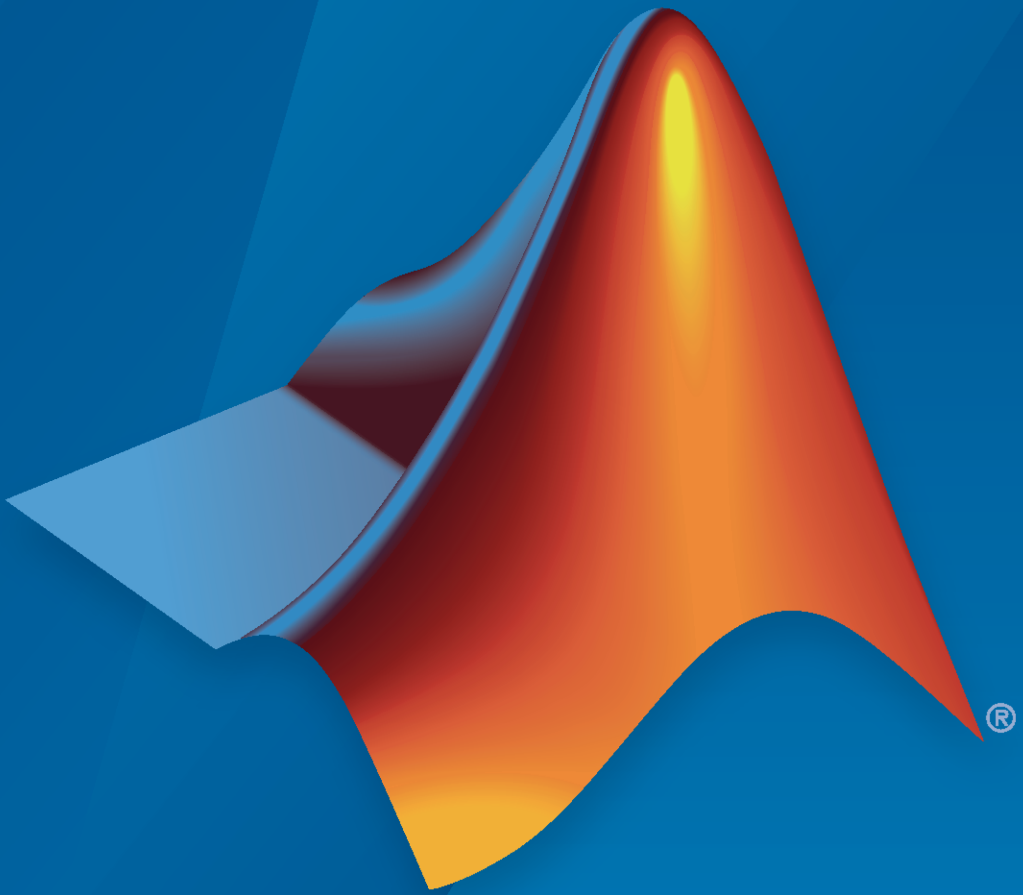
Requirements Toolbox™

Getting Started Guide



MATLAB® & SIMULINK®

R2023a

MathWorks®

How to Contact MathWorks

Requirements Toolbox™ Getting Started Guide

Revision History

September 2017	Online Only	New for Version 1.0 (Release 2017b)
March 2018	Online Only	Revised for Version 1.1 (Release 2018a)
September 2018	Online Only	Revised for Version 1.2 (Release 2018b)
March 2019	Online Only	Revised for Version 1.3 (Release 2019a)
September 2019	Online Only	Revised for Version 1.4 (Release 2019b)
March 2020	Online only	Revised for Version 1.5 (Release 2020a)
September 2020	Online only	Revised for Version 1.6 (Release 2020b)
March 2021	Online only	Revised for Version 1.7 (Release 2021a)
September 2021	Online only	Revised for Version 1.8 (Release 2021b)
March 2022	Online only	Revised for Version 2.0 (Release 2022a)
September 2022	Online only	Revised for Version 2.1 (Release 2022b)
March 2023	Online only	Revised for Version 2.2 (Release 2023a)

Contents

1 Getting Started with Requirements Toolbox

Getting Started with Requirements Toolbox

Requirements Toolbox Product Description

Author, link, and validate requirements for designs and tests

Requirements Toolbox™ lets you author, link, and validate requirements within MATLAB® or Simulink®. You can create requirements using rich text with custom attributes or import them from requirements management tools.

You can link requirements to MATLAB code, System Composer™ or Simulink models, and tests. The toolbox analyzes the traceability to identify gaps in implementation or testing. The design highlighting and traceability matrix summarize where links exist across multiple artifacts and guide you to address any gaps. When requirements change, linked artifacts are highlighted, and you can determine the upstream and downstream artifacts affected using a traceability diagram. Generated code from Simulink designs includes code comments that document where requirements are implemented to assist with reviews.

You can formalize requirements and analyze them for consistency, completeness, and correctness using the Requirements Table. The Requirements Perspective enables you to view and manage requirements together with design. When used with Simulink, you can create links to blocks with a simple drag and drop.

Support for industry standards is available through IEC Certification Kit (for ISO 26262 and IEC 61508) and DO Qualification Kit (for DO-178).

Work with Requirements in the Requirements Editor

Requirements Toolbox enables you to author, organize, and edit requirements in the **Requirements Editor**. When working in a Simulink model, you can use the Requirements Perspective to visualize the links between requirements and the parts of a model. Using an integrated environment simplifies linking requirements to the parts of your model that implement them.

This integrated environment has other advantages. For more information, see “Introduction to Requirements Toolbox” on page 1-13.

Author Requirements in MATLAB or Simulink

In Requirements Toolbox, you organize your requirements in groups called requirement sets. In each requirement set, you can create additional levels of hierarchy if you need to further describe a requirement's details.

In this tutorial, you use the **Requirements Editor** to create a requirement set, organize related requirements, and add requirements to the set. If you have Simulink, you can also use the Requirements Perspective to author requirements without leaving the Simulink Editor. For more information about using the Requirements Perspective, see “Link Blocks and Requirements”.

Suppose that you are writing requirements for a controller model of an automobile cruise control system. You develop these requirements using your company’s numbering standard (R1, R2, and so on).

ID and Description	Rationale
R1: The maximum input throttle is 100%	The maximum value of the throttle from the acceleration pedal can be no greater than 100%.
R2: Cruise control has a speed operation range	Cruise control has a minimum and maximum operating speed.
R2.1: The vehicle speed must be at least 40 km/h	The speed of the vehicle must be at least 40 km/h for the cruise control system to engage.
R2.2: The vehicle speed cannot be greater than 100 km/h	The maximum operational speed of the cruise control system for the vehicle is 100 km/h.

These requirements capture functionality modeled in a model called `crs_controller`.

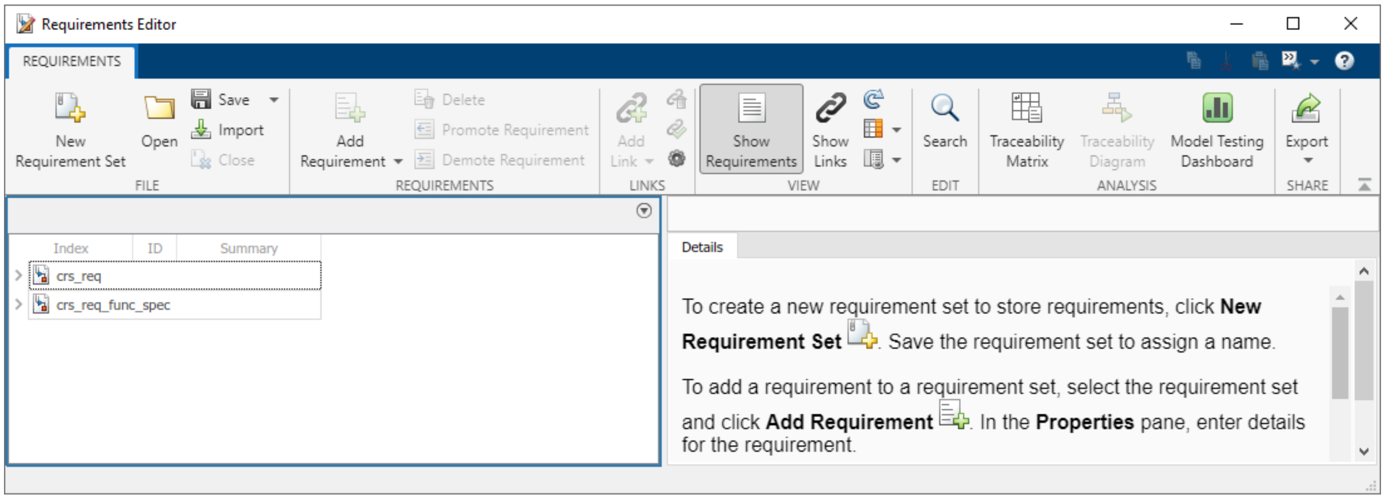
1 Open the project that includes the model and supporting files. At the MATLAB command prompt, enter:

```
slreqCCProjectStart
```

2 Open the requirement set `crs_req` in the **Requirements Editor**. At the command prompt, enter:

```
slreq.open("crs_req")
```

3 The **Requirements Editor** displays the requirements arranged by requirement set. The project has two requirement sets: `crs_req_func_spec` and `crs_req`.



To create a new requirement set to store requirements, click **New Requirement Set**. Save the requirement set to assign a name.

To add a requirement to a requirement set, select the requirement set and click **Add Requirement**. In the **Properties** pane, enter details for the requirement.

4 Add a requirement set. From the **Requirements Editor** toolstrip, click **New Requirement Set**.

5 Save the requirement sets to external files. Save your requirement set to a writable location and name it `cruise_control_reqset.slreqx`.

6 Add a requirement to your requirement set by selecting the requirement set and clicking **Add Requirement**.

7 In the right pane, under **Properties**, enter the details for the requirement. Enter the details for the requirement:

- **Custom ID**: R1
- **Summary**: Max input throttle %
- **Description**: The maximum input throttle is 100%.

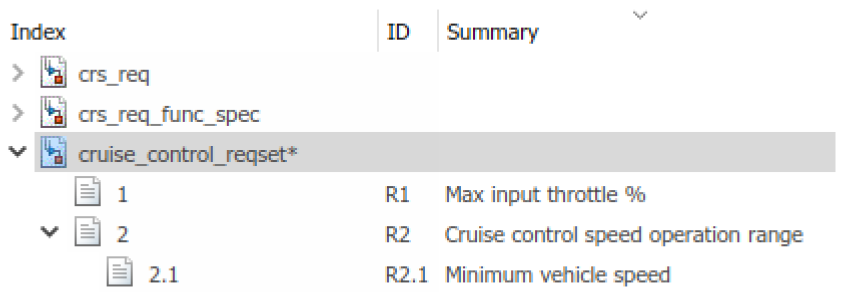
If you do not specify a custom ID, the **Requirements Editor** numbers requirements in order. Custom IDs enable you to use your company standards for labeling requirements and to set the numeric order. (Custom IDs cannot contain a # character.) You can also use an ID to help locate a requirement when searching. Keywords aid in searching for a requirement.

8 Create the requirement R2. Click **Add Requirement**. Enter the details for the requirement:

- **Custom ID**: R2
- **Summary**: Cruise control speed operation range
- **Description**: Cruise control has a minimum and maximum operating speed.

9 Create child requirements for R2 by selecting R2 and clicking **Add Requirement** > **Add Child Requirement**. Enter the details for the requirement:

- **Custom ID**: R2.1
- **Summary**: Minimum vehicle speed
- **Description**: The speed of the vehicle must be at least 40 km/h for the cruise control system to engage.



Index	ID	Summary
crs_req		
crs_req_func_spec		
cruise_control_reqset*		
1	R1	Max input throttle %
2	R2	Cruise control speed operation range
2.1	R2.1	Minimum vehicle speed

Repeat this step to add other child requirements to R2.

You can rearrange the hierarchy by using **Promote Requirement** or **Demote Requirement**.

Author and Edit Requirements Content by Using Microsoft Word

To author and edit the **Description** and **Rationale** fields of your requirements, open Microsoft® Word from within the **Requirements Editor** or the Requirements Perspective View.

Note This functionality is available only on Microsoft Windows® platforms.

Using Microsoft Word to edit rich text requirements enables you to:

- Spell-check requirements content.
- Resize images.
- Insert and edit equations.
- Insert and edit tables.

On the Edit field toolbar, in either the **Description** or **Rationale** fields, click the icon. Save the changes to your requirements content within Microsoft Word to see them reflected in Requirements Toolbox.

When you use Microsoft Word to edit requirements content, you cannot edit requirements in the built-in editor.

Customize Requirements Browser View

You can view or hide columns in the **Requirements Editor** when you click **Columns > Select Attributes**. Add, remove, and reorder attribute columns in the Column Selector. The view configuration is saved across sessions. You can export view settings to a MAT-file by using the `slreq.exportViewSettings` function and import them by using the `slreq.importViewSettings` function. You can reset view configurations by using the `slreq.resetViewSettings` function.

Filter Requirements Content

You can search requirements content by clicking **Search**. You can find specific requirements within loaded requirement sets based on requirement attributes and descriptions.

Specify Filter Text Strings — As you enter text in the **Search** text box, the Requirements Browser performs a dynamic search and displays the results. The search operation applies only to attributes you choose to display in the Requirements Browser.

The text strings you enter must be consistent with the guidelines described in the following sections.

Case Sensitivity — By default, the Requirements Browser ignores case as it filters.

If you want the Requirements Browser to respect case sensitivity, put that text string in quotation marks.

Specify Attributes and Attribute Values — To restrict the filtering to requirements with a specific attribute, type the attribute name, followed by a colon. The Requirements Browser displays only the requirements that have that attribute.

To filter for requirements for which a specific attribute has a specific value, type the attribute name, followed by a colon (:), then the value. For example, to filter the contents to display only the requirements where the `Summary` attribute has a value that includes `Aircraft`, enter `Summary: Aircraft` (alternatively, you could put the whole string in quotation marks to enforce case sensitivity).

Wildcards and MATLAB Expressions Are Not Supported — The Requirements Browser does not recognize wildcard characters, such as *. For example, searching `fuel*` returns no results, even if requirements contain the text string `fuel`.

Also, if you specify a MATLAB expression in the **Search** text box, the Requirements Browser interprets that string as literal text, not as a MATLAB expression.

See Also

Requirements Editor

More About

- “Link Blocks and Requirements”
- “Introduction to Requirements Toolbox” on page 1-13
- “Create and Store Links”
- “Define Requirements Hierarchy”

Link Test Cases to Requirements

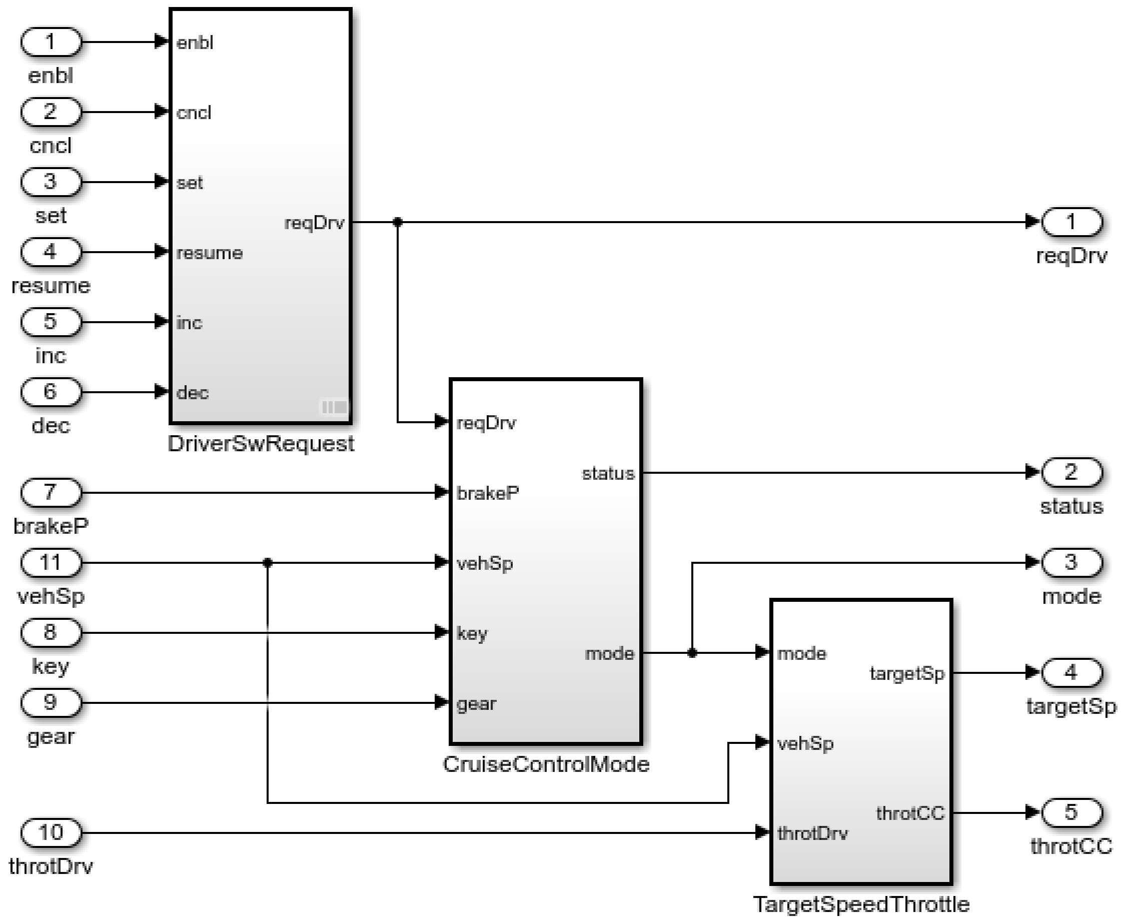
If you have Simulink Test™ and Requirements Toolbox, you can link your requirements to test cases in the Test Manager. Linking requirements to tests allows you to verify that the implementation of the requirement behaves as expected. You can use the requirement verification status to track verification progress. For more information, see “Review Requirements Verification Status”.

Link a Test Case to a Requirement Example

This example shows how to link a test case to a requirement associated with a controller model of an automobile cruise control system. After you run the tests, you include the results in the Requirements Editor.

To link a requirement to a test case, open the project, `CruiseRequirementsExample`. Then, open the `crs_controller` model.

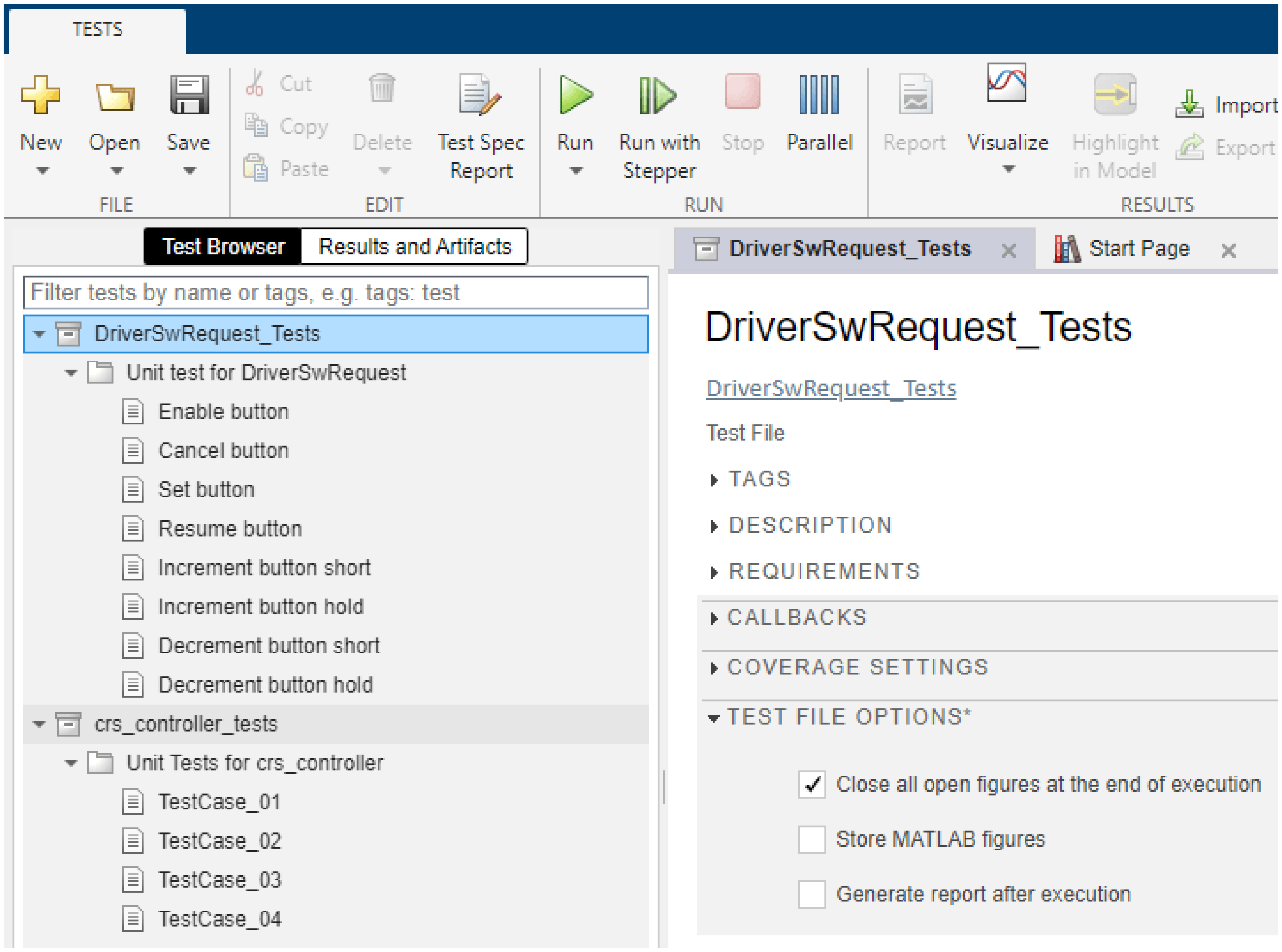
```
openProject("CruiseRequirementsExample");
open_system("models/crs_controller");
```



Load the test files that contain the tests you want to link.

1. From the model, open the Test Manager. In the **Apps** tab, click **Simulink Test**. In the **Tests** tab, click **Simulink Test Manager**.
2. In the Test Manager, open the `DriverSwRequest_Tests.mldatx` and `crs_controller_tests.mldatx` test files. In the **File** section, click **Open**. Load the test files in the `CruiseRequirementsExample\tests` folder.

The test files contain the test cases for several of the requirements in the `crs_controller` model. Most of these test cases already link to requirements.



Filter tests by name or tags, e.g. tags: test

- DriverSwRequest_Tests
  - Unit test for DriverSwRequest
    - Enable button
    - Cancel button
    - Set button
    - Resume button
    - Increment button short
    - Increment button hold
    - Decrement button short
    - Decrement button hold
- crs_controller_tests
  - Unit Tests for crs_controller
    - TestCase_01
    - TestCase_02
    - TestCase_03
    - TestCase_04

DriverSwRequest_Tests

DriverSwRequest_Tests

Test File

TAGS

DESCRIPTION

REQUIREMENTS

CALLBACKS

COVERAGE SETTINGS

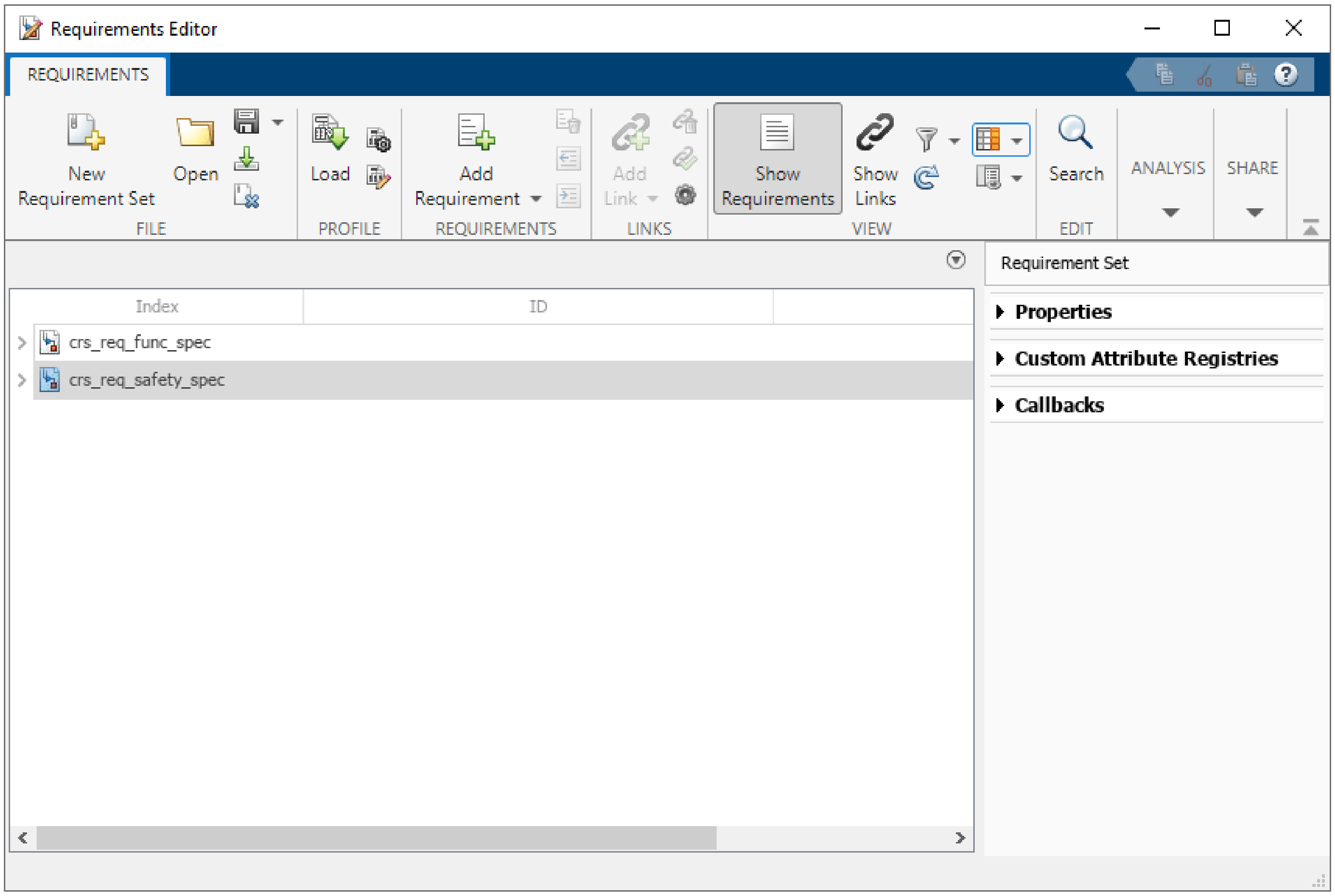
TEST FILE OPTIONS*

[✓] Close all open figures at the end of execution

[ ] Store MATLAB figures

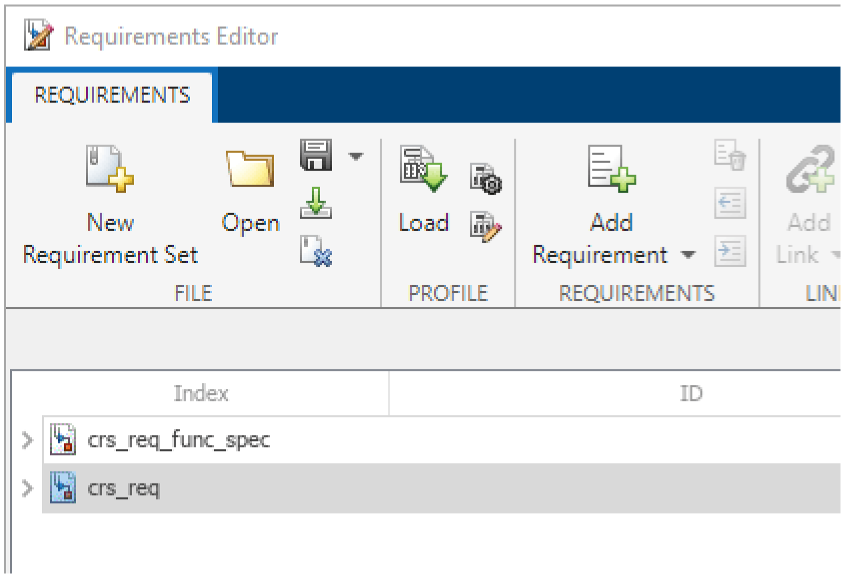
[ ] Generate report after execution

In this example, you link the `Increment button hold` test to a requirement. In the left pane, click **DriverSWRequest_Tests > Unit test for DriverSwRequest > Increment button hold**. In the model, open the Requirements Editor. In the **Apps** tab, in the **Apps** section, click **Requirements Editor**. Click **Show Requirements**. The Requirements Editor displays two requirement sets, `crs_req_func_spec`, and `crs_req_safety_spec`.



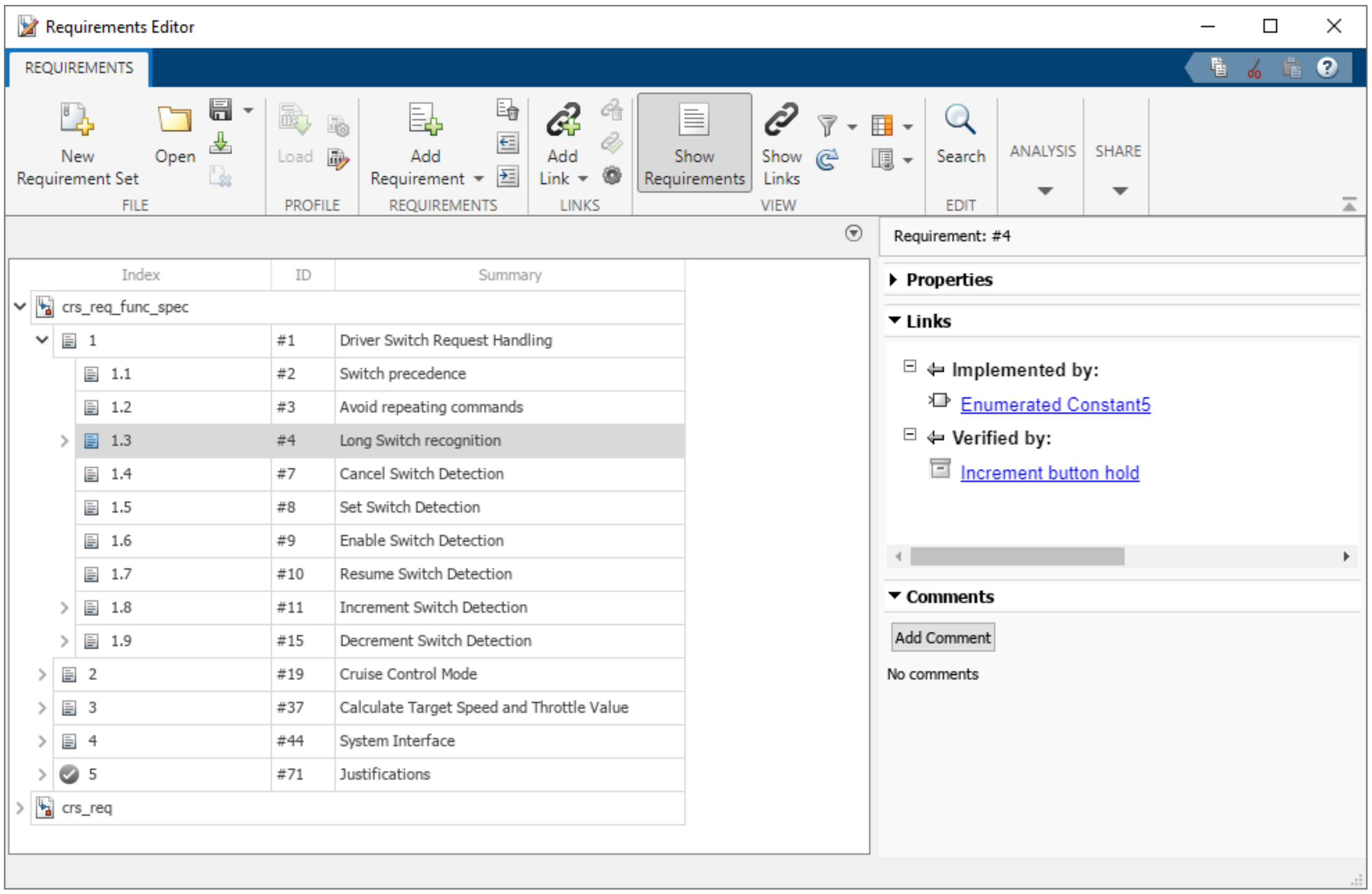
crs_req_func_spec
crs_req_safety_spec

In this example, you do not test the requirements in `crs_req_safety_spec`, and you must load another requirement set to load the tested requirements. To close the `crs_req_safety_spec` requirement set, select `crs_req_safety_spec` and, in the **File** section, click **Close**. In the **File** section, click **Open**. In `CruiseRequirementsExample\documents`, open the file `crs_req.slreqx`. The Requirements Editor updates the loaded requirements.



Index	ID
crs_req_func_spec	
crs_req	

Link the requirement to the test case. Expand the `crs_req_func_spec` requirement set and expand the requirement with the index 1. Select the requirement with the index `1.3`. In the **Links** section, click **Add Link > Link from Selected Test Case**. The link to the test case appears in the right pane, under **Links**.



Index	ID	Summary
crs_req_func_spec		
1	#1	Driver Switch Request Handling
1.1	#2	Switch precedence
1.2	#3	Avoid repeating commands
1.3	#4	Long Switch recognition
1.4	#7	Cancel Switch Detection
1.5	#8	Set Switch Detection
1.6	#9	Enable Switch Detection
1.7	#10	Resume Switch Detection
1.8	#11	Increment Switch Detection
1.9	#15	Decrement Switch Detection
2	#19	Cruise Control Mode
3	#37	Calculate Target Speed and Throttle Value
4	#44	System Interface
5	#71	Justifications
crs_req		

Requirement: #4

Properties

Links

Implemented by:
Enumerated Constant5

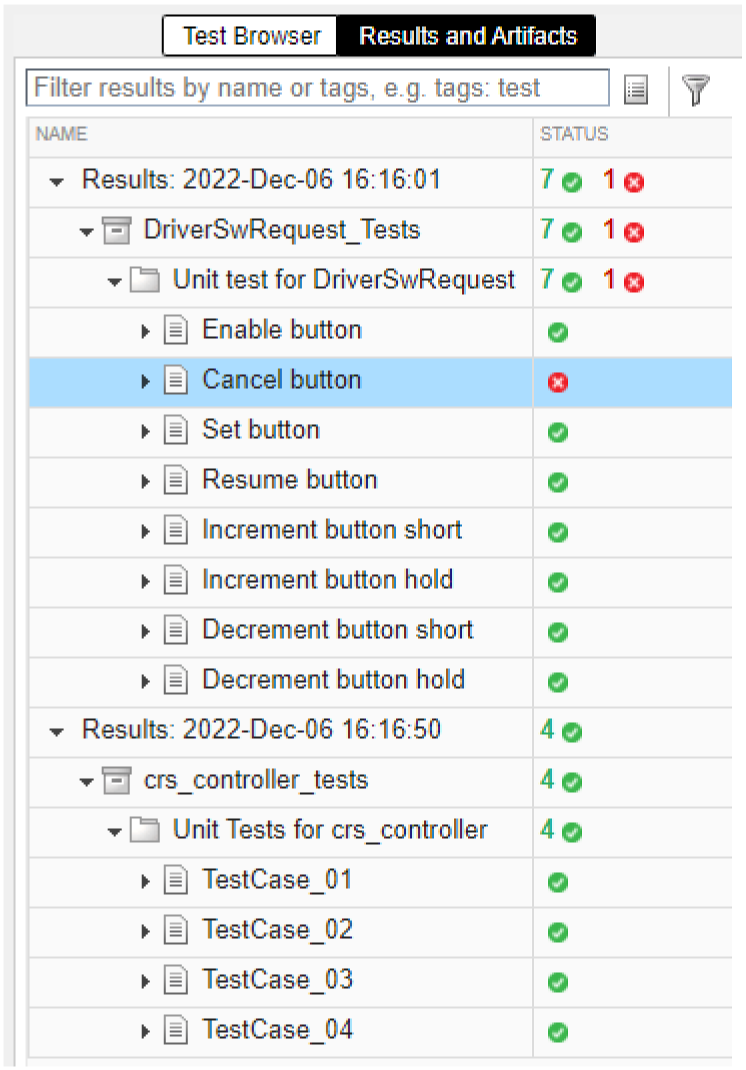
Verified by:
Increment button hold

Comments

No comments

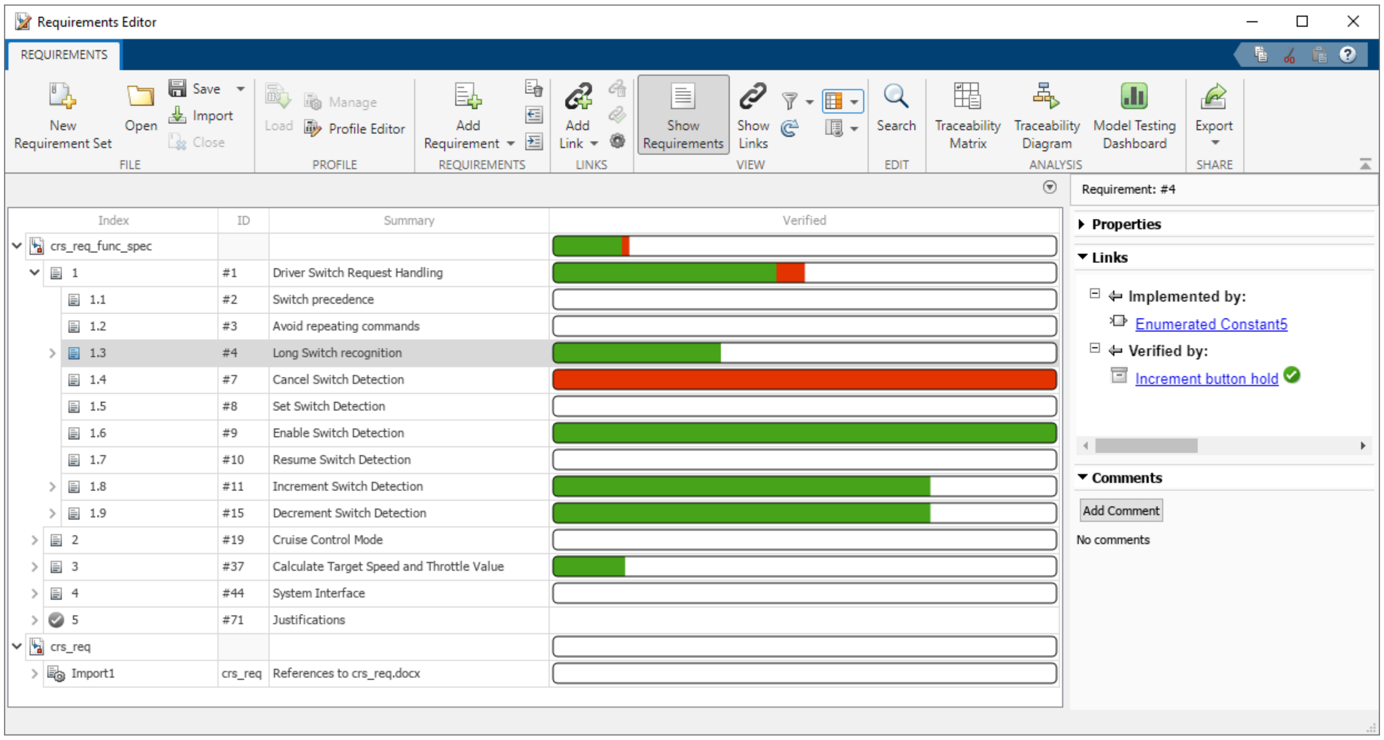
You can view verification information for other requirements by selecting each requirement.

Run the linked tests. In the Test Manager, select the top node in the test hierarchy in the **Test Browser** pane that corresponds to each test file and click **Run**. The **Results and Artifacts** pane shows that 7 tests passed and 1 test failed in the `DriverSwRequest_Tests` test file, and that 4 of the tests passed in the `crs_controller_tests` test file. Expand the results of each test run, test file, and test suite. In the test results for `DriverSwRequest_Tests`, the `Cancel button` test failed.



NAME	STATUS
Results: 2022-Dec-06 16:16:01	7 ✓ 1 ✗
DriverSwRequest_Tests	7 ✓ 1 ✗
Unit test for DriverSwRequest	7 ✓ 1 ✗
Enable button	✓
Cancel button	✗
Set button	✓
Resume button	✓
Increment button short	✓
Increment button hold	✓
Decrement button short	✓
Decrement button hold	✓
Results: 2022-Dec-06 16:16:50	4 ✓
crs_controller_tests	4 ✓
Unit Tests for crs_controller	4 ✓
TestCase_01	✓
TestCase_02	✓
TestCase_03	✓
TestCase_04	✓

To view the verification status associated with the requirements linked to the tests in the Requirements Editor, in the **View** section, click **Columns > Verification Status**. Some requirements do not have tests, and some requirements are fully verified. The bar in the **Verified** column shows the proportion of child requirements that have links to verification. The color of the display indicates the proportion of tests that have passed, failed, or not run.



Index	ID	Summary	Verified
crs_req_func_spec			
1	#1	Driver Switch Request Handling	
1.1	#2	Switch precedence	
1.2	#3	Avoid repeating commands	
1.3	#4	Long Switch recognition	
1.4	#7	Cancel Switch Detection	
1.5	#8	Set Switch Detection	
1.6	#9	Enable Switch Detection	
1.7	#10	Resume Switch Detection	
1.8	#11	Increment Switch Detection	
1.9	#15	Decrement Switch Detection	
2	#19	Cruise Control Mode	
3	#37	Calculate Target Speed and Throttle Value	
4	#44	System Interface	
5	#71	Justifications	
crs_req			
Import1	crs_req	References to crs_req.docx	

Requirement: #4

Properties

Links

Implemented by:

Enumerated Constant5

Verified by:

Increment button hold ✓

Comments

Add Comment

No comments

In this example, the verification status shows that the test that you linked to the requirement `1.3` passed, and the test linked to `Cancel Switch Detection` failed. Some requirements are partially verified because the child requirements are not yet verified, such as requirement `1.8`. To view a summary of details about the child requirements, point to the verification status bar of the parent requirement. Other requirements are unverified because they are not linked to a test case.

See Also

More About

- "Create and Store Links"
- "Link Test Cases to Requirements Documents"
- "Work with Requirements in the Requirements Editor" on page 1-3

Introduction to Requirements Toolbox

In this section...

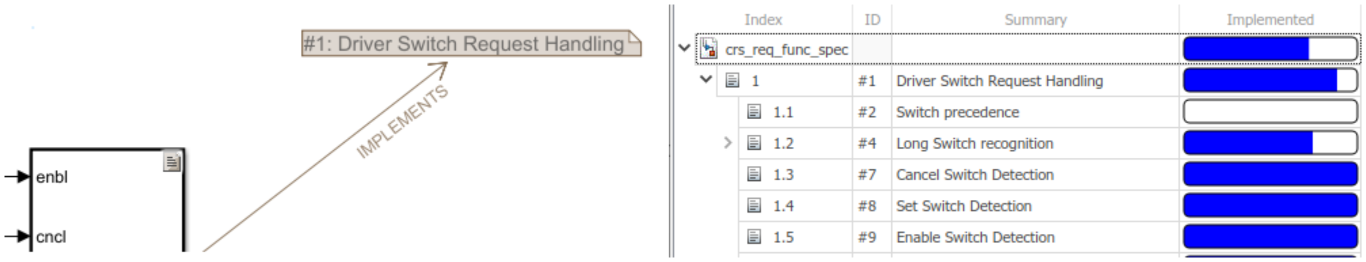
Requirements Toolbox integrates requirements authoring and management with your modeling environment. You can author requirements in Simulink in the **Requirements Editor**, where you can organize and manage them. You can also import them from Microsoft Word or Excel® on some platforms. For details, see "Import Requirements from Third-Party Applications".

Link Between Requirements and Implementation

You can link from requirements to the Simulink blocks or Stateflow® objects that implement them. The connection is bidirectional, meaning that you can locate a requirement from a model element and a model element from a requirement.

You can:

- View implementation progress, including identifying missing implementations.
- React to requirement changes by updating model elements as requirements change, and clarifying requirements as your model evolves. You can find changed requirements by using a single command.
- Confirm that model changes conform to the associated requirement.



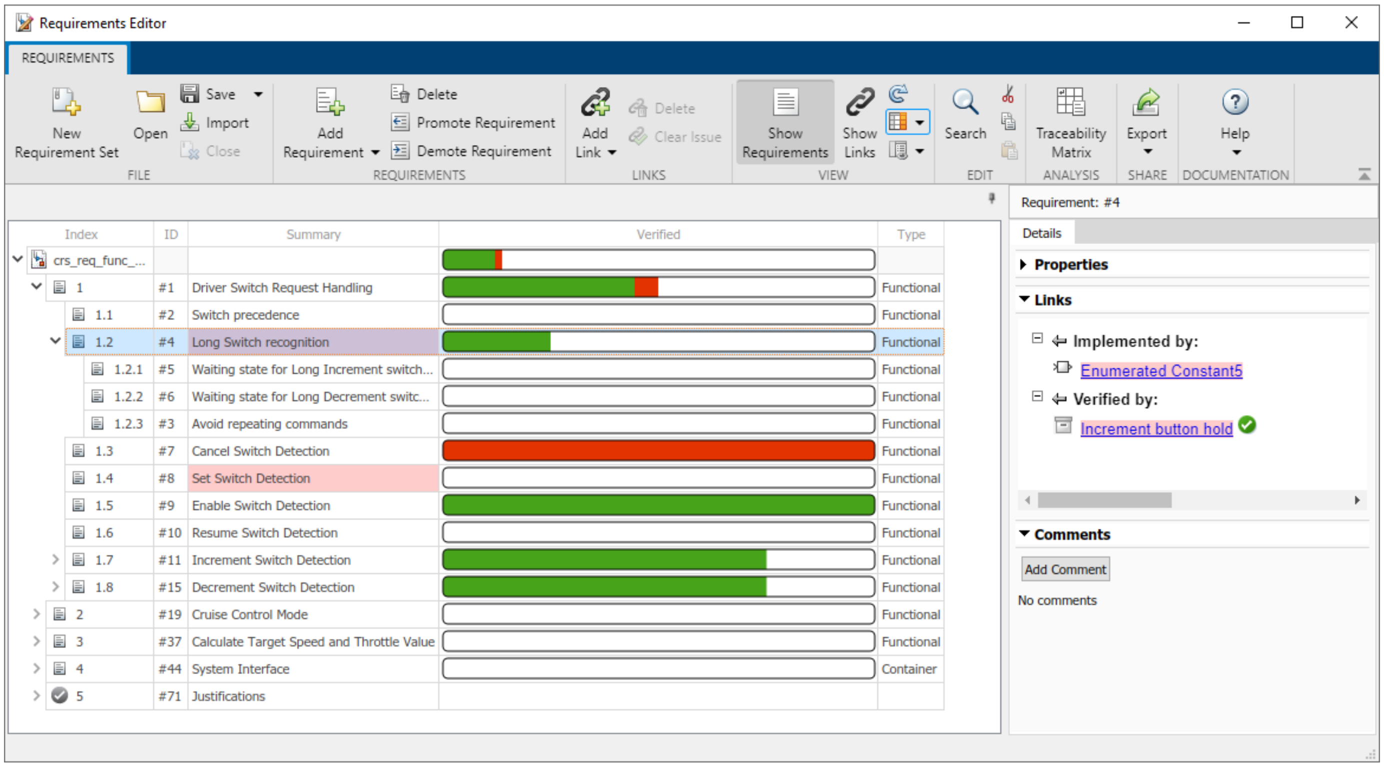
Index	ID	Summary	Implemented
crs_req_func_spec			
1	#1	Driver Switch Request Handling	
1.1	#2	Switch precedence	
1.2	#4	Long Switch recognition	
1.3	#7	Cancel Switch Detection	
1.4	#8	Set Switch Detection	
1.5	#9	Enable Switch Detection	

For more information, see "Create and Store Links".

Link Between Requirements and Simulink Test

If you have Simulink Test, you can link between requirements and tests that verify them. You can associate a requirement or set of requirements with tests that you create in Test Manager. When you run a test in Test Manager that you linked to a requirement, you can see the pass/fail results in the **Requirements Editor**.

Because you can track test results in Requirements Toolbox, you can see the progress toward verification. The verification status also helps you to identify missing information and clusters of requirements associated with failing tests. You can use this information to understand the impact and complexity of those requirements.



Index	ID	Summary	Verified	Type
crs_req_func_...			🟩🟥	
1	#1	Driver Switch Request Handling	🟩🟥	Functional
1.1	#2	Switch precedence		Functional
1.2	#4	Long Switch recognition	🟩	Functional
1.2.1	#5	Waiting state for Long Increment switch...		Functional
1.2.2	#6	Waiting state for Long Decrement switc...		Functional
1.2.3	#3	Avoid repeating commands		Functional
1.3	#7	Cancel Switch Detection	🟥	Functional
1.4	#8	Set Switch Detection		Functional
1.5	#9	Enable Switch Detection	🟩	Functional
1.6	#10	Resume Switch Detection		Functional
1.7	#11	Increment Switch Detection	🟩	Functional
1.8	#15	Decrement Switch Detection	🟩	Functional
2	#19	Cruise Control Mode		Functional
3	#37	Calculate Target Speed and Throttle Value		Functional
4	#44	System Interface		Container
5	#71	Justifications		

Requirement: #4

Details

Properties

Links

Implemented by:

Enumerated Constant5

Verified by:

Increment button hold ✓

Comments

Add Comment

No comments

For more information, see "Link Test Cases to Requirements Documents".

Additional Requirements Traceability Links

With Requirements Toolbox, you can create several other types of traceability links and establish many relationships within your model and to external documents. You can create these types of traceability links:

- Implements, in which a design element implements a requirement
- Verifies, in which a test case verifies a requirement
- Related to, in which you establish a trace relationship between a model element and a requirement
- Derives, in which a requirement is derived from another requirement
- Refines, in which one requirement refines another requirement

You can link between other types of documents, for example, HTML or DOORS® items, and requirements and to additional model elements such as dictionary objects.

For more information, see "Create and Store Links".

Share and Reuse Requirements

You save requirements files separately from your model files. You can then reference requirement files from multiple models, and each model can reference multiple requirement files. Saving requirements in separate files lets you modularize common requirements across models while also managing requirements that are model-specific. This approach minimizes potential for copy-and-paste errors and keeps the requirements in sync across the models that share them.

You can compare requirements files (`.slreqx` files) by using the MATLAB file comparison tool. This tool helps you to identify differences in similar requirement sets. For more information, see “Compare Requirement Sets”.

You can also include requirements files in Projects. When you open a project, load requirement sets into the **Requirements Editor** from the project explorer. For more information, see “Requirements-Based Development in Projects”.

Another way to share information about requirements is to generate a report that includes the requirements definition, links, implementation details, verification status, and so on. For more information, see “Generate Requirements Reports Using Simulink”.

See Also

More About

- “Work with Requirements in the Requirements Editor” on page 1-3
- “Import Requirements from Third-Party Applications”
- “Link Test Cases to Requirements” on page 1-7

Access Frequently Used Features and Commands from the Requirements Editor

You can access your most frequently used features and commands in the **Requirements Editor** by using the quick access toolbar. You can customize the toolbar by adding actions from the toolstrip, reorganizing the toolbar buttons, and showing the button labels. You can also add commands that run MATLAB language statements to the quick access toolbar.

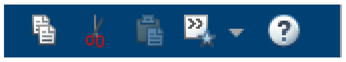
The quick access toolbar preferences persist across MATLAB sessions. The toolbar is always visible, even if the toolstrip is minimized.

Access the Quick Access Toolbar

Open the **Requirements Editor**. For more information, see Requirements Editor.

You can also add the **Requirements Editor** to the MATLAB or Simulink quick access toolbars from the **Apps** tab by right-clicking the **Requirements Editor** app and selecting **Add to Quick Access Toolbar**. For more information, see “Customize MATLAB Toolbars” and “Access Frequently Used Features and Commands in Simulink” (Simulink).

The quick access toolbar is in at the top right corner of the **Requirements Editor**.



The default buttons allow you to copy, cut, and paste requirements, referenced requirements, and justifications. You can create and run favorite commands by clicking the Favorites icon to open the Favorite Commands menu. You can also access the Requirements Toolbox documentation by clicking the help button .

Customize the Quick Access Toolbar

You can customize the quick access toolbar by adding and removing actions, rearranging the buttons, and showing the button labels. You can also restore the default toolbar buttons.

Add and Remove Actions

You can add actions to the quick access toolbar by right-clicking a toolstrip button and selecting **Add to Quick Access Toolbar**. You can also add favorite commands to the toolbar by using the Favorite Commands menu.

You can remove non-default actions, including favorite commands, by right-clicking the button in the toolbar and selecting **Remove from Quick Access Toolbar**.

Rearrange Buttons

You can rearrange the quick access toolbar buttons by clicking and dragging the buttons.

A partition separates the default and non-default toolbar buttons. You cannot move buttons across the partition.

Show and Hide Button Labels

Each quick access toolbar button has a label that describes the action it performs. You can show the label for a button by right-clicking the toolbar button and selecting **Show Label**.

You can hide the label by right-clicking the button and selecting **Hide Label**.

Restore the Default Toolbar

The default quick access toolbar contains actions to copy, cut, paste, open the Favorite Commands menu, and access the documentation.

You can restore the toolbar to its default state by right-clicking in the toolbar and selecting **Restore Defaults**.

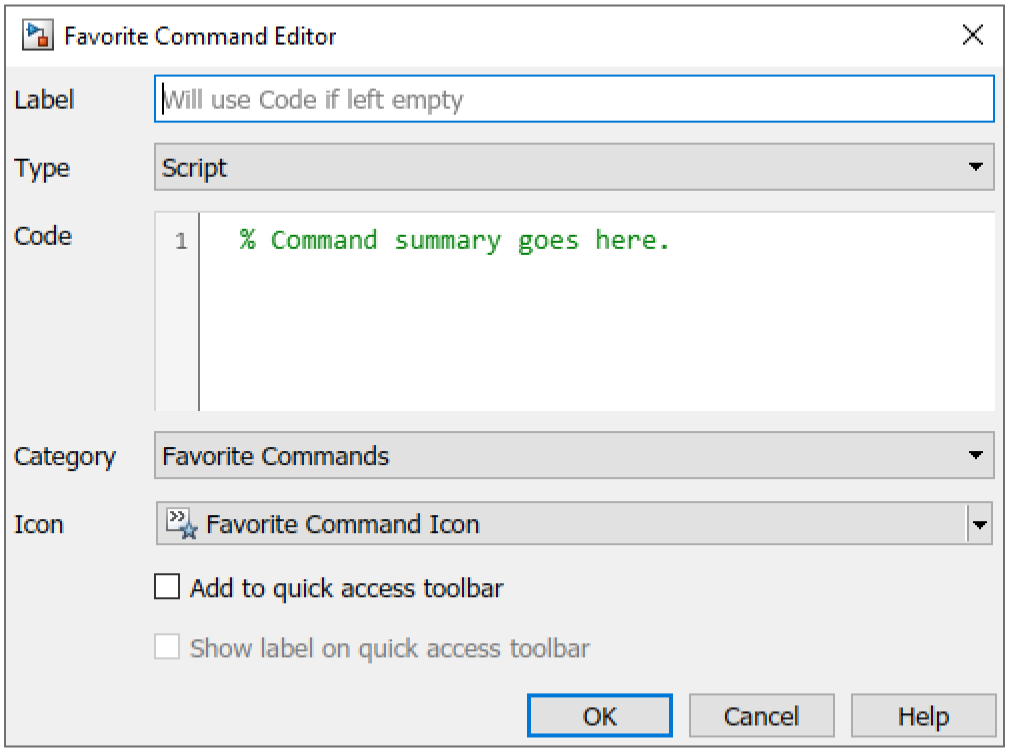
Restoring the default toolbar:

- Removes non-default action buttons, including favorite commands
- Arranges the default buttons to their default order
- Hides button labels

Create and Run Favorite Commands

You can create favorite commands to run a group of MATLAB language statements by using the Favorite Commands menu.

1. In the quick access toolbar, click the Favorites icon.
2. In the Favorite Commands menu, click **New Favorite**.
3. In the Favorite Command Editor, enter a name for the command in the **Label** field.


Favorite Command Editor

Label: Will use Code if left empty

Type: Script

Code:
```
% Command summary goes here.
```

Category: Favorite Commands

Icon: Favorite Command Icon

☐ Add to quick access toolbar

☐ Show label on quick access toolbar

4 Select the command type from the **Type** list. Scripts execute in the base workspace, while functions execute in a limited scope. For more information, see “Base and Function Workspaces”.
5 Enter your MATLAB code in the **Code** field.
6 Select the category to place the command in from the **Category** list. The `Favorite Commands` category is selected by default.
7 Choose an icon for the command by selecting from the **Icon** list. You can use a custom icon by setting **Icon** to `Specify custom icon`.
8 To add the command directly to the quick access toolbar, select **Add to quick access toolbar**. To show the label for the toolbar button, select **Show label on quick access toolbar**.
9 Click **OK**.

You can run the command by clicking the Favorites icon and clicking the command or by clicking the icon in the quick access toolbar.

Add a Favorite Command Category

You can create categories to organize your favorite commands into groups. To create a category, in the Favorite Command Editor, click **New Category**. Enter a name for the category in the **Label** field, then click **OK**.

Edit, Delete, and Organize Favorite Commands

You can edit, delete, and organize existing favorite commands and categories.

To edit a favorite command or category, right-click the command or category and select **Edit Favorite** or **Edit Category**. Make changes in the editor, then click **OK**.

To delete a favorite command or category, right-click the command or category and select **Delete Favorite** or **Delete Category**. Deleting a category deletes all favorite commands in the category.

See Also

More About

- “Author Requirements in MATLAB or Simulink”
- “Customize MATLAB Toolbars”
- “Access Frequently Used Features and Commands in Simulink” (Simulink)

Verify a MATLAB Algorithm by Using Requirements-Based Tests

This example shows how to verify a MATLAB® algorithm by creating verification links from MATLAB code lines in functions and tests to requirements. This example uses a project that contains an algorithm to calculate the shortest path between two nodes on a graph.

Open the project.

```
slreqShortestPathProjectStart
```

Examine the Project Artifacts

The project contains:

- Requirement sets for functional and test requirements, located in the `requirements` folder
- A MATLAB algorithm, located in the `src` folder
- MATLAB unit tests, located in the `tests` folder
- Links from MATLAB code lines to requirements, stored `.slmx` files located in the `src` and `tests` folders
- Scripts to automate project analysis, located in the `scripts` folder

Open the Functional Requirement Set

The `shortest_path_func_reqs` requirement set captures the functional behavior that the `shortest_path` function requires. The requirements describe the nominal behavior and the expected behavior for invalid conditions, such as when the inputs to the function are not valid. Open the requirement set in the **Requirements Editor**.

```
funcReqs = slreq.open("shortest_path_func_reqs");
```

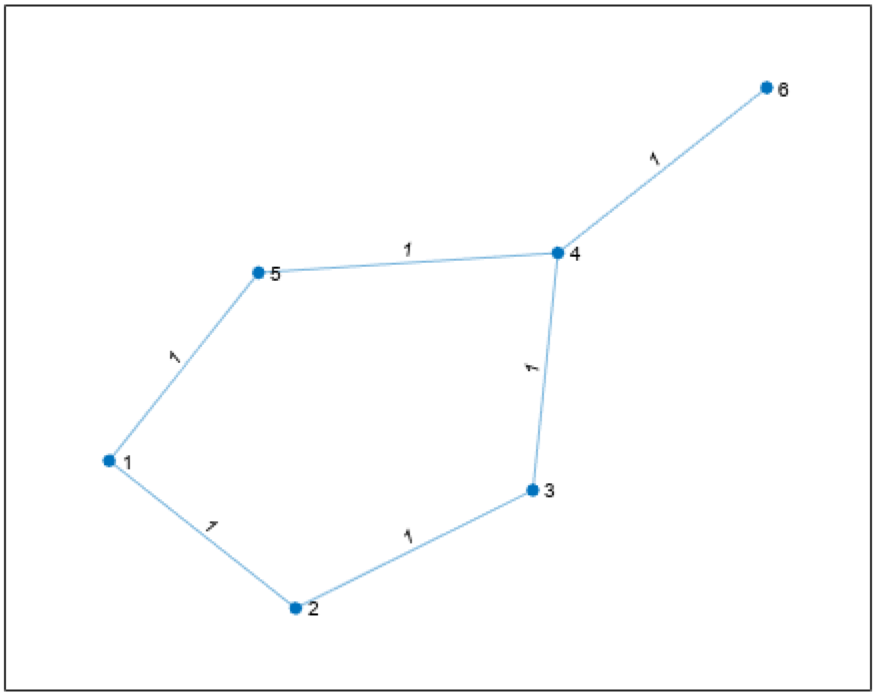
Use the Shortest Path Function

The `shortest_path` function tests the validity of the inputs to the function and then uses the Djikstra algorithm to calculate the number of edges in the shortest path between two nodes on a graph. The inputs to the function are an adjacency matrix that represents a graph, the starting node, and the ending node. For example, consider this adjacency matrix that represents a graph with six nodes.

```
A = [0 1 0 0 1 0;
    1 0 1 0 0 0;
    0 1 0 1 0 0;
    0 0 1 0 1 1;
    1 0 0 1 0 0;
    0 0 0 1 0 0];
```

Create a graph from the matrix and plot it.

```
G = graph(A);
plot(G,EdgeLabel=G.Edges.Weight)
```



Calculate the number of edges in the shortest path between nodes 1 and 6.

```
pathLength = shortest_path(A,1,6)
```

```
pathLength = 3
```

Open the Test Requirement Set

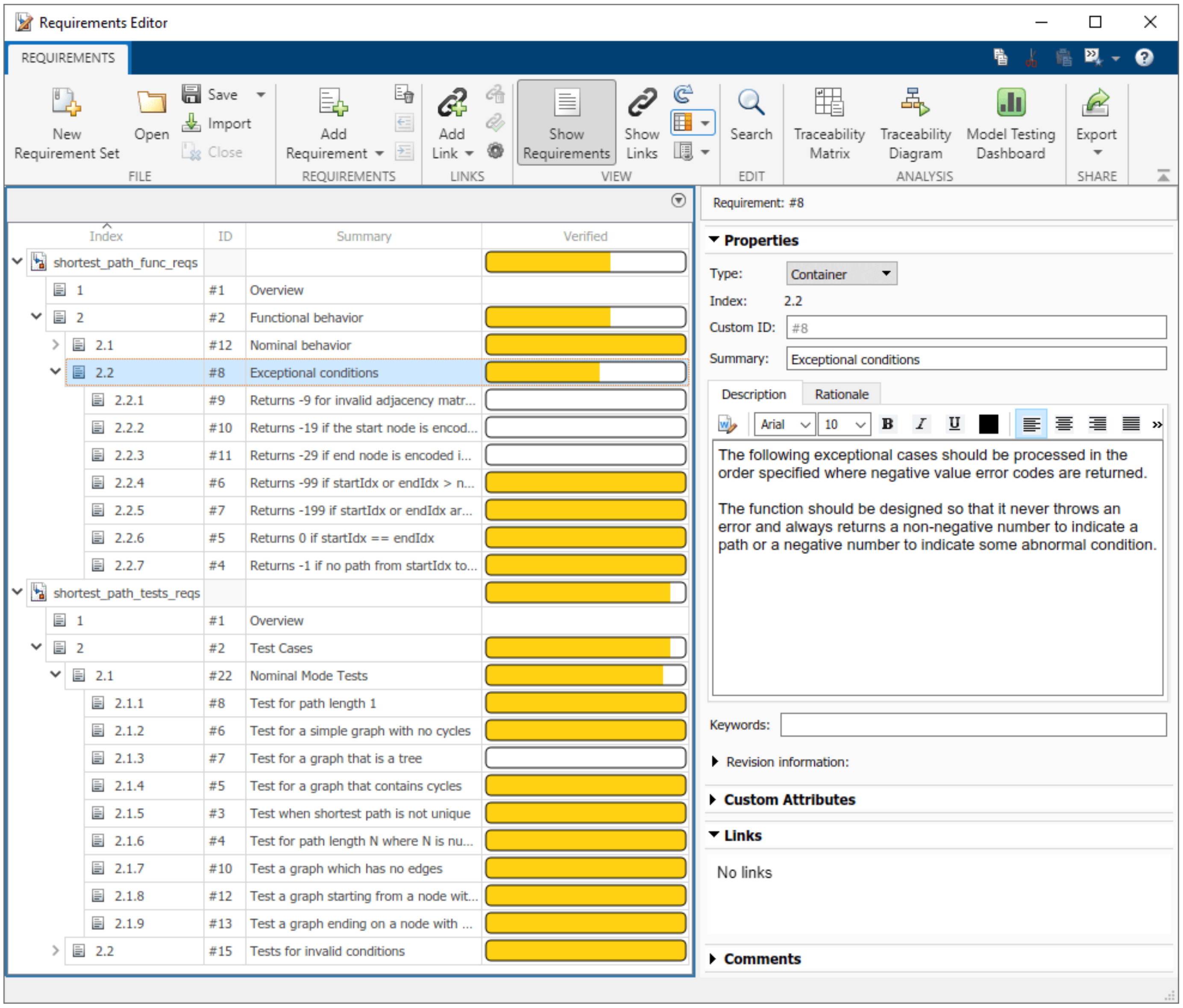
The `shortest_path_tests_reqs` requirement set contains test requirements that describe the functional behavior that must be tested by a test case. The test requirements are derived from the functional requirements. There are test requirements for the nominal behavior and for the invalid conditions. Open the requirement set in the **Requirements Editor**.

```
testReqs = slreq.open("shortest_path_tests_reqs");
```

The class-based MATLAB unit tests in `graph_unit_tests` implement the test cases described in `shortest_path_tests_reqs`. The class contains test methods based on the test requirements from `shortest_path_tests_reqs`. The class also contains the `verify_path_length` method, which the test cases use as a qualification method to verify that the expected and actual results are equal. The class also contains static methods that create adjacency matrices for the test cases.

View the Verification Status

To view the verification status, in the **Requirements Editor** toolstrip, in the **View** section, click **Columns** and select **Verification Status**. Three of the functional requirements and one test requirement are missing verification links. The verification status is yellow for each requirement, which indicates that the linked tests have not run.



Index	ID	Summary	Verified
shortest_path_func_reqs			
1	#1	Overview	
2	#2	Functional behavior	
2.1	#12	Nominal behavior	
2.2	#8	Exceptional conditions	
2.2.1	#9	Returns -9 for invalid adjacency matr...	
2.2.2	#10	Returns -19 if the start node is encod...	
2.2.3	#11	Returns -29 if end node is encoded i...	
2.2.4	#6	Returns -99 if startIdx or endIdx > n...	
2.2.5	#7	Returns -199 if startIdx or endIdx ar...	
2.2.6	#5	Returns 0 if startIdx == endIdx	
2.2.7	#4	Returns -1 if no path from startIdx to...	
shortest_path_tests_reqs			
1	#1	Overview	
2	#2	Test Cases	
2.1	#22	Nominal Mode Tests	
2.1.1	#8	Test for path length 1	
2.1.2	#6	Test for a simple graph with no cycles	
2.1.3	#7	Test for a graph that is a tree	
2.1.4	#5	Test for a graph that contains cycles	
2.1.5	#3	Test when shortest path is not unique	
2.1.6	#4	Test for path length N where N is nu...	
2.1.7	#10	Test a graph which has no edges	
2.1.8	#12	Test a graph starting from a node wit...	
2.1.9	#13	Test a graph ending on a node with ...	
2.2	#15	Tests for invalid conditions	

Requirement: #8

Properties

Type: Container

Index: 2.2

Custom ID: #8

Summary: Exceptional conditions

Description | Rationale

The following exceptional cases should be processed in the order specified where negative value error codes are returned.

The function should be designed so that it never throws an error and always returns a non-negative number to indicate a path or a negative number to indicate some abnormal condition.

Keywords:

Revision information:

Custom Attributes

Links

No links

Comments

Run the tests and update the verification status for the requirement sets by using the `runTests` method.

```
status1 = runTests(funcReqs);
```

```
Running graph_unit_tests
.......... ......
Done graph_unit_tests
__________
```

```
status2 = runTests(testReqs);
```

```
Running graph_unit_tests
.......... ....
Done graph_unit_tests
__________
```

The verification status is green to indicate that the linked tests passed. However, some of the requirements do not have links to tests.

Identify Traceability Gaps in the Project

The functional and test requirements are linked to code lines in the `shortest_path` and `graph_unit_tests` files, but the traceability is not complete. Use a traceability matrix to identify requirements that are not linked to tests and to create links to make the requirements fully traceable.

Find the Missing Links with a Traceability Matrix

Create a traceability matrix for both requirement sets with the requirements on the top and the unit tests on the left. For more information about traceability matrices, see “Track Requirement Links with a Traceability Matrix”

```
mtxOpts = slreq.getTraceabilityMatrixOptions;
mtxOpts.topArtifacts = {'shortest_path_func_reqs.slreqx','shortest_path_tests_reqs.slreqx'};
mtxOpts.leftArtifacts = {'graph_unit_tests'};
slreq.generateTraceabilityMatrix(mtxOpts)
```

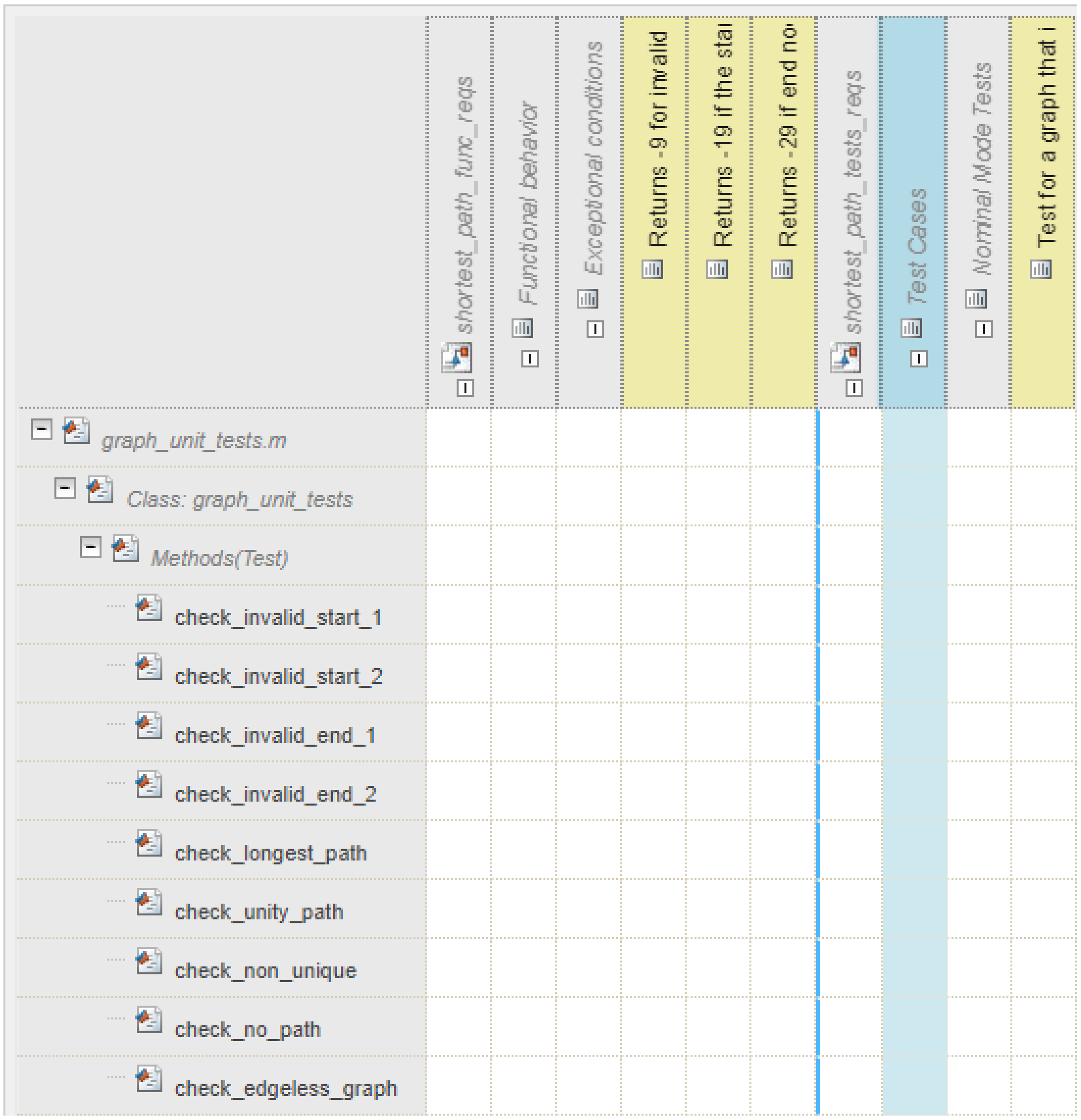
In the **Filter Panel**, in the **Top** section, filter the matrix to show only the functional requirements not linked to tests by clicking:

- **Top** > **Link** > **Missing Links**
- **Top** > **Type** > **Functional**

In the **Left** section, show only the test functions in the `graph_unit_tests` file by clicking:

- **Left** > **Type** > **Function**
- **Left** > **Attributes** > **Test**

Click **Highlight Missing Links** in the toolstrip.



	shortest_path_func_reqs	Functional behavior	Exceptional conditions	Returns -9 for invalid	Returns -19 if the star	Returns -29 if end no	shortest_path_tests_reqs	Test Cases	Nominal Mode Tests	Test for a graph that i
graph_unit_tests.m										
Class: graph_unit_tests										
Methods(Test)										
check_invalid_start_1										
check_invalid_start_2										
check_invalid_end_1										
check_invalid_end_2										
check_longest_path										
check_unity_path										
check_non_unique										
check_no_path										
check_edgeless_graph										

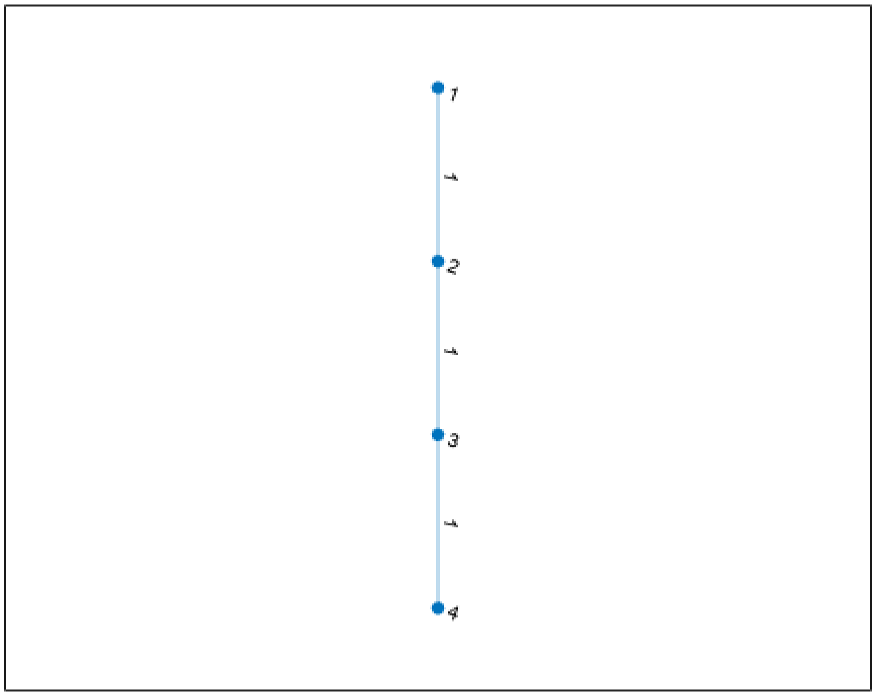
The Traceability Matrix window shows the three functional requirements and one test requirement that are missing verification links.

Create Verification Links for Requirements

The test requirement 2.1.3, `Test for a graph that is a tree`, is not linked to a test. A tree is a graph in which any two nodes are only connected by one path.

The test case `check_invalid_start_1` tests a tree graph by using the `graph_straight_seq` static method to create the adjacency matrix. Use the `graph_straight_seq` method to view the tree graph.

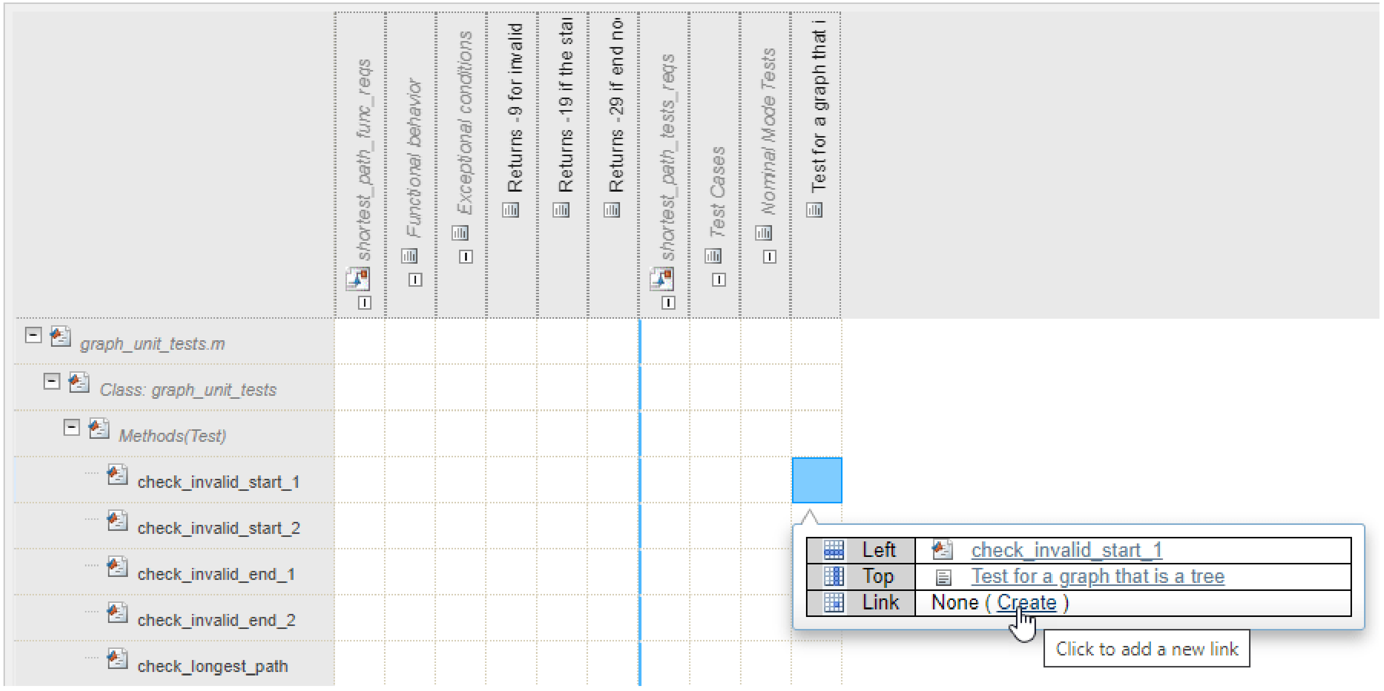
```
A = graph_unit_tests.graph_straight_seq;
G = graph(A);
plot(G,EdgeLabel=G.Edges.Weight)
```



Create a link from the `Test for a graph that is a tree` requirement to the `check_invalid_start_1` test case by using the traceability matrix you previously generated.

```
slreq.generateTraceabilityMatrix(mtxOpts)
```

Click the cell that corresponds to the requirement and the test and select **Create**. In the Create Link dialog box, click **Create**.



Update the verification status in the **Requirements Editor** by running the tests linked to the test requirements. The `check_invalid_start_1` test verifies the `Test for a graph that is a tree` requirement.

```
status3 = runTests(testReqs);
```

```
Running graph_unit_tests
.......... ....
Done graph_unit_tests
__________
```

Additionally, three functional requirements do not have links to tests:

- Requirement 2.2.1: `Returns -9 for invalid adjacency matrices`
- Requirement 2.2.2: `Returns -19 if the start node is encoded incorrectly`
- Requirement 2.2.3: `Returns -29 if end node is encoded incorrectly`

There is a traceability gap for these requirements. You cannot fill this gap by creating links to tests because there are no tests that verify these requirements.

Fix Coverage and Traceability Gaps by Authoring Tests

The three functional requirements that do not have links to tests do have links to lines of code in the `shortest_path` function. Run the tests with coverage to determine if those lines of code in the `shortest_path` function are covered by tests.

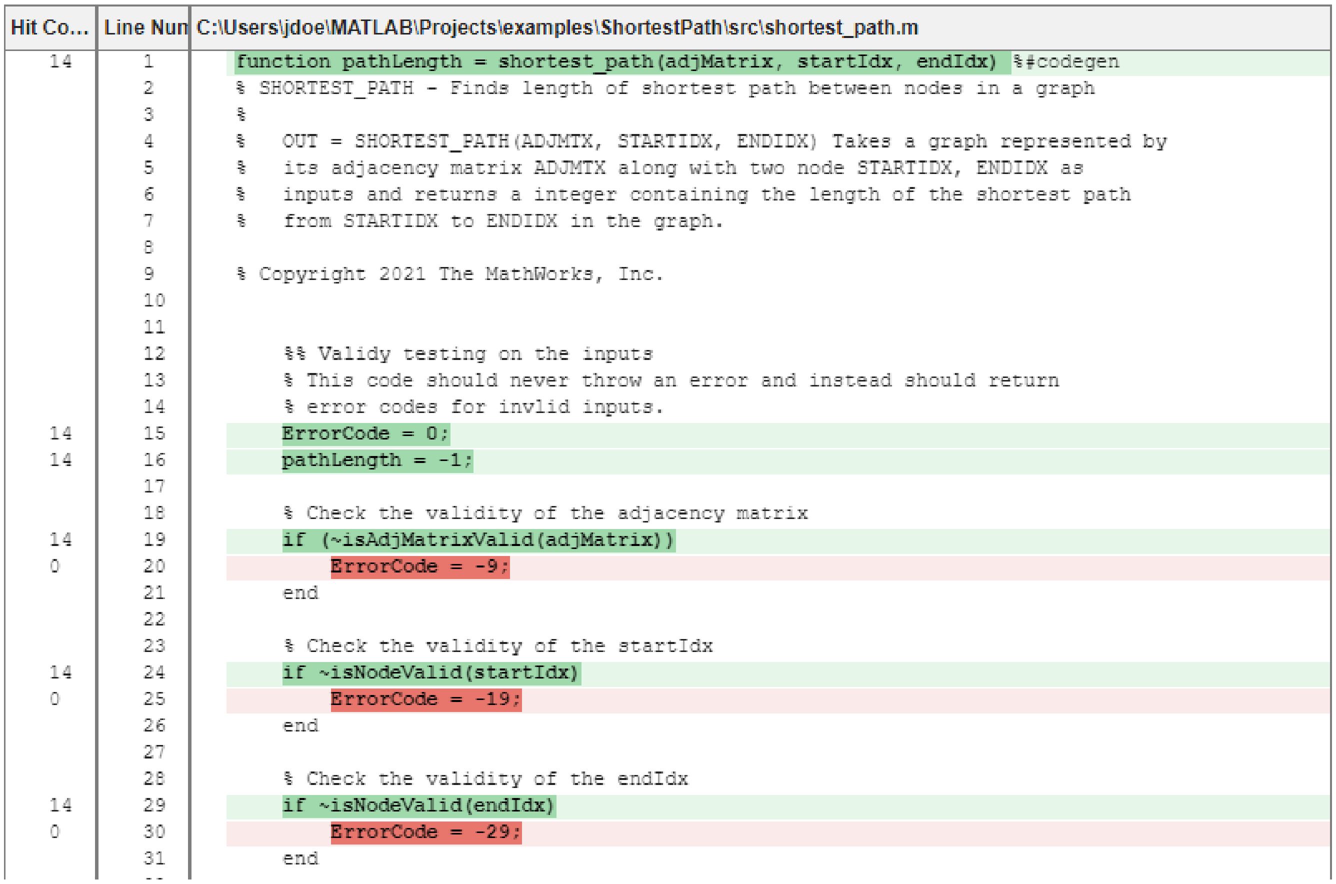
Run Tests with Coverage

Use the `RunTestsWithCoverage` script to run the tests with function and statement coverage and view the coverage in a report. For more information, see “Collect Statement and Function Coverage Metrics for MATLAB Source Code”.

```
RunTestsWithCoverage
```

```
Running graph_unit_tests
.......... ........
Done graph_unit_tests
__________

Code coverage report has been saved to:
 C:\Users\jdoe\MATLAB\Projects\examples\ShortestPath\coverageReport\index.html
```

Open the coverage report. The error code statements on lines 20, 25, and 30 are not covered by tests.



```
Hit Co...  Line Num  C:\Users\jdoe\MATLAB\Projects\examples\ShortestPath\src\shortest_path.m
14         1         function pathLength = shortest_path(adjMatrix, startIdx, endIdx) %#codegen
           2         % SHORTEST_PATH - Finds length of shortest path between nodes in a graph
           3         %
           4         %   OUT = SHORTEST_PATH(ADJMTX, STARTIDX, ENDIDX) Takes a graph represented by
           5         %   its adjacency matrix ADJMTX along with two node STARTIDX, ENDIDX as
           6         %   inputs and returns a integer containing the length of the shortest path
           7         %   from STARTIDX to ENDIDX in the graph.
           8
           9         % Copyright 2021 The MathWorks, Inc.
           10
           11
           12            %% Validy testing on the inputs
           13            % This code should never throw an error and instead should return
           14            % error codes for invlid inputs.
14         15            ErrorCode = 0;
14         16            pathLength = -1;
           17
           18            % Check the validity of the adjacency matrix
14         19            if (~isAdjMatrixValid(adjMatrix))
0          20                ErrorCode = -9;
           21            end
           22
           23            % Check the validity of the startIdx
14         24            if ~isNodeValid(startIdx)
0          25                ErrorCode = -19;
           26            end
           27
           28            % Check the validity of the endIdx
14         29            if ~isNodeValid(endIdx)
0          30                ErrorCode = -29;
           31            end
```

Note that the coverage gap for these code lines and the traceability gap for requirements 2.2.1, 2.2.2, and 2.2.3 refer to the same error codes. You can close the coverage and traceability gaps simultaneously by authoring tests for these lines of code and creating links to the requirements.

Improve Coverage by Authoring New Tests

Create tests that improve the coverage for the tests and verify requirements 2.2.1, 2.2.2, and 2.2.2. Open the `graph_unit_tests` test file.

```
open("graph_unit_tests.m");
```

These functions test the three error codes. Copy and paste the code in line 4, in the test methods section of the `graph_unit_tests` file, then save the file.

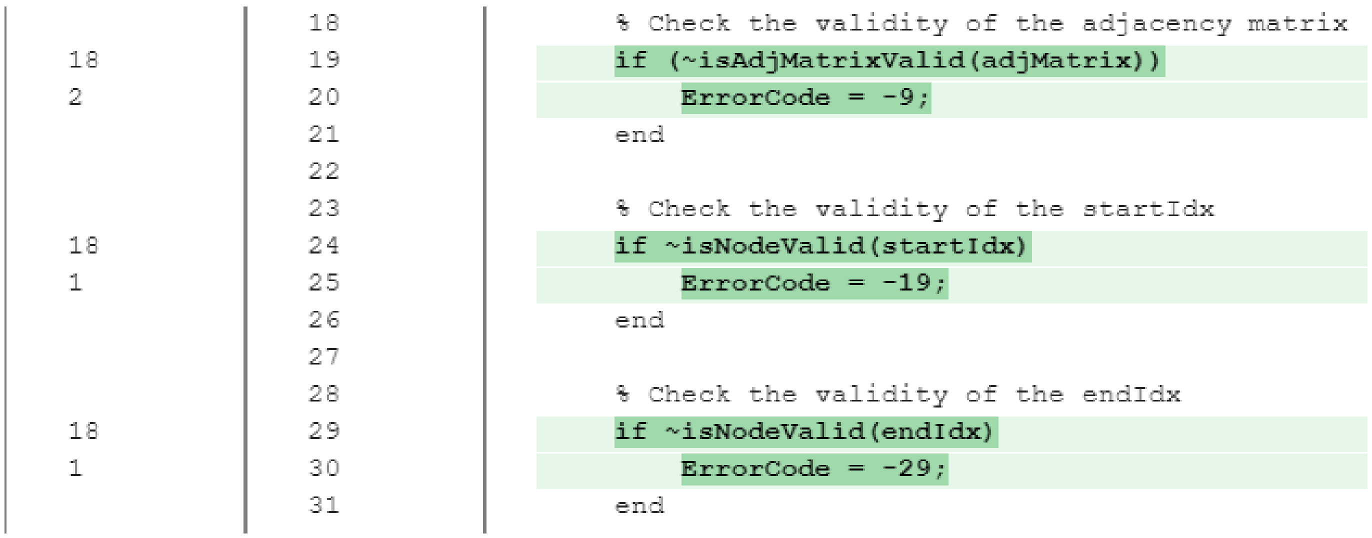
```
function check_invalid_nonsquare(testCase)
    adjMatrix = zeros(2,3);
    startIdx = 1;
    endIdx = 1;
    expOut = -9;
    verify_path_length(testCase, adjMatrix, startIdx, endIdx, expOut, ...
        'Graph is not square');
end

function check_invalid_entry(testCase)
    adjMatrix = 2*ones(4,4);
    startIdx = 1;
    endIdx = 1;
    expOut = -9;
    verify_path_length(testCase, adjMatrix, startIdx, endIdx, expOut, ...
        'Adjacency matrix is not valid');
end

function check_invalid_noninteger_startnode(testCase)
    adjMatrix = zeros(4,4);
    startIdx = 1.2;
    endIdx = 1;
    expOut = -19;
    verify_path_length(testCase, adjMatrix, startIdx, endIdx, expOut, ...
        'Start node is not an integer');
end

function check_invalid_noninteger_endnode(testCase)
    adjMatrix = zeros(4,4);
    startIdx = 1;
    endIdx = 2.2;
    expOut = -29;
    verify_path_length(testCase, adjMatrix, startIdx, endIdx, expOut, ...
        'End node is not an integer');
end
```

Rerun the tests with coverage and open the coverage report.

```
RunTestsWithCoverage
```

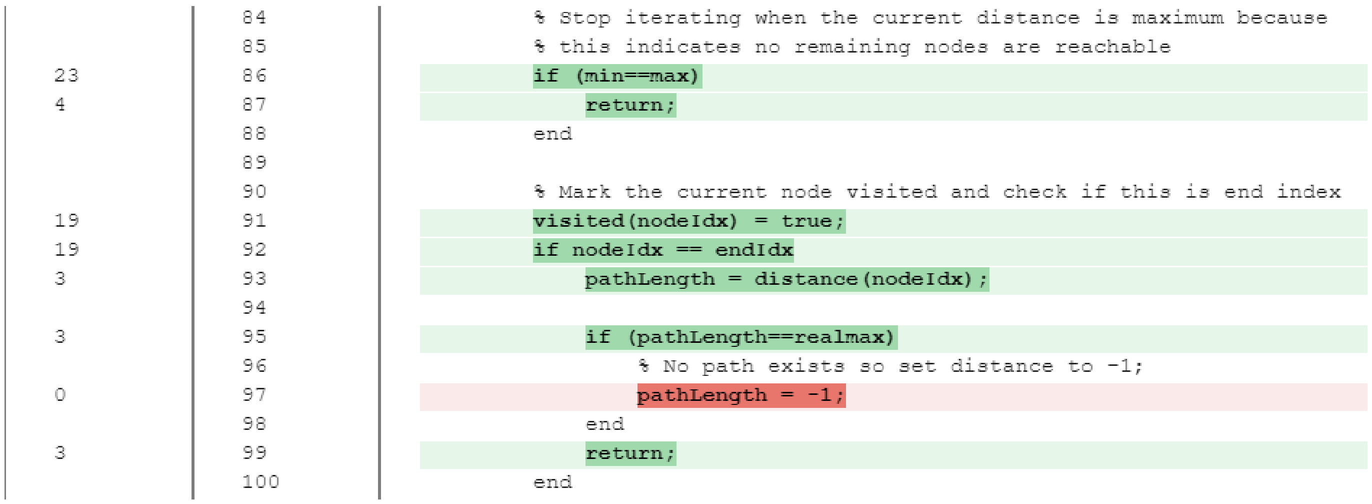
```
Running graph_unit_tests
.......... ........
Done graph_unit_tests
__________

Code coverage report has been saved to:
 C:\Users\jdoe\MATLAB\Projects\examples\ShortestPath\coverageReport\index.html
```

The tests now cover the error code statements.



```
        18        % Check the validity of the adjacency matrix
18      19        if (~isAdjMatrixValid(adjMatrix))
2       20            ErrorCode = -9;
        21        end
        22
        23        % Check the validity of the startIdx
18      24        if ~isNodeValid(startIdx)
1       25            ErrorCode = -19;
        26        end
        27
        28        % Check the validity of the endIdx
18      29        if ~isNodeValid(endIdx)
1       30            ErrorCode = -29;
        31        end
```

However, there is a statement on line 97 that the tests do not cover. The conditions that require the tests to cover the statement on line 97 also cause the `return` on line 87 to execute, which means that the statement on 97 is not reachable and is dead logic.



```
        84        % Stop iterating when the current distance is maximum because
        85        % this indicates no remaining nodes are reachable
23      86        if (min==max)
4       87            return;
        88        end
        89
        90        % Mark the current node visited and check if this is end index
19      91        visited(nodeIdx) = true;
19      92        if nodeIdx == endIdx
3       93            pathLength = distance(nodeIdx);
        94
3       95            if (pathLength==realmax)
        96                % No path exists so set distance to -1;
0       97                pathLength = -1;
        98            end
3       99            return;
        100       end
```

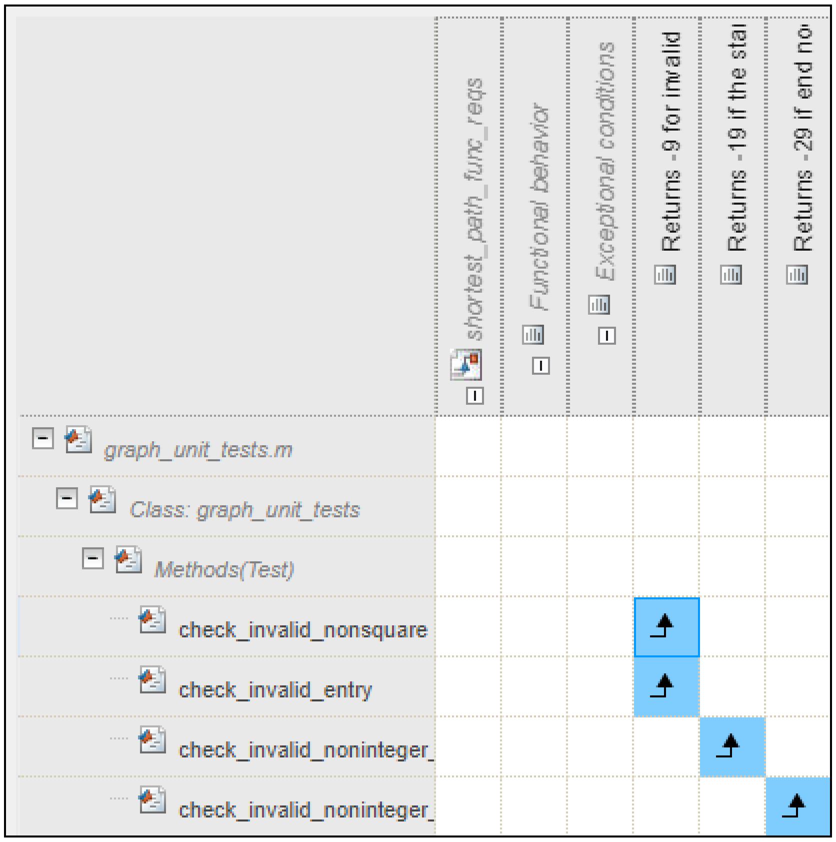
Fix Requirement Traceability Gaps

Regenerate the traceability matrix, apply the same filters from before, then click **Highlight Missing Links** in the toolstrip.

```
slreq.generateTraceabilityMatrix(mtxOpts)
```

- **Top** > **Link** > **Missing Links**
- **Top** > **Type** > **Functional**
- **Left** > **Type** > **Function**
- **Left** > **Attributes** > **Test**

Create links between the error code requirements and the new tests.



	shortest_path_func_reqs	Functional behavior	Exceptional conditions	Returns -9 for invalid	Returns -19 if the sta	Returns -29 if end no
graph_unit_tests.m						
Class: graph_unit_tests						
Methods(Test)						
check_invalid_nonsquare				↑		
check_invalid_entry				↑		
check_invalid_noninteger_					↑	
check_invalid_noninteger_						↑

Update the verification status in the **Requirements Editor** by re-running the tests linked to both requirement sets.

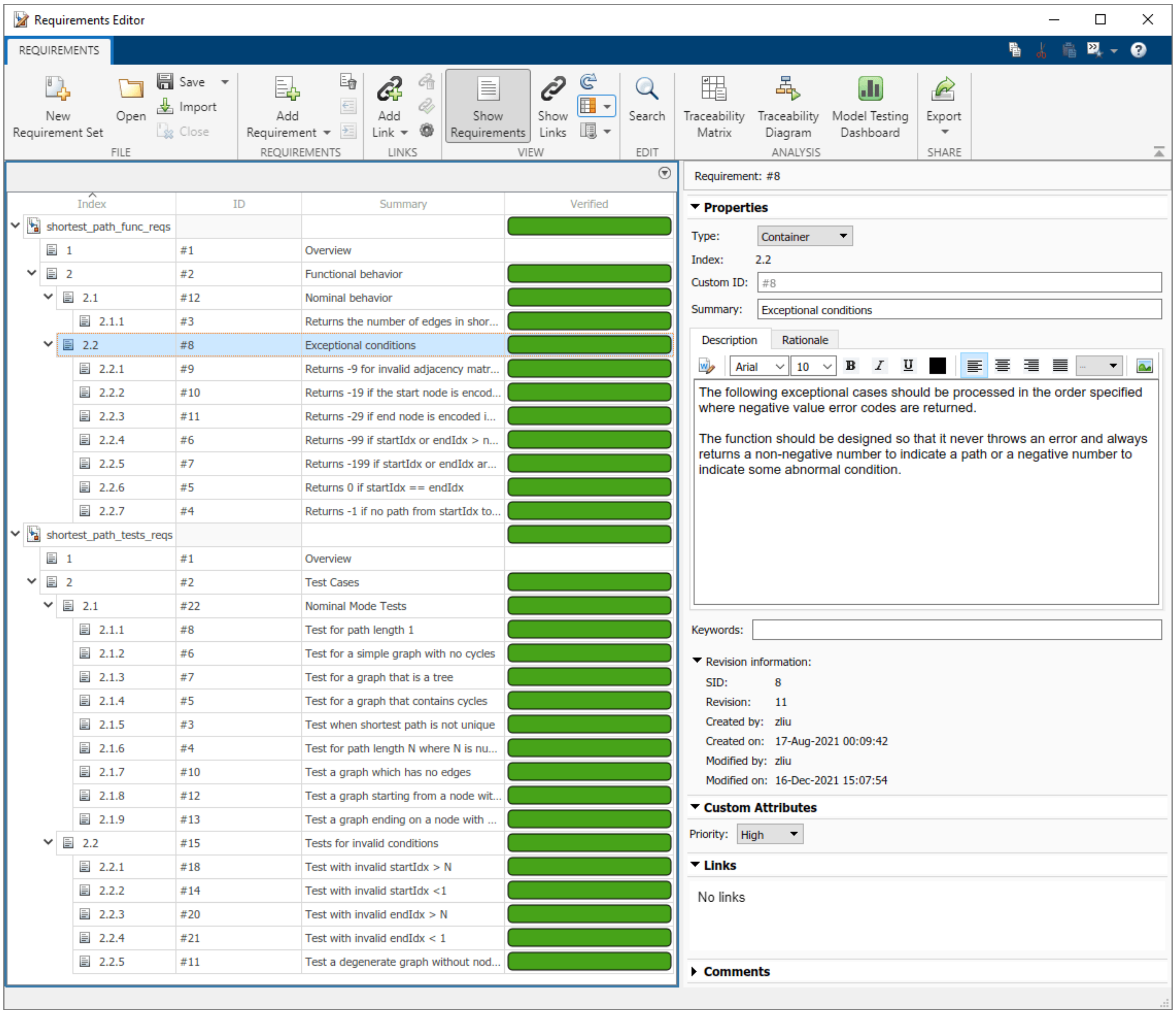
```
status4 = runTests(funcReqs);
```

```
Running graph_unit_tests
.......... ......
Done graph_unit_tests
__________
```

```
status5 = runTests(testReqs);
```

```
Running graph_unit_tests
.......... ....
Done graph_unit_tests
__________
```

All requirements have links to tests and all tests pass.



Index	ID	Summary	Verified
shortest_path_func_reqs			🟩
1	#1	Overview	
2	#2	Functional behavior	🟩
2.1	#12	Nominal behavior	🟩
2.1.1	#3	Returns the number of edges in shor...	🟩
2.2	#8	Exceptional conditions	🟩
2.2.1	#9	Returns -9 for invalid adjacency matr...	🟩
2.2.2	#10	Returns -19 if the start node is encod...	🟩
2.2.3	#11	Returns -29 if end node is encoded i...	🟩
2.2.4	#6	Returns -99 if startIdx or endIdx > n...	🟩
2.2.5	#7	Returns -199 if startIdx or endIdx ar...	🟩
2.2.6	#5	Returns 0 if startIdx == endIdx	🟩
2.2.7	#4	Returns -1 if no path from startIdx to...	🟩
shortest_path_tests_reqs			🟩
1	#1	Overview	
2	#2	Test Cases	🟩
2.1	#22	Nominal Mode Tests	🟩
2.1.1	#8	Test for path length 1	🟩
2.1.2	#6	Test for a simple graph with no cycles	🟩
2.1.3	#7	Test for a graph that is a tree	🟩
2.1.4	#5	Test for a graph that contains cycles	🟩
2.1.5	#3	Test when shortest path is not unique	🟩
2.1.6	#4	Test for path length N where N is nu...	🟩
2.1.7	#10	Test a graph which has no edges	🟩
2.1.8	#12	Test a graph starting from a node wit...	🟩
2.1.9	#13	Test a graph ending on a node with ...	🟩
2.2	#15	Tests for invalid conditions	🟩
2.2.1	#18	Test with invalid startIdx > N	🟩
2.2.2	#14	Test with invalid startIdx <1	🟩
2.2.3	#20	Test with invalid endIdx > N	🟩
2.2.4	#21	Test with invalid endIdx < 1	🟩
2.2.5	#11	Test a degenerate graph without nod...	🟩

Requirement: #8

Properties

Type: Container
Index: 2.2
Custom ID: #8
Summary: Exceptional conditions

Description | Rationale

The following exceptional cases should be processed in the order specified where negative value error codes are returned.

The function should be designed so that it never throws an error and always returns a non-negative number to indicate a path or a negative number to indicate some abnormal condition.

Keywords:

Revision information:
SID: 8
Revision: 11
Created by: zliu
Created on: 17-Aug-2021 00:09:42
Modified by: zliu
Modified on: 16-Dec-2021 15:07:54

Custom Attributes

Priority: High

Links

No links

Comments

Trace Requirements in Generated Code

Use Embedded Coder® to generate code from the `shortest_path` algorithm and include requirements comments that allow you to trace the requirements in the generated code. For more information, see “Requirements Traceability for Code Generated from MATLAB Code”.

Create a code configuration object to generate code with a LIB build type.

```
cfg = coder.config("lib","ecoder",true);
```

Enable the code configuration parameter to include requirements comments in the generated code.

```
cfg.ReqsInCode = true;
```

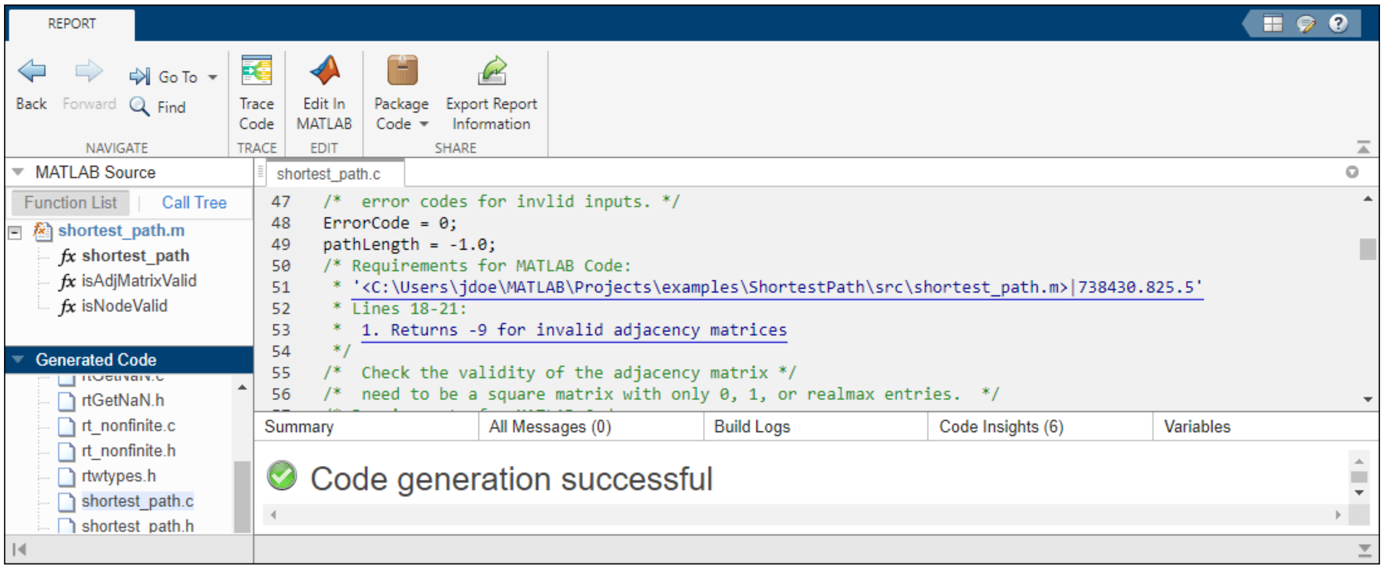
Use `coder.typeof` (MATLAB Coder) to define a variable-sized double array with a maximum size of 100x100 and a scalar double to use as inputs in the generated code.

```
mtxType = coder.typeof(ones(100,100),[],1);
scalarDblType = coder.typeof(1);
```

Generate C code from the `shortest_path` algorithm with the specified code configuration parameters and input types. Create a code generation report and launch the report.

```
codegen shortest_path -config cfg -args {mtxType, scalarDblType, scalarDblType} -launchreport
```

```
Code generation successful: View report
```



```
/*  error codes for invlid inputs. */
ErrorCode = 0;
pathLength = -1.0;
/* Requirements for MATLAB Code:
 * '<C:\Users\jdoe\MATLAB\Projects\examples\ShortestPath\src\shortest_path.m>|738430.825.5'
 * Lines 18-21:
 *  1. Returns -9 for invalid adjacency matrices
 */
/*  Check the validity of the adjacency matrix */
/*  need to be a square matrix with only 0, 1, or realmax entries.  */
```

Code generation successful

The `shortest_path.c` file contains comments with the summary of the linked requirement, the full file path of the `shortest_path.m` file, and the linked code lines.

See Also

`runTests` | `codegen` | `coder.runTest`

More About

- "Review Requirements Verification Status"
- "Requirements Traceability for Code Generated from MATLAB Code"
- "Author Class-Based Unit Tests in MATLAB"
- "Collect Statement and Function Coverage Metrics for MATLAB Source Code"