coverage analysis, otherwise the code is instrumented for coverage analysis. Use the *TreatAsTopMdl* option to handle the model as a top-level model, otherwise it is regarded as a referenced model. Use the fifth argument to optionally specify an author’s name. Use the *CI* option to operate within the context of continuous integration. This option suppresses launching of all reports upon completion. It also returns a result object that captures the operational outcome.  This utility creates the following (*SIL*):   * A test report named after the model with a postfix of *\_SIL\_SLDV\_Based\_Test\_Report* under *ISO\_05\_Code \verification\_results\eoc\_test\_results\MODEL\host\SBT* * A coverage report named after the model with a postfix of *\_SLDV\_Based\_Code\_Coverage\_Report* under *ISO\_05\_Code \verification\_results\code\_coverages\MODEL\host\SBT*   This utility depends on the following:  *ISO\_04\_Design\MODEL\test\_cases\SBT \MODEL\_SLDV\_Based\_Test.mldatx* |
| *verifyObjCode2Reqs* | Verify if the executable object code of the model complies with the software requirements. All tests exercise the compiled code on the target computer via PIL simulations.  Syntax:   * *verifyObjCode2Reqs('MODEL', 'SIL')* * *verifyObjCode2Reqs('MODEL', 'PIL')* * *verifyObjCode2Reqs('MODEL', 'SIL', 'DisableCoverage')* * *verifyObjCode2Reqs('MODEL', 'SIL', [], 'TreatAsTopMdl')* * *verifyObjCode2Reqs('MODEL', 'SIL', [], [], 'AUTHOR')* * *verifyObjCode2Reqs('MODEL', 'SIL', [], [], [], 'CI')*   Use the second argument to control if the tests exercise the compiled code on the host computer via SIL simulations or the target computer via PIL simulations. Use the *DisableCoverage* option to suppress coverage analysis, otherwise the code is instrumented for coverage analysis. Use the *TreatAsTopMdl* option to handle the model as a top-level model, otherwise it is regarded as a referenced model. Use the fifth argument to optionally specify an author’s name. Use the *CI* option to operate within the context of continuous integration. This option suppresses launching of all reports upon completion. It also returns a result object that captures the operational outcome.  This utility creates the following (*SIL*):   * A test report named after the model with a postfix of *\_SIL\_REQ\_Based\_Test\_Report* under *ISO\_05\_Code \verification\_results\eoc\_test\_results\MODEL\host\RBT* * A coverage report named after the model with a postfix of *\_REQ\_Based\_Code\_Coverage\_Report* under *ISO\_05\_Code \verification\_results\code\_coverages\MODEL\host\RBT*   This utility depends on the following:  *ISO\_04\_Design\MODEL\test\_cases\RBT \MODEL\_REQ\_Based\_Test.mldatx* |
| *checkCodeStds* | Check the generated code of the model against software code standards, and then generate the Bug Finder report.  Syntax:   * *checkCodeStds('MODEL')* * *checkCodeStds('MODEL', 'TreatAsTopMdl')* * *checkCodeStds('MODEL', [], 'CI')*   Use the *TreatAsTopMdl* option to handle the model as a top-level model, otherwise it is regarded as a referenced model. Use the *CI* option to operate within the context of continuous integration. This option suppresses launching of all reports upon completion. It also returns a result object that captures the operational outcome.  This utility creates the following:  A report named after the model with a postfix of *\_Code\_Standards\_Report* under *ISO\_05\_Code \verification\_results\code\_standard\_checks\MODEL*  This utility depends on the following:   * *open\_MODEL* * Code under *MODEL\_ert\_rtw* or *slprj\ert\MODEL* in *ISO\_05\_Code\specification* |
| *checkModelStds* | Check the model against software model standards, and then generate the Model Advisor report.  Syntax:   * *checkModelStds('MODEL')* * *checkModelStds('MODEL', 'TreatAsTopMdl')* * *checkModelStds('MODEL', [], 'CI')*   Use the *TreatAsTopMdl* option to handle the model as a top-level model, otherwise it is regarded as a referenced model. Use the *CI* option to operate within the context of continuous integration. This option suppresses launching of all reports upon completion. It also returns a result object that captures the operational outcome.  This utility creates the following:  A report named after the model with a postfix of *\_Model\_Advisor\_Report* under *ISO\_04\_Design\MODEL \verification\_results\design\_standard\_checks*  This utility depends on the following:  *open\_MODEL* |
| *computeCodeMetrics* | Compute metrics for the generated code of the model, and then generate the Bug Finder report.  Syntax:   * *computeCodeMetrics('MODEL')* * *computeCodeMetrics('MODEL', 'TreatAsTopMdl')*   Use the *TreatAsTopMdl* option to handle the model as a top-level model, otherwise it is regarded as a referenced model.  This utility creates the following:  A report named after the model with a postfix of *\_Code\_Metrics\_Report* under *ISO\_05\_Code\verification\_results \code\_metrics\MODEL*  This utility depends on the following:   * *open\_MODEL* * Code under *MODEL\_ert\_rtw* or *slprj\ert\MODEL* in *ISO\_05\_Code\specification* |
| *computeModelMetrics* | Compute metrics for the model, and then generate the Model Advisor report.  Syntax:   * *computeModelMetrics('MODEL')* * *computeModelMetrics('MODEL', 'TreatAsTopMdl')*   Use the *TreatAsTopMdl* option to handle the model as a top-level model, otherwise it is regarded as a referenced model.  This utility creates the following:  A report named after the model with a postfix of *\_Model\_Metrics\_Report* under *ISO\_04\_Design\MODEL \verification\_results\model\_metrics*  This utility depends on the following:  *open\_MODEL* |
| *detectCodingErrs* | Detect coding errors in the generated code of the model, and then generate the Bug Finder report.  Syntax:   * *detectCodingErrs('MODEL')* * *detectCodingErrs('MODEL', 'TreatAsTopMdl')* * *detectCodingErrs('MODEL', [], 'CI')*   Use the *TreatAsTopMdl* option to handle the model as a top-level model, otherwise it is regarded as a referenced model. Use the *CI* option to operate within the context of continuous integration. This option suppresses launching of all reports upon completion. It also returns a result object that captures the operational outcome.  This utility creates the following:  A report named after the model with a postfix of *\_Bug\_Finder\_Report* under *ISO\_05\_Code\verification\_results \coding\_error\_detections\MODEL*  This utility depends on the following:   * *open\_MODEL* * Code under *MODEL\_ert\_rtw* or *slprj\ert\MODEL* in *ISO\_05\_Code\specification* |
| *detectDesignErrs* | Detect design errors or dead logic in the model, and then generate the Design Verifier report.  Syntax:   * *detectDesignErrs('MODEL')* * *detectDesignErrs('MODEL', 'DetectActiveLogic')* * *detectDesignErrs('MODEL', [], 300)* * *detectDesignErrs('MODEL', [], [], 'CI')*   Use the *DetectActiveLogic* option to enable active logic analysis, otherwise the model is not analyzed for active logic. Use the third argument to control the time limit of the analysis. Use the *CI* option to operate within the context of continuous integration. This option suppresses launching of all reports upon completion. It also returns a result object that captures the operational outcome.  This utility creates one of the following:  A report named after the model with a postfix of *\_Design\_Error\_Detection\_Report* under *ISO\_04\_Design \MODEL\verification\_results\design\_error\_detections*  This utility depends on the following:  *open\_MODEL* |
| *proveCodeQuality* | Prove the absence of defects in the generated code of the model, and then generate the Code Prover report.  Syntax:   * *proveCodeQuality('MODEL')* * *proveCodeQuality('MODEL', 'TreatAsTopMdl')* * *proveCodeQuality('MODEL', 'IncludeAllChildMdls')* * *proveCodeQuality('MODEL', 'CI')* * *proveCodeQuality('MODEL', 'TreatAsTopMdl', 'IncludeAllChildMdls', 'CI')*   Use the *TreatAsTopMdl* option to handle the model as a top-level model, otherwise it is regarded as a referenced model. Use the *IncludeAllChildMdls* option to include the generated code of all referenced models for analysis. Use the *CI* option to operate within the context of continuous integration. This option suppresses launching of all reports upon completion. It also returns a result object that captures the operational outcome.  This utility creates the following:  A report named after the model with a postfix of *\_Code\_Prover\_Report* under *ISO\_05\_Code\verification\_results \code\_proving\MODEL*  This utility depends on the following:   * *open\_MODEL* * Code under *MODEL\_ert\_rtw* or *slprj\ert\MODEL* in *ISO\_05\_Code \specification* |