DRIVING EMBEDDED EXCELLENCE

ETAS SCODE Workbench 3.1

Getting Started

www.etas.com

SCODE Workbench 3.1 – Getting Started R01 EN – 04.2022

Contents

1. Safety and Privacy Information

In this chapter, you can find information about the intended use (section 1.2), and information about safety and privacy related topics.

Please adhere to the ETAS Safety Advice (**Help > Safety Advice**) and to the safety information given in the user documentation.

ETAS GmbH cannot be made liable for damage which is caused by incorrect use and not adhering to the safety messages.

1.1. Demands on the Technical State of the Product

The following special requirements are made to ensure safe operation:

- Take all information on environmental conditions into consideration before setup and operation (see the documentation of your computer, hardware, etc.).

1.2. Intended Use

The SCODE Workbench consists of the products SCODE-ANALYZER and SCODE-CONGRA. They provide support in the early phases of ECU software development.

Using the software tools of SCODE Workbench (System CO-DEsign), engineers, control systems technicians, and software developers, among others, can create model-based, structured, and easily understood solutions for ECU software that are then automatically verified.

SCODE-ANALYZER makes it possible to clearly describe and verify complex relationships in closed-loop control systems. To this end, the overall system is divided up into operating areas known as *modes* (for example, idle, full load, limp-home mode). Displaying the system this way is most beneficial when the software makes decisions or has a lot of variants.

With SCODE-CONGRA, software, function developers can describe control systems in exact, easy-to-understand mathematical terms, and graphically visualize the results. The description of system behavior is specified textually or graphically. Rule violations, inconsistencies, algebraic loops, and other important characteristics of the system are displayed precisely in the graph, and the user is offered options and functions to correct these *errors in the system* immediately.

ETAS GmbH cannot be made liable for damage which is caused by incorrect use and not adhering to the safety messages.

NOTE

This ETAS product fulfills standard quality management requirements. If requirements of specific safety standards (e.g. IEC 61508, ISO 26262, DO-178b, EN50128 and other similar standards) need to be fulfilled, these requirements must be explicitly defined and ordered by the customer. Before use of the product, customers must verify the compliance.

1.3. Classification of Safety Messages

Safety messages warn of dangers that can lead to personal injury or damage to property:

DANGER	**DANGER** indicates a hazardous situation that, if not avoided, will result in death or serious injury.
WARNING	**WARNING** indicates a hazardous situation that, if not avoided, could result in death or serious injury.
CAUTION	**CAUTION** indicates a hazardous situation that, if not avoided, could result in minor or moderate injury.
NOTICE	***NOTICE*** indicates a situation that, if not avoided, could result in damage to property.

1.4. Safety Information

Please adhere to the ETAS Safety Advice and to the following safety information to avoid injury to yourself and others as well as damage to the property.

See also section 1.3.

Further safety advice for this ETAS product is available in the following formats:

- In electronic form on the DVD. See Documentation\ETAS Safety Advice.pdf for details.
- The "ETAS Safety Advice" window that opens when you start the program, or when you select **Help > ETAS Safety Advice**.

1.5. Privacy Notice

Note that personal data is processed when using this product. As the controller, the purchaser undertakes to ensure the legal conformity of these processing activities in accordance with Art. 4 No. 7 of the General Data Protection Regulation (GDPR). As the manufacturer, ETAS GmbH is not liable for any mishandling of this data.

1.5.1. Data Categories

Note that this product creates files containing file names and file paths, e.g. for purposes of error analysis, referencing source libraries, or for communicating with third party programs.

The same file names and file paths may contain personal data, if they refer to the current user's personal directory or subdirectories (e.g., C:\Users\<*UserId*>\Documents\...).

If you do not want personal information to be included in the generated files, please make sure that

- the workspace of the product points to a directory without personal reference.
- all settings in the product (see menu **Window > Preferences** in the product) refer to directories and file names without personal reference.
- all project settings in the product (see menu **Project > Properties**) refer to directories and file names without personal reference.
- Windows environment variables refer to directories without personal reference because these environment variables are used by the product.

In this case, also make sure that the users of this product have read and write access to all relevant directories.

When using the ETAS License Manager in combination with user-based licenses, particularly the following personal data (categories) and/or data (categories) that can be traced back to a specific individual is recorded for the purposes of license management:

- User data: UserID
- Communication Data: IP address

1.5.2. Technical and Organizational Measures

This product does not itself encrypt the personal data that it records. Please ensure that the data recorded is secured by means of suitable technical or organizational measures in your IT system, e.g. by using classic anti-theft and access protection on the measurement hardware.

Personal data in generated files can be deleted by tools in the operating system.

2. About SCODE Workbench

This Getting Started guide provides relevant information to all the users who want to install and get to know the ETAS SCODE Workbench. The SCODE Workbench hosts the SCODE tools, SCODE-ANALYZER and SCODE-CONGRA.

SCODE Workbench is distributed as a standard Microsoft Windows installer. See chapter 3 for detailed installation information.

SCODE Workbench is an Eclipse-based product. If you are familiar with using an Eclipse environment then you should feel at home. If SCODE Workbench is the first Eclipse-based application you have used, open the help viewer and go to the Workbench User Guide to get more information on the basic Eclipse features.

SCODE-ANALYZER

SCODE-ANALYZER makes it possible to clearly describe and verify complex relationships in closed-loop control systems. The overall system is divided up into operating areas known as *modes* (for example, idle, full load, limp-home mode). Displaying the system this way is most beneficial when the software makes decisions or has a lot of variants.

New users of SCODE-ANALYZER are referred to chapter 4. You will learn how to work with SCODE-ANALYZER using examples.

A quick introduction to SCODE-ANALYZER is given in chapter 6, particularly section 6.1.

SCODE-CONGRA

SCODE-CONGRA is designed to help you define and analyze continuous systems, simulate them and generate code.

The system is described purely in form of variables, relations, and equations. The equations are undirected. Depending on which variables are marked as inputs, the equations are solved in the corresponding direction, and code is generated representing the results of this direction of equations.

New users of SCODE-CONGRA are referred to chapter 5. You will learn how to work with SCODE-CONGRA using various examples.

A quick introduction to SCODE-CONGRA is given in chapter 6, particularly section 6.2.

2.1. Finding Out More

Besides this User Guide, the online help is recommended — particularly when working with the user interface. It can be called up via **Help > Help Content** or context-sensitive (with F1) in the respective open operating window.

3. Installing SCODE Workbench

This chapter provides relevant information to all users who install, maintain or uninstall SCODE Workbench on a PC or a network.

NOTE

The SCODE Workbench installation includes both SCODE-ANALYZER and SCODE-CONGRA.

The licenses for SCODE-ANALYZER and SCODE-CONGRA must be bought separately.

3.1. Preparing the Installation

Check the delivery package to make sure it is complete and make sure your system corresponds to the system requirements. Depending on the operating system and network connection used, you must ensure that you have the necessary user privilege.

3.1.1. Delivery Scope

The installation disk of the SCODE Workbench contains the following content:

- SCODE-ANALYZER and SCODE-CONGRA program files
- PDF documentation for SCODE-ANALYZER and SCODE-CONGRA
- ETAS Safety Advice in PDF format
- Information on open-source components used in SCODE-ANALYZER and SCODE-CONGRA

3.1.2. Software Prerequisites and System Requirements

The software prerequisites and system requirements are listed in the release notes of the SCODE Workbench.

3.2. Installation

When you install the SCODE Workbench, both SCODE-ANALYZER and SCODE-CONGRA are installed automatically.

Keep in mind that you need separate licenses for SCODE-ANALYZER and SCODE-CONGRA.

3.2.1. Installation via Dialog Windows

To start the SCODE Workbench installation

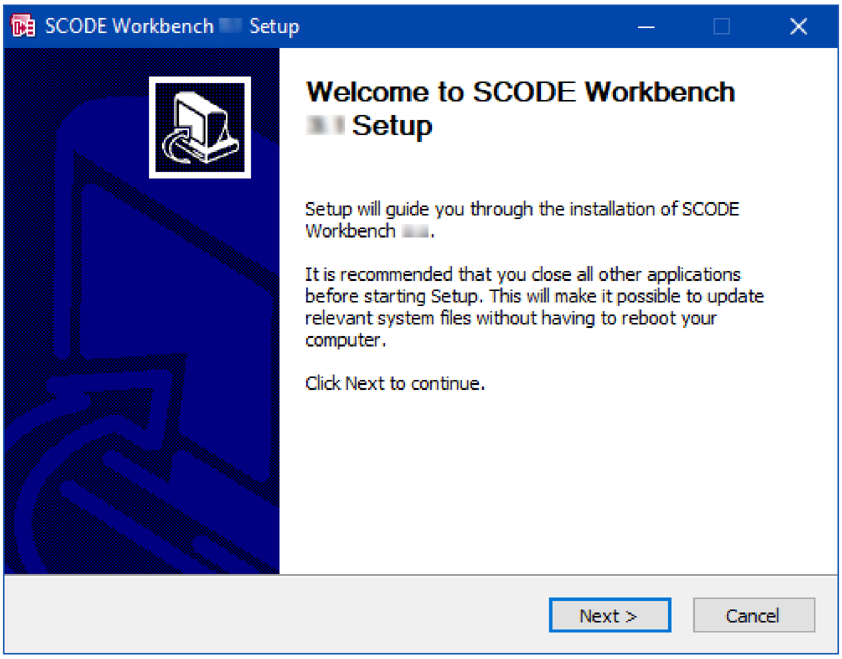
1. Insert the data carrier in the respective drive on your computer.

   An installation dialog window opens.

2. Follow the Installation link, then follow the Install SCODE Workbench 3.1 link.

3. Alternatively, select the driver in the Windows Explorer and run the `setup.exe` file from the Installation folder.

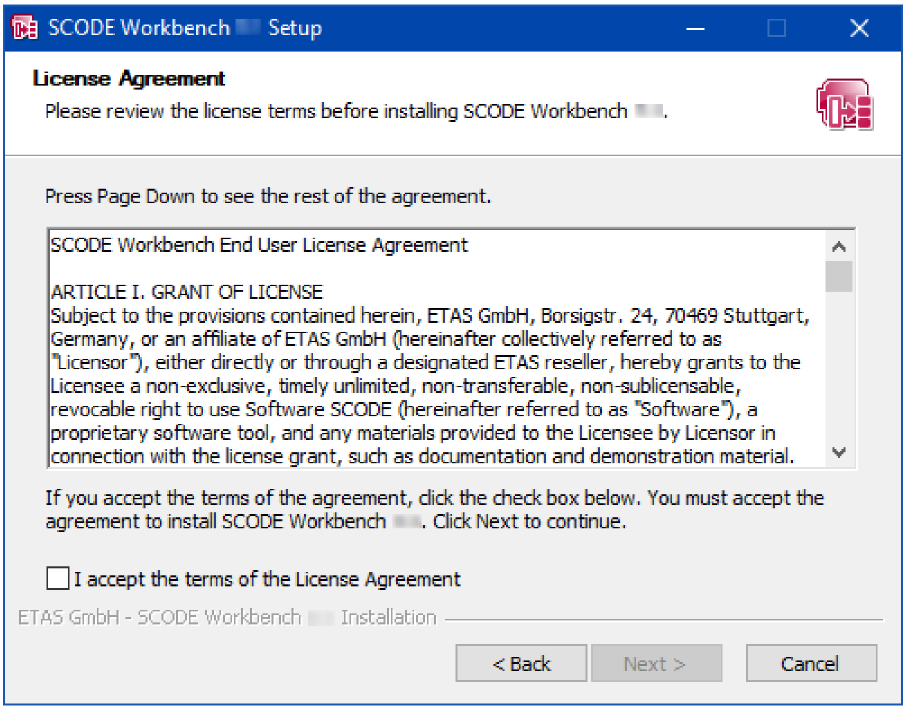
   The ETAS Installer is launched.



4. Click **Next** to get to the next installation window.

License agreement

Next, you have to accept the End User License Agreement.

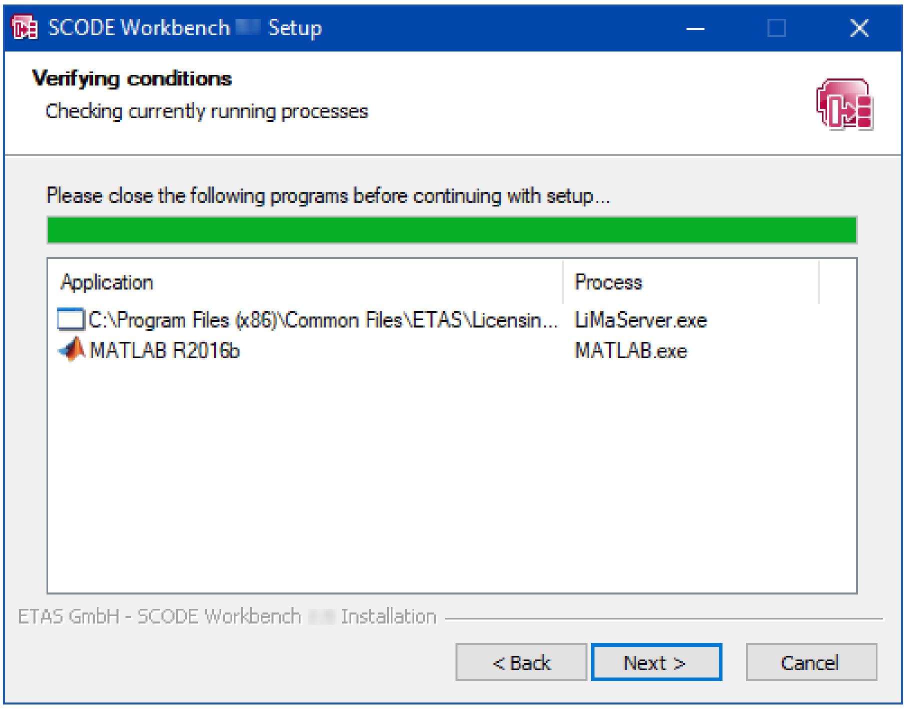


1. Read the license agreement, then activate the **I accept the terms of the License Agreement** option.

2. Click **Next**.

To check for blocking applications

The "Verifying conditions" window shows running applications that block the installation.



1. Close each blocking application with its native closing mechanism.

Or

2. Click **Next**.

   You are asked if you want to close the applications.

3. Click **Yes** to continue.

   If an application cannot be closed normally, you are asked if you want to kill the respective process.

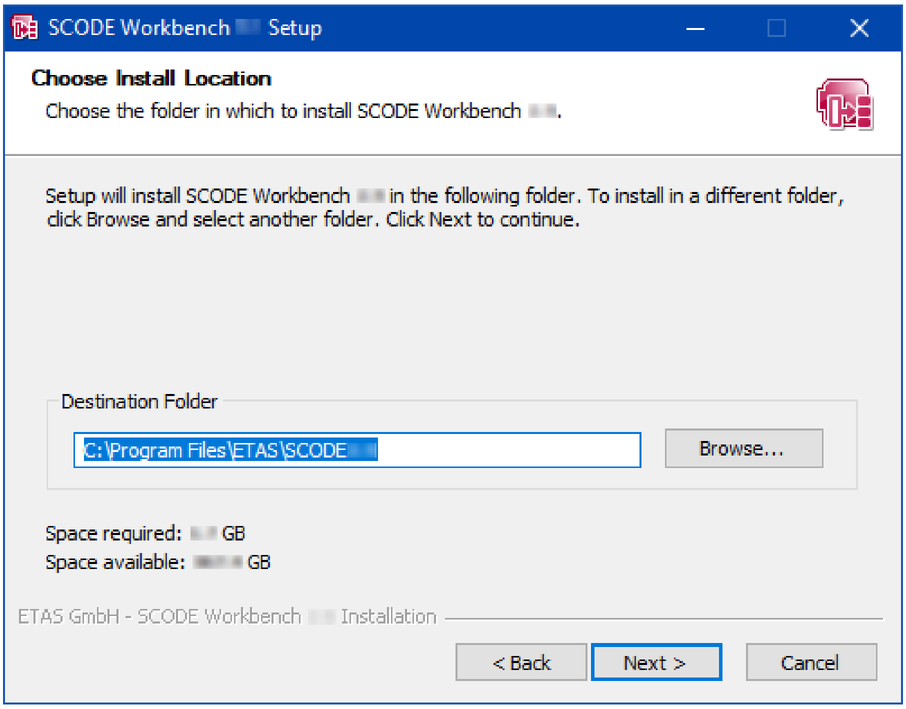
NOTICE	**Data loss due to process killing** Killing a process can lead to data loss. Save your data and make sure that no data will be lost before you agree to kill the process.

4. Click **Yes** to continue.

Once all blocking applications are closed, the installation continues automatically.

To define path settings

In the "Choose Install Location" window, you are prompted to enter a destination directory for the SCODE Workbench.



1. Enter or select (via the **Browse** button) a valid path.

   An invalid path deactivates the **Next** button. You have to correct the path before you can continue.

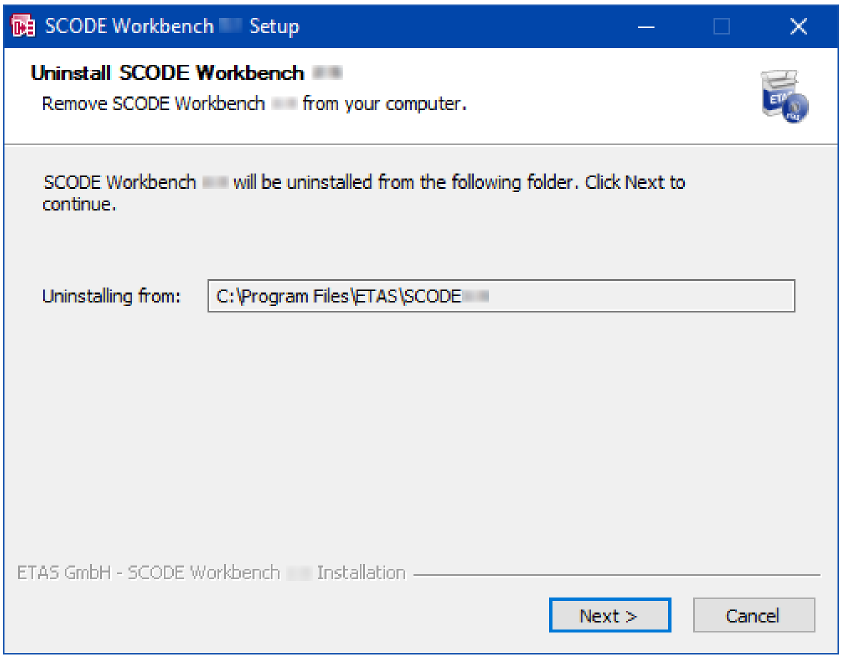
2. Click **Next**.

   If you selected an existing directory, the installer assumes that the SCODE Workbench 3.1 is installed in the selected directory. You are asked to uninstall the existing installation.

3. Click **Yes** to continue.

   If the existing folder does ***not*** contain an installation of the SCODE Workbench, the folder is deleted. Continue reading at "To specify a folder in the Start menu".

   If you selected an existing folder that contains an installation of the SCODE Workbench, the "Uninstall ETAS SCODE Workbench" window opens.



NOTICE

If you continue with **Next**, the connections between the old version and MATLAB® and Simulink® are kept.

This means the new version cannot be connected to MATLAB and Simulink during installation.

It is therefore strongly recommended that you do the following:

1. Cancel the installation.
2. Remove all connections between the old version and MATLAB and Simulink.

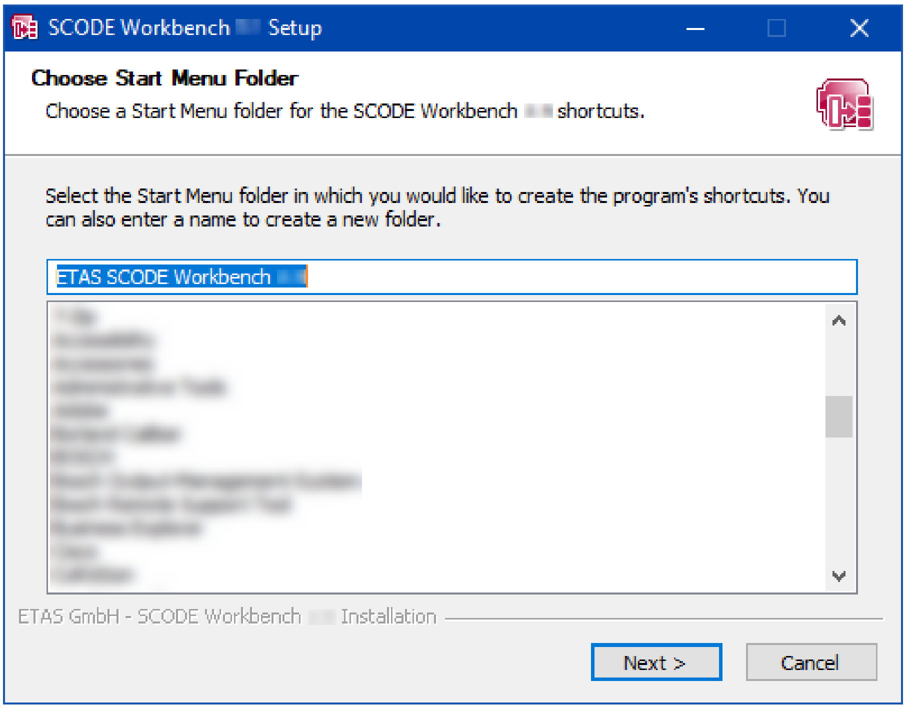
   See section 6.3.2 for an instruction.
3. Re-start the installation.

4. Click **Next**.

   The existing version is uninstalled. Once uninstallation is complete, the **Close** button is available.
5. Click **Close**.

   The installation continues.

To specify a folder in the Start menu



1. Do one of the following:
   - Accept the default folder name.
   - Enter a new folder name.

     You can enter folder and subfolder.

To install the SCODE Workbench

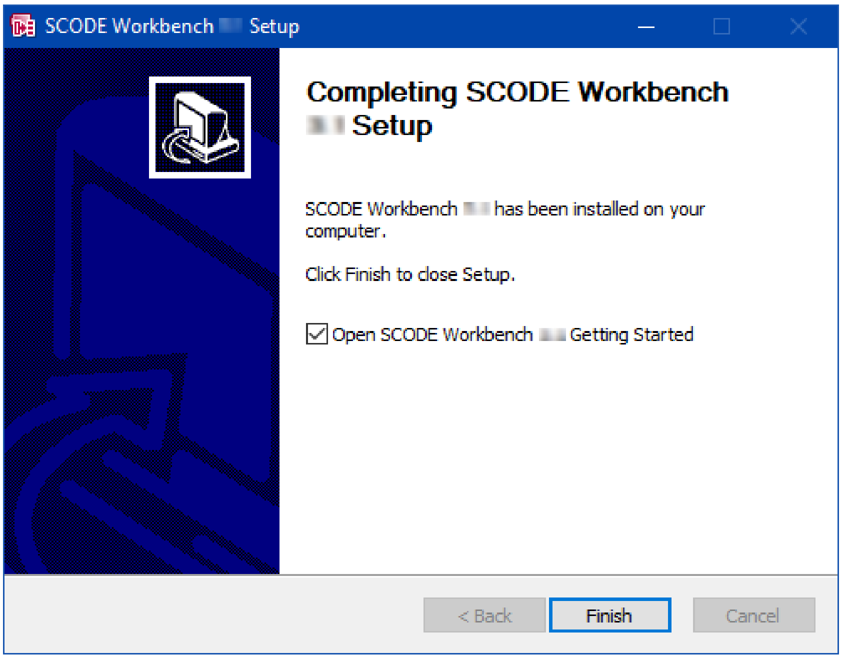
NOTE — The next step starts the installation. You cannot abort it.

1. In the "Choose Start Menu Folder" window, click **Install**.

   The installation is performed. A progress indicator shows how the installation is progressing. When the installation is complete, the "Installation Complete" window opens.

2. Click **Next**.

   You are prompted to finish the installation.



3. If desired, activate the **Open ETAS SCODE Workbench 3.1 Getting Started** option.
4. Click **Finish** to complete the installation.

In the Start menu, the specified folder (named `ETAS SCODE Workbench 3.1` by default; see also "To specify a folder in the Start menu") is created. It contains the following entries:

- **SCODE Workbench 3.1**

  The SCODE Workbench is started.

- **SCODE Workbench 3.1 Getting Started**

  Link to the Getting Started manual for SCODE Workbench.

- **SCODE Workbench 3.1 Release Notes**

  Link to the Release Notes for SCODE Workbench.

The ETAS License Manager has an entry **ETAS License Manager** in the **ETAS** program group of the Start menu.

The following icon is placed on the desktop of your computer:

3.2.2. Command-Line Installation

This section describes the command-line installation. Installation via dialog windows is described in section 3.2.1.

When you start the SCODE Workbench installation from a command line, you can use several command-line parameters to customize the installation.

NOTE The command-line options are case-sensitive. For example, `/S` will cause a silent installation, but `/s` will not.

`/?` or `/h`

Opens a window with the valid command line arguments.

`/S` or `/silent`

Silent installation mode. With this installation mode, no dialog windows requiring user information open.

Default values are used for all information normally requested in installation windows. Error messages are hidden, too.

NOTE `/silent` must be the first command-line argument. If other arguments precede it, `/silent` has no effect.

`/NCRC`

Skips CRC check of the installer (ignored if `CRCCheck force` is set in the installer).

`/D`

Sets the installation directory (`$INSTDIR`).

`/D` must be the last parameter in the *`command`* line. `/D` must not contain any quotes.

Syntax

without spaces — `/D=C:\ETAS\SCODE<x>.<y>` [1]

with spaces — `/D=C:\Program Files\SCODE`

Examples

`setup.exe /S /EULAAccepted`

Triggers a silent installation with default installation path and CRC check.

`setup.exe /NCRC /D=C:\Tools\SCODE<x>.<y>` [1]

Triggers a non-silent installation without CRC check and with user-defined installation directory.

3.3. Licensing

A valid license is required for using SCODE-ANALYZER, and a separate valid license is required for using SCODE-CONGRA. You can obtain the license file(s) required for licensing either from your tool coordinator or through a self-service portal under www.etas.com/support/licensing. To request the license file(s), you have to enter the activation number which you received from ETAS during the ordering process.

In the Windows Start menu, go to the app list and select **E > ETAS > ETAS License Manager**.

Follow the instructions given in the license manager dialog. For further information about, for example, the ETAS license models and borrowing a license, press F1 in the ETAS License Manager.

If you do not have a valid license for either SCODE-ANALYZER or SCODE-CONGRA, the respective tool will be available in grace mode for 14 days. After that, SCODE-ANALYZER or SCODE-CONGRA can no longer be used.

3.4. Uninstallation

The entire SCODE Workbench is uninstalled. You cannot uninstall SCODE-ANALYZER or SCODE-CONGRA individually.

NOTE

Before you uninstall a version of the SCODE Workbench, you must remove all connections between that version and MATLAB®/Simulink®.

Otherwise, a new version cannot be connected to MATLAB ®/Simulink®.

See section 6.3.2 for an instruction.

Use one of the following ways to start the uninstall process for the SCODE Workbench:

- **Programs and Features** from the Windows control panel
- **Apps > Apps & features** from the Windows Settings

To uninstall the SCODE Workbench

1. Start the uninstall procedure.

   A safety inquiry opens.

   NOTE

   The next step will start the uninstallation. The entire content of the installation directory will be deleted.

   You ***cannot*** cancel the uninstallation once it is running.

2. Click **Yes** to continue.

   A progress indicator shows how the uninstallation is progressing. Once uninstallation is complete, a success window opens.

3. Click **Close** to end the uninstallation.

[1] *<x>.<y>* is the SCODE Workbench version number

4. SCODE-ANALYZER Tutorial

This chapter contains a tutorial for SCODE-ANALYZER. A tutorial for SCODE-CONGRA can be found in chapter 5.

4.1. Introduction

Users who are not yet familiar with SCODE-ANALYZER will learn the basic working steps of SCODE-ANALYZER in this tutorial. The tutorial does not require any knowledge of SCODE-ANALYZER, but does assume that you are familiar with the Windows operating system and with Eclipse in general.

Motivation

The SCODE methodology aims at the following:

- reducing complexity
- determinism (100% complete, 100% consistent, all mode transitions are valid)
- 100% test coverage
- proof for correctness throughout the tool chain
- easy and fast modeling

For that purpose, the SCODE methodology separates control flow (discrete logic) and data flow (continuous computation). SCODE-ANALYZER handles the discrete control flow, while SCODE-CONGRA handles continuous data flow.

Workflow

Working with SCODE-ANALYZER comprises the following steps, which are covered by this tutorial:

A. Create a SCODE-ANALYZER project.

   See also section 4.2, “Lesson 1: Creating a SCODE-ANALYZER Project”.

B. Define the problem space, the combinatorial combinations of the system context.

   See also section 4.3, “Lesson 2: Defining the Problem Space”.

C. Define the valid and invalid operational modes via rules on the problem space.

   See also section 4.4, “Lesson 3: Defining Modes”.

D. Generate code for the modes.

   See also section 4.5, “Lesson 4: Code Generation from Mode Invariants”.

E. Define the mode transitions / events via rules.

   See also section 4.6, “Lesson 5: Defining Events and Transitions”.

F. Generate code for the mode transition matrix.

   See also section 4.7, “Lesson 6: Code Generation from Mode Transition Matrix”.

G. Generate a report for the SCODE-ANALYZER project.

   See also section 4.8, “Lesson 7: Generating a Report”.

4.1.1. Example: Hybrid Car

The example system for this tutorial is a car with combustion engine and electric engine/generator. It consists of the following components:

Combustion engine	Can get disconnected (e.g. by clutch).
Electric engine/ generator	Converts mechanical power to electrical or vice versa. Can be disconnected (e.g. by clutch).
Battery	
Brake	The system can recuperate energy while braking.
Switch	Used to select electric operation.

Table 1. Example system — components

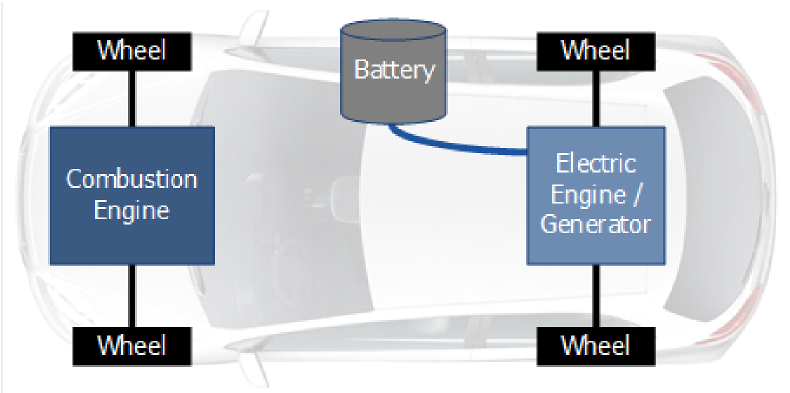


Figure 1. Example system — draft

4.2. Lesson 1: Creating a SCODE-ANALYZER Project

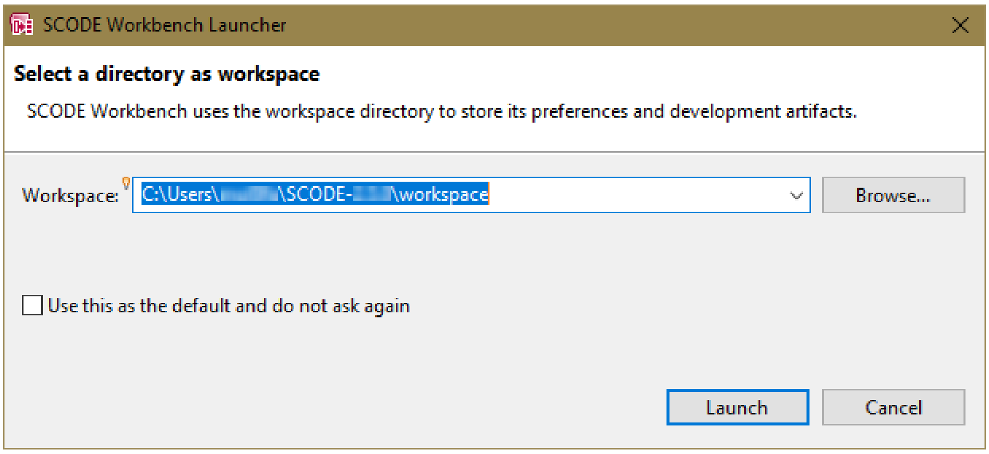
In the first lesson of this tutorial, you will start the SCODE Workbench, open a workspace, and create a SCODE-ANALYZER project.

NOTE	It is recommended that you use a separate workspace for the tutorial.

To create a workspace

1. Start the SCODE Workbench.

   The "SCODE Workbench Launcher" window opens, asking for a workspace location.



2. In that window, enter or select (via the **Browse** button) a path and name for your workspace.

   This tutorial uses a workspace named `WS_tutorial`.

3. Click **OK**.

   If you entered a directory that does not yet exist, it is created now.

   The SCODE Workbench opens. It shows the welcome page.

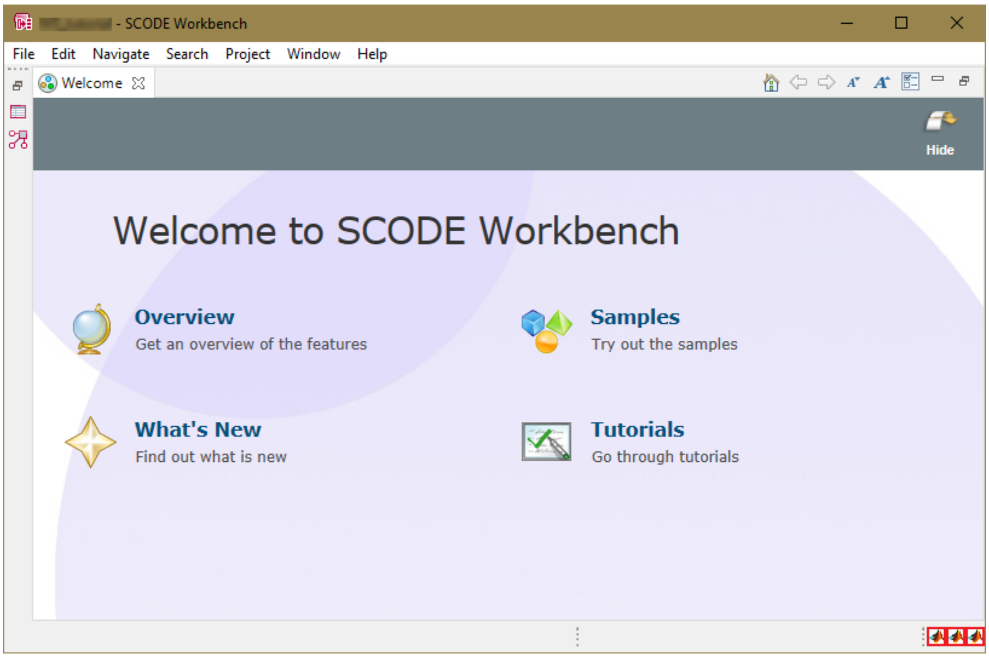


Figure 2. SCODE Workbench window, showing the Welcome page

4. To reach the workbench, click the **Hide** button at the top right.

   If you selected a new workspace, all views are empty (see Figure 3). If you selected an existing workspace, that workspace is shown in the views.

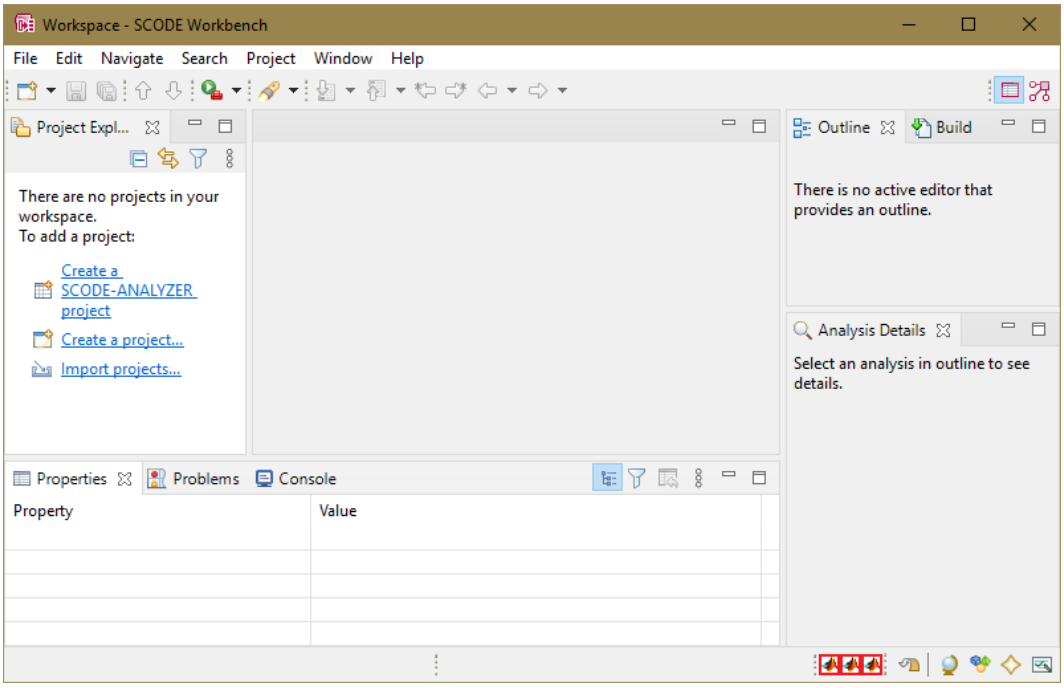


Figure 3. SCODE Workbench (SCODE-ANALYZER perspective) with empty workspace

If you used the SCODE Workbench with SCODE-CONGRA before you started this tutorial, your window will look different than Figure 3. To open the SCODE-ANALYZER perspective, click the **SCODE-ANALYZER** button at the right of the toolbar.

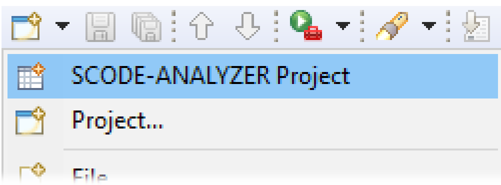
The SCODE-ANALYZER perspective shows the following views:

- top left: Project Explorer
- top middle: reserved for various editors
- top right: "Outline" view and "Build" view
- bottom left: "Problems" view, "Properties" view, Execution Environment, "Console" view
- bottom right: "Analysis Details" view

You can now create a project for the tutorial.

To create a SCODE-ANALYZER project

1. In the SCODE Workbench window, do one of the following:
    - Select **File > New > SCODE-ANALYZER Project**.
    - Click the arrow next to the **New** button and select **SCODE-ANALYZER Project**.

    The "SCODE-ANALYZER project" window opens.

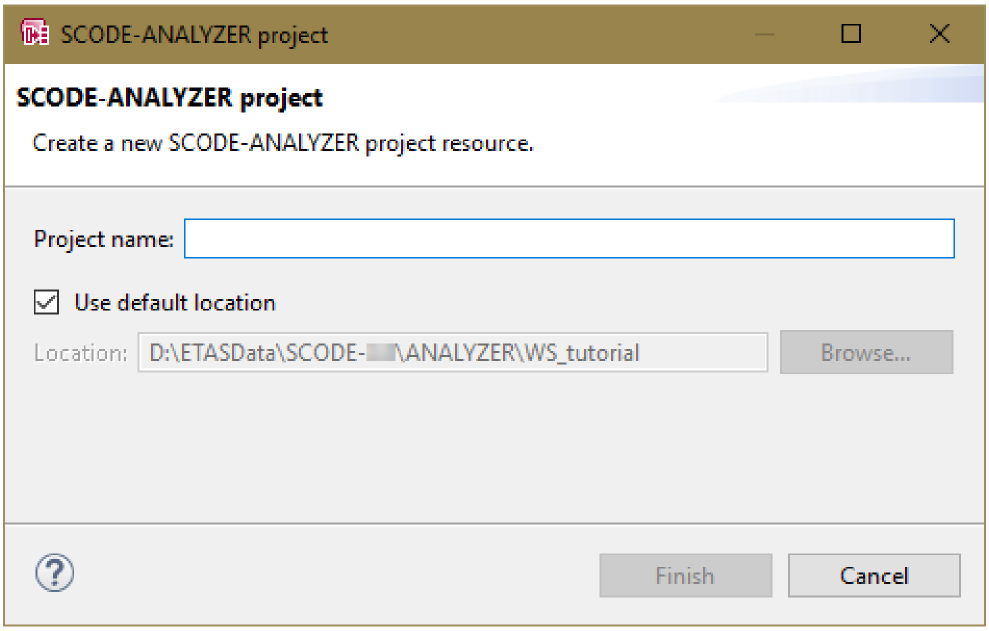


Figure 4. "SCODE-ANALYZER project" window

2. Enter a project name, e.g., `hybridCar`.

   It is recommended that you use the default location for this tutorial.

3. Click **Finish**.

   The project is created, together with some default elements. The "Problem Space" page is shown in the SCODE Workbench window.

4. Expand the tree in the Project Explorer.

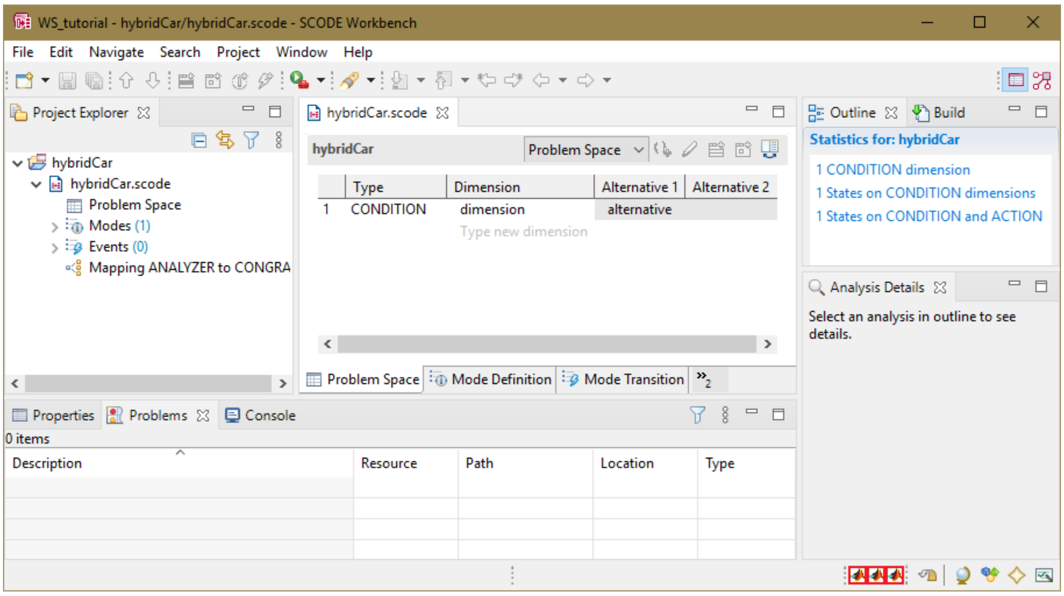


Figure 5. SCODE Workbench window with newly created SCODE-ANALYZER project

4.3. Lesson 2: Defining the Problem Space

In this lesson, you will define the problem space of the system.

This is usually carried out as a structured discussion between domain experts and SCODE analysts. The domain experts provide information about the system context, requirements and system know-how. The SCODE analysts provide the competence for the method & tooling. The analysis defines the problem space — also called condition and action space — by a Zwicky box in terms of

- *Dimensions* — Conditions (inputs) and actions (outputs): aspects of the system or its context that cause or represent different system behaviors (or cause-effect chains) In the hybrid car example, one condition is the state of charge of the battery, or battery SOC for short.
- *Alternatives* — possible values or value ranges of a dimension Alternatives for the battery SOC condition would be full (i.e. no further charging possible), empty, and normal.

To determine the dimensions

1. Write down the dimensions of the system, and the alternative values each dimension can have.

Dimension	Alternative
battery SOC	full / empty / normal
...	

NOTE

When you name a dimension and its alternatives, you should rather base the names on the physical meaning than on the current implementation.

When you consider your list complete, you can enter the dimensions in SCODE-ANALYZER. One condition dimension has been created automatically when you created the project; you can add as many dimensions as required.

To edit an existing condition

1. Go to the "Problem Space" page of your project.

   This page contains the Zwicky box.

2. Click in the "Dimension" cell of the existing condition and enter a name.
3. Click in the "Alternative 1" cell of the condition and enter the first alternative.
4. In the "Alternative 2" cell, enter the second alternative.

   A new, empty alternative is added.

5. If required, enter further alternatives.

   You do not have to change the type of the dimension.

For the `battery SOC` condition of the tutorial, the row should look like this:

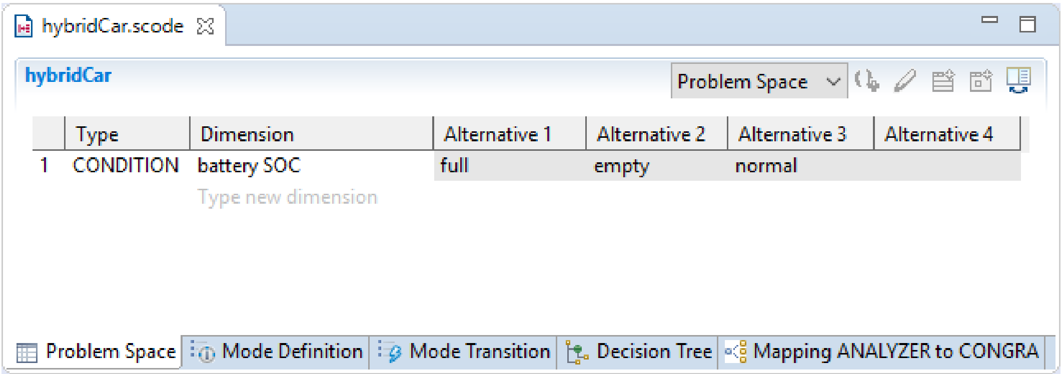
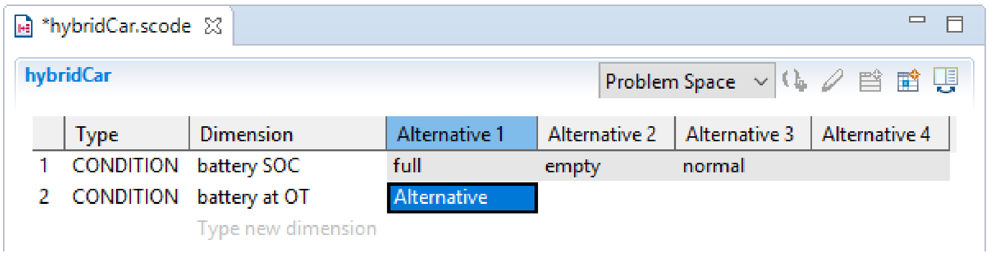


Figure 6. "Problem Space" page with one condition

To add a new condition

1. In the "Problem Space" page, click the text `Type new dimension` below the last dimension.
2. Enter the name of the new dimension.

   The dimension is created. The type `Condition` is assigned automatically, and the first alternative is set to a default value.

3. Enter the alternatives as described in To edit an existing condition.

   Conditions are allowed to have different numbers of alternatives. One condition can have 3 or more alternatives, while another has just 2 alternatives. Extra alternatives are left empty.
4. Add the other conditions you need.

The "Outline" view on the right of the SCODE-ANALYZER window shows the statistic of the problem space.

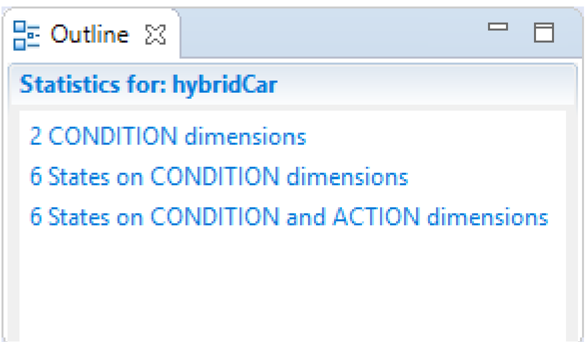


Figure 7. "Outline" view with statistics for the problem space

If desired, you can add a comment to a condition or to a single alternative.

<u>To add a comment to a condition or alternative</u>

1. In the "Problem Space" page, click the condition or alternative you want to comment.
2. Go to the "Properties" view.

   By default, the "Properties" view is displayed at the bottom left of the SCODE Workbench window.
3. Enter your comment.

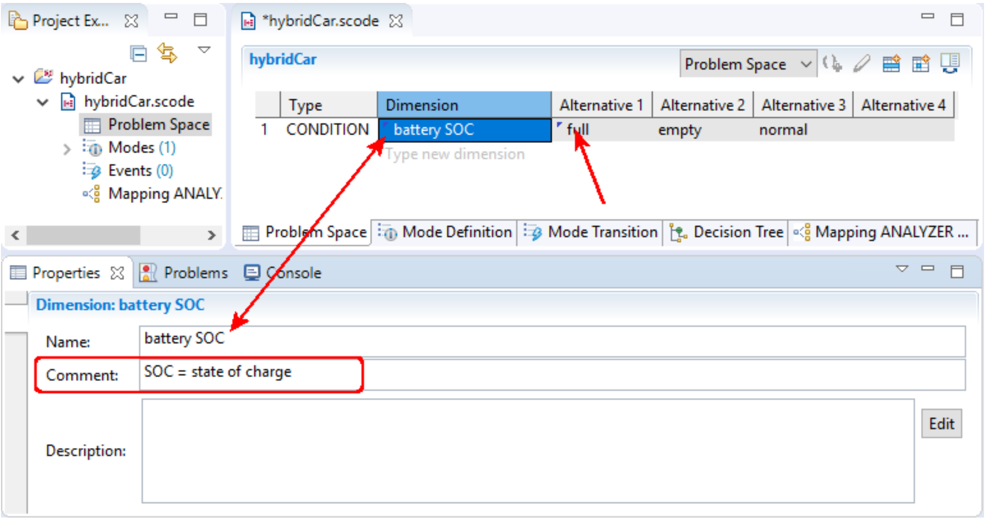
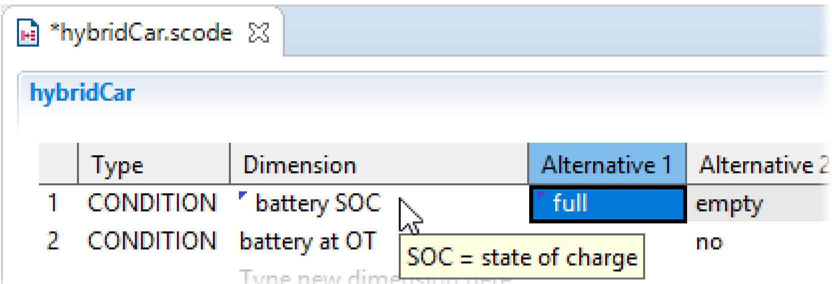


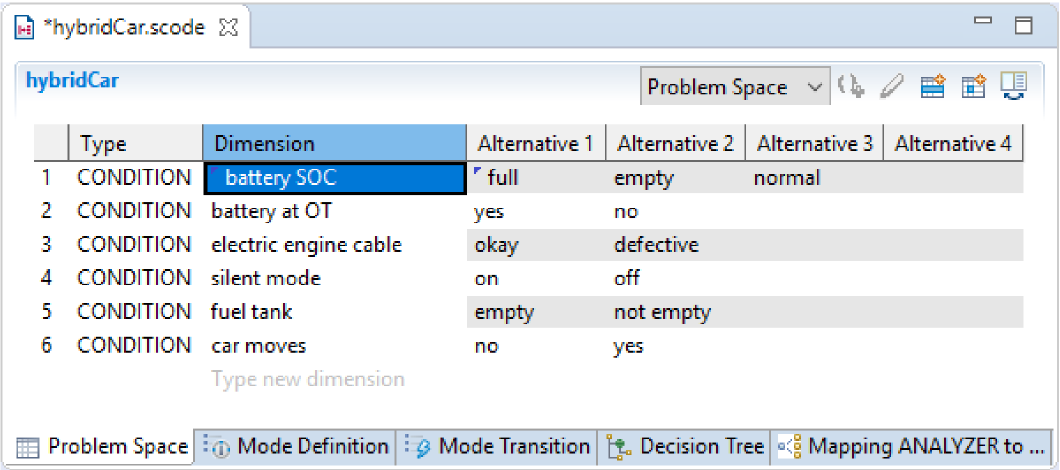
Figure 8. "Properties" view for a dimension selected in the "Problem Space" page

In the "Problem Space" page, the condition or dimension is marked with a triangle in the upper left corner of its table cell.

If the mouse pointer hovers over the cell, the comment appears as tooltip.



Add all conditions you need. When you have entered all conditions, the "Problem Space" page may look like this:



4.4. Lesson 3: Defining Modes

The static analysis of the condition and action space described by the Zwicky box decomposes the condition and action space into multiple non-overlapping subspaces that model partial problems. A partial problem is now characterized by the fact that the context of the system is in a so-called mode, i.e., in a specific situation. In this situation, the system has to behave in a specific way, i.e., the system resides also in a mode corresponding to this situation. Thus, a mode can also be understood as a so-called situation module.

Modes that are relevant for the problem solution and model the corresponding system are also called normal modes or *system modes*. In SCODE-ANALYZER, system modes are marked by this icon:

Impossible or meaningless combinations of conditions, and combinations that are possible by nature, but ruled out by design, are stored in so-called *non-system modes*. In SCODE-ANALYZER, non-system modes are marked by this icon:

NOTE

It is strongly recommended that you use system modes for combinations that are possible, but ruled out by design. This is especially important for safety-critical systems.

4.4.1. Creating and Editing Modes

The next thing to do is to define the modes of the system.

Again, this is usually carried out as a structured discussion between domain experts and SCODE analysts.

In the hybrid car example, one mode is the situation that the car is *charging*.

To determine the modes

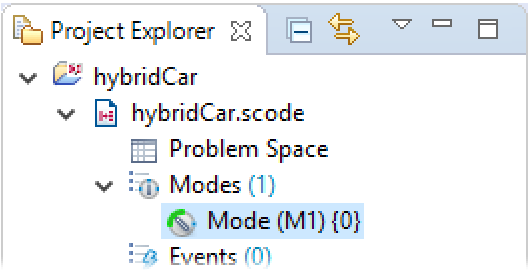
NOTE

The modes must not overlap. If they do, SCODE-ANALYZER will issue an error.

1. Write down the modes of the system and the states of the conditions for each mode.

Mode	Conditions		
	battery SOC	**battery at OT**	**others**
charging	empty or normal	yes	
...			

When you consider your list complete, you can define the modes in SCODE-ANALYZER. One mode has been created automatically when you created the project; you can add as many modes as required.



Before you start adding and editing modes in SCODE-ANALYZER, read the following list of requirements:

A System modes must not overlap.

If they do, an error message is shown in the "Outline" view (see Figure 10).

B You can select no, one, several, or all alternatives of a condition.

If a condition is irrelevant for the current mode, you can select none or all of its alternatives. Such a condition is sometimes called a *don't care dimension*.

NOTE

If you add a new alternative to the condition, that alternative is, by default, not selected in any rule. This means that the condition loses its *don't care* property if you selected **all** old alternatives.

It is therefore recommended that you select no alternative for *don't care* dimensions.

C If you select two or more alternatives of one condition, the alternatives are ORed:

```
alternative_1 OR alternative_2
```

D If one alternative of a condition is forbidden for the mode, you can either select the forbidden alternative and activate the option in the "NOT" column, or you can select all conditions except the forbidden one.

Type	Dimension	NOT	Alternative 1	Alternative 2	Alternative 3
CONDITION	battery SOC	☑	full	empty	normal

is equivalent to

Type	Dimension	NOT	Alternative 1	Alternative 2	Alternative 3
CONDITION	battery SOC	☐	full	empty	normal

NOTE

If you add a new alternative to the condition, that alternative is not selected in any rule. This means that the two possibilities are no longer equivalent.

The first possibility allows all alternatives except the forbidden one, i.e. *the new alternative is allowed*.

The second possibility allows only those alternatives that are explicitly selected, i.e. *the new alternative is forbidden*.

E The setting in the "NOT" column of a condition applies to all selected alternatives.

If you select several alternatives and activate the option in the "NOT" column, the rule for this condition is

```
NOT(alternative_1 OR alternative_2)
```

F Alternatives from different conditions are ANDed:

```
battery SOC = empty AND battery at OT = no
```

G You can specify one or more rules for one mode. Each mode must have at least one rule; otherwise, an error is issued.

A state belongs to a mode if it matches one of the mode's rules.

Table 2. Requirements for modes and mode definition rules

To edit an existing mode

1. In the Project Explorer, do one of the following:
   - Double-click the existing mode.
   - Right-click the existing mode and select **Open in Editor** from the context menu.

   The editor for the mode opens in the "Mode Definition" page. The "Mode" field allows renaming the mode and switching from system mode to non-system mode or back. The "Rule Editor" field is used to select, for each condition, those alternatives that define the mode.

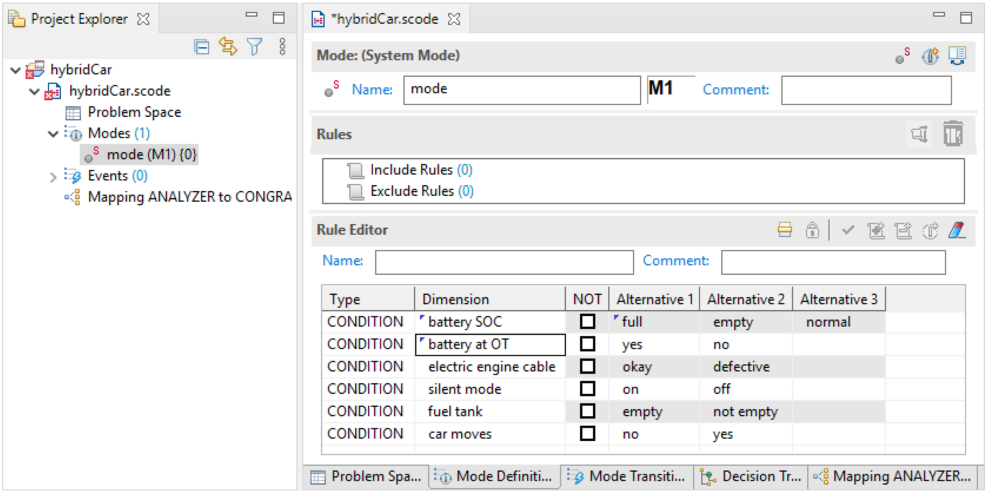
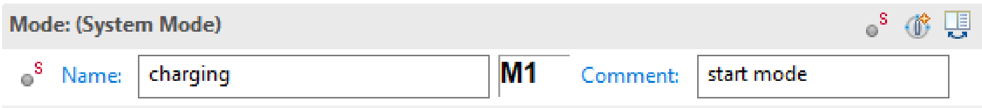


Figure 9. "Mode Definition" page with mode editor

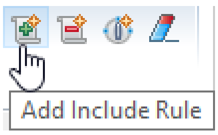
2. In the "Name" field of the "Mode" pane, enter a meaningful name.
3. If desired, enter a comment in the "Comment" field.



4. In the "Rule Editor" field, click in the cells of all alternatives that define the mode.

   Keep in mind the requirements listed in Table 2.

5. When you have selected all relevant alternatives, click the **Add Include Rule** button.

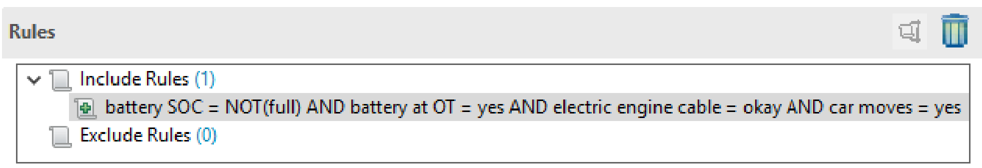


NOTE

Include rules and exclude rules are both valid, but exclude rules tend to be more difficult to understand.

It is therefore strongly recommended that you use only include rules.

The rule is added to the "Include Rules" list in the "Rules" field.



Since the mode is the only one in the project, it is marked as start mode ($\circ^S$).

You can enter a name and a comment for the rule in the "Properties" view.

The "Outline" view shows the statistics for the mode definition. Red font indicates errors.

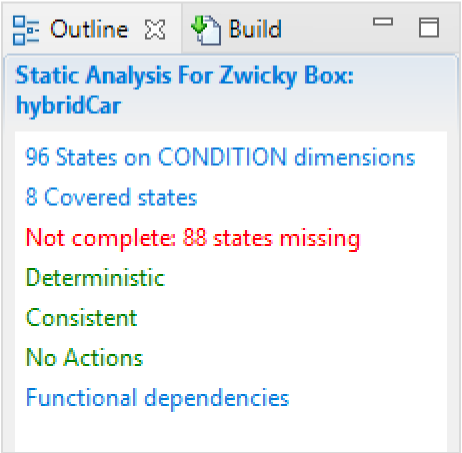
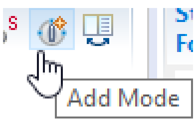


Figure 10. "Outline" view with statistics for the mode definition

To add a new mode

1. To add a new empty mode, do one of the following:
   - In the "Mode Definition" page, "Mode" field, click the **Add Mode** button.

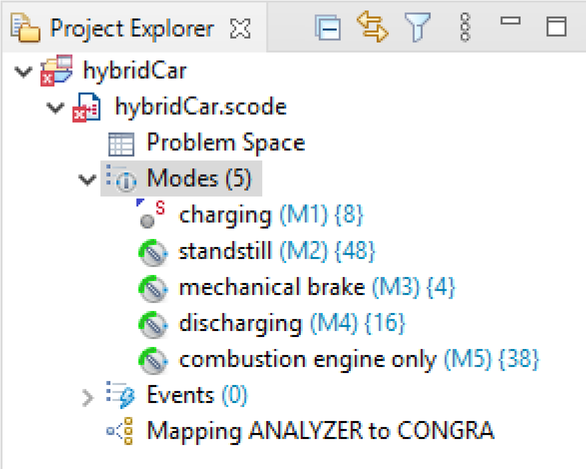
   - In the Project Explorer, right-click the `Modes` node and select **Add Mode** from the context menu.

   The mode is created as a system mode. It is added to the `Modes` node in the Project Explorer, and it is opened in the mode editor in the "Mode Definition" page.
2. To add a non-empty mode, proceed as follows:
   i. In the "Rule Editor" field, click in the cells of all alternatives that define the mode.
   ii. Click the **Add Mode from Rule** button.

   The mode is added, together with the rule you specified.
3. Edit the mode as described in To edit an existing mode.

Add and edit all modes you need. [2] When you have entered all conditions, the `Modes` folder in the Project Explorer may look like this:



The content of the "Mode Definition" page depends on your selection in the Project Explorer.

4.4.2. Checking Modes

While you are adding and editing modes, the modes are analyzed for completeness, determinism, and consistency. The "Outline" view shows an overview of the results. You cannot generate code until all errors are corrected.

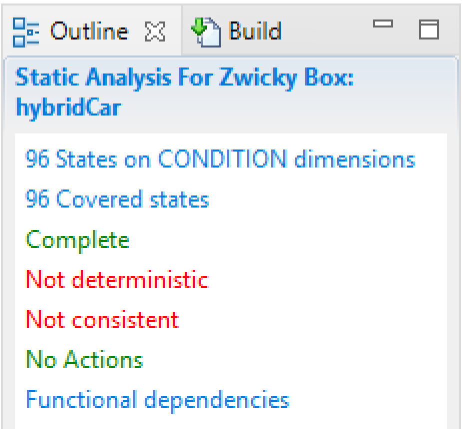


Figure 11. "Outline" view with statistic analysis for the modes and rules in Table 25

Red lines indicate errors. If you click a red line, detailed information appears in the "Analysis Details" view.

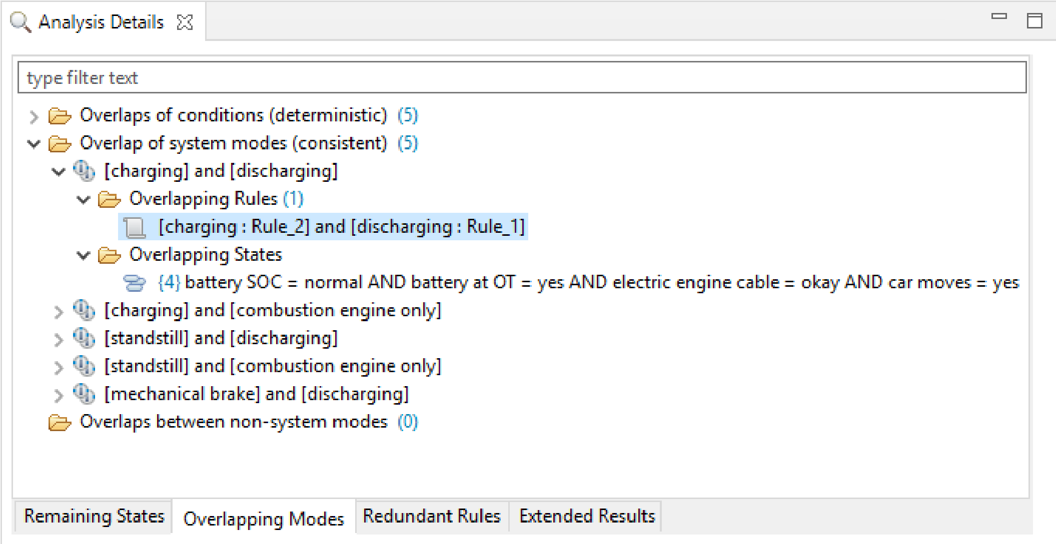


Figure 12. "Analysis Details" view for the modes and rules in Table 25

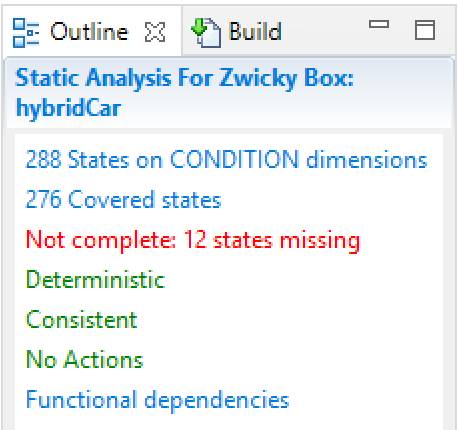
You can display the decision tree (see section 4.4.4) for an easy review.

You have to remove inconsistent and/or non-deterministic settings before you can generate code. There are two ways to remove the errors: You can try to re-define the rules, or you can check if your list of conditions is really complete, and add a condition, if necessary.

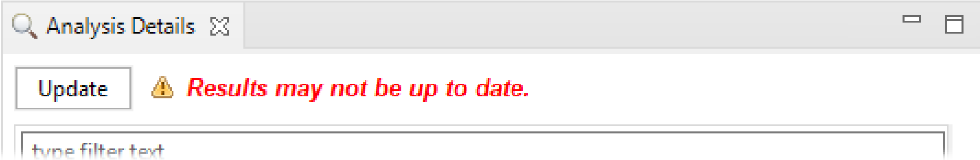
To start removing the errors

1. For this tutorial, assume that a condition `desired acceleration` is missing.
2. Add the condition with suitable alternatives. [3]
3. Assign appropriate alternatives to the mode definition rules. [4]

   If you determined and used the desired acceleration as shown in Table 26, the "Outline" view looks as follows.



   The number of states has increased, due to the new condition. The system is now both deterministic and consistent, but some states are missing. In the "Analysis Details" view, the results are marked as possibly outdated, and an **Update** button appears.



4. Click **Update**.

   The "Analysis Details" view suggests rules for the missing states.

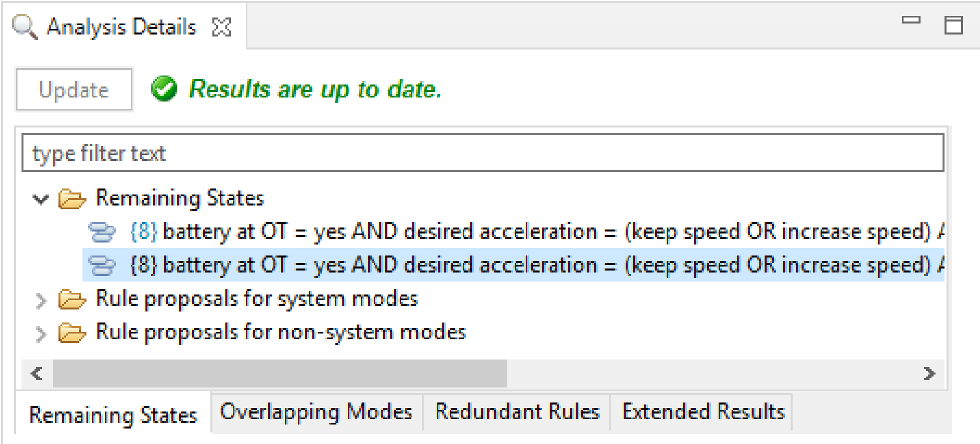
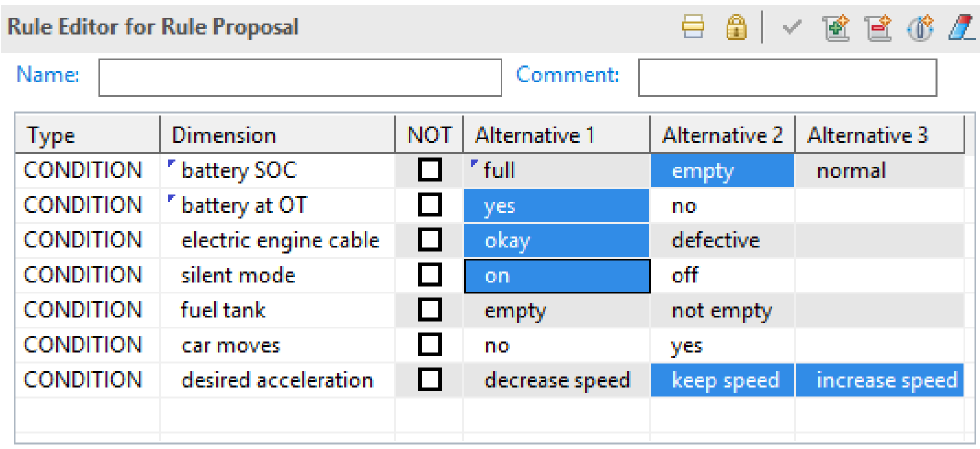


Figure 13. "Analysis Details" view with suggested rules for missing states

5. Double-click a suggested rule to display it in the "Rules Editor" field.



6. Check if the rule is physically possible. [5]
7. Repeat the last two steps for the other suggestion(s).

4.4.3. Inserting a Non-System Mode

Impossible or meaningless combinations of conditions, and combinations that are possible by nature, but ruled out by design, are stored in *non-system modes*.

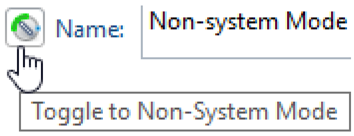
To start removing errors

1. In the "Analysis Details" window, double-click a suggestion to display the rule in the "Rule Editor" field.

2. In the "Rule Editor" field, click the **Add Mode from Rule** button.

The mode is added (as a system mode), together with the suggested rule.

3. Enter a name for the mode.
4. Click the **Toggle to Non-System Mode** button.



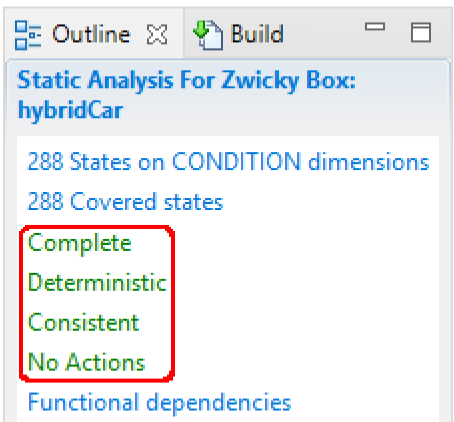
With that, the mode becomes a non-system mode. The mode icon in the Project Explorer changes from to .

The static analysis in the "Outline" view is updated automatically. The "Analysis Details" view is not updated automatically, it is just marked as possibly outdated.

To add the remaining states

1. Update the "Analysis Detail" view.
2. Display the remaining suggestion in the "Rules Editor" field.
3. Check if the rule is physically possible. [6]
4. Do one of the following:
   - If the rule is not possible, add it to the non-system mode.
   - If the rule is possible, add it to a system mode.

   With that, the system is complete, deterministic, and consistent.



4.4.4. Viewing the Decision Tree

The mode definition rules can be visualized in a so-called *decision tree*, which is displayed in the "Decision Tree" page. This decision tree can be used to check modes and rules, and it is easier to read than the Zwicky box and the mode list.

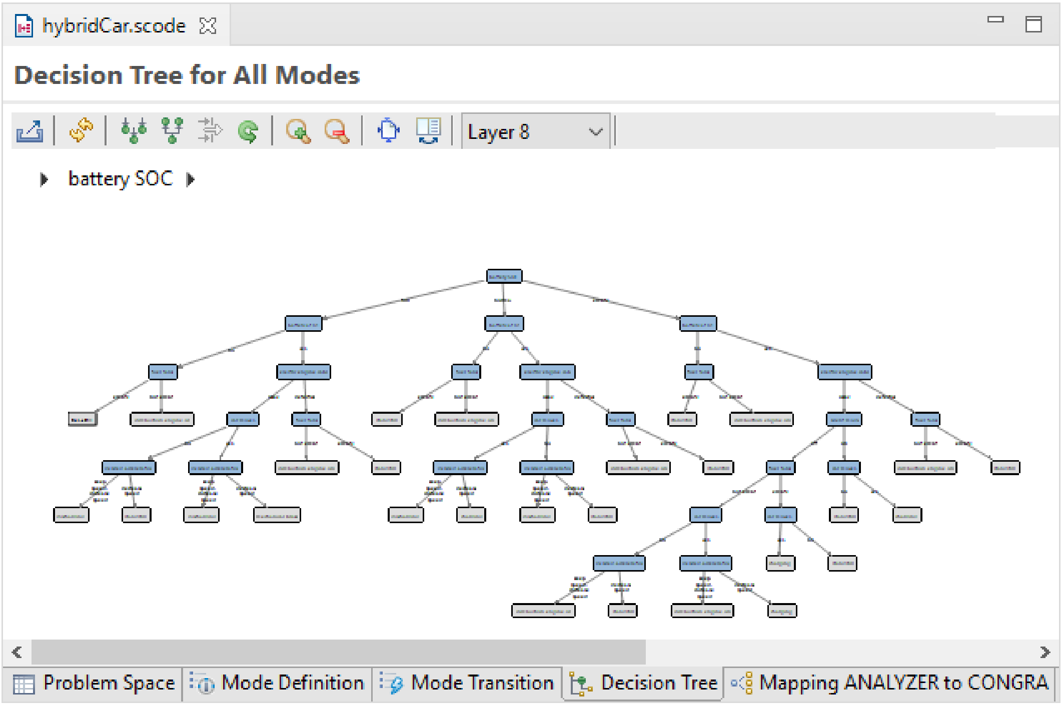


Figure 14. "Decision Tree" page

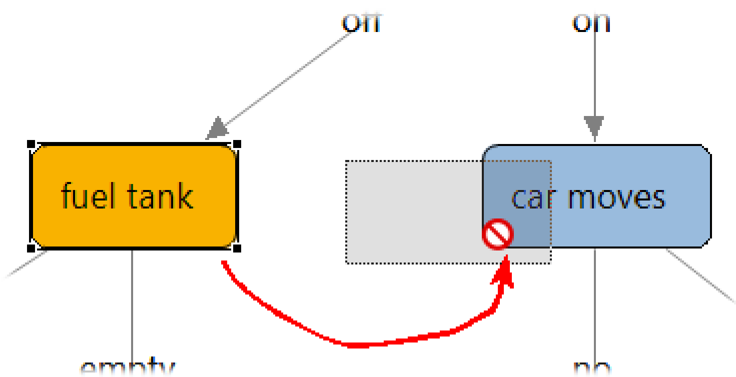
In the decision tree, the conditions are displayed as named blue boxes, normal modes are displayed as named grey boxes. Non-system modes are hidden by default, but you can display them, if desired. They appear as unnamed grey boxes. The alternatives of a condition are displayed as named arrows pointing to another condition or a mode.

By default, the conditions appear in roughly the same order as in the Zwicky box. However, you have several possibilities to change the look of the decision tree.

To change the view of the decision tree

1. Drag any condition and drop it onto another.

   In some cases, dropping a condition onto another is forbidden. A prohibition icon is shown in these cases.

   If dropping is permitted, the dragged condition takes the place of the condition it is dropped onto. The decision tree is re-arranged so that it still covers all decisions.

   Figure 105 shows a re-arranged decision tree of the tutorial project.

2. Click the **Toggle orientation** button to change the direction of the decision tree.

   Figure 106 shows the decision tree with horizontal orientation.

3. Click the **Toggle view between tree and dag** button to switch from tree view mode to directed acyclic graph (DAG) view mode or back.

   Figure 107 and Figure 108 show examples for the DAG view mode.

4. Open the "Layer" combo box and select the number of tree levels you want to display.

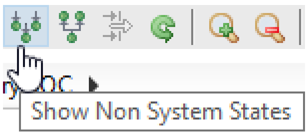
NOTE	You should do this only in the tree view mode. The DAG view mode is suitable only for complete trees.

   Only the selected number of levels is shown. Figure 109 shows an example.

5. Use the **Zoom in** and **Zoom out** buttons to zoom the decision tree.

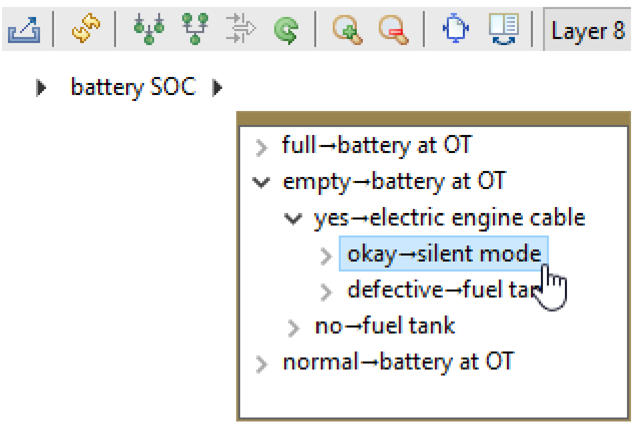
   The **Fit to page** button scales the decision tree to the current size of the "Decision Tree" page.

6. Click the **Show non-system states** button to display the non-system modes.



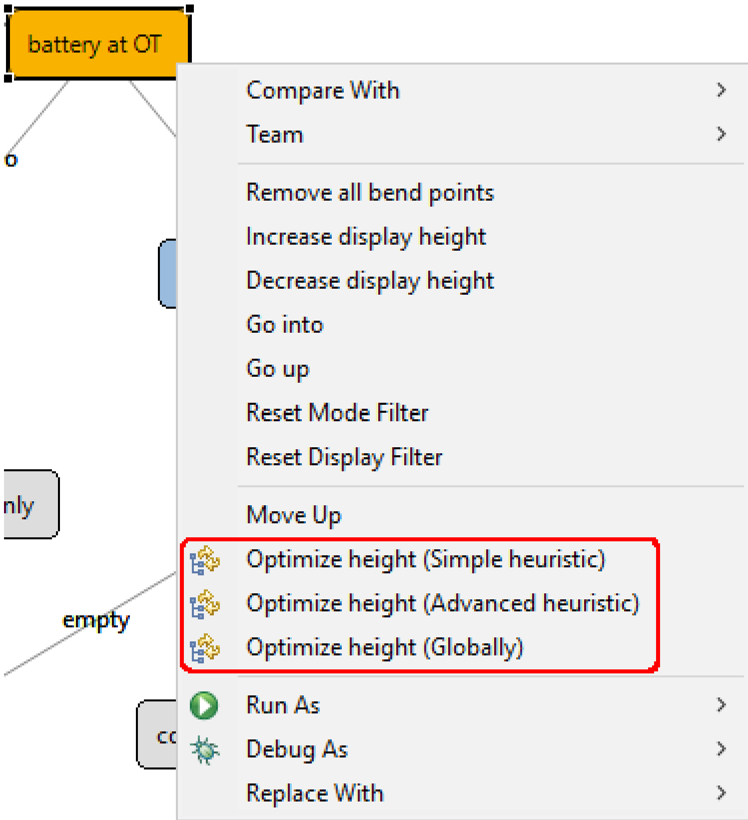
   See Figure 110 for an example.

7. To select a sub-tree, click the triangle to the right of the top node name and select one of the alternatives.



   Only the selected subtree is displayed. See Figure 110 for an example.

8. To optimize the height of the decision tree, right-click a node at or near the top of the tree and select an **Optimize height** * entry from the context menu.



The nodes of the tree are re-arranged so that the height of the tree is minimized. See Figure 111 and Figure 112 for examples.

4.5. Lesson 4: Code Generation from Mode Invariants

The purpose of code generation is to transform this model into executable code that reflects the same functionality as the model.

As soon as your model is complete, deterministic, and consistent, you can generate code, even though transitions are still missing. In this case, the source for code generation is based on the mode definition only; it is named *Mode Invariants*.

SCODE-ANALYZER offers the following setup possibilities:

- for the entire workspace (accessible via **Window > Preferences**)
- for a particular project (accessible via a project's context menu or via **Project > Properties**)

Project-specific settings override workspace settings. In this tutorial, you will use workspace settings.

To prepare code generation from mode invariants

1. Select **Window > Preferences**.
2. In the "Preferences" window, open the "SCODE-ANALYZER\Generator" node.
3. In that node, do the following:
    i. Select one or more generators.

ii. For the "Generation Source" property, select `Mode Invariants`.

4. Click **Apply and Close**.

   The settings should look like those in Figure 15.

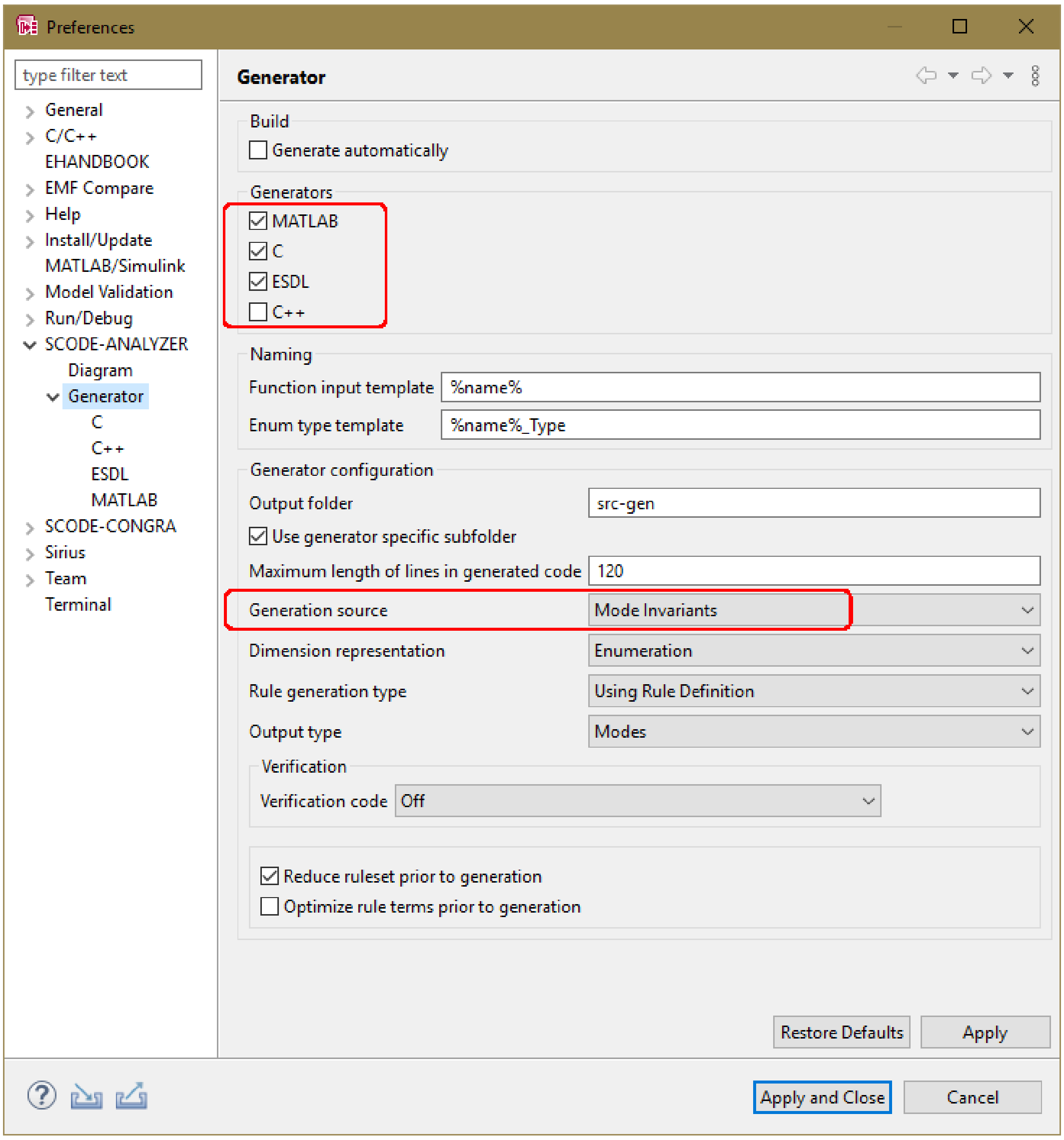


Figure 15. "Preferences" window with settings for code generation from mode invariants

With that, you can generate the code.

To generate code from mode invariants

1. In the Project Explorer, right-click the SCODE file and select **Generate Code** from the context menu.

   Code is generated for the selected generators (ESDL, C code, C++ code, or MATLAB). The resulting files are stored in the SCODE-ANALYZER project; the output folder is named `src-gen` by default.

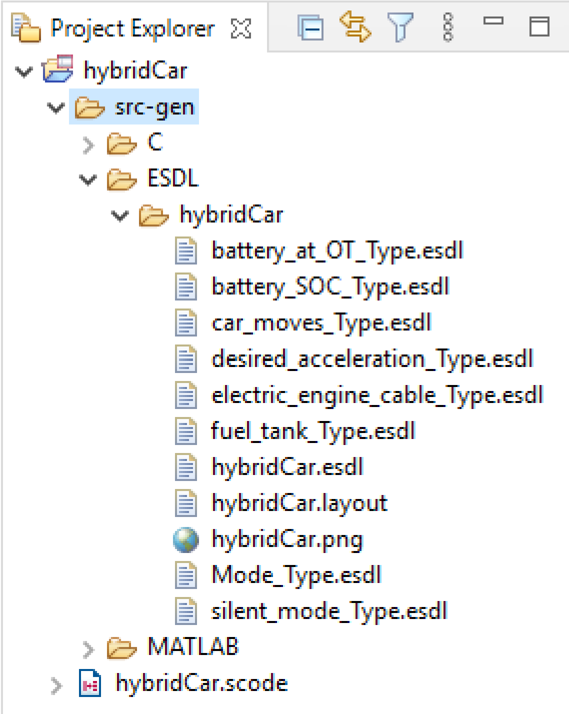


Figure 16. Output folder for code generation

2. Open the `hybridCar.*` file(s) and look at the code.

   It is recommended that you keep a copy of this file for later use.

3. If desired, open other generated files and inspect the code.

For code generation from mode invariants, the output folder contains the following items:

- folder *<generator_name>* (e.g., `ESDL` or `C`)

  NOTE

  Only created when you activated the code generation option **Use generator specific subfolder** (see Figure 15).

  Contains all files generated for the respective generator.

  - folder *<project_name>* (e.g., `hybridCar`)

    Contains the files that define conditions and modes and the file that determines the current mode.

    For the ESDL generator, the files are named as follows:

    - *<condition_name>*`_Type.esdl` (define the conditions; one file per condition)
    - *<project_name>*`.esdl` (determines the current mode)
    - *<project_name>*`.*` (required if you want to use the generated ESDL code in ASCET-DEVELOPER)
    - `Mode_Types.esdl` (defines the modes)

For the MATLAB generator, the files are named as follows:

- `<condition_name>_Type.m` (define the conditions; one file per condition)
- `<project_name>_ModeSelector.m` (determines the current mode)
- `Mode_Types.m` (defines the modes)

For the C and C++ generators, the files are named as follows:

- `<project_name>_Types.h` or `<project_name>_Types.hpp` (defines all conditions and modes)
- `<project_name>.c` or `<project_name>.cpp` (determines the current mode)
- `<project_name>.h` or `<project_name>.hpp`

4.6. Lesson 5: Defining Events and Transitions

The dynamic analysis step of SCODE-ANALYZER allows to specify which situation changes in the context are possible and how the system shall react to them in terms of transitions between modes. This is done in terms of a mode transition table that defines which event triggers a transition from a source mode to a target mode. It is also possible to define that there is no transition possible from a specific source mode to a target mode.

Just like modes, events are defined based on rules on sets of alternatives for each condition dimension in the underlying Zwicky box. This enables the tool-based analysis of properties such as liveliness, stability and determinism:

- *Liveliness* of a transition means that the corresponding event is able to really trigger the transition to the target mode (i.e. the event conditions are not already fulfilled by the rules of the source mode).
- *Stability* of a transition means that the corresponding event is compatible with the target mode (i.e. the event conditions are fulfilled by the rules of the target mode). Otherwise, an immediate further mode change would be the consequence and the system would "oscillate".
- *Determinism* (or consistency) means that the events of all outgoing transitions of one mode do not overlap, i.e. that there is always only one transition possible and the target mode is uniquely defined.

4.6.1. Creating and Editing Events and Transitions from One Mode

The next thing to do is to define the transitions from one mode, e.g., charging, to the other modes, as well as the events that trigger the transitions. Again, this is usually carried out as a structured discussion between domain experts and SCODE analysts.

An event is a set of one or more *mode transition rules* that must be fulfilled. Mode transition rules are very similar to mode definition rules, except that mode transition rules are always include rules. The requirements for modes and mode definition rules listed in Table 2 are valid for events and mode transition rules, too.

In the hybrid car example, the transition from `charging` to `discharging` can occur, for example, if the battery is at operating temperature and fully loaded, the cable is okay, the car is moving, and the driver wants to keep or increase the speed.

To determine events

1. Decide which transitions are allowed and which are forbidden.
2. Decide what event triggers which allowed transition.

current mode	next mode				
	charging	discharging	standstill	combustion engine only	mechanical brake
charging	*	battery at operating temperature and fully charged AND electric engine cable okay AND car moves AND desired acceleration is keep or increase speed			
discharging		*			
standstill			*		
combustion engine only				*	
mechanical brake					*

When you consider your list complete, you can define the events and transitions in SCODE-ANALYZER.

In SCODE-ANALYZER, events and transitions are specified on the "Mode Transition" page. That page has two views, the "Event Overview and Implementation" view and the "Mode Transition" view. A button at the top right of the "Mode Transition" page (marked red in Figure 17) is used to toggle the views.

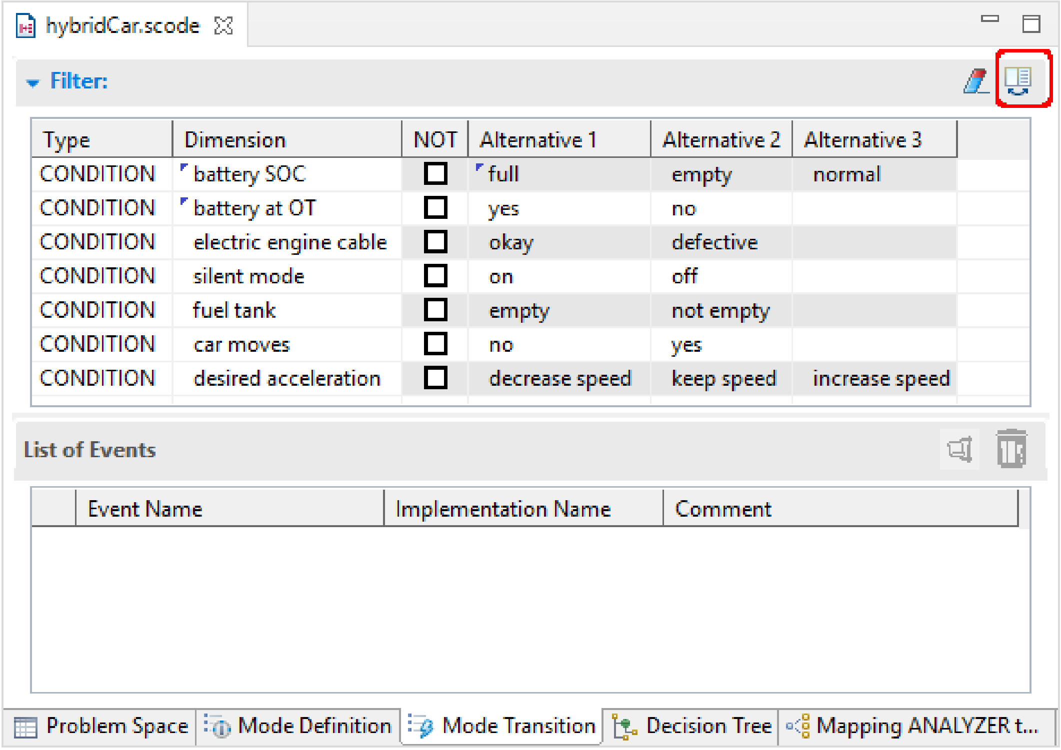


Figure 17. "Mode Transition" page with "Event Overview and Implementation" view

By default, SCODE-ANALYZER assigns events to transitions based on the definition of the respective target mode. In this tutorial, you will deactivate this default behavior and define all transitions manually.

To set transition behavior

1. Open the "Preferences" window and go to the SCODE-ANALYZER node.
2. In that node, set the default transition behavior to `non-transition`.

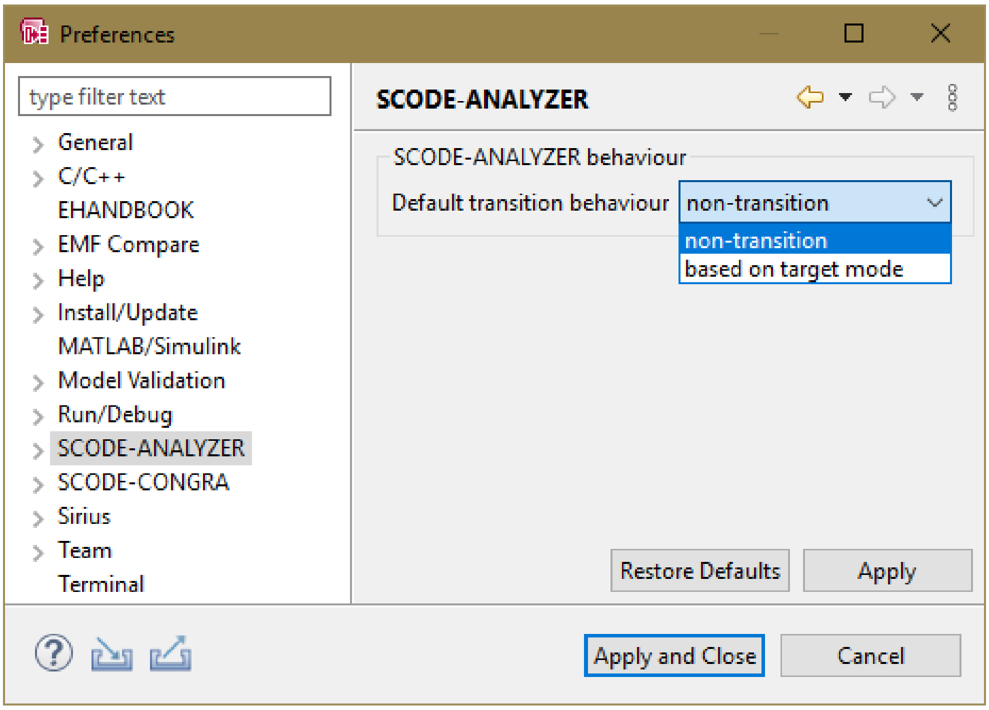


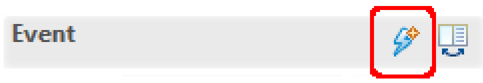
Figure 18. "Preferences" window, "SCODE-ANALYZER" node

3. Click **Apply and Close**.

The "Mode Transition" view (see Figure 19) is used to specify transitions and events. There are several ways to create and specify transitions and events:

A. To add an empty event, you can do one of the following:

   - Right-click the Events folder in the Project Explorer and select the **Add Event** context menu option.
   - Use the **Add Event** button in the event viewer (b in Figure 19).



These events are then assigned to transitions in the transition matrix (a in Figure 19) and specified via the rule editor (c in Figure 19).

B. To add a non-empty event, you can specify a rule in the rule editor and then use the **Add Event from Rule** button to add an event with the specified rule.

This event is then assigned to a transition in the transition matrix and refined.

The last way is used in this tutorial.

4.6.1.1. First Transition

<u>To add an event from a rule</u>

1. Go to the "Mode Transition" view of the "Mode Transition" page.

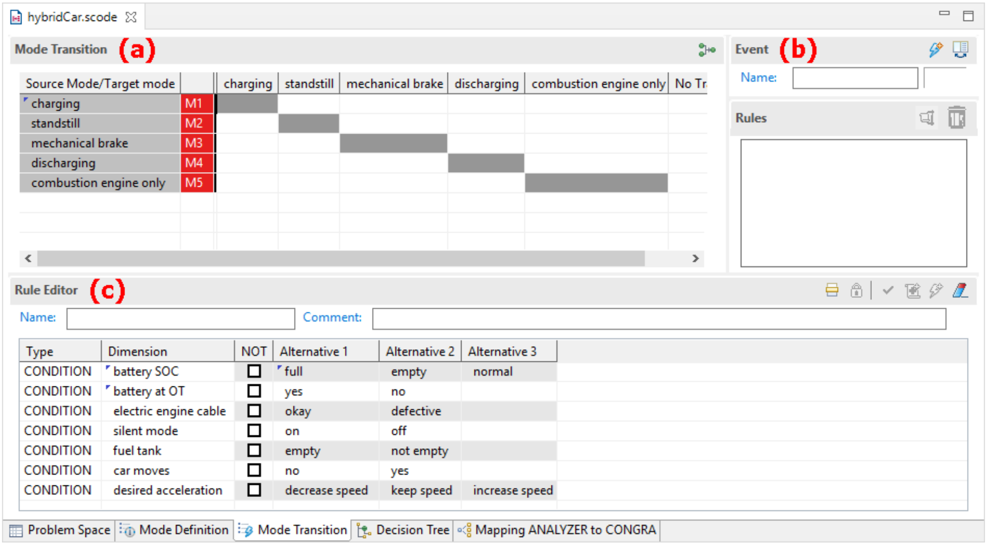
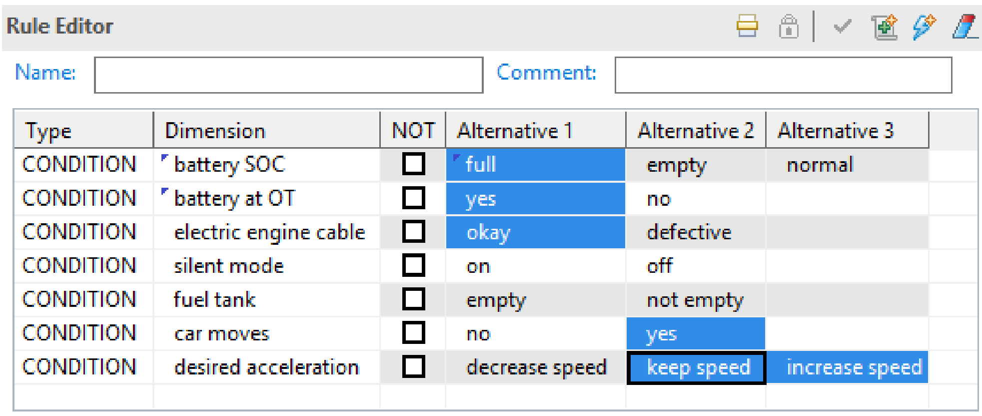


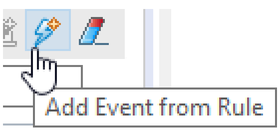
Figure 19. "Mode Transition" page with "Mode Transition" view

The upper part of the window contains a transition matrix (a) and an event viewer (b). The lower part contains a rule editor (c) similar to the one used in section 4.4.1, "Creating and Editing Modes".

2. In the rule editor, specify a rule for a transition.



3. If desired, enter a name and/or a comment for the rule.
4. When you have selected all relevant alternatives, click the **Add Event from Rule** button.



An event is added, together with the rule you specified. The event is shown in the event viewer, with default name and short name.

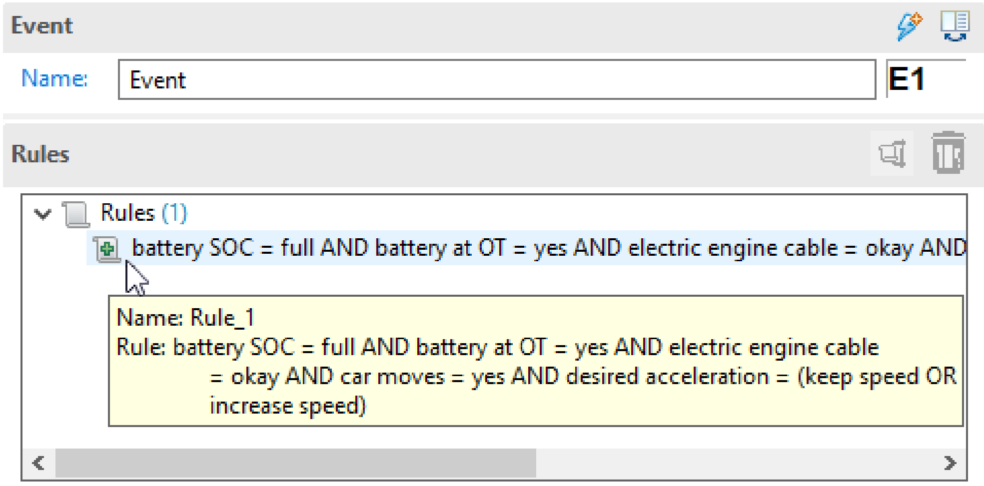


Figure 20. Event viewer with new event

5. In the "Name" field, enter a meaningful name for the event that shows the purpose of the event.

   This tutorial uses the names of source mode and target mode as event name. The event in Figure 20 is named `charging_discharging`.

   You cannot change the short name.

6. If desired, enter a comment for the event in the "Properties" view.

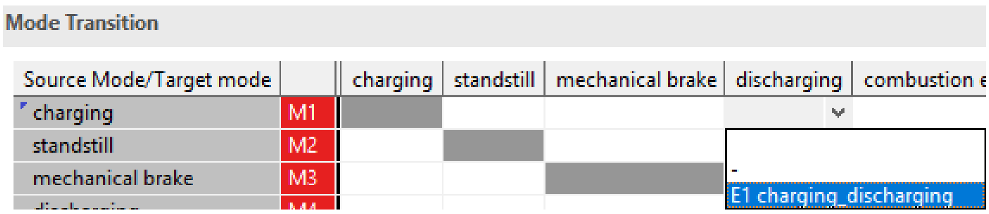
The event is created, but not yet assigned to a transition.

To assign an event to a transition

1. In the transition matrix, double-click in the cell of a transition.

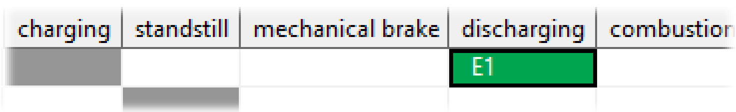
   The event in Figure 18 shall be assigned to the transition from `charging` to `discharging`.

   A combo box opens that offers all existing events for selection. In addition, you can select an empty row to remove an event assignment, and you can select — to mark the transition as forbidden.



2. Select the event that you want to assign to the transition.

   The event's short name appears in the cell. If the event is valid, the cell background becomes green.



Each row in the transition matrix contains transitions from one mode.

A. The transitions from one mode must fulfill the following requirements:
   a. All states that lead away from the source mode must be covered by the transitions from that mode.

      The number of states that must be covered, and of states that are covered, are given in the "Outline" tab.

   b. The events must not overlap.

B. A single transition from the source mode must fulfill the following requirements:
   a. All states of the source mode must react to the event.
   b. The event must be fully enclosed in the target mode.

      This rule is violated if at least one state in one of the rules is not part of the target mode.

   c. The event must not overlap with a non-system mode.

      This rule is violated if at least one state in one of the rules is part of a non-system mode.

   d. The event must not overlap with dynamic non-transitions.
   e. The rules for the transition must not overlap with the source mode.

      This rule is violated if at least one state in one of the rules is part of the source mode.

The "Outline" view shows the statistics for the source mode, as well as for a selected transition. The selected transition is okay, but most states are not covered.

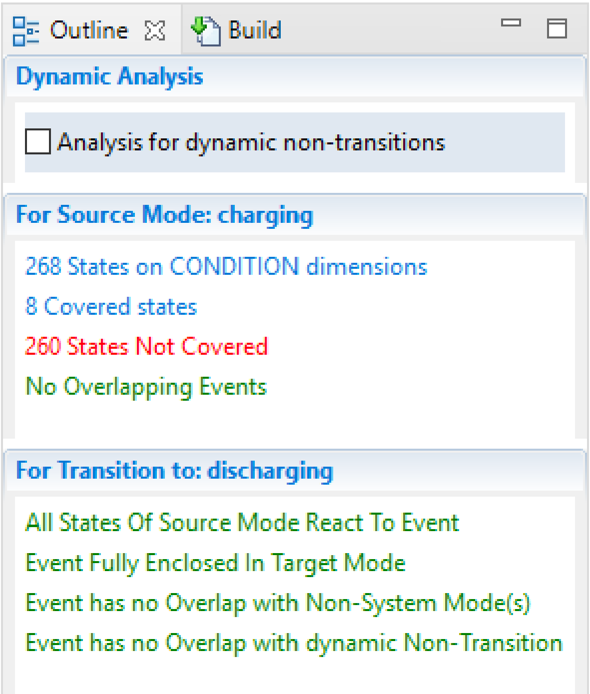


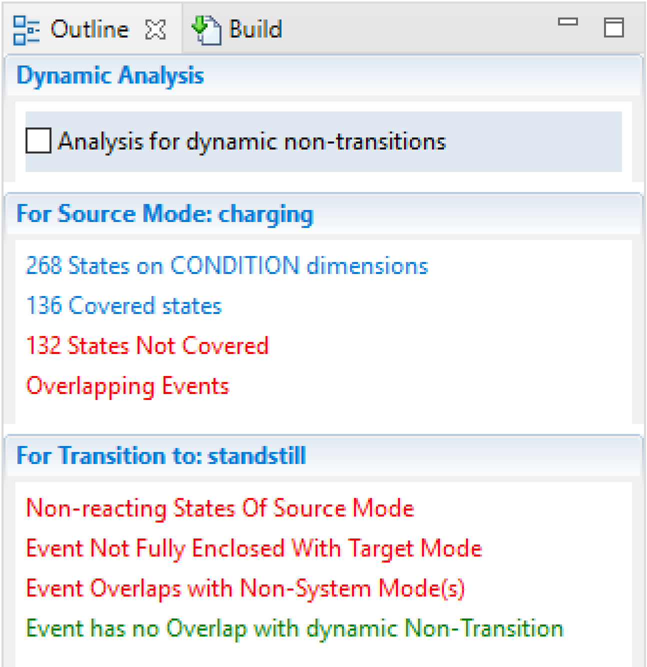
Figure 21. "Outline" view with statistics for mode transitions

4.6.1.2. Second Transition

To set up and assign another event

1. Create another event with the rule `fuel tank = empty`.
2. Assign the event to the transition from `charging` to `standstill`.

The "Outline" view shows several errors (red lines).



Error message	Violated requirement
x States not covered	rule A.a
Overlapping Events	rule A.b
Non-reacting States of Source Mode	rule B.a
Event Not Fully Enclosed With Target Mode	rule B.b
Event Overlaps with Non-System Mode(s)	rule B.c

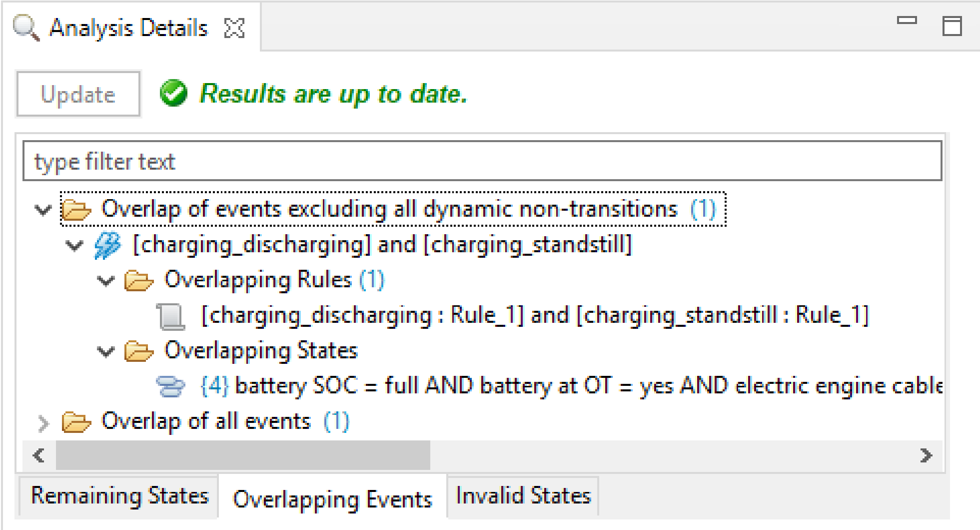
To check and remove the errors

1. Click the line `Overlapping Events`.

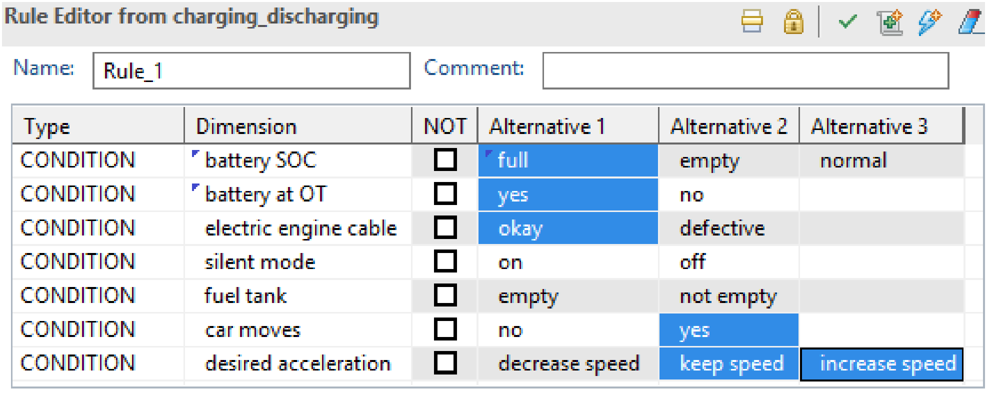
   The "Analysis Details" view displays the "Overlapping Events" tab.

   Since the tutorial does not (yet) use dynamic non-transitions, the content of both folders is identical.

2. Expand the first folder and all of its children.



3. To display one of the overlapping rules, right-click an entry in the `Overlapping Rules` folder and select **Open in Editor > *<rule_name>* of *<mode_name>*** from the context menu.



4. Refine one or both overlapping rules to remove the overlap.

   In this example, the overlap can be removed, e.g., if the rule for the second event is changed to `fuel tank = empty AND electric engine cable = defective`.

5. Update the display in the "Analysis Details" view.
6. If there are more errors, repeat this procedure to solve these, too.

4.6.1.3. Remaining Transitions

Once all errors for existing transitions are removed, only the `x States Not Covered` line remains red. If you click that line, detailed information appears in the "Analysis Details" view. Use that information to specify the other transitions from the charging mode.

To specify transitions with the "Analysis Details" view

1. In the "Outline" view, click the red line with `x States not covered`.

   The "Analysis Details" view offers suggestions for further rules.

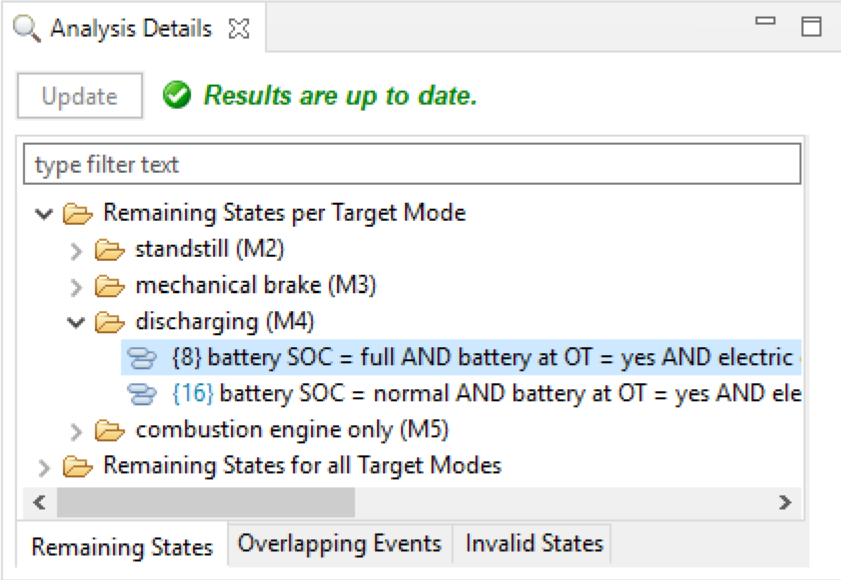
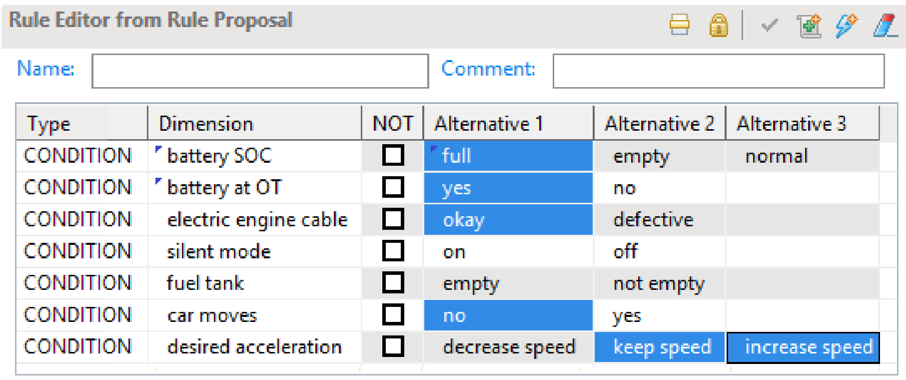


Figure 22. "Analysis Details" view with suggested rules for transitions

2. To complete the transition from charging to discharging, do the following: [8]

   i. Double-click a suggestion in the `Remaining States per Target Mode\discharging *` folder to display it in the "Rules Editor" field.



   ii. Click the **Add Include Rule** button to add this rule to the `charging_discharging` event.

   The statistics in the "Outline" view are updated automatically, the suggestions in the "Analysis Details" view are not.

   iii. In the "Analysis Details" view, click **Update** for new suggestions.

   iv. Repeat these steps for the remaining suggestions in the `Remaining States per Target Mode\discharging *` folder.

3. To specify another transition from `charging`, do the following:

   i. Double-click a suggestion to display it in the "Rules Editor" field.

   E.g., click the entry in the `combustion engine only` folder.

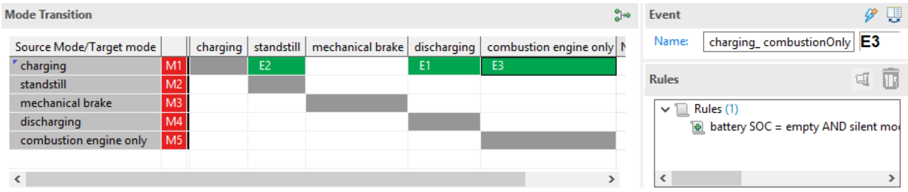
   ii. Click the **Add Event from Rule** button.

   The event is added, together with the rule you selected.

iii. Enter a meaningful name for the new event.

For the transition from `charging` to `combustion engine only`, for example, name the event `charging_ combustionOnly`.

iv. Assign the event to a transition.



v. Update the "Analysis Details" view.

vi. Add rules from the other suggestions for this transition.

4. Repeat step 3 until all transitions from `charging` are complete and no errors remain. [9]

   Events without errors are displayed in green in the transition matrix.

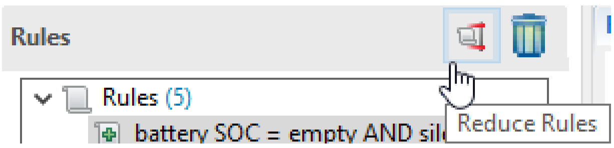
   If all transitions from a mode are complete and free of errors, the mode is displayed in green, too.

4.6.2. Optimizing the Rules

An event can have many rules that may overlap or appear overly complex. You can try to optimize the rules for a selected event and reduce their number and/or complexity.

To optimize rules:

1. Select the event whose rules you want to optimize.
2. In the event viewer, select one or more rules of the event.
3. Click the **Reduce Rules** button.



The "Reduce Rules" window shows the results. If the rules could be reduced, the event viewer is updated.

[Figure 23](#) shows an example for successfully reduced rules.

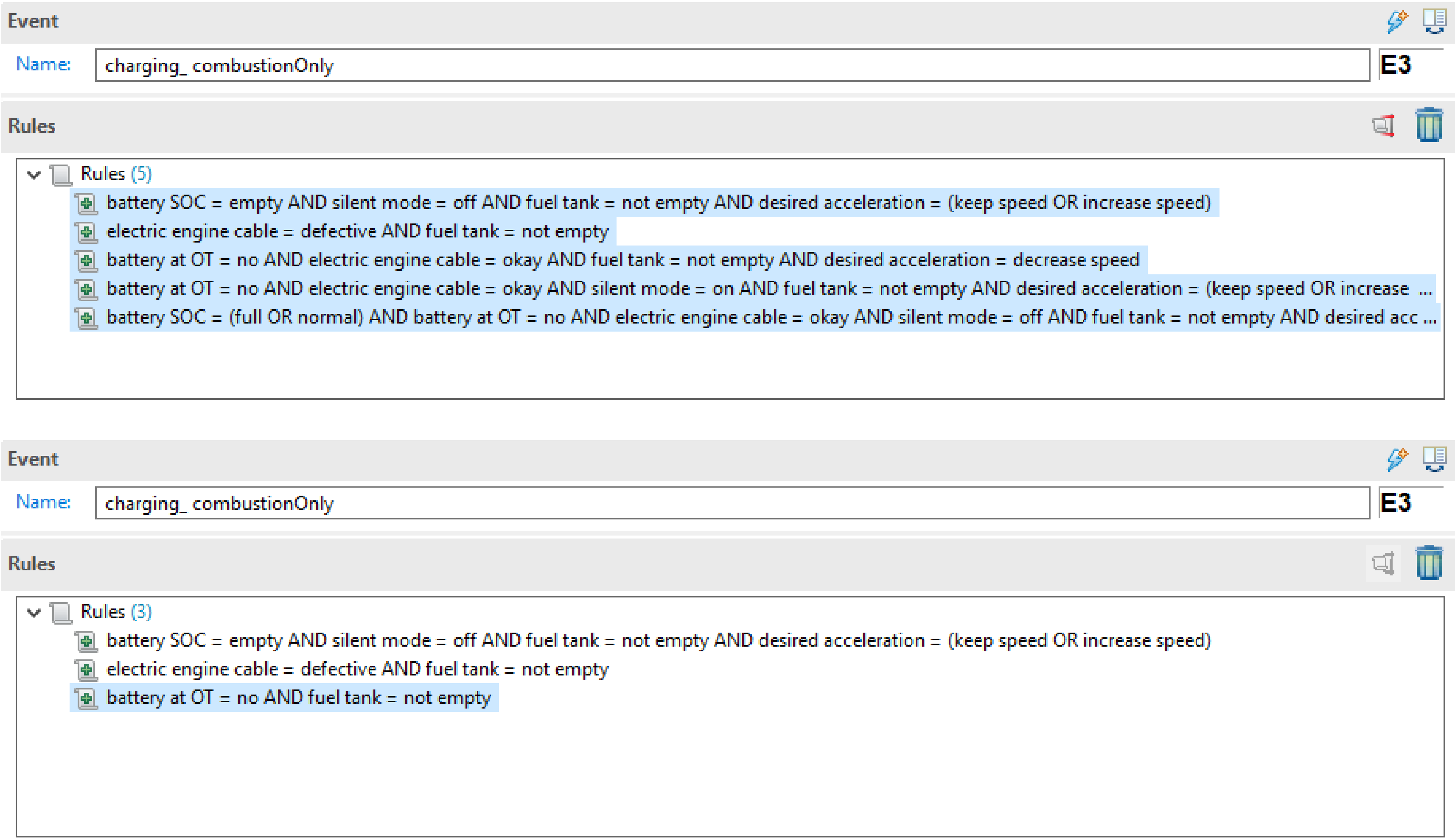


Figure 23. Event `charging_combustionOnly` before and after rule optimization

4.6.3. Completing the Transition Matrix

The transitions away from the charging mode are specified. Use the procedures from section 4.6.1, “Creating and Editing Events and Transitions from One Mode” and section 4.6.2, “Optimizing the Rules” to specify the entire transition matrix [10] and optimize the results.

The completed transition matrix looks like this:

Mode Transition

Source Mode/Target mode		charging	standstill	mechanical brake	discharging	combustion engine only	No Transition
charging	M1		E2	E4	E1	E3	
standstill	M2	E5		E6	E7	E8	
mechanical brake	M3	E9	E10		E11	E12	
discharging	M4	E13	E14	E15		E16	
combustion engine only	M5	E17	E18	E19	E20		

The **Mode Transition Graph** button at the top right of the mode transition matrix creates a graphical display of the transitions; see Figure 24 for an example.

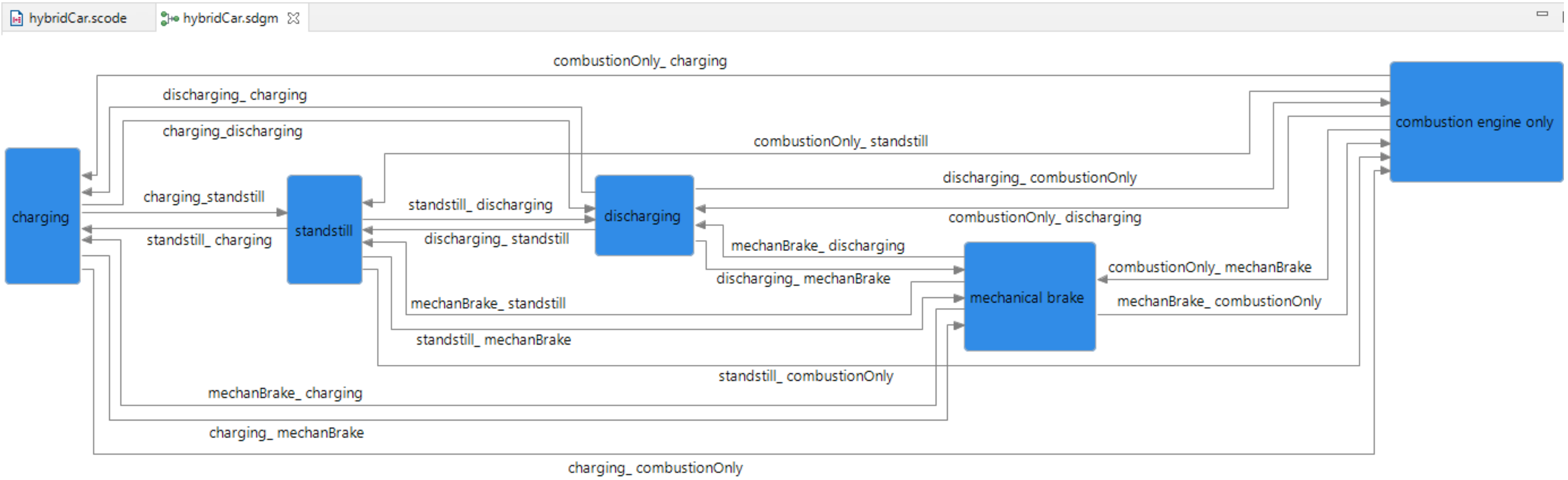


Figure 24. Mode transition graph for the completed transition matrix

4.7. Lesson 6: Code Generation from Mode Transition Matrix

The purpose of code generation is to transform this model into executable code that reflects the same functionality as the model.

As soon as your model is complete, deterministic, and consistent, and the transitions are completely specified, you can generate code from the transition matrix.

To prepare code generation from the transition matrix

1. Open the "Preferences" window and go to the "SCODE-ANALYZER\Generator" node.[7]
2. In that node, do the following:
   i. Select one or more generators.
   ii. For the "Generation Source" property, select `Mode Transition Matrix`.

      For the other properties, as well as for the generator-specific settings in the subnodes, you can use the default values.
3. Click **Apply and Close**.

The settings should look like those in Figure 25.

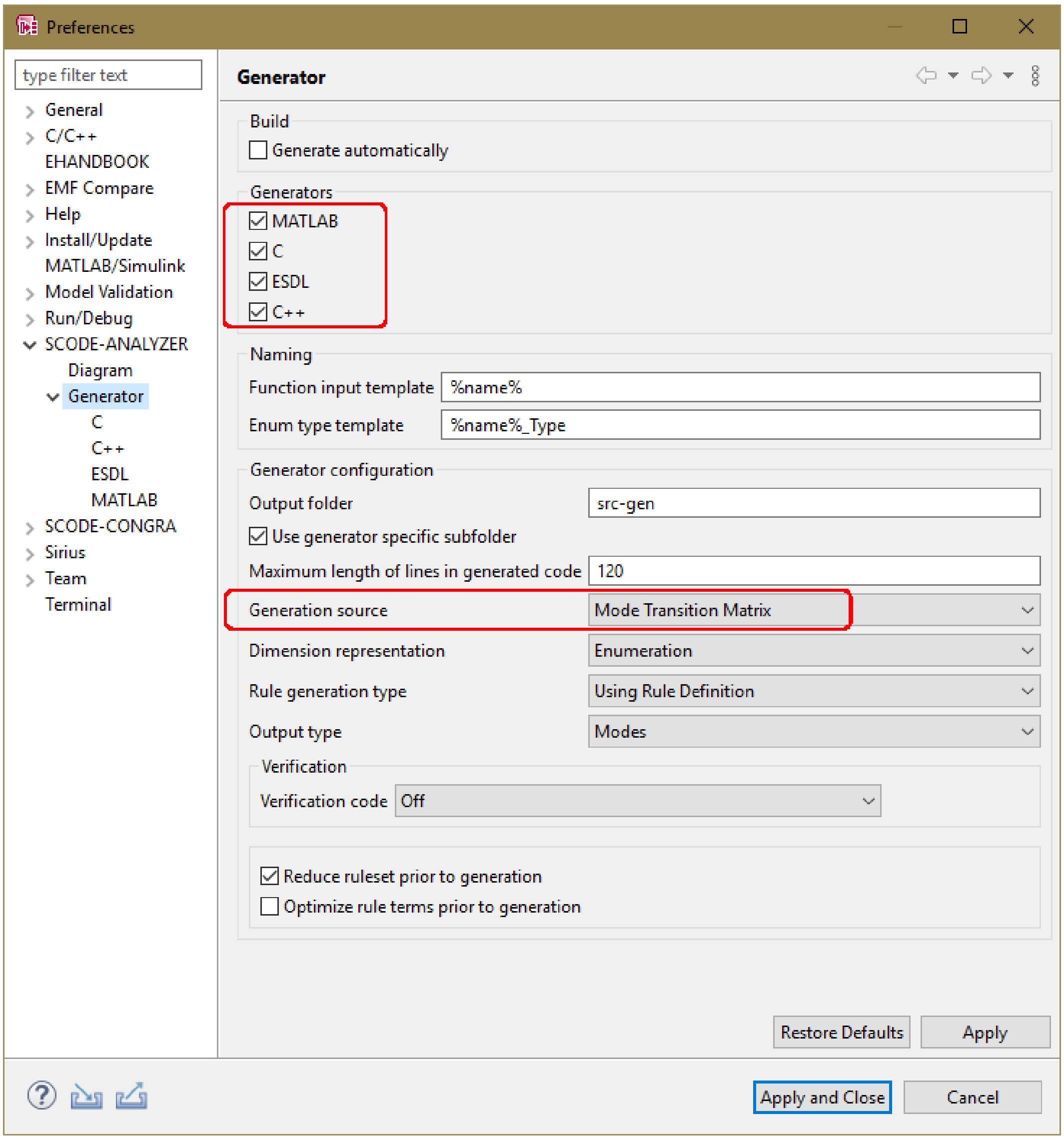


Figure 25. "Preferences" window with settings for code generation from the transition matrix

With that, you can generate the code.

To generate code from the transition matrix

1. In the Project Explorer, right-click the SCODE file and select **Generate Code** from the context menu.

   Code is generated for the selected generators (ESDL, C code, C++ code, or MATLAB). The resulting files are stored in the SCODE-ANALYZER project; the output folder is named `src-gen` by default.

   Figure 16 shows the output folder for generated code from mode invariants. The same files with the same names are created for code generated from the transition matrix.

2. Open the *<project_name>*`.*` file(s) and look at the code.

   See section 9.1.5, “Code Generation: Transition Matrix” for the `hybridCar.esdl` file.

3. If desired, compare the `<project_name>.*` file(s) from this section with the respective files from section 4.5, “Lesson 4: Code Generation from Mode Invariants”.

4.8. Lesson 7: Generating a Report

To get a description of your project, you can generate a report. Generated reports can be read without a SCODE Workbench installation.

To generate a report

1. Right-click the SCODE file and select **Run As > Report (ANALYZER)** from the context menu.

   If this is the first time you use the **Run As > Report (ANALYZER)** on this SCODE file, the "Edit Configuration" window opens for the report launch configuration.

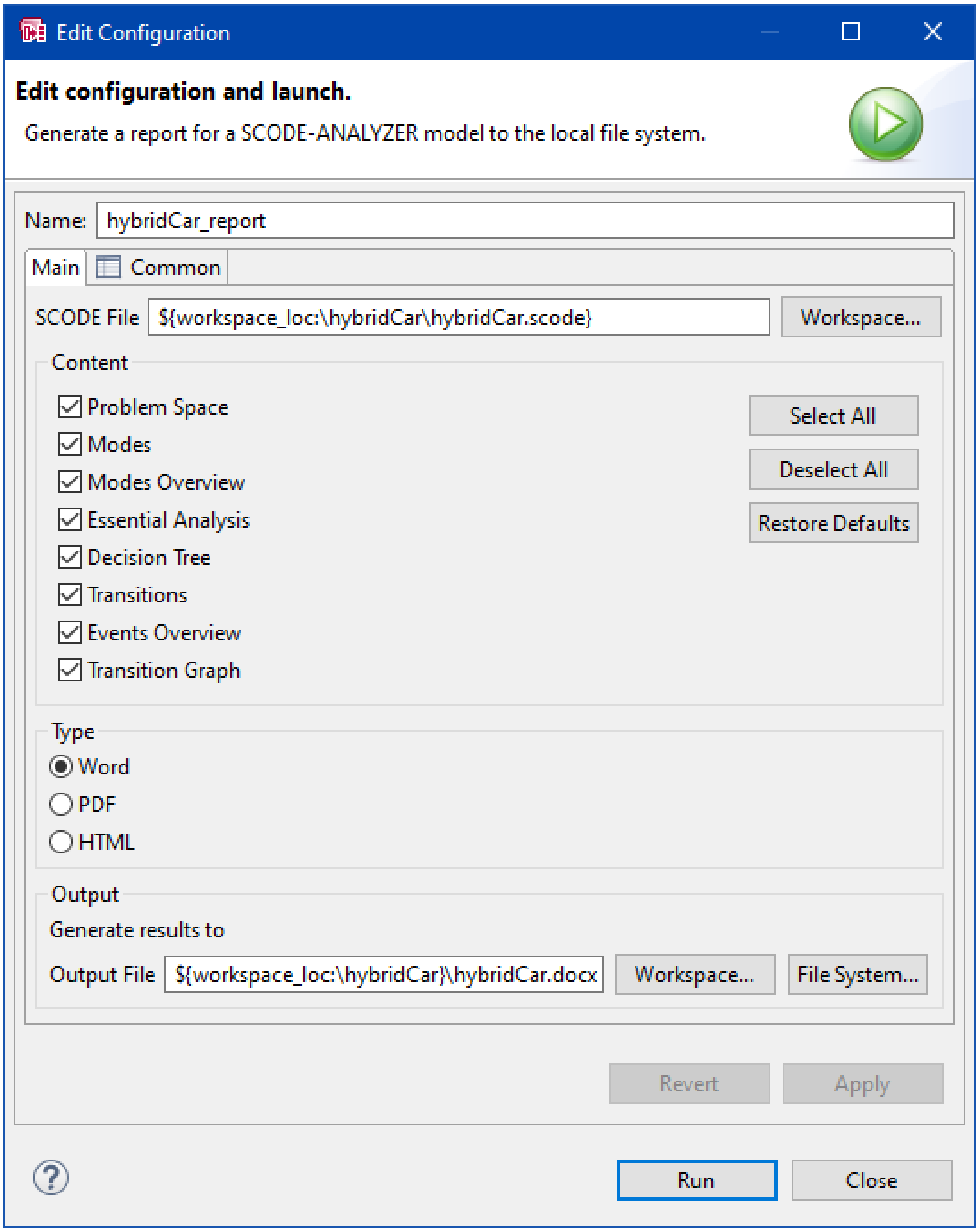


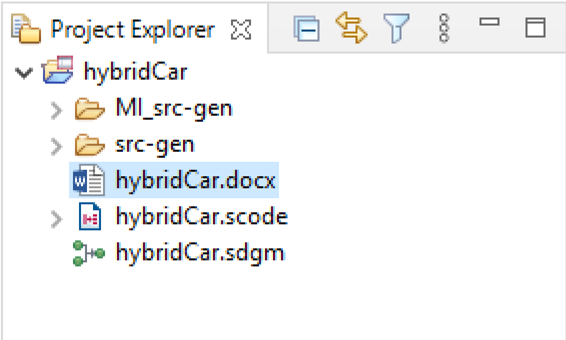
Figure 26. "Edit Configuration" window for a SCODE-ANALYZER report launch configuration

2. In the "Edit Configuration" window, enter a name for the report file.
3. In the "SCODE File" field, enter or select (via **Workspace** button) the SCODE file that contains the model you want to export.
4. In the "Content" area, activate at least one content option.
5. In the "Type" field, activate the option for the desired report file type.
6. In the "Output File" field, enter or select (via **Workspace** or **File System** button) an existing folder for the report.

NOTE If you enter the name of an existing file with the selected type, that file is overwritten without further inquiry.

7. Click **Run** to generate the report.

   The report is generated with the selected format and stored in the selected folder. If you selected a folder inside your workspace, you can see the report in the Project Explorer.

8. In the confirmation window, click **Yes** to open the report.

A report for the `hybridCar` project, with all report parts generated, is shown in section 9.1.6.

[2] See Table 25 for a suggestion of modes and rules.

[3] If you need help, see To add a new condition.

[4] If you need help, see To add a new condition and/or Table 26.

[5] If you need help, see Table 27, “Suggested rules for the missing states. Alternatives that cannot be true at the same time are marked.”

[6] If you need help, see Table 28, “Suggested rules for the states that are still missing after suggestion 1 from the previous table has been inserted as non-system mode”.

[7] If you need help, see, e.g., To prepare code generation from mode invariants.

[8] If you need help, see Table 30.

[9] If you need help, see Table 29 and references therein.

[10] If you need help, see the tables in section 9.1.3, “Events and Transitions”.

5. SCODE-CONGRA Tutorial

This chapter contains a tutorial for SCODE-CONGRA. A tutorial for SCODE-ANALYZER can be found in chapter 4.

5.1. Introduction

Users who are not yet familiar with SCODE-CONGRA will learn the basic working steps of SCODE-CONGRA in this tutorial. The tutorial does not require any knowledge of SCODE-CONGRA, but does assume that you are familiar with the Windows operating system and with Eclipse in general.

To model physical systems in SCODE-CONGRA, you define equation systems.

5.1.1. Concepts

This section introduces the most important concepts and processes used in this tutorial.

Workspace

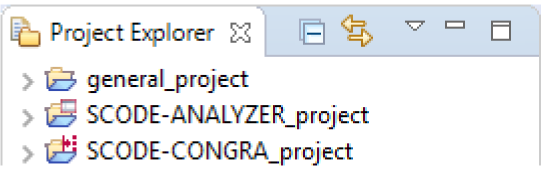
A workspace is a way to store all information specified or produced with SCODE-CONGRA (or SCODE-ANALYZER).

In SCODE-CONGRA, a workspace is structured into projects, folders, and files. On the Windows file system, a workspace is stored in form of folders and files with the same structure.

Project

A SCODE-CONGRA project stores a model.

SCODE-CONGRA projects are identified as such by the Eclipse environment. The constraint graph functionality is only available for projects of this type. In the following image, you see the difference between a SCODE-CONGRA project, a SCODE-ANALYZER project, and a general project.



System

A system is defined as a set of variables and relations between the variables. A system is undirected, i.e. no inputs and outputs are specified. You cannot generate executable code from an undirected system.

SYQ file

A textual file in the SYQ language that contains the semantic description of the system.

A SYQ file is the textual base of each SCODE-CONGRA project. Here, all variables, relations, units, and flows are defined or stored (when you are working in the graphical editor).

Each SCODE-CONGRA project must have at least one SYQ file.

Variable

A variable is an element that can be read and written during the execution of a SCODE-CONGRA model.

In SCODE-CONGRA, all variables are deemed to be continuous.

Relation

A relation describes how different variables of a system are interrelated. It does not imply a computation direction.

The relations between different variables are specified by mathematical equations, e.g., Einstein’s famous relation: $E - m*c^2 = 0$

Flow

A flow is a system with specified inputs and specified or derived outputs.

If a flow is valid, the equations in the system become directed to produce the imposed outputs of the relations.

For example, if m and c are given, then E is computed as $E = m * c^2$.
If E and c are given, then m is computed as $m = E/c^2$.

A valid flow is the basis for code generation.

Computation

A computation is the result of solving a flow, an executable sequence of computation steps. It captures the solved equations, and also orders the computation steps in a linear way.

5.1.2. Preparations

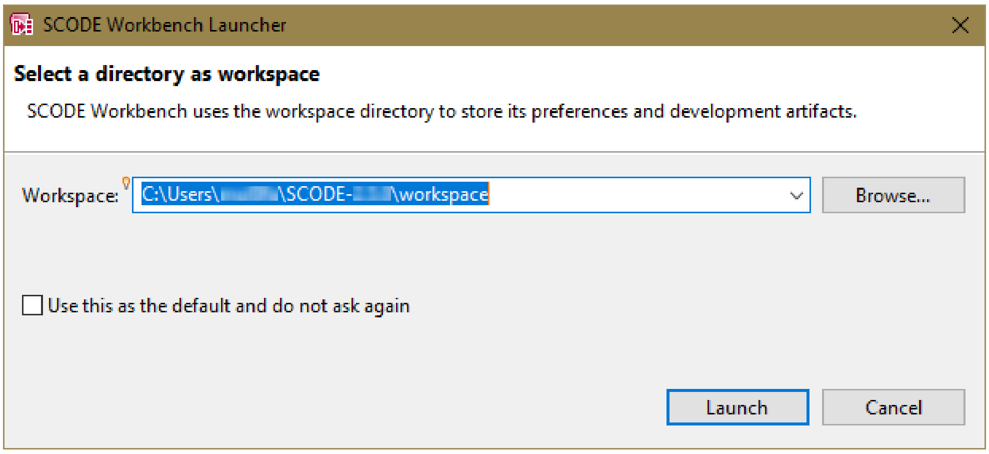
The first thing to do is to start the SCODE Workbench and open a workspace.

It is recommended that you use a separate workspace for the tutorial.

To create a SCODE-CONGRA workspace

1. Start the SCODE Workbench.

The "SCODE Workbench Launcher" window opens. It asks for a workspace location.



2. In that window, enter or select (via the **Browse** button) a path and name for your workspace.

   This tutorial uses a workspace named `WS_tutorial`.

3. Click **OK**.

   If you entered a directory that does not yet exist, it is created now.

   The SCODE Workbench opens. If you selected a new workspace, all views are empty (see Figure 27). If you selected an existing workspace, that workspace is shown in the views.

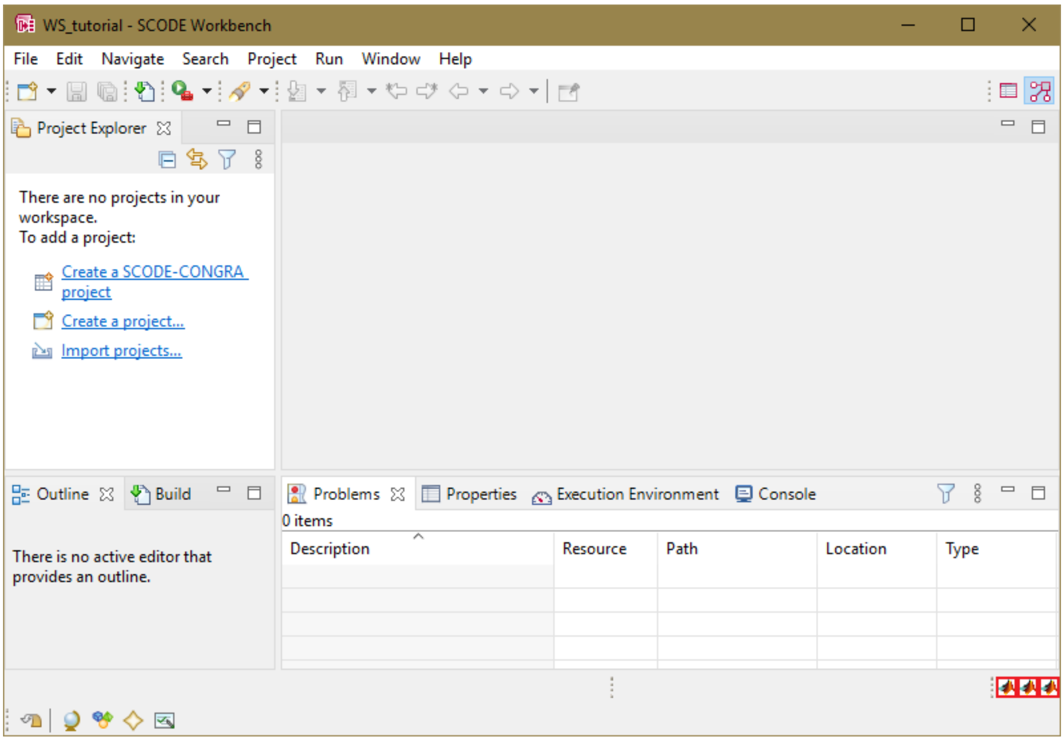


Figure 27. SCODE Workbench (SCODE-CONGRA perspective) with empty workspace

If you used the SCODE Workbench with SCODE-ANALYZER before you started this tutorial, your window may look different from Figure 27. To open the SCODE-CONGRA perspective, click the **SCODE-CONGRA** button at the right of the toolbar.

The SCODE-CONGRA perspective shows the following views:

- top left: Project Explorer
- top right: reserved for various editors
- bottom left: "Outline" view and "Build" view
- bottom right: "Problems" view, "Properties" view, Execution Environment, "Console" view

By default, the built-in solver and the Maxima solver are selected for the entire workspace.

With that, SCODE-CONGRA is ready to be used.

However, a connection to MATLAB can be useful for working with SCODE-CONGRA. See To connect SCODE Workbench and MATLAB for further information.

5.2. Lesson 1: Simple Equation

In the first lesson of this tutorial, you will create a new project and specify a simple equation, i.e., Ohm's law,

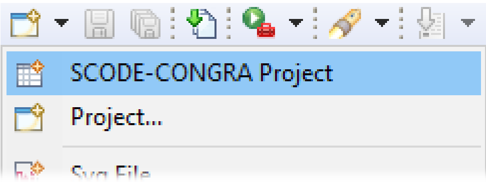
Equation 1: Ohm's law

```
U = R * I
```

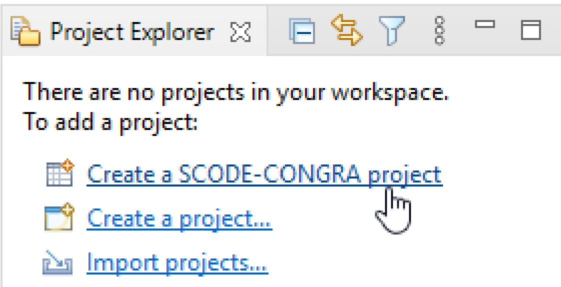
In Equation 1, R is the resistance in ohms, U is the voltage in volts, and I is the current in amperes.

To create a SCODE-CONGRA project

1. In the SCODE Workbench window, do one of the following:
   - Select **File > New > SCODE-CONGRA Project**.
   - Click the arrow next to the **New** button and select **SCODE-CONGRA Project**.

   - Follow the Create a SCODE-CONGRA project link in the Project Explorer.

The "SCODE-CONGRA Project" window opens.

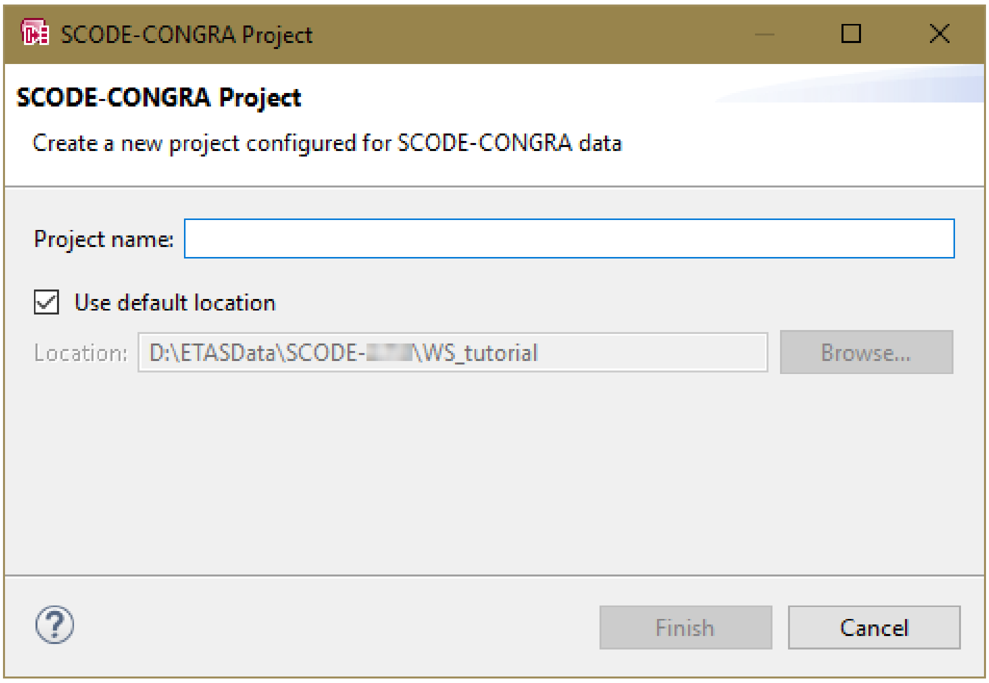


Figure 28. "SCODE-CONGRA Project" window

2. Enter a project name, e.g., `Simple_Equation`.

   It is recommended that you use the default location for this tutorial.

3. Click **Finish**.

   The project is created, together with some default elements. The `*.syq` file is shown in the SCODE Workbench window.

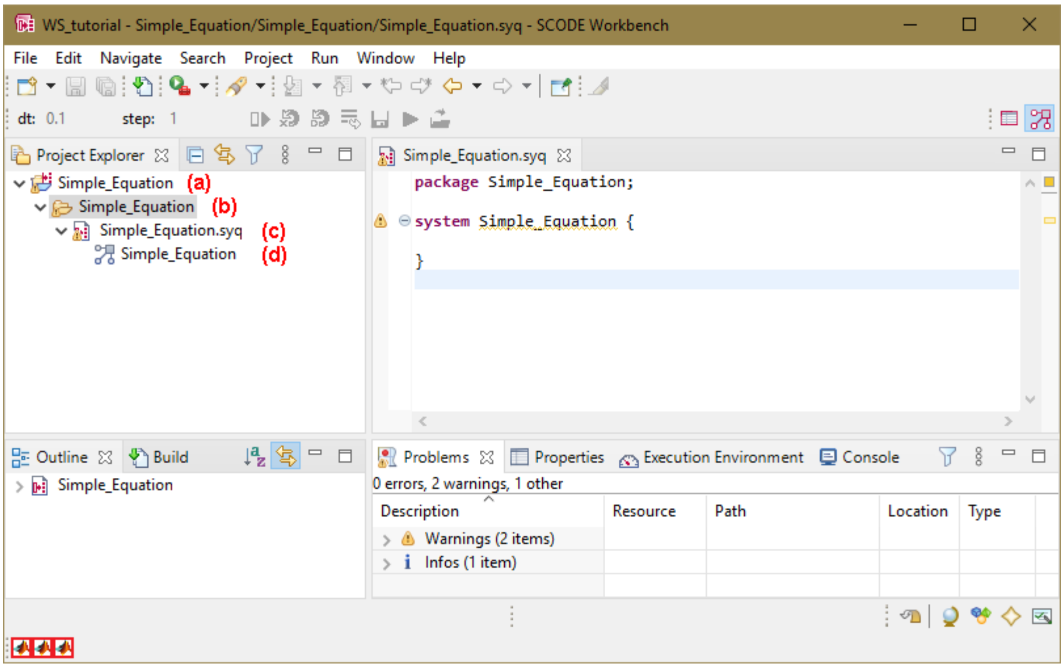


Figure 29. SCODE Workbench window with newly created SCODE-CONGRA project (a: project folder, b: system folder, c: system equation language package (`*.syq` file), d: system graph)

5.2.1. Defining the Equation

The equation system is specified graphically in the system graph (d in Figure 29). For this example, you will create one relation between three variables. The variables themselves are created automatically once the relation is specified.

To specify the equation

1. In the Project Explorer, double-click the `Simple_Equation` system.

   The system opens in the graphical editor.

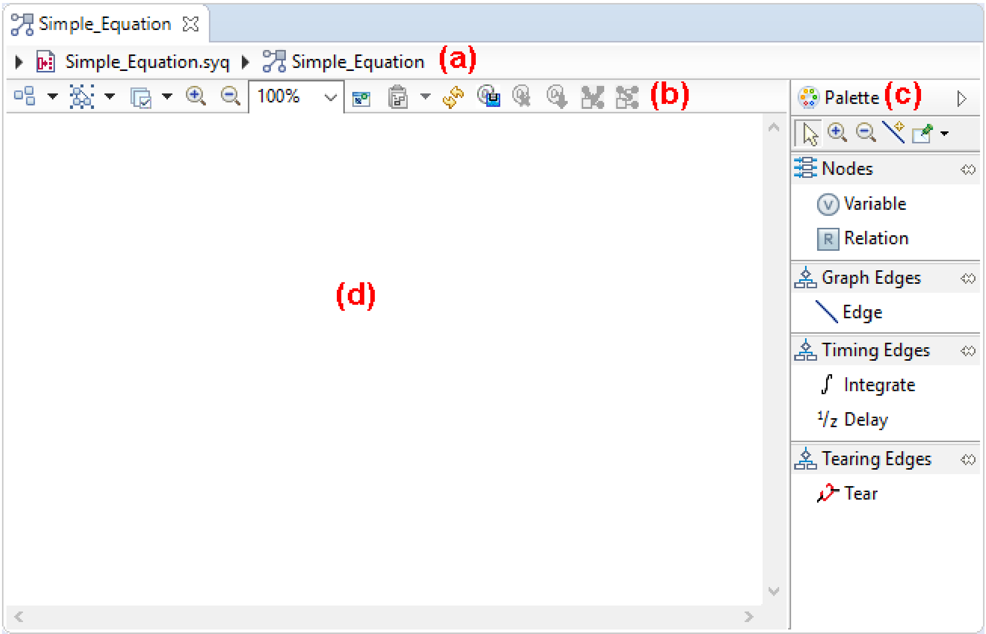
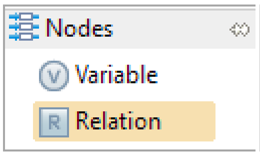
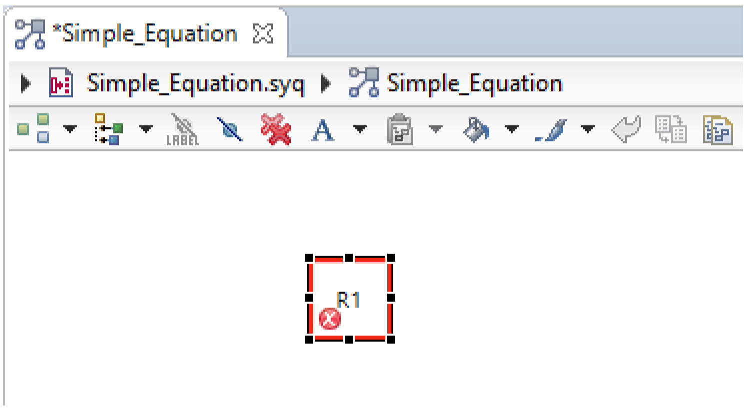


Figure 30. Graphical editor (a: breadcrumbs row, b: toolbar for general editor functionality, c: palette with tools for graphical elements, d: empty canvas)

2. In the "Nodes" group of the palette, click **Relation**.



3. Click the canvas to insert the relation.



The relation is represented by a rectangle. In the screenshot above, the rectangle border is red because the relation is over-determined. The error symbols is shown because a relation needs at least one variable.

4. Open the "Properties" view for the relation. [11]

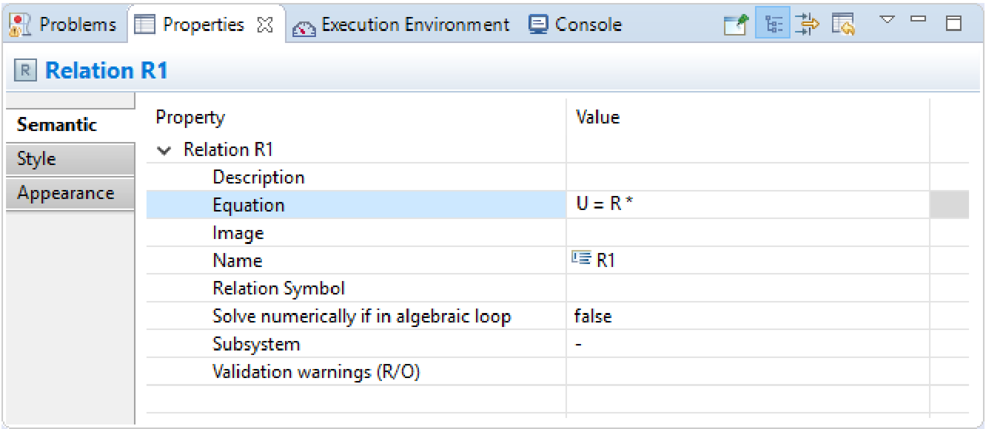


Figure 31. "Properties" view for the new relation (equation still incomplete)

5. In the "Properties" view, "Semantic" node, click in the "Value" column next to "Equation".

   The cell becomes an input field.

6. Enter the equation and press Return.

   The equation is accepted. The rectangle border is now blue. Variables (represented by blue circles) for R, U, and I are automatically added to the graph. Blue lines connect the variables and the relation.

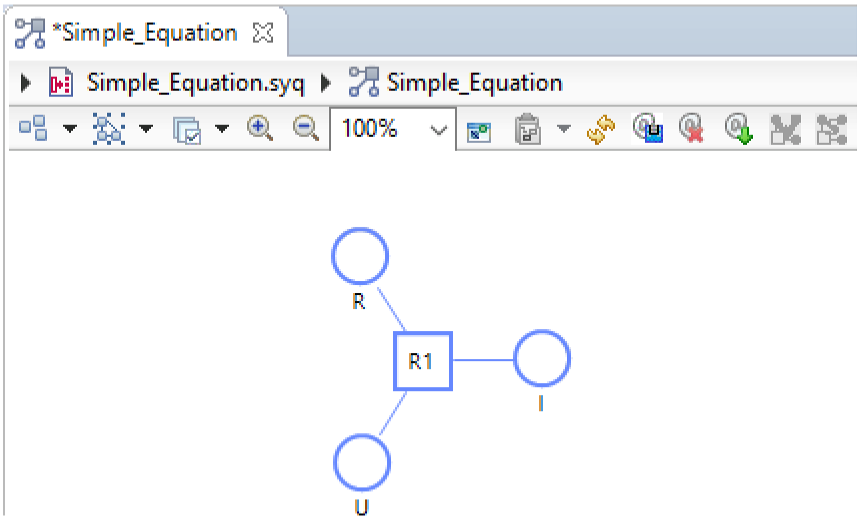
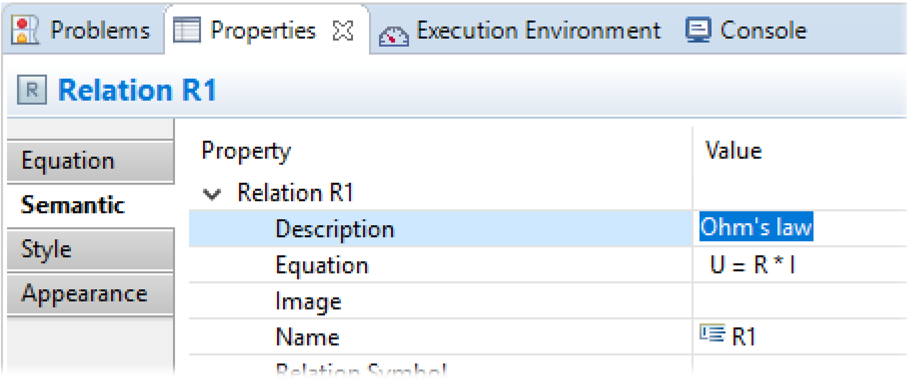


Figure 32. Canvas with relation and variables

7. If desired, enter a description and/or rename the relation.



8. Save the project.

   The graphical specification is added to the `*.syq` file.

```
package Simple_Equation;

system Simple_Equation {

@geo(162, 195)
    var U;
    @geo(161, 92)
    var R;
    @geo(252, 143)
    var I;

    @description("Ohm\'s law")
    R1(U, R, I) ::= U = R * I;
}
```

Figure 33. `Simple_Equation.syq` file with variables and relation

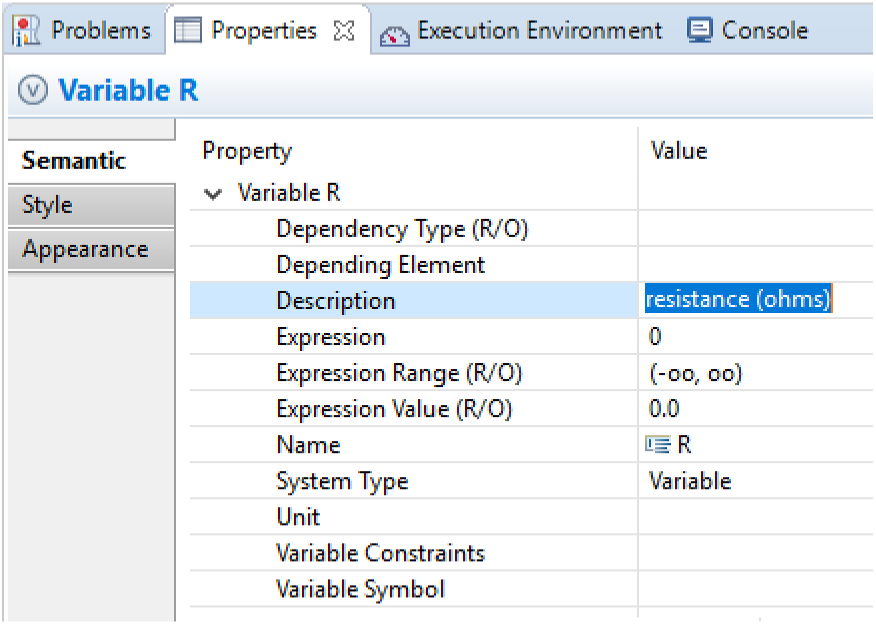
Next, you edit the variables.

To edit the variables

1. In the graphical editor, select the variable R.
2. Open the "Properties" view for the variable.
3. In the "Description" row, "Value" column, enter a description for resistance R.

   This tutorial uses the description `resistance (ohms)`.

4. In the "Expression" row, enter a default value.



5. Enter descriptions and default values for U and I, too.

> **NOTE** Unless otherwise stated, this tutorial uses the default value 0 for all variables.

6. Save the project.

   The `*.syq` file is updated.

With that, your system is defined. It is undirected, i.e. you have not yet specified inputs and outputs.

5.2.2. Specifying Directions

Before computation can start, a direction must be added, i.e. you have to determine which variables are to be computed, and which variables are inputs.

For the current example, one equation is sufficient to compute either R, or U, or I.

Directions are not added to the system graph itself. You create a flow from the system graph and specify the direction in that flow. The latter is done by assigning types to the variables. The following types are available:

type	set via variable's context menu	description	see also
input		shown in diagram as ◯	Figure 35
fixed	**Set Type > ***	Variable with fixed value *in current flow*; can be calibrated shown in diagram as ◯	section 5.4.3
free		default value	

type	set via variable's context menu	description	see also
output	implicitly or via **Set Type > Output**	shown in diagram as (implicit output) or (explicit output) if the automatic analysis finds that the output is computable	impl.: Figure 35 expl.: Figure 61
parameter	**Set Type > Parameter**	Fixed value in entire system; can be calibrated shown in diagram as	section 5.4.2
constant	**Set Type > Constant**	Fixed value in entire system; cannot be calibrated hidden in diagram; can be edited only in the `*.syq` file	section 5.4.1

Table 3. Variable types available in a flow

To create a flow

In the first flow you create, R shall be computed from U and I.

1. In the Project Explorer, right-click the system graph and select **New > Flow** from the context menu.

   A new flow is created with a default name. It opens in the graphical editor. Since no inputs are defined, all elements of the graph show an error mark because they cannot be computed right now. The blue color indicates that the flow is under-determined.

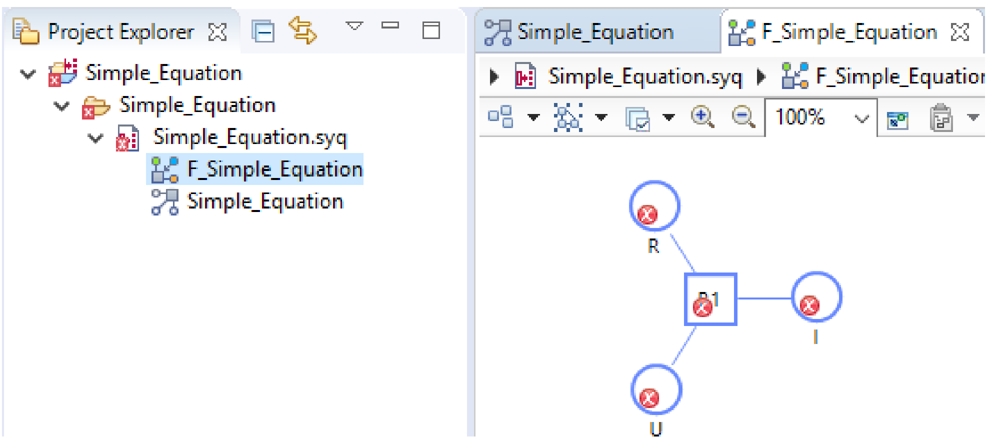


Figure 34. New flow in the graphical editor

2. Right-click a variable you want to use as input and select **Set Type > Input** from the context menu.

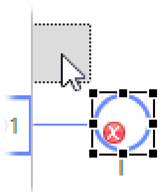
The variable is now shown as a blue circle with black border. The connection from the variable to the relation is now an arrow with a white head.

3. Specify the second input.

   The remaining variable of type free can now be computed. It is implicitly treated as an output because no output is defined explicitly.

   Arrows with white heads mark incoming elements, arrows with black heads mark outgoing elements. [12]

4. If desired, rearrange the variables and the relation on the canvas. [13]



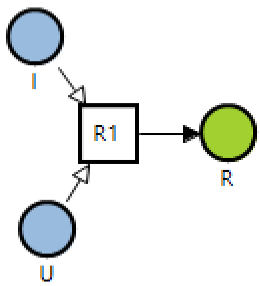


Figure 35. Flow after I and U have been defined as inputs (the variables were rearranged).

NOTE

Layout changes in one flow affect all other flows, computations, and the system graph.

5. Rename the flow and enter a meaningful, unique name.

   In this tutorial, flows are named according to the following scheme:

   `F_<system name>in<inputs>`, e.g., `F_Simple_Equation_in_IU`

6. Save the project.

   A computation is created for the flow. It consists of a `*.syq` file and a graph, stored in the project, in the code generation folder.

   By default, a computation is named `c_<flow name>`.

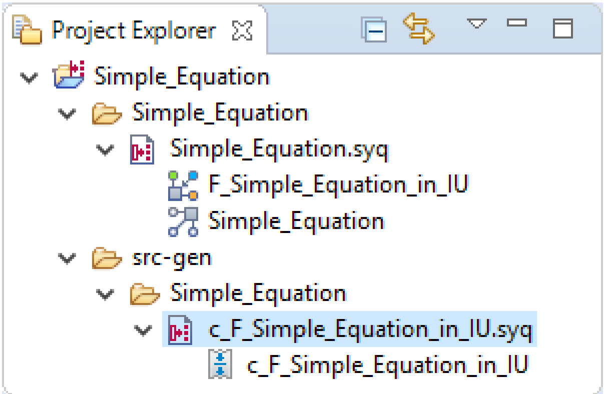


Figure 36. Computation `c_F_Simple_Equation_in_IU` in the Project Explorer

More information on computations is given in section 5.2.3, “Working with Computations”.

You can use the system to create as many flows as you need. For the simple equation system, create two more flows, one that computes U, and one that computes I.

5.2.3. Working with Computations

Each time you save the SCODE-CONGRA project, a computation is created or updated for each valid flow. A computation collects the solved equations, and also orders the computation steps. Protections, e.g., against division by zero, are inserted automatically.

Computations are stored in the code generation subfolder (named `src-gen` by default) of the project, in a subfolder named *<system name>*. See Figure 36 for an example.

The `*.syq` file of the `computation c_F_Simple_Equation_in_IU` reads as follows:

```
/**
 * @warning    AUTOMATICALLY GENERATED FILE! DO NOT EDIT!
 *
 * @source     F_Simple_Equation_in_IU
 *
 * @tool       ETAS SCODE-CONGRA 3.1.0
 *
 **/

package Simple_Equation;

computation c_F_Simple_Equation_in_IU (I, U)
            implements Simple_Equation
            from F_Simple_Equation_in_IU {
    // Variable computation for level 2
   @level(2, 1)
   R = if (0.0!=I) then U/I else  <- R1(I, U);
                       //  [Source: Built-In Solver]
   [R,I] = if (0.0!=I) then -U/I^2 else  <- R1(I, U);
                       //  [Source: Built-In Solver]
   [R,U] = if (0.0!=I) then 1/I else  <- R1(I, U);
                       //  [Source: Built-In Solver]
}
```

Table 4. `*.syq` file for the `c_F_Simple_Equation_in_IU` computation. Line 15 shows the equation used to compute R, lines 16 and 17 show the partial derivatives of the equation.

The computation graph, shown in Figure 37, looks very similar to the flow graph shown in Figure 35, except that the computation graph shows the following items:

- values of the variables (black text in Figure 37)
- computation level (red number in Figure 37)

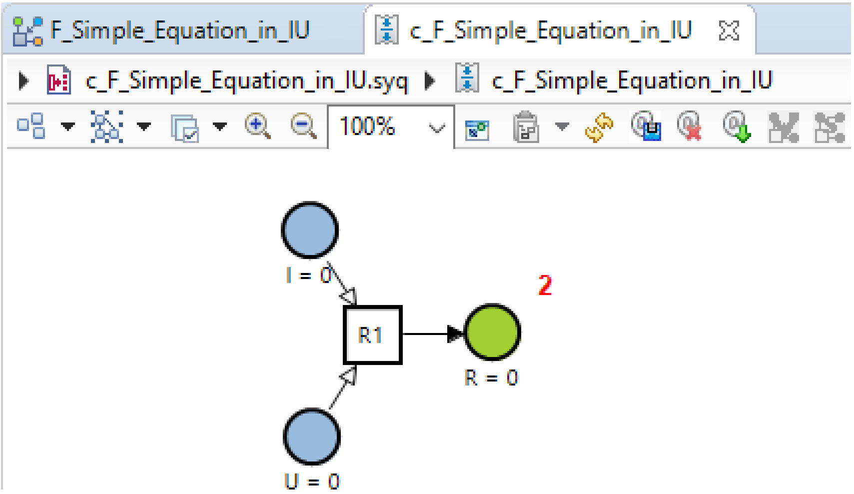
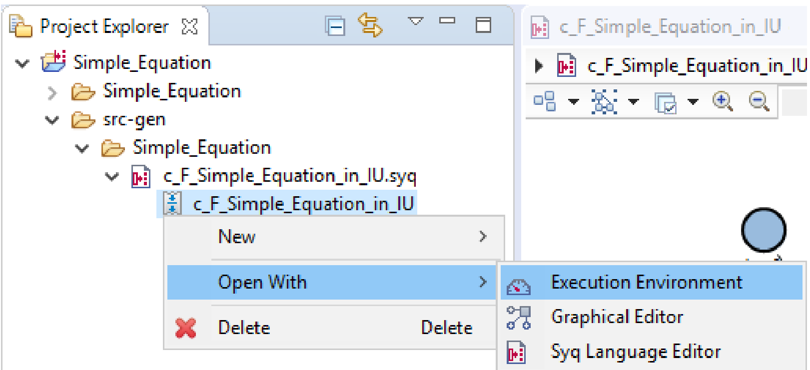


Figure 37. Graph for the `c_F_Simple_Equation_in_IU` computation.

To work with computations, SCODE-CONGRA provides the *Execution Environment*.

To open the Execution Environment

1. In the Project Explorer, right-click the desired computation graph and select **Open With > Execution Environment** from the context menu.



   The Execution Environment opens and displays the elements of the flow. The computation graph is not opened.

Or

2. Select **Window > Show View > Execution Environment**.

   By default, the Execution Environment is visible in the SCODE-CONGRA perspective, so that this step can be omitted.

3. In the Project Explorer, double-click the desired computation graph to open it in the graphical editor.

   The flow elements then appear in the Execution Environment (see Figure 38). The output row is shown in red to mark an error, here a division by 0.

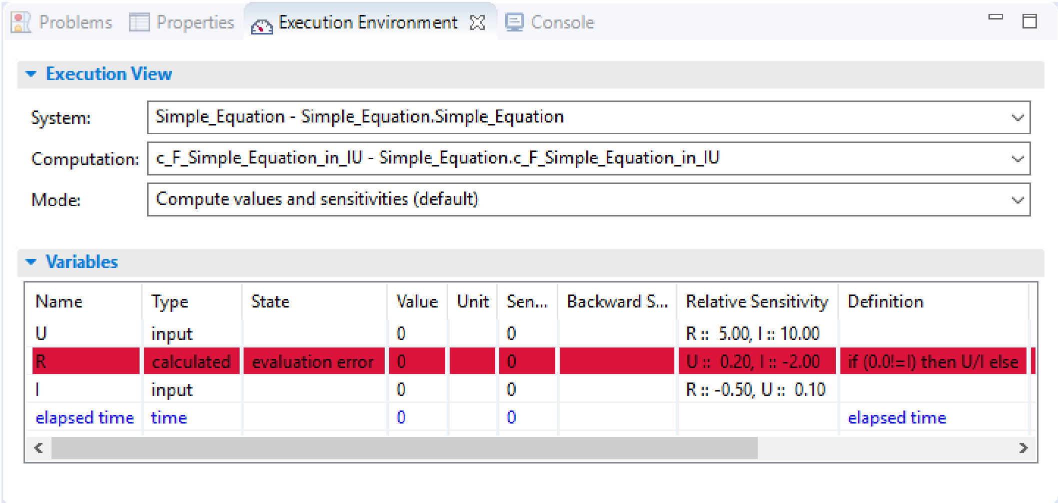
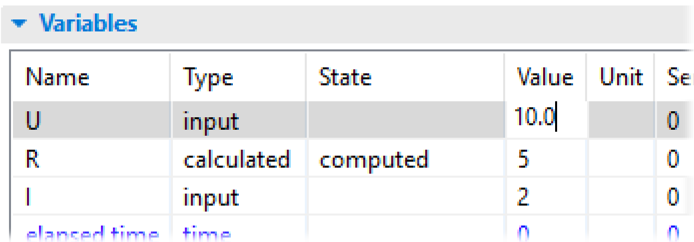


Figure 38. Execution Environment with a computation

In the Execution Environment, you can enter input values and sensitivities, and see the result immediately. If desired, the values can be shown in the graphical editor for the computation.

To check values in the Execution Environment

1. Click in the "Value" column of both inputs and enter values.

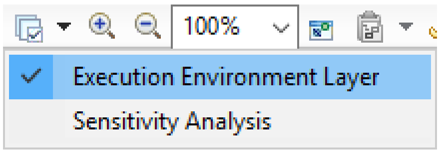


   The output is calculated immediately. As soon as the error due to the initial value `I = 0` is removed, the red color disappears from the output row.

2. In the graphical editor, click an empty place in the canvas.

   NOTE: The **Layers** button is only available if no element is selected in the canvas.

3. Click the **Layers** button and select **Execution Environment Layer** from the dropdown menu.



The current values of the elements are displayed.

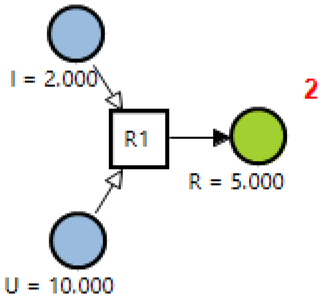


Figure 39. Computation graph with element values

You can use the Execution Environment also to perform a sensitivity analysis, i.e. to check the effect of a change (sensitivity) in some variables to other variables in the system.

The sensitivity analysis works as follows:

A relation uses the inputs $x_1, x_2, \ldots x_n$ to compute a variable y via the function f:

$$y = f(x_1, \ldots x_n)$$

The input sensitivities $Dx_1, \ldots Dx_n$ are given. For a particular operating point ($x_1, x_2, \ldots x_n$), the sensitivity `Dy` of the output `y` is computed as follows:

$$Dy = \sum_{i=1}^{n} \frac{\partial f(x1, \ldots, xn)}{\partial xi} * Dxi$$

Equation 2: Output sensitivity `Dy`

`df/dxi` is a partial derivative, i.e. the derivative of `f` with respect to x_i.

In this tutorial, `f` is a flow you created; x_i and `y` are the inputs and output of the flow. See rows 18 — 20 in Table 4 for an example of a function and its partial derivatives.

To check sensitivities in the Execution Environment

1. Display a computation in the Execution Environment and in the graphical editor.
2. In the graphical editor, click the **Layers** button and select **Sensitivity Analysis** from the dropdown menu.

   The current sensitivities are shown in the diagram.
3. For a manual sensitivity check, do the following:
   i. Click in the "Sensitivity" column of both inputs and enter values.

Variables

Name	Type	State	Value	Unit	Sensitivity	Backward
U	input		10		1.4	
R	calculated	computed	5		0.45	
I	input		2		0.1	
elapsed time	time		0		0	

The output sensitivity is calculated immediately.

In the graphical editor, the input sensitivities are shown, as well as the contribution of each input to the output sensitivity.

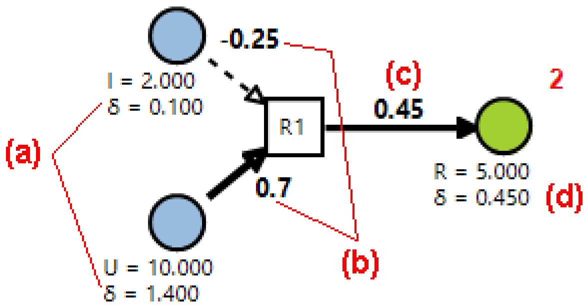


Figure 40. Computation graph with input sensitivities (a), their contributions (b) to the output sensitivity (c and d). The thickness of the arrows represents the relative sensitivities.

ii. If necessary, click the **Refresh diagram** button to update the values in the graphical editor.

4. For a forward sensitivity analysis, do the following:

   i. In the Execution Environment or in the graphical editor, right-click an input and select **Forward Sensitivity Analysis** from the context menu.

   The sensitivity of that input is set to 1, the sensitivities of other inputs is set to 0. The output sensitivity is computed according to Equation 2.

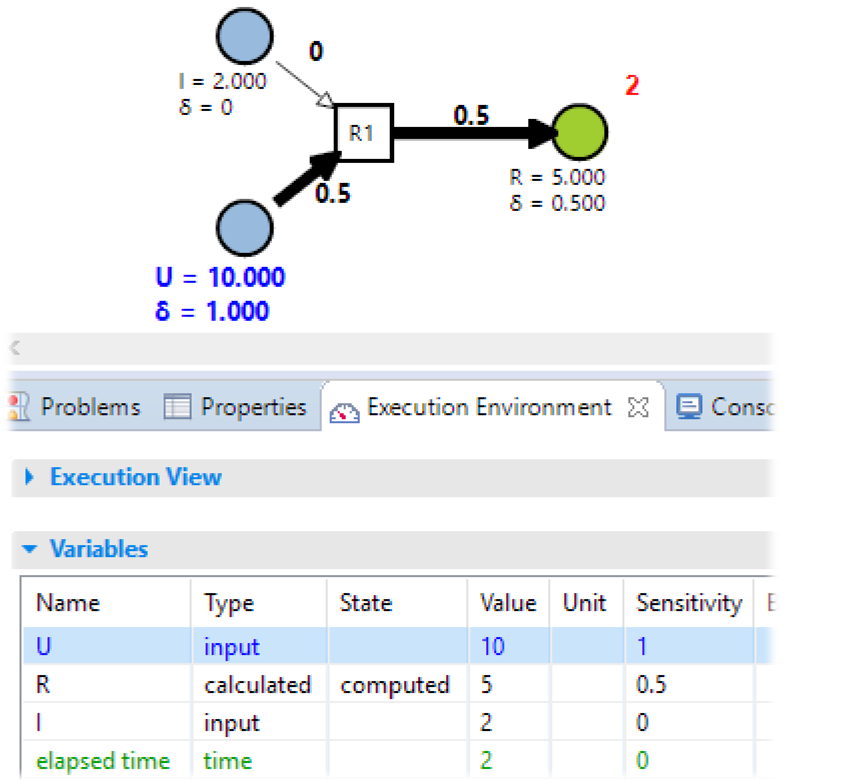


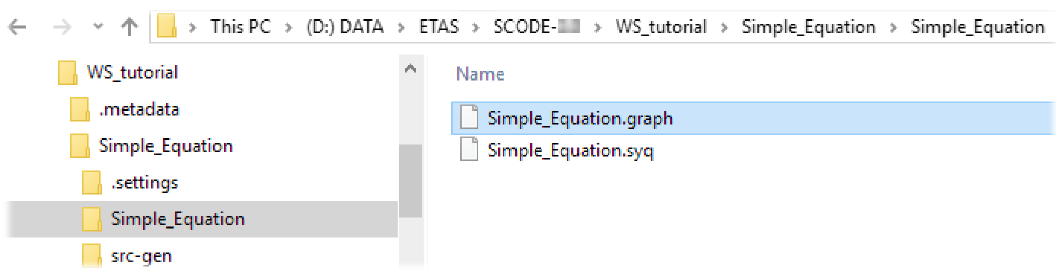
Figure 41. Computation graph and Execution Environment with results of forward sensitivity analysis

5.2.4. Additional Task

This section is not mandatory for the lesson on simple equations. However, it contains useful information.

Storing Layout Changes

When you change the diagram layout as described in To create a flow, the positions of the diagram elements are stored in a file named `<system name>.graph`, which is visible only in the Windows file system, not in the Project Explorer.



You can store the element positions in the `*.syq` file, as a set of `@geo` annotations (see, e.g., Figure 33).

A `@geo` annotation looks as follows:

```
@geo(<x>, <y>, <width>, <height>)
```

<x> and *<y>* are the horizontal and vertical positions of the top-left corner of the diagram element, measured in pixels from the top-left corner of the canvas.

<width> and *<height>* determine the size of a relation in the diagram. The size of a variable or parameter cannot be set.

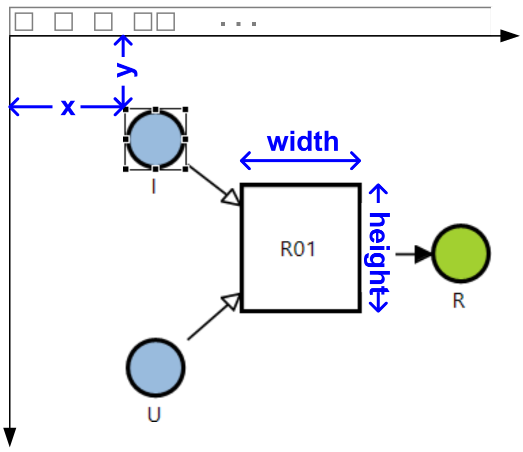
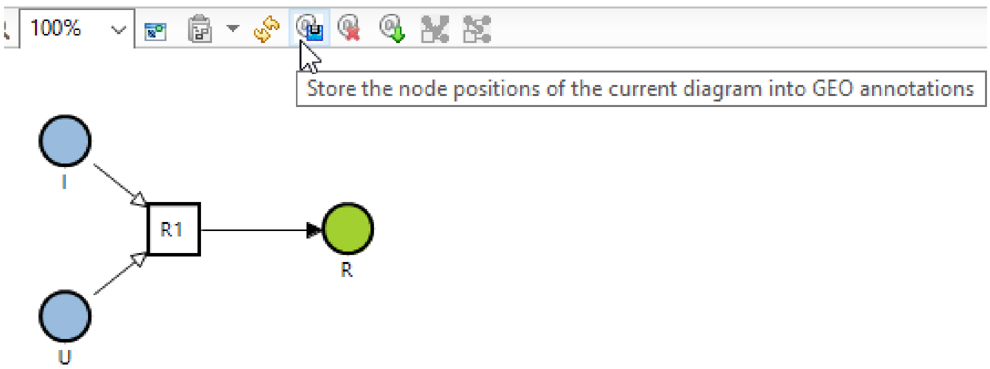


Figure 42. Schematic view of the numbers in `@geo` annotations

To store layout changes manually

1. In the system graph or flow graph, arrange the elements according to your needs.
2. Click an empty place of the canvas, so that no diagram element is selected.

   You cannot store the node positions while an element is selected.
3. Click the **Store the node positions * into GEO annotations** button.



4. Save the project.

   With that, existing `@geo` annotations in the `*.syq` file are updated, and missing `@geo` annotations are added.

```
4    system Simple_Equation {
5
6    @geo(264, 108)
7        @description("resistance (ohms)")
8        var R = 0;
9        @geo(109, 60)
10       @description("current (amperes)")
11       var I = 0;
12       @geo(109, 156)
13       @description("voltage (volts)")
14       var U = 0;
15
16       @description("Ohm\'s law")
17       @geo(168, 108, 30, 30)
18       R1(U, I, R) ::= U = R * I;
19   }
```

Table 5. `Simple_Equation` system with changed (lines 6, 9, 12) or added (line 17) `@geo` annotations

You can activate automatic storage of layout changes as `@geo` annotations when you save a project or workspace.

To activate automatic storage for layout changes

1. In the SCODE Workbench window, select **Window > Preferences**.
2. In the "Preferences" window, go to the "SCODE-CONGRA\Diagram Options" node.
3. Activate the **Update @geo annotation(s) by saving action** option.

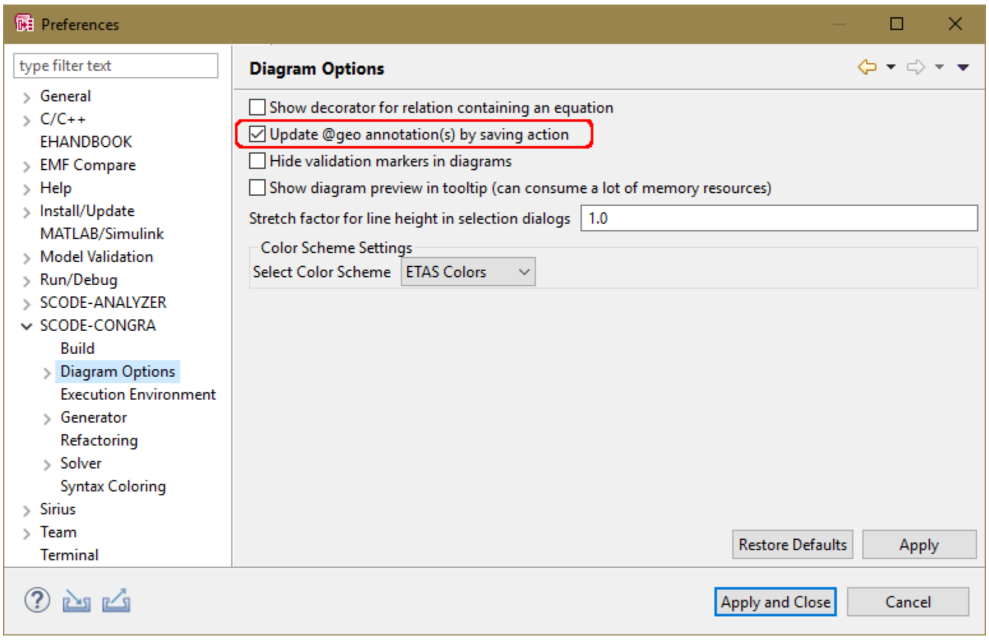


Figure 43. "Preferences" window, "Diagram Options" node

4. Click **Apply and Close**.

   The next time you save a project or workspace, unsaved layout changes are saved.

5.3. Lesson 2: Non-Linear Equation

The second lesson focuses on having multiple solutions, one of which you have to select. The lesson combines Ohm's law, U = R * I, with the power of an ohmic resistor, P = U * I.

Using both equations as relations in one system leads to an algebraic loop. This will be treated in a later lesson.

Here, you will use the combination equation:

Equation 3: Joule's law: power of an ohmic resistor (P -- electrical power in watt, R -- resistance in ohms, I -- current in amperes.)

$$P = R * I^2$$

5.3.1. Preparing the Project

For this lesson, MuPAD® is recommended as solver. To use MuPAD, you need a working installation of MATLAB® that includes Symbolic Math Toolbox™, you have to connect the SCODE Workbench and MATLAB, and you have to activate the MuPAD solver.

If you cannot use the MuPAD solver, use the Maxima solver, which is shipped with SCODE-CONGRA.

To connect SCODE Workbench and MATLAB®

1. In the SCODE Workbench window, select **Window > Preferences**.
2. In the "Preferences" window, go to the "MATLAB/Simulink" node.

   This node lists all versions of MATLAB installed on your computer.

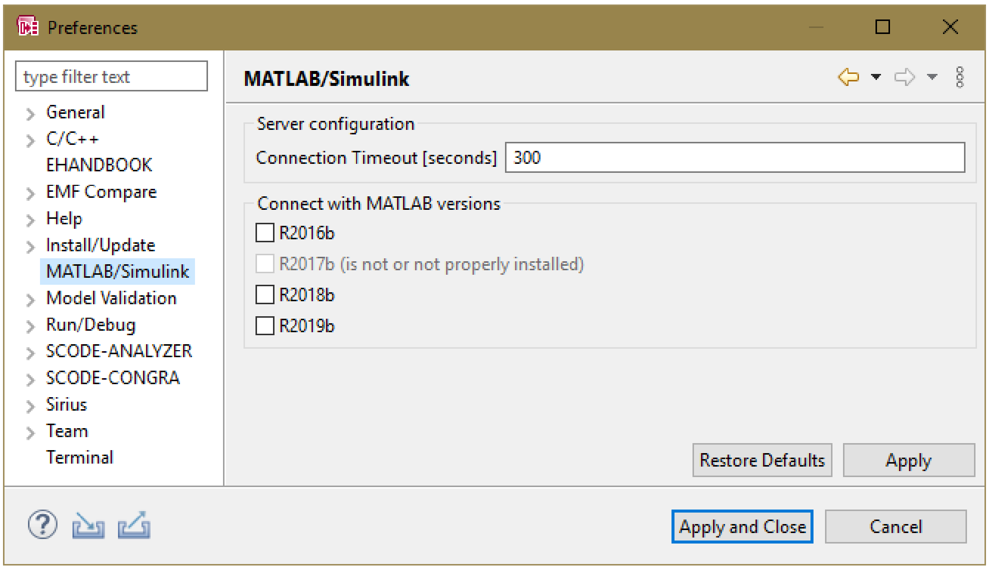


Figure 44. "Preferences" window, "MATLAB/Simulink" node

3. Select (☑) the version(s) you want to connect.
4. Select a MATLAB version for the MuPAD solver.
5. Click **Apply and Close**.

To select a MATLAB® version

1. Open the "Preferences" window.
2. In that window, expand the "SCODE-CONGRA" node and the "Solver" subnode, then go to the "MuPAD" subnode.

   This node contains settings for the MuPAD® solver. The "Select MATLAB installation for solving with MuPAD" combo box contains all MATLAB versions connected to the SCODE Workbench.
3. Select the MATLAB version you want to use.

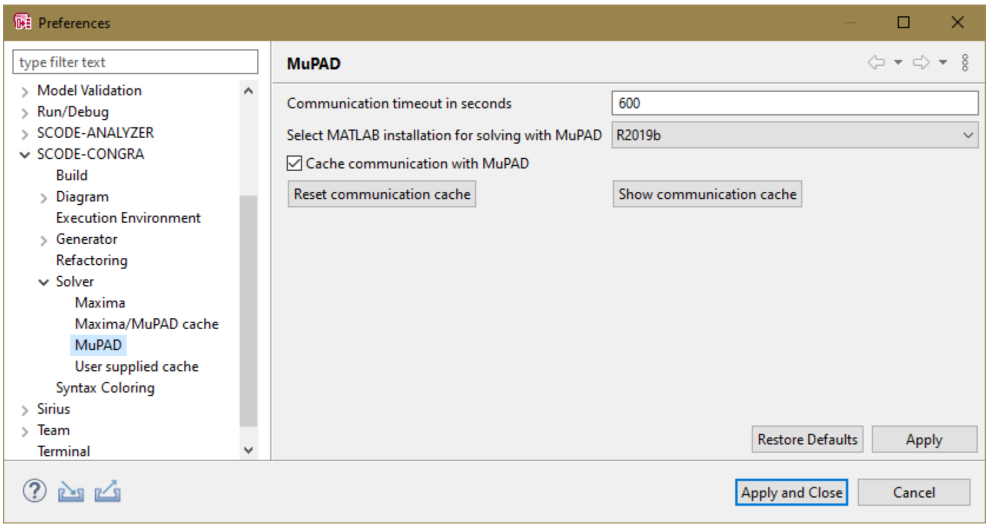


Figure 45. "Preferences" window, "SCODE-CONGRA\Solver\MuPAD" node

4. Click **Apply and Close**.

NOTE You can select a MATLAB version only for the entire workspace. Selecting a MATLAB version for a particular project is not possible.

SCODE-CONGRA offers the following setup possibilities:

- for the entire workspace (accessible via **Window > Preferences**)
- for a particular project (accessible via a project's context menu or via **Project > Properties**)

Project-specific settings override workspace settings. In this lesson, you will use project-specific settings.

To activate the MuPAD solver

1. Create a SCODE-CONGRA project and name it, e.g., `QuadraticEquation`.
2. In the Project Explorer, right-click the project and select **Properties** from the context menu.

   The "Properties for <*project*>" window opens.
3. In that window, expand the "SCODE-CONGRA" node and go to the "Solver" subnode.

By default, **Enable project specific settings** is deactivated; the project uses the workspace settings.

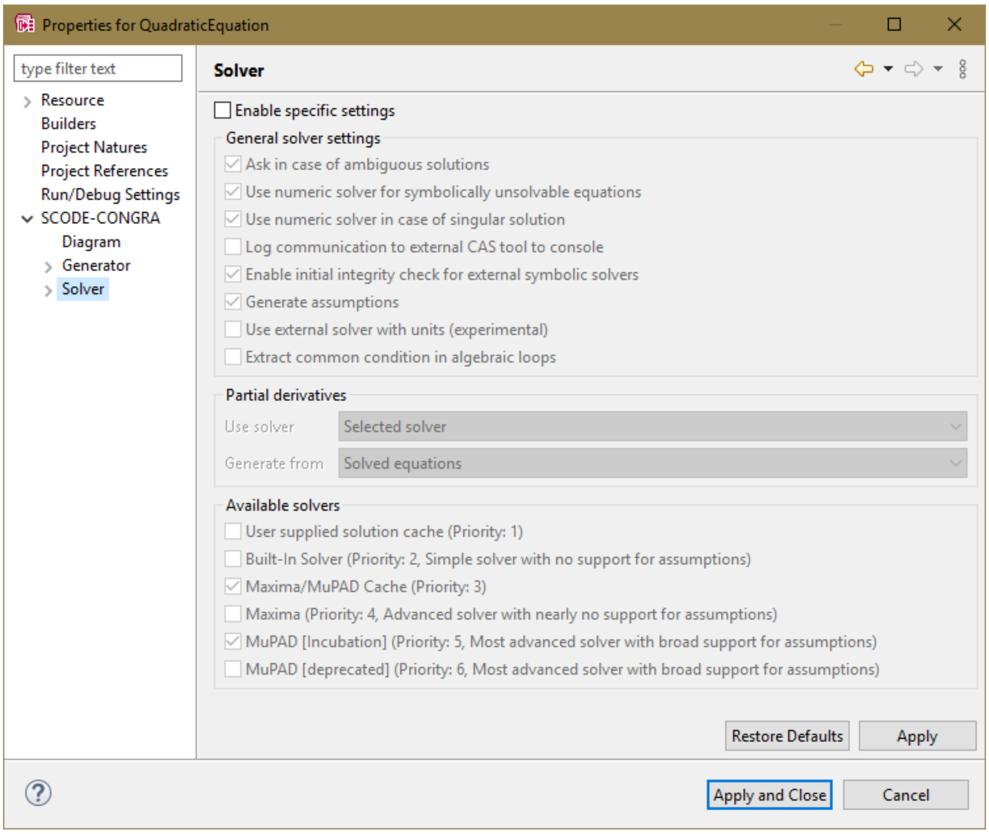


Figure 46. "Properties for <*project*>" window, "Solver" node

4. Activate **Enable project specific settings**.

   With that, the project-specific settings become available. They override the workspace settings.

5. In the "Use solver" combo box, select `Selected solver`.

   With that, the solver selected in the "Available solvers" area is used. If you select several solvers, the one with the highest priority is used. [14] If that solver is unable to solve the equation, the next one is used.

6. In the "Available Solvers" area, do the following:

   i. Activate **Maxima/MuPAD Cache** or **MuPAD [Incubation]** or **MuPAD [deprecated]**. [15]

      If MuPAD is unavailable, use **Maxima** instead.

   ii. Deactivate solvers with higher priority.

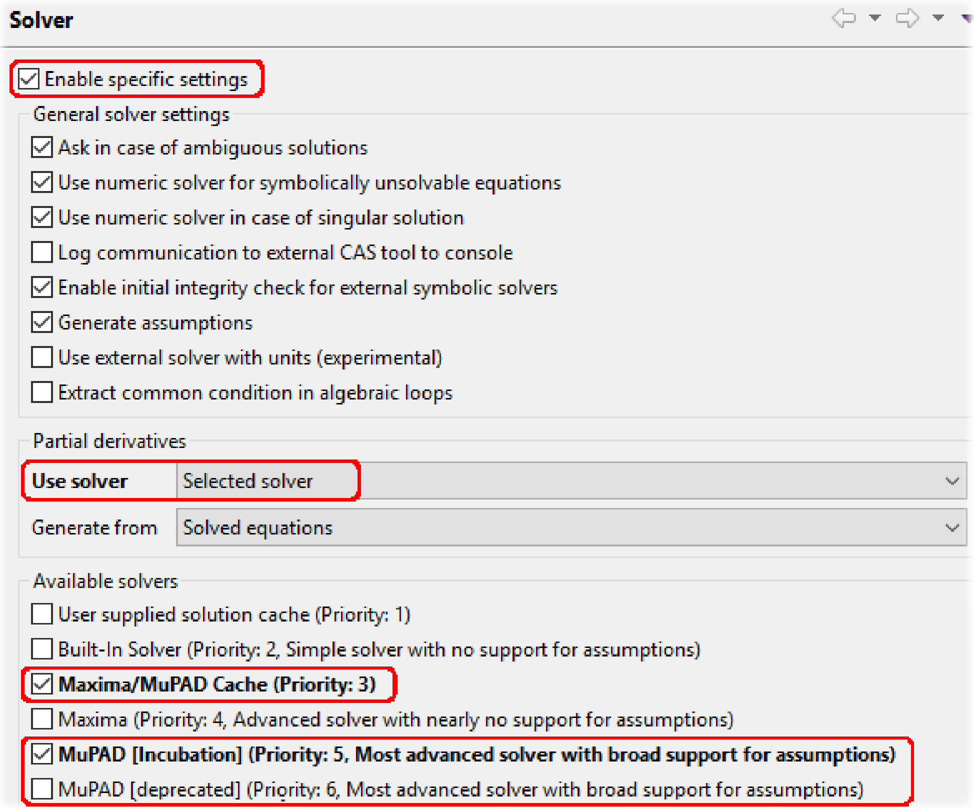


Figure 47. "Solver" settings for a project with quadratic equation (project settings that differ from workspace settings appear in bold font)

7. Click **Apply and Close**.

5.3.2. Equation System and Computation

After configuring the solver in the previous section, you will now solve the system. If there are multiple valid solutions for a given system, the solver may need user input to pick the correct solution for a given system. Here is an example of a system that will need user input.

To create the system for the quadratic equation

1. Specify the equation for the power of an ohmic resistor. [16]
2. Create a flow with inputs P and R. [17]

   With that, the remaining variable I is treated as an output.
3. Name the flow `F_QuadraticEquation_in_PR`.
4. Save the system.

   The equation is quadratic, i.e. it has two solutions, – $(P/R)^{1/2}$ and $(P/R)^{1/2}$. When you save the system, the "Please pick solution for request" window opens, which offers the possible solutions for selection.

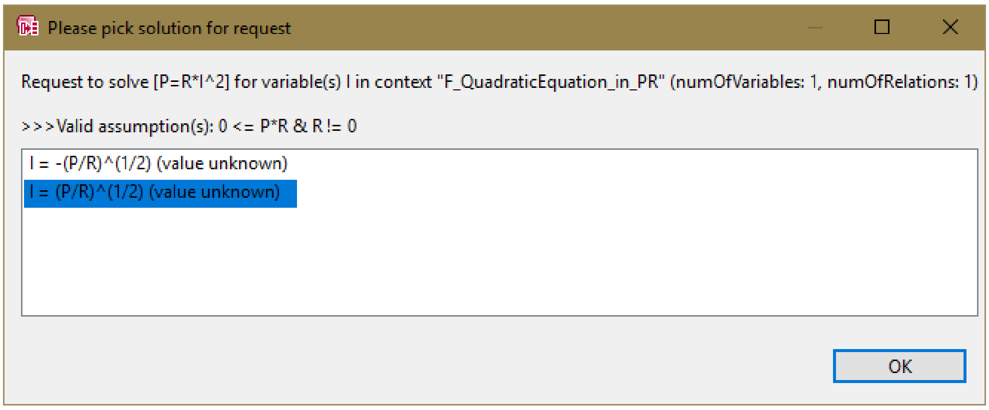


Figure 48. "Please pick solution for request" window with possible solutions for the quadratic equation example

5. Select one solution and click **OK**.

> **NOTE**
>
> The selection is not saved. The next time you save the project, or generate code, you are asked to pick a solution again.

A computation is created for the `F_QuadraticEquation_in_PR` flow.

The selected solution is written to the `*.syq` file of the computation (line 15 in Table 6):

```
1    /**
2     * @warning    AUTOMATICALLY GENERATED FILE! DO NOT EDIT!
...  ...
9     **/
10    package Quadratic_Equation;
11
12    computation c_F_QuadraticEquation_in_PR (R, P)
                  implements Quadratic_Equation
                  from F_QuadraticEquation_in_PR {
13      // Variable computation for level 2
14      @level(2, 1)
15      I = if ((0 <= P*R) && (R!=0)) then (P/R)^(1/2)
            else  <- R01(P, R); //  [Source: MuPAD [Incubation]]
16      [I,P] = if ((0 <= P*R) && (R!=0)) then
          if (((0.0!=R) && (0.0 <= P/R))
              && (((0.0!=R) && (0.0!=P/R))))
          then 1/(2*R*(P/R)^(1/2))
          else
        else  <- R01(P, R);  //  [Source: MuPAD [Incubation]]
17      [I,R] = if ((0 <= P*R) && (R!=0)) then
           if (((0.0!=R) && (0.0 <= P/R))
               && (((0.0!=R) && (0.0!=P/R))))
           then -P/(2*R^2*(P/R)^(1/2))
           else
        else  <- R01(P, R);  //  [Source: MuPAD [Incubation]]
18   }
```

Table 6. `*.syq` file for the `c_ F_QuadraticEquation_in_PR` computation

Check values and sensitivities in the Execution Environment. [18]

5.3.3. Additional Tasks

This section is not mandatory for the lesson on simple equations. However, it contains useful information.

Selecting Solutions

By default, the selected solution (cf. Figure 48) is not saved. The next time you save the project, or generate code, the "Please pick solution for request" window opens again. If desired, you can disable the question, or you can store the selected solution.

To disable the request to select a solution

1. Open the "Properties for <*project*>" window and go to the "Solver" node.
2. In the "Solver" node, deactivate **Ask in case of ambiguous solutions**.
3. Click **Apply and Close**.

   The next time code is generate, the "Please pick solution for request" window is suppressed, and SCODE-CONGRA uses the first of the possible solutions.

To store a selected solution

1. Open the "Properties for <*project*>" window and go to the "Solver" node.

   NOTE Settings in the "Properties for <*project*>" window apply only to the current project. For workspace-specific settings, use the "Preferences" window.

2. In that node, ensure the following:
   - **Enable specific settings** is activated.
   - In the "Use solver" combo box, `Selected solver` is selected.
   - Only the **Maxima / MuPAD Cache** solver and the **MuPAD [Incubation]** solver are selected.
3. Go to the "Maxima / MuPAD cache" node and do the following:
   i. Activate **Enable specific settings**.
   ii. Activate **Activate Learning Mode**.

   With that, SCODE-CONGRA adds new solved equations to the cache, so that the database of solved equation is growing. Solved equations are added only if they are unknown to SCODE-CONGRA.

   iii. If desired, enter path and file name (including the extension `*.xml`) for the "Internal Solver Cache File".

   By default, the project-specific cache file (named `internal_cache.xml`) is placed in the project's root folder.

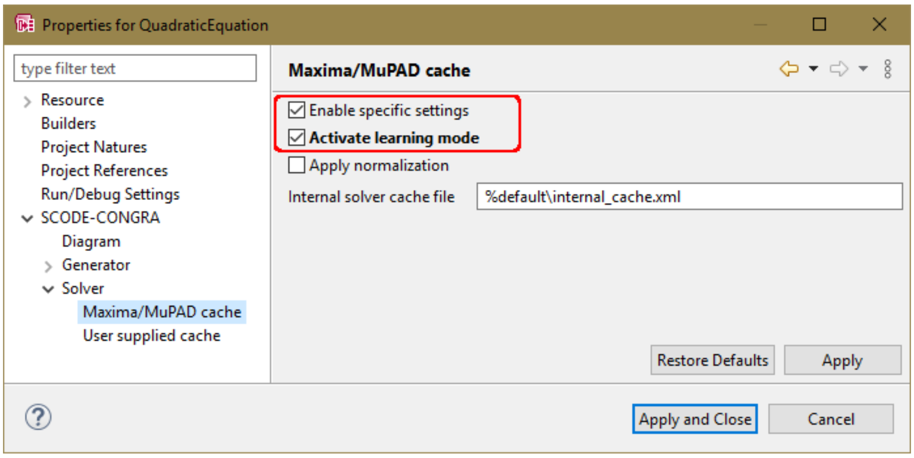


Figure 49. "Properties for <*project*>" window, "Maxima / MuPAD cache" node

4. Click **Apply and Close**.

   The next time code is generated, you are asked to select a solution. That solution is written to the cache file in the project folder.

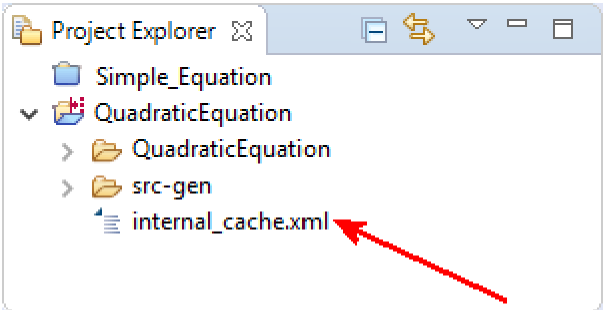


Figure 50. Project Explorer with project-specific cache file

Closing Projects

By default, code is generated for all open projects (marked with the icon) in the workspace. This may result in many "Please pick solution for request" windows, which can be annoying.

To close projects

1. Right-click the project you are working on and select **Close Unrelated Projects** from the context menu.

   All projects not related to the selected one are closed. They are marked with the icon.

Or

2. Right-click a project you want to close and select **Close Project** from the context menu.

   The selected project is closed and marked with the icon.

The next time code is generated, the closed projects are ignored.

5.4. Lesson 3: Constants, Parameters, Fixed Variables

In this lesson, you will learn how to use constants, parameters, and fixed variables. You will use Ohm's law again (see Equation 1), and compute U from R and I. Use the resistance R as constant, parameter, or fixed variable.

5.4.1. Constants

Constants store values that can be used in the model. Unlike parameters, constants cannot be changed from outside the model; they are fixed at specification time.

A constant does not appear in the system graph or in flow graphs. When code is generated, e.g., C code, the value of the constant is entered; there is no reference.

To set up the project

1. Create a SCODE-CONGRA project and name it, e.g., `Constants`. [19]
2. Specify the relation for Ohm's law.[16]

Optional:

3. To enter a value for R, do the following:
   i. In the system graph, select R.
   ii. Open the "Properties" view for R.
   iii. In the "Properties" view, "Semantic" node, click in the "Value" column next to "Expression".

      The cell becomes an input field.

   iv. Enter the value and press Return.

      If you do not enter a value here, you have to enter it later in the `*.syq` file (see step 4 in the next instruction).

You cannot create a constant directly; you have to convert an existing variable. You can create a constant graphically either in the system graph or in a flow.

To create a constant

1. To create a constant in the system graph, do the following:
   i. Open the system graph.

      Figure 32 shows an image of the relation.

   ii. Right-click the variable R and select **Set Type > Constant** from the context menu.

Or

2. To create a constant in a flow, do the following:
   i. Create and open a flow.
   ii. In the flow, right-click the variable R and select **Set Type > Constant** from the context menu.

   R is converted to a constant. It is no longer visible in the system graph (left) or flow graph (right).

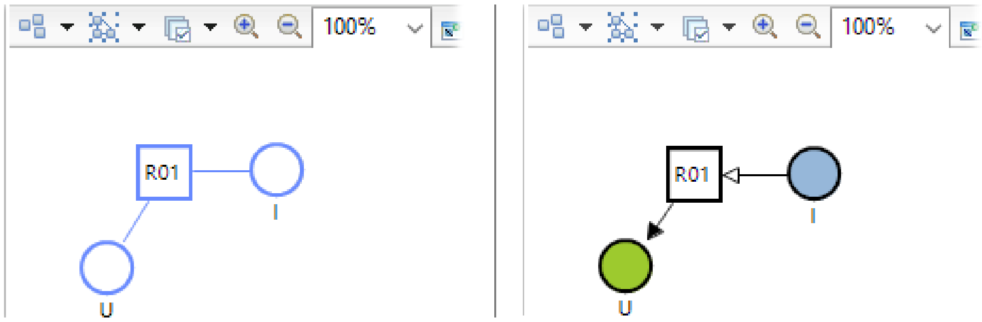


Figure 51. Constant R invisible in the system graph (left) and in the flow (right)

3. Save the project.

   The definition of R in the `*.syq` code changes as follows:

```
const R = 0;
```

   The definition of the relation changes as follows:

```
R01(I, U) ::= U = R * I;
```

   The input list of the existing flow changes as follows:

```
flow F_Constants_in_I for Constants {
    inputs: I;
}
```

4. If necesssary, open the `*.syq` file and enter an appropriate value in the definition of R.

   This example uses `R = 2`.

5. Save the project again.

To convert a constant into a variable, you have to edit the `*.syq` file as follows:

	R as constant	R as variable
declaration	`const R = 2.0;`	`var R = 2.0;`
relation	`R01(I, U) ::= U = R * I;`	`R01(I, R, U) ::= U = R * I;`

flow	flow F_*<flow_name>*
	    for *<system>* {
	  inputs: I;
	}
		flow F_*<flow_name>*
		    for *<system>* {
		  inputs: I, R;
		}

Table 7. Changes in `*.syq` file to convert a constant into a variable

The computation `*.syq` file contains the value of the constant, see lines 15 and 16 in Table 8.

```
/**
 ...
 **/

 package Constants;

 computation c_F_Constants_in_I(I) implements Constants
                           from c_F_Constants_in_I {
    // Variable computation for level 2
    @level(2, 1)
    U = 2*I <- R01(I);  //   [Source: Maxima]
    [U,I] = 2 <- R01(I);  //   [Source: Maxima]
 }
```

Table 8. `*.syq` file for a computation with a constant

In the Execution Environment, the value of R appears in the definition and partial derivative of the output U.

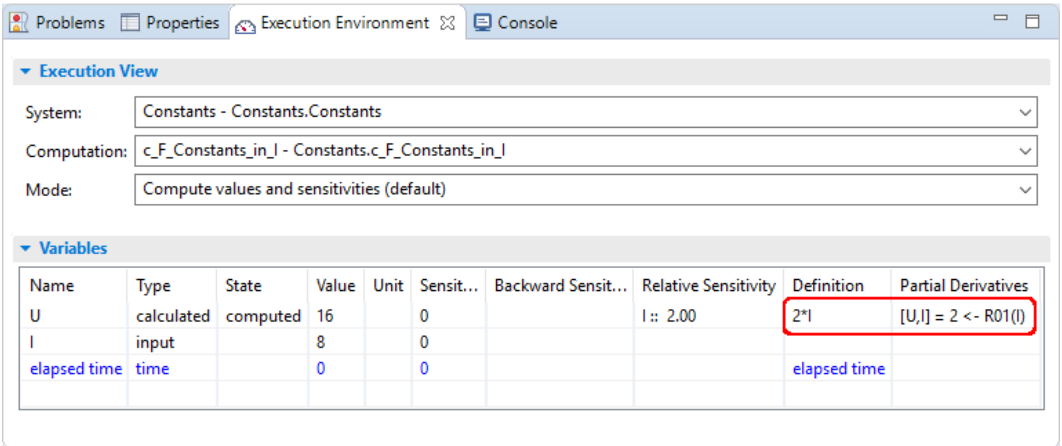


Figure 52. Execution Environment showing a computation with a constant

5.4.2. Parameters

Like constants, parameters store values that can only be read from inside the model. Unlike constants, parameters can also be calibrated, i.e. written to from outside the model. The idea is that parameters will be flashed on the car by an application engineer.

To set up the project

1. Create a SCODE-CONGRA project and name it, e.g., `Parameters`.
2. Specify the relation for Ohm's law.

Optional:

3. Enter a value for R. [20]

   This example uses `R = 15`.

You cannot create a parameter directly; a parameter is created by converting an existing variable. You can create a parameter graphically either in the system graph or in a flow.

<u>To create a parameter</u>

1. To create a parameter in the system graph, do the following:
    i. Open the system graph.

       Figure 32 shows an image of the relation.

    ii. Right-click the variable R and select **Set Type > Parameter** from the context menu.

Or

2. To create a parameter in a flow, do the following:
    i. Create and open a flow.
    ii. In the flow, right-click the variable R and select **Set Type > Parameter** from the context menu.

       R is converted to a parameter. It is shown as a grey circle with black frame () in the system graph (left) or flow graph (right).

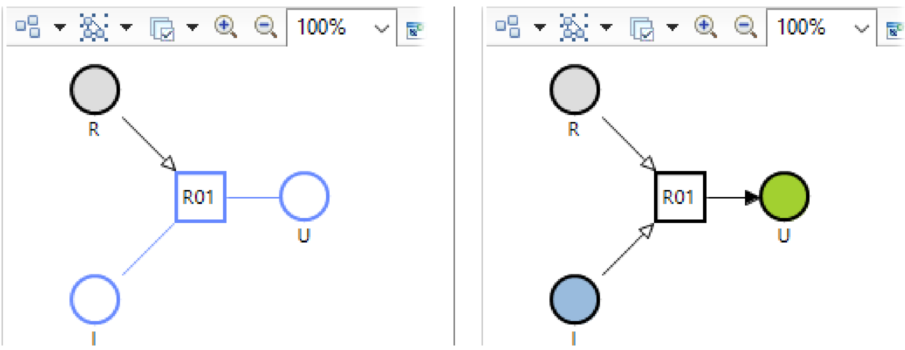


Figure 53. Parameter R in the system graph (left) and in the flow (right)

3. Save the project.

   The definition of R in the `*.syq` code changes as follows:

```
param R = 15;
```

   The definition of the relation does not change.

```
R01(U, I, R) ::= U = R * I;
```

   The input list of the existing flow changes as follows:

```
flow F_Constants_in_I for Constants {
    inputs: I;
}
```

4. Enter a value for R.

   You can do so either in the `*.syq` file or via the system graph or flow graph, as described in step 3 of To set up the project.

5. Save the project again.

To convert a parameter into a variable, you can do the following:

- Use the **Set Type > Variable** option in the parameter's context menu and correct the flows in the `*.syq` file.
- Enter all required changes (see the right column in Table 7) directly in the `*.syq` file.

The computation `*.syq` file refers to the parameter; see lines 16 — 18 in Table 9.

```
1     /**
...    ...
8      **/
9
10     package Parameters;
12
13     computation c_F_ Parameters_in_I(I) implements
                    Parameters from c_F_ Parameters_in_I {
14        // Variable computation for level 2
15        @level(2, 1)
16        U = I*R <- R01(I,R);  //   [Source: Maxima]
17        [U,I] = R <- R01(I,R);  //   [Source: Maxima]
18        [U,R] = I <- R01(I, R);  //   [Source: Maxima]
19     }
```

Table 9. `*.syq` file for a computation with a parameter

In the Execution Environment, the parameter R appears in a separate row and in the marked attributes of the output U.

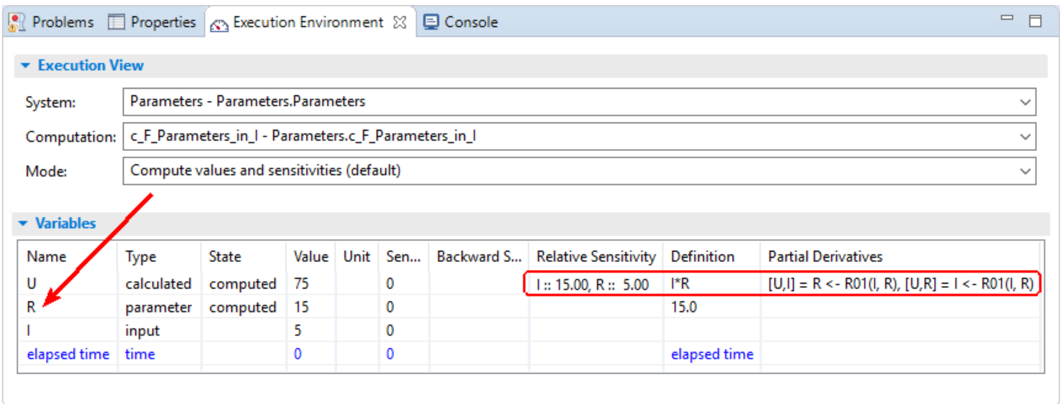


Figure 54. Execution Environment showing a computation with a parameter

5.4.3. Fixed Variables

Parameters and constants are declared as such in the system. Their properties are the same throughout the system; you cannot change them only for a particular flow.

If you need a constant value only in a particular flow, you can fix a variable in that flow. With that, the flow can no longer change the value of the variable. In all other flows, the variable is still a variable that can be read and written.

To set up the project

1. Create a SCODE-CONGRA project and name it, e.g., `FixedVariable`.
2. Specify the relation for Ohm's law.

   Figure 32 shows an image of the relation.

A variable can be fixed only in a flow.

To fix a variable in a flow

1. Create and open a flow.
2. In the flow, right-click the variable R and select **Set Type > Fixed** from the context menu.

   R is fixed. It is shown as a grey circle with black frame (◯) in the flow graph (left side of Figure 55). In the system graph (right), R remains unchanged.

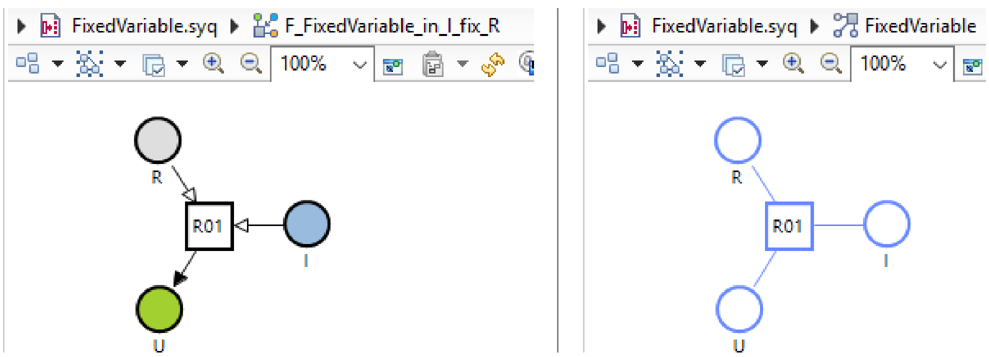


Figure 55. Fixed variable R in the flow (left) and in the system graph (right)

3. Enter a value for R:
   i. In the flow, select R.
   ii. Open the "Properties" view for R.
   iii. In the "Properties" view, "Semantic" node, enter a value in the "Value" column next to "Expression".
4. Save the project.

   The definition of R and the definition of the relation in the `*.syq` code remain unchanged:

```
var R = 0;
R01(U, I, R) ::= U = R * I;
```

   The definition of the flow changes as follows:

```
flow F_FixedVariable_in_I_fix_R for FixedVariable {
     inputs: I;
     fixed: R = 5;
}
```

   Other existing flows remain unchanged.

To convert a fixed variable into a variable, use the **Set Type > Free** option in the fixed variable's context menu.

The computation `*.syq` file refers to the fixed variable. Lines 14 and 15 in Table 10 determine the fixed variable. In the following rows, R appears like other variables.

```
1     /**
...    ...
8      **/
9
10     package FixedVariable;
11
12     computation c_F_FixedVariable_in_I_fix_R(I)
                   implements FixedVariable from
                   F_FixedVariable_in_I_fix_R [
13        // Constant input and intrinsic BNS initialization
14        @level(0, 1)
15        R = 5;
16     ]
17     {
18        // Variable computation for level 2
19        @level(2, 1)
20        U = I*R <- R01(I,R);  //   [Source: Maxima]
21        [U,I] = R <- R01(I,R);  //   [Source: Maxima]
22        [U,R] = I <- R01(I, R);  //   [Source: Maxima]
23     }
```

Table 10. `*.syq` file for a computation with a fixed variable

In the Execution Environment, the fixed state of variable R is marked by the entries in the "Type" and "State" columns.

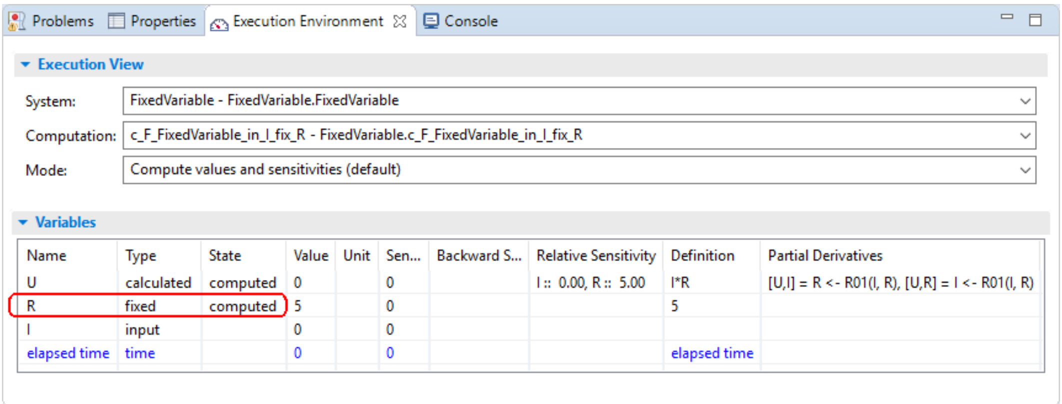


Figure 56. Execution Environment showing a computation with a fixed variable

5.4.4. Generating Code

Next, you will generate C code, MATLAB code, and ESDL code for the three projects of this lesson.

To select code generators

1. In the SCODE Workbench window, select **Window > Preferences**.

   The "Preferences" window opens.

2. In the "Preferences" window, expand the "SCODE-CONGRA" node and go to the "Generator" subnode.

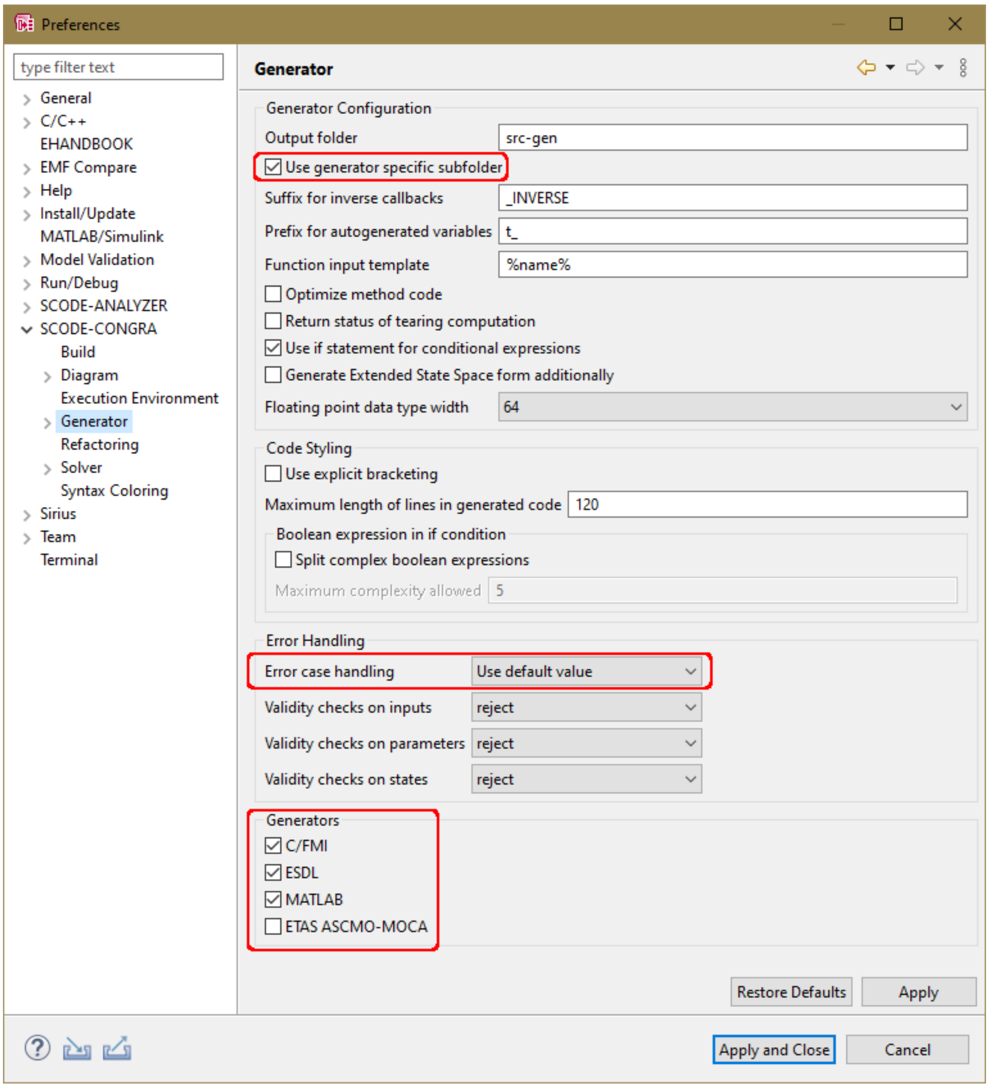


Figure 57. "Preferences" window with "Generator" settings for SCODE-CONGRA

3. In the "Generators" area, activate the generators you want to use.
4. If you selected the **ESDL** generator, select one of the `Use *` entries in the "Error case handling" combo box.

   This example uses `Use default value`.

> **NOTE**
>
> The `Report error/abort execution` error case handling is not allowed in combination with ESDL code generation.
>
> Depending on your selection, you have to provide default values or upper/lower limits for each variable.

5. If desired, activate the **Use generator specific subfolder** option.

   With that, code generated for each generator is stored in its own folder below the code generation folder.
6. Click **Apply and Close**.

The next time code is generated, the computation and code for the selected generators are generated.

To generate code

1. If desired, close projects you do not need, as described in To close projects.
2. In the Project Explorer, right-click one of the following items and select **Generate Code** from the context menu.

project	(e.g., FixedVariable)
system folder	(e.g., FixedVariable)
`*.syq` file	(e.g., FixedVariable.syq)

   Code is generated for each selected generator.

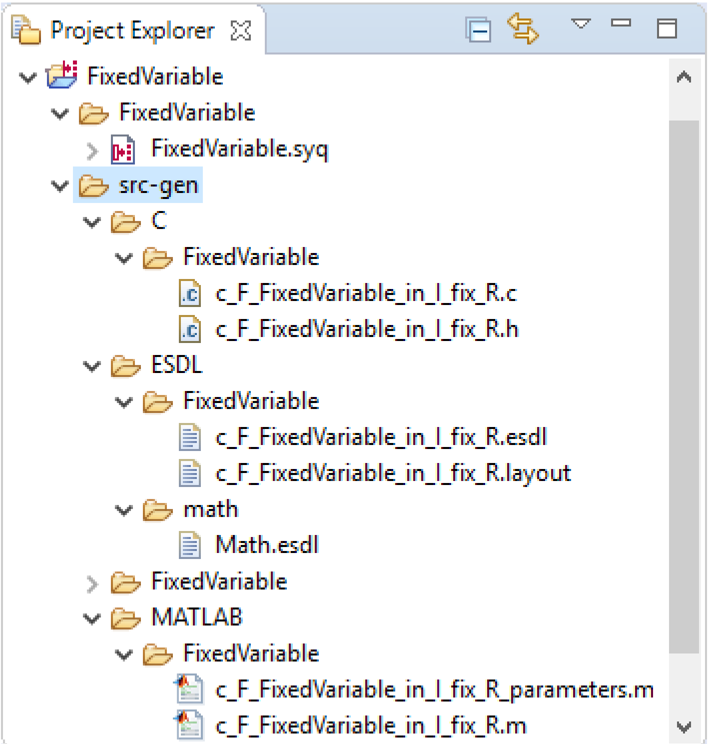


Figure 58. Code generation folder for the `FixedVariable` project, with generated C, ESDL, and MATLAB files

3. Open the generated files and look at the differences between constants, parameters, and fixed variables.

In section 9.2.1, “C Code for Lesson 3”, you can find generated C code for this lesson, and the following tables briefly explain the generated files.

Table 11 lists the files generated during C code, ESDL, and MATLAB code generation. Table 12 shortly describes the content of the files.

	C Code	ESDL	MATLAB
folder **src-gen\\<*generator*>\\<*system*>**			
for each flow	c_<*flow*>.c c_<*flow*>.h	c_<*flow*>.esdl c_<*flow*>.layout	c_<*flow*>.m c_<*flow*>_parameters.m
others	libscode.a scode.h	---	---
folder **src-gen\\<*generator*>\\math**			
others	---	Math.esdl	---

Table 11. Files generated during C, ESDL, and MATLAB code generation

	file	content
All types	c_<*flow*>.*	Everything required to execute the <*flow*>.
C code	libscode.a	Library with standard implementations of service routines supplied by SCODE-CONGRA.
	scode.h	Header file required for libscode.a.
ESDL	c_<*flow*>.layout	Layout definitions for the <*flow*>. These can be used in ASCET-DEVELOPER.
MATLAB	c_<*flow*>_parameters.m	Definitions for all parameters in the <*flow*>.

Table 12. Content of the files generated during C, ESDL, and MATLAB code generation

5.5. Lesson 4: Inverting Models

In this lesson, you will create a system with two connected relations. You will use the system to invert a model.

The first relation is Ohm's law (see Equation 1), the second is the power of an Ohmic resistor:

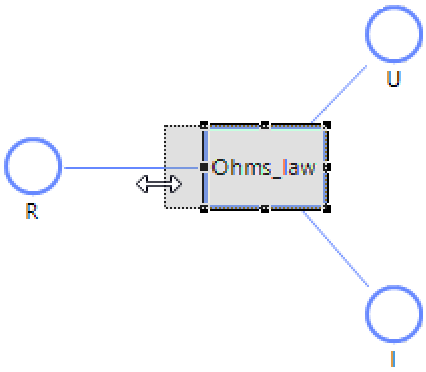
Equation 4: Power of an ohmic resistor (P -- electrical power in watt, U -- voltage in volts, I -- current in amperes)

```
P = U * I
```

For this lesson, MuPAD is recommended as solver. If you cannot use the MuPAD solver, use the Maxima solver, which is shipped with SCODE-CONGRA. For more details, see section 5.3.1, "Preparing the Project".

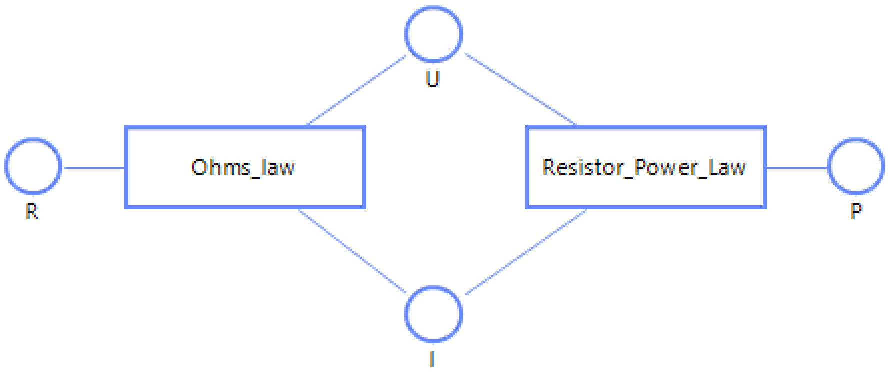
To set up the project

1. Create a SCODE-CONGRA project and name it, e.g., Resistor_Power.
2. Specify the first relation for Ohm's law and name it, e.g., Ohms_law.
3. Use the handles to resize the relation.



4. Specify the second relation for the power of an ohmic resistor (Equation 4) and name it, e.g., `Resistor_Power_Law`.

   The second relation is automatically connected to the existing variables I and U.



5. Enter default values for the variables.
6. Save the project.

To specify original flow and inverted flow

1. Create a flow that uses R and U to compute I and P.

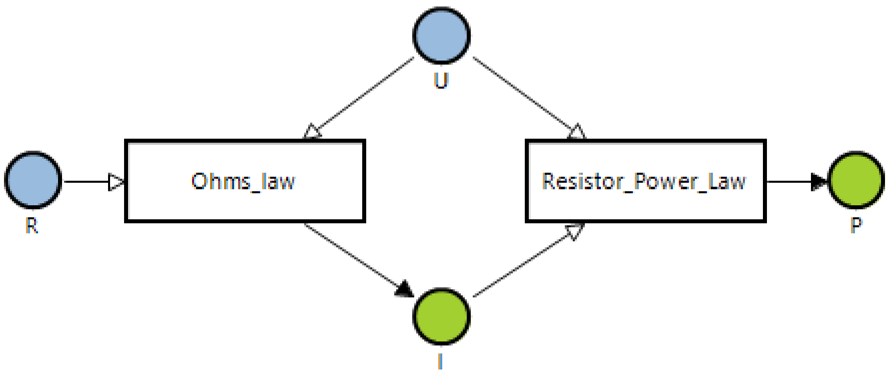


Figure 59. Flow with original direction

U is used as input for both relations. Therefore, two white-headed arrows point from U to the relations.

I is the output of the first relation, and an input of the second relation. A black-headed arrow points from `Ohms_law` to I, and a white-headed arrow points from I to `Resistor_Power_Law`.

2. To invert the flow, create a second flow that uses I and P to compute R and U.

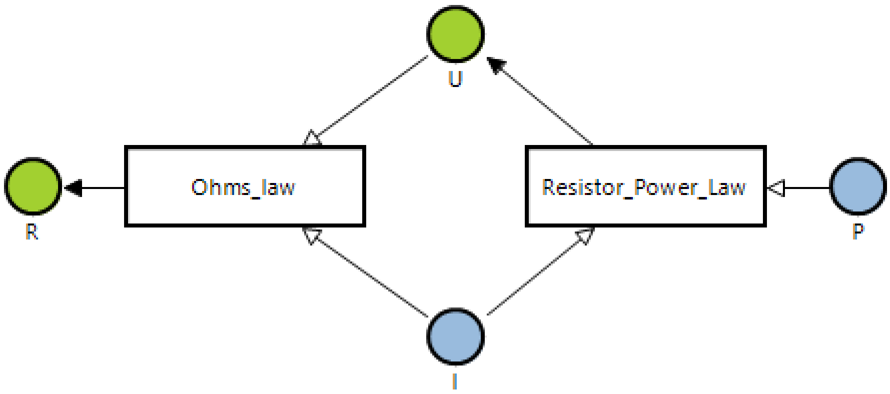


Figure 60. Flow with inverted direction

To generate code for original and inverted flows

In addition to the computations, you will generate C code.

1. In the Project Explorer, right-click the project and select **Properties** from the context menu.

   The "Properties for <*project*>" window opens.

2. In that window, do the following:

   i. Expand the "SCODE-CONGRA" node and go to the "Generator" subnode.

      By default, **Enable project specific settings** is deactivated; the project uses the workspace settings.

   ii. Activate **Enable project specific settings**.

      The project-specific settings become available. They override the workspace settings.

   iii. In the "Generators" area, activate the **C/FMI** generator.

      If you activate the **ESDL** generator, too, you have to select one of the `Use *` entries in the "Error case handling" combo box.

   iv. If desired, activate the **Use generator specific subfolder** option.

      With that, code generated for each generator is stored in its own folder below the code generation folder.

   v. Click **Apply and Close**.

3. Generate code. [21]

See section 9.2.2, “C Code for Lesson 4” for the generated code.

5.6. Lesson 5: Explicit Outputs

In the previous lessons, the outputs were determined automatically. In this lesson, you will use an explicitly defined output. With an explicit output, only the code needed to compute the output is generated. Model parts not necessary to compute the defined output are ignored.

To set up the project

1. Create a SCODE-CONGRA project and name it, e.g., `DefinedOutput`.
2. Create and specify the relations for Ohm's law and the power of an ohmic resistor (Equation 4).
3. If desired, name the relations, e.g., `Ohms_law` and `Resistor_Power_Law`.
4. Enter default values for the variables.
5. Save the project.

To define the output

1. Create a flow with inputs R and U.

   Since both I and P can be computed, both are filled with green; see Figure 59.

2. If desired, name the flow `F_DefinedOutput_in_RU_out_I`.
3. Right-click I and select **Set Type > Free and Output** from the context menu.

   I is now marked with a thick border; see Figure 61. Since the second relation and the variable P are irrelevant for computing I, they are marked with grey borders.

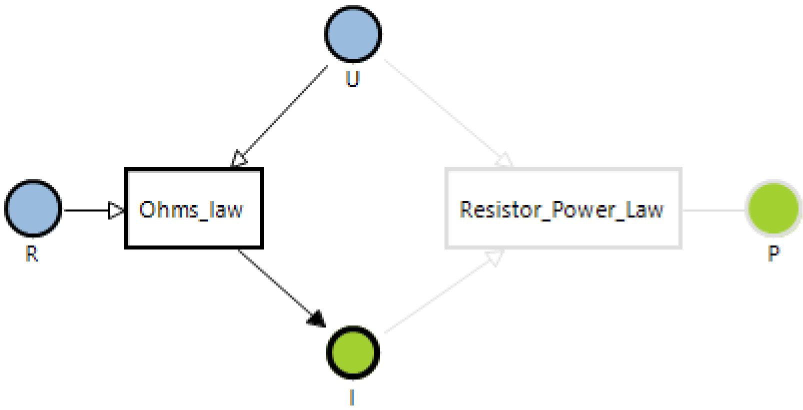
   

   Figure 61. Flow with inputs R and U and explicit output I. Irrelevant parts of the flow are marked.

4. Create a second flow with inputs R and U, without explicit output.
5. Save the project.

In the `*.syq` file, the two flows appear as follows:

```
flow F_DefinedOutput_in_RU_out_I for DefinedOutput {
     inputs: R, U;
     outputs: I;
}

flow F_DefinedOutput_in_RU for DefinedOutput {
     inputs: R, U;
}
```

To generate code with an explicit output

In addition to the computations, you will generate C code and ESDL code.

1. Open the "Properties for <*project*>" window and enable code generation for C and ESDL. [22]

2. Make sure that "Error case handling" is set to `Use default value`.
3. Generate code ().[21]
4. Compare the generated C code or ESDL code for both flows.

See section 9.2.3, “ESDL Code for Lesson 5” for the generated ESDL code.

5.7. Lesson 6: Algebraic Loop

In this lesson, you will use the same model as in lessons 4 and 5. This time, however, you will specify an algebraic loop, which can be solved by the underlying computer algebra system.

To set up the project

1. Create a SCODE-CONGRA project and name it, e.g., `AlgebraicLoop`.
2. Create and specify the relations for Ohm’s law and the power of an ohmic resistor (Equation 4).
3. Enter default values for the variables.
4. Open the "Properties for <*project*>" window and enable code generation for C, ESDL and MATLAB.
5. Save the project.

To define the flow with algebraic loop

1. Create a flow with inputs R and P.
2. If desired, name the flow `F_AlgebraicLoop_in_PR`.

   Both relations and the variables I and U form the algebraic loop. They, as well as the connections between them are shown with yellow borders.

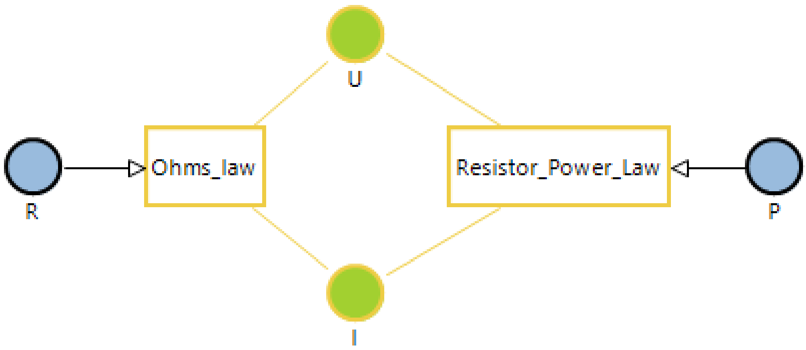
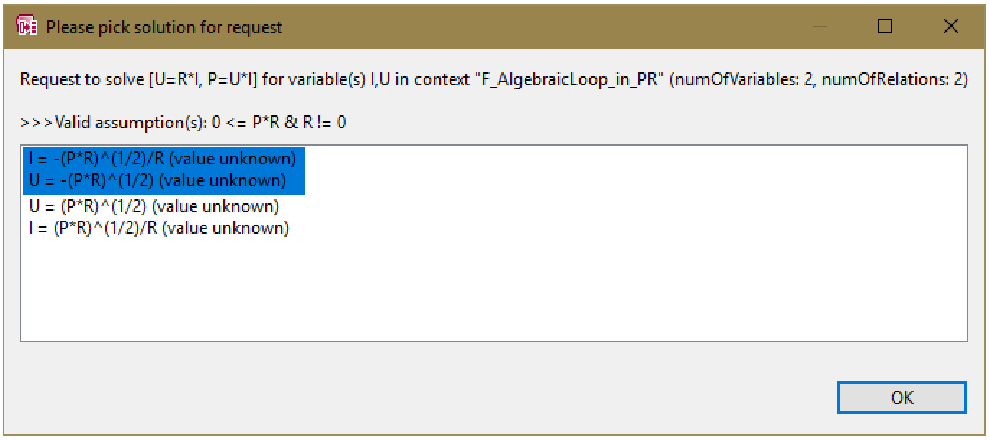
   

   Figure 62. Flow with algebraic loop

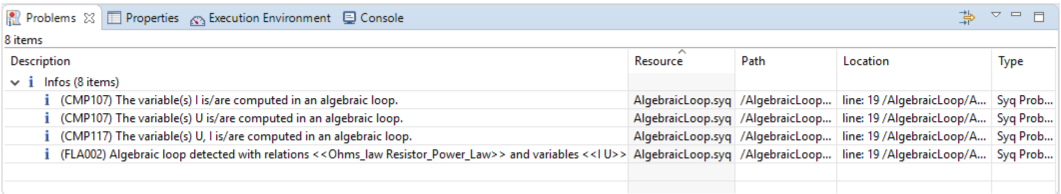
3. Save the project.

   The "Please pick solution for request" window opens.



4. Select one solution set and click **OK**.

   Code is generated for each selected generator. Messages regarding the algebraic loop are shown in the "Problems" view. [23]



See section 9.2.4, “Generated Code for Lesson 6” for several extracts from generated code for this lesson.

5.8. Lesson 7: Constraints and Verification

In the previous lessons, all variables, parameters, etc. were unconstrained. In this lesson, you will assign constraints to variables (section 5.8.1) and parameters (section 5.8.3). You will also activate the generation of verification code (section 5.8.2).

You will use the same model as in lessons 4 to 6.

Table 13 lists the available constraint types. To use several types, connect them with `and`.

<table>
<tr><th>Constraint Type</th><th>Meaning</th><th>Remarks</th></tr>
<tr><td><</td><td>less than</td><td rowspan="4">Allowed for variables and parameters.</td></tr>
<tr><td><=</td><td>less than or equal</td></tr>
<tr><td>>=</td><td>greater than or equal</td></tr>
<tr><td>></td><td>greater than</td></tr>
<tr><td>!=</td><td>not</td><td>Allowed only for parameters.</td></tr>
</table>

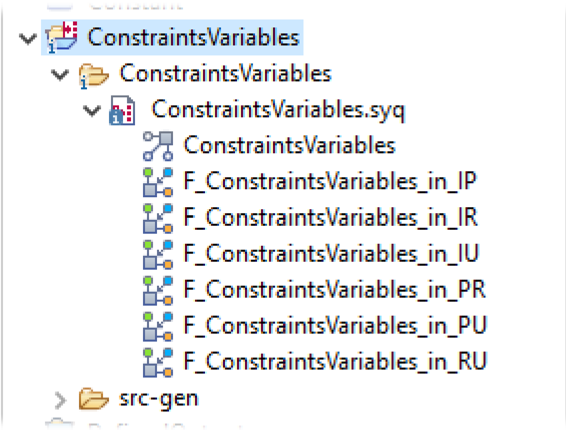
Table 13. Available constraint types

5.8.1. Constraints for Variables

Here, you will enter constraints for all variables.

<u>To set up the project</u>

1. Create a SCODE-CONGRA project and name it, e.g., `ConstraintsVariables`.
2. Create and specify the relations for Ohm's law (Equation 1) and the power of an ohmic resistor (Equation 4).
3. Create a flow for each possible input pair.



4. Enable code generation for C code.
5. Save the project.

<u>To enter constraints for the variables</u>

1. Open the `ConstraintsVariables` system in the graphical editor.
2. Open the "Properties" view for the variable R.
3. In the "Variable Constraints" row, "Value" column, enter the following constraint: `>0 and <100`

   The constraints are copied to the "Expression Range" row.

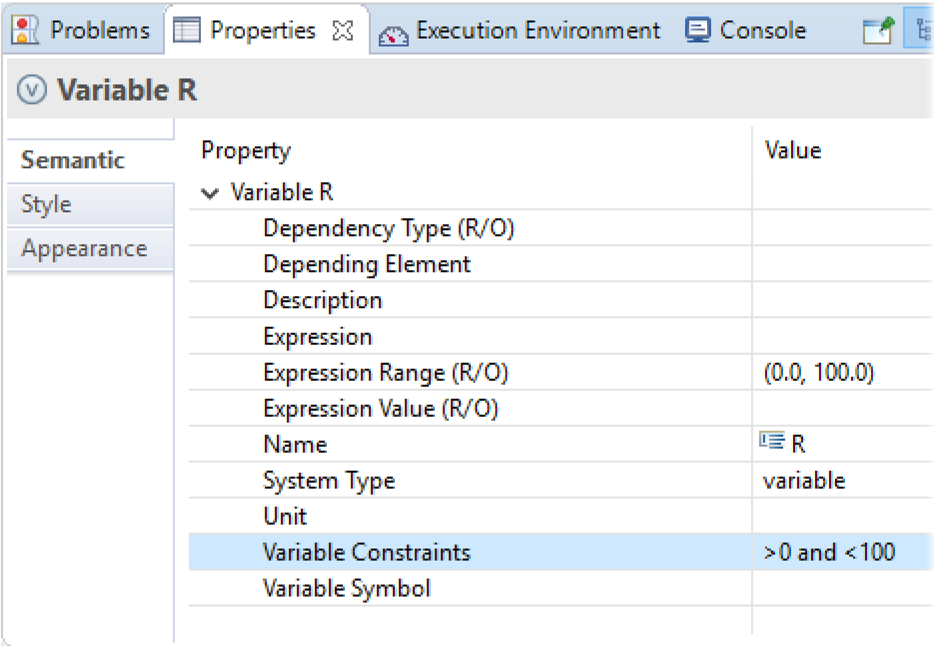


Figure 63. "Properties" view with constraints for a variable

4. For the other variables, enter the following constraints:

I	`>=0 and <10`
P	`>0 and <2500`
U	`>0 and <=230`

1. Save the project.

   The constraints are added to the `*.syq` file, see Table 14.

2. Generate code ( ).
3. Open the generated files and check the effects of the constraints.

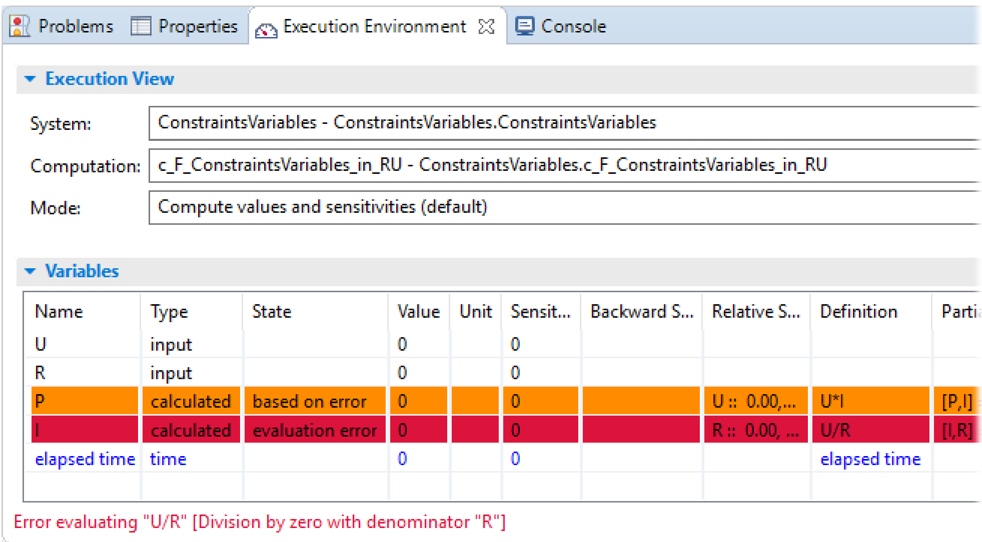
See section 9.2.5.1, “C Code for a Flow with Constraints” for a generated C code file.

```
1    package ConstraintsVariables;
2
3    system ConstraintsVariables {
4
5    @geo(312, 180)
6        var U is > 0 and <= 230;
7        @geo(84, 112)
8        var R is > 0 and < 100;
9        @geo(312, 40)
10       var I is >= 0 and < 10;
11       @geo(528, 112)
12       var P is > 0 and < 2500;
13
14       @geo(216, 111, 61, 32)
15       @description("Ohm\'s law")
16       Ohms_Law(R, I, U) ::= U = R * I;
17       @geo(408, 112, 90, 30)
18       @description("Power of a Resistor")
19       Resistor_Power_Law(P, I, U) ::= P = U * I;
20   }
...  ...
```

Table 14. `*.syq` file for the `ConstraintsVariables` system. Lines 6, 8, 10, and 12 show the constraints for the variables.

<u>To execute the computation</u>

1. Open the `c_F_ConstraintsVariables_in_RU` computation in the Execution Environment.



   If you did not specify start values for U and R, I has the state `evaluation error`, and P has the state `based on error`.

2. Enter values that are inside the limits for U and R.

   For example, enter `100` for U and `20` for R.

   The values for I and P are computed.

3. Now change the value of U to `220`.

The values of both U and R are still inside the respective constraint, but the resulting I = U/R exceeds the upper constraint of `10`. Therefore, I is limited to the upper constraint, and marked accordingly (see Figure 64).

P is marked, too, because its value is within the constraints of P, but is based on the limited value of I.

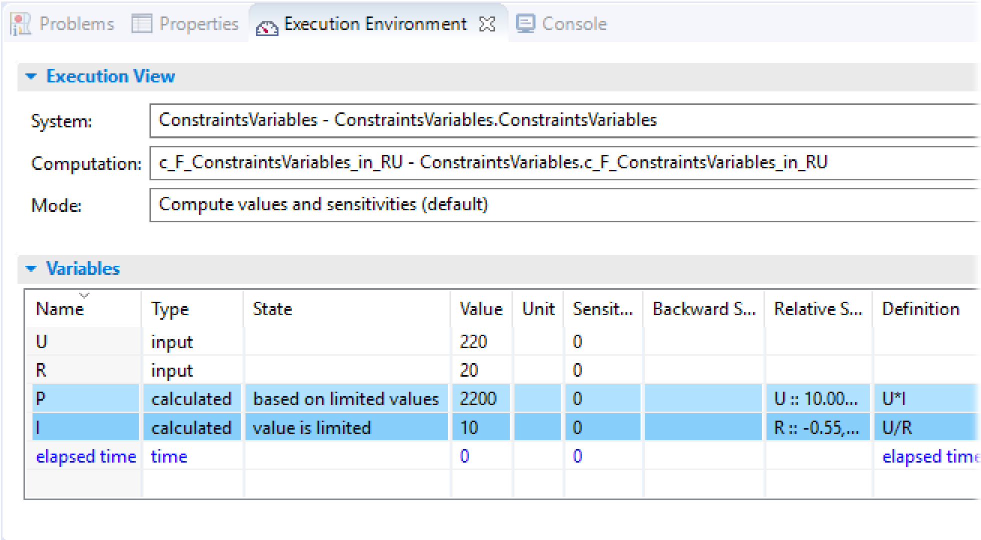


Figure 64. Execution Environment with a limited variable and a variable with a value based on the limited variable.

4. Now enter a value outside the constraints for R, e.g., `110`.

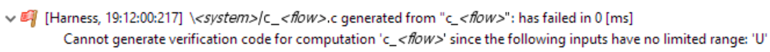
   This time, R itself is limited to its upper constraint, and marked accordingly. Both I and P are marked as based on limited values.

5.8.2. Verification Code

The verification code runs the code with varying input values. For each test run with a given set of input values, the resulting values are tested against the original equation from which the code was derived. For that purpose, the *normalized* equation is evaluated and its residue compared with a given verification threshold. [24]

NOTE

In order to run the verification code, all input variables must have upper and lower bounds. Otherwise, an error message appears in the "Build" view.

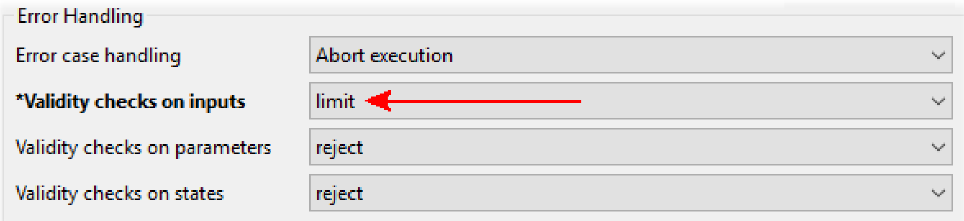


Verification code can be generated in a separate C file, the *verification harness*.

In the `ConstraintsVariables` project, all variables are constrained, and the precondition for verification code generation is met. Therefore, you will add the verification code to the `ConstraintsVariables` project.

To enable and generate verification code

1. Open the "Properties for <*project*>" window for the `ConstraintsVariables` project.
2. In the "Generator" node, activate the C/FMI generator.
3. If you did not specify start values for U and R, set the "Error case handling" to `Use upper limit` or `Use lower limit`.
4. If desired, set "Validity checks on inputs" to `limit`.



5. In the "Verification" subnode, activate **Generate Verification Code**.

   This enables the generation of verification code in a separate file, the *verification harness*.

6. If desired, activate also **Inform about Limitations**.

   This option is effective only if "Validity checks on inputs" is set to `limit`. It creates a text file `<flow_name>_<date>_<time>.txt` that lists all limitations.

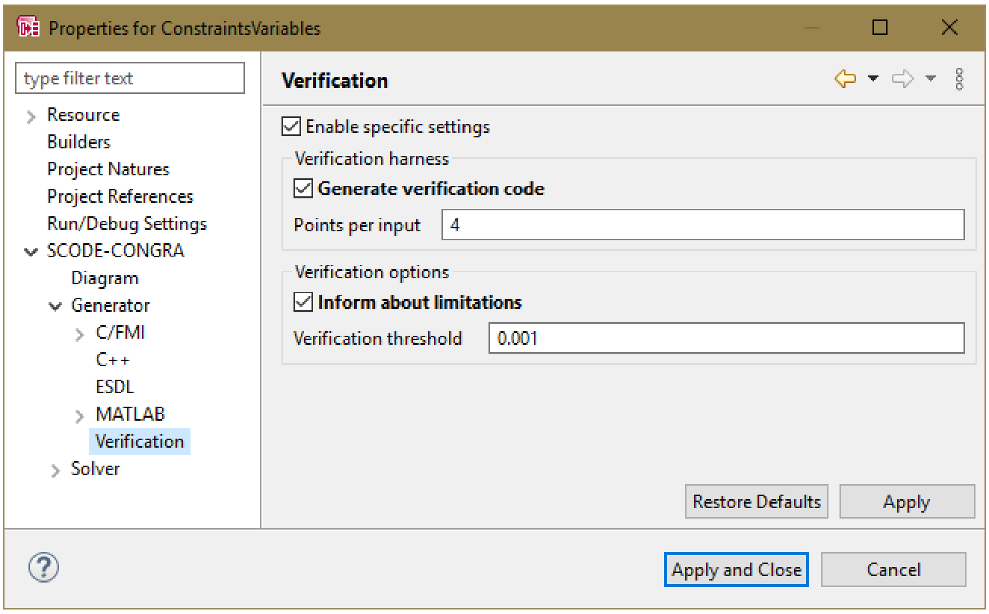


Figure 65. "Properties for <*project*>" window, "Verification" node

7. In the "C/FMI" subnode, activate the **Compile and verify code** option.

   With that, the generated verification harness is automatically compiled and executed.

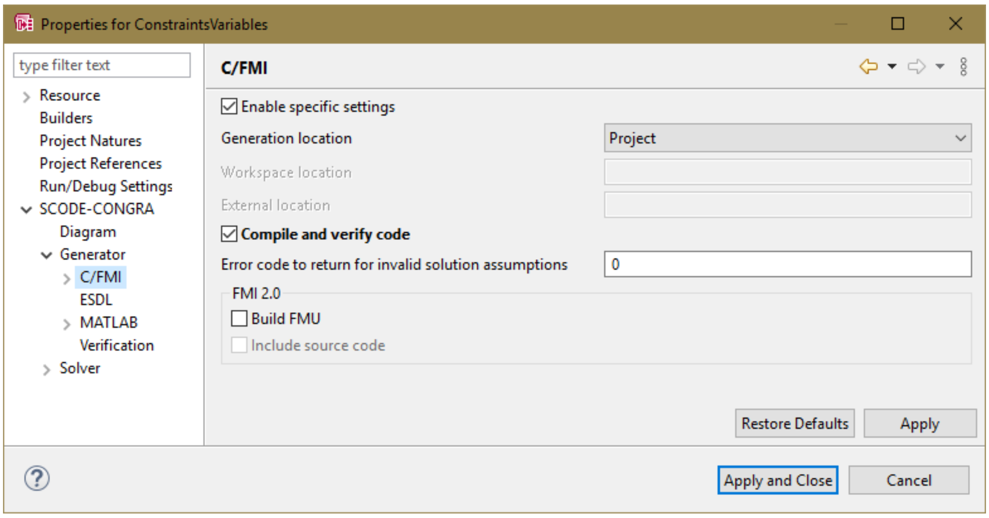


Figure 66. "Properties for *<project>*" window, "C/FMI" node

8. Save the project.
9. Generate code ( ).

The "Build" view summarizes the results. The verification harness issues one row for each flow, see the highlighted rows in Figure 67.

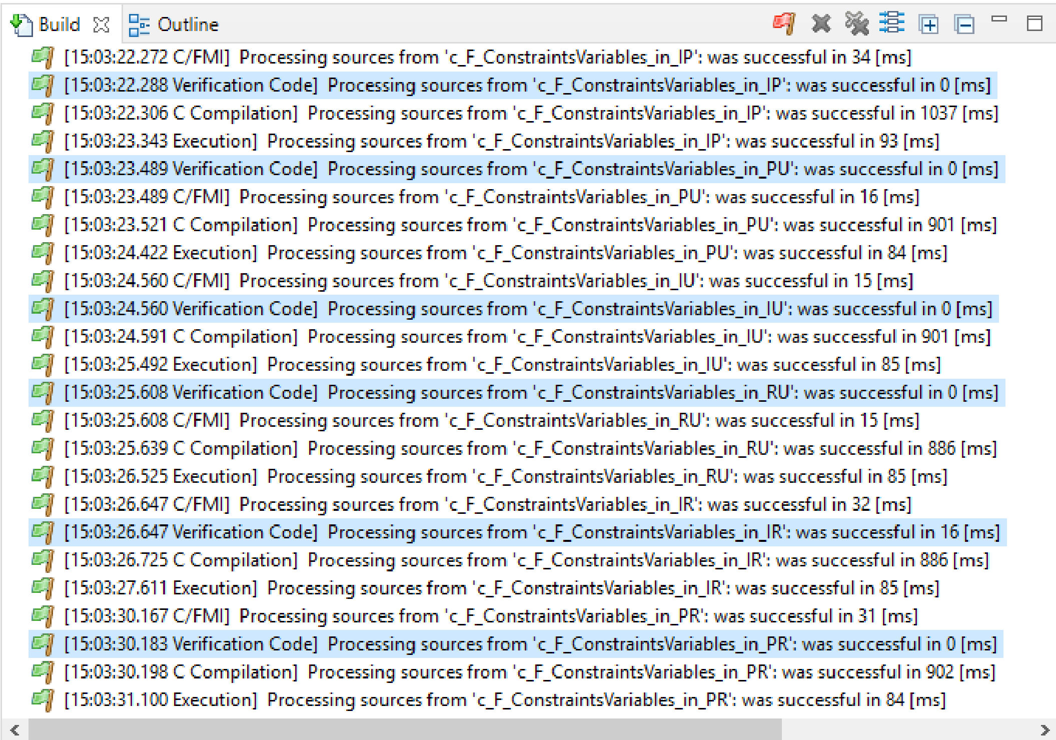


Figure 67. "Build" view with results for C code generation with verification harness

In addition to the usual files (see Table 11), a `c_<flow>_harness.c` file and a `c_<flow>_harness.h` file are created for each flow.

An example for such a file is shown in section 9.2.5.2.

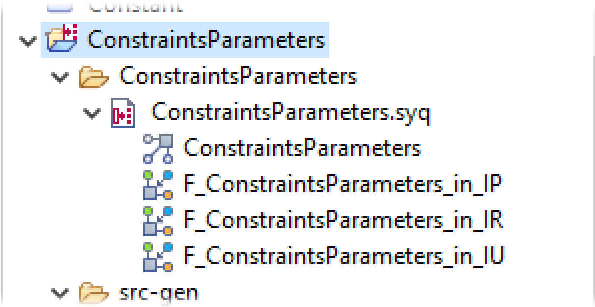
If you activated **Compile and verify code** in the "C/FMI" subnode, an executable file `c_<flow>_harness.exe` is created for each flow.

5.8.3. Constraints for Parameters

Here, you will create a parameter and enter constraints for the parameter.

To set up the project

1. Create a SCODE-CONGRA project and name it, e.g., `ConstraintsParameters`.
2. Create and specify the relations for Ohm's law and the power of an ohmic resistor (Equation 4).
3. Enter default values of 0.
4. Create a flow for each possible input pair that includes I.

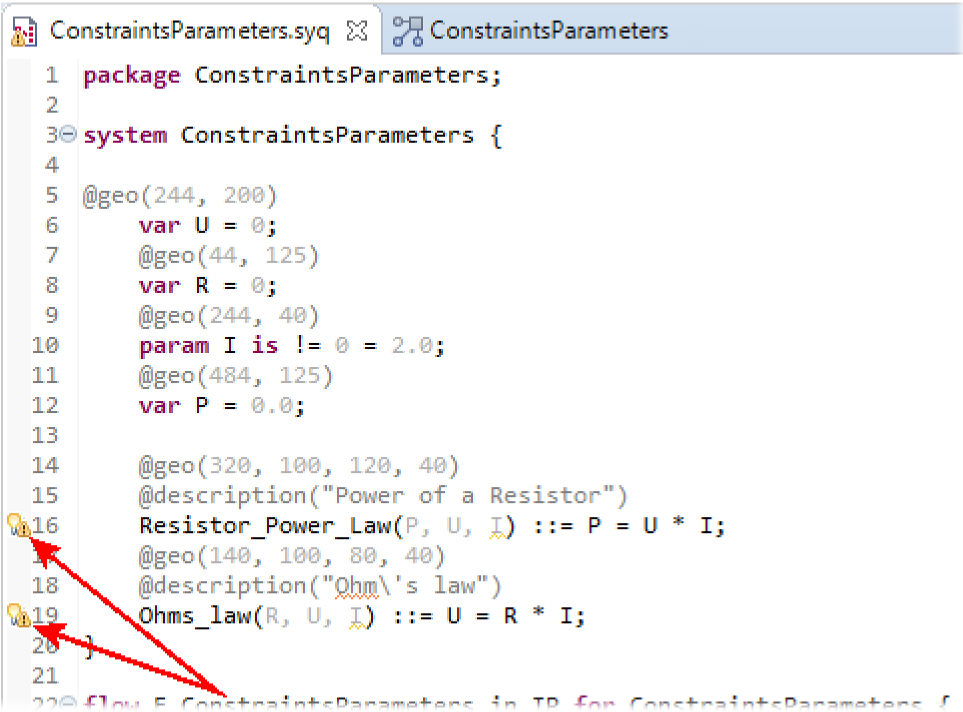


5. Enable code generation for C code and MATLAB code.
6. Set the "Error case handling" to `Use default value`.
7. Save the project.

To create and set up a parameter

1. Open the `ConstraintsParameters` system in the graphical editor.
2. Convert the variable I into a parameter and assign a default value.
3. Open the `*.syq` file in the text editor.

   Warning icons can be seen next to the definitions of the relations. The light bulbs show that quick fixes are available.



4. Click in one of the marked lines and press Ctrl + 1.

   A white box opens. It shows the available quick fixes. A yellow box may open on the right and show additional information.

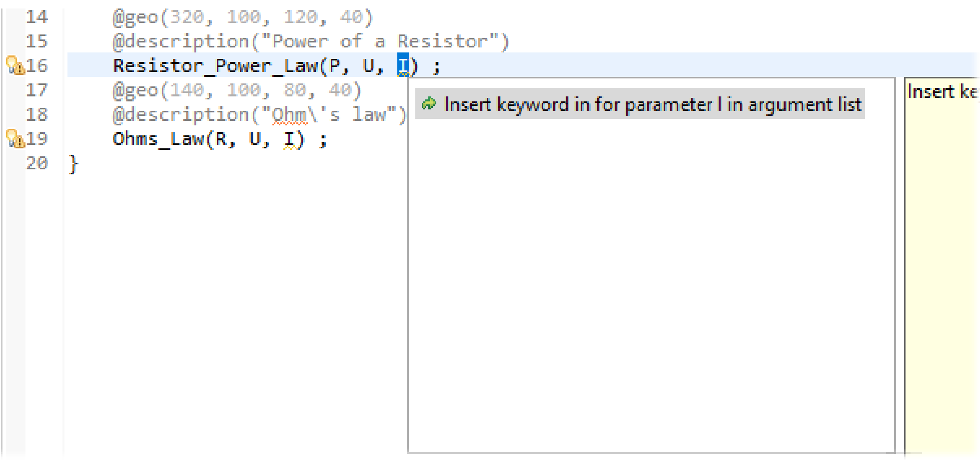
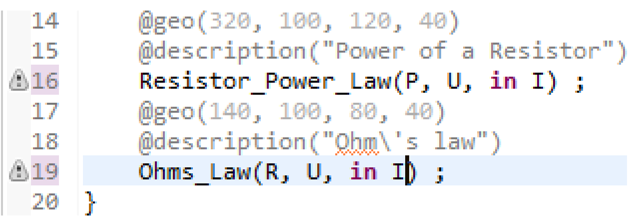


Figure 68. Pop-up with quick fix

5. Double-click the quick fix.

   I is marked as input.

6. Repeat steps 4 and 5 for the second marked line.



To enter constraints for the parameter

1. Open the "Properties" view for the parameter I and do the following:

   i. In the "Expression" row, "Value" column, set the parameter value to `2.0`.

   ii. In the "Variable Constraints" row, "Value" column, enter the following constraint: `!=0`

   The constraint is copied to the "Expression Range" row.

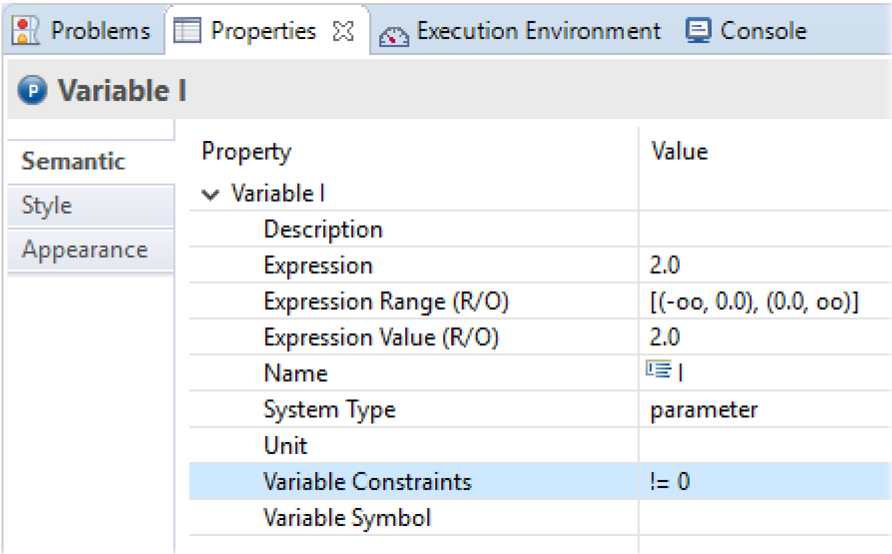
   

   Figure 69. "Properties" view with constraints for a parameter

   The other variables remain unconstrained. This means that verification code cannot be generated.

2. Save the project.

   Constraint (`!= 0`) and value (`= 2.0`) for parameter I are added to the `*.syq file.`

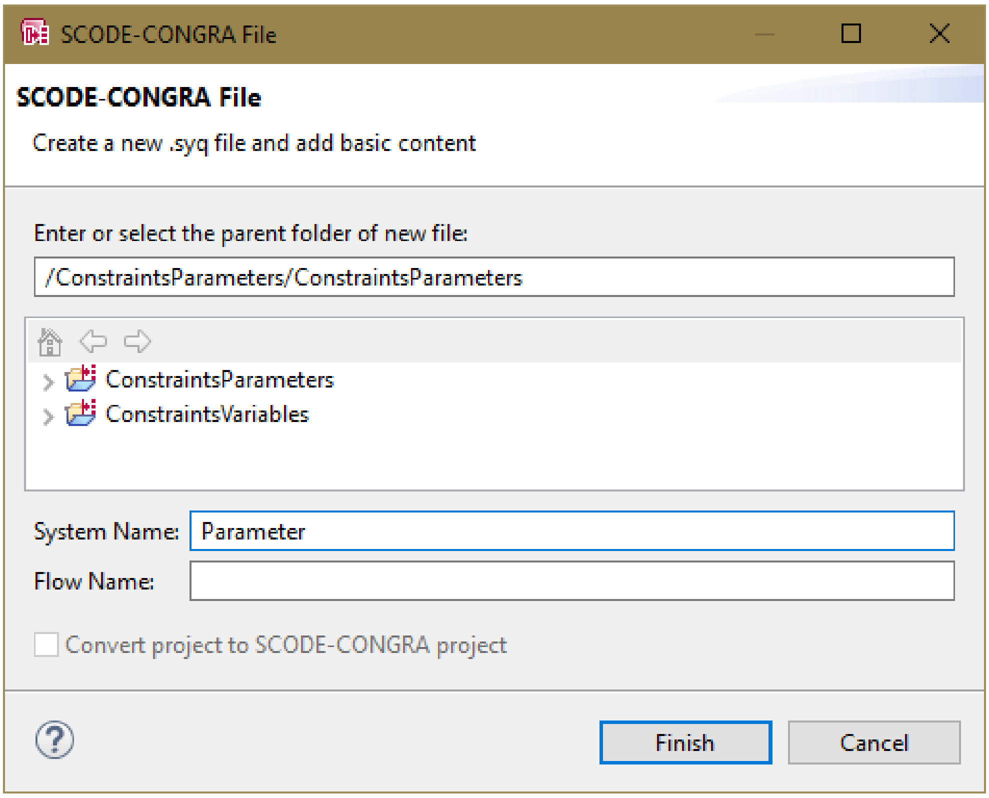
   ```
   ...
      @geo(244, 40)
      param I is != 0 = 2;
   ...
   ```

3. Generate code.
4. Open the generated files and check the effect of the constraint.

To further illustrate the effect of the constraint on parameter I, create a comparison system with no constraint on I.

To specify the comparison system

1. In the Project Explorer, right-click the `ConstraintsParameters` folder and select **New > SCODE-CONGRA File**.
2. In the "SCODE-CONGRA File" window, enter a system name, e.g., `Parameter`, then click **Finish**.



3. Specify the `Parameter` system the same way as the `ConstraintsParameters` system, but leave out the constraint `!=0` for parameter I.
4. Save the project.
5. Generate code.

Section 9.2.5.3 compares generated code with and without parameter constraint.

5.9. Lesson 8: Variables with Physical Units

In the previous lessons, all variables, parameters, etc. were treated as unitless numbers. In this lesson, you will assign physical units to the variables.

Expressions consider units. First, there are checks for "dimension compliance". This means that SCODE-CONGRA ensures, for all additive (or comparative) expressions, that the dimensions of the operands of the operation comply with respect to the physical unit dimension. The same happens to the sides of an equation.

For this lesson, you will use the same model as in lessons 4 to 7.

To set up the project

1. Create a SCODE-CONGRA project and name it, e.g., `PhysicalUnits`.
2. Create and specify the relations for Ohm's law (Equation 1) and the power of an ohmic resistor (Equation 4).
3. Create at least one flow, e.g., with I and U as inputs.
4. Enable code generation for C code.
5. Save the project.

In SCODE-CONGRA, you have to define units in a `*.syq` file before you can use them. This can be done only in the text editor. Units have to be defined outside the system and outside the flows. Each unit must have a unique name.

This tutorial uses the *International System of Units* (SI). This system comprises a coherent system of units of measurement built on seven base units , which are the *second*, *meter*, *kilogram*, *ampere*, *kelvin*, *mole*, *candela*. [25] The system also specifies names for 22 derived units [26], among them ohm, volt, watt, etc., for other common physical quantities.

Base units are defined directly, while derived units are defined as combinations of base units. This means that you have to define all base units that you use directly and indirectly, i.e. for derived units.

Variable	Unit	Unit Type	in SI base units
I	A (ampere, electric current)	base	---
U	V (volt, voltage)	derived	$(kg * m^2) / (A * s^2)$
R	Ω (ohm, resistance)	derived	V/A
P	W (watt, electric power)	derived	V*A

Table 15. Some variables with units

You can define units in one central place and import them into the systems where they are needed. That procedure is described in section 5.9.1, “Defining Units in Separate Files”.

Alternatively, you can define units in the system `*.syq` file. The procedure is described in section 5.9.2, “Defining Units in the System SYQ File”.

The SCODE-CONGRA online help recommends the first way to define units.

5.9.1. Defining Units in Separate Files

This section describes the definition of units in a central place (a special project), where they can be accessed from other projects.

NOTE

Defining units in separate files is the recommended way to define units.

Separate files for units are easy to maintain, and they can be shared by many projects.

Defining units in the system’s `*.syq` file is described in section 5.9.2, “Defining Units in the System SYQ File”.

To define units in a special project

1. Create the SCODE-CONGRA project that will contain the unit definition file(s).

   This tutorial uses a project named `UnitDefinitions`.

   The system `*.syq` file opens automatically. It contains the following content:

```
package UnitDefinitions;

system UnitDefinitions {

}
```

2. If you want to use this project only for unit definitions, delete the lines below the package declaration.

   Units have to be defined outside systems and flows. In a file that only defines units, a system is unnecessary.

3. Define the necessary base units as follows:

   `unit <base_unit name>;`

   NOTE

   The base unit for time needs a special definition:

   `unit <time_unit name> is time;`

   The additional `is time` marks *<time_unit name>* as time in seconds.

   In the context of a system, only one unit can be defined with `is time`. A second definition `unit <name> is time;` causes an error.

4. Define the necessary derived units as follows:

   `unit <derived_unit name> = <expression>;`

   <expression> is a combination of base units, derived units and/or scaling factors. You can combine units and factors via * or / operators, and you can use brackets, e.g. to enclose a denominator. See the following example:

   ```
    unit N = kg * m / (s*s);
   . If desired, enter comments that describe the units.(((SYQ
   file,comment)))
   ```

   In the SYQ language, a comment is included in `/* … */`. You can place the comment in a line that contains code, or you can place the comment in one or more separate lines.

5. Save the project.

Examples for base units, derived units, and comments are given in section 9.2.6.1, “Example: Unit Definitions in a `*.syq` File”.

If you want to spread the unit definitions over several files, or if you want to add a unit definition file to an existing SCODE-CONGRA project, you have to create additional `*.syq` files.

To create an additional unit definition file

1. Select the SCODE-CONGRA project that will contain the unit definition file(s).
2. In the Project Explorer, right-click one of the following items and select **New > File** from the context menu.
    - project
    - system folder
    - `*.syq` file

   The "New File" dialog window opens.
3. In the "New File" dialog window, do the following:
    i. Select the system folder as parent folder for the new file.
    ii. Enter a name and the extension `.syq` for the file.

       You must enter the extension to determine the file type.
    iii. Click **Finish** to create the file.

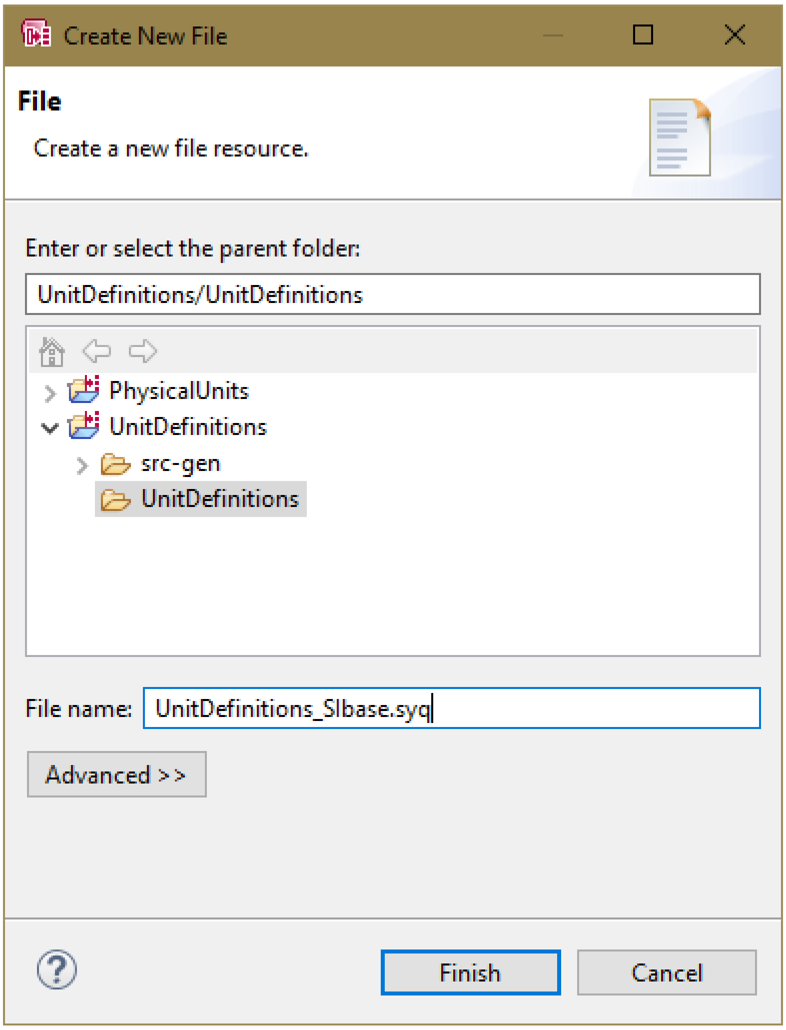


Figure 70. "New File" window

The empty file is created and opened in the text editor.

4. In the first line, enter the package declaration:

```
package <project name>;
```

5. Define the units as described in To define units in a special project, steps 3 and 4.
6. If desired, enter comments that describe the file content and/or the units.
7. Save the project.

   Figure 71 shows a project with unit definition files.

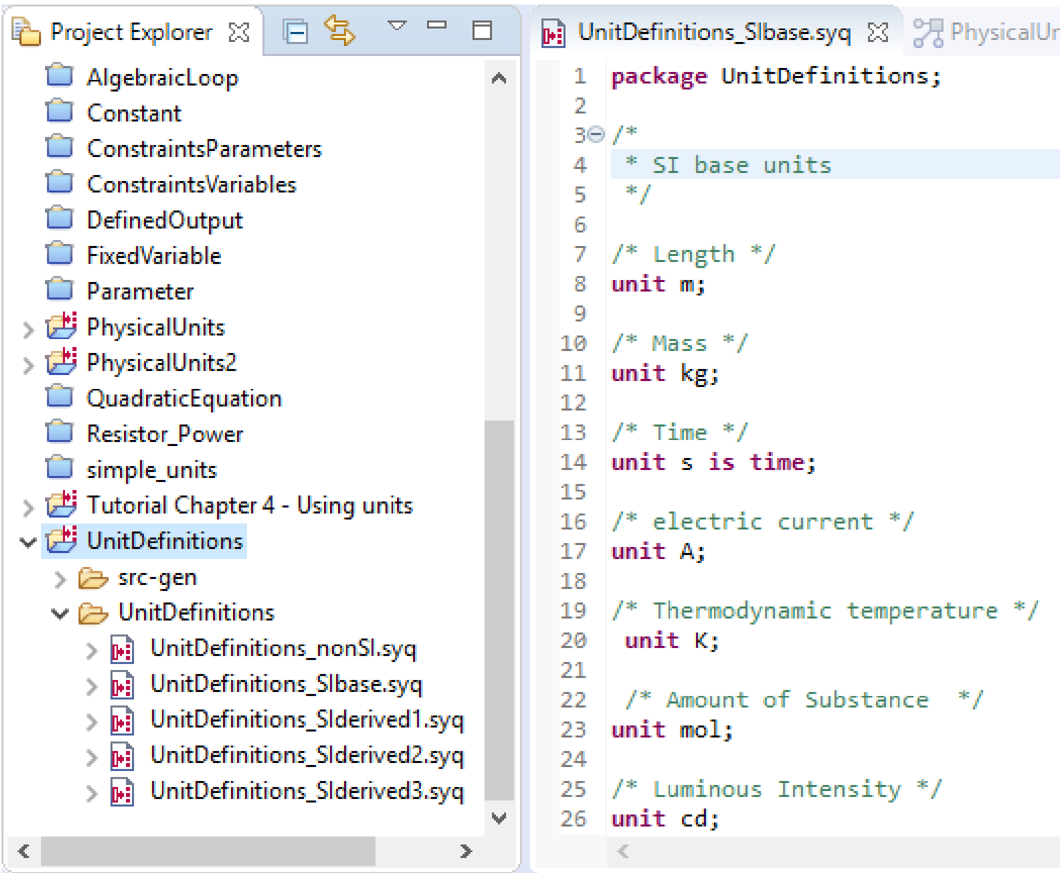


Figure 71. SCODE-CONGRA project `UnitDefinitions` with five unit definition files

Examples for base units, derived units, and comments are given in section 9.2.6.1, "Example: Unit Definitions in a `*.syq` File".

The units in the unit definition file(s) are known to the project that contains the files. To use them in another project, you have to connect the projects, and then import the units.

To connect two projects

1. In the Project Explorer, select the project you want to connect with another project.

   For example, select the project that will use the units defined in a special project.

   This tutorial connects a project named `PhysicalUnits2` with the `UnitDefinitions` project that contains the unit definitions. [27]

2. Right-click the project and select **Properties** from the context menu.

   The "Properties for *<project>*" window opens.

3. In that window, go to the "Project References" node.

   This node lists all projects in the workspace.

4. Select the project you want to connect and click **Apply and Close**.

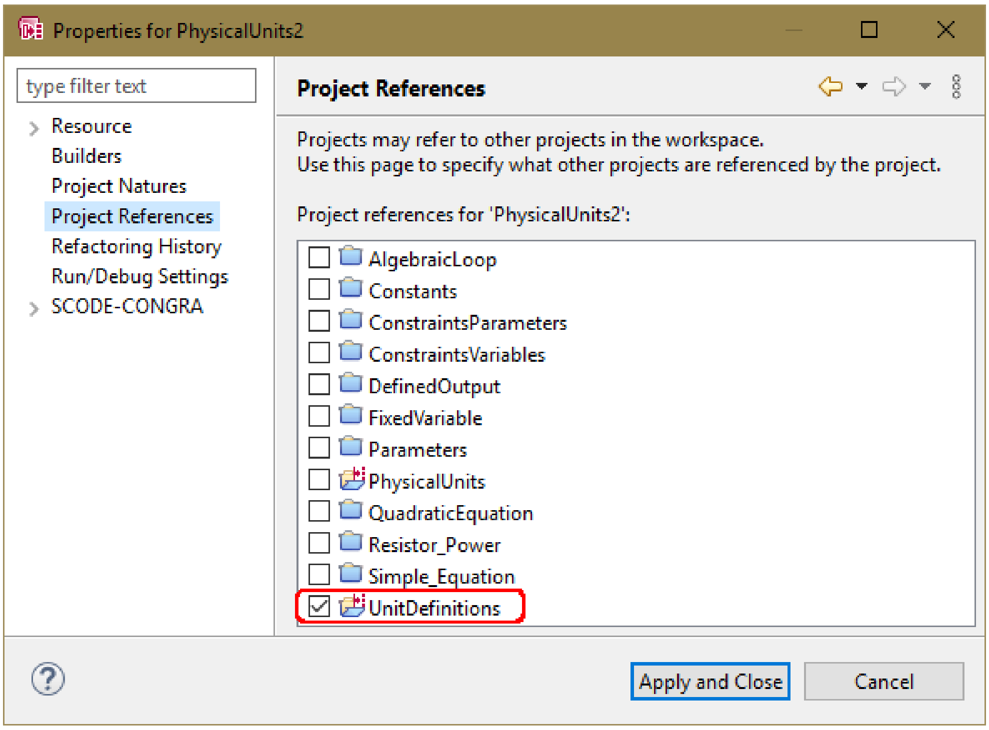


Figure 72. "Properties for <*project*>" window, "Project References" node

The project you first selected now refers to the second project. However, the second project does *not* refer to the first.

In the example shown in Figure 72, the `PhysicalUnits2` project refers to the `UnitDefinitions` project, but `UnitDefinitions` does not refer to `PhysicalUnits2`.

Now you can import content from the referred project (`UnitDefinitions` in Figure 72) into the referring project (`PhysicalUnits2` in Figure 72). You can import the following items:

- units
- systems
- flows
- computations

You cannot import an entire package. Each item you want to import needs its own import declaration.

To import content from another package

1. Open the `*.syq` file into which you want to import content.

   If your system contains variable definitions with undefined units, these definitions are marked as errors.

2. Between package declaration and system definition, enter the following line for each item you want to import:

```
import <package name>.<item name>;
```

<package name> is the name of the package that contains the item to be imported.

<item name> is the name of the item to be imported.

> **NOTE**
>
> You have to explicitly import each unit you want to use.
>
> It is recommended that you import all units used to form the derived units you want to use.

3. To make work easier, use the following method:

   i. Type `import`, followed by a blank.

   ii. Press Ctrl + Space.

   A white box opens. It shows all items you can import. Units are marked with (U). A yellow box opens, too, and shows details for the selected element.

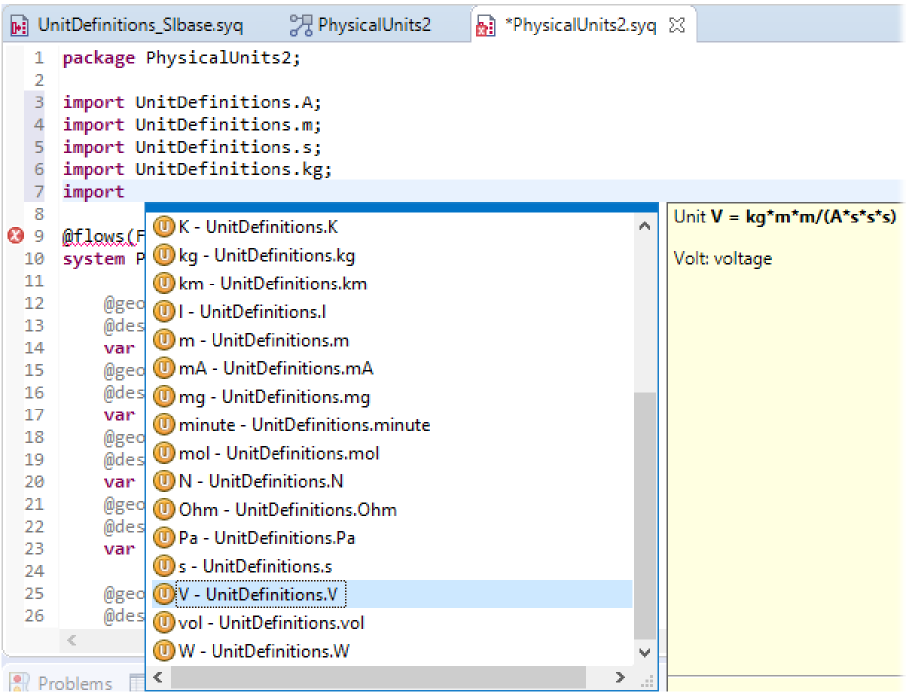


Figure 73. Popup with items that can be imported. The items are listed as follows: icon *<item name>* - *<package name>.<item name>*

   iii. Select the unit you want to import.

   iv. Press Enter to insert your selection.

   v. Enter the closing `;`.

4. Save the project.

```
package PhysicalUnits2;

import UnitDefinitions.A;
import UnitDefinitions.m;
import UnitDefinitions.s;
import UnitDefinitions.kg;
import UnitDefinitions.V;
import UnitDefinitions.Ohm;
import UnitDefinitions.W;

system PhysicalUnits2 {
...
```

Table 16. `*.syq` file with imported units

You can now assign the units to variables; see section 5.9.3.

5.9.2. Defining Units in the System SYQ File

NOTE The recommended way to define units is a separate file.

To define units in the system SYQ file

1. Open the `PhysicalUnits.syq` file in the text editor.

   Units have to be defined outside systems and flows. You can place them, e.g., between the `package` … line and the `system` … line, or at the end of the `*.syq` file.

2. Define the necessary base units as described in To define units in a special project, step 3.

   NOTE The base unit for time needs a special definition:

   `unit` *<time_unit name>* `is time;`

   The additional `is time` marks *<time_unit name>* as time in seconds.

   In the context of a system, only one unit can be defined with `is time`. A second definition `unit` *<name>* `is time;` causes an error.

3. Define the necessary derived units as described in To define units in a special project, step 4.

4. If desired, enter comments that describe the units.

   In the SYQ language, a comment is included in `/*` … `*/`. You can place the comment in a line that contains code, or you can place the comment in one or more separate lines.

5. Save the project.

Examples for base units, derived units, and comments are given in section 9.2.6.1.

You can now assign the units to variables; see section 5.9.3.

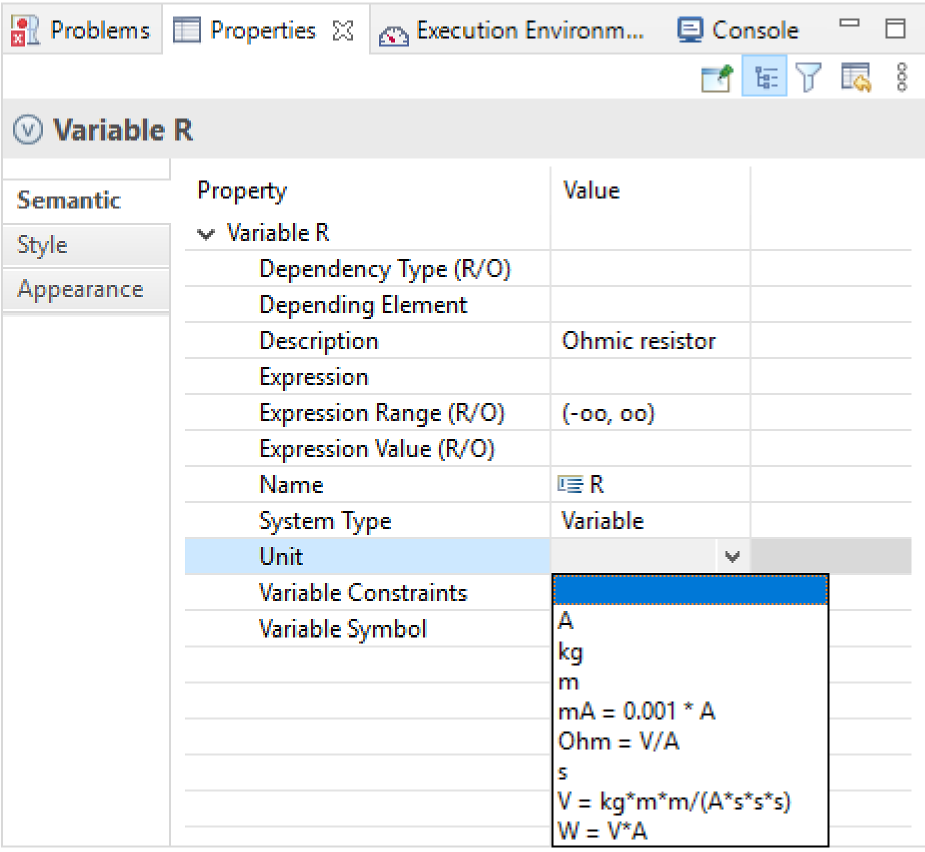
5.9.3. Assigning Units

Now you can assign the units to the variables. You will do this in the graphical editor of the system or the flow.

To assign units to variables

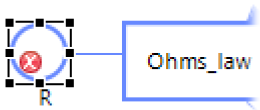
1. Open the system graph or the flow graph.
2. Open the "Properties" view for a variable.
3. In the "Properties" view, "Semantic" node, click in the "Value" column next to "Unit".

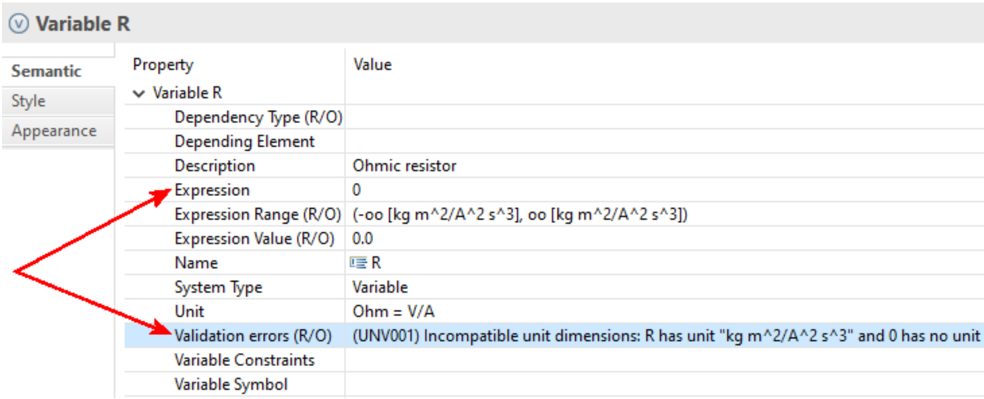
   A dropdown list opens that offers all defined units for selection.



4. Select the appropriate unit.

   The unit is *not* automatically assigned to an existing default value. If you entered a default value when you set up the project, an error is issued.





5. Enter a default value with unit; see also To enter a value with unit in a graph.

   Example: 

6. Select units for the other variables.
7. Save the project.

   The variable definitions in the `*.syq` code change as follows:

```
... ...
6   @geo(420, 145)
7   @description("electric power")
8   var W P;
9   @geo(200, 200)
10  @description("current")
11  var A I ;
12  @geo(40, 145)
13  @description("Ohmic resistor")
14  var Ohm R = 0 [Ohm];
15  @geo(200, 80)
16  @description("voltage")
17  var V U;
... ...
```

   Table 17. `*.syq` file extract: variable definitions with units (lines 8, 11, 14, 17). The unit name appears before the variable name.

You can assign units to variables, parameters, and constants in the `*.syq` file. To do so, insert the unit name before the element name:

```
var <unit name> <element name>;
param <unit name> <element name>;
const <unit name> <element name>;
```

Each time you save the project, SCODE-CONGRA checks if the units of the various variables match. If the units do not match, an error "UNV001 incompatible unit dimensions …" is issued.

The `PhysicalUnits.syq` file with no unit assigned to electric current I is shown in Figure 74, as well as the error markers and the error messages in the "Problems" view.

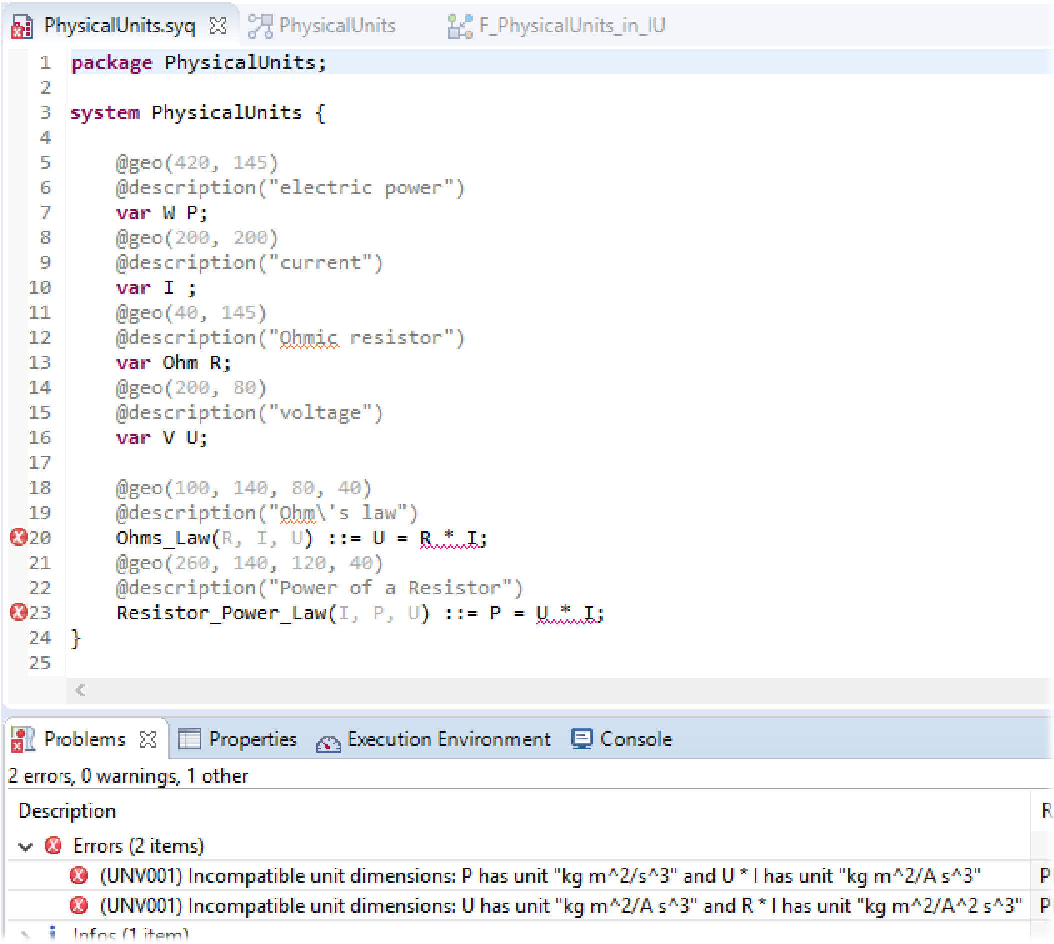


Figure 74. `PhysicalUnits.syq` file and "Problems" view with error markers due to incompatible units

5.9.4. Units and Initial Values/Constraints

Once a unit is assigned to a variable, all assignments to that variable are checked for matching units. This includes start values and constraints.

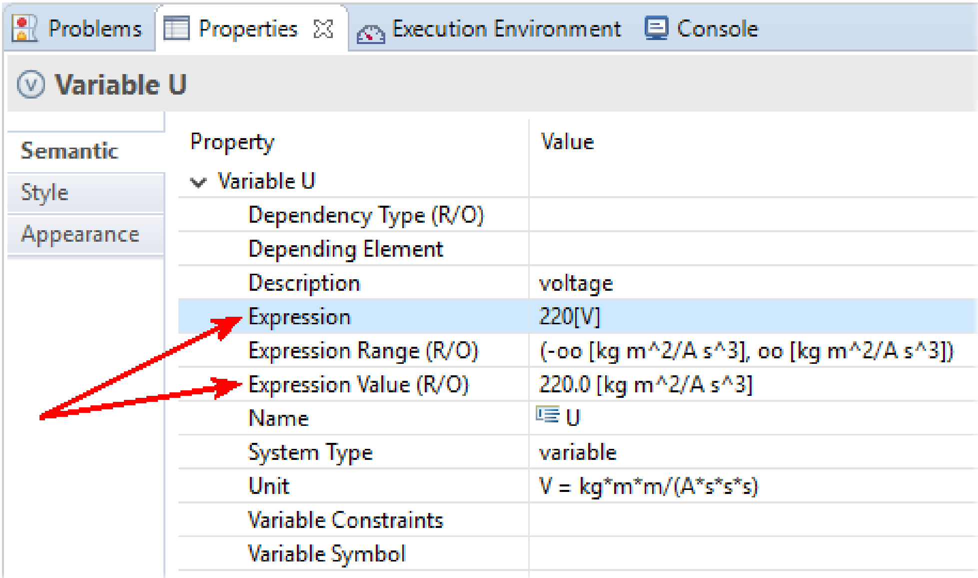
You can enter a start value with unit either in the system graph, or in the `*.syq` file.

To enter a value with unit in a graph

1. Open the "Properties" view for the variable that needs a value.
2. In the "Properties" view, "Semantic" node, click in the "Value" column next to "Expression".

   The cell becomes an input field. ..Enter the desired value, followed by the unit in square brackets.

   For example, enter `220 [V]` as value for U.



Value and unit are transferred to the "Expression Value (R/O)" row. A derived unit is replaced by the combination of units it is derived from.

The `*.syq` file is updated when you save the project.

```
@description("voltage")
var V U = 220[V];
```

If desired, you can enter value and unit for a variable, parameter, or constant directly in the `*.syq file:`

```
<itemType> <unit> <itemName> = <value> [<unit>];
```

<itemType> can be `var`, `param`, or `const`.

Constraints are specified similarly, with the required unit in square brackets. In the `*.syq` file, constraints with units look as follows:

```
<itemType> <unit> <itemName>
          is <constraintType>  <constraintValue>[<unit>]
          and <constraintType> <constraintValue>[<unit>];
```

Example: `var V U is > 0[V] and □ 230[V];`

If you want to specify both a value and constraints in the `*.syq` file, the value must be defined after the constraint. If you place the value definition before the constraints definition, you cause an error.

```
<itemType> <unit> <itemName>
          is <constraintType> <constraintValue>[<unit>]
          and <constraintType> <constraintValue>[<unit>]
          = <value> [<unit>];
```

Example: `var V U is > 0[V] and □ 230[V] = 220[V];`

<itemType> can be **`var`**, **`param`**, or **`const`**. For a list of *<constraintType>* values, see Table 13.

5.9.5. Units in the Generated Code

This section shows the effect of units on generated code. You will use the `PhysicalUnits` project you created in section 5.9.2.

To prepare the project

To see how SCODE-CONGRA deals with different units of the same dimension, e.g., with A and mA = 10^{-3} A as units for electric current, change the project as follows.

1. Define a new unit mA = 10^{-3} A.
2. Assign the new unit to the variable I.

   The units of the other variables remain as they are.
3. Make sure that default values are defined for all variables.
4. Save the project.

To generate code

1. Open the "Properties of <*project*>" window for the `PhysicalUnits` project.
2. Activate generation of C code, ESDL code, and MATLAB code.
3. Generate code.
4. Open the generated `*.c`, `*.m`, and/or `*.esdl` files. [28]

   The conversion factor to convert mA to A is inserted automatically wherever it is required. Otherwise, the units are not visible in the generated C code and MATLAB code.
5. Open the `c_F_PhysicalUnits_in_IU` computation in the Execution Environment. [29]

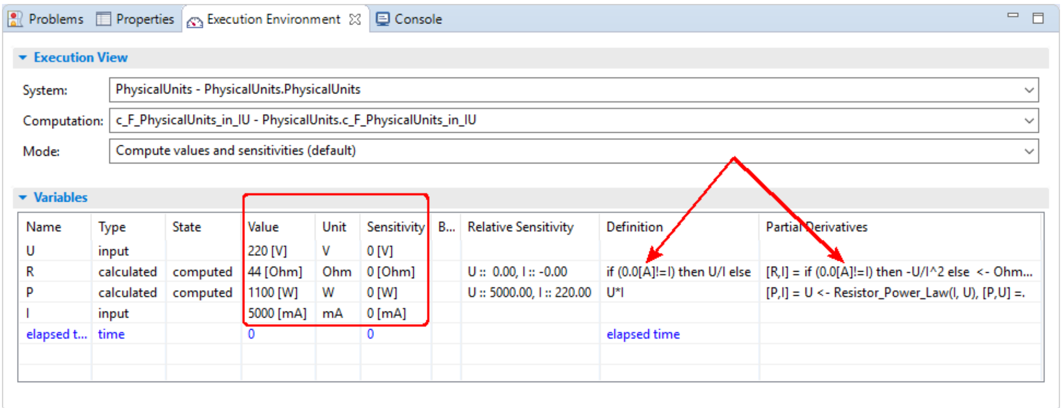


Figure 75. Execution Environment showing a computation with units. Visible units are marked.

If you change a value, or enter a sensitivity, enter the respective unit in square brackets, or not at all. In the latter case, the unit is inserted automatically.

If a derived unit is assigned to a variable, you can enter either the derived unit, or you can enter the combination of units and/or scale factors used to derive the assigned unit (provided all units are defined or imported in the project). For example, you can enter either `6000 [mA]` or `6 [A]` as value for I.

5.9.6. Additional Task

This section is not mandatory for the lesson on variables with physical units. However, it contains useful knowledge.

Generating a Report

To get a description of your system, you can generate a report. Generated reports can be read without a SCODE Workbench installation.

To generate a report

1. Right-click the system `*.syq` file to be documented and select **Run As > Report (CONGRA)** from the context menu.

   If this is the first time you use the **Run As > Report (CONGRA)** on this SCODE file, the "Edit Configuration" window opens for the report launch configuration.

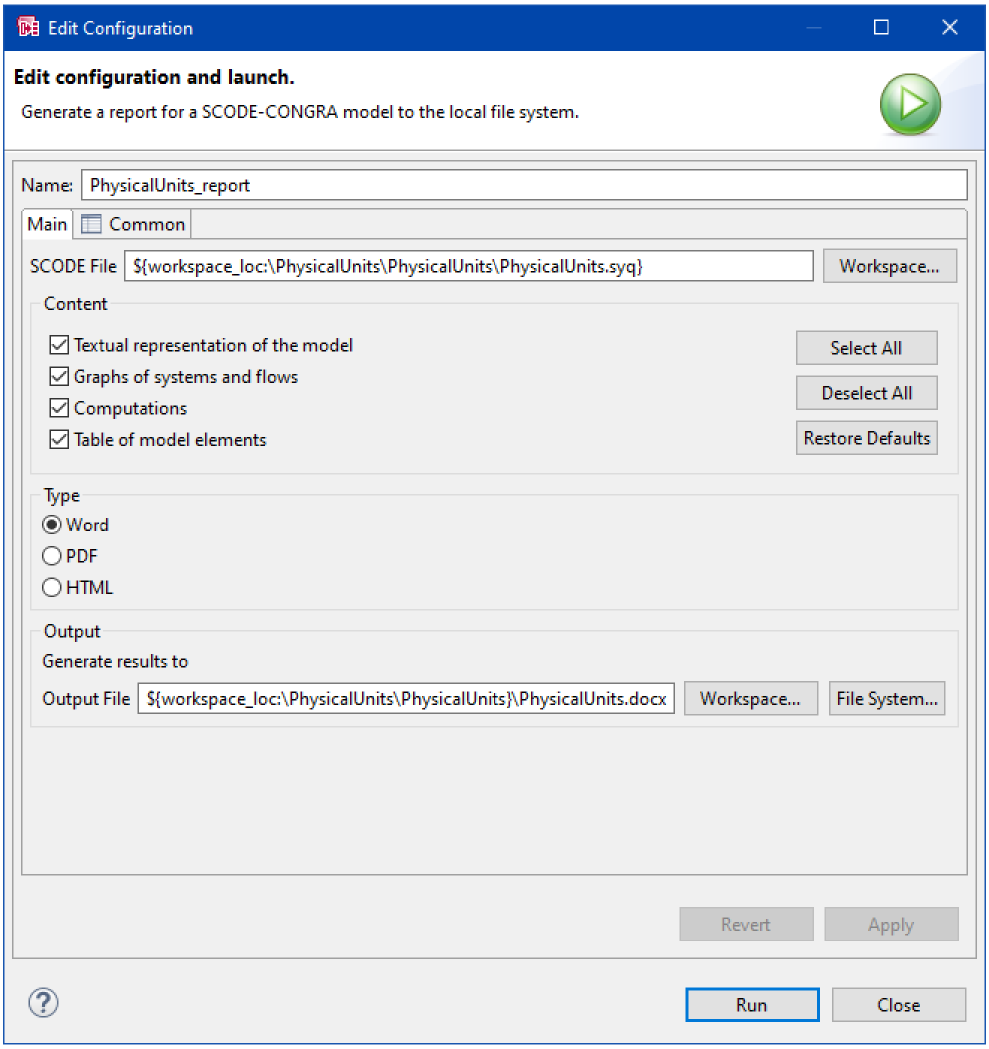
   

   Figure 76. "Edit Configuration" window for a SCODE-CONGRA report launch configuration

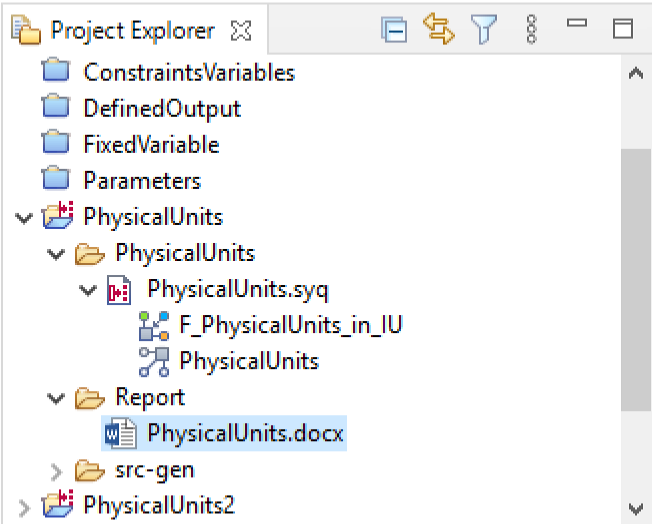
2. In the "Edit Configuration" window, enter a name for the report file.
3. In the "SCODE File" field, enter or select (via **Workspace** button) the SYQ file that contains the model you want to export.

4. In the "Content" area, activate at least one content option.
5. In the "Type" field, activate the option for the desired report file type.
6. In the "Output File" field, enter or select (via **Workspace** or **File System** button) an existing folder for the report.

NOTE	If you enter the name of an existing file with the selected type, that file is overwritten without further inquiry.

7. Click **Run** to generate the report.

   The report is generated with the selected format and stored in the selected folder. If you selected a folder inside your workspace, you can see the report in the Project Explorer.



8. In the confirmation window, click **Yes** to open the report.

A report for the `PhysicalUnits` system, with all report parts generated, is shown in section 9.2.6.4, “SCODE-CONGRA Report”.

[11] If you need help to find the "Properties" view, see Figure 3.

[12] "incoming" and "outgoing" are seen from the relation's point of view.

[13] For further information on layout changes, see Storing Layout Changes.

[14] Solver priorities are numbered in descending order. The highest priority is 1.

[15] **MuPAD [deprecated]** support will be discontinued in future SCODE Workbench versions.

[16] If you need help, see To specify the equation.

[17] If you need help, see To create a Flow.

[18] If you need help, see section 5.2.3, “Working with Computations”.

[19] If you need help, see To create a SCODE-CONGRA project.

[20] If you need help, see To set up the project.

[21] If you need help, see To generate code.

[22] If you need help, see To generate code for original and inverted flows.

[23] If you need help to find the "Problems" view, see Figure 3.

[24] The normalized equation of an equation `<left-hand side> = <right-hand side>` is defined as `<left-hand side> - <right-hand side> = 0`.

[25] See, e.g., en.wikipedia.org/wiki/SI_base_unit .

[26] See, e.g., en.wikipedia.org/wiki/SI_derived_unit#Derived_units_with_special_names .

[27] `PhysicalUnits2` is a copy of `PhysicalUnits` (section 5.9.2, “Defining Units in the System SYQ File”), without unit definitions in the system `*.syq` file.

[28] See section 9.2.6.2, “C Code for a Flow with Units” and section 9.2.6.3, “MATLAB® Code for a Flow with Units” for code examples.

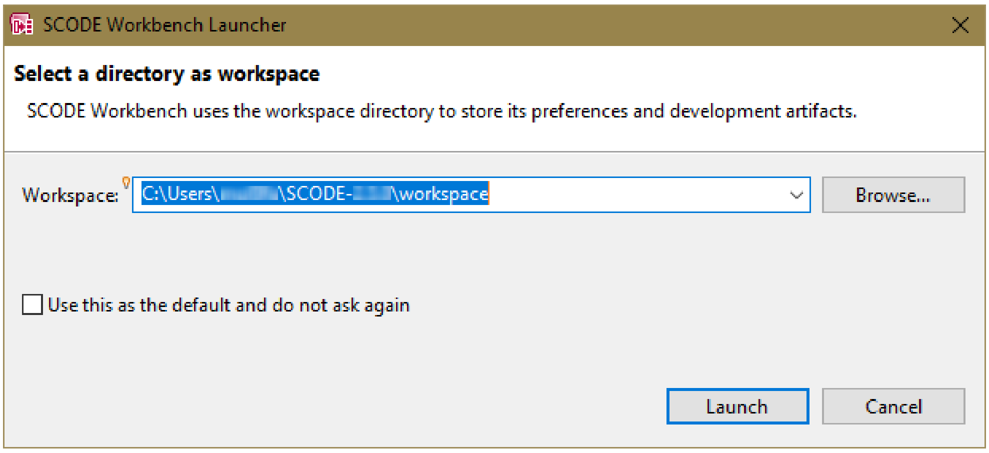
[29] If you need help, see To open the Execution Environment.

6. First Steps with SCODE Workbench

The SCODE Workbench with the SCODE-ANALYZER and SCODE-CONGRA tools is an Eclipse-based product. If you are familiar with using an Eclipse environment, then you should feel at home. If SCODE Workbench 3.1 is the first Eclipse-based application you have used, then this chapter provides some basic information to get you started.

To start the SCODE Workbench for the first time

1. Do one of the following to start the SCODE Workbench:
   - Select **ETAS SCODE Workbench 3.1** from the start menu.
   - Double-click the SCODE Workbench 3.1 desktop icon.



2. In the "SCODE Workbench Launcher" window, enter or select (via the **Browse** button) path and name of the workspace you want to use.

   If you enter a non-existing workspace, it is created.

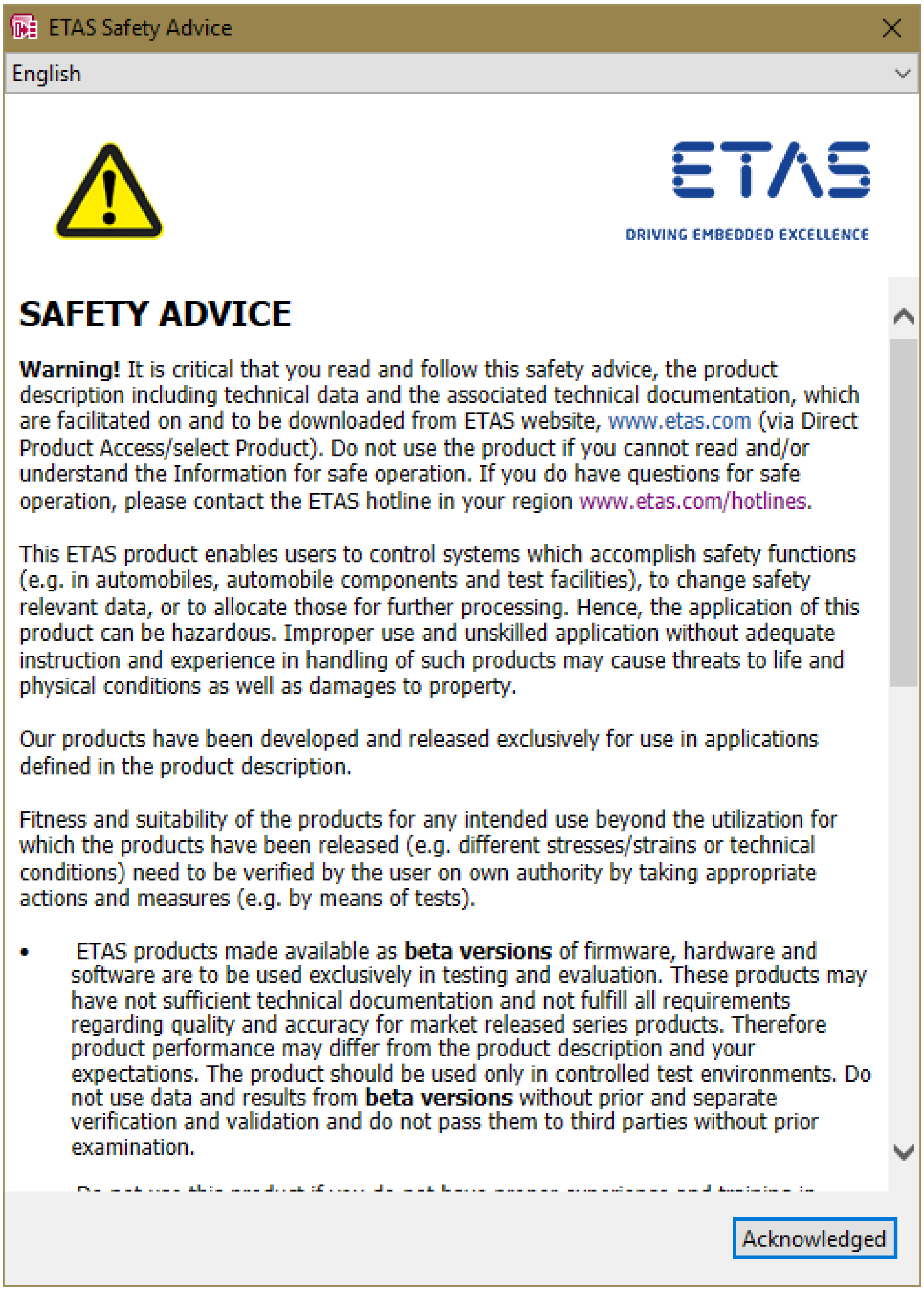
   Later, the "Recent Workspaces" list will show previously used workspaces.

3. If desired, activate the **Use this as the default and do not ask again** option.

   The next time you start the SCODE Workbench, the selected workspace opens automatically.

4. Click **OK**.

   The "ETAS Safety Advice" window opens. It contains safety information in several languages. You can select a language in the combo box at the top of the window.



NOTE

Read the Safety Advice carefully before you click **Acknowledged**.

You can open the Safety Advice in the SCODE Workbench window via **Help > ETAS Safety Advice**. A PDF version, `ETAS Safety Advice.pdf`, is available in the SCODE Workbench installation directory, `documents` subfolder.

5. **Acknowledge** the safety advice.

The SCODE Workbench is now started.

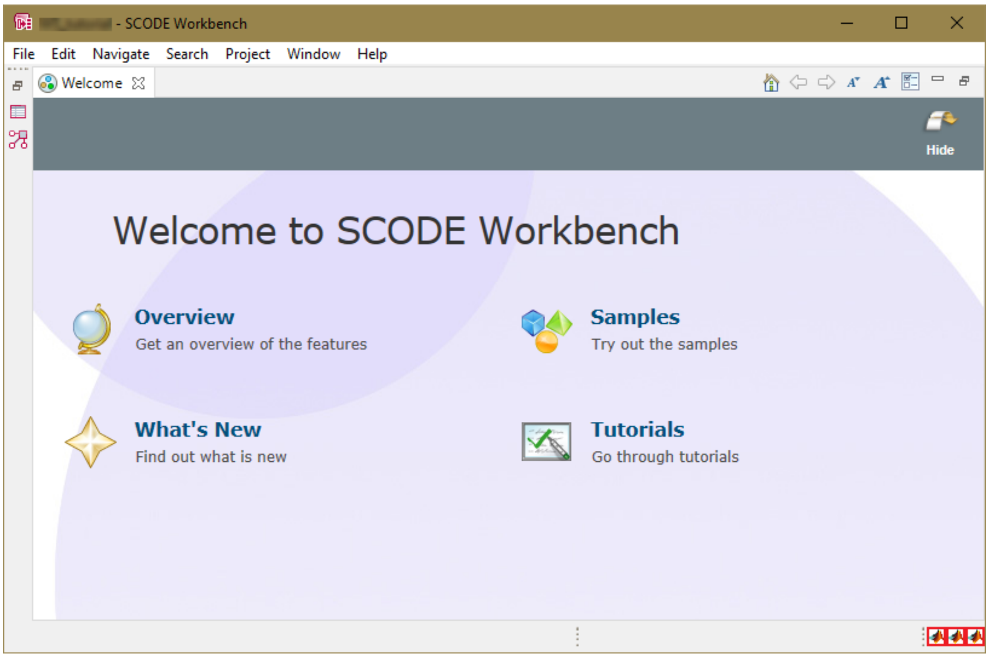


Figure 77. SCODE Workbench window, showing the Welcome page

The Welcome page contains links to useful information.

6. To reach the workbench, click the **Hide** button at the top right.
7. Click the **SCODE-ANALYZER** or **SCODE-CONGRA** button at the top right to select the appropriate perspective for the tool you want to use.

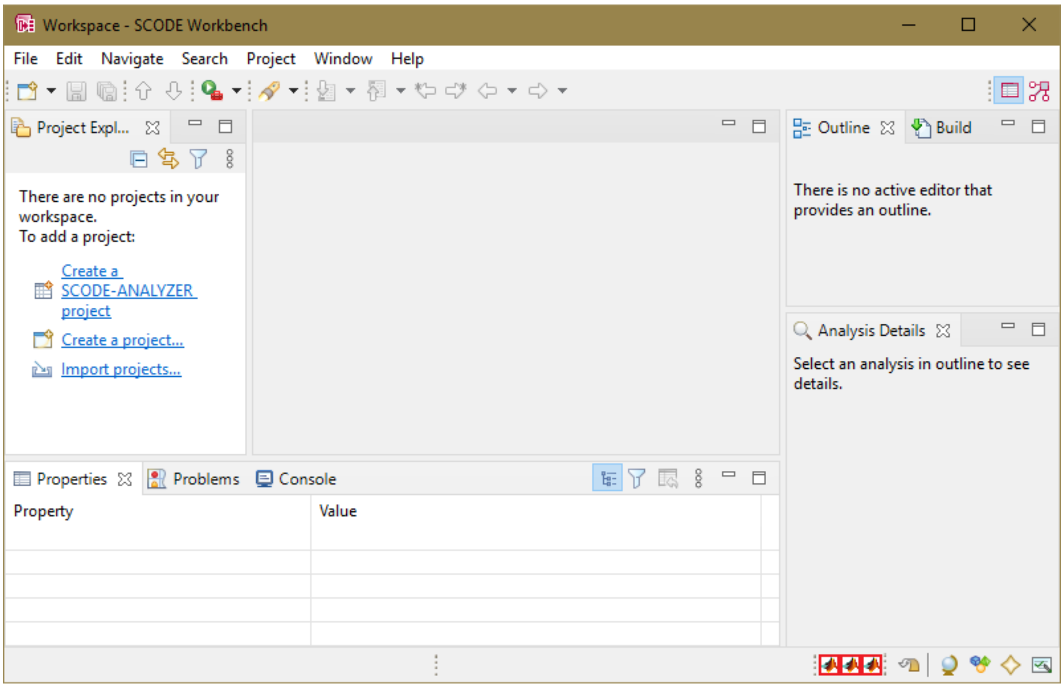


Figure 78. SCODE Workbench window, showing the SCODE-ANALYZER perspective with empty workspace

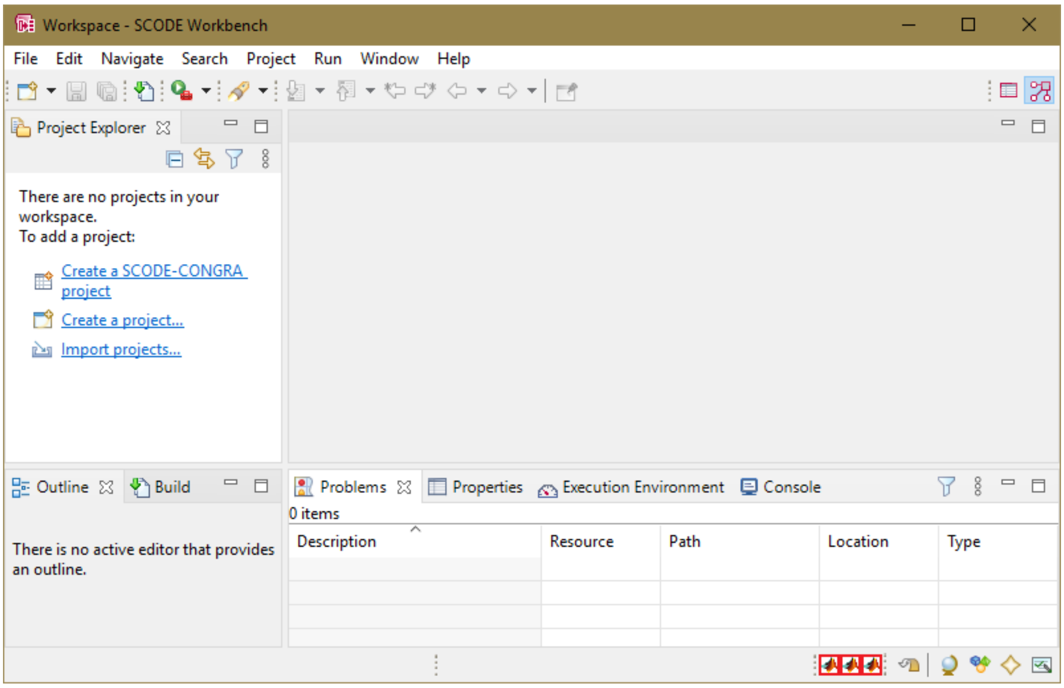


Figure 79. SCODE Workbench window, showing the SCODE-CONGRA perspective with empty workspace

6.1. First Steps with SCODE-ANALYZER

6.1.1. Generator Settings

The following generators are available:

- MATLAB
- C
- ESDL
- C++

You can select and configure a generator in the "Preferences" window.

To select and configure a generator for SCODE-ANALYZER

1. In the SCODE Workbench window, select **Window > Preferences**.

   The "Preferences" window opens.

2. In the "Preferences" window, expand the "SCODE-ANALYZER" node and go to the "Generator" subnode.

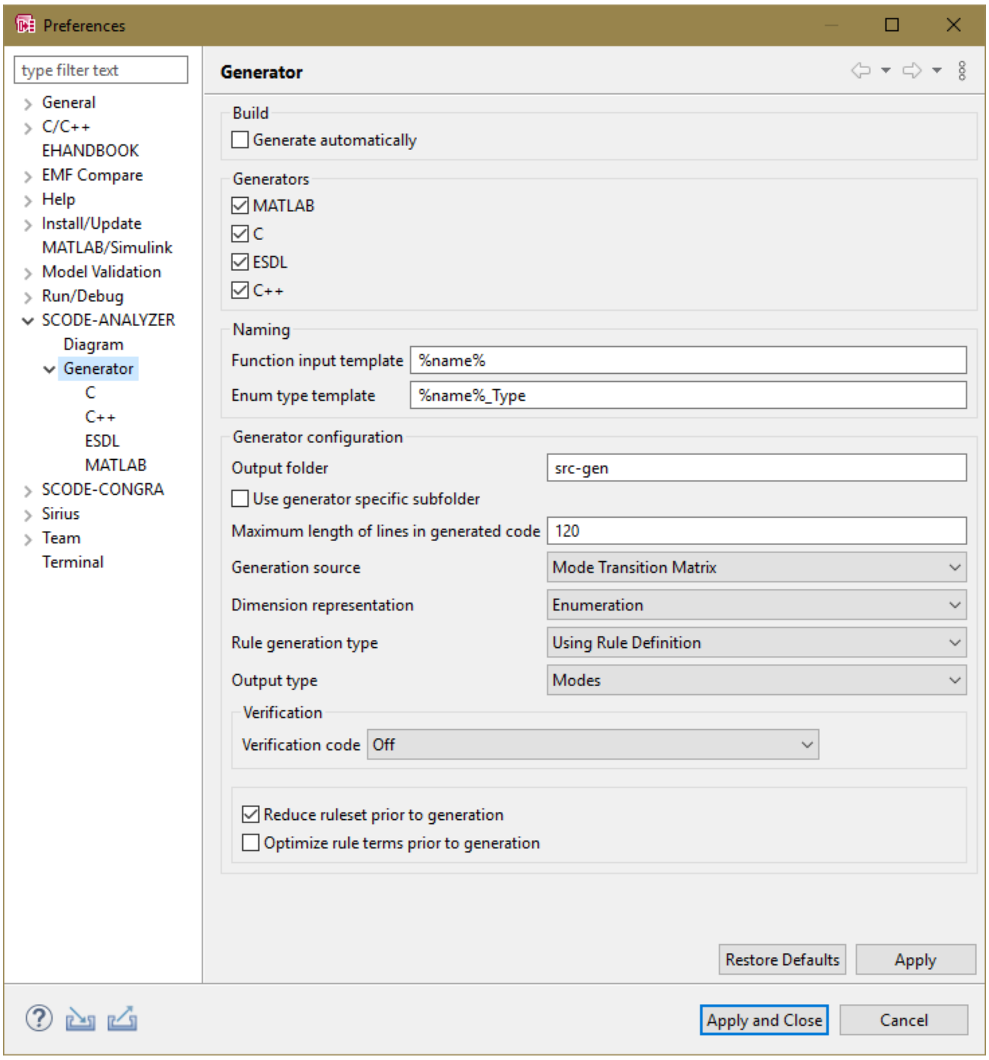


Figure 80. "Preferences" window with generator settings for SCODE-ANALYZER

3. In the "Generators" area, select the generator(s) you want to use.

   More details about the generators and their configuration are given in the SCODE-ANALYZER User Guide, chapter "Tasks", sections "Code Generation" and "Code Generation preferences".

   The user guide is opened via **Help > Help Contents**.

4. Files are generated in the working directory. Make sure that the respective access rights for this folder are available.

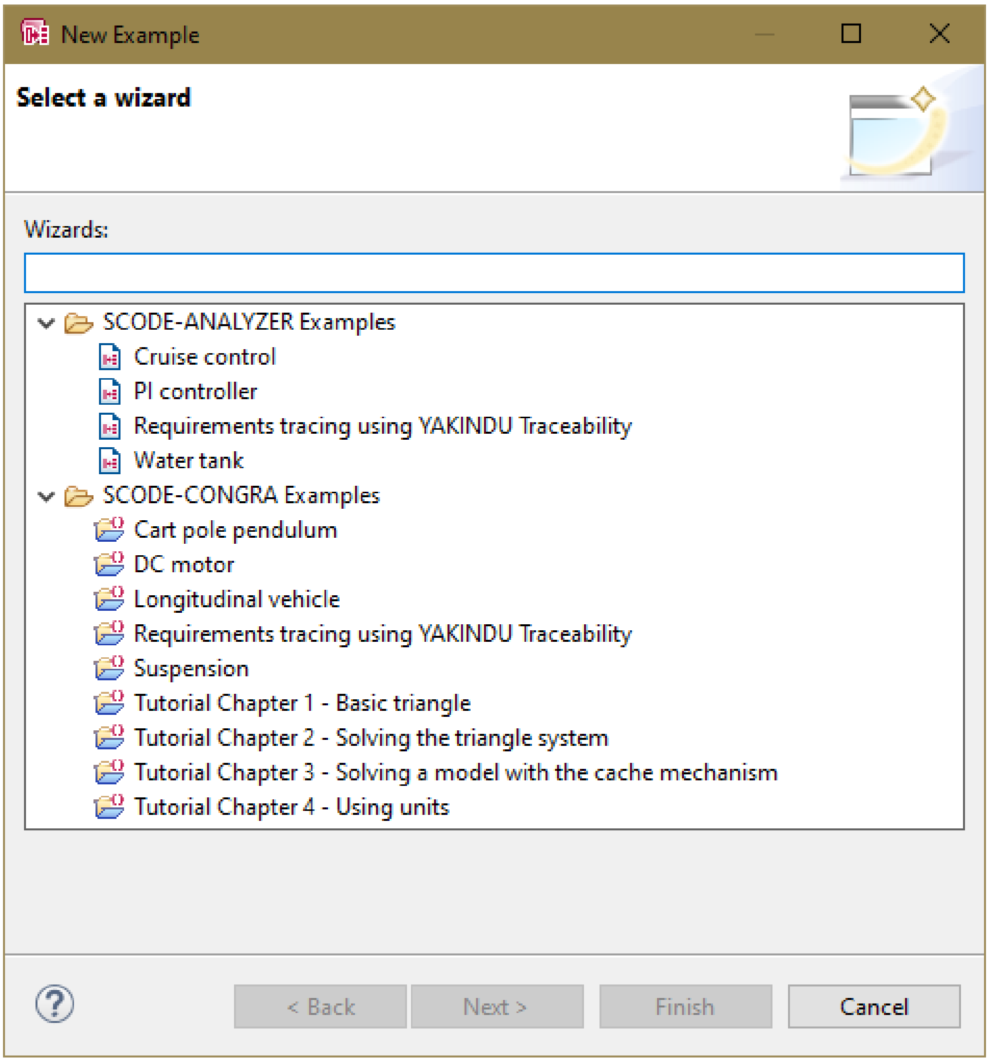
6.1.2. Start Using SCODE-ANALYZER

To start using the features of SCODE-ANALYZER it is helpful to start with one of the examples provided with the tool.

To create an example project for SCODE-ANALYZER

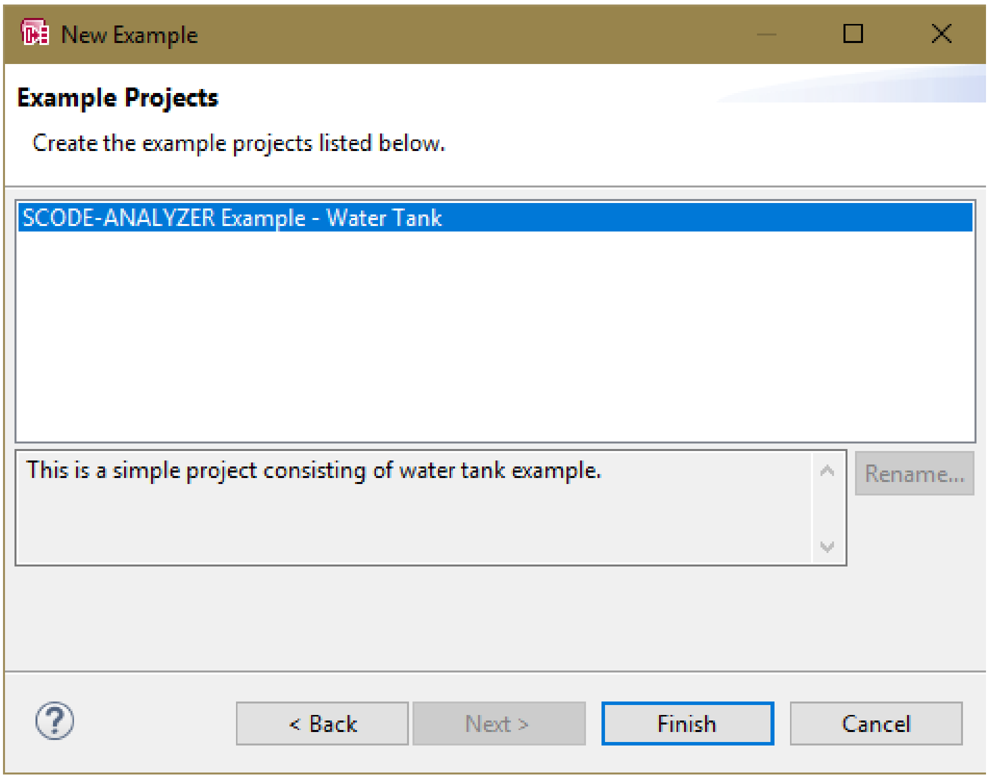
1. Open the SCODE Workbench (see step 1 in the previous instruction).
2. Do one of the following:
    - Right-click in the project explorer and select **New > Example** from the context menu.
    - Select **File > New > Example**.
    - Click the arrow next to the **New** button and select **Example** from the dropdown menu.

   The "New Example" window opens. It shows the examples for SCODE-ANALYZER and SCODE-CONGRA.



3. In that window, select a SCODE-ANALYZER example project and click **Next**.

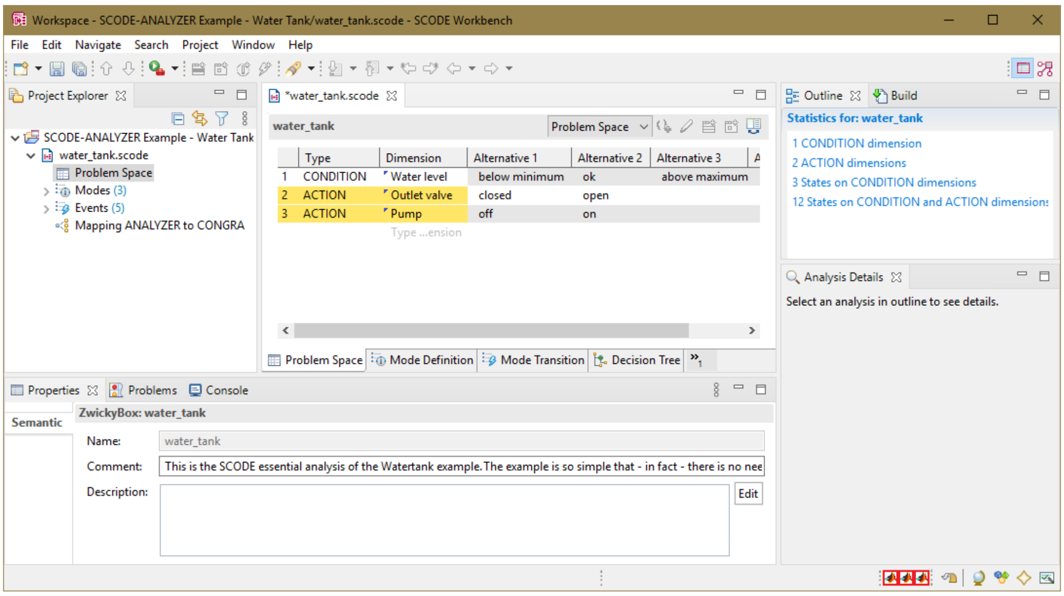
   The selected project is listed.



4. Click **Finish** to create the sample project.

   The example project is imported into your workspace. It is shown in the project explorer.

5. In the project explorer, open the `SCODE-ANALYZER Example - Water Tank` folder and double-click the `water_tank.scode` file.



You are now ready to discover or use SCODE-ANALYZER!

For more information on how to use SCODE-ANALYZER, see chapter 4, *SCODE-ANALYZER Tutorial* in this manual, and the SCODE-ANALYZER User Guide (opened via **Help > Help Contents**).

6.2. First Steps with SCODE-CONGRA

6.2.1. Settings

Before you use SCODE-CONGRA for the first time, Maxima has to be activated. By default, Maxima is activated. If you want to check the activation, proceed as described in the following instruction.

To check Maxima activation

1. In the SCODE Workbench window, select **Window > Preferences**.

   The "Preferences" window opens.

2. In the "Preferences" window, expand the "SCODE-CONGRA" node and go to the "Solver" subnode.

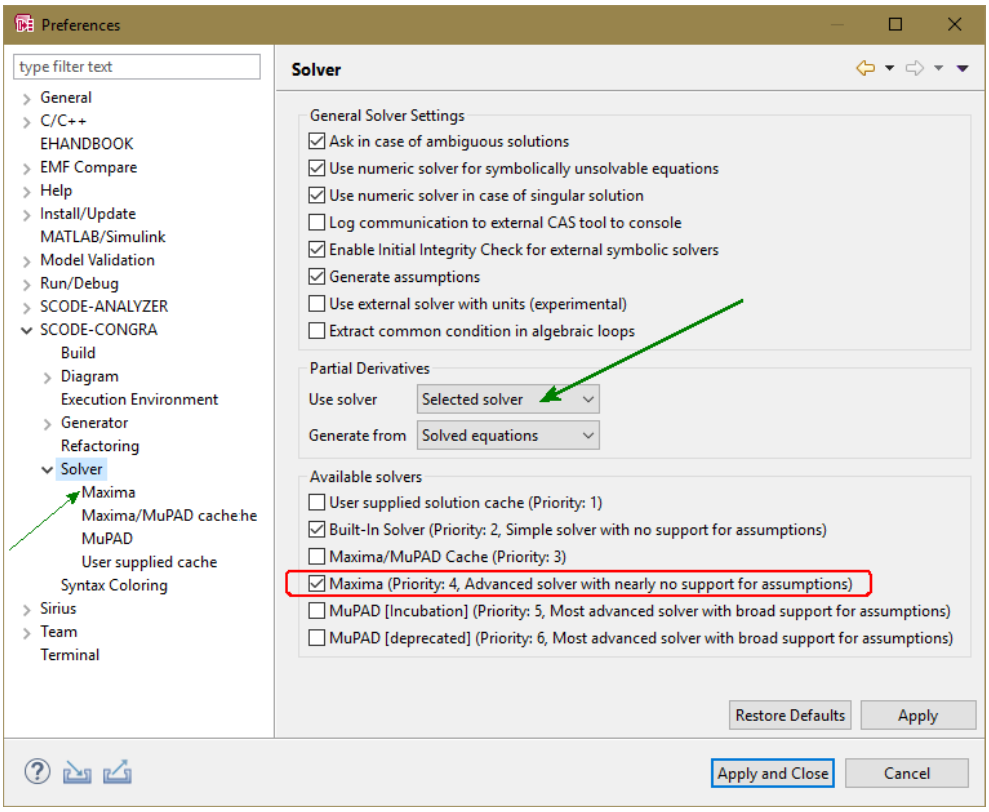


Figure 81. "Preferences" window with "Solver" settings for SCODE-CONGRA

3. In the "Use Solver" combo box (large arrow in Figure 81), select `Selected Solver`.

4. In the "Available Solvers" area, activate **Maxima (Priority: 4 *)**.

   More details about the generators and their configuration are given in the SCODE-CONGRA User Guide, chapter "Tasks", section "Preferences of ETAS SCODE-CONGRA", subsection "Configuration of the Solvers".

   The user guide is opened via **Help > Help Contents**.

5. Click **Apply**.

With that, the internal cache will speed up the tool by reusing solutions already calculated.

6. Configure the Maxima installation directory in the "Maxima" subnode (small arrow in Figure 81).
7. Click **Apply** or **Apply and Close**.

   If you click **Apply**, the "Preferences" window remains open.

To select and configure a generator

To use a code generator, activate the generator as follows.

1. In the SCODE Workbench window, select **Window > Preferences**.

   The "Preferences" window opens.

2. In the "Preferences" window, expand the "SCODE-CONGRA" node and go to the "Generator" subnode.

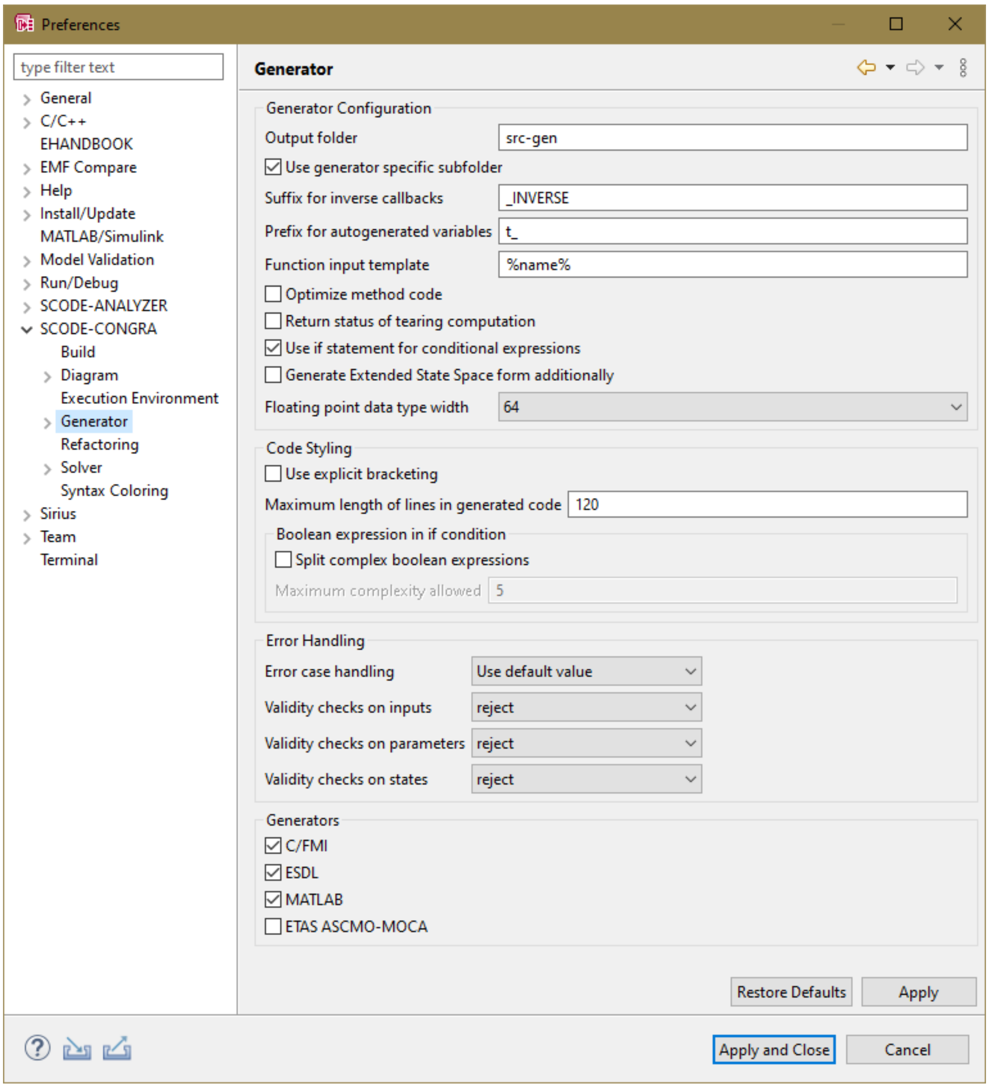


Figure 82. "Preferences" window with "Generator" settings for SCODE-CONGRA

3. In the "Generators" area, select the generator(s) you want to use.

More details about the generators and their configuration are given in the SCODE-CONGRA User Guide, chapter "Tasks", section "Triggering the generators". The user guide is opened via **Help > Help Contents**.

4. Files are generated in the working directory. Make sure that the respective access rights for this folder are available.

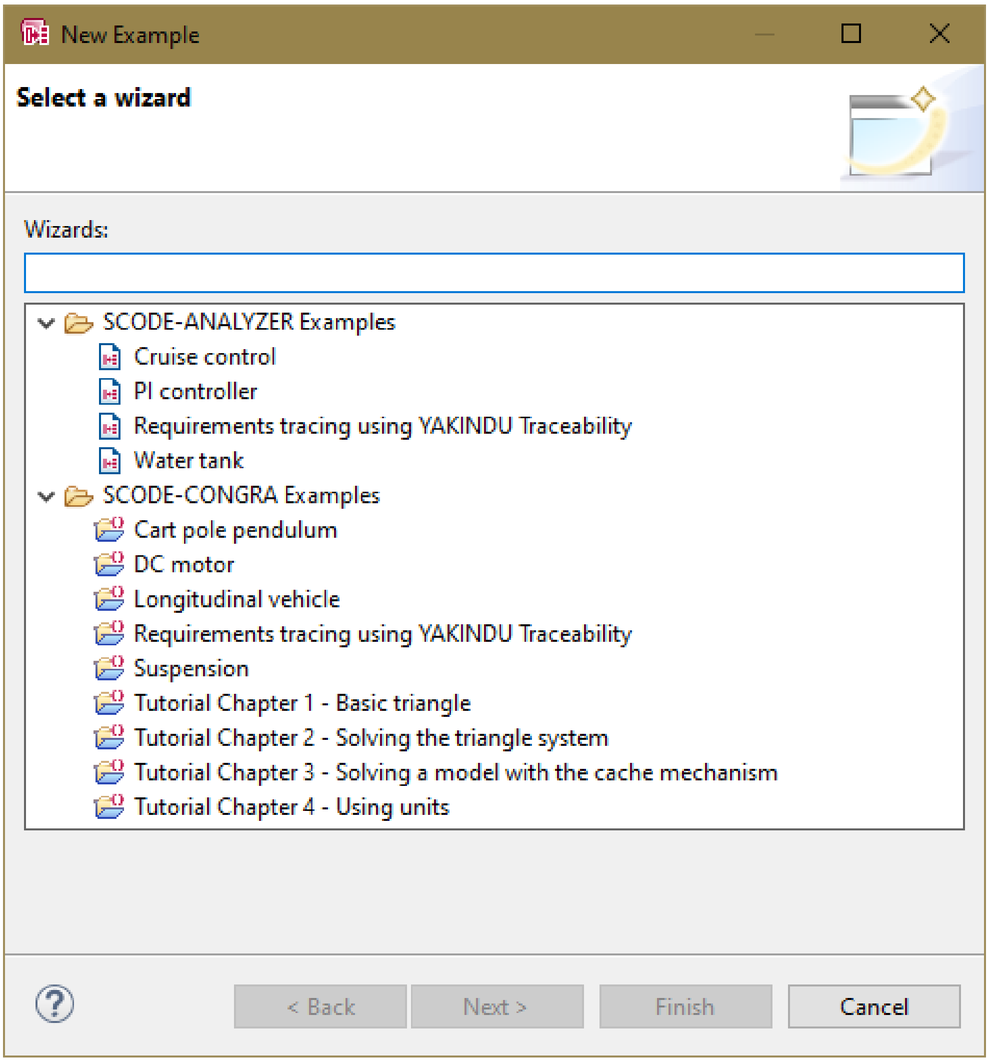
6.2.2. Start Using SCODE-CONGRA

To start using the features of SCODE-CONGRA, it is helpful to start with one of the examples provided with the tool.

To create an example project:

1. Do one of the following:
    - Right-click in the project explorer and select **New > Example** from the context menu.
    - Select **File > New > Example**.
    - Click the arrow next to the **New** button and select **Example** from the dropdown menu.

   The "New Example" window opens. It shows the examples for SCODE-ANALYZER and SCODE-CONGRA.

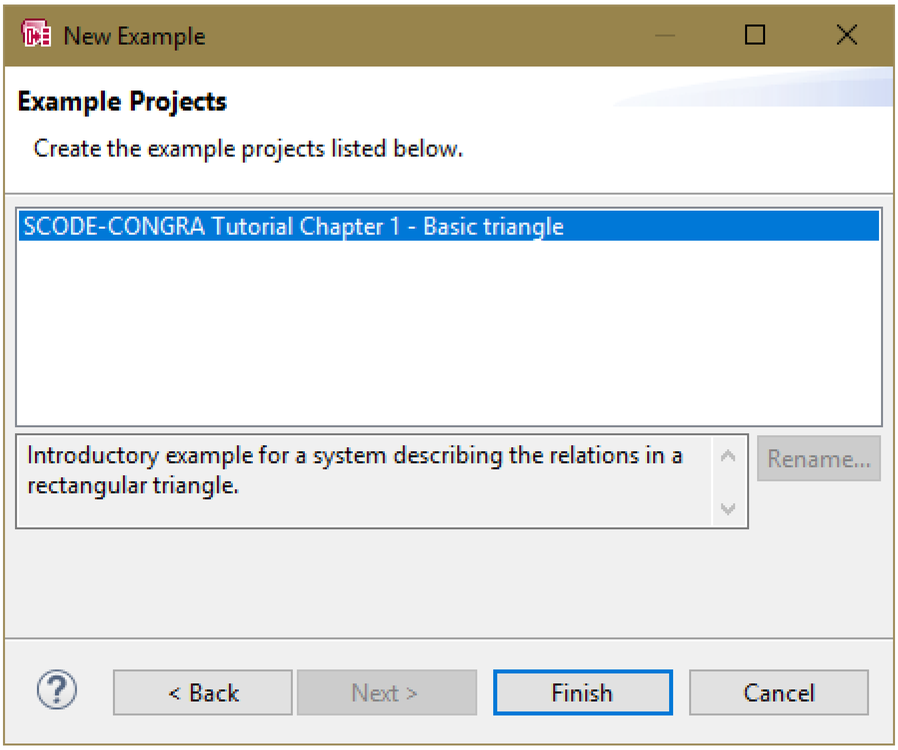


NOTE

For most of the functionality of SCODE-CONGRA, it is necessary to activate a Computer Algebra System as solver; see section 6.2.1.

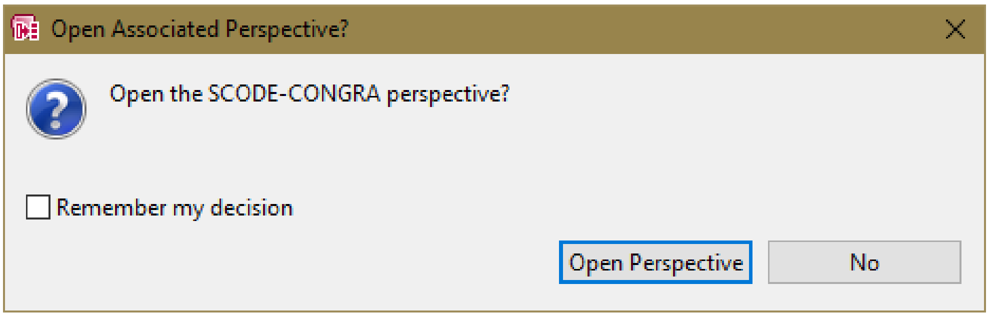
2. In the "New Example" window, select a SCODE-CONGRA example project and click **Next**.

   The selected project is listed.



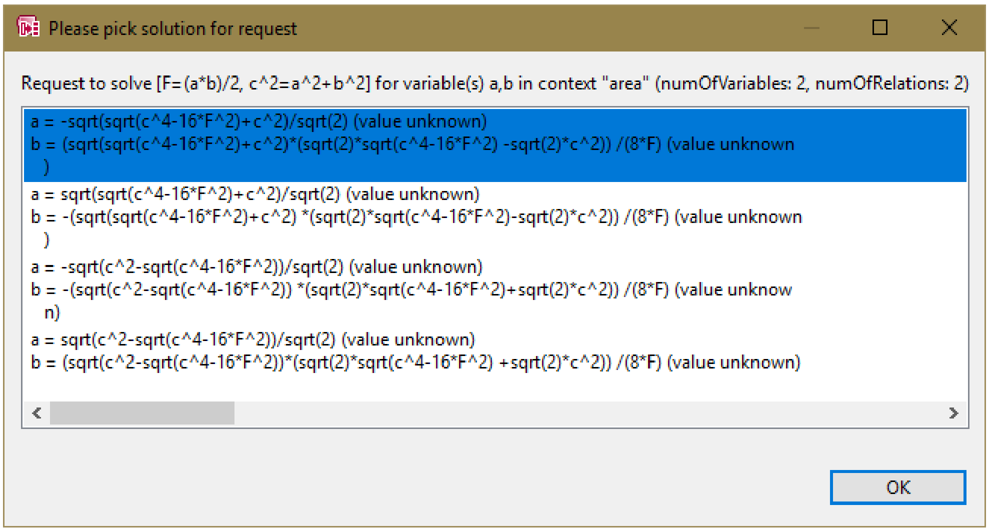
3. Click **Finish** to create the sample project.

   If the SCODE Workbench uses the SCODE-ANALYZER perspective, you are asked if you want to use the *SCODE-CONGRA perspective* instead. The SCODE-CONGRA perspective is a selection of views, tabs and pages optimized for SCODE-CONGRA.



4. Click **Open Perspective** to continue.

   You may be asked to select a solution.

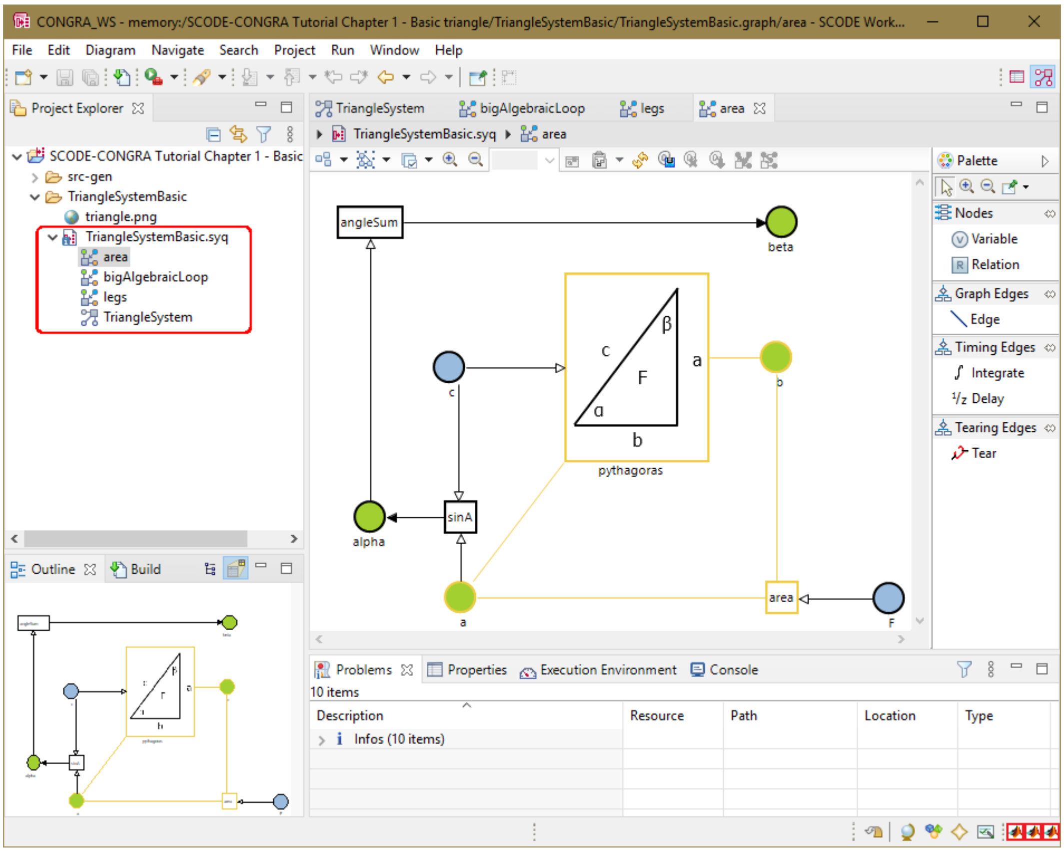


5. Click **OK** to accept the default selection.

   The example project is imported into your workspace. It is shown in the project explorer.

6. In the project explorer, open the `SCODE-CONGRA Tutorial Chapter 1 - Basic triangle` folder and double-click one of the entries below `TriangleSystemBasic.syq`.

   The selected graph opens.



You are now ready to discover or use SCODE-CONGRA!

6.3. Simulation in MATLAB®

NOTE

Working installations of MATLAB® and Simulink® are required.

Tests have been performed with versions R2016b, R2017b, R2018b, and R2019b.

To activate interaction with MATLAB for simulation, the connection between SCODE Workbench and MATLAB has to be configured.

6.3.1. Uninstall Old Connection to MATLAB®

If an old SCODE-CONGRA version (e.g., 1.5.0) is installed on the PC, make sure that first the *MLConnect Client* gets deleted manually. The default path was `C:\Users\<your user id>\Documents\MATLAB`. There, the following files and folder need to be deleted:

- `MATLABClient` folder
- `ETASConnect.m` file
- `sctLaunch.m` file

If the MLConnect Client was installed on a different path, make sure that the same files and folders are deleted from that path.

6.3.2. Connect Current Version

To connect SCODE Workbench and MATLAB®

1. In the SCODE Workbench window, select **Window > Preferences**.
2. In the "Preferences" window, go to the "MATLAB/Simulink" node.

   This node lists all MATLAB installations on your computer.

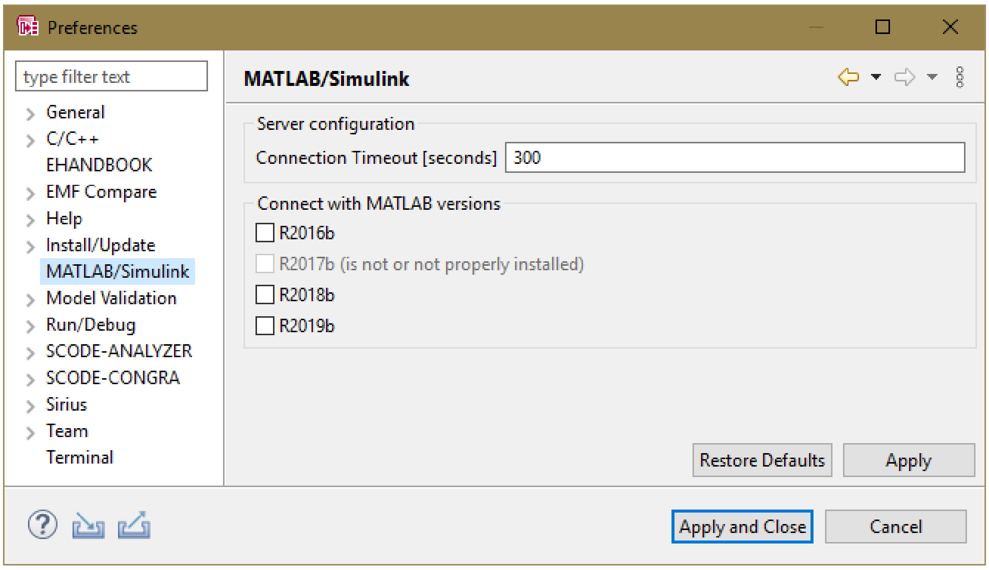


Figure 83. "Preferences" window, "MATLAB/Simulink" node

3. Select (☑) the MATLAB version(s) you want to connect.
4. Deselect (☐) the MATLAB version(s) you want to disconnect.
5. Click **Apply and Close**.

   A message window informs you about the result of the configuration process.

More details are given in the following parts of the online help (opened via **Help > Help Contents**):

- SCODE-ANALYZER User Guide, chapter "Tasks", section "Establish Connection between SCODE and MATLAB"
- SCODE-CONGRA User Guide, chapter "Tasks", section "Using MATLAB and Simulink for simulation"

7. Useful Information

This chapter contains useful information for working with the SCODE product family.

7.1. SCODE-ANALYZER: Generating TPT Test Cases

SCODE-ANALYZER can generate C code for a model. This C code can be tested with TPT [30] test cases, which are also generated by SCODE-ANALYZER.

TPT can only access global variables of the C code, and SCODE-ANALYZER only generates local variables in the generated functions. So, it is necessary to create another C code file that declares global variables which TPT can access, and calls the SCODE-ANALYZER-generated C code. This section explains how this code looks like and what steps are necessary to execute the test cases.

NOTE

It is recommended that you use a TPT version that contains the ***C/C++ Platform***. Versions prior to TPT 16 might not work, or would need some manual modification to the generated TPT file.

This section is based on TPT 17, it uses the C/C++ Platform.

7.1.1. SCODE-ANALYZER Project

You need a SCODE-ANALYZER project with working C code generation. This section uses the water tank example; To create an example project for SCODE-ANALYZER explains how to create the project.

To generate TPT test cases for such a project, meet the following requirements.

- The transition matrix must be complete and free of errors.
- The default transition behavior must be set to `non-transition`.

  You can set the behavior either for the entire workspace in the "Preferences" window (see To set the transition behavior) or for this project in the project properties (Figure 84). Close and re-open the `*.scode` file to see the change.

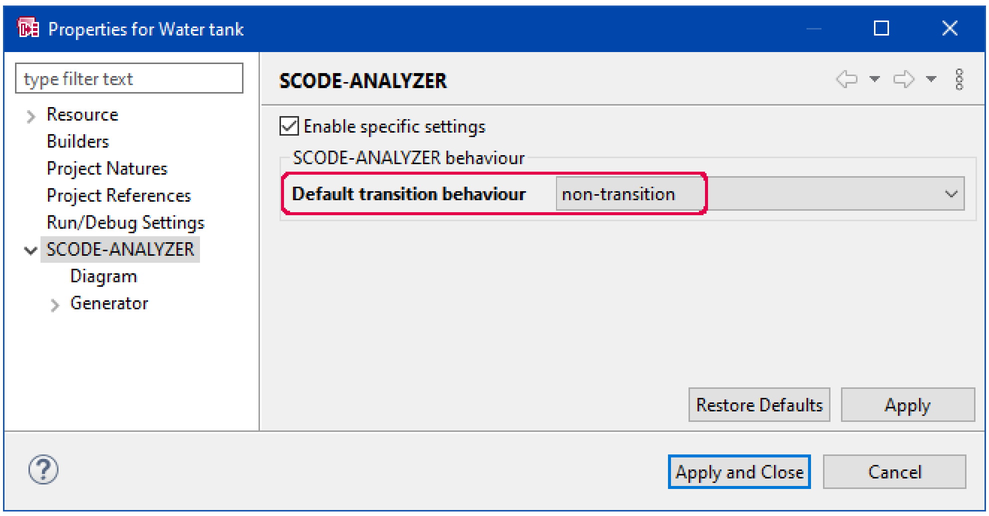


Figure 84. "Properties for *<project>*" window, "SCODE-ANALYZER" node

- The following settings in the "SCODE-ANALYZER\Generator" node are mandatory:
  - The "Generation Source" property must be set to `Mode Transition Matrix`.

    NOTE

    A "Generation Source" property set to `Mode Transition Matrix` means you have to specify the transition matrix correctly.

  - The "Dimension and Node representation" property must be set to `Use custom representation - default: Integer`.
  - C code generation must be activated.
  - As long as you do not focus on testing actions, the "Output type" property can be set to `Modes`.

  Figure 85 shows an example for generator settings that can be used with TPT test case generation.

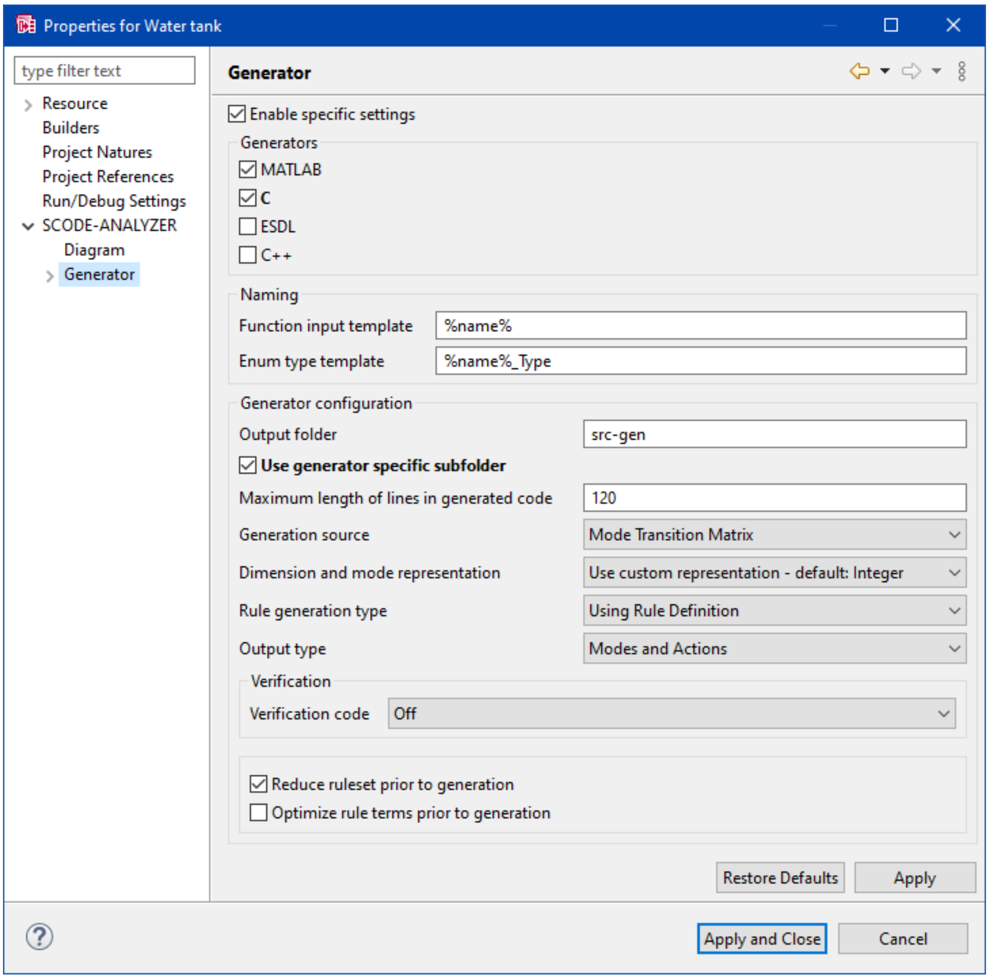
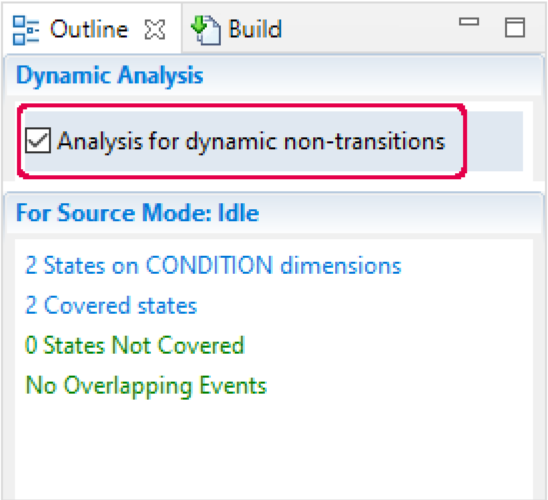


Figure 85. "Properties for *<project>*" window, "SCODE-ANALYZER\Generator" node

- The settings for "Dimension and Node representation" and "Output type" must be identical in the generator settings and in the run configuration (see Figure 86).

- **Analysis for dynamic non-transitions** must be activated in the "Outline" tab.



<u>To create a TPT test case</u>

SCODE-ANALYZER provides a special launch configuration for TPT test case creation.

1. Right-click the SCODE file and select **Run As > TPT (ANALYZER)** from the context menu.

   The "Edit Configuration" window opens.

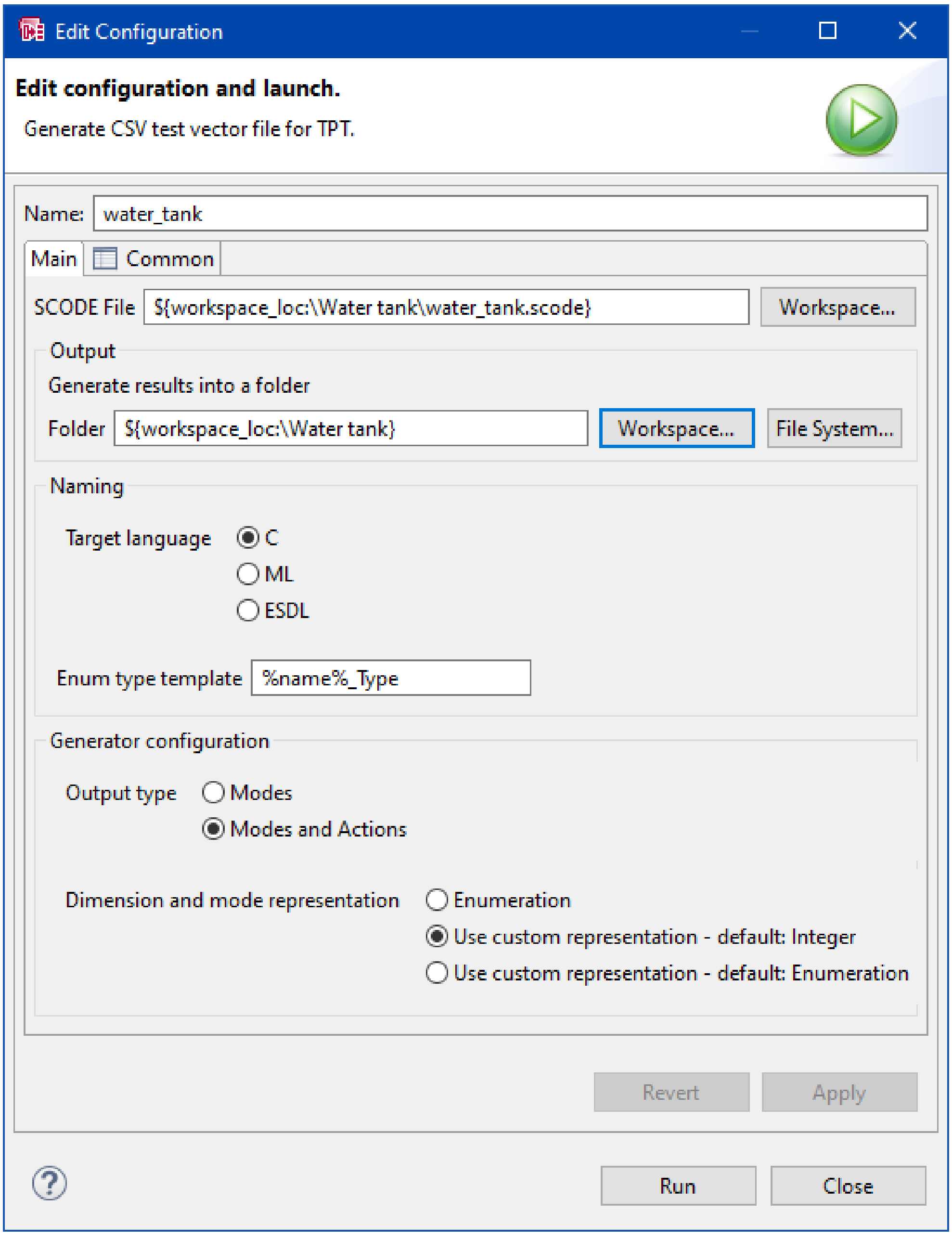
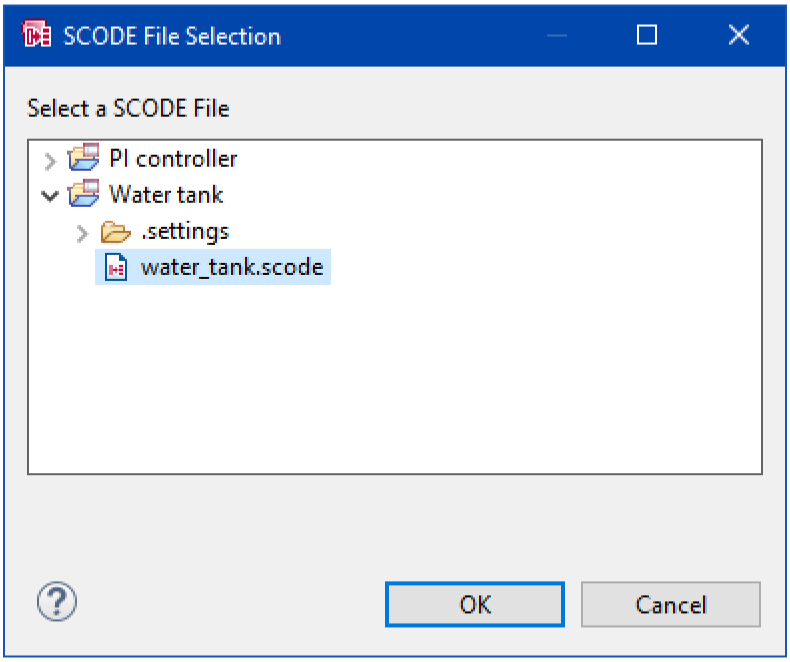


Figure 86. "Edit Configuration" window for the TPT launch configuration

2. To select the source for TPT test case generation, enter or select (via **Workspace** button) the `<scode file name>.scode` file in the "SCODE File" field.



3. To select an output folder, enter or select (via **Workspace** or **File System** button) a folder in your workspace.

   With the **Workspace** button, you can select a folder in your current workspace. With the **File System** button, you can select an output folder anywhere on your file system.

4. Activate the option for the desired output type.

   Modes ignores actions in the SCODE-ANALYZER project.
   Modes and Actions includes actions in the SCODE-ANALYZER project.

   The output type you select here must be the same as the output type selected in the generator settings (see also Figure 85).

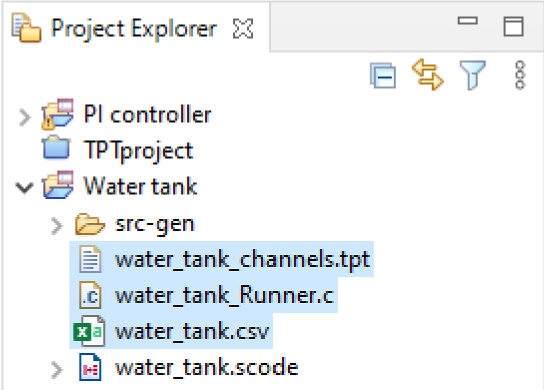
5. In the "Dimension and Node representation" area, activate Use custom representation - default: Integer.

   Other settings are currently not fully supported.

6. Click **Run** to create the test case.

The following files are created:

- *<scode file name>*_channels.tpt (see Table 18 for an example)
- *<scode file name>*_Runner.c (see Table 19 for an example)
- *<scode file name>*.csv (see Table 20 for an example)


Project Explorer
PI controller
TPTproject
Water tank
src-gen
water_tank_channels.tpt
water_tank_Runner.c
water_tank.csv
water_tank.scode

```
<?xml version="1.0" encoding="ASCII"?>
<tspec fileformatversion="19">
  <header>
    <declarations>
      <channels>
        <channel log="true" name="Water_level" role="output" type="uint8"
                 description="Water level"/>
        <channel log="true" name="Outlet_valve" role="input" type="uint8"
                 description="Outlet valve"/>
        <channel log="true" name="Pump" role="input" type="uint8"
                 description="Pump"/>
        <channel log="true" name="currentMode" role="output" type="uint8"
                 description=""/>
        <channel log="true" name="mode" role="input" type="uint8"
                 description=""/>
      </channels>
    </declarations>
  </header>
  <body>
    <testlet name="main">
      <content_ts/>
    </testlet>
    <assesslets>
      <assessletgroup name="Report">
        <assessletgroup name="Test Information">
          <assesslet name="Test Information" type="SectionReportAssessletType"/>
          <assesslet name="Report Meta Information"
                     type="MetaInformationReportAssessletType"/>
          <assesslet name="Timeout" type="timeout"/>
          <assesslet name="Report Linked Requirements" type="rmassesslet"/>
        </assessletgroup>
        <assessletgroup name="Signals">
          <assesslet name="Signals" type="SectionReportAssessletType"/>
          <assesslet name="Assesslet Summary"
                     type="AssessletSummaryReportAssessletType"/>
          <assesslet name="Assessments" type="SignalTableReportAssessletType">
            <assesslet_signalTableReport>
              <reportAssessletFilter_signalTable showAssessmentVariables="true"
                        showScriptVariables="true"/>
            </assesslet_signalTableReport>
          </assesslet>
          <assesslet name="Inputs" type="SignalTableReportAssessletType">
            <assesslet_signalTableReport>
              <reportAssessletFilter_signalTable showInputs="true"/>
            </assesslet_signalTableReport>
          </assesslet>
          <assesslet name="Outputs" type="SignalTableReportAssessletType">
            <assesslet_signalTableReport>
              <reportAssessletFilter_signalTable showOutputs="true"/>
            </assesslet_signalTableReport>
          </assesslet>
          <assesslet name="Parameters" type="SignalTableReportAssessletType">
            <assesslet_signalTableReport>
              <reportAssessletFilter_signalTable showMeasurements="true"/>
            </assesslet_signalTableReport>
          </assesslet>
          <assesslet name="System Constants" type="SignalTableReportAssessletType">
            <assesslet_signalTableReport>
              <reportAssessletFilter_signalTable showDeclSysConstants="true"/>
            </assesslet_signalTableReport>
          </assesslet>
        </assessletgroup>
        <assesslet name="Assesslets" type="SectionReportAssessletType"/>
      </assessletgroup>
      <assesslet name="Test Step List Assessments" type="StepListAssessments"/>
    </assesslets>
  </body>
</tspec>
```

Table 18. `water_tank_channels.tpt` (XML-based TPT project file for the water tank example)

```
/**
 * @warning        AUTOMATICALLY GENERATED FILE! DO NOT EDIT!
 *
 * @source         water_tank.scode
 *
 * @tool           SCODE-ANALYZER 3.1
 *
 * @options
 *   Generation source:                   Mode Transition Matrix
 *   Dimension and mode representation:   Use custom representation -
 *                                        default: Integer
 *   Output type:                         Modes and Actions
 *
 **/

#include "water_tank.h"

unsigned char currentMode;
unsigned char Water_level;
unsigned char mode;
unsigned char Outlet_valve;
unsigned char Pump;


void water_tank_Runner() {
    mode = water_tank_SelectorAction(currentMode, Water_level,
                                     &Outlet_valve, &Pump);
}   /* water_tank_Runner*/
```

Table 19. `water_tank_Runner.c` (C file to create global variables accessible to TPT)

```
time;currentMode;Water_level;Outlet_valve;Pump;mode;
1;0;1;0;0;0;
2;0;0;0;1;1;
3;0;2;1;0;2;
4;1;0;0;1;1;
5;1;1;0;0;0;
6;1;2;0;1;1;
7;2;2;1;0;2;
8;2;1;0;0;0;
9;2;0;1;0;2;
```

Table 20. `water_tank.csv` (contains test data)

C Code

Generate C code for the project. If you need help, see To generate code from the transition matrix. Without this C code, you cannot run the test.

The generated code is stored according to your settings. The following files are generated:

- `water_tank.c` (Table 21)
- `water_tank.h` (Table 22)

1

```
/**
 * @warning       AUTOMATICALLY GENERATED FILE! DO NOT EDIT!
 *
 * @source        water_tank.scode
 *
 * @tool          SCODE-ANALYZER 3.1
 *
 * @options
 *   Generation source:                   Mode Transition Matrix
 *   Dimension and mode representation:   Use custom representation -
 *                                        default: Integer
 *   Rule generation type:                Using Rule Definition
 *   Output type:                         Modes and Actions
 *
 * @metrics:
 *    Number_of_modes:          3
 *    Number_of_mode_rules:     3
 *    Number_of_events:         5
 *    Number_of_event_rules:    5
 *    Number_of_transitions:    6
 **/

#include "water_tank.h"
```

2

```
unsigned char water_tank_ModeSelector(unsigned char currentMode,
              unsigned char Water_level) {
  unsigned char mode = 0U;
  switch (currentMode) {
    case 0U:
      if ((Water_level == 0U)) {
        mode = 1U;
      } else if ((Water_level == 2U)) {
        mode = 2U;
      } else {
        mode = 0U;
      }
      break;

    case 1U:
      if ((Water_level == 1U)) {
        mode = 0U;
      } else {
        mode = 1U;
      }
      break;

    case 2U:
      if ((Water_level == 1U)) {
        mode = 0U;
      } else {
        mode = 2U;
      }
      break;

    default: {
      mode = currentMode;
      break;
    }
  }  /* switch (input_currentMode)*/
  return mode;
}  /* water_tank_ModeSelector*/
```

3	`unsigned char water_tank_Action(unsigned char mode, unsigned char * Pump) {`
	`  unsigned char Outlet_valve = 0U;`
	`  switch (mode) {`
	`    case 0U:`
	`      /* case for Idle */`
	`      Outlet_valve = 0U;`
	`      *Pump = 0U;`
	`      break;`
	
	`    case 1U:`
	`      /* case for Fill-up */`
	`      Outlet_valve = 0U;`
	`      *Pump = 1U;`
	`      break;`
	
	`    default: {`
	`      /* case for Drain */`
	`      Outlet_valve = 1U;`
	`      *Pump = 0U;`
	`      break;`
	`    }`
	`  } /* switch (mode)*/`
	`  return  Outlet_valve;`
	`}  /* water_tank_Action*/`

4	`unsigned char water_tank_SelectorAction(unsigned char currentMode,`
	`              unsigned char Water_level, unsigned char * Outlet_valve, unsigned`
	`char * Pump) {`
	`    unsigned char mode = 0U;`
	`    mode = water_tank_ModeSelector(currentMode, Water_level);`
	`    *Outlet_valve = water_tank_Action(mode, Pump);`
	`  return mode;`
	`}  /* water_tank_SelectorAction*/`

Table 21. `water_tank.c` (C file generated for the water tank example)

```
/**
 ...
 **/

#ifndef WATER_TANK_H
#define WATER_TANK_H

extern unsigned char water_tank_ModeSelector(unsigned char
                    currentMode, unsigned char Water_level);

extern unsigned char water_tank_Action(unsigned char mode,
                     unsigned char * Pump);

extern unsigned char water_tank_SelectorAction(unsigned char
                    currentMode, unsigned char Water_level,
unsigned char * Outlet_valve,
                    unsigned char * Pump);

#endif /* WATER_TANK_H */
```

Table 22. `water_tank.h` (corresponding header file for `water_tank.c`)

The **`water_tank_ModeSelector`** function (Table 21, block 2) contains the mode logic. The **`water_tank_Action`** function (Table 21, block 3) contains the action logic. The **`water_tank_SelectorAction`** function (Table 21, block 4) calls the other two.

As the variables are all local variables, the `water_tank_Runner.c` file is needed, which is created during TPT project generation. That file (see Table 19) defines the inputs (`Water_level`, `currentMode`) and the output (`mode`) as global variables. The `water_tank_Runner` function is the function that calls the mode selector generated by SCODE-ANALYZER. The included header file is used to make the code know about the function calls of the mode selector.

7.1.2. Working in TPT

This section describes how to set up the TPT test project, using the `*.tpt` file (see To create a TPT test case) and the generated C code (see C Code).

7.1.2.1. Preparations

TPT needs to know the compiler it is supposed to use.

To create a compiler configuration

1. Start TPT.
2. If the TPT "Welcome" page opens, click **New** to create a new, empty project.
3. In the TPT window, select **Options > Preferences**.
4. In the "Preferences" window, go to the "General\C Compiler" node.
5. Do one of the following:
    - Make sure that your compiler configuration is correct.
    - Add a new compiler configuration.

NOTE

TPT can use only compilers that are configured in the "General\C Compiler" node.

The CTC++ code coverage check requires a *Visual Studio* compiler.

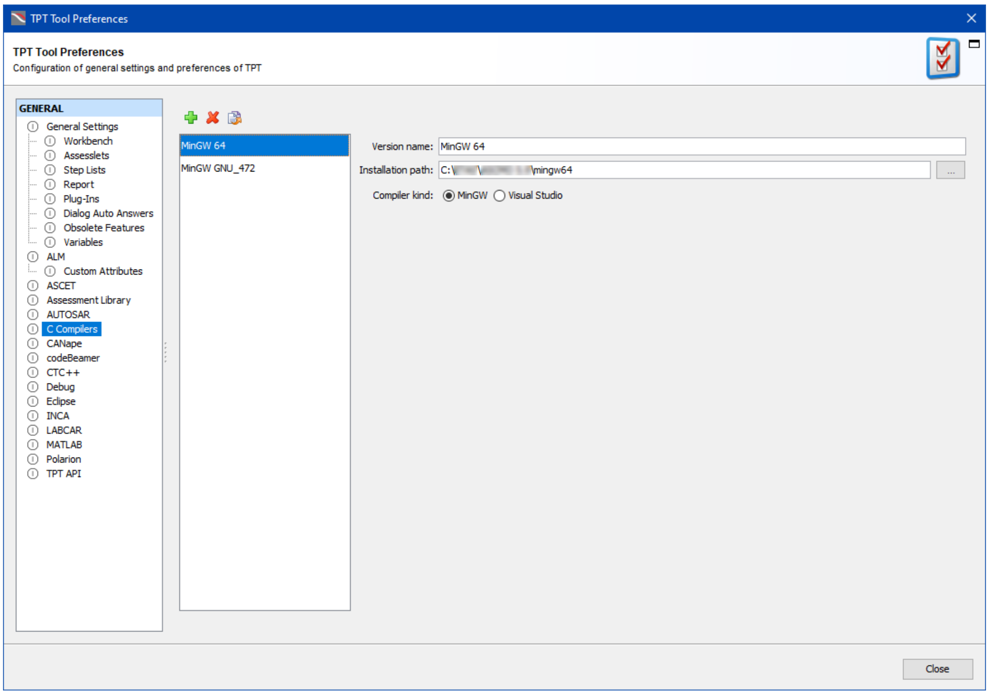


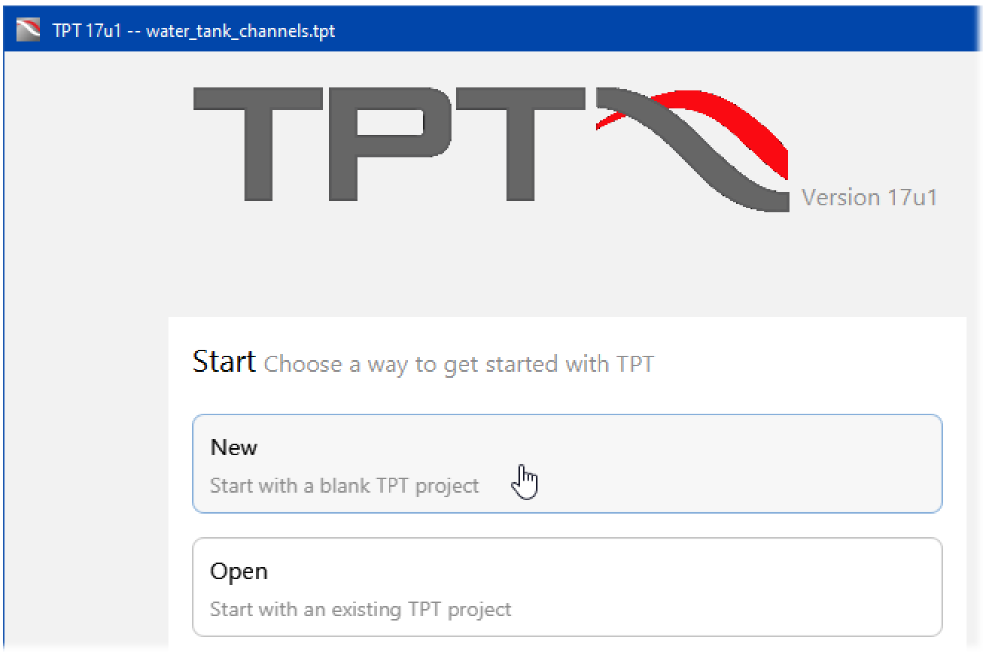
Figure 87. TPT "Preferences" window, "General\C Compiler" node

6. Close the "Preferences" window.

7.1.2.2. TPT Project

To create the TPT project

1. Start TPT.
2. If the TPT "Welcome" page opens, click **New** to create a new, empty project.



3. If desired, close the empty project.

4. In the main TPT window, select **File > Open**.
5. In the file selection window, select the `*.tpt` file you created with {project-nameA}, then click **Open**.

   The `*.tpt` file is imported. The following tree is shown in the TPT "Project" tab:

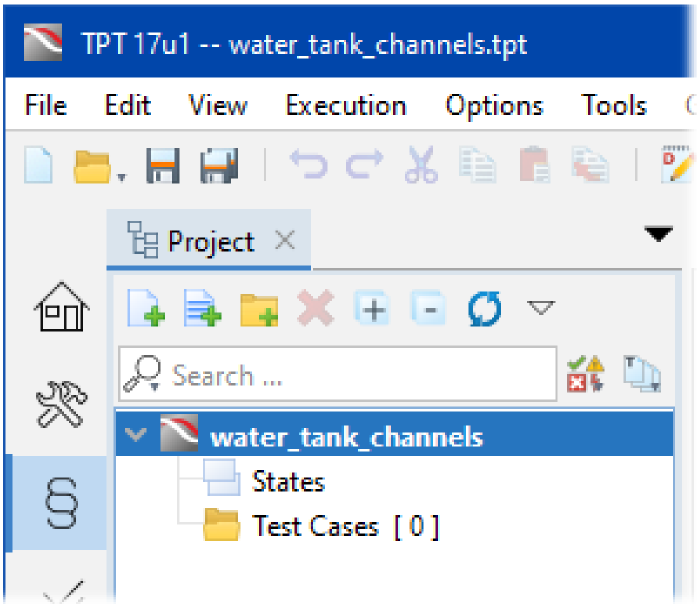


Figure 88. `*.tpt` file opened in TPT

6. Click **Save**.

   You are warned that the TPT file was created using a previous version of TPT. If you proceed, the file format will be changed to TPT 17u1 and the file cannot be opened with older versions of TPT any longer.
7. Click **Yes** to save the project.

To execute the test cases, three steps are required:

A. Generate a test case; see Test Case.

B. Configure the platform; see Platform.

C. Execute the test; see Execution.

Test Case

To generate a test case

1. In the main TPT window, select **Generate Test Cases > from Test Data**.

   The "Generate Test Cases from Test Data" window opens.
2. In the "Data directory" field, enter or select (via the **Browse** button) the directory that contains your test case files (see To create a TPT test case).

3. Make sure that the following values are set:

Parameter	Value	Remarks
"File pattern"	`*.csv`	The `*.csv` file (see Table 20) contains the test data.
Embed signal data	activated	
Create signal comparison assesslets	activated	
Create local reference channels	activated	
Add termination condition	activated	
Assign values to matching parameters	activated	

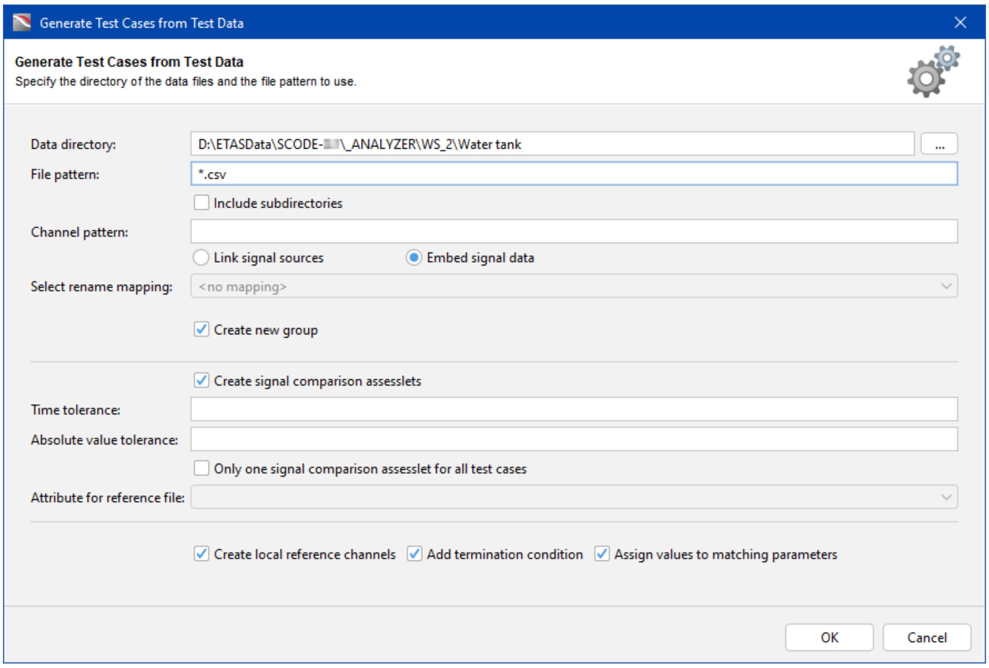
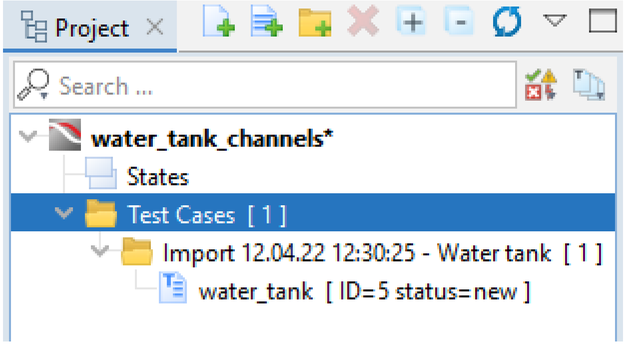


Figure 89. TPT "Generate Test Cases from Test Data" window

4. Click **OK**.

   The test case is added to the TPT project.



5. Double-click the test case to see it in the TPT "Contents" tab.
6. In the "Contents" tab, select a signal to display it in the lower half of the "Content" tab.

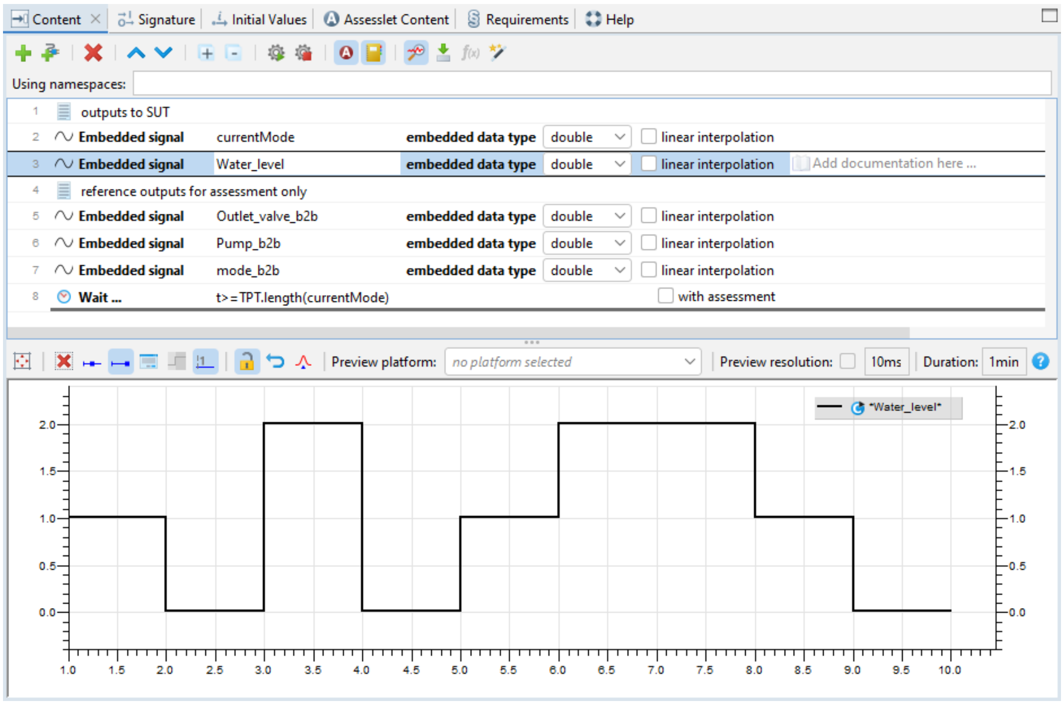


Figure 90. TPT "Content" tab with test case and signal

Platform

The *C/C++ Platform* allows you to directly test C code from source files by automatically generating an appropriate test frame. You can import functions and global variables via C files, and you can set up step size and timeout of the test.

Step size

Sampling time, i.e., a constant time between one simulation step and the subsequent one.

Timeout

Maximum test execution time before test execution stops.

Assume a test data file with 100 steps, a step size of `1s`, and a timeout of `1min`. When you run the test, the simulation stops after 60 seconds □ 60 steps, and the remaining steps are ignored steps. If the test data files has less than 60 steps, the simulation stops after the last step.

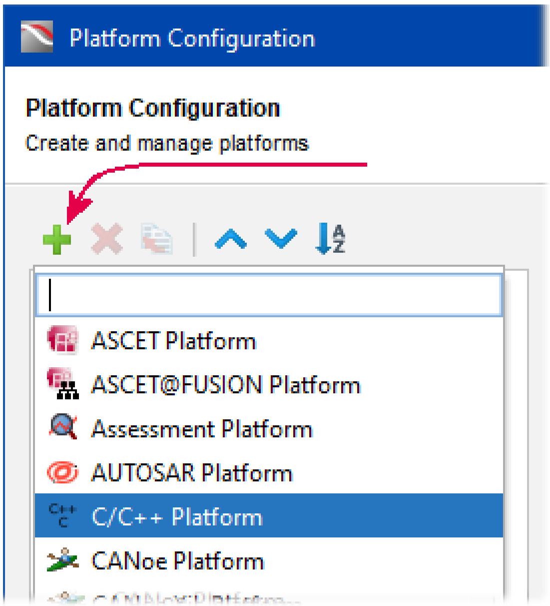
So, when you set up the platform (see step 6 below), make sure that the maximum execution time contains all steps. You may add some extra time, just to be sure.

To configure the platform

1. In the main TPT window, select **Execution > Platform Configuration**.

   The "Platform Configuration" window opens.

2. In that window, click the **Add Platform Configuration** button and create a new `C/C++ Platform`.

   The platform is shown in the "Platform Configuration" window (Figure 91).

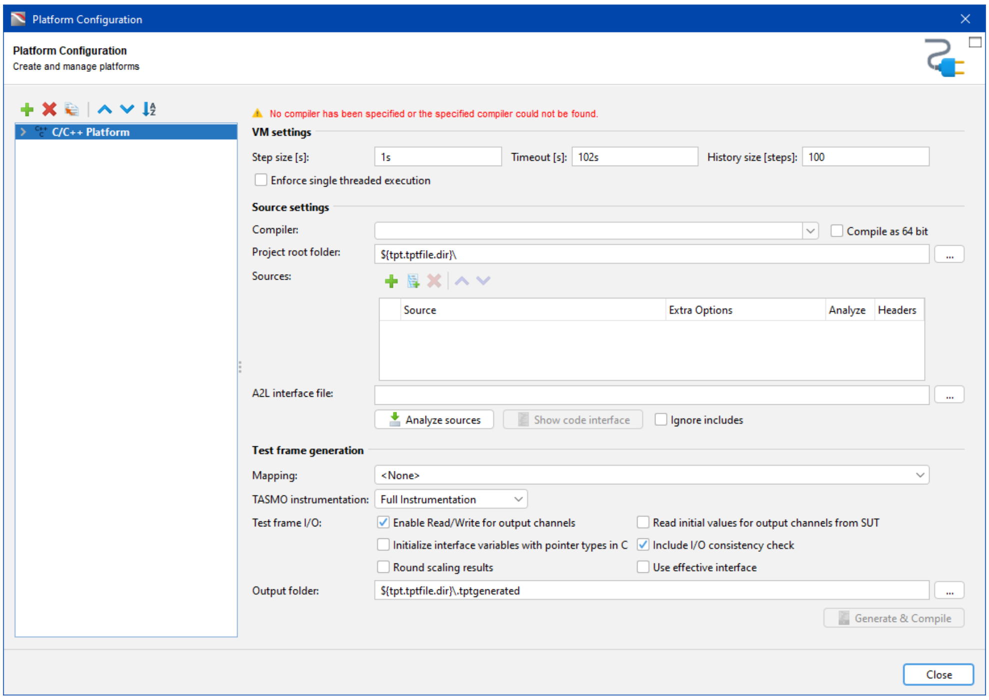
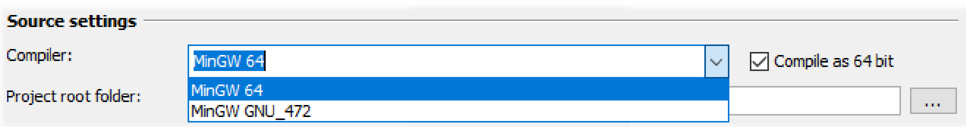


Figure 91. "Platform Configuration" window with newly created platform

3. In the "Compiler" combo box, select the compiler you want to use.

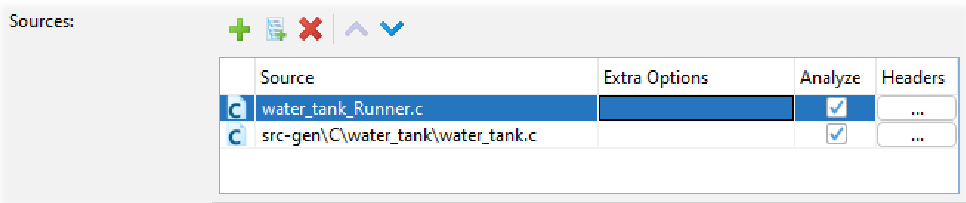
NOTE Only compilers defined in the "Preferences" window (see To create a compiler configuration) are available.

4. If you are using a 64 bit compiler, activate the **Compile as 64 bit** option.

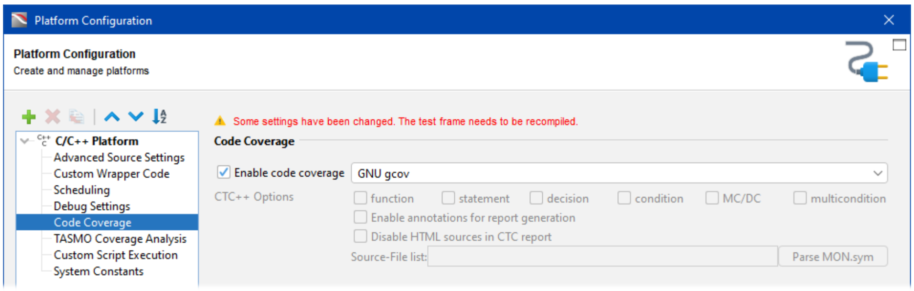


5. In the "Sources" area, add the C files.

   In the example, `water_tank_Runner.c` (generated with the TPT file; see To create a TPT test case) and `water_tank.c` (see also C Code) are added.



6. Configure "Step size" and "Timeout".
7. Configure other parameters.
8. To activate code coverage check, do the following:
   i. Go to the "Code Coverage" node and activate **Enable code coverage**.



ii. In the combo box next to the option, select the code coverage tool.

NOTE

`CTC++` code coverage check requires a *Visual Studio* compiler. For other compilers, select `CNU gcov`.

The "C/C++ Platform" node of the "Platform Configuration" window should look as follows:

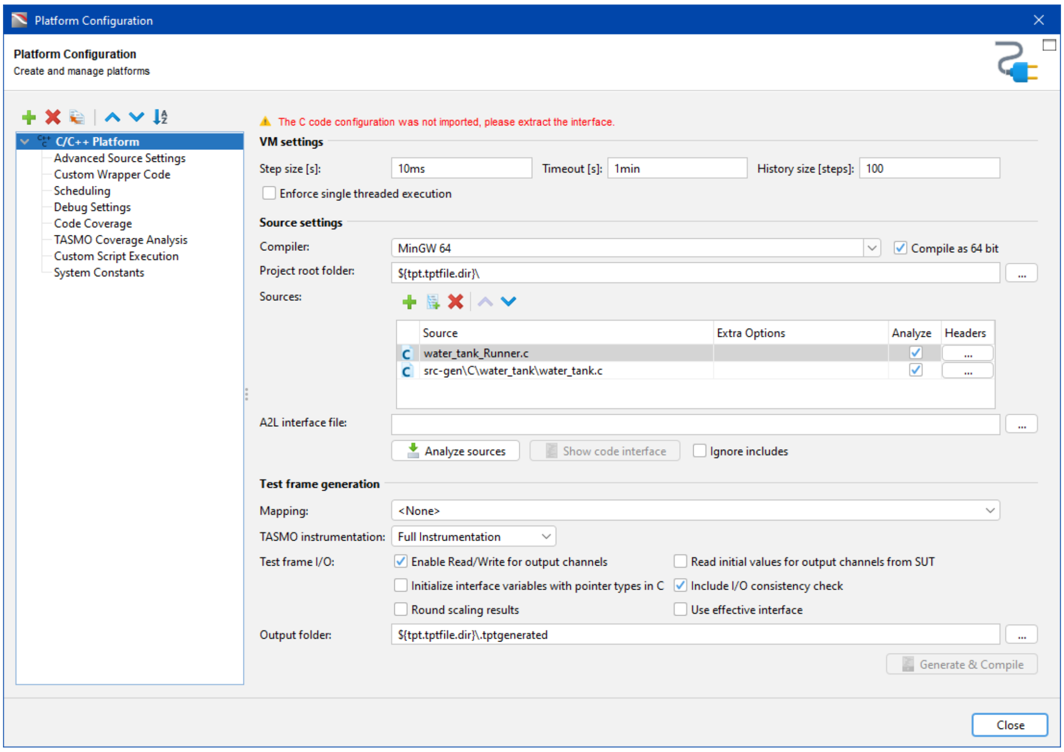


Figure 92. "Platform Configuration" window with configured platform

Next, the variables and functions are analyzed, and the interface is exported.

To import the interface

1. In the "C/C++ Platform" node of the "Platform Configuration" window (Figure 92), click the **Analyze sources** button.

   The "Code interface" window opens. It lists the added C files and their elements.

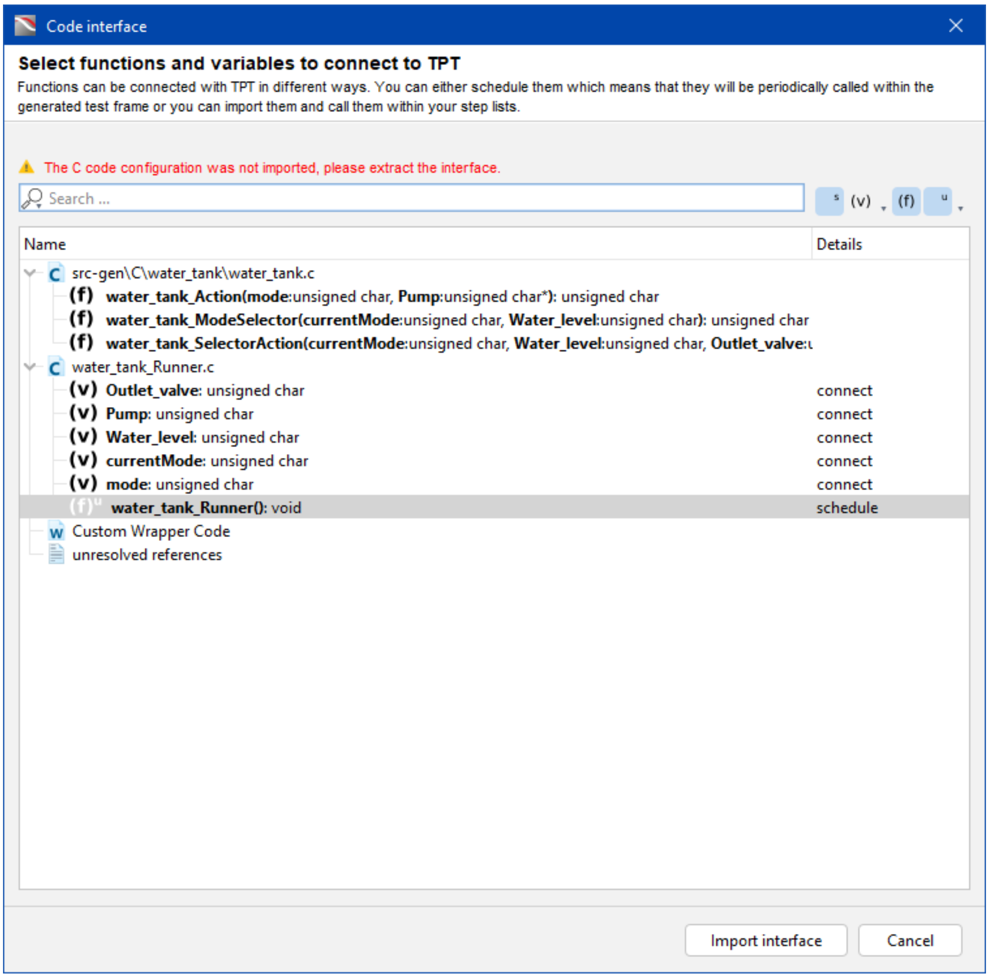
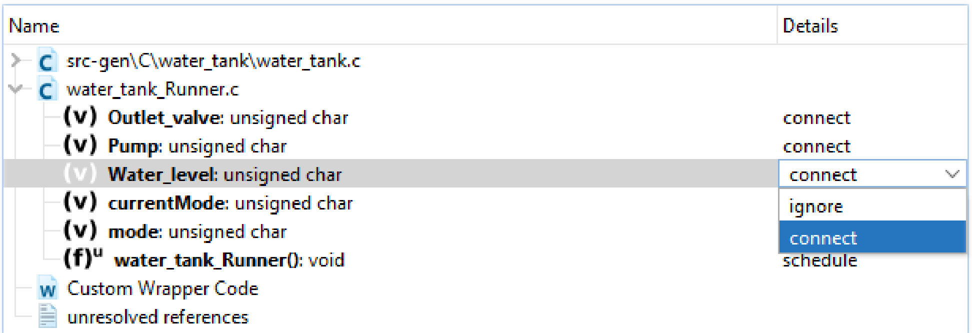


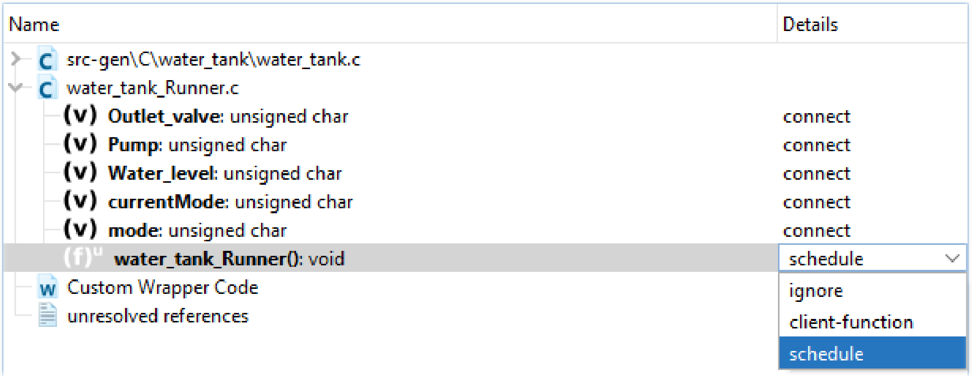
Figure 93. "Code interface" window

2. Make sure that the global variables in the `*_Runner.c` file are connected.



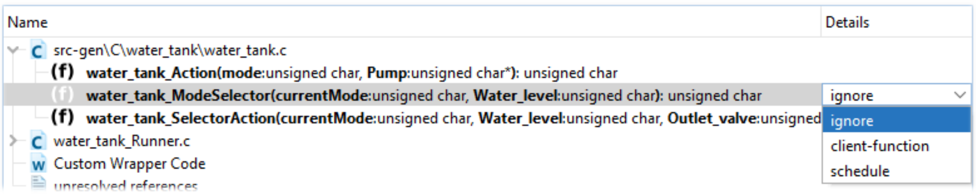
TPT can access only connected variables.

3. For the function in the `*_Runner.c` file, select `schedule`.



This function (`water_tank_Runner` in the example) calls the function under test. With the `schedule` setting, it is executed periodically.

4. For the functions in the `<scode file name>_.c` file (see Table 21 for an example), select `ignore`.



5. Click **Import interface**.

   The "Import Interface" window opens. It displays the information found in the C files.

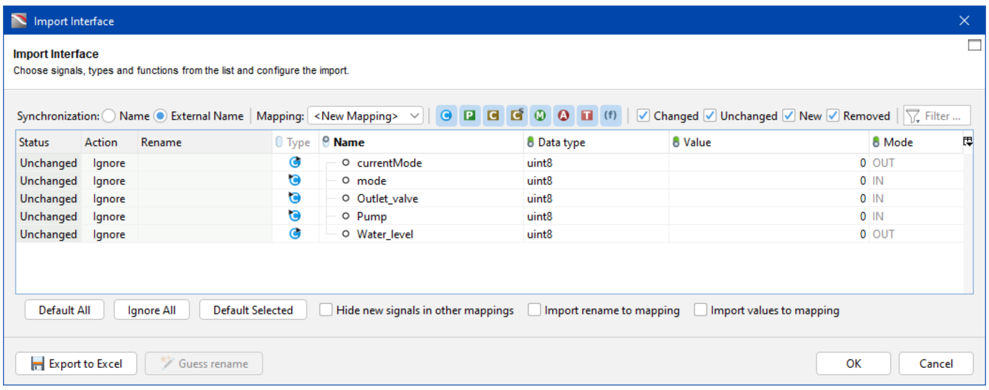


Figure 94. "Import Interface" window

6. Click **Default All** to select the actions TPT suggests for the elements.
7. Click **OK** to close the "Import Interface" window and import the interface.

   If the import was successful, a message window informs you that the functions list was updated.

8. Click **OK** to close the message window.

9. In the "Platform Configuration" window, click **Generate and compile test driver**.
10. If no errors occurred, close the "Platform Configuration" window.

    If errors occurred, refer to the TPT documentation for help.

Execution

Once test driver generation was successful, you can execute the test cases.

To execute test cases

1. In the main TPT window, select **Execution > Execution Configuration**.
2. In the "Execution Configuration" window, edit the configuration according to your needs.

   The example uses default settings.

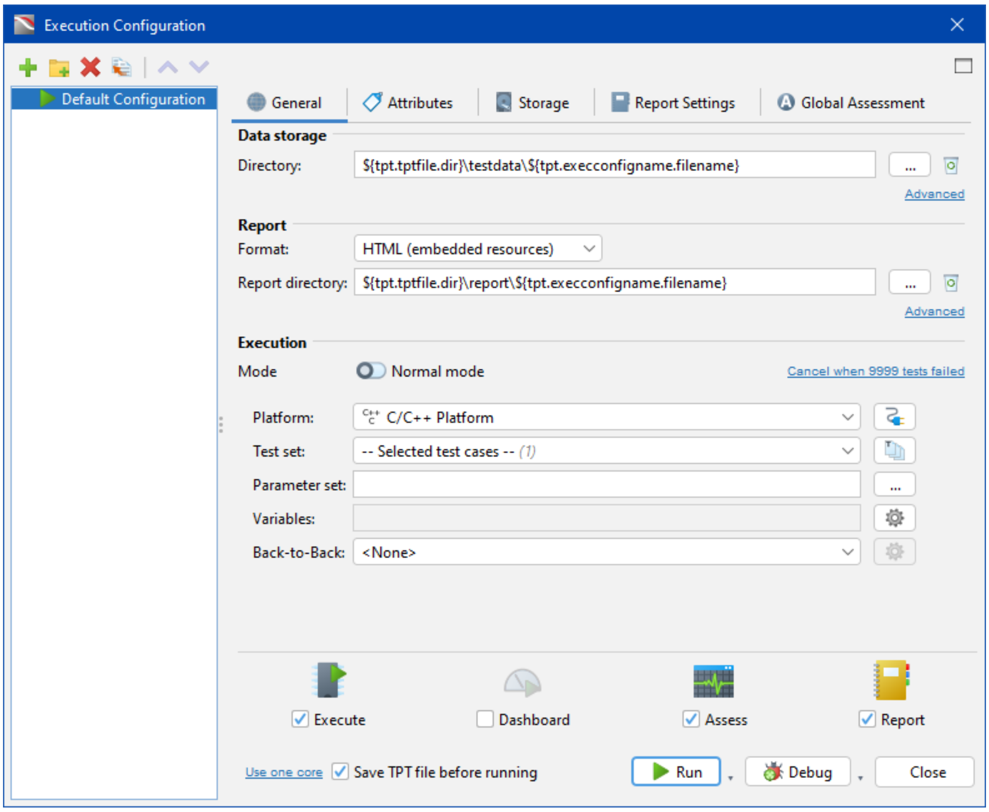


Figure 95. "Execution Configuration" window

3. If desired, activate **Save TPT file before running**.
4. Click **Run**.

   If a warning regarding file format changes opens, click **yes** to continue.

The tests are executed. Results are displayed in the "Build Progress" window. Passed tests are marked with green hooks (Done. ; see also Figure 96), failed tests are marked with red flashes (Done (execution error).).

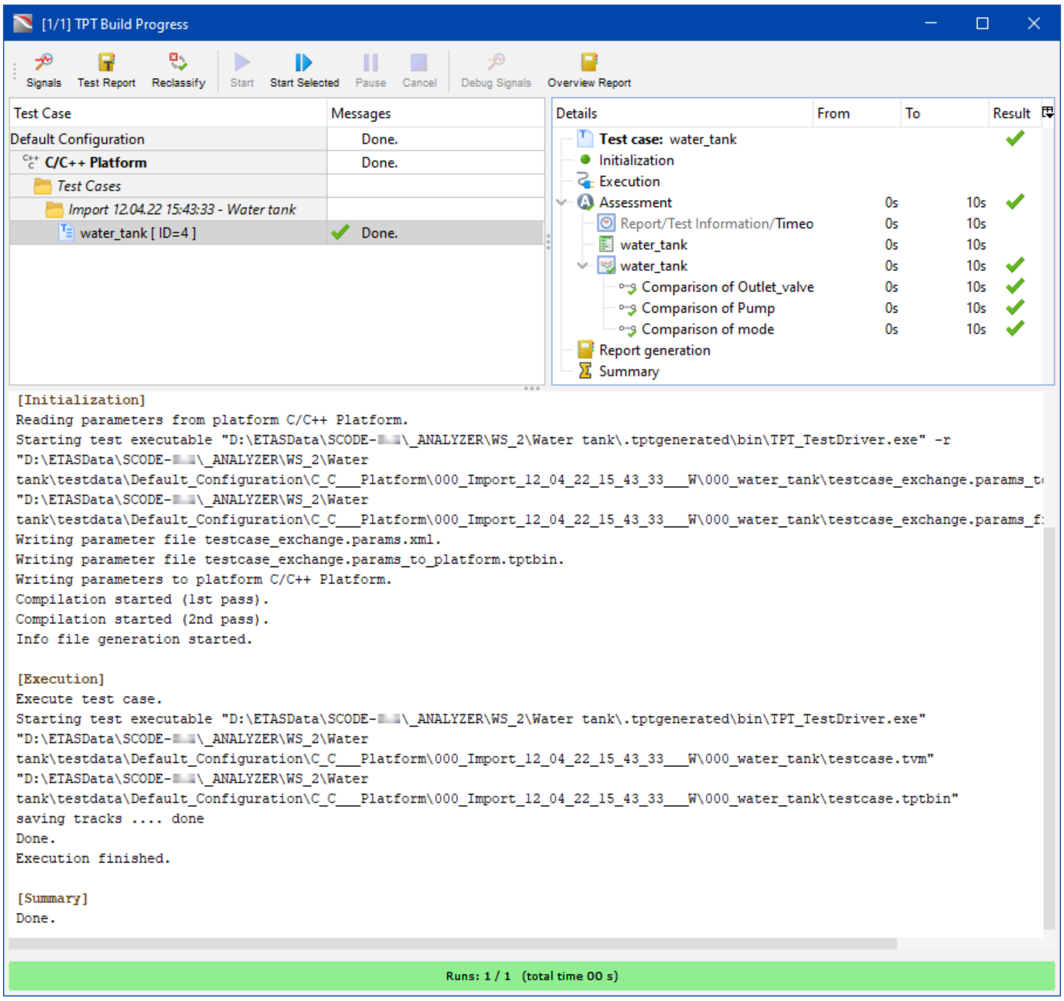


Figure 96. "TPT Build Progress" window, all tests passed

The * **Report** buttons become available when the report is created. **Test Report** opens the test report in a web browser. The **Overview Report** button creates and opens an overview report, which provides summary information and a link to the test report.

To analyze the signals, select a test case and click the **Signals** button [31] in the toolbar of the "TPT Build Progress" window to open the TPT Signal Viewer. See the TPT online help for further information.

7.2. SCODE-CONGRA: Colors

Color	Meaning	Example
Fill colors		
light grey	undefined node	see Example: Explicit Output, Unused Nodes
green1	free variable	see Example: Inputs, Implicit Outputs, Algebraic Loop
light grey	parameter	see Example: Parameter, Fixed Variable
orange1[a]	argument	
blue1	input	see Example: Inputs, Implicit Outputs, Algebraic Loop
green2[a]	tearing variable	
white	relation	see Example: Inputs, Implicit Outputs, Algebraic Loop
dark grey[a]	relation with subsystem	
light yellow[a]	relation with char. table/map	
orange2[a]	relation with conditional equation subsystem	
pink[a]	tearing relation	
Edge/border colors		
black	normal edge	see Example: Inputs, Implicit Outputs, Algebraic Loop
blue2	underconstrained (sub-)graph	see Example: Underconstrained and Overconstrained
red	overconstrained (sub-)graph	
yellow	algebraic loop	see Example: Inputs, Implicit Outputs, Algebraic Loop
brown[a]	subgraph with intrinsic BNS	
pink[a]	teared algebraic loop	
rosy[a]	algebraic loop in teared algebraic loop	

a: See the SCODE-CONGRA User Guide for more information.

Table 23. SCODE-CONGRA graphs — CONGRA Classic colors and meanings

Example: Inputs, Implicit Outputs, Algebraic Loop

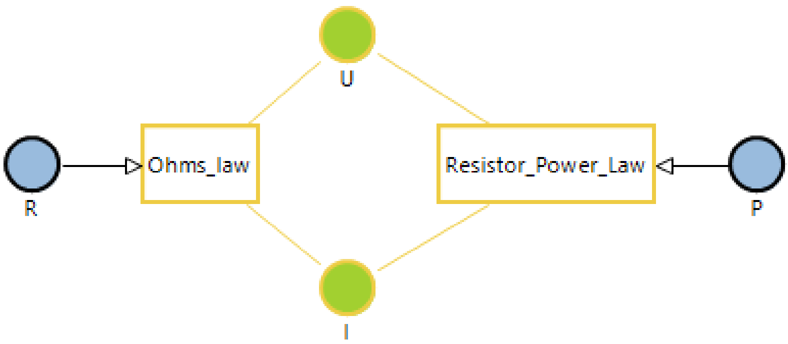


Figure 97. Flow with inputs, implicit outputs, and algebraic loop

The flow in Figure 97 contains the following elements:

- variables I, P, R, U
- relation `Ohms_law`: U = R * I
- relation `Resistor_Power_Law`: P =U * I

Variables R and P are marked as inputs. They use the fill color blue1 and thin black borders.

Variables U and I are free, but they can be computed. Therefore, they use fill color green1.

The relations use the fill color white.

Both relations and the variables I and U form the algebraic loop. They, as well as the connections between them, use the border color yellow.

Example: Explicit Output, Unused Nodes

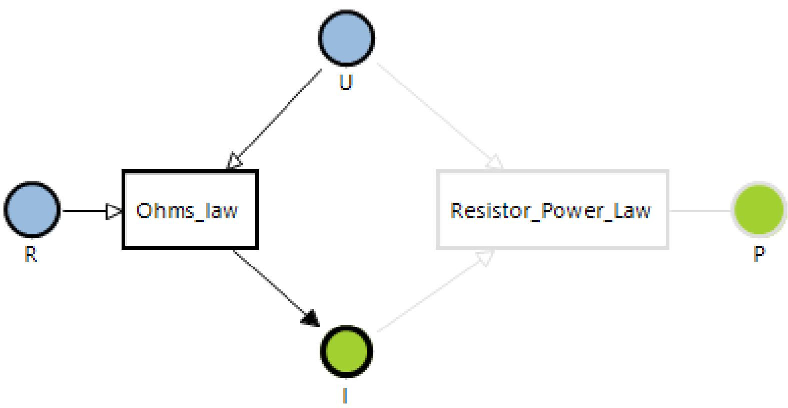


Figure 98. Flow with relations, inputs, explicit output, and unused parts

The flow in Figure 98 contains the same elements as the flow in Figure 97.

Variable I is marked as output. It uses fill color green1 and a thick black border.

Relation `Resistor_Power_Law` and variable P are not required to compute the explicit output I. Therefore, they, and the connecting edges, use light grey as border color.

Variables P is free, but it can, in principle, be computed. Therefore, it uses fill color green1.

Example: Parameter, Fixed Variable

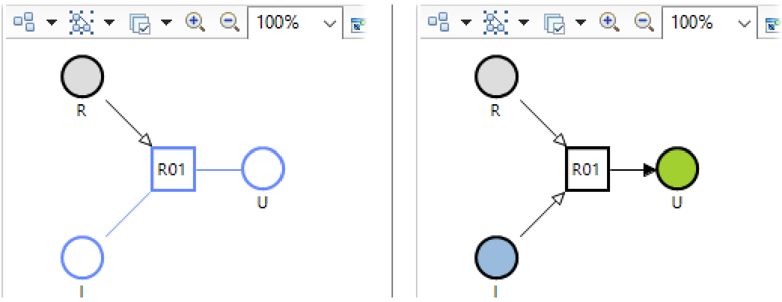


Figure 99. System graph (left) and flow (right) with relation, parameter, variables

System graph and flow in Figure 99 contain the following elements:

- variables I, U
- parameter R
- relation R01: U = R * I

R is specified as parameter, it uses the fill color light grey, both in the system graph and in the flow.

A fixed variable uses the fill color light grey in a flow, but it looks like free variables in the system graph. See Figure 55 for an example.

Example: Underconstrained and Overconstrained

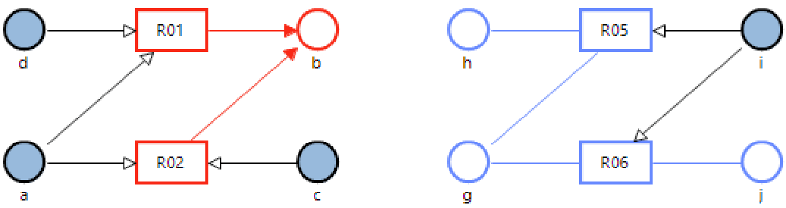


Figure 100. Flow with underconstrained and overconstrained parts

The flow in Figure 100 contains the following elements:

- variables a, b, c, d, g, h, i, j
- relation R01: a = b + 2 + d
- relation R02: 3*b = a + c
- relation R05: g = h - i
- relation R06: g - 2*i = j

Variable b is determined by R01 and R02, i,e, b is overconstrained. Therefore, the borders of b, R01, R02 and the connecting edges use the border color red.

One input variable, i, is not sufficient to compute free variables g, h and j; g, h, and j are underconstrained. Therefore, the borders of g, h, j, R05, R06 and the connecting edges use the border color blue2.

7.3. SCODE Workbench: Installing Yakindu Traceability

This section describes the installation of Yakindu Traceability in the SCODE Workbench.

NOTE

Yakindu Traceability is a requirements traceability management tool created and sold by *itemis AG*.

For any information beyond how to install Yakindu Traceability into SCODE Workbench, please contact www.itemis.com/.

When you buy Yakindu Traceability, you will receive a ZIP file, the *YT repository*. Unzip that repository to a local folder on your PC.

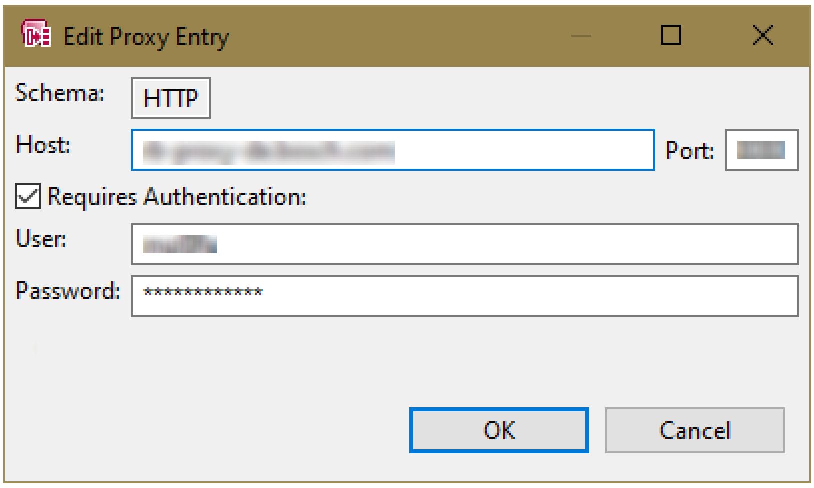
To set up SCODE Workbench for Yakindu Traceability installation

1. Start the SCODE Workbench.
2. Select **Window** → **Preferences**.
3. In the "Preferences" window, go to the "General\Network Connections" node and do the following:

   i. Set the "Active Provider" to `Manual` (A in Figure 101).

   ii. Select the `HTTP` schema (B in Figure 101) and click the **Edit** button.

   iii. In the "Edit Proxy Entry" window, enter host, port, your user and your password, then click **OK**.



   iv. Edit the `HTTPS` (C in Figure 101) schema in the same way.

   v. Click **Apply**.

The "General\Network Connections" node should look as follows:

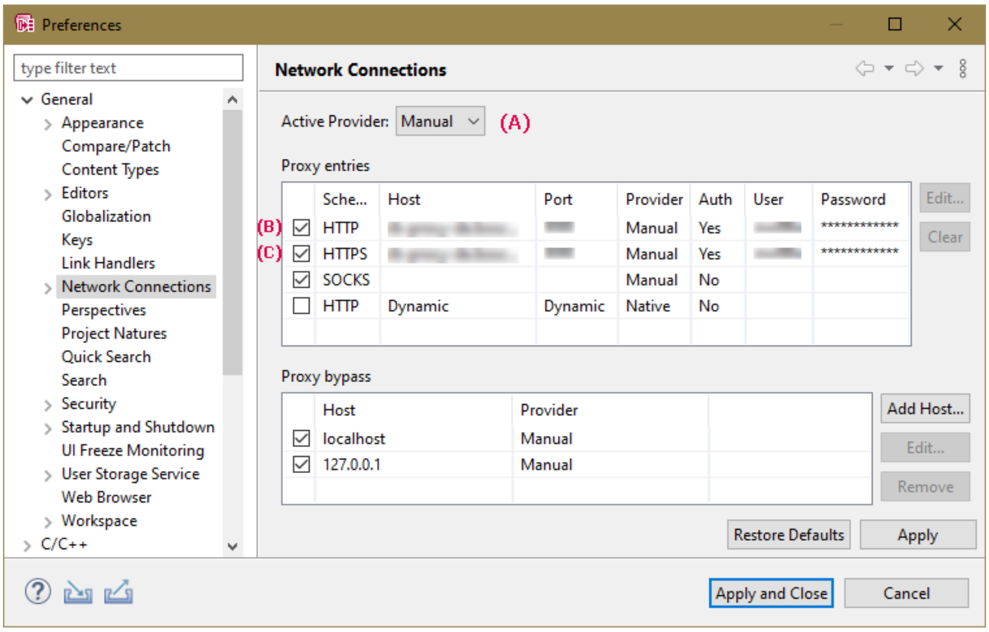


Figure 101. "Preferences" window, "General\Network Connections" node

4. Go to the "Install/Updates\Available Software Sites" node and make sure that *only* the following update sites are enabled:
   - Eclipse Luna for BIRT 4.4.2

     (available at download.eclipse.org/releases/luna/)
   - Eclipse (current TP version)

     (available at download.eclipse.org/releases/2020-09/)

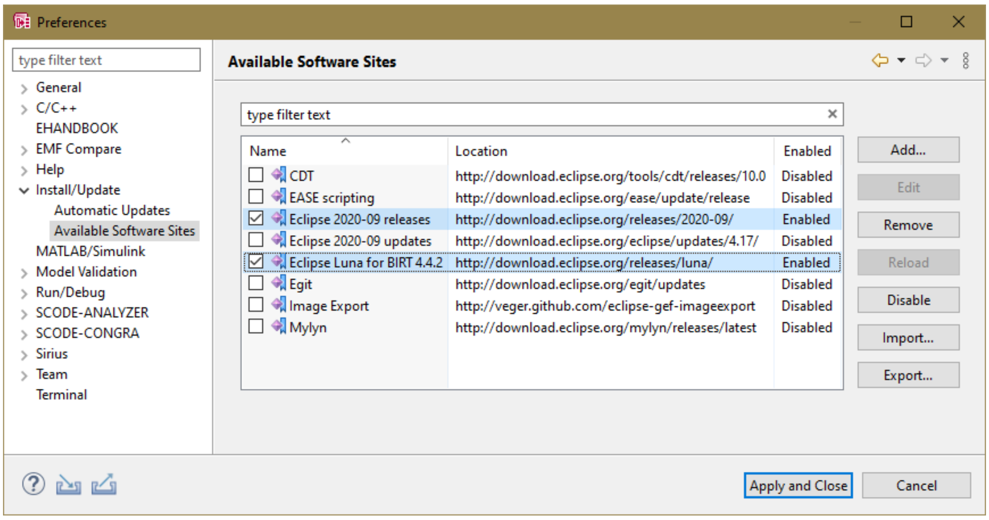
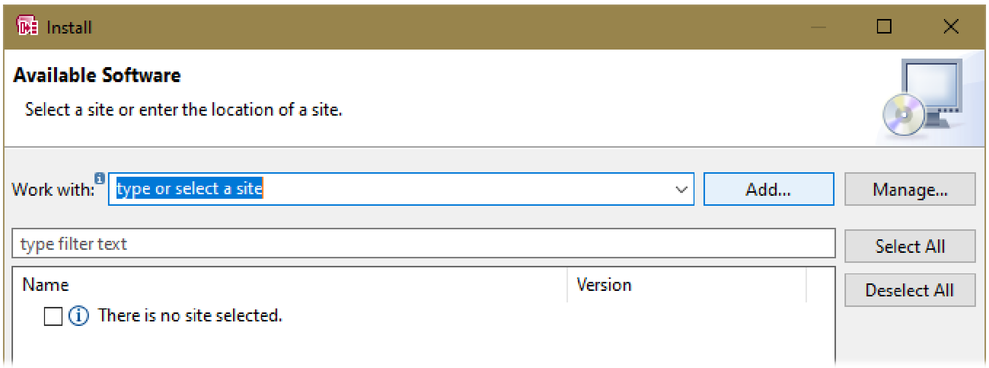


Figure 102. "Preferences" window, "Install/Update\Available Software Sites" node

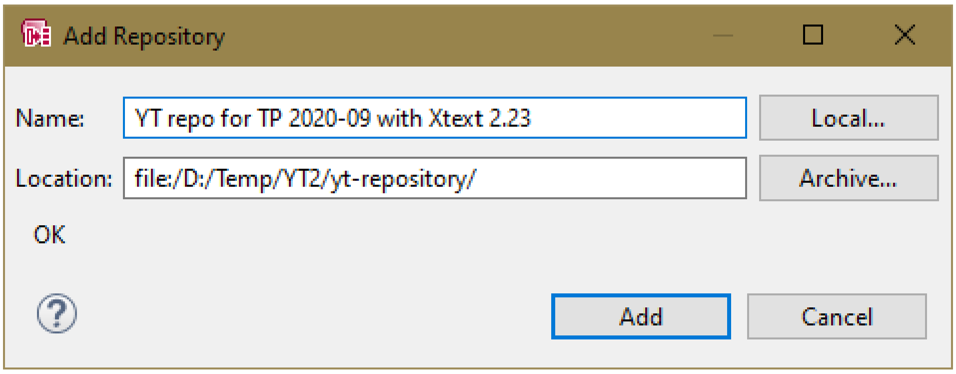
5. If they are not listed, use the **Add** button to add the missing site(s).
6. Click **Apply and Close**.

To install Yakindu Traceability into SCODE Workbench

1. In the SCODE Workbench window, select **Help** → **Install New Software**.

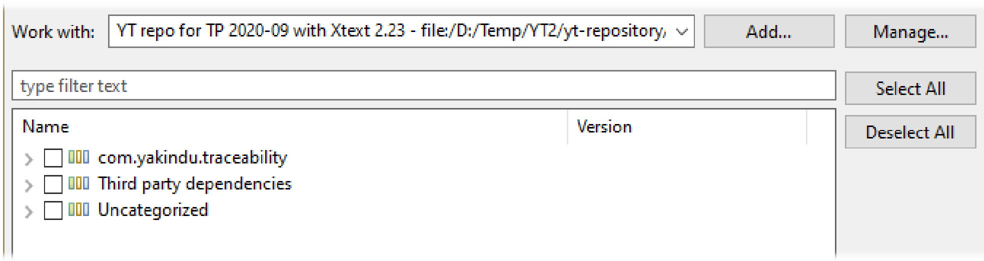


2. In the "Install" window, click **Add**.
3. In the "Add Repository" window, do the following:



   i. In the "Name" field, enter a meaningful name for the repository.
   ii. Click **Local**.
   iii. In the file selection window, select the folder where you stored the repository, then click **Select folder**.
   iv. Click **Add**.

   The repository name and file path appear in the "Work with" field of the "Install" window. The repository content is shown in the table below.



4. Expand the top and bottom nodes and select the features as shown in Figure 103.

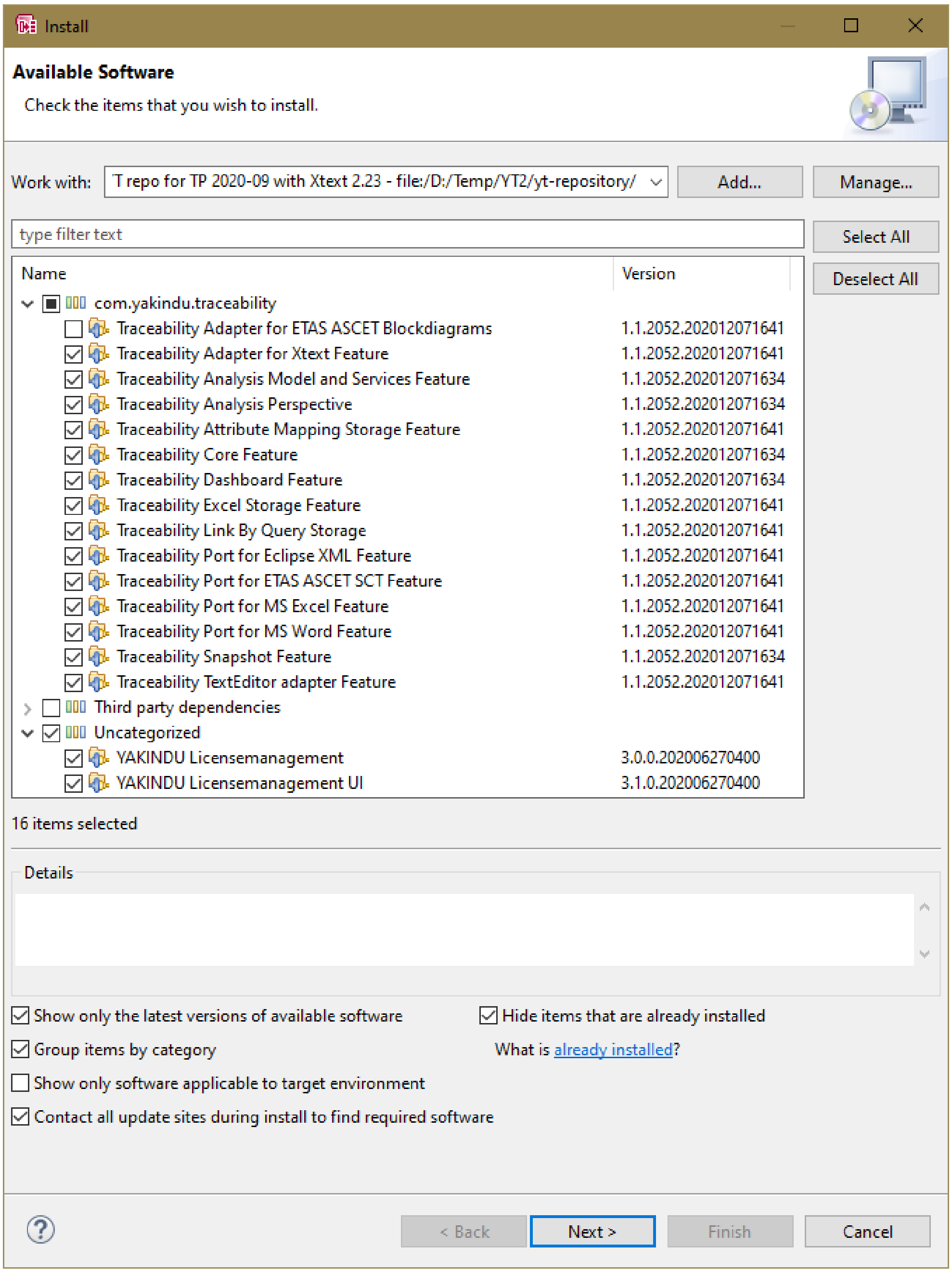


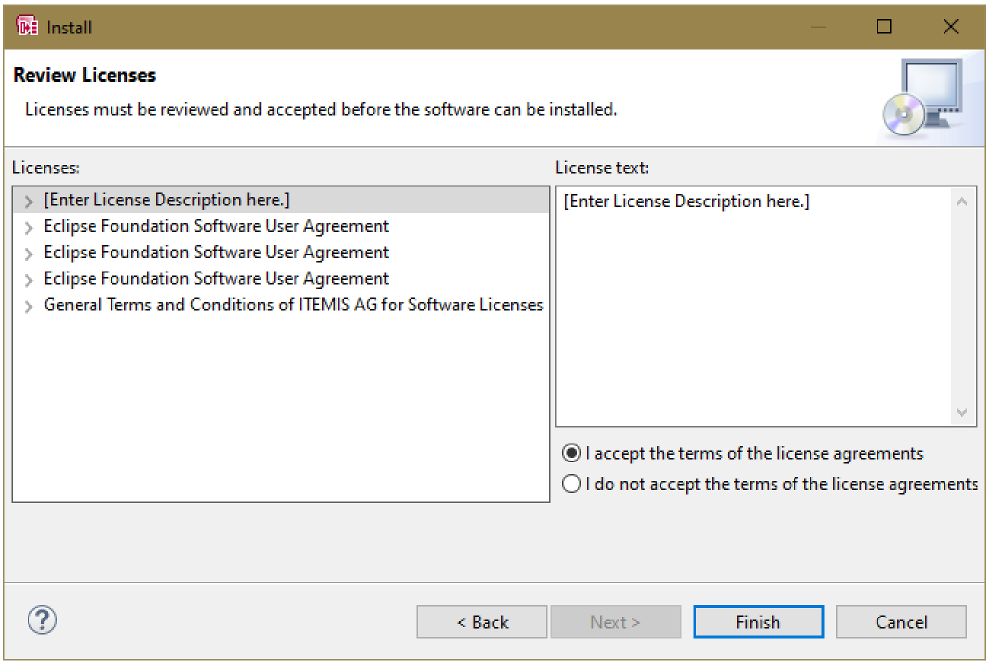
Figure 103. "Install" window with Yakindu Traceability features selected for installation

5. Click **Next** to continue.

   In the "Install" window, the "Install Details" page opens. It lists all components selected for installation.

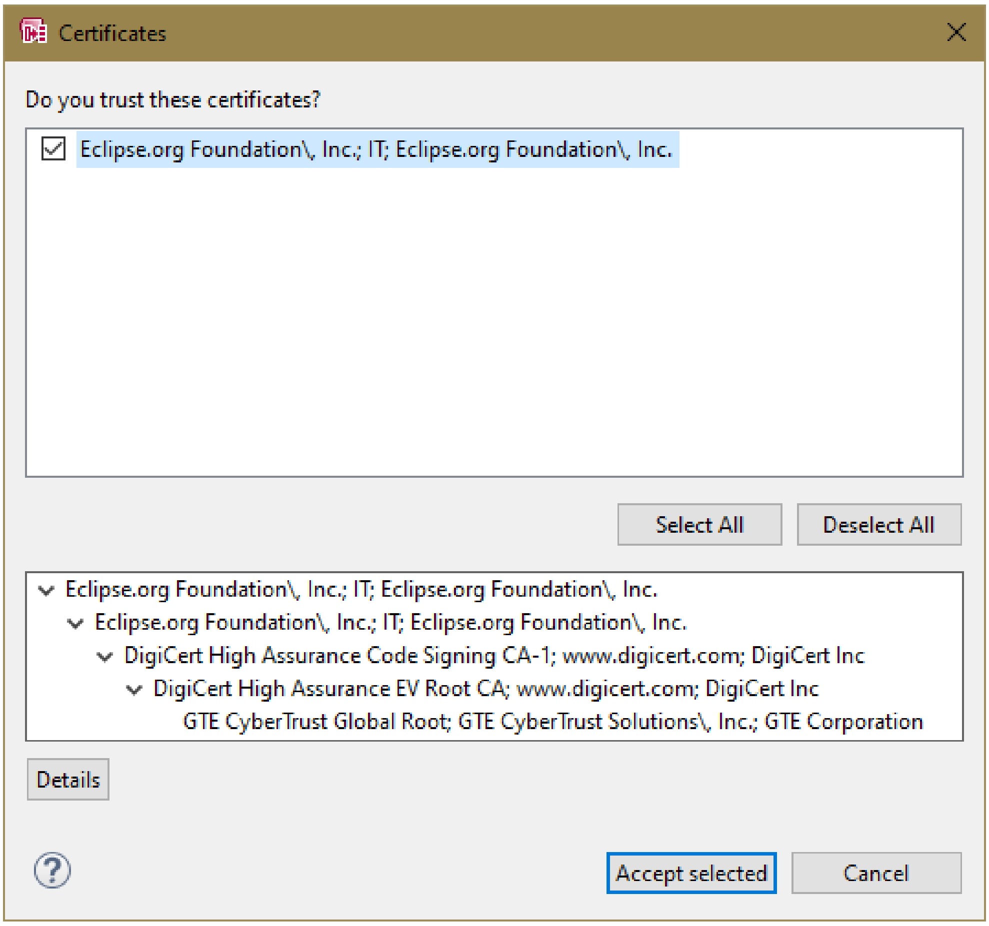
6. Click **Next** to continue.

   In the "Install" window, the "Review Licenses" page opens. It lists the liccense agreements for the selected components.



7. Read the license agreements, then activate **I accept the terms of the license agreements**.
8. Click **Finish** to start the installation.

   Installing Yakindu Traceability can take quite some time. During the process, the "Certificates" window opens.



9. In the "Certificates" window, select the certificate(s) you trust, then click **Accept selected**.

When the installation is complete, you are asked to restart the SCODE Workbench.

Do ***not*** restart the SCODE Workbench. Instead, click **Cancel** and exit the SCODE Workbench. After that, proceed as described in To update the SCODE Workbench.

To update the SCODE Workbench

NOTE	This procedure requires administrator rights.

1. Download the JDK needed to run Yakindu Traceability.

   The JDK is available at github.com/AdoptOpenJDK/openjdk11-binaries/releases/download/jdk-11.0.10+9/OpenJDK11U-jdk_x64_windows_hotspot_11.0.10_9.zip.

2. Unzip the JDK to a folder (e.g., `C:\Data\`**`jdk-11.0.10+9`**) on your computer.
3. In the Windows file system, navigate to your SCODE Workbench installation.
4. Replace the content of the **`jre`** folder with the JDK content you downloaded in the previous step (e.g., to `C:\Data\`**`jdk-11.0.10+9`**).

   To do so, you may perform the following two steps:

   i. In the SCODE Workbench installation directory, rename the existing `jre` folder (e.g., to `jre-SCODE Workbench`).

ii. Copy or move the unzipped JDK folder (e.g., `jdk-11.0.10+9`) to the SCODE Workbench installation directory and rename it to `jre`.

5. Start the SCODE Workbench and use it together with Yakindu Traceability.

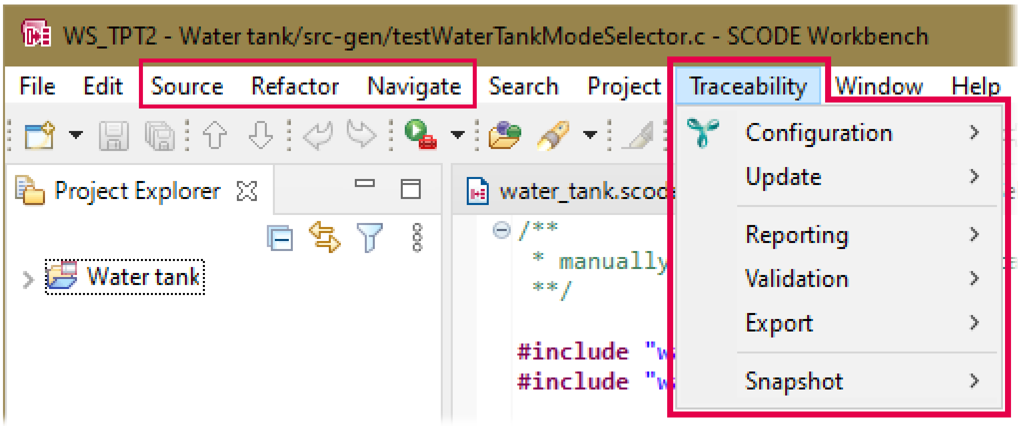


Figure 104. SCODE Workbench with menus added by Yakindu Traceability

[30] testing tool by Piketec GmbH

[31] In previous TPT versions, this button was named **View Assessment Results**.

8. Glossary

This chapter lists terms and abbreviations relevant for the SCODE Workbench, SCODE-ANALYZER (section 8.1), and SCODE-CONGRA (section 8.2).

MATLAB®

A multi-paradigm numerical computing environment and proprietary programming language developed by MathWorks®.

project

A project stores a model.

The SCODE Workbench offers two special project types, the SCODE-ANALYZER project and the SCODE-CONGRA project. Both project types are identified as such by the Eclipse environment. An open SCODE-ANALYZER project is marked by the icon. An open SCODE-CONGRA project is marked by the icon.

SCODE

System **CO**-**DE**sign

Simulink®

Tool for modeling, simulation and analysis of dynamic systems. Developed by MathWorks®.

8.1. SCODE-ANALYZER

action dimension

Action dimensions are adapted as an effect of a mode change.

alternative

An alternative is a certain state that a dimension can assume. In a real system, this is often an abstraction of a set or a range of real values.

condition dimension

A dimension that causes mode changes.

DAG

directed **a**cyclic **g**raph

dimension

Dimensions are aspects of the system or its context that cause or represent different system behaviors (or cause-effect chains). The dimensions may comprise, e.g., discrete states of the contexts, external requests to the system.

There are three types of dimensions: condition dimension, action dimension, and info dimension.

decision tree

A graphical visualization of the mode definition rules.

Essential Analysis

The SCODE Essential Analysis is based on the Essential Systems Analysis, developed by McMenamin and Palmer originally for IT systems, and extends and modifies it to enable application for physically dominated systems.

The successful application of the SCODE Essential Analysis yields a decomposition of the overall problem in several smaller subproblems which can be solved separately and more easily. The integration of the subproblem solutions then provides the overall solution of the original problem.

event

An event describes the conditions for the transition from one mode to another. An event is described by a set of rules that define its trigger conditions using the same rule definitions.

Only inclusion rules created from condition dimensions are used for the definition of events.

ICE

internal **c**ombustion **e**ngine

info dimension

Info dimensions are useful as information in analysis.

mode

A specific situation. In this situation, the system has to behave in a specific way, i.e., the system resides in the mode.

A mode is represented by a set of states in the problem space, i.e., in a Zwicky box by combinations of selected sets of alternatives for each dimension. Modes partition the system states of a Zwicky box into different sets of states using inclusion and exclusion rules.

non-system event

An event with impossible or meaningless rules, or with rules that are possible by nature, but ruled out by design.

non-system mode

A mode that stores impossible or meaningless combinations of conditions, and combinations that are possible by nature, but ruled out by design.

no transition

If a non-system event occurs, no transition between modes takes place.

overlapping

Two modes or two events overlap if at least one state is present in both modes or both events.

Overlapping modes or events make the system non-deterministic; they lead to errors.

rule

Rules define the conditions for the system states that belong to a mode or an event.

SOC

state **o**f **c**harge

source mode

The mode where a mode transition starts.

Not to be confused with the start mode of the system.

start mode

The mode the system enters first at the start of the execution.

Not to be confused with the source mode of a transition.

state

A state in SCODE-ANALYZER is characterized by selecting a specific alternative for each dimension. Each state is represented by a discrete set of alternatives.

The total number of states in a system, n_{total} is the product of the numbers of alternatives, na_i, of all n_{cond} conditions.

$$n_{total} = \prod_{i=1}^{i=n_{cond}} na_i$$

One or more states can be grouped into a mode.

system mode

A mode that is relevant for the problem solution and models the corresponding system.

target mode

The mode where a transition ends.

TPT

Time **P**artition **T**esting tool by Piketec GmbH

Zwicky box

A Zwicky box is a grid box (table) to support morphological analysis for multi-dimensional, non-quantifiable problems.

The Zwicky box is named after the developer of this method, Fritz Zwicky (February 14, 1898 — February 8, 1974), a Swiss astronomer.

8.2. SCODE-CONGRA

computation

A computation is the result of solving a flow, an executable sequence of computation steps. It captures the solved equations, and also orders the computation steps in a linear way, via levels.

ESDL

Embedded **S**oftware **D**evelopment **L**anguage; a high-level programming language for writing real-time, deeply embedded software.

flow

A flow defines a computation order in a system. A system itself can have any numbers of flows attached to it.

A flow is associated to a system and defines which variables are considered as input, output or constant to the specific system.

If a flow is valid, the equations in the system become directed to produce the imposed outputs of the relations.

For example, if m and c are given, then E is computed as follows: $E = m*c^2$

If E and c are given, then m is computed as follows: $m = E / c^2$

A valid flow is the basis for code generation.

level

Used to order the steps in a computation.

In the computation SYQ file, the levels are represented by the `@level(i,j)` annotation. In the computation graph, the levels are shown as red numbers.

Maxima

An open-source third-party computer algebra system, which is available on your computer with SCODE-CONGRA.

MuPAD®

Used by the Symbolic Math Toolbox™ as part of its underlying computational engine. Can be used as solver in SCODE-CONGRA.

relation

A relation describes how different variables of a system are interrelated. It does not imply a computation direction. The relations between different variables are specified by mathematical equations, e.g., Einstein's famous relation: $E - m*c^2 = 0$

Symbolic Math Toolbox™

Provides functions for solving, plotting, and manipulating symbolic math equations. The MuPAD solver that can be used in SCODE-CONGRA is included in this toolbox.

SYQ

System Equation Language

SYQ file

A textual file in SYQ that contains the semantic description of the system.

A SYQ file is the textual base of each SCODE-CONGRA project. Here, all variables, relations, units, and flows are defined or stored (when you are working in the graphical editor).

Each SCODE-CONGRA project must have at least one SYQ file.

system

A system is defined as a set of variables and relations between the variables. A system is undirected, i.e. no inputs and outputs are specified. You cannot generate executable code from an undirected system.

System Equation Language

A language developed by ETAS to describe a continuous system in SCODE-CONGRA.

variable

A variable is an element that can be read and written during the execution of a SCODE-CONGRA model.

In SCODE-CONGRA, all variables are deemed to be continuous.

9. Tutorial Hints

This chapter contains reference information for SCODE-ANALYZER (section 9.1) and SCODE-CONGRA (section 9.2).

9.1. SCODE-ANALYZER Tutorial Hints

9.1.1. Problem Space

Dimension	Alternatives
battery SOC [32]	full / empty / normal
battery at OT [33]	yes / no
electric engine cable	okay / defective
silent mode [34]	on / off
desired acceleration	increase speed / decrease speed / keep speed
fuel tank	empty / not empty
car moves	no / yes

Table 24. Problem space — suggestions (see section 4.3)

9.1.2. Modes

Mode	Dimensions					
	battery SOC	battery at OT	electric engine cable	silent mode	fuel tank	car moves
charging	empty or normal	yes	okay			yes
discharging	NOT empty	yes	okay			
standstill		yes	okay			no
		no			empty	
			defective		empty	
combustion engine only			defective		not empty	
		no			not empty	
	empty	yes	okay	off	not empty	
mechanical brake	full	yes	okay			yes

Table 25. Modes and rules — first set of suggestions (see section 4.4.1)

Mode	Dimensions						
	battery SOC	**battery at OT**	**electric engine cable**	**silent mode**	**fuel tank**	**car moves**	**desired acceleration**
charging	empty or normal	yes	okay			yes	decrease speed
discharging	NOT empty	yes	okay				NOT decrease speed
standstill		yes	okay			no	decrease speed
		no			empty		
			defective		empty		
combustion engine only			defective		not empty		
		no			not empty		
	empty	yes	okay	off	not empty		NOT decrease speed
mechanical brake	full	yes	okay			yes	decrease speed

Table 26. Modes and rules — suggestions for additional condition (see section 4.4.2)

Dimensions	Suggestion 1	Suggestion 2
battery SOC	empty	empty
battery at OT	yes	yes
electric engine cable	okay	okay
silent mode	on	
fuel tank		empty
car moves		
desired acceleration	NOT decrease speed	NOT decrease speed

Table 27. Suggested rules for the missing states. Alternatives that cannot be true at the same time are marked.

Dimensions	Suggestion
battery SOC	empty
battery at OT	yes
electric engine cable	okay
silent mode	off
fuel tank	empty
car moves	
desired acceleration	NOT decrease speed

Table 28. Suggested rules for the states that are still missing after suggestion 1 from the previous table has been inserted as non-system mode

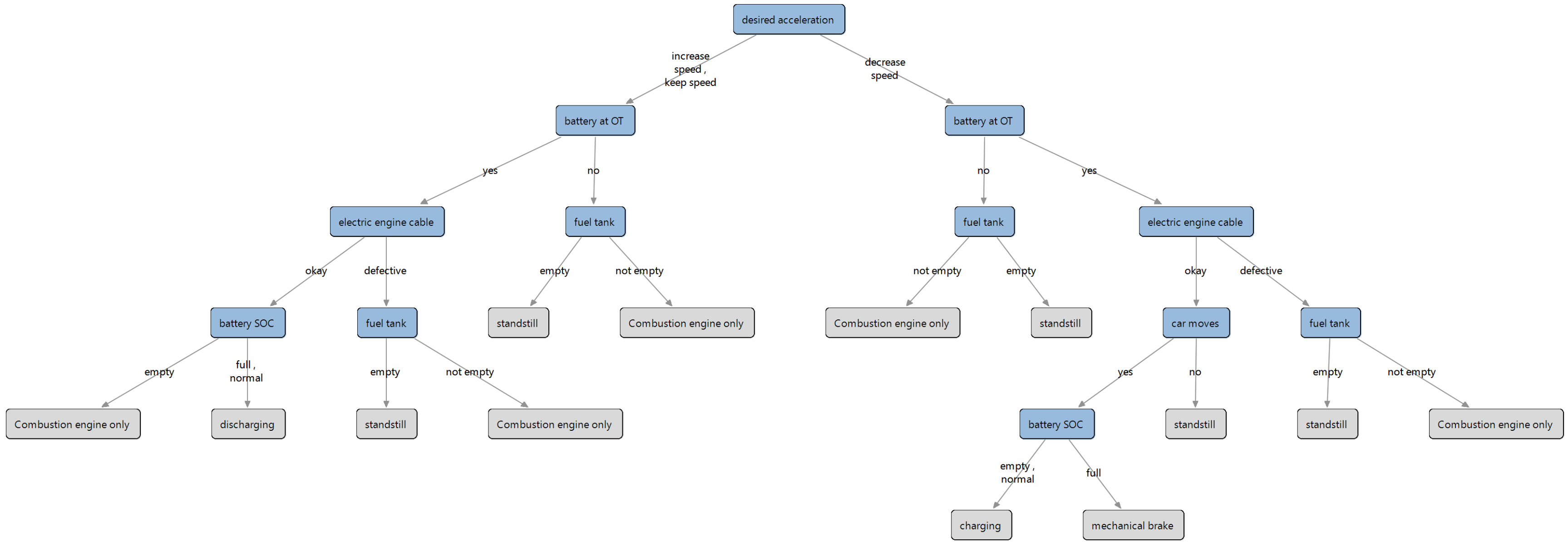


Figure 105. Complete decision tree for the hybrid car example; with condition **desired acceleration** as root (see section 4.4.4)

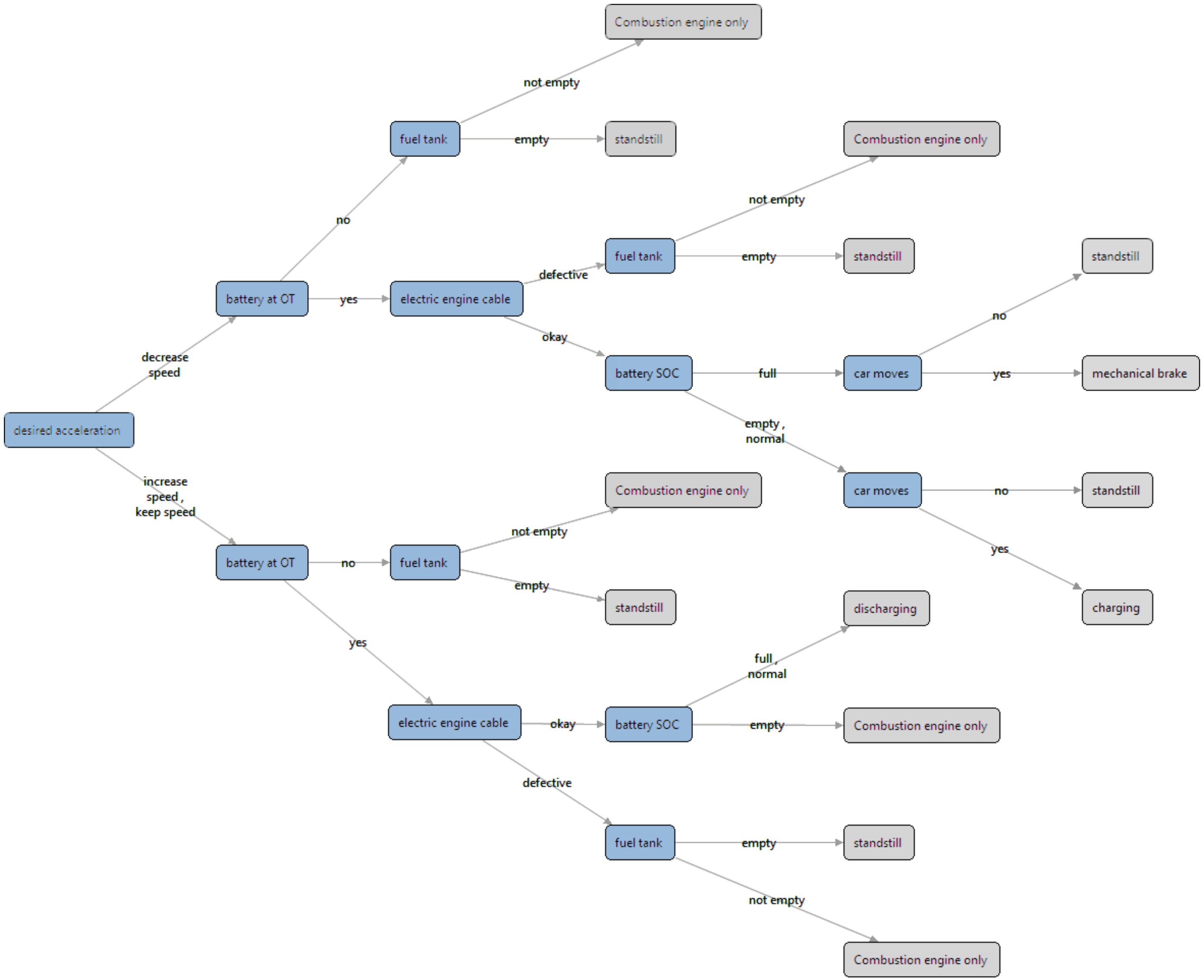


Figure 106. The decision tree from Figure 105 with horizontal orientation (see section 4.4.4)

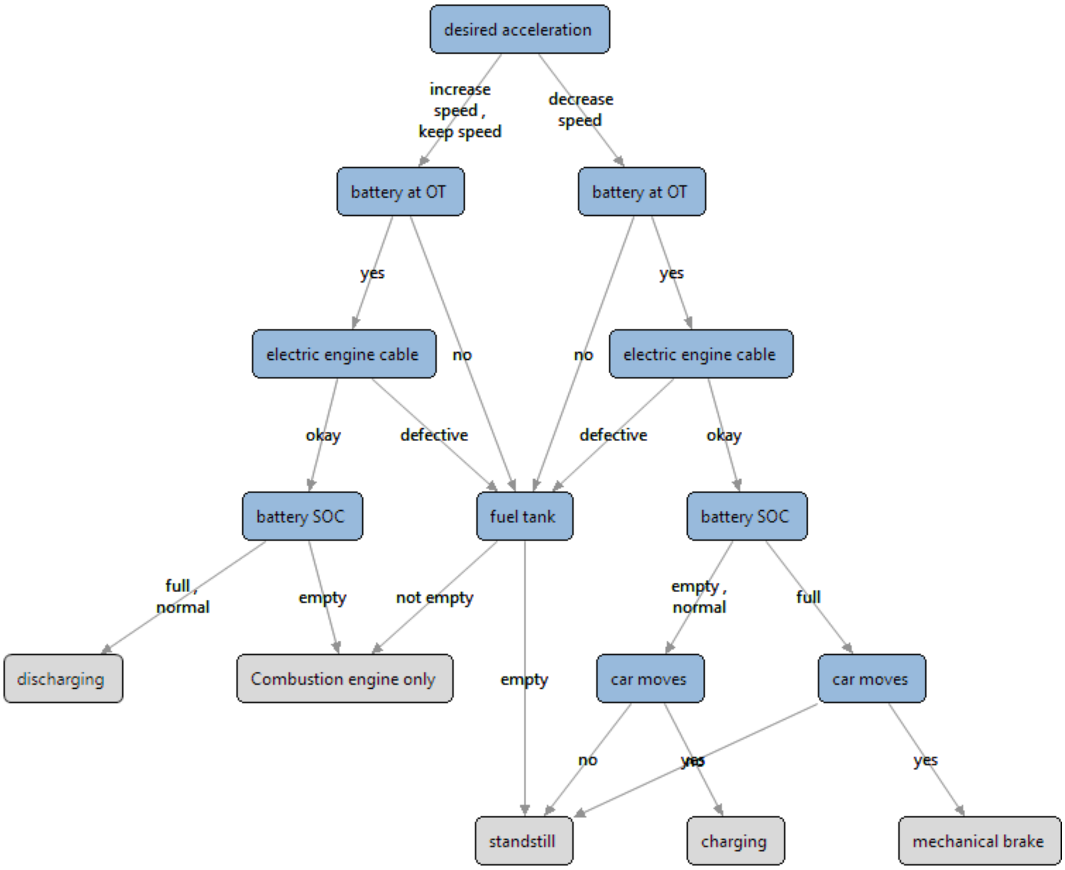


Figure 107. DAG view of the decision tree with vertical orientation (see section 4.4.4)

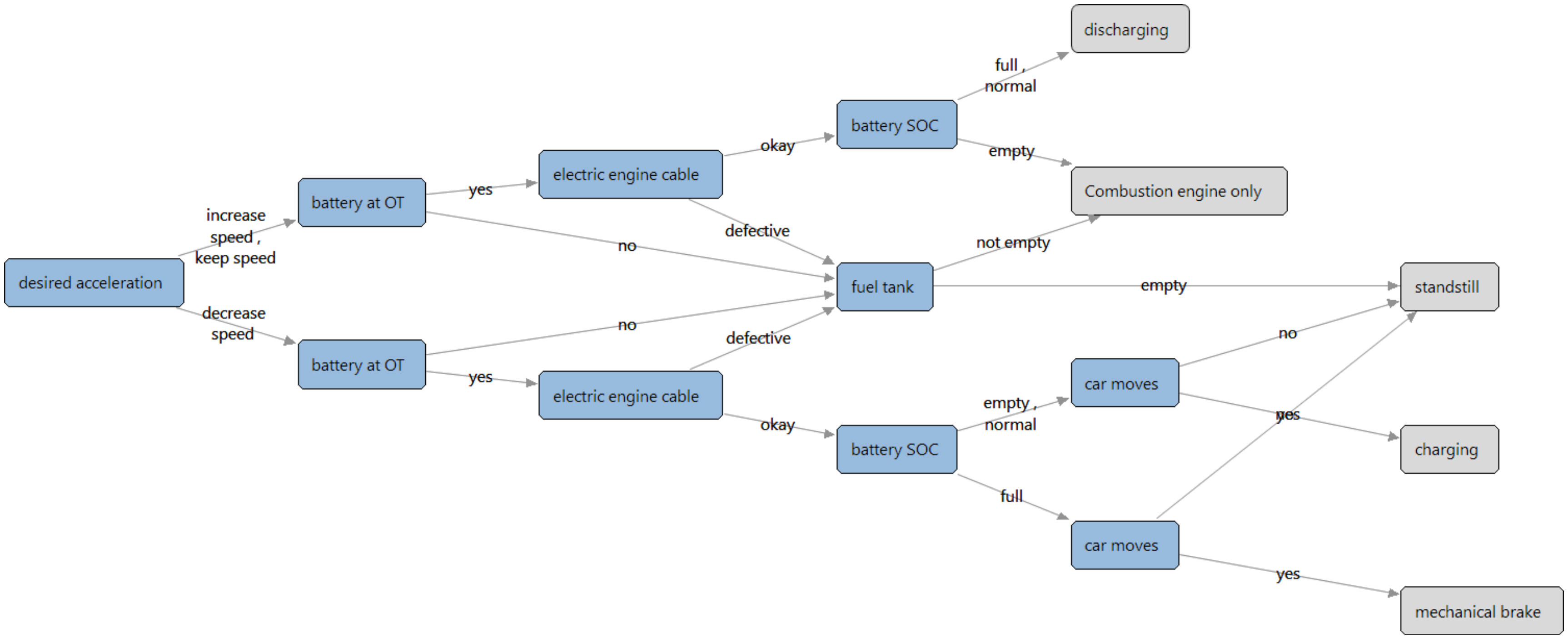


Figure 108. DAG view of the decision tree with horizontal orientation (see section 4.4.4)

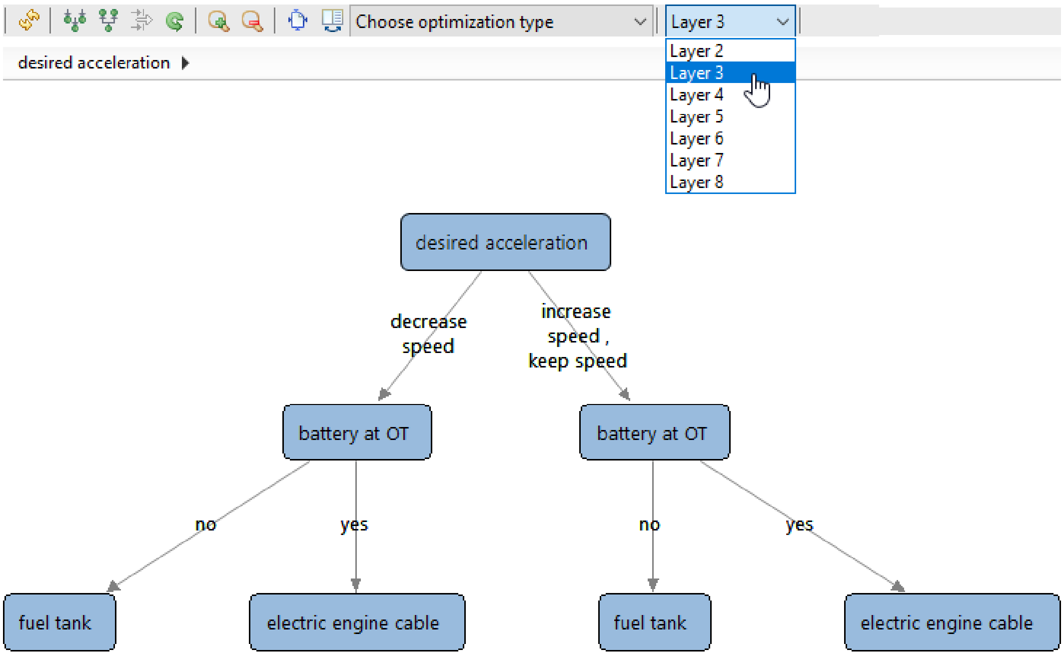


Figure 109. Decision tree with selected layers, first three levels are shown (see section 4.4.4)

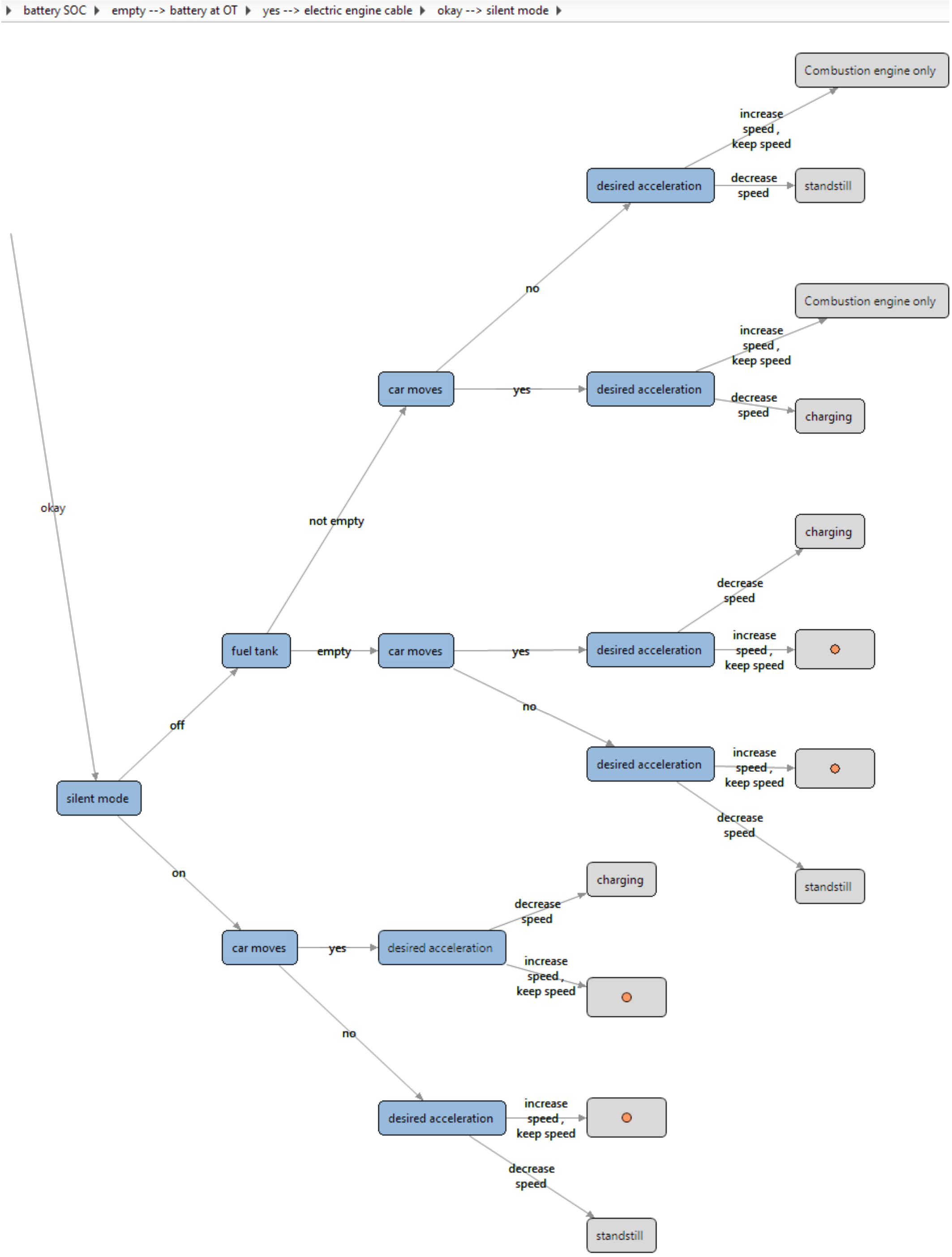


Figure 110. Sub-tree (horizontal orientation) with non-system modes displayed (see section 4.4.4)

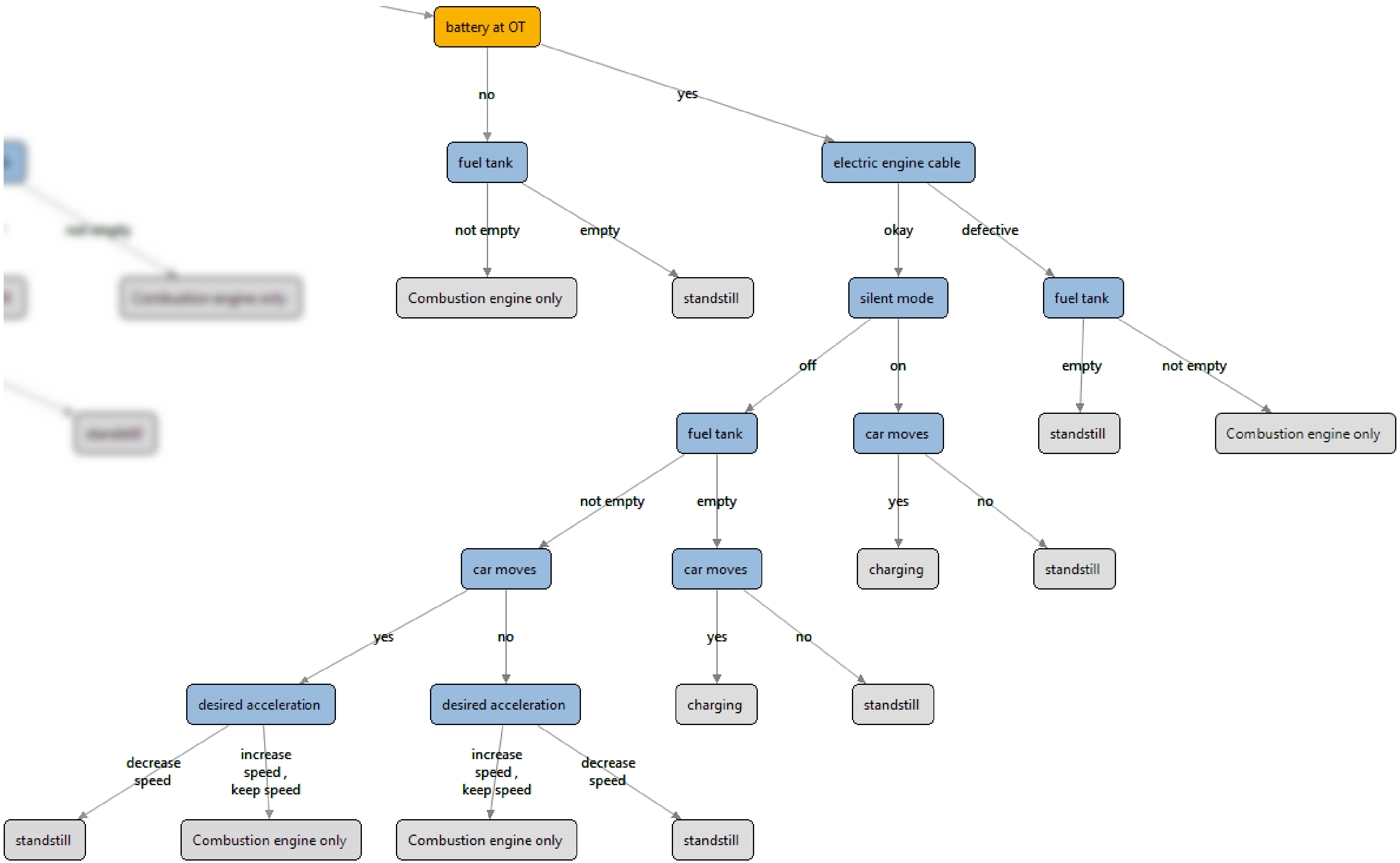


Figure 111. Sub-tree before height optimization (see section 4.4.4)

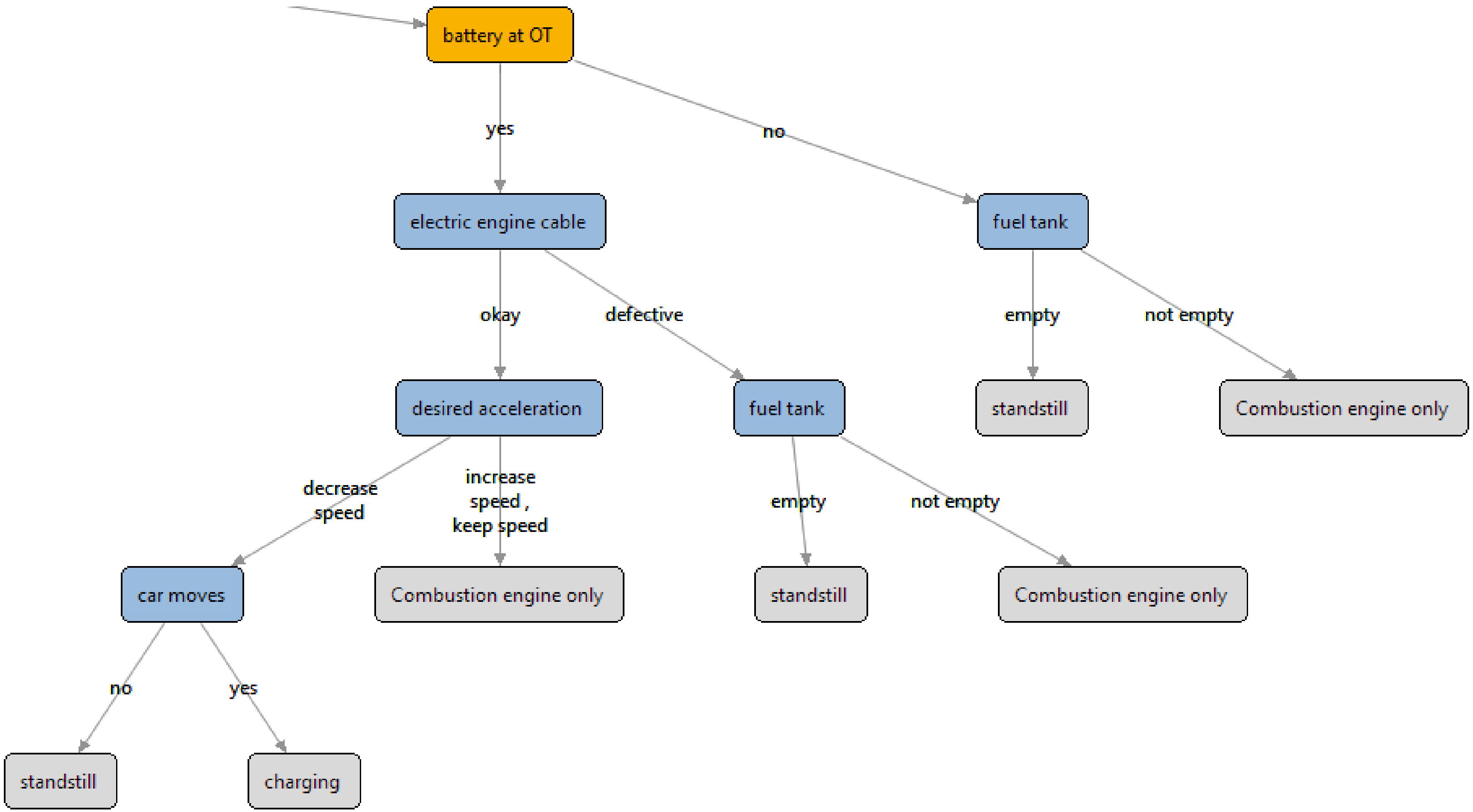


Figure 112. Sub-tree after height optimization (see section 4.4.4)

9.1.3. Events and Transitions

current mode	next mode					
	charging	standstill	mechanical brake	discharging	combustion engine only	non-system
charging	*	E2	E4	E1	E3	
standstill	E5	*	E6	E7	E8	
mechanical brake	E9	E10	*	E11	E12	
discharging	E13	E14	E15	*	E16	
combustion engine only	E17	E18	E19	E20	*	

Table 29. Transitions with associated events (*: no transition; --: forbidden transition) for section 4.6

event		rule(s)
#	name	
E1	`charging_ discharging`	• battery SOC = full AND battery at OT = yes AND electric engine cable = okay AND car moves = yes AND desired acceleration = (increase speed OR keep speed) • battery SOC = full AND battery at OT = yes AND electric engine cable = okay AND car moves = no AND desired acceleration = (increase speed OR keep speed) • battery SOC = normal AND battery at OT = yes AND electric engine cable = okay AND desired acceleration = NOT(decrease speed)
E2	`charging_ standstill`	• electric engine cable = defective AND fuel tank = empty • battery at OT = no AND fuel tank = empty • battery at OT = yes AND electric engine cable = okay AND car moves = no AND desired acceleration = decrease speed
E4	`charging_ mechanBrake`	• battery SOC = full AND battery at OT = yes and electric engine cable = okay AND car moves = yes AND desired acceleration = decrease speed
E3	`charging_ combustionOnly`	• battery SOC = empty AND silent mode = off AND fuel tank = not empty AND desired acceleration = (increase speed OR keep speed) • electric engine cable = defective AND fuel tank = not empty • battery at OT = no AND electric engine cable = okay AND fuel tank = not empty AND desired acceleration = decrease speed • battery at OT = no AND electric engine cable = okay AND silent mode = on AND fuel tank = not empty AND desired acceleration = (increase speed OR keep speed) • battery SOC = (full OR normal) AND battery at OT = no AND electric engine cable = okay AND silent mode = off AND fuel tank = not empty AND desired acceleration = (increase speed OR keep speed)

Table 30. Events and rules for the transitions from mode `charging`

event		rule(s)
#	name	
E5	`standstill_charging`	• battery SOC = (empty OR normal) AND battery at OT = yes AND electric engine cable = okay AND car moves = yes AND desired acceleration = decrease speed
E6	`standstill_mechanBrake`	• battery SOC = full AND battery at OT = yes AND electric engine cable = okay AND car moves = yes AND desired acceleration = decrease speed
E7	`standstill_discharging`	• battery SOC = (full or normal) AND battery at OT = yes AND electric engine cable = okay AND desired acceleration = (increase speed OR keep speed)
E8	`standstill_combustionOnly`	• battery SOC = empty AND silent mode = off AND fuel tank = not empty AND desired acceleration = (increase speed OR keep speed) • electric engine cable = defective AND fuel tank = not empty AND desired acceleration = decrease speed • battery SOC = (full or normal) AND battery at OT = no AND fuel tank = not empty AND desired acceleration = (increase speed OR keep speed) • battery SOC = empty AND battery at OT = no AND silent mode = on AND fuel tank = not empty AND desired acceleration = (increase speed OR keep speed) • battery at OT = yes AND electric engine cable = defective AND silent mode = on AND fuel tank = not empty AND desired acceleration = (increase speed OR keep speed) • battery SOC = (full or normal) AND battery at OT = yes AND electric engine cable = defective AND silent mode = off AND fuel tank = not empty AND desired acceleration = (increase speed OR keep speed) • battery at OT = no AND electric engine cable = okay AND fuel tank = not empty AND desired acceleration = decrease speed

Table 31. Events and rules for the transitions from mode `standstill`

event		rule(s)
#	name	
E9	`mechanBrake_charging`	• battery SOC = (empty or normal) AND battery at OT = yes AND electric engine cable = okay AND car moves = yes AND desired acceleration = decrease speed
E10	`mechanBrake_standstill`	• battery at OT = yes AND electric engine cable = okay AND car moves = no AND desired acceleration = decrease speed • electric engine cable = defective AND fuel tank = empty • battery at OT = no AND electric engine cable = defective AND fuel tank = empty • battery at OT = no AND electric engine cable = okay AND fuel tank = empty
E11	`mechanBrake_discharging`	• battery SOC = (full or normal) AND battery at OT = yes AND electric engine cable = okay AND desired acceleration = (increase speed OR keep speed)
E12	`mechanBrake_combustionOnly`	• electric engine cable = defective AND fuel tank = not empty AND desired acceleration = decrease speed • battery SOC = (full or normal) AND battery at OT = no AND fuel tank = not empty AND desired acceleration = (increase speed OR keep speed) • battery SOC = empty AND battery at OT = no AND silent mode = on AND fuel tank = not empty AND desired acceleration = (increase speed OR keep speed) • battery at OT = yes AND electric engine cable = defective AND silent mode = on AND fuel tank = not empty AND desired acceleration = (increase speed OR keep speed) • battery SOC = (full or normal) AND battery at OT = yes AND electric engine cable = defective AND silent mode = off AND fuel tank = not empty AND desired acceleration = (increase speed OR keep speed) • battery at OT = no AND electric engine cable = okay AND fuel tank = not empty AND desired acceleration = decrease speed • battery SOC = empty AND silent mode = off AND fuel tank = not empty AND desired acceleration = (increase speed OR keep speed)

Table 32. Events and rules for the transitions from mode `mechanical brake`

event		rule(s)
#	name	
E13	discharging_ charging	• battery SOC = (empty or normal) AND battery at OT = yes AND electric engine cable = okay AND car moves = yes AND desired acceleration = decrease speed
E14	discharging_ standstill	• battery at OT = yes AND electric engine cable = okay AND car moves = no AND desired acceleration = decrease speed • electric engine cable = defective AND fuel tank = empty • battery at OT = no AND electric engine cable = okay AND fuel tank = empty
E15	discharging_ mechanBrake	• battery SOC = full AND battery at OT = yes AND electric engine cable = okay AND car moves = yes AND desired acceleration = decrease speed
E16	discharging_ combustionOnly	• battery SOC = empty AND silent mode= off AND fuel tank = not empty AND desired acceleration = (increase speed OR keep speed) • battery SOC = (full or normal) AND battery at OT = no AND fuel tank = not empty • battery at OT = yes AND electric engine cable = defective AND silent mode = on AND fuel tank = not empty • battery at OT = yes AND electric engine cable = defective AND silent mode = off AND fuel tank = not empty AND desired acceleration = decrease speed • battery SOC = (full or normal) AND battery at OT = yes AND electric engine cable = defective AND silent mode = off AND fuel tank = not empty AND desired acceleration = (increase speed OR keep speed) • battery SOC = empty AND battery at OT = no AND fuel tank = not empty AND desired acceleration = decrease speed • battery SOC = empty AND battery at OT = no AND silent mode = on AND fuel tank = not empty AND desired acceleration = (increase speed OR keep speed)

Table 33. Events and rules for the transitions from mode `discharging`

event		rule(s)
#	name	
E17	combustionOnly_charging	• battery SOC = (empty or normal) AND battery at OT = yes AND electric engine cable = okay AND car moves = yes AND desired acceleration = decrease speed
E18	combustionOnly_standstill	• electric engine cable = defective AND fuel tank = empty • battery at OT = yes AND electric engine cable = okay AND car moves = no AND desired acceleration = decrease speed • battery at OT = no AND electric engine cable = okay AND fuel tank = empty
E19	combustionOnly_mechanBrake	• battery SOC = full AND battery at OT = yes AND electric engine cable = okay AND car moves = yes AND desired acceleration = decrease speed
E20	combustionOnly_discharging	• battery SOC = (full or normal) AND battery at OT = yes AND electric engine cable = okay AND desired acceleration = (increase speed OR keep speed)

Table 34. Events and rules for the transitions from mode `Combustion engine only`

9.1.4. Code Generation: Mode Invariants

```
 /**
  * @warning       AUTOMATICALLY GENERATED FILE! DO NOT EDIT!
...
  **/


package hybridCar;


import hybridCar.Mode_Type;
import hybridCar.battery_SOC_Type;
import hybridCar.battery_at_OT_Type;
import hybridCar.electric_engine_cable_Type;
import hybridCar.silent_mode_Type;
import hybridCar.fuel_tank_Type;
import hybridCar.car_moves_Type;
import hybridCar.desired_acceleration_Type;


class hybridCar {


public Mode_Type hybridCar_ModeSelector(battery_SOC_Type battery_SOC, battery_at_OT_Type battery_at_OT,
     electric_engine_cable_Type electric_engine_cable, silent_mode_Type silent_mode, fuel_tank_Type
     fuel_tank, car_moves_Type car_moves, desired_acceleration_Type desired_acceleration) {


Mode_Type mode = Mode_Type.charging;


if ((!(battery_SOC == battery_SOC_Type.full) && battery_at_OT == battery_at_OT_Type.yes
     && electric_engine_cable == electric_engine_cable_Type.okay && car_moves == car_moves_Type.yes
     && desired_acceleration == desired_acceleration_Type.decrease_speed)) {
    mode = Mode_Type.charging;
```

```
} else if ((battery_at_OT == battery_at_OT_Type.yes
     && electric_engine_cable == electric_engine_cable_Type.okay && car_moves == car_moves_Type.no
     && desired_acceleration == desired_acceleration_Type.decrease_speed) ||
     (battery_at_OT == battery_at_OT_Type.no && fuel_tank == fuel_tank_Type.empty) ||
     (electric_engine_cable == electric_engine_cable_Type.defective
      && fuel_tank == fuel_tank_Type.empty)) {
    mode = Mode_Type.standstill;

} else if ((battery_SOC == battery_SOC_Type.full && battery_at_OT == battery_at_OT_Type.yes &&
     electric_engine_cable == electric_engine_cable_Type.okay && car_moves == car_moves_Type.yes &&
     desired_acceleration == desired_acceleration_Type.decrease_speed)) {
    mode = Mode_Type.mechanical_brake;

} else if ((!(battery_SOC == battery_SOC_Type.empty) && battery_at_OT == battery_at_OT_Type.yes &&
     electric_engine_cable == electric_engine_cable_Type.okay && !(desired_acceleration ==
     desired_acceleration_Type.decrease_speed))) {
    mode = Mode_Type.discharging;

} else if ((electric_engine_cable == electric_engine_cable_Type.defective && fuel_tank ==
     fuel_tank_Type.not_empty) || (battery_at_OT == battery_at_OT_Type.no && fuel_tank ==
     fuel_tank_Type.not_empty) || (battery_SOC == battery_SOC_Type.empty && battery_at_OT ==
     battery_at_OT_Type.yes && electric_engine_cable == electric_engine_cable_Type.okay &&
     silent_mode == silent_mode_Type.off && fuel_tank == fuel_tank_Type.not_empty &&
     !(desired_acceleration == desired_acceleration_Type.decrease_speed))) {
    mode = Mode_Type.combustion_engine_only;

} else {
    mode = Mode_Type.charging;

        }
        return mode;
    }   // hybridCar_ModeSelector
}   // hybridCar
```

9.1.5. Code Generation: Transition Matrix

```
/**
 * @warning       AUTOMATICALLY GENERATED FILE! DO NOT EDIT!
...
 **/

package hybridCar;

import hybridCar.Mode_Type;
import hybridCar.battery_SOC_Type;
import hybridCar.battery_at_OT_Type;
import hybridCar.electric_engine_cable_Type;
import hybridCar.silent_mode_Type;
import hybridCar.fuel_tank_Type;
import hybridCar.car_moves_Type;
import hybridCar.desired_acceleration_Type;

class hybridCar {

public Mode_Type hybridCar_ModeSelector(Mode_Type currentMode, battery_SOC_Type battery_SOC,
     battery_at_OT_Type battery_at_OT, electric_engine_cable_Type electric_engine_cable,
     silent_mode_Type silent_mode, fuel_tank_Type fuel_tank, car_moves_Type car_moves,
     desired_acceleration_Type desired_acceleration) {

Mode_Type mode = Mode_Type.charging;
switch (currentMode) {
    case Mode_Type.charging : {
        if ((electric_engine_cable == electric_engine_cable_Type.defective && fuel_tank ==
             fuel_tank_Type.empty) || (battery_at_OT == battery_at_OT_Type.no &&
             fuel_tank == fuel_tank_Type.empty) || (battery_at_OT == battery_at_OT_Type.yes
             && electric_engine_cable == electric_engine_cable_Type.okay && car_moves ==
             car_moves_Type.no && desired_acceleration ==
             desired_acceleration_Type.decrease_speed)) {
            mode = Mode_Type.standstill;
```

```
} else if ((battery_SOC == battery_SOC_Type.full && battery_at_OT == battery_at_OT_Type.yes
     && electric_engine_cable == electric_engine_cable_Type.okay && car_moves ==
     car_moves_Type.yes && desired_acceleration ==
     desired_acceleration_Type.decrease_speed)) {
    mode = Mode_Type.mechanical_brake;

} else if (((battery_SOC == battery_SOC_Type.full || battery_SOC ==
     battery_SOC_Type.normal) && battery_at_OT == battery_at_OT_Type.yes &&
     electric_engine_cable == electric_engine_cable_Type.okay && (desired_acceleration ==
     desired_acceleration_Type.keep_speed || desired_acceleration ==
     desired_acceleration_Type.increase_speed))) {
    mode = Mode_Type.discharging;

} else if ((battery_SOC == battery_SOC_Type.empty && silent_mode == silent_mode_Type.off &&
     fuel_tank == fuel_tank_Type.not_empty && (desired_acceleration ==
     desired_acceleration_Type.keep_speed || desired_acceleration ==
     desired_acceleration_Type.increase_speed)) || (electric_engine_cable ==
     electric_engine_cable_Type.defective && fuel_tank == fuel_tank_Type.not_empty) ||
     (battery_at_OT == battery_at_OT_Type.no && fuel_tank == fuel_tank_Type.not_empty)) {
    mode = Mode_Type.combustion_engine_only;

    } else {
        mode = Mode_Type.charging;
    }
}   // Mode_Type.charging

case Mode_Type.standstill : {
    if (((battery_SOC == battery_SOC_Type.empty || battery_SOC == battery_SOC_Type.normal) &&
         battery_at_OT == battery_at_OT_Type.yes && electric_engine_cable ==
         electric_engine_cable_Type.okay && car_moves == car_moves_Type.yes &&
         desired_acceleration == desired_acceleration_Type.decrease_speed)) {
        mode = Mode_Type.charging;
```

```
} else if ((battery_SOC == battery_SOC_Type.full && battery_at_OT == battery_at_OT_Type.yes
     && electric_engine_cable == electric_engine_cable_Type.okay && car_moves ==
     car_moves_Type.yes && desired_acceleration ==
     desired_acceleration_Type.decrease_speed)) {
    mode = Mode_Type.mechanical_brake;

           } else if (((battery_SOC == battery_SOC_Type.full || battery_SOC ==
                battery_SOC_Type.normal) && battery_at_OT == battery_at_OT_Type.yes &&
    electric_engine_cable == electric_engine_cable_Type.okay && (desired_acceleration ==
 desired_acceleration_Type.keep_speed || desired_acceleration ==
 desired_acceleration_Type.increase_speed))) {
mode = Mode_Type.discharging;

} else if ((battery_SOC == battery_SOC_Type.empty && silent_mode == silent_mode_Type.off &&
     fuel_tank == fuel_tank_Type.not_empty && (desired_acceleration ==
     desired_acceleration_Type.keep_speed || desired_acceleration ==
     desired_acceleration_Type.increase_speed)) || (battery_at_OT == battery_at_OT_Type.no
     && fuel_tank == fuel_tank_Type.not_empty) || (electric_engine_cable ==
  electric_engine_cable_Type.defective && fuel_tank == fuel_tank_Type.not_empty)) {
 mode = Mode_Type.combustion_engine_only;

 } else {
     mode = Mode_Type.standstill;
 }
}    // Mode_Type.standstill

case Mode_Type.mechanical_brake : {
 if (((battery_SOC == battery_SOC_Type.empty || battery_SOC == battery_SOC_Type.normal) &&
      battery_at_OT == battery_at_OT_Type.yes && electric_engine_cable ==
      electric_engine_cable_Type.okay && car_moves == car_moves_Type.yes &&
      desired_acceleration == desired_acceleration_Type.decrease_speed)) {
     mode = Mode_Type.charging;
```

```
} else if ((battery_at_OT == battery_at_OT_Type.yes && electric_engine_cable ==
  electric_engine_cable_Type.okay && car_moves == car_moves_Type.no &&
  desired_acceleration == desired_acceleration_Type.decrease_speed) ||
  (electric_engine_cable == electric_engine_cable_Type.defective && fuel_tank ==
  fuel_tank_Type.empty) || (battery_at_OT == battery_at_OT_Type.no &&
  fuel_tank == fuel_tank_Type.empty)) {
 mode = Mode_Type.standstill;

} else if (((battery_SOC == battery_SOC_Type.full || battery_SOC ==
     battery_SOC_Type.normal) && battery_at_OT == battery_at_OT_Type.yes &&
     electric_engine_cable == electric_engine_cable_Type.okay && (desired_acceleration ==
     desired_acceleration_Type.keep_speed || desired_acceleration ==
  desired_acceleration_Type.increase_speed))) {
 mode = Mode_Type.discharging;

} else if ((battery_SOC == battery_SOC_Type.empty && silent_mode == silent_mode_Type.off &&
     fuel_tank == fuel_tank_Type.not_empty && (desired_acceleration ==
     desired_acceleration_Type.keep_speed || desired_acceleration ==
     desired_acceleration_Type.increase_speed)) || (battery_at_OT == battery_at_OT_Type.no
     && fuel_tank == fuel_tank_Type.not_empty) || (electric_engine_cable ==
  electric_engine_cable_Type.defective && fuel_tank == fuel_tank_Type.not_empty)) {
 mode = Mode_Type.combustion_engine_only;

 } else {
     mode = Mode_Type.mechanical_brake;
 }
}    // Mode_Type.mechanical_brake

case Mode_Type.discharging : {
 if (((battery_SOC == battery_SOC_Type.empty || battery_SOC == battery_SOC_Type.normal) &&
      battery_at_OT == battery_at_OT_Type.yes && electric_engine_cable ==
      electric_engine_cable_Type.okay && car_moves == car_moves_Type.yes &&
      desired_acceleration == desired_acceleration_Type.decrease_speed)) {
     mode = Mode_Type.charging;
```

```
} else if ((battery_at_OT == battery_at_OT_Type.yes && electric_engine_cable ==
  electric_engine_cable_Type.okay && car_moves == car_moves_Type.no &&
  desired_acceleration == desired_acceleration_Type.decrease_speed) ||
  (electric_engine_cable == electric_engine_cable_Type.defective && fuel_tank ==
  fuel_tank_Type.empty) || (battery_at_OT == battery_at_OT_Type.no &&
  fuel_tank == fuel_tank_Type.empty)) {
 mode = Mode_Type.standstill;

} else if ((battery_SOC == battery_SOC_Type.full && battery_at_OT == battery_at_OT_Type.yes
     && electric_engine_cable == electric_engine_cable_Type.okay && car_moves ==
     car_moves_Type.yes && desired_acceleration ==
     desired_acceleration_Type.decrease_speed)) {
 mode = Mode_Type.mechanical_brake;

} else if ((battery_SOC == battery_SOC_Type.empty && silent_mode == silent_mode_Type.off &&
     fuel_tank == fuel_tank_Type.not_empty && (desired_acceleration ==
     desired_acceleration_Type.keep_speed || desired_acceleration ==
     desired_acceleration_Type.increase_speed)) || (electric_engine_cable ==
  electric_engine_cable_Type.defective && fuel_tank == fuel_tank_Type.not_empty) ||
  (battery_at_OT == battery_at_OT_Type.no && fuel_tank == fuel_tank_Type.not_empty)) {
 mode = Mode_Type.combustion_engine_only;

 } else {
     mode = Mode_Type.discharging;
 }
}    // Mode_Type.discharging

case Mode_Type.combustion_engine_only : {
 if (((battery_SOC == battery_SOC_Type.empty || battery_SOC == battery_SOC_Type.normal) &&
      battery_at_OT == battery_at_OT_Type.yes && electric_engine_cable ==
      electric_engine_cable_Type.okay && car_moves == car_moves_Type.yes &&
      desired_acceleration == desired_acceleration_Type.decrease_speed)) {
     mode = Mode_Type.charging;
```

```
} else if ((electric_engine_cable == electric_engine_cable_Type.defective && fuel_tank ==
  fuel_tank_Type.empty) || (battery_at_OT == battery_at_OT_Type.yes &&
  electric_engine_cable == electric_engine_cable_Type.okay && car_moves ==
  car_moves_Type.no && desired_acceleration == desired_acceleration_Type.decrease_speed)
  || (battery_at_OT == battery_at_OT_Type.no && fuel_tank == fuel_tank_Type.empty)) {
 mode = Mode_Type.standstill;

} else if ((battery_SOC == battery_SOC_Type.full && battery_at_OT == battery_at_OT_Type.yes
     && electric_engine_cable == electric_engine_cable_Type.okay && car_moves ==
     car_moves_Type.yes && desired_acceleration ==
     desired_acceleration_Type.decrease_speed)) {
 mode = Mode_Type.mechanical_brake;

} else if (((battery_SOC == battery_SOC_Type.full || battery_SOC ==
     battery_SOC_Type.normal) && battery_at_OT == battery_at_OT_Type.yes &&
     electric_engine_cable == electric_engine_cable_Type.okay && (desired_acceleration ==
     desired_acceleration_Type.keep_speed || desired_acceleration ==
  desired_acceleration_Type.increase_speed))) {
 mode = Mode_Type.discharging;

} else {
 mode = Mode_Type.combustion_engine_only;
}
        }   // Mode_Type.combustion_engine_only

         default: {
                mode = currentMode;
            }
        }   // switch (currentMode)
        return mode;
    }   // hybridCar_ModeSelector
}   // hybridCar
```

9.1.6. SCODE-ANALYZER Report

This section shows a report generated as a Word document (`*.docx`).

SCODE-ANALYZER Report

hybridCar.scode in hybridCar

Generated on [illegible]

This report has been generated by SCODE Workbench [illegible]. SCODE Workbench has been released as an engineering tool and no guarantee is given for the correctness of this output. All contents of this report have to be verified carefully before they are used in further steps of design and implementation.

Generated by SCODE Workbench [illegible] on Apr [illegible]

1

Zwicky Box: hybridCar

Problem Space:

The Zwicky box consists of 7 dimensions and spans space of 288 states and 288 states on input space.

battery SOC	full	empty	normal

battery at OT	yes	no

electric engine cable	okay	defective

silent mode	on	off

fuel tank	empty	not empty

car moves	no	yes

desired acceleration	decrease speed	keep speed	increase speed

Dimension types: Condition dimension, Linked Condition dimension, Foreign Condition dimension, Condition Variant dimension, Action dimension, Linked Action dimension, Info dimension

Description:*Problem Space comments.*

Dimension	Alternative	Comment
battery SOC		SOC = state of charge
battery SOC	full	full
battery at OT		OT = operational temperature

Modes

charging (Start Mode) (Mode1)

Mode charging (Start Mode) (Mode1) is a System mode

Comments:*start mode*

1 of 1	Type: Include Rule	battery SOC = NOT(full) AND battery at OT = yes AND electric engine cable = okay AND car moves = yes AND desired acceleration = decrease speed

Type: Include Rule
Rule: 1
of 1

battery SOC	NOT	full	empty		normal
battery at OT		yes		no	
electric engine cable		okay		defective	
silent mode		on		off	
fuel tank		empty		not empty	
car moves		no		yes	
desired acceleration		decrease speed	keep speed		increase speed

standstill (Mode2)

Mode standstill (Mode2) is a System mode

1 of 3	Type: Include Rule	battery at OT = yes AND electric engine cable = okay AND car moves = no AND desired acceleration = decrease speed
2 of 3	Type: Include Rule	battery at OT = no AND fuel tank = empty
3 of 3	Type: Include Rule	electric engine cable = defective AND fuel tank = empty

Type: Include Rule
Rule: 1
of 3

battery SOC		full	empty		normal
battery at OT		yes		no	
electric engine cable		okay		defective	
silent mode		on		off	
fuel tank		empty		not empty	
car moves		no		yes	
desired acceleration		decrease speed	keep speed		increase speed

Type: Include Rule
Rule: 2
of 3

battery SOC		full	empty		normal
battery at OT		yes		no	
electric engine cable		okay		defective	
silent mode		on		off	
fuel tank		empty		not empty	
car moves		no		yes	
desired acceleration		decrease speed	keep speed		increase speed

Type: Include Rule
Rule: 3

of 3

battery SOC		full	empty	normal
battery at OT		yes	no	
electric engine cable		okay	defective	
silent mode		on	off	
fuel tank		empty	not empty	
car moves		no	yes	
desired acceleration		decrease speed	keep speed	increase speed

mechanical brake (Mode3)

Mode mechanical brake (Mode3) is a System mode

1 of 1	Type: Include Rule	battery SOC = full AND battery at OT = yes AND electric engine cable = okay AND car moves = yes AND desired acceleration = decrease speed

Type: Include Rule
Rule: 1
of 1

battery SOC		full	empty	normal
battery at OT		yes	no	
electric engine cable		okay	defective	
silent mode		on	off	
fuel tank		empty	not empty	
car moves		no	yes	
desired acceleration		decrease speed	keep speed	increase speed

discharging (Mode4)

Mode discharging (Mode4) is a System mode

1 of 1	Type: Include Rule	battery SOC = NOT(empty) AND battery at OT = yes AND electric engine cable = okay AND desired acceleration = NOT(decrease speed)

Type: Include Rule
Rule: 1
of 1

battery SOC	NOT	full	empty	normal
battery at OT		yes	no	
electric engine cable		okay	defective	
silent mode		on	off	
fuel tank		empty	not empty	
car moves		no	yes	
desired acceleration	NOT	decrease speed	keep speed	increase speed

combustion engine only (Mode5)

Mode combustion engine only (Mode5) is a System mode

1 of 3	Type: Include Rule	electric engine cable = defective AND fuel tank = not empty
2 of 3	Type: Include Rule	battery at OT = no AND fuel tank = not empty
3 of 3	Type: Include Rule	battery SOC = empty AND silent mode = off AND fuel tank = not empty AND desired acceleration = (keep speed OR increase speed)

Type: Include Rule
Rule: 1
of 3

battery SOC		full	empty	normal
battery at OT		yes	no	
electric engine cable		okay	defective	
silent mode		on	off	
fuel tank		empty	not empty	
car moves		no	yes	
desired acceleration		decrease speed	keep speed	increase speed

Type: Include Rule
Rule: 2
of 3

battery SOC		full	empty	normal
battery at OT		yes	no	
electric engine cable		okay	defective	
silent mode		on	off	
fuel tank		empty	not empty	
car moves		no	yes	
desired acceleration		decrease speed	keep speed	increase speed

Type: Include Rule
Rule: 3
of 3

battery SOC		full	empty	normal
battery at OT		yes	no	
electric engine cable		okay	defective	
silent mode		on	off	
fuel tank		empty	not empty	
car moves		no	yes	
desired acceleration		decrease speed	keep speed	increase speed

non-system mode (Mode6)

Mode non-system mode (Mode6) is a Non System mode

1 of 2	Type: Include Rule	battery SOC = empty AND battery at OT = yes AND electric engine cable = okay AND silent mode = on AND desired acceleration = (keep speed OR increase speed)
2 of 2	Type: Include	battery SOC = empty AND battery at OT = yes AND electric engine cable = okay AND silent mode

	Rule	= off AND fuel tank = empty AND desired acceleration = (keep speed OR increase speed)

Type: Include Rule
Rule: 1
of 2

battery SOC		full	empty		normal
battery at OT		yes		no	
electric engine cable		okay		defective	
silent mode		on		off	
fuel tank		empty		not empty	
car moves		no		yes	
desired acceleration		decrease speed	keep speed		increase speed

Type: Include Rule
Rule: 2
of 2

battery SOC		full	empty		normal
battery at OT		yes		no	
electric engine cable		okay		defective	
silent mode		on		off	
fuel tank		empty		not empty	
car moves		no		yes	
desired acceleration		decrease speed	keep speed		increase speed

Mode Overview Table

Mode Type	Mode Name	States
Start Mode	charging	8
System Mode	standstill	120
System Mode	mechanical brake	4
System Mode	discharging	32
System Mode	combustion engine only	112
Non System Mode	non-system mode	12

Essential Analysis

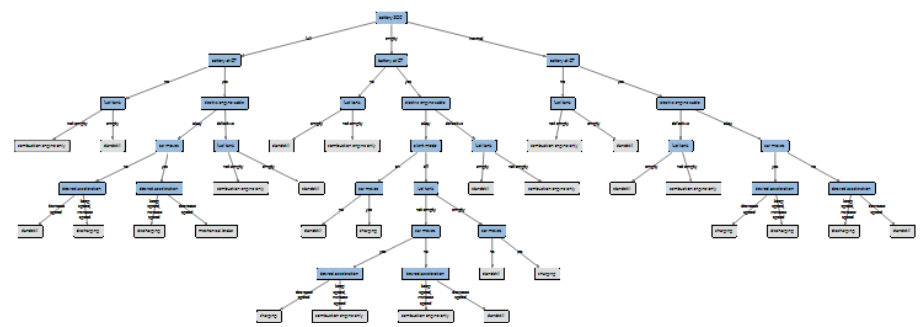
Results of essential analysis according to SCODE method.

Analysis Results

The essential analysis was carried out on the complete problem space
Complete
Consistent
Deterministic
No Actions

Spaces	States	Number of...
problem	288	total states of the complete problem space spanned by the dimensions
covered	288	states covered by any mode
remaining	0	states not covered by any mode

Decision Tree



Events

Transition Table

Source Mode / Target Mode		charging	standstill	mechanical brake	discharging	combustion engine only	No Transition
charging	M1	*	E2	E4	E1	E3	
standstill	M2	E5	*	E6	E7	E8	
mechanical brake	M3	E9	E10	*	E11	E12	
discharging	M4	E13	E14	E15	*	E16	
combustion engine only	M5	E17	E18	E19	E20	*	

charging_discharging[E1]

1 of 1	Type: Include Rule	battery SOC = (full OR normal) AND battery at OT = yes AND electric engine cable = okay AND desired acceleration = (keep speed OR increase speed)

Type: Include Rule
Rule: 1
of 1

battery SOC		full	empty	normal
battery at OT		yes	no	
electric engine cable		okay	defective	
silent mode		on	off	
fuel tank		empty	not empty	
car moves		no	yes	
desired acceleration		decrease speed	keep speed	increase speed

charging_standstill[E2]

1 of 3	Type: Include Rule	electric engine cable = defective AND fuel tank = empty
2 of 3	Type: Include Rule	battery at OT = no AND fuel tank = empty
3 of 3	Type: Include Rule	battery at OT = yes AND electric engine cable = okay AND car moves = no AND desired acceleration = decrease speed

Type: Include Rule
Rule: 1
of 3

battery SOC		full	empty	normal
battery at OT		yes	no	
electric engine cable		okay	defective	
silent mode		on	off	
fuel tank		empty	not empty	
car moves		no	yes	
desired acceleration		decrease speed	keep speed	increase speed

Type: Include Rule
Rule: 2
of 3

battery SOC		full	empty	normal
battery at OT		yes	no	
electric engine cable		okay	defective	
silent mode		on	off	
fuel tank		empty	not empty	
car moves		no	yes	
desired acceleration		decrease speed	keep speed	increase speed

Type: Include Rule
Rule: 3
of 3

battery SOC		full	empty	normal
battery at OT		yes	no	
electric engine		okay	defective	

cable				
silent mode		on	off	
fuel tank		empty	not empty	
car moves		no	yes	
desired acceleration		decrease speed	keep speed	increase speed

charging_ combustionOnly[E3]

1 of 3	Type: Include Rule	battery SOC = empty AND silent mode = off AND fuel tank = not empty AND desired acceleration = (keep speed OR increase speed)
2 of 3	Type: Include Rule	electric engine cable = defective AND fuel tank = not empty
3 of 3	Type: Include Rule	battery at OT = no AND fuel tank = not empty

Type: Include Rule
Rule: 1
of 3

battery SOC		full	empty	normal
battery at OT		yes	no	
electric engine cable		okay	defective	
silent mode		on	off	
fuel tank		empty	not empty	
car moves		no	yes	
desired acceleration		decrease speed	keep speed	increase speed

Type: Include Rule
Rule: 2
of 3

battery SOC		full	empty	normal
battery at OT		yes	no	
electric engine cable		okay	defective	
silent mode		on	off	
fuel tank		empty	not empty	
car moves		no	yes	
desired acceleration		decrease speed	keep speed	increase speed

Type: Include Rule
Rule: 3
of 3

battery SOC		full	empty	normal
battery at OT		yes	no	
electric engine cable		okay	defective	
silent mode		on	off	
fuel tank		empty	not empty	

car moves		no		yes
desired acceleration		decrease speed	keep speed	increase speed

charging_ mechanBrake[E4]

1 of 1	Type: Include Rule	battery SOC = full AND battery at OT = yes AND electric engine cable = okay AND car moves = yes AND desired acceleration = decrease speed

Type: Include Rule
Rule: 1
of 1

battery SOC		full	empty	normal
battery at OT		yes		no
electric engine cable		okay		defective
silent mode		on		off
fuel tank		empty		not empty
car moves		no		yes
desired acceleration		decrease speed	keep speed	increase speed

standstill_ charging[E5]

1 of 1	Type: Include Rule	battery SOC = (empty OR normal) AND battery at OT = yes AND electric engine cable = okay AND car moves = yes AND desired acceleration = decrease speed

Type: Include Rule
Rule: 1
of 1

battery SOC		full	empty	normal
battery at OT		yes		no
electric engine cable		okay		defective
silent mode		on		off
fuel tank		empty		not empty
car moves		no		yes
desired acceleration		decrease speed	keep speed	increase speed

standstill_ mechanBrake[E6]

1 of 1	Type: Include Rule	battery SOC = full AND battery at OT = yes AND electric engine cable = okay AND car moves = yes AND desired acceleration = decrease speed

Type: Include Rule
Rule: 1
of 1

battery SOC		full	empty	normal
battery at OT		yes		no
electric engine cable		okay		defective

silent mode		on		off
fuel tank		empty		not empty
car moves		no		yes
desired acceleration		decrease speed	keep speed	increase speed

standstill_ discharging[E7]

1 of 1	Type: Include Rule	battery SOC = (full OR normal) AND battery at OT = yes AND electric engine cable = okay AND desired acceleration = (keep speed OR increase speed)

Type: Include Rule
Rule: 1
of 1

battery SOC		full	empty	normal
battery at OT		yes		no
electric engine cable		okay		defective
silent mode		on		off
fuel tank		empty		not empty
car moves		no		yes
desired acceleration		decrease speed	keep speed	increase speed

standstill_ combustionOnly[E8]

1 of 3	Type: Include Rule	battery SOC = empty AND silent mode = off AND fuel tank = not empty AND desired acceleration = (keep speed OR increase speed)
2 of 3	Type: Include Rule	battery at OT = no AND fuel tank = not empty
3 of 3	Type: Include Rule	electric engine cable = defective AND fuel tank = not empty

Type: Include Rule
Rule: 1
of 3

battery SOC		full	empty	normal
battery at OT		yes		no
electric engine cable		okay		defective
silent mode		on		off
fuel tank		empty		not empty
car moves		no		yes
desired acceleration		decrease speed	keep speed	increase speed

Type: Include Rule
Rule: 2
of 3

battery SOC		full	empty	normal
battery at OT		yes		no
electric engine		okay		defective

cable							
silent mode		on			off		
fuel tank		empty			not empty		
car moves		no			yes		
desired acceleration		decrease speed		keep speed		increase speed	

Type: Include Rule
Rule: 3
of 3

battery SOC		full		empty		normal	
battery at OT		yes			no		
electric engine cable		okay			defective		
silent mode		on			off		
fuel tank		empty			not empty		
car moves		no			yes		
desired acceleration		decrease speed		keep speed		increase speed	

mechanBrake_ charging[E9]

1 of 1	Type: Include Rule	battery SOC = (empty OR normal) AND battery at OT = yes AND electric engine cable = okay AND car moves = yes AND desired acceleration = decrease speed

Type: Include Rule
Rule: 1
of 1

battery SOC		full		empty		normal	
battery at OT		yes			no		
electric engine cable		okay			defective		
silent mode		on			off		
fuel tank		empty			not empty		
car moves		no			yes		
desired acceleration		decrease speed		keep speed		increase speed	

mechanBrake_ standstill[E10]

1 of 3	Type: Include Rule	battery at OT = yes AND electric engine cable = okay AND car moves = no AND desired acceleration = decrease speed
2 of 3	Type: Include Rule	electric engine cable = defective AND fuel tank = empty
3 of 3	Type: Include Rule	battery at OT = no AND fuel tank = empty

Type: Include Rule
Rule: 1
of 3

battery SOC		full		empty		normal	
battery at OT		yes			no		

electric engine cable		okay			defective		
silent mode		on			off		
fuel tank		empty			not empty		
car moves		no			yes		
desired acceleration		decrease speed		keep speed		increase speed	

Type: Include Rule
Rule: 2
of 3

battery SOC		full		empty		normal	
battery at OT		yes			no		
electric engine cable		okay			defective		
silent mode		on			off		
fuel tank		empty			not empty		
car moves		no			yes		
desired acceleration		decrease speed		keep speed		increase speed	

Type: Include Rule
Rule: 3
of 3

battery SOC		full		empty		normal	
battery at OT		yes			no		
electric engine cable		okay			defective		
silent mode		on			off		
fuel tank		empty			not empty		
car moves		no			yes		
desired acceleration		decrease speed		keep speed		increase speed	

mechanBrake_ discharging[E11]

1 of 1	Type: Include Rule	battery SOC = (full OR normal) AND battery at OT = yes AND electric engine cable = okay AND desired acceleration = (keep speed OR increase speed)

Type: Include Rule
Rule: 1
of 1

battery SOC		full		empty		normal	
battery at OT		yes			no		
electric engine cable		okay			defective		
silent mode		on			off		
fuel tank		empty			not empty		
car moves		no			yes		
desired acceleration		decrease speed		keep speed		increase speed	

mechanBrake_ combustionOnly[E12]

1 of 3	Type: Include Rule	battery SOC = empty AND silent mode = off AND fuel tank = not empty AND desired acceleration = (keep speed OR increase speed)
2 of 3	Type: Include Rule	battery at OT = no AND fuel tank = not empty
3 of 3	Type: Include Rule	electric engine cable = defective AND fuel tank = not empty

Type: Include Rule
Rule: 1
of 3

battery SOC		full	empty	normal
battery at OT		yes		no
electric engine cable		okay		defective
silent mode		on		off
fuel tank		empty		not empty
car moves		no		yes
desired acceleration		decrease speed	keep speed	increase speed

Type: Include Rule
Rule: 2
of 3

battery SOC		full	empty	normal
battery at OT		yes		no
electric engine cable		okay		defective
silent mode		on		off
fuel tank		empty		not empty
car moves		no		yes
desired acceleration		decrease speed	keep speed	increase speed

Type: Include Rule
Rule: 3
of 3

battery SOC		full	empty	normal
battery at OT		yes		no
electric engine cable		okay		defective
silent mode		on		off
fuel tank		empty		not empty
car moves		no		yes
desired acceleration		decrease speed	keep speed	increase speed

discharging_ charging[E13]

1 of 1	Type: Include	battery SOC = (empty OR normal) AND battery at OT = yes AND electric engine cable = okay AND

	Rule	car moves = yes AND desired acceleration = decrease speed

Type: Include Rule
Rule: 1
of 1

battery SOC		full	empty	normal
battery at OT		yes		no
electric engine cable		okay		defective
silent mode		on		off
fuel tank		empty		not empty
car moves		no		yes
desired acceleration		decrease speed	keep speed	increase speed

discharging_ standstill[E14]

1 of 3	Type: Include Rule	battery at OT = yes AND electric engine cable = okay AND car moves = no AND desired acceleration = decrease speed
2 of 3	Type: Include Rule	electric engine cable = defective AND fuel tank = empty
3 of 3	Type: Include Rule	battery at OT = no AND fuel tank = empty

Type: Include Rule
Rule: 1
of 3

battery SOC		full	empty	normal
battery at OT		yes		no
electric engine cable		okay		defective
silent mode		on		off
fuel tank		empty		not empty
car moves		no		yes
desired acceleration		decrease speed	keep speed	increase speed

Type: Include Rule
Rule: 2
of 3

battery SOC		full	empty	normal
battery at OT		yes		no
electric engine cable		okay		defective
silent mode		on		off
fuel tank		empty		not empty
car moves		no		yes
desired acceleration		decrease speed	keep speed	increase speed

Type: Include Rule

Rule: 3
of 3

battery SOC		full	empty	normal
battery at OT		yes	no	
electric engine cable		okay	defective	
silent mode		on	off	
fuel tank		empty	not empty	
car moves		no	yes	
desired acceleration		decrease speed	keep speed	increase speed

discharging_ mechanBrake[E15]

1 of 1	Type: Include Rule	battery SOC = full AND battery at OT = yes AND electric engine cable = okay AND car moves = yes AND desired acceleration = decrease speed

Type: Include Rule
Rule: 1
of 1

battery SOC		full	empty	normal
battery at OT		yes	no	
electric engine cable		okay	defective	
silent mode		on	off	
fuel tank		empty	not empty	
car moves		no	yes	
desired acceleration		decrease speed	keep speed	increase speed

discharging_ combustionOnly[E16]

1 of 3	Type: Include Rule	battery SOC = empty AND silent mode = off AND fuel tank = not empty AND desired acceleration = (keep speed OR increase speed)
2 of 3	Type: Include Rule	electric engine cable = defective AND fuel tank = not empty
3 of 3	Type: Include Rule	battery at OT = no AND fuel tank = not empty

Type: Include Rule
Rule: 1
of 3

battery SOC		full	empty	normal
battery at OT		yes	no	
electric engine cable		okay	defective	
silent mode		on	off	
fuel tank		empty	not empty	
car moves		no	yes	
desired acceleration		decrease speed	keep speed	increase speed

Type: Include Rule
Rule: 2
of 3

battery SOC		full	empty	normal
battery at OT		yes	no	
electric engine cable		okay	defective	
silent mode		on	off	
fuel tank		empty	not empty	
car moves		no	yes	
desired acceleration		decrease speed	keep speed	increase speed

Type: Include Rule
Rule: 3
of 3

battery SOC		full	empty	normal
battery at OT		yes	no	
electric engine cable		okay	defective	
silent mode		on	off	
fuel tank		empty	not empty	
car moves		no	yes	
desired acceleration		decrease speed	keep speed	increase speed

combustionOnly_ charging[E17]

1 of 1	Type: Include Rule	battery SOC = (empty OR normal) AND battery at OT = yes AND electric engine cable = okay AND car moves = yes AND desired acceleration = decrease speed

Type: Include Rule
Rule: 1
of 1

battery SOC		full	empty	normal
battery at OT		yes	no	
electric engine cable		okay	defective	
silent mode		on	off	
fuel tank		empty	not empty	
car moves		no	yes	
desired acceleration		decrease speed	keep speed	increase speed

combustionOnly_ standstill[E18]

1 of 3	Type: Include Rule	electric engine cable = defective AND fuel tank = empty
2 of 3	Type: Include Rule	battery at OT = yes AND electric engine cable = okay AND car moves = no AND desired acceleration = decrease speed
3 of 3	Type: Include Rule	battery at OT = no AND fuel tank = empty

Type: Include Rule
Rule: 1
of 3

battery SOC		full	empty	normal
battery at OT		yes	no	
electric engine cable		okay	defective	
silent mode		on	off	
fuel tank		empty	not empty	
car moves		no	yes	
desired acceleration		decrease speed	keep speed	increase speed

Type: Include Rule
Rule: 2
of 3

battery SOC		full	empty	normal
battery at OT		yes	no	
electric engine cable		okay	defective	
silent mode		on	off	
fuel tank		empty	not empty	
car moves		no	yes	
desired acceleration		decrease speed	keep speed	increase speed

Type: Include Rule
Rule: 3
of 3

battery SOC		full	empty	normal
battery at OT		yes	no	
electric engine cable		okay	defective	
silent mode		on	off	
fuel tank		empty	not empty	
car moves		no	yes	
desired acceleration		decrease speed	keep speed	increase speed

combustionOnly_ mechanBrake[E19]

1 of 1	Type: Include Rule	battery SOC = full AND battery at OT = yes AND electric engine cable = okay AND car moves = yes AND desired acceleration = decrease speed

Type: Include Rule
Rule: 1
of 1

battery SOC		full	empty	normal
battery at OT		yes	no	
electric engine cable		okay	defective	
silent mode		on	off	
fuel tank		empty	not empty	
car moves		no	yes	
desired acceleration		decrease speed	keep speed	increase speed

combustionOnly_ discharging[E20]

Comments:*transition from Combustion Engine Only to Discharging*

1 of 1	Type: Include Rule	battery SOC = (full OR normal) AND battery at OT = yes AND electric engine cable = okay AND desired acceleration = (keep speed OR increase speed)

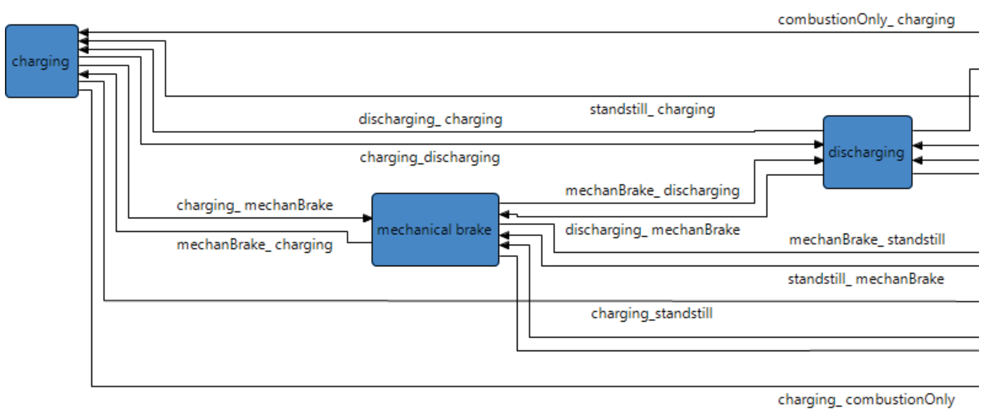
Type: Include Rule
Rule: 1
of 1

battery SOC		full	empty	normal
battery at OT		yes	no	
electric engine cable		okay	defective	
silent mode		on	off	
fuel tank		empty	not empty	
car moves		no	yes	
desired acceleration		decrease speed	keep speed	increase speed

Event Overview Table

Short Name	Name
E1	charging_discharging
E2	charging_standstill
E3	charging_ combustionOnly
E4	charging_ mechanBrake
E5	standstill_ charging
E6	standstill_ mechanBrake
E7	standstill_ discharging
E8	standstill_ combustionOnly
E9	mechanBrake_ charging
E10	mechanBrake_ standstill
E11	mechanBrake_ discharging
E12	mechanBrake_ combustionOnly
E13	discharging_ charging
E14	discharging_ standstill
E15	discharging_ mechanBrake
E16	discharging_ combustionOnly
E17	combustionOnly_ charging
E18	combustionOnly_ standstill
E19	combustionOnly_ mechanBrake
E20	combustionOnly_ discharging

Mode Transition Diagram



9.2. SCODE-CONGRA Tutorial Hints

9.2.1. C Code for Lesson 3

9.2.1.1. C Code for a Flow with Constant

```
1    /**
2    * @warning       AUTOMATICALLY GENERATED FILE! DO NOT EDIT!
3    *
4    * @source        c_F_Constants_in_I.syq
5-7  ...
8    * @options
9    *   Floating point data type Width: 64
10   *   Optimize method code: false
11   *   Validity checks on inputs: reject
12   *   Validity checks on parameters: reject
13   *   Validity checks on states: reject
14   *   Use if statement for conditional expressions: true
15   *   Split complex boolean expressions: false
16   *   Maximum complexity allowed: 5
17   *
18   **/
19
20   #include "c_F_constant_in_I.h"
21
22   void c_F_constant_in_I(double I, double * U) {
23        *U = 2.0 * I;
24   } /* c_F_constant_in_I*/
```

Table 35. Generated C code (`c_F_Constants_in_I.c`) for the `Constants` project (see section 5.4.1). The value of constant R appears in line 23.

9.2.1.2. C Code for a Flow with Parameter

```
1     /**
2     * @warning       AUTOMATICALLY GENERATED FILE! DO NOT EDIT!
3     *
4     * @source        c_F_Parameters_in_I.syq
5-17  ...
18    **/
19
20    #include "c_F_Parameters_in_I.h"
21
22    void c_F_parameter(double I, double R, double * U) {
23         *U = I * R;
24    }   /* c_F_parameter*/
```

Table 36. Generated C code (`c_F_Parameters_in_I.c`) for the `Parameters` project (see section 5.4.2). Parameter R appears in lines 22 and 23.

9.2.1.3. C Code for a Flow with Fixed Variable

```
1     /**
2     * @warning       AUTOMATICALLY GENERATED FILE! DO NOT EDIT!
3     *
4     * @source       c_F_FixedVariable_in_I_fix_R.c
5-17  ...
18    **/
19
20    #include "c_F_FixedVariable_in_I_fix_R.h"
21
22    void c_F_FixedVariable_in_I_fix_R(double I, double * U) {
23         double R = 5.0;
24         *U = I * R;
25    }   /* c_F_FixedVariable_in_I_fix_R*/
```

Table 37. Generated C code (`c_F_FixedVariable_in_I_fix_R.c`) for the `FixedVariable` project (see section 5.4.3). Fixed variable R appears in lines 23 and 24.

9.2.2. C Code for Lesson 4

c_F_Resistor_Power_in_RU.c	**c_F_Resistor_Power_in_IP.c**
/** * @warning AUTOMATICALLY GENERATED FILE! DO NOT EDIT! * * @source c_F_Resistor_Power_in_RU.syq ... **/	/** * @warning AUTOMATICALLY GENERATED FILE! DO NOT EDIT! * * @source c_F_Resistor_Power_in_IP.syq ... **/
#include "c_F_Resistor_Power_in_RU.h" #include "scode.h"	#include "c_F_Resistor_Power_in_IP.h" #include "scode.h"

void c_F_Resistor_Power_in_RU(double R, double U,
        double * I, double * P) {
  if (!scode_double_eq(0.0, R)) {
    *I = U / R;
  } else {
    *I = 0.0;
  } /* Ohms_law(R, U) */
 	
    *P = U * **I;** **/** Resistor_Power_Law(I, U) */
 	
} /* c_F_Resistor_Power_in_RU*/
	void c_F_Resistor_Power_in_IP(double P, double I,
	        double * R, double * U) {
	  if (!scode_double_eq(0.0, I)) {
	    *U = P / I;
	  } else {
	    *U = 0.0;
	  } /* Resistor_Power_Law(I, P) */
	  if (!scode_double_eq(0.0, I)) {
	    *R = *U / I;
	  } else {
	    *R = 0.0;
	  } /* Ohms_law(I, U) */
	} /* c_F_Resistor_Power_in_IP*/

Table 38. Generated C code for both flows in section 5.5

9.2.3. ESDL Code for Lesson 5

c_F_DefinedOutput_in_RU_out_I.esdl

```
/**
* @warning AUTOMATICALLY GENERATED FILE! DO NOT EDIT!
*
* @source c_F_DefinedOutput_in_RU_out_I.syq
...
**/
package DefinedOutput;
class c_F_DefinedOutput_in_RU_out_I {
  public void c_F_DefinedOutput_in_RU_out_I(real R,
         real U, real out I) {
    if (!Math.eq(0.0, R)) {
       I = U / R;
     } else {
      I = 0.0;
     } // Ohms_law(R, U)

   } // c_F_DefinedOutput_in_RU_out_I
} // c_F_DefinedOutput_in_RU_out_I
```

c_F_DefinedOutput_in_RU.esdl

```
/**
* @warning AUTOMATICALLY GENERATED FILE! DO NOT EDIT!
*
* @source c_F_DefinedOutput_in_RU.syq
...
**/
package DefinedOutput;
class c_F_DefinedOutput_in_RU {
  public void c_F_DefinedOutput_in_RU(real R,
       real U, real out I, real out P) {
    if (!Math.eq(0.0, R)) {
       I = U / R;
     } else {
      I = 0.0;
     } // Ohms_law(R, U)
    P = I * U; // Resistor_Power_Law(I, U)
   } // c_F_DefinedOutput_in_RU
} // c_F_DefinedOutput_in_RU_out_I
```

Table 39. Generated ESDL code for the flows with (left) and without (right) explicit output in section 5.6.

9.2.4. Generated Code for Lesson 6

This section shows generated code for the example in section 5.7, “Lesson 6: Algebraic Loop”.

9.2.4.1. Computation SYQ Code

```
/**
* @warning AUTOMATICALLY GENERATED FILE! DO NOT EDIT!
*
* @source c_F_AlgebraicLoop_in_PR
...
**/

package AlgebraicLoop;

computation c_F_AlgebraicLoop_in_PR(R, P)
        implements AlgebraicLoop from F_AlgebraicLoop_in_PR {
  // Variable computation for level 2
  @level(2, 1)
  I = if ((0 <= P*R) && (R!=0))
    then (P*R)^(1/2)/R
  else <- Ohms_law[I, U](R), Resistor_Power_Law[I, U](P);
            // [Source: MuPAD [Incubation]]
  [I,P] = if ((0 <= P*R) && (R!=0))
    then
      if ((0.0 <= P*R) && (((0.0!=P) && (0.0!=R))))
        then 1/(2*(P*R)^(1/2))
      else
  else <- Ohms_law[I, U](R), Resistor_Power_Law[I, U](P);
            // [Source: MuPAD [Incubation]]
  [I,R] = if ((0 <= P*R) && (R!=0))
    then
      if (((0.0 <= P*R) && ((((0.0!=R) && (0.0!=P))
              && (0.0!=R)))) && (0.0!=R))
        then P/(2*R*(P*R)^(1/2))-(P*R)^(1/2)/R^2
      else
  else <- Ohms_law[I, U](R), Resistor_Power_Law[I, U](P);
            // [Source: MuPAD [Incubation]]
  @level(2, 4)
  U = if ((0 <= P*R) && (R!=0))
    then (P*R)^(1/2)
  else <- Ohms_law[I, U](R), Resistor_Power_Law[I, U](P);
            // [Source: MuPAD [Incubation]]
  [U,P] = if ((0 <= P*R) && (R!=0))
  then
    if ((0.0 <= P*R) && (((0.0!=P) && (0.0!=R))))
      then R/(2*(P*R)^(1/2))
    else
  else <- Ohms_law[I, U](R), Resistor_Power_Law[I, U](P);
            // [Source: MuPAD [Incubation]]
```

```
  [U,R] = if ((0 <= P*R) && (R!=0))
    then
      if ((0.0 <= P*R) && (((0.0!=P) && (0.0!=R))))
        then P/(2*(P*R)^(1/2))
      else
  else <- Ohms_law[I, U](R), Resistor_Power_Law[I, U](P);
            // [Source: MuPAD [Incubation]]
}
```

Table 40. `*.syq` file for the computation `c_F_AlgebraicLoop_in_PR`

9.2.4.2. C Code

```
#include "c_F_AlgebraicLoop_in_RP.h"
#include "scode.h"

void c_F_AlgebraicLoop_in_RP(double R, double P, double * I,
            double * U) {
  if (((P * R) >= 0.0) && !scode_double_eq(R, 0.0)) {
    *I = scode_double_pow(P * R, 1.0 / 2.0) / R;
  } else {
    *I = 0.0;
  } /* Ohms_law[I, U](R), Resistor_Power_Law[I, U](P) */
  if (((P * R) >= 0.0) && !scode_double_eq(R, 0.0)) {
    *U = scode_double_pow(P * R, 1.0 / 2.0);
  } else {
    *U = 0.0;
  } /* Ohms_law[I, U](R), Resistor_Power_Law[I, U](P) */
} /* c_F_AlgebraicLoop_in_RP*/
```

Table 41. Generated C code for the flow `F_AlgebraicLoop_in_PR`

9.2.4.3. ESDL Code

```
package AlgebraicLoop;

import math.Math;

class c_F_AlgebraicLoop_in_RP {

  public void c_F_AlgebraicLoop_in_RP(real R, real P,
              real out I, real out U) {
    if (((P * R) >= 0.0) && !Math.eq(R, 0.0)) {
      I = Math.pow(P * R, 1.0 / 2.0) / R;
    } else {
```

```
      I = 0.0;
    } // Ohms_law[I, U](R), Resistor_Power_Law[I, U](P)
    if (((P * R) >= 0.0) && !Math.eq(R, 0.0)) {
      U = Math.pow(P * R, 1.0 / 2.0);
    } else {
      U = 0.0;
    } // Ohms_law[I, U](R), Resistor_Power_Law[I, U](P)
  } // c_F_AlgebraicLoop_in_RP
} // c_F_AlgebraicLoop_in_RP
```

Table 42. Generated ESDL code for the flow `F_AlgebraicLoop_in_PR`

9.2.4.4. MATLAB® Code

```
function [U, I] = c_F_AlgebraicLoop_in_PR(R, P)
  if ((P * R) >= 0.0) && ~eq(R, 0.0)
    I = double(power(P * R, 1.0 / 2.0) / R);
  else
    I = 0.0;
  end % Ohms_law[I, U](R), Resistor_Power_Law[I, U](P)
  if ((P * R) >= 0.0) && ~eq(R, 0.0)
    U = double(power(P * R, 1.0 / 2.0));
  else
    U = 0.0;
  end % Ohms_law[I, U](R), Resistor_Power_Law[I, U](P)
end % Ohms_law[I, U](R), Resistor_Power_Law[I, U](P)
```

Table 43. Generated MATLAB code for the flow `F_AlgebraicLoop_in_PR`

9.2.5. Generated Code for Lesson 7

9.2.5.1. C Code for a Flow with Constraints

```
/**
* @warning AUTOMATICALLY GENERATED FILE! DO NOT EDIT!
*
* @source c_F_ConstraintsVariables_in_RU.syq
...
**/

#include "c_F_ConstraintsVariables_in_RU.h"
#include "scode.h"

void c_F_ConstraintsVariables_in_RU(double R, double U,
            double * I, double * P) {
  if ((R < 100.0) && (R > 0.0) && (U <= 230.0)
            && (U > 0.0)) {
    *I = U / R; /* Ohms_law(R, U) */
    *I = scode_double_min(scode_double_max(*I, 0.0),
            9.999999999999998); /* Ohms_law(R, U) */
    *P = *I * U; /* Resistor_Power_Law(I, U) */
    *P = scode_double_min(scode_double_max(*P,
            2.2250738585072014E-308), 2499.9999999999995);
             /* Resistor_Power_Law(I, U) */
  } else {
    *I = 9.999999999999998;
    *P = 2499.9999999999995;
  }
} /* c_F_ConstraintsVariables_in_RU*/
```

Table 44. `c_F_ConstraintsVariables_in_RU.c` file with constraints, but no verification code (section 5.8.1).

9.2.5.2. C Harness for Flow `F_ConstraintsVariables_in_RU`

```
/**
* @warning AUTOMATICALLY GENERATED FILE! DO NOT EDIT!
*
* @source c_F_ConstraintsVariables_in_RU
...
* @options
* Floating point data type width: 64
* Optimize method code: false
* Validity checks on inputs: limit
* Validity checks on parameters: reject
* Validity checks on states: reject
* Use if statement for conditional expressions: true
* Points per input: 4
* Inform about limitations: true
* Verification threshold: 0.001
*
**/

#include "c_F_ConstraintsVariables_in_RU_harness.h"
#include "scode.h"
#include "c_F_ConstraintsVariables_in_RU.h"

signed int check_c_F_ConstraintsVariables_in_RU(double R,
              double U, double I, double P) {
  signed int errorCount = 0U;
  /* checking correctness for relation Ohms_Law with
              equation U=R*I realized in computation
              step I = U/R */
  if ((I > 0.0) && (I < 9.999999999999998)) {
    /* checking correctness only if the computed value is
              not limited */
    if (scode_double_abs(U - (R * I)) > (0.001 *
              (scode_double_abs(I) + scode_double_abs(R) +
              scode_double_abs(U)))) {
      errorCount = errorCount + 1U;
      scode_printf_info("Error checking computation of I\n
              from equation: \"U=R*I\"\n realized in
              computation step I = U/R\n with values I = %f,
              R = %f, U = %f and error: %f\n", I, R, U,
              scode_double_abs(U - R * I));
    }
```

```
  } else {
    scode_printf_info(" No check due to potential limitation
              of I with value %f in equation: \"U=R*I\" with
              values I = %f, R = %f, U = %f\n", I, I, R, U);
  }
  /* checking correctness for relation Resistor_Power_Law
              with equation P=U*I realized in computation
              step P = I*U */
  if ((P > 2.2250738585072014E-308) &&
              (P < 2499.9999999999995)) {
    /* checking correctness only if the computed value is
              not limited */
    if (scode_double_abs(P - (U * I)) > (0.001 *
              (scode_double_abs(I) + scode_double_abs(P) +
              scode_double_abs(U)))) {
      errorCount = errorCount + 1U;
      scode_printf_info("Error checking computation of P\n
              from equation: \"P=U*I\"\nrealized in
              computation step P = I*U\n with values I = %f,
              P = %f, U = %f and error: %f\n", I, P, U,
              scode_double_abs(P - (U * I)));
    }
  } else {
    scode_printf_info(" No check due to potential limitation
              of P with value %f in equation: \"P=U*I\" with
              values I = %f, P = %f, U = %f\n", P, I, P, U);
  }
  /* checking that the value is also within its limits */
  if (!((I >= 0.0) && (I <= 9.999999999999998))) {
    errorCount = errorCount + 1U;
    scode_printf_info("Value %f for I is out of its range
              [0.0, 10.0)\n", I);
  }
  /* checking that the value is also within its limits */
    if (!((P >= 2.2250738585072014E-308) &&
              (P <= 2499.9999999999995))) {
      errorCount = errorCount + 1U;
      printf("Value %f for P is out of its range (0.0,
              2500.0)\n", P);
  }
  return errorCount;
} /* check_c_F_ConstraintsVariables_in_RU*/

signed int c_F_ConstraintsVariables_in_RU_harness() {
  signed int totalErrorCount = 0U;
  double R_vals[6] = {
    -33.333333333333336,
    2.2250738585072014E-308,
```

```
  33.333333333333336,
  66.66666666666667,
  99.99999999999999,
  133.33333333333334
};

double U_vals[6] = {
  -76.66666666666667,
  2.2250738585072014E-308,
  76.66666666666667,
  153.33333333333334,
  230.0,
  306.6666666666667
};

{
  unsigned char iter_R;
  for (iter_R = 0U; iter_R <= 5U; iter_R++) {
    double R = R_vals[iter_R];
    {
      unsigned char iter_U;
      for (iter_U = 0U; iter_U <= 5U; iter_U++) {
        double U = U_vals[iter_U];
        double I = 0.0;
        double P = 0.0;
        c_F_ConstraintsVariables_in_RU(R, U, &I, &P);
        if ((R > 0.0) && (R < 100.0) && (U > 0.0) && (U <=230.0)) {
          totalErrorCount = totalErrorCount + check_c_F_ConstraintsVariables_in_RU(R, U, I, P);
        } else {
          R = scode_double_min(scode_double_max(R, 2.2250738585072014E-308), 99.99999999999999);
          U = scode_double_min(scode_double_max(U, 2.2250738585072014E-308), 230.0);
          totalErrorCount = totalErrorCount + check_c_F_ConstraintsVariables_in_RU(R, U, I, P);
        }
      }
    }
  }
```

```
  }
  scode_printf_info("Total number of violations is %d\n", +
              totalErrorCount);
  return totalErrorCount;
} /* c_F_ConstraintsVariables_in_RU_harness*/
/* Additional main function for direct execution */

signed int main() {
  signed int totalErrorCount = 0U;
  totalErrorCount = c_F_ConstraintsVariables_in_RU_harness();
  return totalErrorCount;
} /* main*/
```

Table 45. `c_F_ConstraintsVariables_in_RU_harness.c` file (see section 5.8.2)

9.2.5.3. Comparison: Generated Code with/without Parameter Constraint

This section shows the generated computation SYQ, C, ESDL, and MATLAB files for the example in section 5.8.3, "Constraints for Parameters".

c_F_ConstraintsParameters_in_IU.syq

```
...
computation c_F_ConstraintsParameters_in_IU(U)
            implements ConstraintsParameters
            from F_ConstraintsParameters_in_IU {
  // Variable computation for level 2
  @level(2, 1)
  P = I*U <- Resistor_Power_Law(I, U);
              // [Source: Maxima]
  [P,I] = U <- Resistor_Power_Law(I, U);
              // [Source: Maxima]
  [P,U] = I <- Resistor_Power_Law(I, U);
              // [Source: Maxima]
  @level(2, 4)
  R = U/I <- Ohms_Law(I, U);
              // [Source: Maxima]

  [R,I] = -U/I^2 <- Ohms_Law(I, U);
              // [Source: Maxima]

  [R,U] = 1/I <- Ohms_Law(I, U);
               // [Source: Maxima]

}
```

c_F_Parameter_in_IU.syq

```
...
computation c_F_Parameter_in_IU(U)
            implements Parameter
            from F_Parameter_in_IU {
  // Variable computation for level 2
  @level(2, 1)
  P = I*U <- Resistor_Power_Law(I, U);
              // [Source: Maxima]
  [P,I] = U <- Resistor_Power_Law(I, U);
              // [Source: Maxima]
  [P,U] = I <- Resistor_Power_Law(I, U);
              // [Source: Maxima]
  @level(2, 4)
  R = if (0.0!=I) then U/I
    else <- Ohms_Law(I, U);
              // [Source: Maxima]
  [R,I] = if (0.0!=I) then -U/I^2
    else <- Ohms_Law(I, U);
              // [Source: Maxima]
  [R,U] = if (0.0!=I) then 1/I
    else <- Ohms_Law(I, U);
              // [Source: Maxima]
}
```

Table 46. Comparison of computation `*.syq` files with (left) and without (right) parameter constraint

c_F_ConstraintsParameters_in_IU.c

```c
...
#include "c_F_ConstraintsParameters_in_IU.h"

void c_F_ConstraintsParameters_in_IU(double U,
            double I, double * P, double * R) {
  if ((I < 0.0) || (I > 0.0)) {
    *P = U * I; /* Resistor_Power_Law(I, U) */

    *R = U / I; /* Ohms_law(I, U) */
  } else {
    *P = 0.0;
    *R = 0.0;
  }
} /* c_F_ConstraintsParameters_in_IU*/
```

c_F_Parameter_in_IU.c

```c
...
#include "c_F_Parameter_in_IU.h"
#include "scode.h"
void c_F_Parameter_in_IU(double U,
            double I, double * P, double * R) {

  *P = I * U; /* Resistor_Power_Law(I, U) */
  if (!scode_double_eq(0.0, I)) {
    *R = U / I;
  } else {

    *R = 0.0;
  } /* Ohms_law(I, U) */
} /* c_F_Parameter_in_IU*/
```

Table 47. Comparison of generated C code for computations with (left) and without (right) parameter constraint

c_F_ConstraintsParameters_in_IU.esdl

```
...
package ConstraintsParameters;


class c_F_ConstraintsParameters_in_IU {

  public void c_F_ConstraintsParameters_in_IU(real U,
          real I, real out P, real out R) {
    if ((I < 0.0 || I > 0.0)) {
      P = I * U; // Resistor_Power_Law(I, U)

      R = U / I; // Ohms_law(I, U)
    } else {
      P = 0.0;
      R = 0.0;
    }
  } // c_F_ConstraintsParameters_in_IU
} // c_F_ConstraintsParameters_in_IU
```

c_F_Parameter_in_IU.esdl

```
...
package ConstraintsParameters;

import math.Math;

class c_F_Parameter_in_IU {

  public void c_F_Parameter_in_IU(real U,
          real I, real out P, real out R) {

    P = I * U; // Resistor_Power_Law(I, U)
    if (!Math.eq(0.0, I)) {
      R = U / I;
    } else {

      R = 0.0;
    } // Ohms_law(I, U)
  } // c_F_Parameter_in_IU
} // c_F_Parameter_in_IU
```

Table 48. Comparison of generated ESDL code for computations with (left) and without (right) parameter constraint

<table>
<tr><th>c_F_ConstraintsParameters_in_IU.m</th><th>c_F_Parameter_in_IU.m</th></tr>
<tr><td><pre>
...
function [P, R] = c_F_ConstraintsParameters_in_IU(U, I)
  if (I < 0.0) || (I > 0.0)
    P = double(U * I); % Resistor_Power_Law(I, U)

    R = double(U / I); % Ohms_law(I, U)
  else
    P = 0.0;
    R = 0.0;
  end
end % c_F_ConstraintsParameters_in_IU
</pre></td><td><pre>
...
function [P, R] = c_F_Parameter_in_IU(U, I)

  P = double(U * I); % Resistor_Power_Law(I, U)
  if ~eq(0.0, I)
    R = double(U / I);
   else

    R = 0.0;
  end % Ohms_law(I, U)
end % c_F_Parameter_in_IU
</pre></td></tr>
</table>

Table 49. Comparison of generated MATLAB files for computations with (left) and without (right) parameter constraint

9.2.6. Hints for Lesson 8

9.2.6.1. Example: Unit Definitions in a `*.syq` File

```
/*
 * unit definitions
 */

/* length */
unit m;
unit km = 1000.0 * m; /* scaled base unit */

/* time */
unit s is time;

/* mass */
unit kg;
unit g = 1.0e-3 * kg;

/* electric current */
unit A;
unit mA = 0.001 * A;

/* voltage */
unit V = kg *m*m / (A * s*s*s); /* derived from base units */

/* electric resistance */
unit Ohm = V / A;

/* electric power */
unit W = V * A; /* derived from base and derived unit */
```

Table 50. Unit definitions (see section 5.9)

9.2.6.2. C Code for a Flow with Units

```
...

#include "c_F_PhysicalUnits_in_IU.h"
#include "scode.h"

void c_F_PhysicalUnits_in_IU(double I, double U,
            double * P, double * R) {
  *P = U * I * 0.001; /* Resistor_Power_Law(I, U) */
  if (!scode_double_eq(0.0, I)) {
    *R = U / I * 1000.0;
  } else {
    *R = 20.0;
```

```
  } /* Ohms_law(I, U) */
}   /* c_F_PhysicalUnits_in_IU*/
```

Table 51. Generated C code for the flow `F_PhysicalUnits_in_IU` (see section 5.9.5). The units are invisible, but the scaling factor for mA ↔ A is inserted automatically.

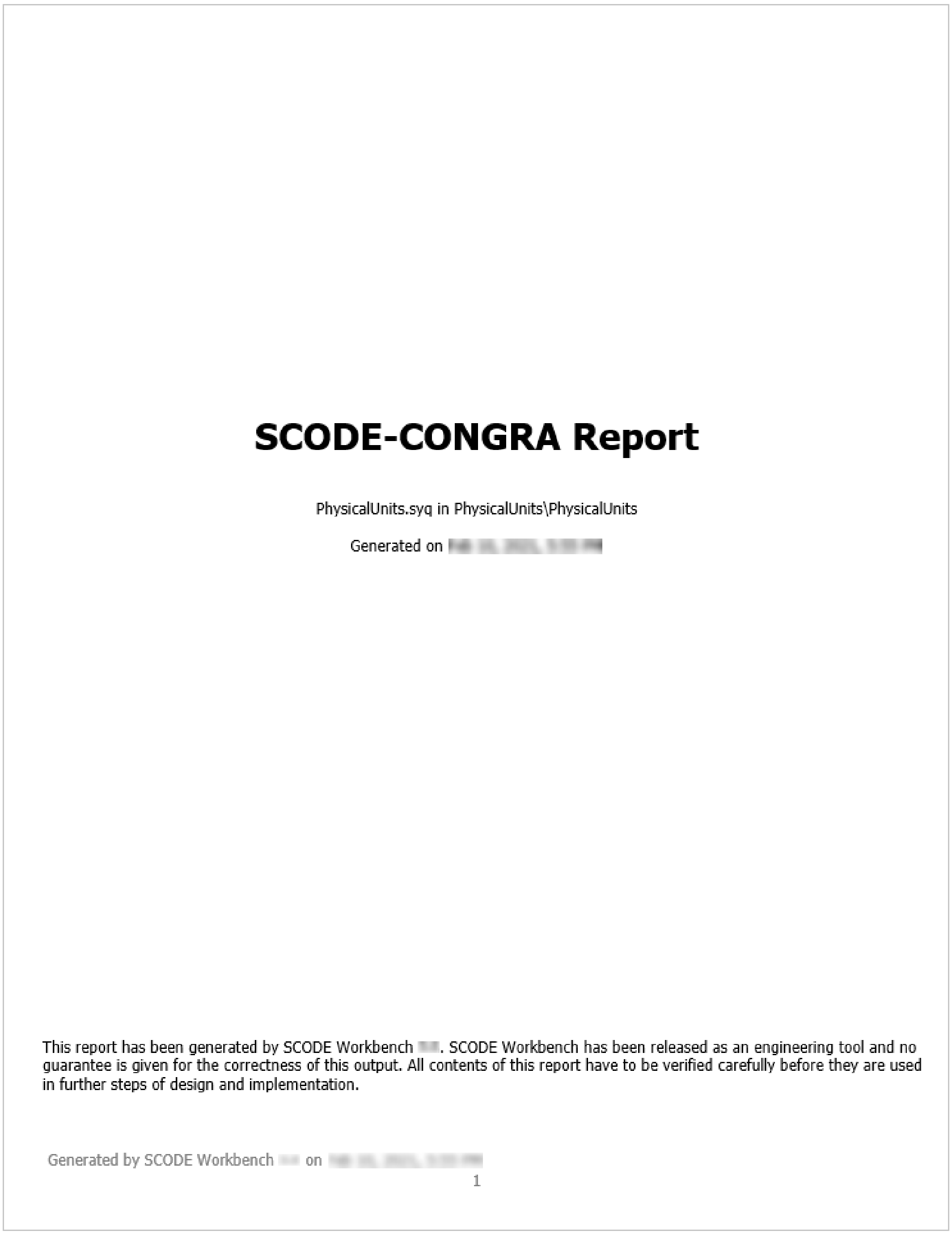
9.2.6.3. MATLAB® Code for a Flow with Units

```
...
function [P, R] = c_F_PhysicalUnits_in_IU(I, U)
  P = double(U * I * 0.001); % Resistor_Power_Law(I, U)
  if ~eq(0.0, I)
    R = double(U / I * 1000.0);
  else
    R = 20.0;
  end % Ohms_law(I, U)
end   % c_F_PhysicalUnits_in_IU
```

Table 52. Generated MATLAB code for the flow `F_PhysicalUnits_in_IU` (see section 5.9.5). The units are invisible, but the scaling factor for mA ↔ A is inserted automatically.

9.2.6.4. SCODE-CONGRA Report

This section shows screenshots of a report generated as a Word document (`*.docx`).



SCODE-CONGRA Report

PhysicalUnits.syq in PhysicalUnits\PhysicalUnits

Generated on [illegible]

This report has been generated by SCODE Workbench [illegible]. SCODE Workbench has been released as an engineering tool and no guarantee is given for the correctness of this output. All contents of this report have to be verified carefully before they are used in further steps of design and implementation.

Generated by SCODE Workbench [illegible] on [illegible]

1

PhysicalUnits

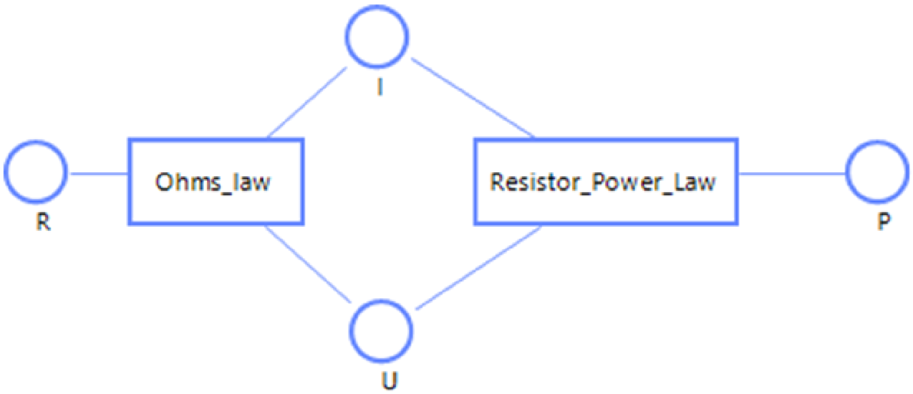
PhysicalUnits.syq

```
package PhysicalUnits;

/*
* unit definitions
*/

/* length */
unit m;

/* time */
unit s is time;

/* mass */
unit kg;

/* electric current */
unit A;
unit mA = 0.001 * A;

/* voltage */
unit V = kg *m*m / (A * s*s*s); /* derived from base units */

/* electric resistance */
unit Ohm = V / A;

/* electric power */
unit W = V * A; /* derived from base and derived unit */

system PhysicalUnits {

@geo(220, 217)
    @description("voltage")
    var V U = 200[V];
    @geo(64, 145)
    @description("resistor")
    var Ohm R = 20 [Ohm];
    @geo(218, 84)
    @description("current")
    var mA I = 0 [A];
    @geo(444, 145)
    @description("power")
    var W P = 1000 [W];

    @geo(120, 144, 80, 40)
    Ohms_law(R, U, I) ::= U = R * I;
    @geo(276, 144, 120, 40)
    Resistor_Power_Law(U, P, I) ::= P = U * I;
}
flow F_PhysicalUnits_in_IU for PhysicalUnits {
    inputs: I, U;
}
```

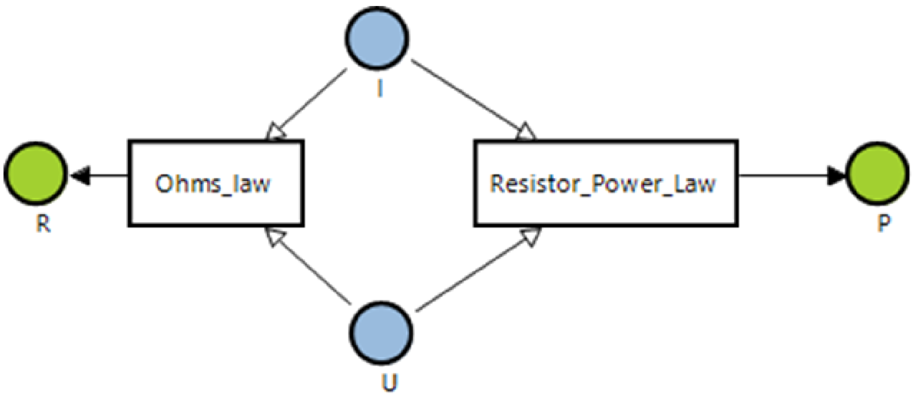
Generated by SCODE Workbench on

2

System: PhysicalUnits



Flow: F_PhysicalUnits_in_IU



Computations:

c_F_PhysicalUnits_in_IU.syq

```
/**
 * @warning    AUTOMATICALLY GENERATED FILE! DO NOT EDIT!
 * @warning    GENERATED BY A NOT RELEASED BUILD! DO NOT USE FOR PRODUCTION!
 *
```

Generated by SCODE Workbench [illegible] on [illegible]

3

```
 * @source       F_PhysicalUnits_in_IU
 *
 * @tool         ETAS SCODE-CONGRA [illegible]
 *
 **/

package PhysicalUnits;

computation c_F_PhysicalUnits_in_IU(I, U) implements PhysicalUnits from F_PhysicalUnits_in_IU {
    // Variable computation for level 2
   @level(2, 1)
   P = U*I <- Resistor_Power_Law(I, U); //  [Source: Built-In Solver]
   [P,I] = U <- Resistor_Power_Law(I, U); //  [Source: Built-In Solver]
   [P,U] = I <- Resistor_Power_Law(I, U); //  [Source: Built-In Solver]
   @level(2, 4)
   R = if (0.0[A]!=I) then U/I else  <- Ohms_law(I, U); //  [Source: Built-In Solver]
   [R,I] = if (0.0[A]!=I) then (-U)/I^2 else  <- Ohms_law(I, U); //  [Source: Built-In Solver]
   [R,U] = if (0.0[A]!=I) then I/I^2 else  <- Ohms_law(I, U); //  [Source: Built-In Solver]
}
```

Table Of Model Elements

Systems

Name	Constants	Library	Image	Extended System
PhysicalUnits		no		

PhysicalUnits

Variables

Variable Name	System Type	Description	Unit	Variable Constraints	Expression	Variable Symbol
U	variable	voltage	V		200[V]	
R	variable	resistor	Ohm		20 [Ohm]	
I	variable	current	mA		0 [A]	
P	variable	power	W		1000 [W]	

Relations

Relation Name	Equation	Subsystem	Description	Image	Relation Symbol
Ohms_law	U = R * I				
Resistor_Power_Law	P = U * I				

Flows

Name	Inputs	Outputs	Extended Flow

Generated by SCODE Workbench [illegible] on [illegible]

4

Name	Inputs	Outputs	Extended Flow
F_PhysicalUnits_in_IU	I U		

Generated by SCODE Workbench [illegible] on [illegible]

5

[32] state of charge

[33] operating temperature

[34] car runs only on the electric motor

10. Contact Information

ETAS Headquarters

ETAS GmbH		
Borsigstraße 24	Phone:	+49 711 3423-0
70469 Stuttgart	Fax:	+49 711 3423-2106
Germany	Internet:	www.etas.com

ETAS Subsidiaries and Technical Support

For details of your local sales office as well as your local technical support team and product hotlines, take a look at the ETAS website:

ETAS subsidiaries	Internet:	www.etas.com/en/contact.php
ETAS technical support	Internet:	www.etas.com/en/hotlines.php

Figures

Tables

Index