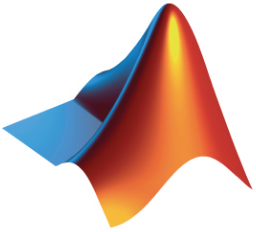
**DO Startup Kit**

User Guide

**R2017b**



**Revision History**

October 2017 New for Version 1.0.0 (Release R2017a)

December 2017 Major updates for 1.1.0 (Release 2017b)

January 2018 Minor updates for 1.1.1 (Release 2017b)

February 2018 Minor updates for 1.1.2 (Release 2017b)

Table of Contents

[Release Notes 4](#_Toc505850475)

[Introduction 5](#_Toc505850476)

[Product Description 5](#_Toc505850477)

[System Requirements 6](#_Toc505850478)

[Installation and Configuration 7](#_Toc505850479)

[Software Installation 7](#_Toc505850480)

[Installed Files and Folders 13](#_Toc505850481)

[Software Uninstallation 14](#_Toc505850482)

[Getting Started 16](#_Toc505850483)

[Creating a New DO Project 16](#_Toc505850484)

[Navigating the DO Project 18](#_Toc505850485)

[Reconfiguring the Standard Model Configurations 19](#_Toc505850486)

[Adding a New Model 21](#_Toc505850487)

[Case Study 22](#_Toc505850488)

[Creating the Case Study Project 22](#_Toc505850489)

[Navigating the Case Study Project (Before Exercising Workflow Demonstration) 24](#_Toc505850490)

[Exercising the Workflow Demonstration in the Case Study Project 27](#_Toc505850491)

[Navigating the Case Study Project (After Exercising Workflow Demonstration) 28](#_Toc505850492)

[Utility Functions 31](#_Toc505850493)

# Release Notes

Version 1.0.0

* New for MATLAB R2017a.

Version 1.1.0

* New for MATLAB R2017b.

# Introduction

## Product Description

The Do Startup Kit is a template of Simulink Project that is designed to store all documents, requirements, designs, code, tests, results, and tools that are relevant for DO-178C applications. This Simulink Project template is prepopulated with:

* Planning document templates
* Recommended modeling and coding standards
* Design constraints in the forms of libraries, model templates, model configurations, and checks
* Tooling for task automation

The goals of the DO Startup Kit is to:

* Streamline adoption of Model-Based Design for software development by
* Preventing incompatibilities among tools
* Making “what to do” and “how to do” obvious
* Provide tools and utilities to exercise applicable DO objectives
* Provide a sample project to illustrate the Model-Based Design workflow for DO-178C

## System Requirements

* Operating system: Windows® (64-bit only), Linux® (64-bit only). Note that MATLAB is no longer available for 32-bit Windows starting from R2016a.
* MATLAB R2017b.
* 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™
  + Simulink Code Inspector™
  + Polyspace Bug Finder™
  + Polyspace Code Prover™
* External software requirements:
  + Supported C/C++ compiler per <https://www.mathworks.com/support/compilers.html>.

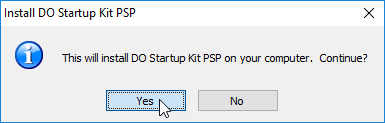
# Installation and Configuration

## Software Installation

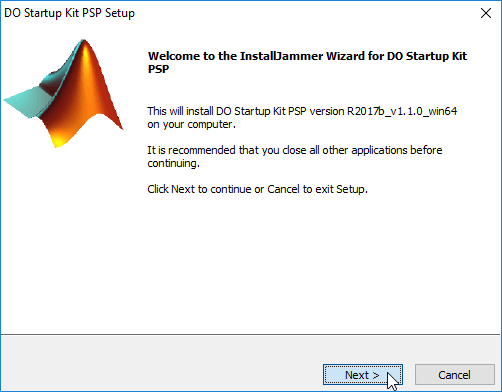
* To install the DO Startup Kit that extends MATLAB R2017b on Windows, run the installer *DOStartupKitPSP-R2017b\_vX.X.X\_win64-Install.exe*.
* You will be asked to select a destination folder when running the installer. Select the root folder of your MATLAB installation as the destination folder. If necessary, you can use the MATLAB command *matlabroot* to determine the root folder where MATLAB is installed.
* Note that if the DO Startup Kit is already installed, you must first uninstall it by running the uninstaller *DOStartupKitPSP-R2017b\_vX.X.X\_win64-Uninstall.exe*. You can find the uninstaller at *matlabroot\uninstall* (e.g., *C:\Program Files\MATLAB\R2017b\uninstall*). See the uninstallation instructions below.

Here are the instructions to run the installer:

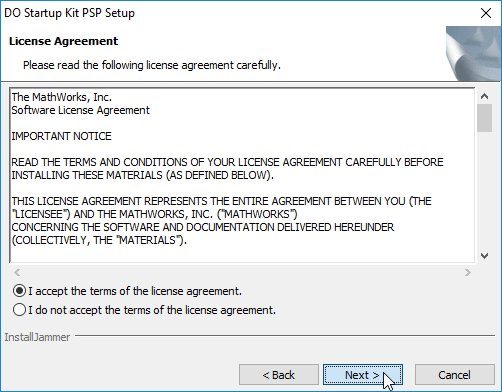
1. Double-click *DOStartupKitPSP-R2017b\_vX.X.X\_win64-Install.exe* to launch the installer.
2. Click *Yes* when the Install DO Startup Kit PSP dialog appears.



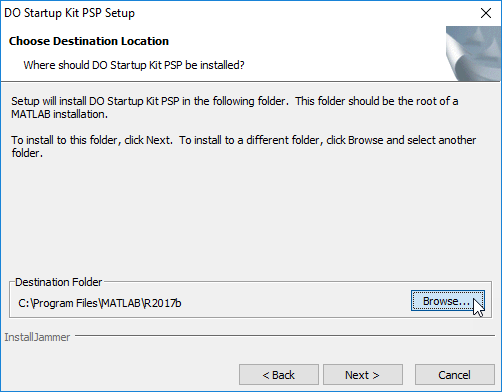
1. When the *Welcome to the InstallJammer Wizard for DO Startup Kit PSP* screen appears in the DO Startup Kit PSP Setup dialog, Click *Next >* to start the installation process.



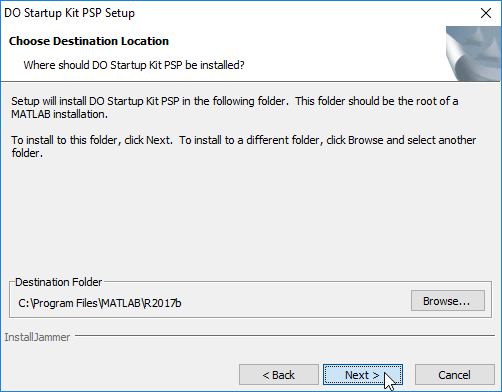
1. When the *License Agreement* screen appears in the DO Startup Kit PSP Setup dialog, read and accept the license agreement, and then click *Next >*.



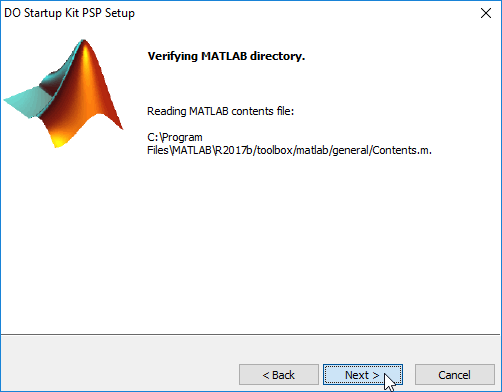
1. When the *Choose Destination Location* screen appears, click *Browse* to select a destination folder. This must be the root folder of your MATLAB R2017b installation.



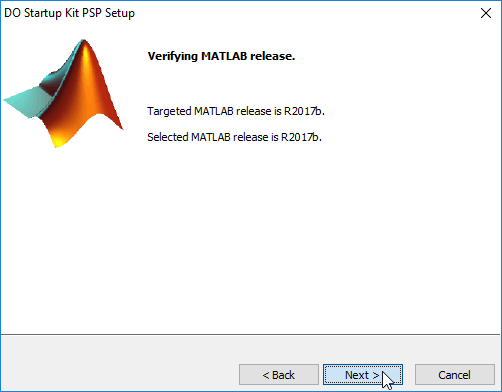
1. Click *Next >* after selecting a destination folder.



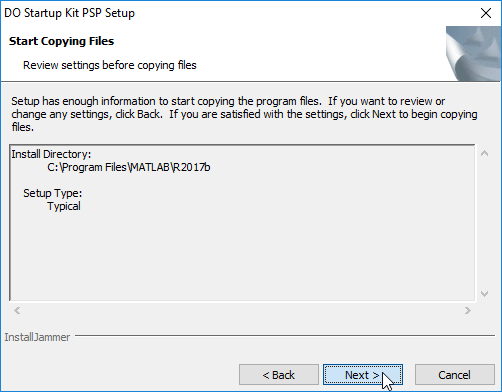
1. The selected MATLAB installation must be verified. When the *Verifying MATLAB Directory* screen appears, click *Next >* to start the release verification step.



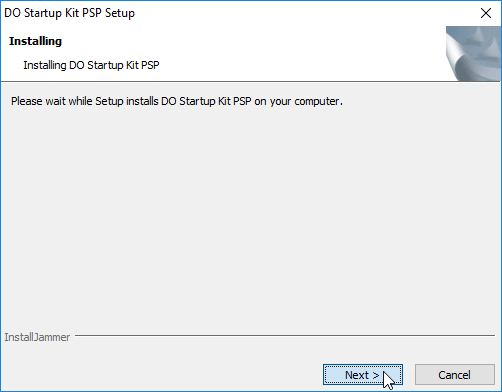
1. Upon verification, the result is displayed in the *Verifying MATLAB Release* screen. If the selected MATLAB release matches the targeted release, click *Next >*. Otherwise, you must return to the *Choose Destination Location* screen to re-select the targeted release.



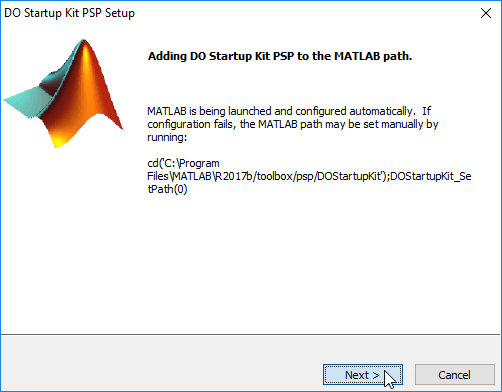
1. When the *Start Copying Files* screen appears, click *Next >* to start file extraction.



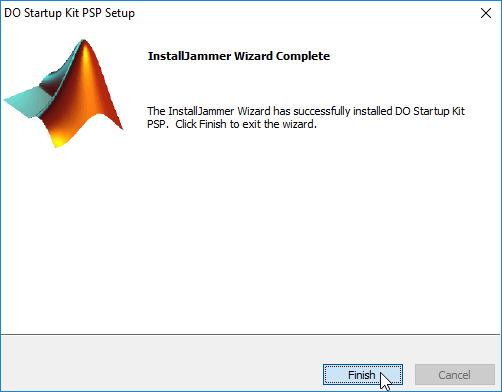
1. The *Installing* screen appears after all files are successfully copied. Click *Next >* to start the configuration process. The configuration process sets up the path so that the DO Startup Kit is available for use when you launch MATLAB next time. Because this step automatically launches MATLAB to perform path addition, it may take a few minutes before the next screen appears.



1. When the *Adding DO Startup Kit PSP to the MATLAB Path* screen appears, click *Next >*.



1. If the installation is successful, you will be brought to the *InstallJammer Wizard Complete* screen. Click *Finish* to close the installer.



## Installed Files and Folders

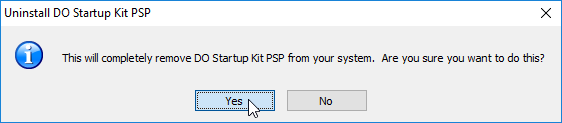
When using the DO Startup Kit, you might find it helpful to know the location of related files and folders as explained below:

* The DO Startup Kit uses the directories and files located in *matlabroot\toolbox\psp\DOStartupKit*. This folder contains project templates that provide a framework for collaborative development of your DO applications. The main template *DOProject.sltx* creates a Simulink Project that streamlines the adoption of Model-Based Design for software development per DO-178C/DO-331.
* The folder *matlabroot\toolbox\psp\DOStartupKit\examples* contains an example template. The example template *DOProjectExample.sltx* creates a Simulink Project for use to demonstrate features of the DO Startup Kit.

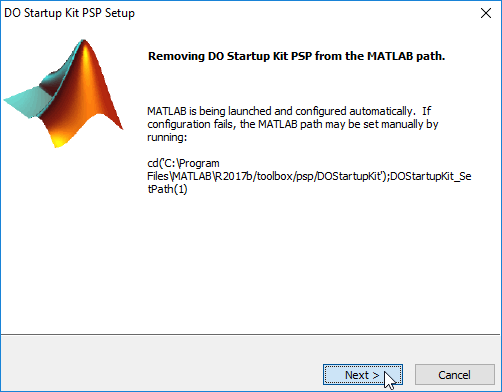
## Software Uninstallation

An uninstaller named *DOStartupKitPSP-R2017b\_vX.X.X\_win64-Uninstall.exe* is automatically created and placed in *matlabroot\uninstall* upon a successful installation. Here are the instructions to remove the DO Startup Kit from MATLAB using the uninstaller:

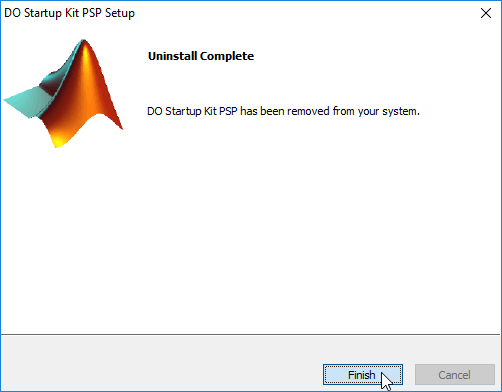
1. Double-click *DOStartupKitPSP-R2017b\_vX.X.X\_win64-Uninstall.exe* to launch the uninstaller.
2. Click *Yes* when the Uninstall DO Startup Kit PSP dialog appears.



1. When the *Removing DO Startup Kit PSP from the MATLAB Path* screen appears in the DO Startup Kit PSP Setup dialog, click *Next >* to start the uninstallation process.



1. If the uninstallation is successful, you will be brought to the *Uninstall Complete* screen. Click *Finish* to close the uninstaller.

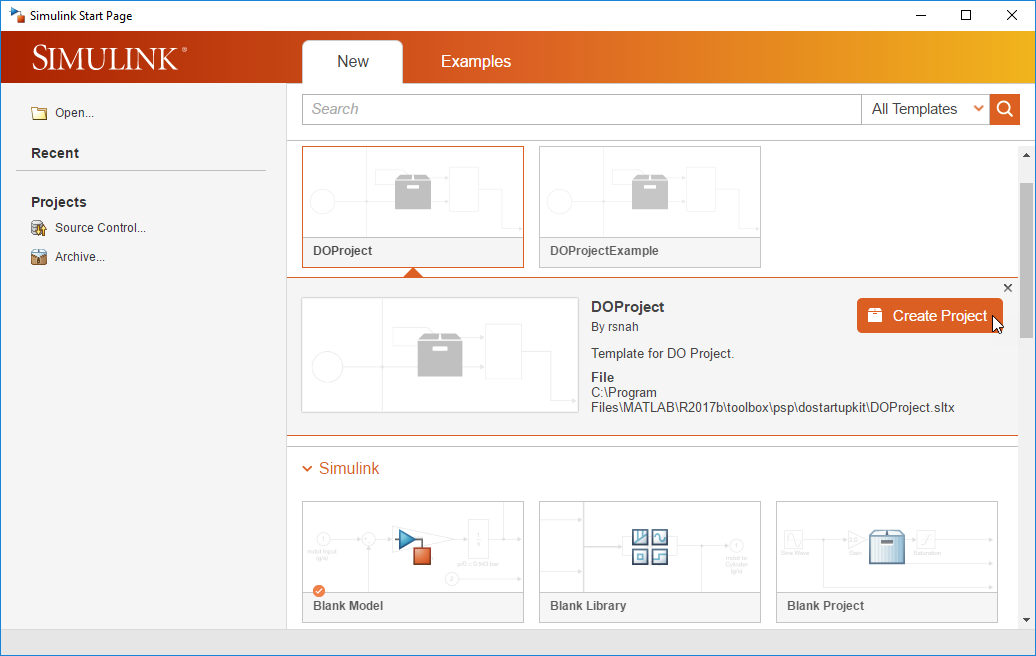


# Getting Started

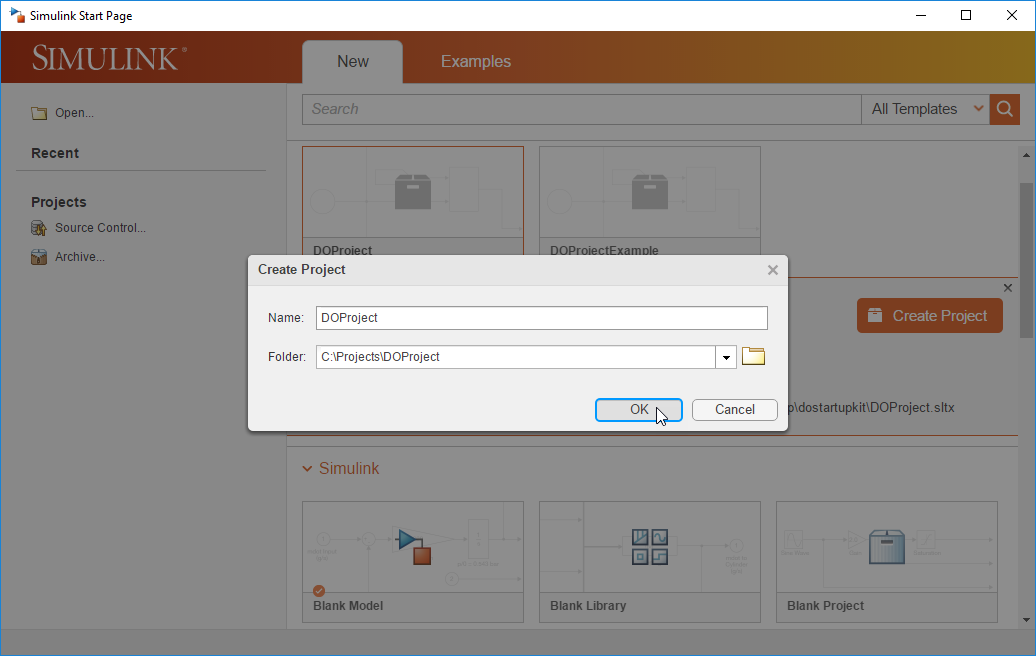
## Creating a New DO Project

The DO Startup Kit adds a project that is suitable for DO application development as a template to your MATLAB installation. 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. On the MATLAB *Home* tab, click *Simulink* to open the Simulink Start Page.
2. In the Simulink Start Page, click *DOProject* under the *My Templates* panel, and then click *Create Project*. This opens the Create Project dialog.



1. In the Create Project dialog, enter the *Project Name* and *Project Folder*, and then click *OK*. This creates a new project based on the *DOProject* template.



## Navigating the DO Project

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

* *DO\_01\_Planning*
* In the *plans* subfolder, you will find templates for Plan for Software Aspects of Certification (PSAC), Software Development Plan (SDP), Software Verification Plan (SVP), Software Configuration Management Plan (SCMP), and Software Quality Assurance Plan (SQAP)
* In the *standards* subfolder, you will find templates for Software Requirements Standards (SRS), Software Design Standards (SDS), Software Code Standards (SCS), and Software Model Standards (SMS).
* *DO\_02\_Requirements*
* In the *specification* subfolder, you will store all documents that capture your high-level software requirements and derived requirements.
* In the *verification* subfolder, you will store checklists related to all review activities of the high-level software requirements and derived requirements.
* *DO\_03\_Design*
* In the *common* subfolder, you will create block libraries and data 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, and store artifacts related to all verification activities of the model in the *verification* subfolder.
* *DO\_04\_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* subfolder, you will store artifacts related to all verification activities of the source code and executable object code.
* *patches*

You will find the necessary software patches in this folder. Follow the instructions given in *readme.txt* to install the software patches.

* *tools*
* In the *checks* subfolder, you will find the configuration files for Model Advisor checks and Polyspace MISRA checks designed to complement the Software Code Standards (SCS) and Software Model Standards (SMS) provided in this kit.
* In the *compatible\_blocks* subfolder, you will a block library that holds all primitive blocks that are compatible with Simulink Code Inspector, which serves as the software design constraints.
* In the *templates* subfolder, you will find model configurations that are compatible with the software design constraints and software model standards. You will also find a model template that apply one of these configurations.
* In the *utilities* subfolder, you will find utilities for project management, report generation, code generation, and model/code verification.
* *work*

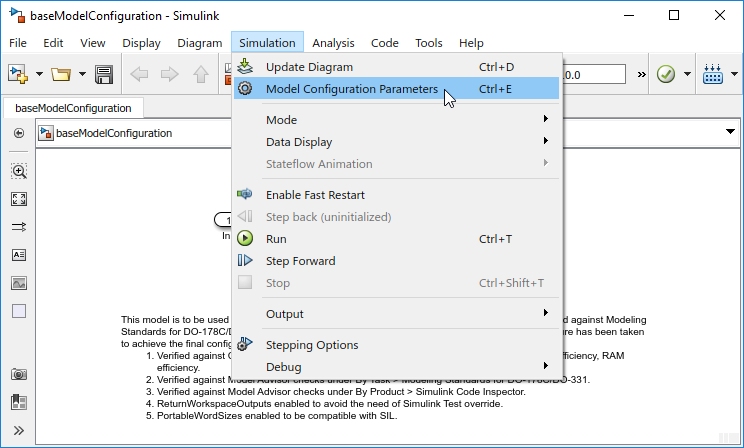
The default working folder. This folder also contains the *cache* folder in which model build artifacts used for simulation are stored.

## Reconfiguring the Standard Model Configurations

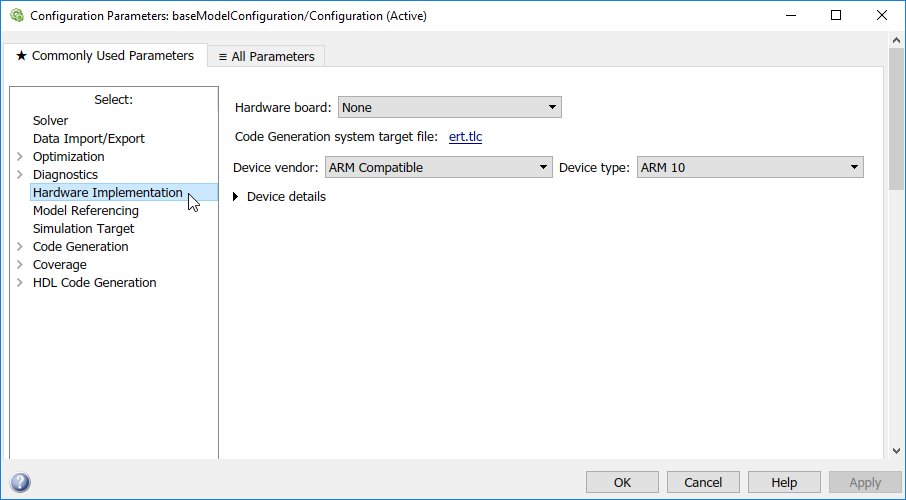
Every time when you launch the project, notice that there are two model configurations loaded into the workspace automatically. These model configurations are meant to be used by all models in the project. For reusable (multi-instantiable) models, use the configuration *csMultiInstance*. Otherwise, use the configuration *csSingleInstance*. Except for the *ModelReferenceNumInstancesAllowed* property, both *csMultiInstance* and *csSingleInstance* have the same settings. Both configurations, defined by *reusableModelConfig.m* and *nonreusableModelConfig.m*, derive their standardized settings from *doConfig.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,

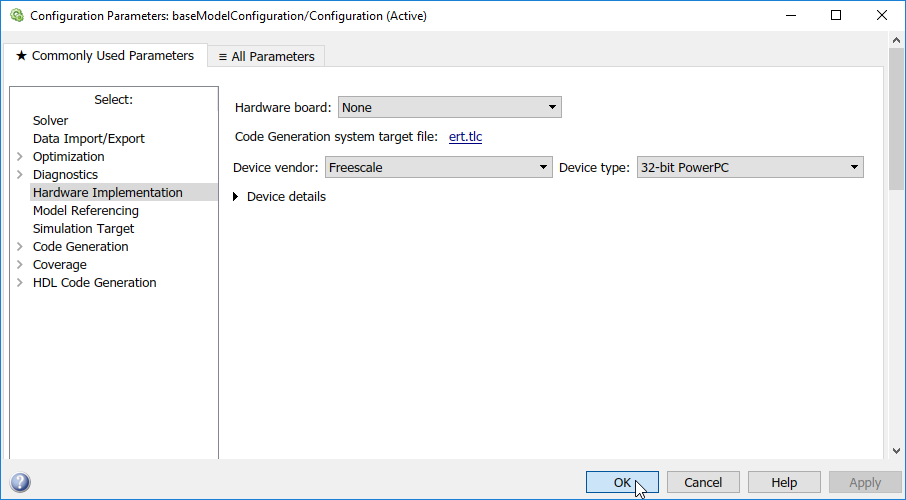
1. Enter *baseModelConfiguration* to open the model.
2. Select *Simulation > Model Configuration Parameters* to open the Configuration Parameters dialog.



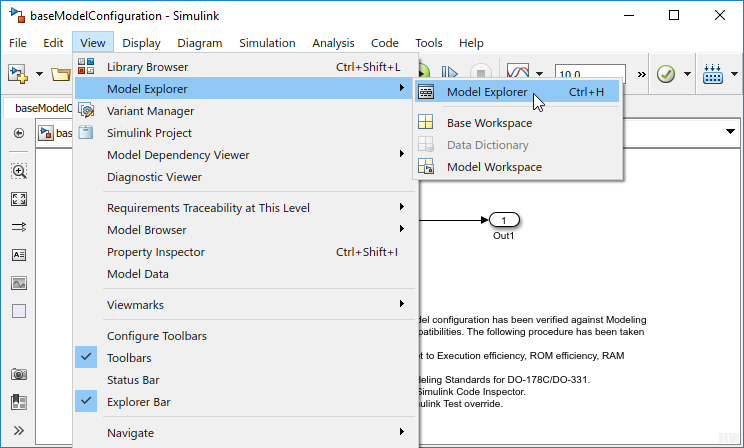
1. In the Configuration Parameters dialog, click the *Hardware Implementation* item to open the *Hardware Implementation* pane.



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



1. Save the model.
2. Select *View > Model Explorer > Model Explorer* to open the Model Explorer.



1. 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 *doConfig.m*, overwriting the existing settings in the file:

*>> configSet.saveAs('..\tools\templates\model\_configurations\doConfig');*

**Note:**

Do NOT use *clear* to clear the MATLAB workspace because it will clear *csMultiInstance* and *csSingleInstance* as well. Use the utility *resetWorkspace* instead.

## Adding a New Model

Based on the high-level software requirements, you create models that serve as the low-level software requirements. This kit expects that each model has its own model folder under the *DO\_03\_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 the following command:

*>> addModelFolder('controller')*

This command creates a new *controller* folder under *DO\_03\_Design*. Like *sample\_model*, this new folder contains a *specification* and a *verification* subfolders. In the *specification* subfolder, you will find an empty model *controller.slx*. By default, the empty model uses *csMultiInstance* as its model configuration. If this model is not meant to be reusable, you should switch its model configuration to *csSingleInstance*. You are now ready to build the model using permissible blocks from the library *doLib*. All blocks from *doLib* are accessible from the Simulink Library Browser under *DO-178C/DO-331 Primitive Library*.

When constructing a model, you often have to define data in the workspace to help characterize its behavior. Keep in mind that data in the workspace 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. To hold model specific data, you should create a MATLAB script that is named after the model with a prefix of *DD\_* in the *specification\data* subfolder of the model folder. If the model is reusable (multi-instantiable), you may actually create instant specific data in the model workspace. To hold instant specific data, you should create a separate MATLAB script that is named after the model with a prefix of *localDD\_* in the *specification\data* subfolder of the model folder. Using the example above, you would create a *DD\_controller.m* and/or *localDD\_controller.m* in *DO\_03\_Design\controller\specification\data*. For shared data, you create a MATLAB script named *DD\_common* in *DO\_03\_Design\common\data*.

Upon constructing the model and creating its data files, you should create a complementary MATLAB script that can be used to open the model. The script is required if you intend to leverage utilities in the DO Startup Kit to perform downstream tasks. This script must be named after the model with a prefix of *open\_*. It should load all data that is consumed by the model before opening the model. Using the example above, you would create an *open\_controller.m* to achieve this goal.

# Case Study

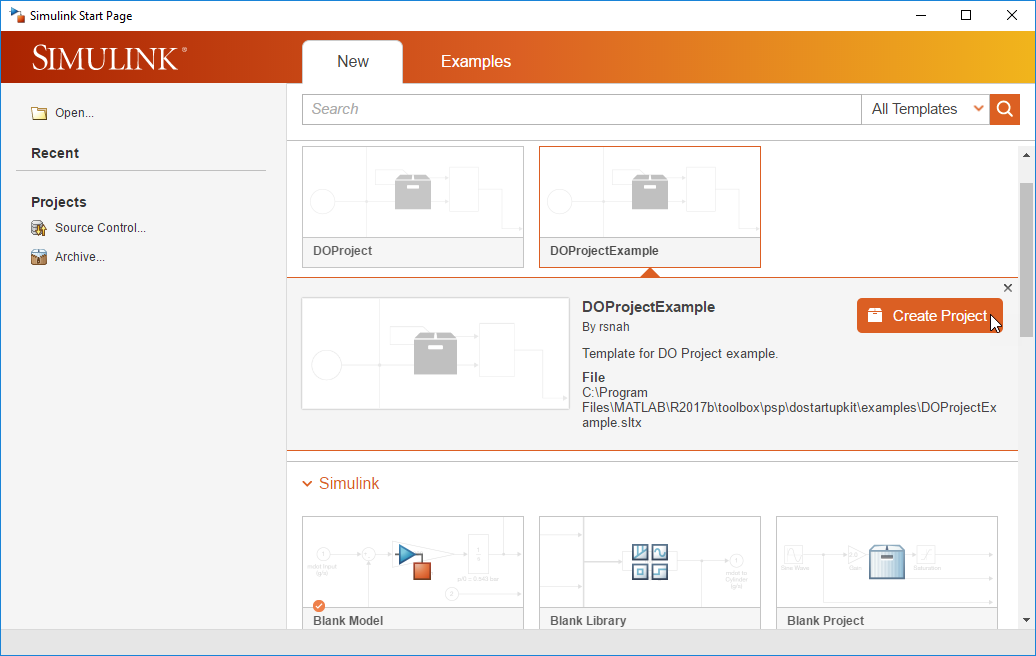
**Note:**

Please follow the instructions given in *patches\readme.txt* to install the required software patches before proceeding.

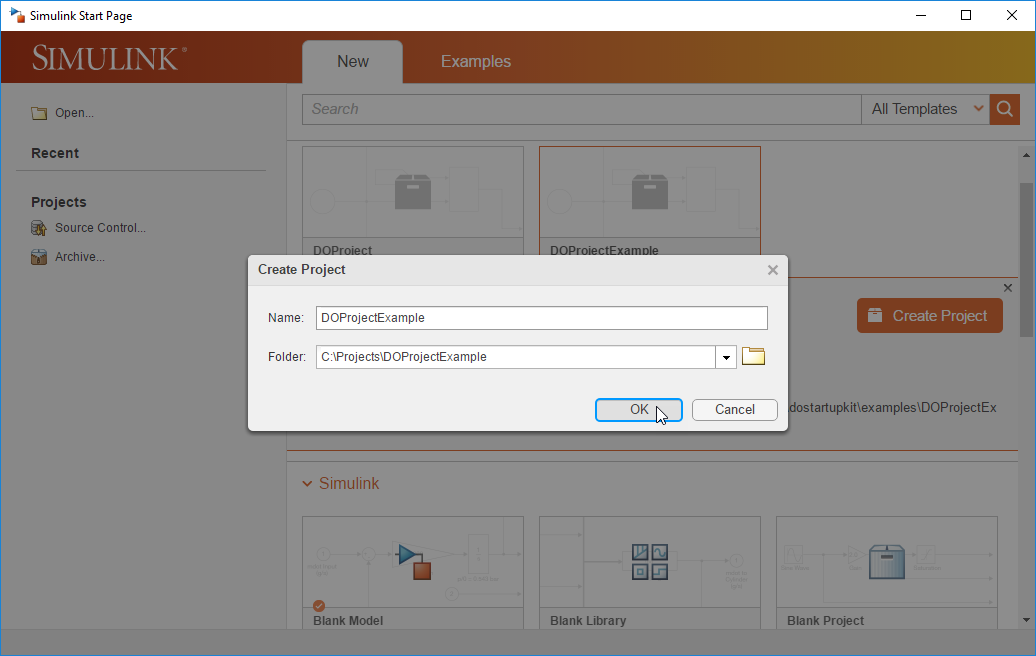
## Creating the Case Study Project

The DO Startup Kit adds a project that illustrates the Model-Based Design workflow for DO-178C as a template to your MATLAB installation. To access the case study in this project, you must create a new project based on this template. To do this,

1. On the MATLAB *Home* tab, click *Simulink* to open the Simulink Start Page.
2. In the Simulink Start Page, click *DOProjectExample* under the *My Templates* panel, and then click *Create Project*. This opens the Create Project dialog.



1. In the Create Project dialog, enter the *Project Name* and *Project Folder*, and then click *OK*. This creates a new project based on the *DOProjectExample* template.



## Navigating the Case Study 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 raw DO Project. The example project does have the following additional contents used to explain the workflow and the available utilities:

* Under *DO\_02\_Requirements*
* In the *specification* subfolder, you will find documents and trace data for system requirements and high-level software requirements. Authored using the Requirement Editor, both system requirements and high-level software requirements are captured in requirement sets *SR.slreqx* and *HLR.slreqx*, respectively. Along with the *.slreqx* files, you will also find a *.slmx* file that stores the trace data between the high-level software requirements and the associated system requirements.
* Under *DO\_03\_Design*
* In the *common\block\_libraries\specification* subfolder, you will find two user libraries *ControlLib.slx* and *FilterLib.slx*. As part of the convention when using the DO Startup Kit, notice that each library has a complementary MATLAB script for use to open the library.
* There are 5 model folders, all derived from the *sample\_model* folder template. These folders contain the low-level software requirements, in the form of Simulink models, that trace to the high-level software requirements captured in *DO\_02\_Requirements\specification\HLR.slreqx*.
* Software architecture is typically derived from how the design is organized. With Model-Based Design, software architecture is inherently established by the model hierarchy of the design, based on natural grouping of the high-level software requirements. In this case study, the design is subdivided into several models to satisfy the various requirements in *HLR.slreqx*. Within the model hierarchy of the design, *Flight\_Control* is the top-level model that implements the overall control strategy, while the rest are lower-level models that characterize various parts of the design. Note that *Actuator\_Control* is created as a reusable model per organization of the design.
* In the *specification* subfolder of each 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 high-level software requirements. As part of the convention when using the DO Startup Kit, notice that every model has a complementary MATLAB script for use to open the model.
* In the *specification\data* subfolder of each model folder, you will find a data dictionary file that defines required data specific to the model if there is any.
* The *specification\documents* subfolder of each model folder is used for holding a design description document of the model. This folder is empty initially.
* In the *verification* subfolder of each 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\model\_reviews* subfolder of each model folder is used for holding design review checklists of the model. This folder is empty.
* The *verification\simulation\_tests* subfolder of each model folder contains a *high\_level\_tests* folder and a *low\_level\_tests* folder that are used for holding simulation test results of the model.
* The *high\_level\_tests* folder is used for keeping test cases developed to verify the model against the high-level requirements it implements. In this folder, you create a Simulink Test file to manage all high-level test cases. This test file must be named after the model with a postfix of *\_REQ\_Based\_Test*. In this case study, we have developed three high-level test cases for *AHRS\_Voter*. You will find them in *AHRS\_Voter\_REQ\_Based\_Test.mldatx* under *AHRS\_Voter\verification\simulation\_tests\high\_level\_tests*. These test cases utilize test harnesses *AHRS\_Voter\_Harness\_HLR\_11.slx*, *AHRS\_Voter\_Harness\_HLR\_12.slx*, and *AHRS\_Voter\_Harness\_HLR\_13.slx* to verify *AHRS\_Voter* against requirements *HLR\_11*, *HLR\_12*, and *HLR\_13*, respectively. You can find these externally stored test harnesses under *AHRS\_Voter\specification*.
* The *low\_level\_tests* folder is used for keeping test cases derived from the model. These low-level tests are often needed to supplement the high-level tests to achieve the necessary test coverage. This folder is empty initially.
* The *verification\model\_coverages* subfolder of each model folder contains a *high\_level\_tests* folder and a *low\_level\_tests* folder that are used for holding coverage results of the model. These folders are empty initially.
* The *verification\design\_error\_detections* subfolder of each model folder contains a *dead\_logic* folder and a *design\_error* folder that are used for holding Simulink Design Verifier analysis results of the model. These folders are empty initially.
* The *verification\design\_standard\_checks* subfolder of each model folder is used for holding Model Advisor check results of the model. This folder is empty initially.
* The *verification\model\_metrics* subfolder of each model folder is used for holding design metrics of the model. This folder is empty initially.
* Under *DO\_04\_Code*
* In the *verification* subfolder, you will folders for the various code verification tasks. These tasks include code review, 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\code\_reviews* subfolder contains a folder for each model that is used for holding Simulink Code Inspector verification results of the generated code. The folder created for each model is empty initially.
* The *verification\executable\_tests* subfolder contains a folder for each model that is used for holding software-in-the-loop and hardware-in-the-loop test results of the generated code. The folder created for each model is empty initially.
* The *verification\code\_coverages* subfolder contains a folder for each model that is used for holding coverage results of the generated code. The folder created for each model is empty initially.
* The *verification\coding\_error\_detections* subfolder contains a folder for each model that is used for holding Bug Finder analysis results of the generated code. The folder created for each model is empty initially.
* The *verification\code\_proving* subfolder contains a folder for each model that is used for holding Code Prover verification results of the generated code. The folder created for each model is empty initially.
* The *verification\code\_standard\_checks* subfolder contains a folder for each model that is used for holding Bug Finder coding rule checker results of the generated code. The folder created for each model is empty initially.
* The *verification\code\_metrics* subfolder contains a folder for each model that is used for holding code metrics of the generated code. The folder created for each model is empty initially.
* Under *work*
* This folder contains additional system-level models and MATLAB scripts created for use in this case study to illustrate the Model-Based Design workflow for DO-178C.
* In the *html* subfolder, you will find a published markup script *Heli\_demo.html*. This file contains instructions to the workflow demonstration.

## Exercising the Workflow Demonstration in the Case Study Project

After launching the case study project, you can simply follow the instructions given in *work\html\Heli\_demo.html* to access the workflow demonstration. To do this,

1. In the *Files* view of the opened project, expand *work* and then *html*.
2. Double-click *Heli\_demo.html* top open it.
3. Following the explanations given in the file, use its 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. Note that the hyperlinks in *Heli\_demo.html* actually execute the applicable DO Startup Kit utilities via MATLAB scripts in the *work* folder.

## Navigating the Case Study Project (After Exercising Workflow Demonstration)

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

* Under *DO\_03\_Design*
* In the *specification* subfolder of each model folder,
* In the *specification\documents* subfolder of each model folder, you will find the design description document of the model. This report is generated from the model using *genSDD* (*tools\utilities\report\_generation\genSDD.m*).
* In the *verification\simulation\_tests\high\_level\_tests* subfolder of *AHRS\_Voter* (test cases for the other models are not available), you will find the high-level simulation test results and report of *AHRS\_Voter*. These test results are generated from running the test cases that verify *AHRS\_Voter* against the high-level requirements it implements using *verifyModel2Reqs* (*tools\utilities\verification\dynamic\_testing\verifyModel2Reqs.m*).
* In the *verification\simulation\_tests\low\_level\_tests* subfolder of *AHRS\_Voter*, you will find the low-level test generation results and report of *AHRS\_Voter*. These tests are generated from *AHRS\_Voter* to satisfy objectives that are not achieved in its high-level test coverage using *genLowLevelTests* (*tools\utilities\verification\dynamic\_testing\genLowLevelTests.m*). Note that the low-level test cases are automatically composed in *AHRS\_Voter\_SLDV\_Based\_Test.mldatx*. This generated test file has the following dependencies:
* A test harness *AHRS\_Voter\_Harness\_SLDV.slx* created under *AHRS\_Voter\specification*.
* A set of baseline data files created under *sl\_test\_baselines*.
* An input data file *AHRS\_Voter\_sldvdata.mat*.
* In the *verification\simulation\_tests\low\_level\_tests* subfolder of *AHRS\_Voter*, you will also find the low-level simulation test results and report of *AHRS\_Voter*. These test results are generated from running the test cases derived from *AHRS\_Voter* to address unfulfilled objectives in its high-level test coverage using *verifyModel2LowLevelTests* (*tools\utilities\verification  
  \dynamic\_testing\verifyModel2LowLevelTests.m*).
* In the *verification\model\_coverages\high\_level\_tests* subfolder of *AHRS\_Voter*, you will find the high-level model coverage results and report of *AHRS\_Voter*. These coverage results are collected from running the test cases that verify *AHRS\_Voter* against the high-level requirements it implements using *verifyModel2Reqs* (*tools\utilities\verification  
  \dynamic\_testing\verifyModel2Reqs.m*).
* In the *verification\model\_coverages\low\_level\_tests* subfolder of *AHRS\_Voter*, you will find the low-level model coverage results and report of *AHRS\_Voter*. These coverage results are collected from running the test cases derived from *AHRS\_Voter* to address unfulfilled objectives in its high-level test coverage using *verifyModel2LowLevelTests* (*tools\utilities\verification  
  \dynamic\_testing\verifyModel2LowLevelTests.m*).
* In the *verification\model\_coverages* subfolder of *AHRS\_Voter*, you will find the cumulative model coverage report of *AHRS\_Voter*. These coverage results are assembled from merging both the high-level and low-level model coverage results using *mergeModelCoverage* (*tools\utilities\verification\dynamic\_testing\mergeModelCoverage.m*).
* In the *verification\design\_error\_detections\dead\_logic* subfolder of each model folder, you will find the dead logic detection report of the model. This report is generated from examining the model against dead logic with Simulink Design Verifier using *detectDesignErrs* (*tools\utilities  
  \verification\static\_analysis\detectDesignErrs.m*).
* In the *verification\design\_error\_detections\design\_error* 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 run-time errors with Simulink Design Verifier using *detectDesignErrs* (*tools\utilities\verification\static\_analysis\detectDesignErrs.m*).
* In the *verification\design\_standard\_checks* subfolder of each model folder, you will find the modeling standard compliance report of the model. This report is generated from examining the model against Model Advisor checks configured in *tools\checks\doChecks.mat* using *checkModelStds* (*tools\utilities\verification\static\_analysis\checkModelStds.m*).
* Under *DO\_04\_Code*
* In the *specification* subfolder, you will find the generated code of all five models developed for use in this cases study. This source code is generated from the models using *genSrcCode* (*tools\utilities\code\_generation\genSrcCode.m*). Note that the code for the top-level model is placed under *Flight\_Control\_ert\_rtw* while the code for each of the lower-level models is placed under *slprj\ert* along with the shared code.
* In the *verification\code\_reviews* subfolder, you will find the code inspection report of the generated code for each model. This report is generated from inspecting the source code of each model with Simulink Code Inspector using *verifySrcCode2Model* (*tools\utilities\verification  
  \static\_analysis\verifySrcCode2Model.m*).
* In the *verification\executable\_tests\AHRS\_Voter\host\high\_level\_tests*, you will find the high-level software-in-the-loop test results and report of the generated code for *AHRS\_Voter*. These test results are generated from running the test cases that verify *AHRS\_Voter* against the high-level requirements it implements in software-in-the-loop mode using *verifySrcCode2Reqs* (*tools\utilities\verification\dynamic\_testing\verifySrcCode2Reqs.m*).
* In the *verification\executable\_tests\AHRS\_Voter\host\low\_level\_tests*, you will find the low-level software-in-the-loop test results and report of the generated code for *AHRS\_Voter*. These test results are generated from running the test cases derived from *AHRS\_Voter* to complement its high-level test coverage in software-in-the-loop mode using *verifySrcCode2LowLevelTests* (*tools\utilities\verification\dynamic\_testing\verifySrcCode2LowLevelTests.m*).
* In the *verification\code\_coverages\AHRS\_Voter\high\_level\_tests* subfolder, you will find the high-level code coverage results and report of the generated code for *AHRS\_Voter*. These coverage results are collected from running the test cases that verify *AHRS\_Voter* against the high-level requirements it implements in software-in-the-loop mode using *verifySrcCode2Reqs* (*tools\utilities\verification\dynamic\_testing\verifySrcCode2Reqs.m*).
* In the *verification\code\_coverages\ AHRS\_Voter\low\_level\_tests* subfolder, you will find the low-level code coverage results and report of the generated code for *AHRS\_Voter*. These coverage results are collected from running the test cases derived from *AHRS\_Voter* to complement its high-level test coverage in software-in-the-loop mode using *verifySrcCode2LowLevelTests* (*tools\utilities\verification\dynamic\_testing  
  \verifySrcCode2LowLevelTests.m*).
* In the *verification\code\_coverages\AHRS\_Voter* subfolder, you will find the cumulative code coverage report of the generated code for *AHRS\_Voter*. These coverage results are assembled from merging both the high-level and low-level code coverage results using *mergeCodeCoverage* (*tools\utilities\verification\dynamic\_testing\mergeCodeCoverage.m*).
* In the *verification\code\_proving\Flight\_Control* subfolder, you will find the Code Prover verification report of the generated code for *Flight\_Control* and its lower-level models. This report is generated from checking the source code of *Flight\_Control* and its lower-level models against run-time errors with Polyspace Code Prover using *proveCodeQuality* (*tools\utilities  
  \verification\static\_analysis\proveCodeQuality.m*).
* In the *verification\code\_standard\_checks* subfolder, you will find the coding standard compliance report of the generated code for each model. This report is generated from checking the source code of each model against Bug Finder coding rules configured in *tools\checks\MISRA\_C\_2012\_ACG* using *checkCodeStds* (*tools\utilities\verification  
  \static\_analysis\checkCodeStds.m*).

# Utility Functions

The following table summarizes the utilities available in the DO Startup Kit:

|  |  |
| --- | --- |
| **Utility Name** | **Description** |
| *addModelFolder* | Create a new model folder based on the folder template *DO\_03\_Design\sample\_model*. This also creates model specific folders under the subfolders in *DO\_04\_Code\verification*.  Syntax:  *addModelFolder('MODEL')* |
| *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')* |
| *resetWorkspace* | Clear all variables in the workspace except the default model configurations *csMultiInstance* and *csSingleInstance*. Use this in lieu of *clear*.  Syntax:  *resetWorkspace()* |
| *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')* |

|  |  |
| --- | --- |
| *genSDD* | Generate a System Design Description (SDD) report from the model.  Syntax:   * *genSDD('MODEL')* * *genSDD('MODEL', 'AUTHOR')*   Use the second argument to optionally specify an author’s name.  This utility creates the following:  A document named after the model with a postfix of *\_SDD* under *DO\_03\_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', 'TreatAsMdlRef')*   Use the *TreatAsMdlRef* option to handle the model as a referenced model, otherwise it is regarded as a top-level model.  This utility creates the following:  Code under *MODEL\_ert\_rtw* or *slprj\ert\MODEL* in *DO\_04\_Code\specification*  This utility depends on the following:  *open\_MODEL* |
| *genLowLevelTests* | Generate low-level tests for the model based on existing coverage data using Simulink Design Verifier.  Syntax:   * *genLowLevelTests('MODEL')* * *genLowLevelTests('MODEL', 'AbsTol', 1e-6)* * *genLowLevelTests('MODEL', 'RelTol', 1e-6)* * *genLowLevelTests('MODEL', 'AbsTol', 1e-6, 'RelTol', 1e-3)*   Use the *AbsTol* and *RelTol* settings to control the baseline criteria tolerances of the generated test cases.  This utility creates the following:  A test file named after the model with a postfix of *\_SLDV\_Based\_Test* under *DO\_03\_Design\MODEL\verification \simulation\_test\low\_level\_tests*.  This utility depends on the following:   * *open\_MODEL* * *DO\_03\_Design\MODEL\verification\model\_coverages \high\_level\_tests\MODEL\_REQ\_Based\_Model\_Coverage.cvt* |

|  |  |
| --- | --- |
| *mergeCodeCoverage* | Merge coverage results from verifying the generated code of the model against the high-level software requirements and low-level tests generated by Simulink Design Verifier.  Syntax:  *mergeCodeCoverage('MODEL')*  This utility creates the following:  A cumulative coverage report named after the model with a postfix of *\_Merged\_Code\_Coverage\_Report* under *DO\_04\_Code\verification\code\_coverages\MODEL*.  This utility depends on the following:   * *DO\_04\_Code\verification\executable\_tests\MODEL\host \high\_level\_tests\MODEL\_SIL\_REQ\_Based\_Test\_Results.mldatx* * *DO\_04\_Code\verification\executable\_tests\MODEL\host \low\_level\_tests\MODEL\_SIL\_SLDV\_Based\_Test\_Results.mldatx* |
| *mergeModelCoverage* | Merge coverage results from verifying the model against the high-level software requirements and low-level tests generated by Simulink Design Verifier.  Syntax:  *mergeModelCoverage('MODEL')*  This utility creates the following:  A cumulative coverage report named after the model with a postfix of *\_Merged\_Model\_Coverage\_Report* under *DO\_03\_Design\MODEL\verification\model\_coverages*.  This utility depends on the following:   * *DO\_03\_Design\MODEL\verification\simulation\_tests \high\_level\_tests\MODEL\_REQ\_Based\_Test\_Results.mldatx* * *DO\_03\_Design\MODEL\verification\simulation\_tests \low\_level\_tests\MODEL\_SLDV\_Based\_Test\_Results.mldatx* |
| *verifyModel2LowLevelTests* | Verify the model against low-level 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', 'AUTHOR')*   Use the second argument to optionally specify an author’s name.  This utility creates the following:   * A test report named after the model with a postfix of *\_SLDV\_Based\_Test\_Report* under *DO\_03\_Design\MODEL \verification\simulation\_tests\low\_level\_tests* * A coverage report named after the model with a postfix of *\_SLDV\_Based\_Model\_Coverage\_Report* under *DO\_03\_Design \MODEL\verification\model\_coverages\low\_level\_tests*   This utility depends on the following:  *DO\_03\_Design\MODEL\verification\simulation\_tests \low\_level\_tests\MODEL\_SLDV\_Based\_Test.mldatx* |
| *verifyModel2Reqs* | Verify if the model complies with the high-level 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', 'AUTHOR')*   Use the second argument to optionally specify an author’s name.  This utility creates the following:   * A test report named after the model with a postfix of *\_REQ\_Based\_Test\_Report* under *DO\_03\_Design\MODEL \verification\simulation\_tests\high\_level\_tests* * A coverage report named after the model with a postfix of *\_REQ\_Based\_Model\_Coverage\_Report* under *DO\_03\_Design \MODEL\verification\model\_coverages\high\_level\_tests*   This utility depends on the following:  *DO\_03\_Design\MODEL\verification\simulation\_tests \high\_level\_tests\MODEL\_REQ\_Based\_Test.mldatx* |
| *verifyObjCode2LowLevelTests* | Verify the executable object code of the model against low-level tests generated by Simulink Design Verifier. All tests exercise the compiled code on the target computer via PIL simulations.  Syntax:   * *verifyObjCode2LowLevelTests('MODEL')* * *verifyObjCode2LowLevelTests('MODEL', 'AUTHOR')*   Use the second argument to optionally specify an author’s name.  This utility creates the following:  A test report named after the model with a postfix of *\_PIL\_SLDV\_Based\_Test\_Report* under *DO\_04\_Code\verification \executable\_tests\MODEL\target\low\_level\_tests*  This utility depends on the following:  *DO\_03\_Design\MODEL\verification\simulation\_tests \low\_level\_tests\MODEL\_SLDV\_Based\_Test.mldatx* |
| *verifyObjCode2Reqs* | Verify if the executable object code of the model complies with the high-level software requirements. All tests exercise the compiled code on the target computer via PIL simulations.  Syntax:   * *verifyObjCode2Reqs('MODEL')* * *verifyObjCode2Reqs('MODEL', 'AUTHOR')*   Use the second argument to optionally specify an author’s name.  This utility creates the following:  A test report named after the model with a postfix of *\_PIL\_REQ\_Based\_Test\_Report* under *DO\_04\_Code\verification \executable\_tests\MODEL\target\high\_level\_tests*  This utility depends on the following:  *DO\_03\_Design\MODEL\verification\simulation\_tests \high\_level\_tests\MODEL\_REQ\_Based\_Test.mldatx* |
| *verifySrcCode2LowLevelTests* | Verify the generated code of the model against low-level tests generated by Simulink Design Verifier, and then perform code coverage analysis. All tests exercise the compiled code on the target computer via SIL simulations.  Syntax:   * *verifySrcCode2LowLevelTests('MODEL')* * *verifySrcCode2LowLevelTests('MODEL', 'AUTHOR')*   Use the second argument to optionally specify an author’s name.  This utility creates the following:   * A test report named after the model with a postfix of *\_SIL\_SLDV\_Based\_Test\_Report* under *DO\_04\_Code\verification \executable\_tests\MODEL\host\low\_level\_tests* * A coverage report named after the model with a postfix of *\_SLDV\_Based\_Code\_Coverage\_Report* under *DO\_04\_Code \verification\code\_coverages\MODEL\low\_level\_tests*   This utility depends on the following:  *DO\_03\_Design\MODEL\verification\simulation\_tests \low\_level\_tests\MODEL\_SLDV\_Based\_Test.mldatx* |
| *verifySrcCode2Reqs* | Verify if the generated code of the model complies with the high-level software requirements, and then perform code coverage analysis. All tests exercise the compiled code on the target computer via SIL simulations.  Syntax:   * *verifySrcCode2Reqs('MODEL')* * *verifySrcCode2Reqs('MODEL', 'AUTHOR')*   Use the second argument to optionally specify an author’s name.  This utility creates the following:   * A test report named after the model with a postfix of *\_SIL\_REQ\_Based\_Test\_Report* under *DO\_04\_Code\verification \executable\_tests\MODEL\host\high\_level\_tests* * A coverage report named after the model with a postfix of *\_REQ\_Based\_Code\_Coverage\_Report* under *DO\_04\_Code \verification\code\_coverages\MODEL\high\_level\_tests*   This utility depends on the following:  *DO\_03\_Design\MODEL\verification\simulation\_tests \high\_level\_tests\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', 'TreatAsMdlRef')*   Use the *TreatAsMdlRef* option to handle the model as a referenced model, otherwise it is regarded as a top-level model.  This utility creates the following:  A report named after the model with a postfix of *\_SCS\_Conformance\_Report* under *DO\_04\_Code\verification \code\_standard\_checks\MODEL*  This utility depends on the following:   * *open\_MODEL* * Code under *MODEL\_ert\_rtw* or *slprj\ert\MODEL* in *DO\_04\_Code\specification* |
| *checkModelStds* | Check the model against software model standards, and then generate the Model Advisor report.  Syntax:   * *checkModelStds('MODEL')* * *checkModelStds('MODEL', 'TreatAsMdlRef')*   Use the *TreatAsMdlRef* option to handle the model as a referenced model, otherwise it is regarded as a top-level model.  This utility creates the following:  A report named after the model with a postfix of *\_SMS\_Conformance\_Report* under *DO\_03\_Design\MODEL \verification\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', 'TreatAsMdlRef')*   Use the *TreatAsMdlRef* option to handle the model as a referenced model, otherwise it is regarded as a top-level model.  This utility creates the following:  A report named after the model with a postfix of *\_Code\_Metrics\_Report* under *DO\_04\_Code\verification \code\_metrics\MODEL*  This utility depends on the following:   * *open\_MODEL* * Code under *MODEL\_ert\_rtw* or *slprj\ert\MODEL* in *DO\_04\_Code\specification* |

|  |  |
| --- | --- |
| *computeModelMetrics* | Compute metrics for the model, and then generate the Model Advisor report.  Syntax:   * *computeModelMetrics('MODEL')* * *computeModelMetrics('MODEL', 'TreatAsMdlRef')*   Use the *TreatAsMdlRef* option to handle the model as a referenced model, otherwise it is regarded as a top-level model.  This utility creates the following:  A report named after the model with a postfix of *\_Model\_Metrics\_Report* under *DO\_03\_Design\MODEL \verification\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', 'TreatAsMdlRef')*   Use the *TreatAsMdlRef* option to handle the model as a referenced model, otherwise it is regarded as a top-level model.  This utility creates the following:  A report named after the model with a postfix of *\_Bug\_Finder\_Report* under *DO\_04\_Code\verification \coding\_error\_detections\MODEL*  This utility depends on the following:   * *open\_MODEL* * Code under *MODEL\_ert\_rtw* or *slprj\ert\MODEL* in *DO\_04\_Code\specification* |
| *detectDesignErrs* | Detect design errors or dead logic in the model, and then generate the Design Verifier report.  Syntax:   * *detectDesignErrs('MODEL')* * *detectDesignErrs('MODEL', 'DetectDeadLogic')*   Use the *DetectDeadLogic* option to find dead logic, otherwise it looks for design errors.  This utility creates one of the following:   * A report named after the model with a postfix of *\_Dead\_Logic\_Detection\_Report* under *DO\_03\_Design\MODEL \verification\design\_error\_detections\dead\_logic* * A report named after the model with a postfix of *\_Design\_Error\_Detection\_Report* under *DO\_03\_Design\MODEL \verification\design\_error\_detections\design\_error*   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', 'TreatAsMdlRef')*   Use the *TreatAsMdlRef* option to handle the model as a referenced model, otherwise it is regarded as a top-level model.  This utility creates the following:  A report named after the model with a postfix of *\_Code\_Prover\_Report* under *DO\_04\_Code\verification \code\_proving\MODEL*  This utility depends on the following:   * *open\_MODEL* * Code under *MODEL\_ert\_rtw* or *slprj\ert\MODEL* in *DO\_04\_Code\specification* |
| *verifySrcCode2Model* | Verify if the generated code complies with and is traceable to the model, and then generate the Simulink Code Inspector report.  Syntax:   * *verifySrcCode2Model('MODEL')* * *verifySrcCode2Mode('MODEL', 'TreatAsMdlRef')*   Use the *TreatAsMdlRef* option to handle the model as a referenced model, otherwise it is regarded as a top-level model.  This utility creates the following:  A report named after the model with a postfix of *\_summaryReport* under *DO\_04\_Code\verification \code\_reviews\MODEL*  This utility depends on the following:   * *open\_MODEL* * Code under *MODEL\_ert\_rtw* or *slprj\ert\MODEL* in *DO\_04\_Code\specification* |