EcoStruxure Machine Expert

Code Analysis

User Guide

Original instructions

12/2025

EIO0000002710.08

www.se.com

Legal Information

The information provided in this document contains general descriptions, technical characteristics and/or recommendations related to products/solutions.

This document is not intended as a substitute for a detailed study or operational and site-specific development or schematic plan. It is not to be used for determining suitability or reliability of the products/solutions for specific user applications. It is the duty of any such user to perform or have any professional expert of its choice (integrator, specifier or the like) perform the appropriate and comprehensive risk analysis, evaluation and testing of the products/solutions with respect to the relevant specific application or use thereof.

The Schneider Electric brand and any trademarks of Schneider Electric SE and its subsidiaries referred to in this document are the property of Schneider Electric SE or its subsidiaries. All other brands may be trademarks of their respective owner.

Schneider Electric does not grant any right or license for commercial use of the document or its content, except for a non-exclusive and personal license to consult it on an "as is" basis.

Schneider Electric reserves the right to make changes or updates with respect to or in the content of this document or the format thereof, at any time without notice.

To the extent permitted by applicable law, no responsibility or liability is assumed by Schneider Electric and its subsidiaries for any errors or omissions in the informational content of this document, as well as any non-intended use or misuse of the content thereof.

Table of Contents

Safety Information

Important Information

Read these instructions carefully, and look at the equipment to become familiar with the device before trying to install, operate, service, or maintain it. The following special messages may appear throughout this documentation or on the equipment to warn of potential hazards or to call attention to information that clarifies or simplifies a procedure.

The addition of this symbol to a “Danger” or “Warning” safety label indicates that an electrical hazard exists which will result in personal injury if the instructions are not followed.

This is the safety alert symbol. It is used to alert you to potential personal injury hazards. Obey all safety messages that follow this symbol to avoid possible injury or death.

⚠ DANGER

DANGER indicates a hazardous situation which, if not avoided, **will result in** death or serious injury.

⚠ WARNING

WARNING indicates a hazardous situation which, if not avoided, **could result in** death or serious injury.

⚠ CAUTION

CAUTION indicates a hazardous situation which, if not avoided, **could result** in minor or moderate injury.

NOTICE

NOTICE is used to address practices not related to physical injury.

Please Note

Electrical equipment should be installed, operated, serviced, and maintained only by qualified personnel. No responsibility is assumed by Schneider Electric for any consequences arising out of the use of this material.

A qualified person is one who has skills and knowledge related to the construction and operation of electrical equipment and its installation, and has received safety training to recognize and avoid the hazards involved.

About the Document

Document Scope

This document describes the graphical user interface of the Code Analysis and the functions it provides.

Validity Note

This document has been updated for the release of EcoStruxure™ Machine Expert V2.5.

The characteristics that are described in the present document, as well as those described in the documents included in the Related Documents section below, can be found online. To access the information online, go to the Schneider Electric home page www.se.com/ww/en/download/.

The characteristics that are described in the present document should be the same as those characteristics that appear online. In line with our policy of constant improvement, we may revise content over time to improve clarity and accuracy. If you see a difference between the document and online information, use the online information as your reference.

Product Related Information

⚠WARNING

LOSS OF CONTROL

- Perform a Failure Mode and Effects Analysis (FMEA), or equivalent risk analysis, of your application, and apply preventive and detective controls before implementation.
- Provide a fallback state for undesired control events or sequences.
- Provide separate or redundant control paths wherever required.
- Supply appropriate parameters, particularly for limits.
- Review the implications of transmission delays and take actions to mitigate them.
- Review the implications of communication link interruptions and take actions to mitigate them.
- Provide independent paths for control functions (for example, emergency stop, over-limit conditions, and error conditions) according to your risk assessment, and applicable codes and regulations.
- Apply local accident prevention and safety regulations and guidelines.[1]
- Test each implementation of a system for proper operation before placing it into service.

Failure to follow these instructions can result in death, serious injury, or equipment damage.

[1] For additional information, refer to NEMA ICS 1.1 (latest edition), *Safety Guidelines for the Application, Installation, and Maintenance of Solid State Control* and to NEMA ICS 7.1 (latest edition), *Safety Standards for Construction and Guide for Selection, Installation and Operation of Adjustable-Speed Drive Systems* or their equivalent governing your particular location.

⚠WARNING

UNINTENDED EQUIPMENT OPERATION

- Only use software approved by Schneider Electric for use with this equipment.
- Update your application program every time you change the physical hardware configuration.

Failure to follow these instructions can result in death, serious injury, or equipment damage.

General Cybersecurity Information

In recent years, the growing number of networked machines and production plants has seen a corresponding increase in the potential for cyber threats, such as unauthorized access, data breaches, and operational disruptions. You must, therefore, consider all possible cybersecurity measures to help protect assets and systems against such threats.

To help keep your Schneider Electric products secure and protected, it is in your best interest to implement the cybersecurity best practices as described in the Cybersecurity Best Practices document.

Schneider Electric provides additional information and assistance:

- Subscribe to the Schneider Electric security newsletter.
- Visit the Cybersecurity Support Portal web page to:
  - Find Security Notifications.
  - Report vulnerabilities and incidents.
- Visit the Schneider Electric Cybersecurity and Data Protection Posture web page to:
  - Access the cybersecurity posture.
  - Learn more about cybersecurity in the cybersecurity academy.
  - Explore the cybersecurity services from Schneider Electric.

Product Related Cybersecurity Information

Refer to the Cybersecurity Guidelines for EcoStruxure Machine Expert, Modicon and PacDrive Controllers and Associated Equipment User Guide.

Related Documents

Title of documentation	Reference number
Cybersecurity Best Practices	CS-Best-Practices-2019-340
Cybersecurity Guidelines for EcoStruxure Machine Expert, Modicon and PacDrive Controllers and Associated Equipment	EIO0000004242

Title of documentation	Reference number
EcoStruxure Machine Advisor Code Analysis User Guide	EIO0000003915 (ENG) EIO0000003917 (FRE) EIO0000003916 (GER) EIO0000003918 (SPA) EIO0000003919 (ITA)
EcoStruxure Machine Expert Programming Guide	EIO0000002854 (ENG) EIO0000002855 (FRE) EIO0000002856 (GER) EIO0000002858 (SPA) EIO0000002857 (ITA) EIO0000002859 (CHS)

To find documents online, visit the Schneider Electric download center (www.se.com/ww/en/download/).

Terminology Derived from Standards

The technical terms, terminology, symbols and the corresponding descriptions in the information contained herein, or that appear in or on the products themselves, are generally derived from the terms or definitions of international standards.

In the area of functional safety systems, drives and general automation, this may include, but is not limited to, terms such as *safety*, *safety function*, *safe state*, *fault*, *fault reset*, *malfunction*, *failure*, *error*, *error message*, *dangerous*, etc.

Among others, these standards include:

Standard	Description
IEC 61131-2:2007	Programmable controllers, part 2: Equipment requirements and tests.
ISO 13849-1:2023	Safety of machinery: Safety related parts of control systems. General principles for design.
EN 61496-1:2020	Safety of machinery: Electro-sensitive protective equipment. Part 1: General requirements and tests.
ISO 12100:2010	Safety of machinery - General principles for design - Risk assessment and risk reduction
EN 60204-1:2006	Safety of machinery - Electrical equipment of machines - Part 1: General requirements
ISO 14119:2013	Safety of machinery - Interlocking devices associated with guards - Principles for design and selection
ISO 13850:2015	Safety of machinery - Emergency stop - Principles for design
IEC 62061:2021	Safety of machinery - Functional safety of safety-related electrical, electronic, and electronic programmable control systems
IEC 61508-1:2010	Functional safety of electrical/electronic/programmable electronic safety-related systems: General requirements.
IEC 61508-2:2010	Functional safety of electrical/electronic/programmable electronic safety-related systems: Requirements for electrical/electronic/programmable electronic safety-related systems.
IEC 61508-3:2010	Functional safety of electrical/electronic/programmable electronic safety-related systems: Software requirements.
IEC 61784-3:2021	Industrial communication networks - Profiles - Part 3: Functional safety fieldbuses - General rules and profile definitions.
2006/42/EC	Machinery Directive

Standard	Description
2014/30/EU	Electromagnetic Compatibility Directive
2014/35/EU	Low Voltage Directive

In addition, terms used in the present document may tangentially be used as they are derived from other standards such as:

Standard	Description
IEC 60034 series	Rotating electrical machines
IEC 61800 series	Adjustable speed electrical power drive systems
IEC 61158 series	Digital data communications for measurement and control – Fieldbus for use in industrial control systems

Finally, the term *zone of operation* may be used in conjunction with the description of specific hazards, and is defined as it is for a *hazard zone* or *danger zone* in the *Machinery Directive* (*2006/42/EC*) and *ISO 12100:2010*.

NOTE: The aforementioned standards may or may not apply to the specific products cited in the present documentation. For more information concerning the individual standards applicable to the products described herein, see the characteristics tables for those product references.

Introduction

General Information on the Code Analysis Component

Overview

Code Analysis is integrated into EcoStruxure Machine Expert to analyze applications.

Code Analysis focuses on the following key elements:

- Understanding the structure of the source code. Software developers can visualize, for example, code dependencies and explore it step by step.
- Identification of code deficiencies to harmonize and improve the source code by defined programming guidelines.
- Measurement of source code quality and identifying the KPIs (Key Performance Indicators).
- Reporting the KPIs to the software developers for personal purpose.

System Requirements

Besides the system requirements for EcoStruxure Machine Expert, Code Analysis has additional requirements regarding the hardware and the operating system.

The minimum requirements are:

- 4-core processor for parallel query execution
- 4 GB RAM

To analyze projects, the following specifications are suggested:

- Windows 64 bit
- ≥ 4-core processor for parallel query execution
- ≥ 8 GB RAM

Installation

To use Code Analysis in EcoStruxure Machine Expert, the component has to be installed with the Schneider Electric Software Installer.

Code Analysis is under license protection:

Function	Without license	With license
Dependency View	Limited to two diagrams.	Not limited.
Metrics Table	Limited to two objects and three active queries.	Not limited.
Conventions Table	Limited to two objects and three active queries.	Not limited.
Query Editor	Not available.	Available.
Python CodeAnalysis API	Not available.	Available.

NOTE: For more information about product licensing, contact your local Schneider Electric representative.

Code Analysis Editors

With the three code analysis editors in EcoStruxure Machine Expert you can analyze and interpret the results of a code analysis:

- **Conventions Table** editor, page 18

  Parts of the application that violate the defined coding conventions (based on coding rules).
- **Dependency View** editor, page 21

  Dependencies between namespaces, libraries, objects (function blocks, POUs, and so on).
- **Metrics Table** editor, page 19

  Results of code quality figures, like LOC (Lines of Code), complexity, and so on.

Multiple instances of code analysis objects can be added beneath **Application**, folders, or **Code Analysis Manager**.

Several code analysis objects can be added to an application.

If a **Conventions Table** or **Metrics Table** object is added beneath **Application** or **Code Analysis Manager**, the whole application is in the scope of the analysis.

If they are added beneath a folder, only this folder is in the scope of the analysis.

So it is possible to store analysis results for example, for each module/folder of an application in dedicated objects.

Adding a **Conventions Table**, **Dependency View**, or a **Metrics Table** to an application automatically adds a **Code Analysis Manager** to the **Application** object.

Block List

A **BlockList** object can be added beneath a **Code Analysis Manager** object. The **BlockList** object contains elements that will not appear in **Conventions Table**, **Metrics Table**, or **Dashboard** results. This, although, is taken into account when you upload a project snapshot, page 31 into Machine Advisor Code Analysis.

Code Analysis Manager

The **Code Analysis Manager** provides a quick overview through a dashboard and you can configure analysis depth and the cloud connection.

The **Code Analysis Manager** provides the tabs:

- **Dashboard** tab, page 29

  Overview of the analyzed application.
- **Configuration** tab, page 30

  Analysis depth of code analysis configuration.
- **Cloud Connection** tab, page 31

  Configuration of the connection to the Machine Advisor Code Analysis.

Code Analysis Query Manager

The **Code Analysis Query Manager** allows you to create and modify customized rule sets, and to manage your metrics and conventions queries. To open the **Code Analysis Query Manager** click **Tools** in the menu bar and select **Code Analysis Query Manager** from the contextual menu.

The **Code Analysis Query Manager** provides the tabs:

- **Rule Sets**

  Rule sets can be created and modified.
- **Metric Queries** tab, page 62

  Queries and query chains available in the **Metrics Table** editor can be created and modified.
- **Convention Queries** tab, page 85

  Queries and query chains available in the **Conventions Table** editor can be created and modified.
- **Dependency (Filter) Queries** tab, page 56

  Queries and query chains available in the **Filters** of the **Dependency View** editor can be created and modified.
- **Dependency (Select) Queries** tab, page 58

  Queries and query chains available in the **Select and Add** dialog box of the **Dependency View** editor can be created and modified.
- **Cloud Connection** tab, page 39

  The connection to Machine Advisor Code Analysis can be configured.

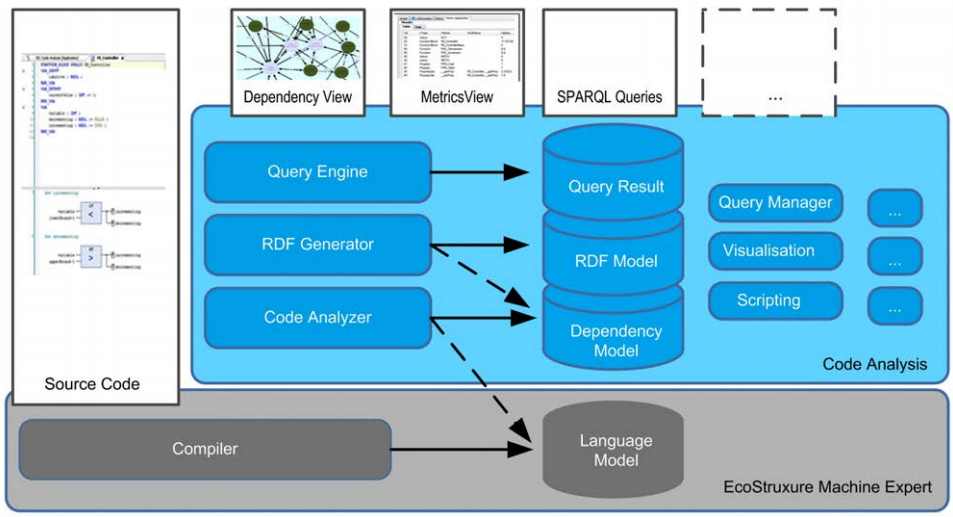
Concept of Code Analysis

Overview

This chapter gives an overview on the concepts of Code Analysis as integrated into EcoStruxure Machine Expert.

Software Components of Code Analysis

The diagram gives an overview of the high-level software components of Code Analysis:



The components can be categorized into three different types:

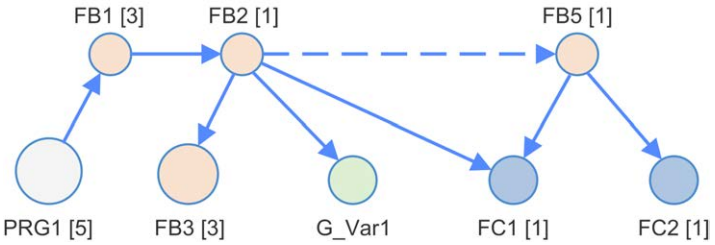
- UI components displaying data:
  - Editors to write the source code.
  - Editors to visualize the results like metrics or conventions, or a graphical representation of the source code structure.

- Data models as input or output of other components:
  - Language model
  - Dependency model
  - Resource Description Framework (RDF) model
  - Query results
- Components transforming data:
  - The source code compiler (with language model as output) processes the source code to check the syntax and build the language model to generate the executable code running on controllers.
  - The source code analyzer (with dependency model as output) analyzes the language model and transforms it into a dependency model (and keeps it up-to-date).
  - The RDF model generator (with RDF model as output) transforms the dependency model into an RDF model to build the bridge to the semantic Web technologies.
  - The Query execution engine (with query results as output) executes SPARQL queries on the RDF model to get the query results.

Analysis Data (Dependency Model) Concept

The application is analyzed and a dependency model is built.

The dependency model is a list of nodes connected through edges.



Examples of node types:

Node type	Description
Function block	Function block (FB) inside the dependency model. Created for every function block added to the EcoStruxure Machine Expert project.
Program	Progam (PRG) inside the dependency model. Created for every program added to the EcoStruxure Machine Expert project.
Function	Function (FC) inside the dependency model. Created for every function added to the EcoStruxure Machine Expert project.
...	...

Examples of edge types:

Edge type	Description
Read	Read operation from code as source to a variable node as target.
Write	Write operation from code as source to a variable node as target.
Call	Call of a function block, method, action, program, and so on, from the code as source to a target node.
Extend	Extension of a basis type. For example, FB extension by another function block.
...	...

Semantic Web Technologies

The open and flexible code analysis feature is based on semantic Web technologies. Some of these technologies are:

- Resource Description Framework (RDF) - RDF Model

  Refer to https://en.wikipedia.org/wiki/Resource_Description_Framework.
- RDF Database (Semantic Web Database) - an RDF Triple Storage

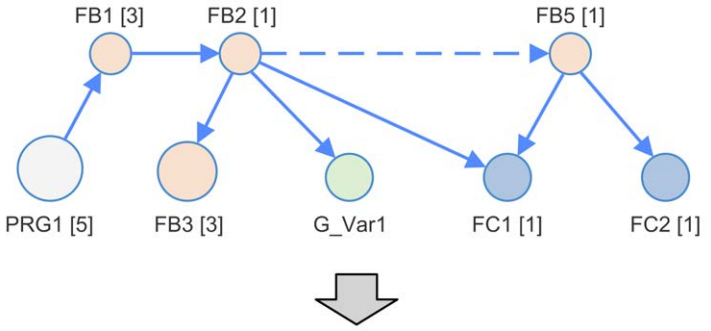
  Refer to https://en.wikipedia.org/wiki/Triplestore
- SPARQL Protocol and RDF Query Language - SPARQL.

  Refer to https://en.wikipedia.org/wiki/SPARQL.

Dependency Model to RDF Model Synchronization

The dependency model is the result of a code analysis run.

To link up to an open, flexible code analysis feature with query language support, the dependency model is synchronized with an RDF model.



Source	Edge type	Target
FB2	Call	FB3
FB2	Read	G_Var1
FB2	Implement	FB5

RDF Triple Storage

To support the analysis of large projects, the RDF model is kept in a separate process called **RDF Triple Storage**.

By default, the **RDF Triple Storage** is used. If required, the behavior can be configured in the **Code Analysis Manager**.

SPARQL and RDF

Resource Description Framework (RDF) is a data model for describing resources and the relations between these resources.

Example:

:(Subject)	:(Predicate)	:(Object)
:Car	:Weights	:1000 kg
:Car	:ConsistsOf	:Wheels
:Car	:ConsistsOf	:Engine

SPARQL is an acronym for Sparql Protocol and RDF Query Language. The SPARQL specification (https://www.w3.org/TR/sparql11-overview/) provides languages and protocols to query and manipulate RDF graphs - similar to SQL queries.

Example of s simple SPARQL query to get the node Ids and their names of the function blocks of an RDF model:

```
SELECT ?NodeId ?Name
WHERE {
       # Select all FunctionBlocks and their names
       ?NodeId a :FunctionBlock ;
             :Name ?Name .
}
```

Code Analysis Editors

Conventions Table

Conventions Table

Overview

With the **Conventions Table**, you can select conventions that have to be met by your application.

The **Conventions Table** provides two parts:

- **Conventions** (left-hand side)

  List of the available conventions.

  Refer to chapter Convention Queries Tab, page 85.
- **Results** window (right-hand side)

  List of elements that do not meet the conventions selected on the left side.

Conventions

The **Conventions** tree lists and groups convention rules that are available by default (by EcoStruxure Machine Expert installation) and the user-defined rules created with the **Query editor**. Use the check boxes to activate/deactivate convention rules.

After clicking the **Enable and start querying** button of the **Results** window toolbar, additional information is displayed in the **Conventions** tree:

Element	Description
Filter by convention names	Enter text to filter the conventions by name.
Rule Set	Select a rule set for filtering the conventions to display.
(<Number>) behind a convention name	Indicates how many hits this convention created. If no number is displayed, this query was not executed.
(Querying...) behind a convention name	This query is being executed.

Toolbar

Element	Description
Analyze Code	Click this button to start the analysis process of the application this **Conventions Table** belongs to. If analysis has already been started and state of analysis is up-to-date, this button is disabled.
Stage x/y (Required: z)	Stage information assigned to the **Analyze Code** button: • Number of stages available (**y**) • Currently reached stage (**x**) • Required stage (**z**) to use this editor
Querying	• **Enable and start querying** button: Click this button to start querying. • **Disable and pause querying** button: Click this button to pause querying. Click the button again to restart querying from the point you stopped querying.

Element	Description
Export	Click this button to export the displayed list as a CSV, HTML, or XML file.
Scope	Displays the folder of which the **Conventions Table** is a part. Only this folder is taken into account for analysis. For a table that is a subnode of the **Code Analysis Manger** the whole application is taken into account for analysis. Only visible if a **Conventions Table** is placed in a folder.

Result List

Element	Description
Filter by ...	Enter text to filter the results. The filter applies to all columns.
Sort a column	To sort a column, click the column header: • **Convention** • **Severity** • **FullName** • **Message**
Double-click a table entry	Opens the associated element in its corresponding editor.
Contextual Menu Commands on a table entry	Right-click a table entry and select one of the contextual menu commands: • **Go to definition** Opens the associated element in its corresponding editor. • **Browse in Cross Reference View** Opens the selected node in the Cross Reference List View. Refer to the Menu Commands Online Help\View Menu Commands\Cross Reference List View. • **Start Rename Refactoring** Opens the **Rename** dialog box to enter the new name of the object. After confirming with **OK**, the **Refactoring** dialog box opens, highlighting the objects of your application that are affected by the renaming. Click **OK** to start the renaming. This contextual menu command is available for name verification conventions, for example, **Complex Type Name Checks**. • **Add to dependency graph** Adds the element to an existing or new dependency graph. • **Add to block list** Adds the element to a block list and removes that element form all **Conventions Tables** until it is deleted from block list again.

Metrics Table

Metrics Table

Overview

With the **Metrics Table**, you can select metrics to be executed on an application and filter and list results.

The **Metrics Table** provides two parts:

- **Metrics** (left-hand side)

  List of the available metrics., page 62

- **Results** window (right-hand side)

  List of metric results.

Metrics

The **Metrics** tree lists metrics that are available by default (by EcoStruxure Machine Expert installation) and the user-defined metrics created with the **Query editor**. Use the check boxes to activate/deactivate metrics.

After clicking the **Enable and start querying** button of the **Results** window toolbar, additional information is displayed in the **Metrics** tree:

Element	Description
Filter by metric names	Enter text to filter the metrics by name.
Rule Set	Select a rule set for filtering the metrics to display.
(<Number>) behind a metric name	Indicates how many hits this metric created. If no number is displayed, this query was not executed.
(Querying...) behind a metric name	This query is being executed.

Toolbar

Element	Description
Analyze Code	Click this button to start the analysis process of the application this **Metrics Table** belongs to. If analysis has already been started and state of analysis is up-to-date, this button is disabled.
Querying	• **Enable and start querying** button: Click this button to start querying. • **Disable and pause querying** button: Click this button to pause querying. Click the button again to restart querying from the point you stopped querying.
Export	Click this button to export the displayed list as a CSV, HTML, or XML file.
Scope	Displays the folder of which the **Metrics Table** is a part. Only this folder is taken into account for analysis. For a table that is a subnode of the **Code Analysis Manger** the whole application is taken into account for analysis. Only visible if a **Metrics Table** is placed in a folder.

Result List

Element	Description
Filter by ...	Enter text to filter the results. The filter applies to all columns.
Sort a column	To sort a column, click the column header: • **Type** • **Name** • **FullName** • **<Metric Name>**

Element	Description
Double-click a table entry	Opens the associated element in its corresponding editor.
Contextual Menu Commands on a table entry	Right-click a table entry and select one of the contextual menu commands: • **Go to definition** Opens the associated element in its corresponding editor. • **Browse in Cross Reference View** Opens the selected node in the Cross Reference List View. Refer to the Menu Commands Online Help\View Menu Commands\Cross Reference List View. • **Add to dependency graph** Adds the element to an existing or new dependency graph. • **Add to block list** Adds the element to a block list and removes that element form all **Metrics Tables** until it is deleted from block list again.

Dependency View

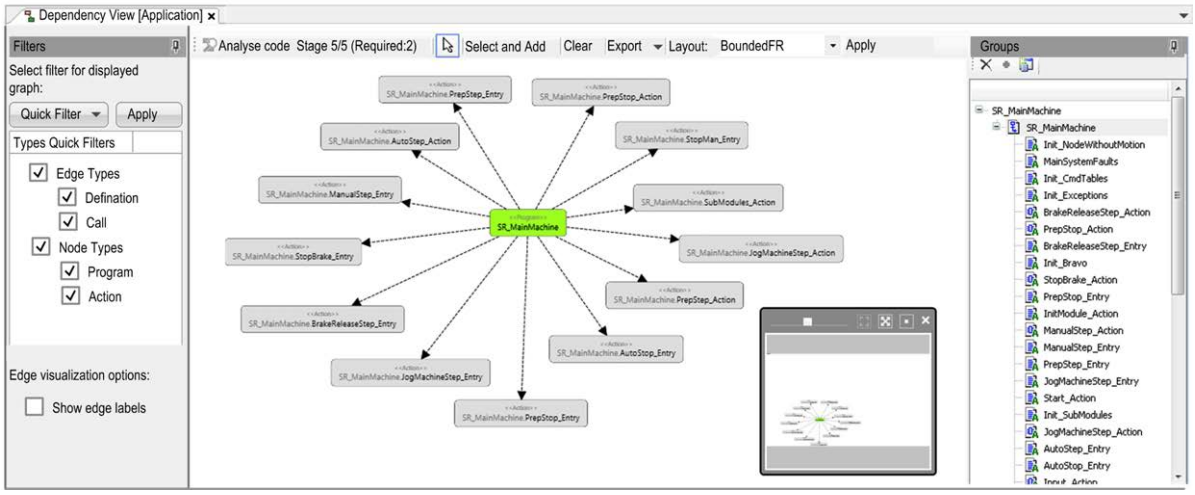
Dependency View (Overview)

Overview

With the **Dependency View**, you can visualize the dependencies of your analyzed application as a dependency graph. You can select the content and the layout of the dependency graph.

The **Dependency View** provides three parts:

- **Filters** (left-hand side), page 22
  You can filter and configure the dependency graph.
- **Dependency Graph** (main window), page 22
  The displayed graph represents dependencies between elements of the analyzed application.
- **Groups** (right-hand side), page 26
  You can structure the application using the **Groups** tree.



Dependency View (Filters)

Overview

Element	Description
Select filter for displayed graph	Select a filter from the list. The filters are defined in the **Dependency (Filter) Queries** tab of the **Code Analysis Query Manager**. Refer to chapter Dependency (Filter) Queries Tab, page 56.
Apply	Click this button to apply the filter to the displayed graph.
Types Quick Filters	This tree lists the **Edge Types** and **Node Types** based on the content in the **Dependency Graph**. If you activate / deactivate **Edge Types** or **Node Types** in the list, the displayed graph is updated immediately.
Edge visualization options	• **Show edge labels** Enables / disables the labels attached to the edges inside the **Dependency Graph**. • **Display parallel Edges** Enables / disables to display parallel edges.

Dependency View (Dependency Graph)

Overview

Also refer to: Contextual Menu Commands of the Dependency Graph, page 24

Toolbar

Element	Description
Analyze Code	Click this button to start the analysis process of the application this **Dependency View** belongs to. If analysis has already been started and state of analysis is up-to-date, this button is disabled.
Select Nodes / Find and Select (binoculars icon)	After code analysis run, click this button to search for objects that are available in your **Dependency View**, and on which you want to set the focus. After confirming with **OK**, the selected objects have the focus in your **Dependency View**, and are highlighted.
Mouse Pointer	Click this button to switch from move mode to select mode.

Element	Description
Select and Add	After code analysis run, click this button to define the **Select Scope** and the **Select Query**. The following **Scopes** are available: • **Project + Libraries** Expected results are the objects from **Application**, from **POU** space, and from the libraries referenced by the **Application**. If the devices are considered in the analysis, they are also part of the scope. For this, the option **Consider Devices** in the **Configuration** tab of the **Code Analysis Manager** must be activated. Refer to Configuration\Consider Devices, page 30. **NOTE:** This can result in a large number of objects. • **Project** Expected results are the objects from **Application** and from **POU** space. If the devices are considered in the analysis, they are also part of the scope. For this, the option **Consider Devices** in the **Configuration** tab of the **Code Analysis Manager** must be activated. Refer to Configuration\Consider Devices, page 30. Library objects are not reported in the results. • **Application** Expected results are the objects defined in **Application** only. • **POU** space Expected results are the objects defined in **POU** space only. Refer to chapter Dependency (Select) Queries Tab, page 58.
Search and Add	After code analysis run, click this button to search for objects that should be added to your **Dependency View**. • **Filter by …** Click the button in front of **Filter by …** to define the scope of the objects that are displayed in the table of objects below. By default, all objects of your project are displayed. Enter plain text to search for objects. This search is not applied to the **Type** column. • Table of objects ◦ The **Name** column provides a check box for each object. Activate a check box to add the respective object to your **Dependency View**. ◦ **Data Type** ◦ **Type** ◦ **FullName** • **All** Click this button to select all objects. • **None** Click this button to deselect all objects. • **OK** Click this button to confirm your selection. The selected objects are added to your **Dependency View**. • **Cancel** Click this button to close the dialog box without adding objects to your **Dependency View**.
Clear	Click this button to remove all nodes from the displayed graph.
Export	Click this button to export the displayed graph as an image (*.jpg, *.png, *.bmp)
Layout	Select a predefined layout for the displayed graph. Several circular, force-directed, and tree layouts are available.
Apply	Click this button to rearrange the objects in the displayed graph according to the selected layout.

Zoom Control

With the zoom control in the lower right corner of the dependency graph window, you can modify the size and the position of the displayed graph.

Type	Description
Slider	Use the slider to zoom in or to zoom out.
Fill Bounds with Content	Click this button to enlarge the graph to the size of the dependency graph.
Center Content	Click this button to center the graph in the dependency graph.
Close	Click this button to close the zoom control. To reopen, click the icon in the lower right corner of the dependency graph.

To move the displayed content in the dependency graph window, **click into the window > hold and move the mouse key**.

Drag-and Drop Nodes to the Dependency Graph

You can also add nodes and subnodes to the dependency graph by drag-and-drop. Therefore drag a node from the **Application** tree and drop it to the dependency graph window.

Dependency View (Contextual Menu Commands of the Dependency Graph)

Contextual Menu Commands of the Graph Nodes

Menu command	Description
Select all	Selects the nodes on graph. Can be executed anywhere on graph.
Go to definition	Opens the editor where the selected node is defined. Cannot be executed on the following nodes: **Application**, **POUspace**, **Task**, **Folder**.
Browse in Cross Reference View	Opens the selected node in the Cross Reference List View. Refer to the Menu Commands Online Help\View Menu Commands\Cross Reference List View.
Remove node(s)	Removes the selected node(s) from graph.
Remove all except me	Removes the node(s) from graph except the one that is selected.
Remove all except me and my relations	Removes the node(s) from graph except the one that is selected and its related nodes. Cannot be executed on groups.
Delete group(s)	Deletes the selected groups from dependency view and removes it completely from project. The assigned nodes are not in a group any more. Can be executed only on groups.
Assign to new or existing group	Assigns the selected nodes to a new or an existing group. Can be executed on the nodes of active application except: **Action**, **Method**, **Property**, **Transition**, **Variable**, **Library**.
Add all my variables and properties	Adds the variables and properties to graph that are defined in the selected node. Can be executed on: **GlobalVariableList**, **Program**, **FunctionBlock**, **Function**, **Method**, **Transition**, **Action**.
Add all my group children	Adds the nodes and groups to graph that are assigned to the selected group.

Menu command	Description
Add my parent group	Adds the group to graph the selected node is assigned to.
Add all my Actions	Adds the actions to graph that are defined in the selected **FunctionBlock** or **Program**.
Add all my Methods	Adds the methods to graph that are defined in the selected **FunctionBlock** or **Program**.
Add all my Transitions	Adds the transitions to graph that are defined in the selected **FunctionBlock** or **Program**.
Add all my Enum Values	Adds the enumeration values that are defined in the selected **Enumeration**.
Add all Interfaces I am implementing	Adds the interfaces to graph that are implemented by the selected **FunctionBlock**.
Add all FunctionBlocks extending me	Adds the function blocks to graph that are extending the selected **FunctionBlock**.
Add all Interfaces extending me	Adds the interfaces to graph that are extending the selected **Interface**.
Add all FunctionBlocks I am extending	Adds the **FunctionBlocks** to graph that the selected **FunctionBlock** is extending.
Add all Interfaces I am extending	Adds the interfaces to graph that the selected **Interface** is extending.
Add all references (variables/properties) instantiating me	Adds the variables and properties that refer to the selected **FunctionBlock** or **Enumeration**.
Add all nodes this XX is calling	Adds the nodes to graph that the selected node is calling. Can be executed on: **Program**, **FunctionBlock**, **Function**, **Method**, **Transition**, **Action**.
Add all nodes this XX is called	Adds the nodes to graph that call the selected node. Can be executed on: **Program**, **FunctionBlock**, **Function**, **Method**, **Transition**, **Action**.
Add all variables or properties I am reading or writing	Adds the variables and properties to graph that the selected node is reading or writing. Can be executed on: **Program**, **FunctionBlock**, **Function**, **Method**, **Transition**, **Action**, **Property**.
All nodes reading or writing me	Adds the nodes to graph that are reading or writing the selected **Property** or **Variable**.
Add my datatype	Adds the data type node to graph of the selected **Property** or **Variable** if data type is no base data type (BOOL, REAL, STRING, and so on).
Add my parent	Cannot be executed on the nodes **Application** and **Library**.
Add my return type	Adds the return type node to graph of the selected **Function** or **Method** if data type is no base data type BOOL, REAL, STRING, and so on).
Add Testmanager executing me	Adds the **Testmanager** the selected **Testcase** is executed by.
Add Testcases I am executing	Add the **Testcases** the selected **Testmanager** is executing.
Add Structs I am utilizing	Adds the structures the selected **TestCase** is using to execute.
Add TestCases I am utilized	Adds the **TestCase** using me as structure to be executed.
Add TestCases and TestResources using me	Adds the **TestCases** and **TestResources** using the selected **TestResource** or **TestCase**.
Add TestSets and TestResources I am using	Adds the **TestCases** and **TestResources** used by the selected **TestResource** or **TestCase**.

Dependency View (Groups)

Overview

Objects displayed in a dependency graph can be assigned to groups. These groups can be real modules of the machine or any other grouping.

The following considerations have to be taken into account:

- Objects to be assigned to a group must be in the same application as the **Dependency View** object.
- Assigning a function block or a program to a group also assigns the methods, actions, properties, and transitions of that object to that group. These elements are displayed in the **Groups** tree as subnodes of the function block/ program.

Toolbar

Element	Description
Create new	Click this button to create a new and empty group. You can start renaming of a group by selecting a group in the tree and click the **Spacebar**.
Delete	Click this button to delete the selected group.
Open Wizard	Click this button to open the **Groups Wizard**.

Assigning Objects to Groups

Assigning objects to groups by:

Element	Description
Drag-and-drop	The nodes and subnodes of an **Application** can be assigned to a group by drag-and-drop. If a folder is dropped, its subnodes are assigned to the group. (You are asked to confirm).
Contextual menu in the **Application** tree	The nodes and subnodes of an **Application** can be assigned to a group by the contextual menu command, page 40 **Add to group**.
Contextual menu in the dependency graph of the **Dependency View**	One or several elements selected in the dependency graph can be assigned to an existing or new group by contextual menu command, page 40 **Add to dependency graph**

Groups Wizard

Use the **Groups Wizard** to create groups and assign objects to groups basing on a name search. The **Groups Wizard** provides options to filter the objects that could be assigned to new or existing groups.

Element	Description
Object name filter	Enter plain text or a regular expression (**Regex**) to filter your application for objects to be added to a group. This filter is not applied to folders. **Regex** examples: • [0-9]: This filters out objects that do not have any digit(s) in their name. • [_]: This filters out objects that do not have any underscore in their name.
Object type filter	Use these check boxes to reduce the number of displayed objects.

Element	Description
	Name search can be reduced to object types: • **Function Block** • **Program**
Display only unassigned objects	Activate the check box to display only objects not assigned to a group. Deactivate the check box to also display the already assigned objects. The check box is activated by default.
Group name	Basing on the object name the suggested module name is generated. This suggested name can be modified by a regular expression (**Regex**) or a meaningful name. In the list, the already existing group names are listed. If nothing is entered, the original project name is used.
Display folder structure	Activate the check box to display the objects in folder structure. The check box is activated by default.
Name column	Use the check boxes in front of the names to select the objects to add to the group. If you activate the check box of an object with subnodes, the check boxes of the subnodes are activated too. The group name of the subnodes is changed to the group name of the main node.
Group Name column	To edit a suggested group name, select the name and press the **Spacebar**.
Create	Click this button to create a group from the selected objects.

Block List

Block List

Overview

A **BlockList** object can be added beneath a **Code Analysis Manager** object. The **BlockList** object contains elements that will not appear in **Conventions Table**, **Metrics Table**, or **Dashboard** results. This, although, is taken into account when you upload a project snapshot, page 31 into Machine Advisor Code Analysis.

To add an element to a block list, right-click the element in the navigator tree and select **Code Analysis > Add to BlockList > Add to Users-BlockList** or **Add to Project-BlockList** from the contextual menu. These contextual menu commands are also available in the result lists of the **Conventions Table** or **Metrics Table**.

The **BlockList** provides two tabs:

- **Conventions**

  List of objects that should not appear in **Conventions Table** and **Dashboard** results.
- **Metrics**

  List of objects that should not appear in **Metrics Table** and **Dashboard** results.

Each tab provides two parts:

- **Project-Blocklist** (left-hand side)

  Contains blocked elements that are available for the users that use the project.
- **Users-Blocklist** (right-hand side)

  Contains blocked elements that should be available only on your PC for this project.

Elements of the Block List

Element	Description
Element	Icon and name of the blocked element.
Comment	Comment on the blocked element. Select the comment cell and press the **Spacebar** to edit or enter a comment.
Path	Path of the blocked element. Double-click the path to open the associated object in its corresponding editor.
Contextual Menu Commands on a table entry	Right-click a table entry and select one of the contextual menu commands: • **Go to definition** Opens the associated element in its corresponding editor. • **Delete from BlockList** Removes the element from the block list.
Double-click a table entry	Opens the associated element in its corresponding editor.

Code Analysis Manager

Dashboard

Overview

The **Dashboard** provides an application overview.

The **Dashboard** tab provides four parts:

- **Toolbar**

  Time stamp of code analysis.

  If the dependency model is out of date, this is indicated in the toolbar, and the analysis can be restarted with **Analyze code**. With the **Check conventions** button you can additionally analyze the conventions.

- **Filter** (left-hand side)

  You can filter the **Dashboard** content based on object types.

  By default, all object types are selected. If you modify the filter, the content of the whole dashboard is updated in real time. The **(<Number>)** behind an object type indicates the total number of objects per type that are included in the code analysis. All object types are included.

- **Metrics** (upper part of the main window)

  Based on the applied filter, the most relevant metrics are displayed.

- **Conventions** (lower part of the main window)

  The numbers of code violations are displayed: total number and number by severity (**Warning**, **Error**, **Info**).

Metrics

Based on the applied filter, the most relevant metrics are displayed.

Element	Description
Lines of Code Total	Total sum of code lines for the objects.
Lines of Code Average	**Lines of Code Total** divided by the number of objects with lines of code.
Halstead Difficulty Max	The maximum value of **Halstead Difficulty**.
Halstead Difficulty Average	The sum of the **Halstead Difficulty** values divided by the number of objects. Displays **n/a** if the applied filter contains no objects with a **Halstead Difficulty** value.

Below the metrics, two bar charts are displayed:

- **Lines of Code Top 5**

  Displays the 5 objects with the highest **Lines of Code** value.

- **Halstead Difficulty Top 5**

  Displays the 5 objects with the highest **Halstead Difficulty** value.

The objects displayed in a bar chart provide additional information through a tooltip.

Conventions

When the code analysis process is finished, the following is displayed:

- A bar chart with the five objects that contain the highest number of code convention violations.

  Each object displayed in the bar chart provides additional information through a tooltip.
- The total number of code violations.
- The total number of code violations with severity **Error**.
- The total number of code violations with severity **Warning**.
- The total number of code violations with severity **Info**.

The content of the top 5 convention violations charts is not the same as on the **Conventions Table**. The **Dashboard** summarizes on POU level, whereas on **Conventions Table** the violations are listed in detail.

Dashboard Information Saved with Project

The **Dashboard** information is saved with your project. The next time you open the project, this information is displayed. The **Analysis as of** label provides a time stamp for this information.

Configuration

Overview

After adding a **Code Analysis Manager** object, a default configuration is applied which supports the most use cases. In special cases, you can modify the depth of code analysis to meet your requirements.

Use the check boxes to define the content to be analyzed.

Element	Description
Consider Implicit Methods	This option is disabled by default. After enabling this option, implicit generated methods like *FB_INIT*, *FB_EXIT*, *FB_REINIT*, and so on, are considered during code analysis. In most cases, these methods are not relevant to explore the source code or to list them in metrics and conventions. Deactivate this option to reduce the amount of analysis data and to improve the performance.
Consider Property Accessor Functions	This option is disabled by default. After enabling of this option, the generated property accessor functions like *GetTextProperty()*, *GetBooleanProperty()*, *GetNumberProperty()*, *GetCompany()*, *GetTextProperty2()*, *GetTitle()*, *GetVersion()*, and *GetVersionProperty()* are considered during code analysis. In most cases, these functions are not relevant to explore the source code or to list them in metrics and conventions. Deactivate this option to reduce the amount of analysis data, to increases the performance, and to improve the usability of **Dependency View** content added through a **Select** query.
Consider Check Functions	This option is disabled by default. After enabling this option, the check functions like *CheckBounds()*, *CheckDiv…()*, *CheckPointer()*, and so on, are considered during code analysis. In most cases, these functions are not relevant to explore the source code or to list them in metrics and conventions.This option reduces the amount of analysis data, increases the performance, and improves the usability of **Dependency View**.

Element	Description
Consider Code Analysis of Libraries in Deep	This option is disabled by default. The more libraries are referenced and used by a project (direct or indirect library references), the more compile output must be analyzed. In general, functions, programs, function blocks, and so on, which are part of the compilation (called, read, written, ...) are part of the compile output independent of the origin (application, library, POU space, …). This requires processor time during code analysis and results in a larger data model. If you enable this option, a medium size project with a long list of referenced libraries can result in multiple GB of RAM usage (8...12 GB) and an increased processor time to analyze the read, write, and call dependencies between the libraries. Also, the query execution times to get the conventions and metrics results can result in execution timeouts. In most cases, the content of a library (not directly used by the application through a call, read, or write), can be skipped during code analysis run. Deactivate this option to reduce the amount of required RAM.
Consider Devices	This option is disabled by default. With this option, all devices are considered during code analysis. The analysis of devices is not relevant to get conventions or metrics. For exploring your own functions, programs, function blocks, and so on, through the **Dependency View**, devices are also not relevant. If it is necessary to explore the connection between these functions, programs, and so on, to the devices and their parameters, it is necessary to consider devices, too. **NOTE:** The additional amount of RAM needed depends on the number of devices in the project. A typical use case where devices are relevant is for drag-and-drop of a device into the **Dependency View**. For example, to navigate to its corresponding function block instance and identify which programs or functions are accessing devices parameters directly.
Consider Code and Data Size (Requires Generate Code)	This option is disabled by default. Generating this data requires code generation and increases the time for creating the data model. If you activate this option, the metrics Application Size (Code), Application Size (Code+Data), and Application Size (Data) deliver results.

Cloud Connection

Overview

This option enables you to upload your analysis model (snapshot) into the Machine Advisor Code Analysis cloud.

Code Analysis Cloud Login

Step	Action
1	Open the **Cloud Connection** tab.
2	Click the **Login** button.
3	Log in with your credentials.

If you do not have an account, follow the instructions in the EcoStruxure Machine Advisor Code Analysis User Guide\Create a Company Account and Invite Users.

Code Analysis Cloud Context

If you have successfully logged in:

Step	Action
1	Click in the **Company** field and select the company where the snapshot is to be uploaded.
2	Click in the **Analysis Project** field and select the project where the snapshot is to be uploaded.

Snapshot Upload

The snapshot is not created by default.

If the snapshot is unavailable or outdated:

Step	Action
1	Click the **Analyze project** button to create a snapshot.
2	Choose to directly **Upload Snapshot** or **Store Snapshot** in the file system by clicking the appropriate button.

The snapshot stored in the file system can be uploaded manually. Refer to EcoStruxure Machine Advisor Code Analysis User Guide\Create a Snapshot and Upload.

Code Analysis Query Manager

Overview

The **Code Analysis Query Manager** provides tabs to create and modify customized rule sets and to manage your metrics and conventions queries.

To open the **Code Analysis Query Manager** click **Tools** in the menu bar and select **Code Analysis Query Manager** from the contextual menu.

Rule Sets

Overview

With the **Rule Sets**, you can provide a subset of queries to use for analyzing the project. You can use pre-defined rule sets as templates for your own rule sets and modify them according to your need. Your new rule sets can be synchronized with Machine Advisor Code Analysis too.

The main window of **Rule Sets** provides two parts:

- List of rule sets
- Queries that are assigned to the several rule sets.

List of Rule Sets

The rule sets list displays the pre-defined rule sets (installed with EcoStruxure Machine Expert), and the user-defined rule sets.

Element	Description
Name	Name of the rule set.
Assignments	Number of assignments.
Sync status	Synchronization status with EcoStruxure Machine Advisor Code Analysis. The synchronization status is only visible if you are connected to Machine Advisor Code Analysis.

Toolbar

Element	Description
New	Click this button to create a user-defined rule set and enter a name for the new rule set.
Duplicate	Click this button to duplicate the selected rule set and enter a name for the new rule set.
Remove	Click this button to remove the selected rule set. You are prompted to confirm. Confirm with **Yes** or cancel with **No**.
Restore	Click this button to restore the selected rule set. You are prompted to confirm. Confirm with **Yes** or cancel with **No**. If you confirm, the customized rule set is restored to the original rule used for customization.
Refresh	Click this button to start the comparison for rule sets between EcoStruxure Machine Expert and Machine Advisor Code Analysis. If EcoStruxure Machine Expert is not logged into Machine Advisor Code Analysis, the **Cloud Connection** tab is opened to enter your user credentials.

Element	Description
Download missing	Click this button to download the rule sets from Machine Advisor Code Analysis that are not available on EcoStruxure Machine Expert. If EcoStruxure Machine Expert is not logged into Machine Advisor Code Analysis, the **Cloud Connection** tab is opened to enter your user credentials.
Download selected	Click this button to download the selected rule sets from Machine Advisor Code Analysis to EcoStruxure Machine Expert. If EcoStruxure Machine Expert is not logged into Machine Advisor Code Analysis, the **Cloud Connection** tab is opened to enter your user credentials.
Upload selected	Click this button to upload and update the selected rule sets to Machine Advisor Code Analysis. If EcoStruxure Machine Expert is not logged into Machine Advisor Code Analysis, the **Cloud Connection** tab is opened to enter your user credentials.

Details of the Selected Rule Set

Element	Description
Name	Name of the rule set.
Description	Detailed description of the rule set.
Assigned Query Chains	The **Assigned Query Chains** tree provides the available **Metrics** and **Conventions** query chains. You can add or remove query chains to the rule set by activating/deactivating the check boxes of the respective query chains. Click **Summary information** (in the upper right corner) to open a text file that provides the assigned query chains of the rule set. You can save this text file.

Queries Repositories

Overview

This generic description is valid for the following repositories:

- **Metrics**
- **Conventions**
- **Dependency (Filter) Queries**
- **Dependency (Select) Queries**

For details, refer to the respective chapters.

You can organize and create queries and query chains for all repository categories. A query chain consists of one or more queries.

Each repository provides a list of pre-defined queries and query chains that can be used as templates for user-defined queries/query chains. The pre-defined queries and query chains can also be modified directly.

The main window of a repository provides three parts:

- **Query chains** (left-hand side)

  List of available query chains. For metrics and conventions the synchronization status with Machine Advisor Code Analysis is provided.

- **Queries** (right-hand side)

  List of available queries.
- **Results** (bottom)

  Results of the selected and executed query chain.

The rightmost repository editor provides the **Parameters Editor**.

Query Chains

The **Query Chains** tree contains query chains that are available by default (by EcoStruxure Machine Expert installation) and the user-defined query chains created with the **Query Chain Settings Editor**.

The predefined query chains provided by Schneider Electric can be modified and also be reset to their initial values.

Toolbar

Element	Description
Execute	Click this button to execute the selected query chain. The result is displayed in the **Results** window at the bottom.
Edit	Click this button to edit the selected query chain. You can edit only the user-defined query chains. **Edit** opens the **Query Chain Editor**, page 38.
Duplicate	Click this button to duplicate the selected query chain and enter a name for the new query chain. Double-click the duplicated query chain to open the **Query Chain Settings Editor**.
New	Click this button to add a new query chain. **New** opens the **Query Chain Settings Editor**
Remove	Click this button to remove the selected query chain. You can remove only the user-defined query chains.
Restore	Click this button to restore the selected rule set. You are prompted to confirm. Confirm with **Yes** or cancel with **No**. If you confirm, the customized rule set is restored to the original rule used for customization.
Refresh	Click this button to start the comparison for rule sets between EcoStruxure Machine Expert and Machine Advisor Code Analysis. If EcoStruxure Machine Expert is not logged into Machine Advisor Code Analysis, the **Cloud Connection** tab is opened to enter your user credentials.
Download missing	Click this button to download the rule sets from Machine Advisor Code Analysis that are not available on EcoStruxure Machine Expert. If EcoStruxure Machine Expert is not logged into Machine Advisor Code Analysis, the **Cloud Connection** tab is opened to enter your user credentials.
Download selected	Click this button to download the selected rule sets from Machine Advisor Code Analysis to EcoStruxure Machine Expert. If EcoStruxure Machine Expert is not logged into Machine Advisor Code Analysis, the **Cloud Connection** tab is opened to enter your user credentials.
Upload selected	Click this button to upload and update the selected rule sets to Machine Advisor Code Analysis. If EcoStruxure Machine Expert is not logged into Machine Advisor Code Analysis, the **Cloud Connection** tab is opened to enter your user credentials.

Assign and Order Queries to Chains

Element	Description
Arrow buttons	Use the **Arrow** buttons between the **Query Chains** and the **Queries** to add a query to or to remove a query from a selected query chain. You can also use these buttons to modify the order of queries in a selected query chain. You can edit only the user-defined query chains.
Drag-and-drop	You can also use drag-and-drop to add a query to or to remove a query from a user-defined query chain.

Queries

The queries list shows queries available by default (installed with EcoStruxure Machine Expert), and the user-defined queries.

You cannot remove or edit the predefined queries. You can only duplicate these queries.

Toolbar

Element	Description
Edit	Click this button to edit the selected query. You can edit only the user-defined queries. **Edit** opens the **Query Editor**, page 36.
Duplicate	Click this button to duplicate the selected query and enter a name for the new query. Double-click the duplicated query to open the **Query Editor**.
New	Click this button to add a new query. **New** opens the **Query Editor**.
Remove	Click this button to remove the selected query. You can remove only the user-defined queries.
Restore	Click this button to restore the selected query. You are prompted to confirm. Confirm with **Yes** or cancel with **No**. If you confirm, the customized query is restored to the original query.

Parameters Editor

The **Parameters Editor** displays the parameters for the selected query / query chain.

Refer to Parameters Editor, page 38.

Query Editor

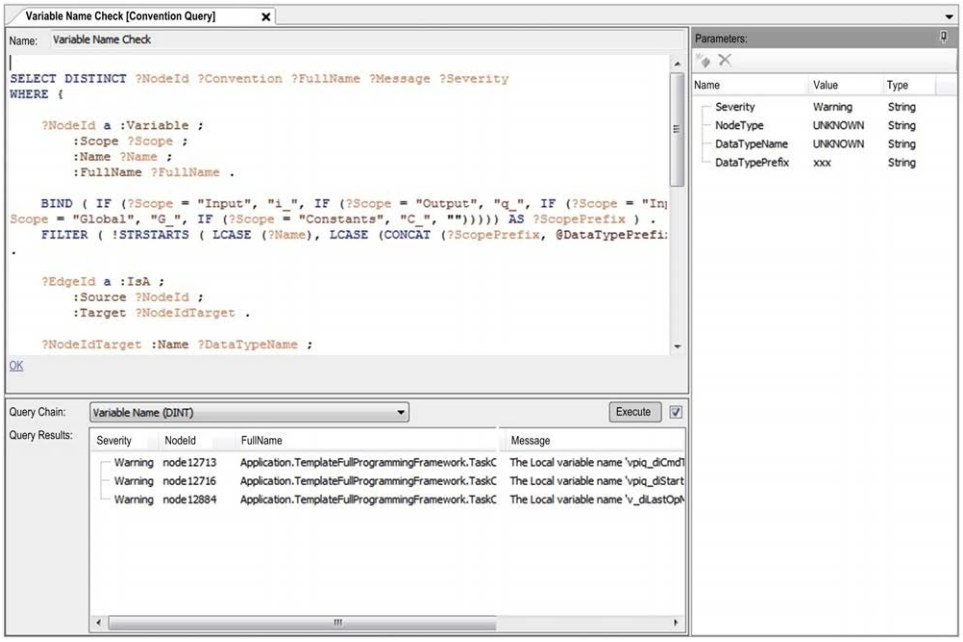
Overview

With the **Query Editor**, you can edit or create user-defined queries.

The main window of the **Query Editor** provides two parts:

- SPARQL Editor
- **Query Results**

Rightmost the **Query Editor** provides the **Parameters Editor**.



SPARQL Editor

Element	Description
Name	Edit the query name.
SPARQL Editor	Edit the SPARQL query.
Syntax validation messages	At the bottom of the query editor, detected SPARQL syntax errors are displayed. Click this message to jump to the detected syntax error in the query editor.

Query Results

Element	Description
Query Chain	Queries can only be executed in the environment of a query chain. If a query is already assigned to query chains, these query chains are available in this list. There is also a default query chain without parameters available.
Query Results	The results of a query are displayed in this **Query Results** table.
Execute check box	Deselect this check box to disabled automatic query execution.
Execute button	Click this button to start execution.

Parameters Editor

The **Parameters Editor** displays the parameters for the query.

Refer to Parameters Editor, page 38.

Query Chain Settings Editor

Overview

With the **Query Chain Settings Editor**, you can configure details for user-defined query chains.

The availability of query chain settings depends on the repository this query belongs to.

Element	Description	Available for
Description	Description of the query chain.	All query tabs.
Group	Defines the group name to display the queries in a grouped way.	• **Dependency (Select) Queries** This grouping is included in the **Select and Add** dialog box of the **Dependency view** editor. • **Convention Queries** This grouping is included in the **Conventions** tree of the **Conventions Table** editor.

Color Coding

Color coding is available for **Metric Queries**.

For the **Metrics Table**, you can highlight dedicated metric results with specific colors.

The color coding rules are based on four parameters:

Parameter	Description
Value	Enter a value the real metric result value is compared with.
Comparer	Select a value for how the metrics result value is compared. • GT (>) • LT (<) • GTE (≥) • LTE (≤) • EQ (=) • NEQ (≠)
Comparison type	Enter a fix value or a percentage value. (100% is the highest value of this metric query).
Color	Enter an hexadecimal value for the color the metric result is displayed if the rule is met.

Parameters Editor

Overview

With the **Parameters Editor**,

- You can display the parameters for the selected query / query chain.
- You can add and remove parameters.
- You can edit the **Name**, **Value**, and **Type** of parameters that belong to user-defined queries / query chains.

Element	Description
Name	Variable name
Value	Default value
Type	Variable types available: • **Boolean** • **Int32** • **Double** • **String**
Add	Click this button to add a parameter.
Remove	Click this button to remove the selected parameter.

Cloud Connection

Overview

This option enables you to upload your analysis model (snapshot) into the Machine Advisor Code Analysis Web app.

Code Analysis Cloud Login

Step	Action
1	Open the **Cloud Connection** tab.
2	Click the **Login** button.

If you do not have an account, follow the instructions in the EcoStruxure Machine Advisor Code Analysis User Guide\Create a Company Account and Invite Users.

Code Analysis Cloud Context

If you have successfully logged in, click in the **Company** field and select the company where the snapshot is to be uploaded.

Query Synchronization

To verify if the queries on your PC and on Machine Advisor Code Analysis are the same, click the **Compare Queries** button.

If not, you can click the **Upload All Queries** button to upload your queries to Machine Advisor Code Analysis.

Contextual Menu Commands

Contextual Menu Commands of Navigators

Overview

On all objects beneath the **Application** object the following contextual menu commands are available for code analysis:

- **Clean all**
- **Add to group**
- **Add to dependency graph**

Clean All

Use this contextual menu command to remove analysis data belonging to the application the selected object is part of.

Add to Group

Use this contextual menu command to add the selected object to an existing or new group.

Add to Dependency Graph

Use this contextual menu command to add the selected object to an existing or new dependency graph.

After the code analysis procedure, the new or the already existing **Dependency View** is opened and the selected objects are added to the dependency graph in the upper left corner.

You can also add variables to a dependency graph. In your program code, right-click a variable and select **Code Analysis > Add to dependency graph** from the contextual menu.

Pragma Instructions for Code Analysis

Pragma Instructions for Code Analysis

Overview

With Pragma instructions, it is possible to decide if the source code should be taken into account in code analysis.

In contrast to this, block lists are used to filter out elements after analysis run. The effect is the same.

Using Pragma instructions you can:

- Mark source code (for example, functions, programs, function blocks, variables, and so on) to be ignored by code analysis (use case 1).
- Mark source code to filter results displayed in **Conventions Table** results, **Metrics Table** results, or **Dependency View** only (use case 2).

  Also refer to chapter Block List, page 27.

Use Case 1

Code that should not be analyzed in general can be marked with the `ignore` Pragma instruction. Elements marked this way are not part of the analysis data model.

Attribute	Description
`{attribute 'code_analyzer' := 'ignore'}`	• Marks a POU or a variable to be ignored. • Works in recursive mode (for example, the variables contained in the POU are also ignored, or the methods below a function block are ignored). • Highest priority (compared to `'public_only'` attribute).
`attribute 'code_analyzer' := 'public_only'}`	• Marks a POU to consider only the public elements (VarIn, VarOut, VarInOut, public properties, public methods). Other elements are ignored. • Works in recursive mode (for example, methods below a function block are handled in the same way as the function block).

Use Case 2

Code that cannot be changed, but appears in convention and/or metric results, can be marked to be hidden in these result lists.

Attribute	Description
`{attribute 'code_analysis_ui' := 'filter_in_results'}`	• Marks a POU or variable to be filtered in results (**Conventions Table**, **Metrics Table**, and **Dependency View**). • Works non-recursive.
`{attribute 'code_analysis_ui' := 'filter_in_conventions'}`	• Marks a POU or variable to be filtered in **Conventions Table** results (through the UI or scripting API). • Works non-recursive.

Attribute	Description
`{attribute 'code_analysis_ui' := 'filter_in_metrics'}`	• Marks a POU or variable to be filtered in **Metrics Table** results (through the UI or scripting API). • Works non-recursive.
`{attribute 'code_analysis_ui' := 'filter_in_dependency_view'}`	• Marks a POU or variable to be filtered in **Dependency View** (**Select and Add** dialog box). • Works non-recursive.

Examples

Ignore a public variable

```
METHOD PUBLIC PublicMethod
VAR_INPUT
    {attribute 'code_analyzer' := 'ignore'}
    i_iPublicButIgnoredVar: INT;

    i_iPublicVar2: INT;
END_VAR
VAR
    iPrivateVar2: INT;
END_VAR
```

Consider only public elements

```
{attribute 'code_analyzer' := 'public_only'}
FUNCTION_BLOCK FB_PublicOnlyTest1
VAR_INPUT
    i_iPublicVar: INT;
END_VAR
VAR
    iPrivateVar: INT;
END_VAR
```

Filter (POU) in results

```
{attribute 'code_analysis_ui' := 'filter_in_results'}
PROGRAM SR_FilterInResultsTest1
VAR
END_VAR
```

Filter (variable) in results of metrics and/or conventions

```
PROGRAM SR_FilterVars
VAR
    {attribute 'code_analysis_ui' := 'filter_in_metrics'}
    iTestVar1: INT;
    {attribute 'code_analysis_ui' := 'filter_in_results'}
    iTestVar2: INT;
    {attribute 'code_analysis_ui' := 'filter_in_
conventions'}
    FAILED_iTestVar3: INT;
END_VAR
```

Python Script Interface

Scripting Interface

Overview

A scripting API is available to automate code analysis. This allows you to integrate the code analysis mechanism into ALM (Application Lifecycle Management) / CI (Continuous Integration) environments.

Also refer to the following chapters in the online help:

- EcoStruxure Machine Expert Programming Guide\Appendices\Python Script Language (see Programming Guide)
- Script Engine Plugin API Reference
- Script Engine Class Library

Scripting Object Extensions

Application

The **Application** object is extended by the following property:

Property	Returned Object / Value	Description
`code_analysis`	ScriptCodeAnalysis with attached functions / properties	The returned object provides access to the code analysis features like `perform_full_analysis()`. Refer to Python Scripting Objects, page 44.

Conventions Table

The **Conventions Table** object is extended by the following function:

Function	Returned Object / Value	Description
`conventions_table()`	`ConventionsTableResults` with attached functions / properties.	By calling this method, the convention table results are built by analyzing the code. The returned object provides access to the convention table results. The configured convention of this convention table object is used. Refer to Python Scripting Objects, page 44.

Metrics Table

The **Metrics Table** object is extended by the following function:

Function	Returned Object / Value	Description
`metrics_table()`	`MetricsTableResults` with attached functions / properties.	By calling this method, the metrics table results are built by analyzing the code. The returned object provides access to the metrics table results. The configured metrics of this metrics table object is used. Refer to Python Scripting Objects, page 44.

Scripting Objects (Code Analysis API)

Overview

Using the Scripting Object Extensions, page 43, the returned scripting objects can be used, for example, to trigger a code analysis or to access the conventions or metrics results.

Code Analysis API

This object provides general access to code analysis of an **Application** object.

Property	Returned Object / Value	Description
`metrics`	`MetricsTable` with attached functions / properties	The `metrics` property provides access to the metrics-specific scripting API, page 45.
`conventions`	`ConventionsTable` with attached functions / properties	The `conventions` property provides access to the conventions-specific scripting API, page 45.
`configuration`	`Configuration` with attached functions / properties	The `configuration` property provides access to the conventions-specific scripting API, page 45.
`cloud_connection`	`CloudConnection` with attached functions / properties	The `cloud_connection` property provides access to the upload of snapshots to Triple Storages, and to configuration of the cloud connection.

Functions	Returned Object / Value	Description
`clear()`	None	Removes the dependency model from memory. Another code analysis run can be started.
`perform_full_analysis()`	None	Starts a full code analysis run. The dependency model is built and reused for example by metrics API.
`store_dependency_model(filename)`	None	Stores the dependency model into an XML file.
`store_ttl (filename)`	None	Stores the RDF model in TTL format to disk. TTL (Turtle Language) is a standard format of https://www.w3.org to store RDF models.

Metrics API

Functions	Returned Object / Value	Description
`get_all_metrics()`	`string[]` with available metrics names	Gets the available metrics query names.
`full_metrics_table()`	`MetricsTableResult` with attached functions / properties	Builds the full metrics table of the available metrics queries.Refer to Metrics Table Result API, page 45.
`metrics_table()`	`MetricsTableResult` with attached functions / properties.	Builds the metrics table with specified metrics queries. Refer to Metrics Table Result API, page 45.

Metrics Table Result API

Properties	Returned Object / Value	Description
`successful`	BOOL	Gets the result outcome.
`message`	STRING	Gets the readable outcome message.
`project_path`	STRING	Gets the project path where this results are based on.
`analysis_started_at`	STRING	Gets the start time this result table was built.
`analysis_finished_at`	STRING	Gets the finished time this results table was built.
`analyser_version`	STRING	Gets the code analyzer version used to build this result table.
`columns`	STRING	Gets the list of columns in this result table.
`rows`	STRING	Gets the table (rows with columns: [Array] of [Array]) with the result values.

Functions	Returned Object / Value	Description
`store_as_xml(…)`	None	Stores the metrics table results as an XML file to disk.
`store_as_csv(…)`	None	Stores the metrics table results as a CSV file to disk.
`store_as_html(…)`	None	Stores the metrics table results as an HTML to disk. Provides to specify an XSLT file for XML to HTML transformation, if required.

Conventions API

Functions	Returned Object / Value	Description
`get_all_conventions()`	STRING[] with available convention names	Gets the available convention query names.
`full_conventions_table()`	`ConventionTableResult` with attached functions / properties	Builds the full conventions table of the available convention queries. Refer to Conventions Table Result API, page 46.
`conventions_table()`	`ConventionTableResult` with attached functions / properties.	Builds the conventions table with specified convention queries. Refer to Conventions Table Result API, page 46.

Conventions Table Result API

Properties	Returned Object / Value	Description
`successful`	BOOL	Gets the result outcome.
`message`	STRING	Gets the readable outcome message.
`project_path`	STRING	Gets the project path where this results are based on.
`analysis_started_at`	STRING	Gets the start time this result table was built.
`analysis_finished_at`	STRING	Gets the finished time this results table was built.
`analyser_version`	STRING	Gets the code analyzer version used to build this result table.
`columns`	STRING	Gets the list of columns in this result table.
`rows`	STRING	Gets the table (rows with columns: [Array] of [Array]) with the result values.

Functions	Returned Object / Value	Description
`store_as_xml(…)`	None	Stores the conventions table results as an XML file to disk.
`store_as_csv(…)`	None	Stores the conventions table results as a CSV file to disk.
`store_as_html(…)`	None	Stores the conventions table results as an HTML to disk. Provides to specify an XSLT file for XML to HTML transformation, if required.

Configuration API

Property	Returned Object / Value	Description
`triple_storage_backend_type`	Enumeration `TripleStorageBackend-Types`	Configures the RDF Triple Storage backend type to be used to handle code analysis data.
`thresshold_for_out_proc_backend_usage`	INT	Configures the threshold for auto-selection of used storage backend type (In-Memory or Out-Proc).
`max_upload_triple_count_per_request`	INT	Configures the number of triples per upload request.
`query_execution_timeout`	Long	Configures the query execution timeout.
`update_execution_timeout`	Long	Configures the update execution timeout for a query.
`server_uri`	STRING	Configures the server URI for http-based storage backends.
`relative_query_endpoint`	STRING	Configures the query endpoint for http-based storage backends (part of complete query endpoint url).
`relative_update_endpoint`	STRING	Configures the update endpoint for http-based storage backends (part of complete update endpoint url).
`dataset`	STRING	Configures the dataset name for http-based storage backends (part of the endpoint url).

Property	Returned Object / Value	Description
`relative_data_endpoint`	STRING	Configures the data endpoint name for http based storage backends (part of the endpoint url).
`relative_sparql_endpoint`	STRING	Configures the SPARQL endpoint name for http based storage backends (part of the endpoint url).
`graph_name`	STRING	Configures the graph name for http based storage backends (part of the endpoint url).
`sparql_endpoint`	STRING	Read-only. Gets access to the complete SPARQL endpoint url.
`data_endpoint`	STRING	Read-only. Gets access to the complete data endpoint url.
`query_endpoint`	STRING	Read-only. Gets access to the complete query endpoint url.
`update_endpoint`	STRING	Read-only. Gets access to the complete update endpoint url.

Functions	Returned Object / Value	Description
`reset()`	None	Resets the code analysis configuration.

Cloud Connection API

Property	Returned Object / Value	Description
`configuration`	`CloudConfiguration` with attached functions / properties.	The cloud configuration property provides access to cloud configuration-specific scripting API.

Functions	Returned Object / Value	Description
`upload_to_triple_storage(…)`	None	Starts a snapshot upload of the RDF model to the configured RDF Triple Storage.

Cloud Configuration API

Property	Returned Object / Value	Description
`http_backend_type`	Enumeration `HttpBackendTypes`	Configures the http backend type (for example, generic, Apache Fuseki, Stardog, and so on). This configuration value is only considered if Triple Storage backend type is set to "Http".
`max_upload_triple_count_per_request`	Integer	Configures the number of triples per upload request.
`update_execution_timeout`	Long	Configures the update execution timeout for a query.
`server_uri`	String	Configures the server URI for http based storage backends.
`relative_query_endpoint`	String	Configures the query endpoint for http based storage backends (part of complete query endpoint URI).
`relative_update_endpoint`	String	Configures the update endpoint for http based storage backends (part of complete update endpoint URI).

Property	Returned Object / Value	Description
`dataset`	String	Configures the dataset name for http based storage backends (part of the endpoint URI).
`relative_data_endpoint`	String	Configures the data endpoint name for http based storage backends (part of the endpoint URI)
`relative_sparql_endpoint`	String	Configures the SPARQL endpoint name for http based storage backends (part of the endpoint URI).
`sparql_endpoint`	String	Read-only. Gets access to the complete SPARQL endpoint URI.
`data_endpoint`	String	Read-Only. Gets access to the complete data endpoint URI.
`query_endpoint`	String	Read-Only. Gets access to the complete query endpoint URI.
`update_endpoint`	String	Read-Only. Gets access to the complete update endpoint URI.

Functions	Returned Object / Value	Description
`reset()`	None	Resets the code analysis cloud configuration.

How to Add Code Analysis Editors

How to Get a Quick Application Overview through the Dashboard

Overview

The **Dashboard** provides an application overview.

Refer to Dashboard, page 29.

Open or Create a Project

Open your preferred project or create a new project with **File > New Project > From Project Template > Template Full**.

Add a Code Analysis Manager

The **Dashboard** tab is provided by the **Code Analysis Manager**. So first you have to add a **Code Analysis Manager** to your project.

Step	Action	Result / Comment
1	Right-click the **Application** node in the **Tools** tree (or in another navigator view like the **Devices** tree).	-
2	Select **Add Object > Code Analysis Manager** from the contextual menu.	The **Add Code Analysis Manager** editor is displayed.
3	Click the **Add** button.	• A **Code Analysis Manager** object is added beneath the **Application** object in the **Tools** tree. • The **Dashboard** tab of the **Code Analysis Manager** is displayed.

Analyze Code

Step	Action	Result / Comment
1	Click the **Analyze code** button in the **Dashboard** tab.	The project is analyzed and the **Metrics** are displayed. If you modify the filter on the left-hand side, the content of the dashboard is updated in real time.

Summary

Element	Description
Lines of Code Total	Total sum of code lines for the objects.
Lines of Code Average	**Lines of Code Total** divided by the number of objects with lines of code.

Element	Description
Halstead Difficulty Max	The maximum value of **Halstead Difficulty**.
Halstead Difficulty Average	The sum of the **Halstead Difficulty** values divided by the number of objects. Displays **n/a** if the applied filter contains no objects with a **Halstead Difficulty** value.

Below the metrics, two bar charts are displayed:

- **Lines of Code Top 5**

  Displays the 5 objects with the highest **Lines of Code** value.
- **Halstead Difficulty Top 5**

  Displays the 5 objects with the highest **Halstead Difficulty** value.

The objects displayed in a bar chart provide additional information through a tooltip.

Add Conventions

Step	Action	Result / Comment
1	To add further results, click the **Add conventions** button the **Dashboard** tab.	The project is analyzed and the **Convention** results are displayed.

How to Get Detailed Metric Results of Your Application

Overview

With the **Metrics Table**, you can select metrics to be executed on an application and filter and list results.

Refer to Metrics Table, page 19.

Open or Create a Project

Open your preferred project or create a new project with **File > New Project > From Project Template > Template Full**.

Add a Metrics Table

Step	Action	Result / Comment
1	Right-click the **Application** object in the **Tools** tree (or in another navigator view like the **Devices** tree).	-
2	Select **Add Object > Metrics Table** from the contextual menu.	The **Add Code Analysis Metrics Table** editor is displayed.
3	Click the **Add** button.	• A **Code Analysis Metrics Table** object is added beneath the **Application** object in the **Tools** tree. • The **Metrics Table** is displayed.

Analyze Code

Step	Action	Result / Comment
1	In the **Name** column, deactivate the check box **All**.	If the **All** check box is activated, analysis takes more time.
2	Activate the check boxes for **Halstead Difficulty** and **Lines Of Code**.	-
3	Click the **Analyze code** button.	The project is analyzed and the results are displayed.

How to Use the Results

Now you can, for example, sort the **Halstead Difficulty** values by clicking the column header; and then review your project focusing on the highest values for **Halstead Difficulty**.

Double-clicking a table entry opens the associated application object in its corresponding editor. You can also use the **Go to definition** contextual menu command of a table entry.

For objects that should not appear in the **Metrics Table** results you can use the **Add to BlockList** contextual menu command of a table entry.

How to Get Detailed Convention Results of Your Application

Overview

With the **Conventions Table**, you can select conventions that have to be met by your application.

Refer to Conventions Table, page 18.

Open or Create a Project

Open your preferred project or create a new project with **File > New Project > From Project Template > Template Full**.

Add a Conventions Table

Step	Action	Result / Comment
1	Right-click the **Application** object in the **Tools** tree (or in another navigator view like the **Devices** tree).	-
2	Select **Add Object > Conventions Table** from the contextual menu.	The **Add Code Analysis Conventions Table** editor is displayed.
3	Click the **Add** button.	• A **Code Analysis Conventions Table** object is added beneath the **Application** object in the **Tools** tree. • The **Conventions Table** is displayed.

Analyze Code

Step	Action	Result / Comment
1	In the **Name** column, deactivate the check box **All**.	If the **All** check box is activated, analysis takes more time.
2	Activate the check box for **Variable Name Checks (Complex Types)**.	-
3	Click the **Analyze code** button.	The project is analyzed and the results are displayed.

How to Use the Results

Now you can, for example, activate the check box for **Variable Name (Array)** to list only the respective results. Then you can review your project focusing on this convention results.

Double-clicking a table entry opens the associated application object in its corresponding editor. You can also use the **Go to definition** contextual menu command of a table entry.

For objects that should not appear in the **Conventions Table** results you can use the **Add to BlockList** contextual menu command of a table entry.

How to Display Dependencies of Your Application with Help of Predefined Queries on Dependency View

Overview

With the **Dependency View**, you can visualize the dependencies of your analyzed application as a dependency graph. You can select the content and the layout of the dependency graph.

Refer to Dependency View, page 21.

Open or Create a Project

Open your preferred project or create a new project with **File > New Project > From Project Template > Template Full**.

Add a Dependency View

Step	Action	Result / Comment
1	Right-click the **Application** object in the **Tools** tree (or in another navigator view like the **Devices** tree).	-
2	Select **Add Object > Dependency View** from the contextual menu.	The **Add Code Analysis Dependency View** editor is displayed.
3	Click the **Add** button.	• A **Code Analysis Dependency View** object is added beneath the **Application** object in the **Tools** tree. • The **Dependency View** is displayed.

Analyze Code

Step	Action	Result / Comment
1	Click the **Analyze code** button.	The project is analyzed.

Select Scope and Query

Step	Action	Result / Comment
1	Click the **Select and Add** button.	-
2	**Select Scope > Project**.	This reduces the number elements that are added to the dependency graph to nodes that are defined in your project only.
3	**Select Query > Call Graph**.	This adds the calls of your application to the dependency graph.
4	Click the **Apply** button.	The project is analyzed and the dependency graph is displayed.

How to Explore Stepwise the Dependencies of Your Application on Dependency View

Overview

You can also add nodes and subnodes to the dependency graph by drag-and-drop.

Step	Action	Result / Comment
1	Add a new dependency view to the **Application** node.	See above.
2	In the **Application** tree, select the **SR_ MainMachine (PRG)** node.	**Application > TemplateFullProgrammingFramework > TaskCalls**
3	Drag the node to the new dependency graph and drop it.	-
4	Right-click **SR_MainMachine** in the dependency graph and select **Add all my > [Properties and Variables/Methods/ Actions]** from the contextual menu.	The **SR_MainMachine** and the associated selections are displayed in an unstructured way.
5	Open the **Layout** list and select **LinLog** and click **Apply**.	The **SR_MainMachine** and the associated selections are displayed in a structured way.

NOTE: This is an example. You can drag-and-drop all objects used in your application to the dependency graph.

Appendices

What's in This Part

Dependency (Filter) Queries

What's in This Chapter

Dependency (Filter) Queries

Dependency (Filter) Queries

The following queries are available by default (by EcoStruxure Machine Expert installation).

Name	Description
Call Graph	Description: This query chain is used to get the call edges and the connected nodes (source or target of the edge). The result is applied to the current content of a dependency view to show or hide nodes and edges. Use Case: After generating a dependency view with content of the Call Graph query, this filter helps to ensure that only the called nodes and call edges are displayed.
Extend Graph	Description: This query chain is used to get the extend edges and the connected nodes (source or target of the edge). The result is applied to the content of the dependency view to show or hide nodes and edges. Use Case: After generating a dependency view with content of query Extend Graph query, this filter helps to ensure that only the nodes extended or extending other nodes and extend edges are displayed.
GVL Graph	Description: This query chain is used to get the GVL nodes. The result is applied to the content of the dependency view to show or hide nodes. Use Case: After generating a dependency view with content of the GVL Graph query, this filter helps to ensure that only the GVL nodes are displayed.
GVL+Variable Graph	Description: This query chain is used to get the GVLs and variable nodes and the edges connecting these nodes (source or target of the edge). The result is applied to the content of the dependency view to show or hide nodes and edges. Use Case: After generating a dependency view with content of the GVL +Variable Graph query, this filter helps to ensure that only the GVL and variable nodes and edges connecting them are displayed.
Implement Graph	Description: This query chain is used to get the implement edges and the connected nodes (source or target of the edge). The result is applied to the content of the dependency view to show or hide nodes and edges. Use Case: After generating a dependency view with content of the Implement Graph query, this filter helps to ensure that only the nodes implemented or implementing other nodes and extend edges are displayed.
Implement+Extend Graph	Description: This query chain is used to get the extend and implement edges and the connected nodes (source or target of the edge). The result is applied to the content of the dependency view to show or hide nodes and edges. Use Case: After generating a dependency view with content of the Implement+Extend Graph query, this filter helps to ensure that only the nodes extended or extending and implemented or implementing other nodes and extend and implement edges are displayed.
Library Stack	Description: This query chain is used to get the library nodes. The result is applied to the content of the dependency view to show or hide nodes and edges. Use Case: After generating a dependency view with content of the Library Stack query, this filter helps to ensure that only the library nodes are displayed and the uses edges connecting them together.

Name	Description
Node Graph	Description: This query chain is used to get the nodes (all types). The result is applied to the content of the dependency view to show or hide nodes and edges. Use Case: After generating a dependency view with content of the Node Graph query, this filter helps to ensure that all node types are displayed and the edges connecting them together.
POU Graph	Description: This query chain is used to get the POU nodes. The result is applied to the content of the dependency view to show or hide nodes and edges. Use Case: After generating a dependency view with content of the POU Graph query, this filter helps to ensure that only the POU nodes are displayed and the edges connecting them together.
POU+Variable Graph	Description: This query chain is used to get the POUs and variable nodes and the edges connecting these nodes (source or target of the edge). The result is applied to the content of the dependency view to show or hide nodes and edges. Use Case: After generating a dependency view with content of the POU +Variable Graph query, this filter helps to ensure that only the POU and Variable nodes and edges connecting them are displayed.
Read Graph	Description: This query chain is used to get the POUs and variable nodes and the edges connecting these nodes through a read edge (source or target of the edge). The result is applied to the content of the dependency view to show or hide nodes and edges. Use Case: After generating a dependency view with content of the Read Graph query, this filter helps to ensure that only the POU and variable nodes are displayed connected through read edges.
Read+Write Graph	Description: This query chain is used to get the POUs and variable nodes and the edges connecting these nodes through a read or write edge (source or target of the edge). The result is applied to the content of the dependency view to show or hide nodes and edges. Use Case: After generating a dependency view with content of the Read +Write Graph query, this filter helps to ensure that only the POU and variable nodes are displayed connected through read or write edges.
Write Graph	Description: This query chain is used to get the POUs and variable nodes and the edges connecting these nodes through a write edge (source or target of the edge). The result is applied to the content of the dependency view to show or hide nodes and edges. Use Case: After generating a dependency view with content of the Write Graph query, this filter helps to ensure that only the POU and variable nodes are displayed connected through write edges.
Device Graph	Description: This query chain is used to get the devices and the edges connecting these nodes. Use Case: After generating a dependency view content of **Select and Add** query, page 58 **Device Graph**, this filter helps to ensure that only the devices are displayed.
Test Element Graph	Description: This query chain is used to get the test elements (**TestCase**, **TestResource**, **TestSet**, and so on) and the edges connecting these nodes. Use Case: After generating a dependency view content of **Select and Add** query, page 58 **Test Element Graph**, this filter helps to ensure that only the test elements are displayed.

Dependency (Select) Queries

What's in This Chapter

Dependency (Select) Queries

Dependency (Select) Queries

The following queries are available by default (by EcoStruxure Machine Expert installation).

Group: Misc

Name	Description
Call Graph	Description: This query chain generates a dependency view with the POUs acting as callers or are called by another POU of the selected scope (for example, defined below the **Application**). Results: If only one task is defined and only one program (PRG) is executed, the resulting graph is a tree of POUs connected through call edges. If multiple tasks are defined or multiple programs (PRGs) are executed, it depends on the implementation whether a common POU is called by both or not. Then the result shows multiple call trees in one dependency view (as a graph). Use Case: The dependency view can be used to analyze the call tree in a graphical way and identify which POUs are used by which POU and which POUs are used more than once. If a POU is called more than once, a modification in this POU is automatically established in both call trees. Source code example: `fbMyFunctionBlock.TestMethod();`
Extend (FunctionBlock)	Description: This query chain generates a dependency view with the function blocks (FBs) of the selected scope (for example, defined below the **Application**) that extend another FB. As an example, a basic function block is called FB_DriveBase. This FB_DriveBase can be extended by FB_LXM52 and FB_LXM62. Results: The result is a graph of function blocks connected through extend edges. From a subnode function block (FB not extended by another FB like FB_LMX62), you get the chain of FBs to its basic FBs. For the FBs in a project, you see the FB siblings using the same basic FB. Use Case: The dependency view can be used to visualize the extend chain for FBs. You can see if there are siblings extending the same basic FB with a similar functionality and could be replaced by another. For example, you can use FB_LXM62 and FB_LXM52 as a method input argument based on FB_DriveBase. This method can handle both type of drives. Source code example: `FUNCTION_BLOCK FB_MyTest EXTENDS FB_MyTestBase`

Name	Description
Extend (Interface)	Description: This query chain generates a dependency view with the interfaces of the selected scope (for example, defined below the **Application**) that extend another interface. As an example, a basic interface is called IF_DriveBase. This IF_DriveBase can be extended by IF_LXM52 and IF_LXM62. Results: The result is a graph of interfaces connected through extend edges. From a subnode interface (interface not extended by another interface like IF_LMX62), you get the chain of interfaces to its basic Interface. For the interfaces in a project, you see the interface siblings using the same basic interface. Use Case: The dependency view can be used to visualize the extend chain for interfaces. You can see if there are siblings extending the same basic interface with a similar functionality and could be replaced by another. For example, you can use FB instances implementing IF_LXM62 or IF_LXM52 as a method input argument based on IF_DriveBase. This method can handle both type of drives. Source code example: `INTERFACE IF_MyTest EXTENDS IF_MyTestBase`

Extend Graph	Description: This query chain generates a dependency view with the interfaces and function blocks (FBs) of the selected scope (for example, defined below the **Application**) that extend another Interface.
	
	As an example, a basic interface is called IF_DriveBase. This IF_DriveBase can be extended by IF_LXM52 and IF_LXM62 (same for FBs).
	
	Results: The result is a graph of Interfaces and FBs connected through extend edges.
	
	From a subnode interface / FB (interface / FB not extended by another interface / FB like IF_ LMX62), you get the chain of interfaces / FBs to its basic interface / FB.
	
	For the interfaces / FBs in a project, you see the interface / FB siblings using the same basic interface / FB.
	
	Use Case: The dependency view can be used to visualize the extend chain for interfaces and FBs together. You can see if there are siblings extending the same basic interface / FB with a similar functionality and could be replaced by another. For example, you can use FB instances implementing IF_LXM62 or IF_LXM52 as a method input argument based on IF_DriveBase. This method can handle both type of drives.
	
	Source code example:
	`FUNCTION_BLOCK FB_MyTest EXTENDS FB_MyTestBase`
	
	or
	`INTERFACE IF_MyTest EXTENDS IF_MyTestBase`

GVL Graph	Description: This query chain generates a dependency view with the defined Global Variable Lists (GVLs) of the selected scope (for example, defined below the **Application**).
	
	Results: The result is a graph of nodes (GVLs) without edges between each other.
	
	Use Case: This dependency view can be used to explore the GVLs. You can add for example, the variables connected to the GVL you are interested in.
	
	Source code example:
	`VAR_GLOBAL`
	`  G_iGVLTestVar: INT;`
	`END_VAR`

GVL+Variable Graph	Description: This query chain generates a dependency view with the defined Global Variable Lists (GVLs) of the selected scope (for example, defined below the **Application**) and the variables defined in these GVLs.
	
	Results: The result is a graph of nodes (GVLs + variables) and its dependencies. If multiple GVLs are defined in an application, you see groups of nodes (a GVL and its variables). The groups are in most cases not connected to each other.
	
	Use Case: This dependency view can be used to explore the GVLs + variable of the project. You can add, for example, reading and writing nodes to visualize if a variable is used by multiple code snippets or used by one node only, to be declared more locally in, for example, a PRG.
	
	Source code example:
	`VAR_GLOBAL`
	`  G_iGVLTestVar: INT;`
	`END_VAR`

Name	Description
Implement Graph	Description: This query chain generates a dependency view with the function blocks (FBs) implementing interfaces of the selected scope (for example, defined below the **Application**). Results: The result is a graph of FBs and interfaces connected through implement edges. For the FBs in a project, you see which other FB is also an implementation of the linked interfaces. Use Case: The dependency view can be used to visualize the implement dependencies of FBs. You can use it to visualize for example, missing interface implementations if, for example, multiple FB types must meet the same specification to be used by a method. Source code example: `FUNCTION_BLOCK FB_MyTest IMPLEMENTS IF_MyTest, IF_MyTest2`
Implement+Extend Graph	Description: This query chain generates a dependency view with the function blocks (FBs) and interfaces of the selected scope (for example, defined below the **Application**) and how they are linked together. Results: The result is a graph of FBs and interfaces connected through implements edges or extends edges. Use Case: This dependency view can be used to visualize the implements and extends dependencies at once. Source code example: `FUNCTION_BLOCK FB_MyTest EXTENDS FB_MyTestBase IMPLEMENTS IF_MyTest, IF_MyTest2`
Library Stack	Description: This query chain generates a dependency view with the used libraries in the project. Results: From application point of view, you generate the library stack your application is based on. You see the directly referenced libraries and the indirectly referenced libraries. Use Case: This dependency view can be used to visualize the library stack at once in a graphical way. You can see the library layers. For example, **Application > Technology Libraries > PacDrive Library > System Libraries (Firmware API)**.
Node Graph	Description: This query chain generates a complete dependency view of the nodes of the selected scope (for example, defined below the **Application**) and how they are linked together. Results: The result graph contains the nodes of any type (FBs, PRGs, FCs, variables, libraries, DUTs, and so on) with the different kind of edges between these nodes. Use Case: This dependency view can be used to visualize the complete project. Keep in mind that this dependency view contains many different node and edge types at the same time and you are not able to see your code from a specific view (for example, compared to the call tree).
POU Graph	Description: This query chain generates a dependency view with the defined function blocks / functions (POUs) of the selected scope (for example, defined below the **Application**). Results: The result is a graph of nodes (POUs). If the POUs are linked, for example, through a call edge, this edge is also part of the result. Use Case: This dependency view can be used to explore the POUs. You can add for example, the variables connected to the POU you are interested in.

POU+Variable Graph	Description: This query chain generates a dependency view with the defined programs / function blocks / functions (POUs) of the selected scope (for example, defined below the **Application**) and the variables defined in these POUs.
	
	Results: The result is a graph of nodes (PRGs + variables) and its dependencies.
	
	Use Case: This dependency view can be used to explore the POUs and variables of the project. You can add for example, reading and writing nodes to visualize if a variable is used by multiple code snippets or used by one node only, to be declared more locally in, for example, a POU.
	
	Source code example:
	`PROGRAM SR_MyTest`
	`VAR`
	`  iSRTestVar: INT;`
	`END_VAR`

Name	Description
Read Graph	Description: This query chain generates a dependency view with the read operations of the selected scope (for example, defined below the **Application**) to a variable. Results: The result is a graph of POUs (programs / functions / function blocks, and so on) and the variables which are linked through a read edge. Use Case: This dependency view can be used to explore the read operations to variables and to see if it is read multiple times or only once in the project. Source code example: `dummy0 := iSRTestVar;`

Read+Write Graph	Description: This query chain generates a dependency view with the read and write operations of the selected scope (for example, defined below the **Application**) to a variable.
	
	Results: The result is a graph of POUs (programs / functions / function blocks, and so on) and the variables which are linked through a read or write edge.
	
	Use Case: This dependency view can be used for example, to explore the read and write operations to variables and to see if it is read multiple times or only once in the project but not written.
	
	Source code example:
	`dummy0 := iSRTestVar;`
	`iSRTestVar := dummy0;`

Write Graph	Description: This query chain generates a dependency view with the write operations of the selected scope (for example, defined below the **Application**) to a variable. Results: The result is a graph of POUs (programs / functions / function blocks, and so on) and the variables which are linked through a write edge. Use Case: This dependency view can be used to explore the write operations to variables and to see if it is written multiple times or only once in the project. Source code example: `iSRTestVar := dummy0;`
Device Graph	Description: This query chain generates a dependency view of the **Devices** tree. Results: The **Devices** tree with the device instances and their subnode dependencies. Use Case: This dependency view can be used to visualize the device tree and explore the linkage to variables representing the devices inside the application
Test Element Graph	Description: This query chain generates a dependency view of the test elements (**TestCases**, **TestResources**, **TestSets**, and so on). Results: The test elements and how they are linked. Use Case: This dependency view can be used to get an overview of which **TestCase** reuses **TestResources** or **TestSets**, and to navigate to the tested elements (functions, methods, function blocks, and so on).

Metrics

What's in This Chapter

Metric: Application Size (Code)

User Description

When logged into a controller, the source code is compiled and an executable is sent to the controller.

The executable consists of code and data sections when loaded into controller memory.

The application code size is the amount of memory needed on the controller to manage the application code.

> **NOTE:** This metric can only be calculated when you activate the **Consider Code and Data Size** option in the code analyzer **Configuration** tab. Refer to Code Analysis User Guide\Code Analysis Manager\Configuration.

Metric Calculation

The application code size is calculated based on the size of generated machine code for each POU (program, function block, function).

Metric: Application Size (Code+Data)

User Description

When logged into a controller, the source code is compiled (incl. Generate Code) and an executable is sent to the controller.

The executable consists of code and data sections when loaded into controller memory.

The application code and data size is the minimum amount of memory needed on the controller to run the application.

NOTE: This metric can only be calculated when you activate the **Consider Code and Data Size** option in the code analyzer **Configuration** tab. Refer to Code Analysis User Guide\Code Analysis Manager\Configuration.

Metric Calculation

The application code size is calculated based on the size of generated machine code for each POU (program, function block, function).

The application data size is calculated based on the size of the variables in the application.

NOTE: The sum of application code size and application data size (see dedicated metric) is not exactly the application code size + data size (this metric) due to alignment or code page size of the underlaying controller hardware.

Metric: Application Size (Data)

User Description

When logged into a controller, the source code is compiled (incl. Generate Code) and an executable is sent to the controller.

The executable consists of code and data sections when loaded into controller memory.

The application data size is the amount of memory needed on the controller to manage the data needed to execute the application code.

NOTE: This metric can only be calculated when you activate the **Consider Code and Data Size** option in the code analyzer **Configuration** tab. Refer to Code Analysis User Guide\Code Analysis Manager\Configuration.

Metric Calculation

The application data size is calculated based on the size of the variables in the application.

Metric: Call In

User Description

The Call In metric is used to get information about who is calling a method, function, function block, and so on.

Metric Calculation

Each call to an implementation is considered, but if the same object (method, function, etc.) is called twice in the same implementation, it is only counted once.

Example

Call In calculation example:

```
METH();

// Some other implementation

METH();
```

Call In Result (for method *METH*)

```
Call In = 1
```

Metric: Call Out

User Description

The Call Out metric is used to get information about which other objects (method, function, function block, etc.) are called by the implementation.

Metric Calculation

Each call to an implementation is considered, but if the same object (method, function, etc.) is called twice in the same implementation, it is only counted once.

Example

Call Out calculation example:

```
METH();

// Some other implementation

METH();
```

Call Out Result (for method *METH*)

```
Call Out = 1
```

Metric: Commented Variables (All) Ratio

User Description

This metric calculates the ratio (Unit: %) between commented and not commented variables in an object.

Metric Calculation

Each variable declaration in objects (function (FUN), function block (FB), Data Unit Type (DUT), Global Variable List (GVL), and so on) is verified whether it is commented or not.

The ratio between these two values is provided with this metric.

Example

Commented variables ratio calculation example:

```
Declaration:
1: PROGRAM SR_Main
2: VAR
3:     xCheck1: BOOL;//flag to identify
4:     uiMyVariable2: UINT;
5:     xFlag: BOOL;
6: END_VAR
```

Commented variables ratio result: 33.33 %

Metric: Commented Variables (In+Out+Global) Ratio

User Description

This metric calculates the ratio (Unit: %) between commented and not commented variables that are defined in *VAR_GLOBAL*, *VAR_INPUT*, *VAR_OUTPUT*, or *VAR_IN_OUT*.

Metric Calculation

Each variable declaration in objects (function (FUN), function block (FB), Data Unit Type (DUT), Global Variable List (GVL), and so on) are verified whether they are commented or not.

The ratio between these two values is provided with this metric.

Example

Commented variables (in+out+global) ratio calculation example:

```
Declaration:
1: PROGRAM SR_Main
2: VAR_IN
3:     i_xCheck1: BOOL;//flag to identify
3:     i_uiMyVariable2: UINT;
4: END_VAR
2: VAR
3:     xFlag: BOOL;
4: END_VAR
```

Commented variables (in+out+global) ratio result: 50 %

Metric: Cyclomatic Complexity

User Description

The Cyclomatic Complexity metric is used to measure the complexity of a program by counting the number of linearly independent paths in the source code.

Metric Calculation

Cyclomatic Complexity is computed using the control flow graph of the program. The complexity depends on the condition and decision points of the control flow graph.

For example:

- No condition or decision point: Complexity = 1 (one path through the code).
- One IF statement: Complexity = 2 (two paths through the code).
- One IF statement with two conditions: Complexity = 3 (three paths through the code).

There are different interpretations/implementation of Cyclomatic Complexity, depending on the analysis tool. Some tools do not consider expressions with AND/OR/etc. in IF, REPEAT, WHILE, etc. statements. The McCabe Cyclomatic Complexity is always increased by +1. Other tools also consider the expressions in the code flow (outside an IF, REPEAT, etc. statement) but later used in an IF or REPEAT statement which results in a higher Cyclomatic Complexity result. The Cyclomatic Complexity implementation in EcoStruxure Machine Expert considers expressions with AND/OR/etc. but does not consider pre-calculated expressions in the code flow or specified in a method call.

Example

Cyclomatic Complexity calculation example:

```
// MCC +1 (Initial Value)

// MCC +0 (Pre calculation of condition not considered)
a := b OR c;

// MCC +0 (Method call with condition not considered in
calling implementation)
METH4(a);

IF a AND b OR c XOR d AND NOT e THEN

   // MCC +5 (IF with 5 conditions)

    CASE i OF
       1..4:
           // MCC +1 (CASE range is considered as one
condition)

           FOR i := 1 TO 10 DO               // MCC +1

               METH1();
           END_FOR

       10, 11, 12, 13:
           // MCC +1 (multiple CASE labels considered as one
condition)

           REPEAT
               // MCC +1 (one condition in REPEAT)

               WHILE (a = TRUE AND b = FALSE) DO
                   // MCC +2 (two conditions in WHILE)

                   METH2();

               END_WHILE

           UNTIL (TRUE) END_REPEAT

        ELSE
            // MCC +0 (Default path through CASE statement)
```

```
            METH3();
    END_CASE
END_IF
```

Cyclomatic Complexity Result

```
Cyclomatic Complexity (MCC) = 12
```

Metric: Extended By

User Description

The Extended By metric is used to get information about how often a function block or an interface is extended by another function block or interface.

Metric Calculation

A function block can extend exactly one function block and implement multiple interfaces. An interface can extend multiple interfaces, but cannot implement other interfaces.

A function block or an interface can be extended by none or several interfaces. The number of direct extended interfaces is counted.

Example

Extended By calculation example:

```
FB_Test extends FB_Base implements IF_Test1, IF_Test2

FB_Base implements IF_Test4

IF_Test2 extends IF_Test3, IF_Test5
```

Implemented By Results

```
Extended By (FB_Base) = 1
Extended By (IF_Test3) = 1
Extended By (IF_Test5) = 1
```

Metric: Extends

User Description

The Extends metric is used to get information about how many interfaces are extended by a function block or an interface.

Metric Calculation

A function block can extend exactly one function block and implement multiple interfaces. An interface can extend multiple interfaces, but cannot implement other interfaces.

A function block or an interface can extend none or several interfaces. The number of direct extended interfaces is counted.

Example

Extends calculation example:

```
FB_Test extends FB_Base implements IF_Test1, IF_Test2
```

```
FB_Base implements IF_Test4

IF_Test2 extends IF_Test3, IF_Test5
```

Implemented By Results

```
Extends (FB_Test) = 1
Extends (IF_Test2) = 2
```

Metric: Fan In

User Description

The Fan In metric is used to get information about how many incoming dependencies (reads, writes, calls, and so on) to a node in the analysis data model (Dependency Model) are available. Incoming dependency means, that for example, a node is called and another node depends on this node.

Metric Calculation

Each incoming dependency is considered for a node. A node can be a function block, program, function, variable, library, property, method, task, and so on.

Example

Dependency example (list of dependencies):

```
FunctionBlockA defines MethodA
FunctionBlockA defines MethodB
FunctionBlockA defines VariableC
FunctionBlockA calls MethodA
MethodA calls MethodB
MethodB reads VariableC
```

Fan In Results

```
Fan In (FunctionBlockA) = 0
Fan In (MethodA) = 2
Fan In (MethodB) = 2
Fan In (VariableC) = 2
```

Metric: Fan Out

User Description

The Fan Out metric is used to get information about how many outgoing dependencies (reads, writes, calls, and so on) a node in the analysis data model (Dependency Model) has. Outgoing dependency means, that for example, a node is called and another node depends on this node.

Metric Calculation

Each outgoing dependency is considered for a node. A node can be a function block, program, function, variable, library, property, method, task, and so on.

Example

Dependency example (list of dependencies):

```
FunctionBlockA defines MethodA
FunctionBlockA defines MethodB
```

```
FunctionBlockA defines VariableC
FunctionBlockA calls MethodA
MethodA calls MethodB
MethodB reads VariableC
```

Fan Out Results

```
Fan Out (FunctionBlockA) = 4
Fan Out (MethodA) = 1
Fan Out (MethodB) = 1
Fan Out (VariableC) = 0
```

Metric: Halstead Complexity

User Description

The Halstead complexity metric is used to measure the complexity of a software program without running the program itself.

This metric is a static testing method where measurable software properties are identified and evaluated. The source code is analyzed and broken down to a sequence of tokens. The tokens are then classified and counted as operators or operands.

The operators and operands are classified and counted as follows:

Parameter	Description
n1	Number of distinct operators
n2	Number of distinct operands
N1	Total number of operators
N2	Total number of operands

There are a number of metric values that can be calculated to represent different aspects of complexity:

- *Halstead Difficulty (D)*
- *Halstead Length (N)*
- *Halstead CalculatedLength (Nx)*
- *Halstead Volume (V)*
- *Halstead Effort (E)*
- *Halstead Vocabulary (n)*

Halstead Complexity for POUs Implemented in Structured Text (ST)

The Halstead complexity was originally developed for textual languages (like C, C ++, Pascal, etc.) and is applicable to POUs implemented in structured text (ST).

NOTE: By default, the *Halstead Difficulty* is displayed.

Halstead Complexity for POUs Implemented in Function Block Diagram (FBD)

The function block diagram (FBD) belongs to the group of graphical implementation languages and is not text-based. A POU consists of multiple FBD networks. Then the Halstead complexity metric must be adapted to be applicable

to graphical languages. Operands and operators and their frequency (per FBD network) are considered as presented to the user (see Example for function block diagram (FBD)).

The Halstead complexity results calculated per FBD network are aggregated across the FBD networks and attached on POU (program, function block, function, method, or property) level.

NOTE: The calculated Halstead values (per FBD network) are *FBD Network Halstead Difficulty* and *FBD Network Halstead Length*.

The following aggregation types are applied per FBD network Halstead metric values (*Halstead Difficulty* and *Halstead Length*):

- Average
- Minimum
- Maximum
- Sum
- Consistency

NOTE: The most relevant aggregated values are *FBD Halstead Difficulty Network Max*, *FBD Halstead Difficulty Network Consistency*, *FBD Halstead Length Network Max*, and *FBD Halstead Length Network Consistency*. All other combinations (**Min**, **Sum**, and **Average**) are calculated and attached to the model but not displayed by default.

Metric Calculation

Value	Formula
Halstead Difficulty (D)	D = (n1 / 2) * (N2 / n2)
Halstead Length (N)	N = N1 + N2
Halstead CalculatedLength (Nx)	Nx = n1 * log2(n1) + n2 * log2(n2)
Halstead Volume (V)	V = N * log2(n)
Halstead Effort (E)	E = V * D
Halstead Vocabulary (n)	n = n1 + n2

NOTE: An expression in an *IF <expression> THEN* statement must not have parenthesis. They are considered as always available.

Metric Aggregation

Metric results like *FBD Network Halstead Difficulty* and *FBD Network Halstead Length* are aggregated across the FBD networks of a POU.

The *values* are the list of values of the same metric (for example, *FBD Network Halstead Length)* of the FBD networks of a POU.

The consistency value is a result of the Gini coefficient. The Gini coefficient is a measure of statistical dispersion. It measures the inequality among values of a frequent distribution. A Gini coefficient of 0 expresses equality, where all values are the same. A Gini coefficient of 1 expresses maximum inequality among values.

Example for structured text (ST)

Halstead calculation example (only the implementation part is considered for calculation):

```
IF (xInit = FALSE) THEN

    PerformInitialization();
    xInit := TRUE;
```

```
ELSE

    FOR i := 1 TO 5 DO

        iAxisId := i + 7;
        sAxisName := Standard.CONCAT('MyAxis ', INT_TO_
STRING(iAxisId));

        // Do some math calculations for each axis here
        udiResult := CalculateStuff(sName := sAxisName, iID
:= iAxisId);
    END_FOR

END_IF
```

List of Operators and its Frequencies:

Operator	Frequency
(operators)	
If	1
Then	1
LeftParenthesis	6
RightParenthesis	6
Equal	1
Semicolon	5
Assign	7
Else	1
For	1
EndFor	1
Do	1
Plus	1
Period	1
INT_TO_STRING	1
Colon	2
EndIf	1
(n1) 16	(N1) 37

List of Operands and its Frequencies:

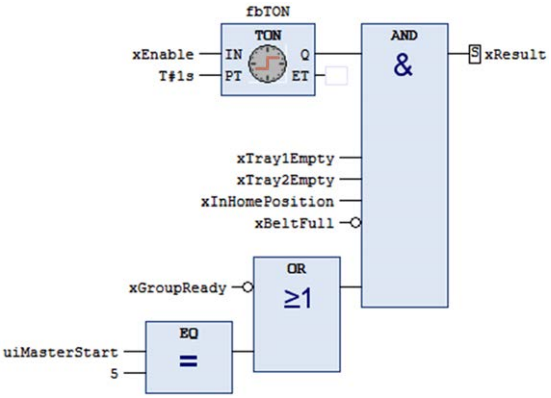
Operand	Frequency
(variables/methods/functions)	
xInit	2
PerformInitialization	1
i	2
iAxisId	3
sAxisName	2
Standard	1
CONCAT	1
udiResult	1
CalculateStuff	1
sName	1
iID	1
(constants)	
FALSE	1
TRUE	1
INT#1	1
INT#5	1
INT#7	1
'MyAxis '	1
(n2) 17	(N2) 22

Halstead Difficulty Result

```
Halstead Difficulty (D = (16/2) * (22/17) = 10.3529411764706
```

Example for function block diagram (FBD)

Halstead calculation example implemented in FBD (only the implementation part is considered for calculation):



List of Operators and its Frequencies:

Operator	Frequency
(operators)	
Assign	4
Set2	1
And	1
Negation2	2
Or	1
Eq	1
(n1) 6	(N1) 10

List of Operands and its Frequencies:

Operand	Frequency
(variables/methods/functions/constants)	
xResult	1
TON	1
fbTON	1
xEnable	1
T#1s	1
IN	1
PT	1
Q	1
ET	1
xTray1Empty	1
xTray2Empty	1
xInHomePosition	1
xBeltFull	1
xGroupReady	1
uiMasterStart	1
5	1
(n2) 16	(N2) 16

FBD Network Halstead Difficulty Result

```
FBD Network Halstead Difficulty (D) = (6/2) * (16/16) = 3
```

FBD Network Halstead Length

```
FBD Network Halstead Length (D) = 10 + 16 = 26
```

Metric: Implemented By

User Description

The Implemented By metric is used to get information about how often an interface is implemented by a function block.

Metric Calculation

A function block can extend exactly one function block and implement multiple interfaces. An interface can extend multiple interfaces, but cannot implement other interfaces.

An interface can be implemented by several function blocks. The number of direct implemented interfaces is counted.

> **NOTE:** If the function block extends another function block or an interface extends other interface, the derived implemented interfaces are not considered. If an interface is implemented in base function block and derived function block, it is counted twice.

Example

Implemented By calculation example:

```
FB_Test extends FB_Base implements IF_Test1, IF_Test2

FB_Base implements IF_Test4, IF_Test1

IF_Test2 extends IF_Test3, IF_Test5
```

Implemented By Results

```
Implemented By (IF_Test1) = 2
Implemented By (IF_Test2) = 1
Implemented By (IF_Test3) = 1
Implemented By (IF_Test4) = 1
Implemented By (IF_Test5) = 1
```

Metric: Implements

User Description

The Implements metric is used to get information about how many interfaces are implemented by a function block.

Metric Calculation

A function block can extend exactly one function block and implement multiple interfaces. An interface can extend multiple interfaces, but cannot implement other interfaces.

A function block can implement none or several interfaces. The number of direct implemented interfaces is counted.

> **NOTE:** If the function block extends another function block or an interface extends other interface, the derived implemented interfaces are not considered.

Example

Implements calculation example:

```
FB_Test extends FB_Base implements IF_Test1, IF_Test2

FB_Base implements IF_Test4

IF_Test2 extends IF_Test3, IF_Test5
```

Implements Results

```
Implements (FB_Test) = 2
Implements (FB_Base) = 1
```

Metric: Lines Of Code (LOC)

User Description

The software metric Lines Of Code (LOC) counts the number of source code lines of a program. This metric can be used to estimate the workload for program development, the programming productivity, and the maintainability of the application.

Metric Calculation

Each line in a textual implemented object (Function (FUN), function block (FB), data unit type (DUT), global variable list (GVL), and so on) is considered in the Lines Of Code metric.

Example

Lines Of Code calculation example:

```
Declaration:
1: PROGRAM SR_Main
2: VAR
3:    x: BOOL;
4: END_VAR

Implementation:
1:
2: IF (x = TRUE) THEN
3:    DoSomething();
4: END_IF
5:
6: // A nice comment
7: SpecialMethod();
```

Lines Of Code Result

```
Lines Of Code (LOC) = 11
```

Metric: Memory Size (Data)

User Description

An application or library is organized by complex types such as programs, function blocks, global variable lists, methods, actions, functions, structures, and so on. Inside each of these types, variables can be defined.

The complex types function blocks and structures can be instantiated multiple times and placed as a block inside the memory.

Each complex type definition (type information and variables), when instantiated, needs a specific amount of memory. Information about how much memory must be allocated and processed, for example in online modification situations or when used as input argument to methods. That information can then be used to identify large complex types which can cause issues when repeatedly instantiated.

Metric Calculation

For a function block or structure, the sizes of the variables are summed up. In addition, the function block type needs memory (list of methods, implemented interfaces, and so on). Based on the underlying controller architecture, memory alignment must be considered too. It is based on the variable type and order.

Example

Memory Size (Data) calculation example:

```
FUNCTION_BLOCK FB_XXX
VAR
    xVar1: BOOL; // 1 bit
    xVar2: BOOL; // 1 bit
    xVar3: BOOL; // 1 bit
    // 5 additional bits for alignment

    iVar4: INT; // 4 byte (on 32-bit architectures)
    xVar5: BOOL; // 1 bit
    // 7 additional bits for alignment

    fbComplex: FB_Test; // 20 byte
END_VAR


FUNCTION_BLOCK FB_YYY
VAREND_VAR

FUNCTION_BLOCK FB_ZZZ
VAR
    iVar1: INT;
END_VAR
```

Memory Size (Data) Results

```
Memory Size (Data) (FB_XXX) = 32
Memory Size (Data) (FB_YYY) = 4
Memory Size (Data) (FB_ZZZ) = 8
```

Metric: Number Of Actions

User Description

The Number of Actions metric is used to get information about how many actions are attached to a program or a function block.

Metric Calculation

Each Action attached to a program or function block is considered. Unused Actions are considered too.

Example

Number Of Action calculation example:

```
FB_MyAlphaModule
  - ACT_InitAxis1 (Action)
  - ACT_InitAxis2 (Action)
  - UpdateStatus (Method)
  - Calculate (Method)
  - Enabled (Property)
     - Get (Property Get)
     - Set (Property Set)
  - Status (Property)
     - Get (Property Get)
     - Set (Property Set)
  - SwitchNextState (Transition)
```

Number Of Actions Result (for variable *FB_MyAlphaModule*)

```
Number Of Actions = 2
```

Metric: Number Of GVL Usages

User Description

The Number Of GVL Usages metric is used to get information about how many global variables a programming object (programs, function blocks, functions, methods, and so on) uses (reading or writing).

Metric Calculation

Each programming object (program, function block, function, method, property get, property set, action,and so on) is handled individually.

Example

Number Of GVL Usages calculation example:

```
FB_MyAlphaModule
VAR
   iMyState : INT;
END_VAR

iMyState := GVL_IOs.G_iState;
if (iMyState = 10) THEN
   ;
END_IF
```

Number Of GVL Usages Result (for variable *FB_MyAlphaModule*)

```
Number Of GVL Usages = 1
```

Metric: Number Of Header Comment Lines

User Description

This metric counts the number of comments in the header of the declaration part.

These comments can help developers to understand what this object is doing, for what it is and how it is working.

Metric Calculation

Each comment line in the header of a declaration part in IEC objects (function (FUN), function block (FB), Data Unit Type (DUT), Global Variable List (GVL), and so on) is counted.

Example

Header comments calculation example:

```
Declaration:
1: //This PRG is the start point
2: //Methodes:
3: // - ....
4: // - ....
5: PROGRAM SR_Main
6: VAR
7:     x: BOOL;
8: END_VAR
```

Header comments metric result: 4

Metric: Number Of Instances

User Description

The Number Of Instances metric is used to get information about how often a complex type (function block, enumeration, structure, and so on) is used as variable type on programming objects (programs, function blocks, and so on).

Metric Calculation

Inside the declaration part, you can define variables. Each variable has an associated data type (complex type or elementary type). When used, the instance count of this data type is increased by +1.

NOTE: If the variable data type is an array data type, the underlaying base data type is used and the instance count is handled as +1. The array length is not considered.

NOTE: Instantiation paths through different complex types are not considered. For example, if a function block is instantiated multiple times, the complex types inside are only counted once.

Example

Number Of Instances calculation example:

```
SR_Main
VAR
    fbMyAlphaModule: FB_MyAlphaModule;
END_VAR

FB_MyAlphaModule
VAR
   astAxisStructures: ARRAY [1..10] OF ST_MyAxisStructure;
   fbSubModule: FB_MySubModule;
END_VAR

FB_MySubModule
VAR
    fbAxis: FB_MyAxis;
END_VAR

ST_MyAxisStructure
VAR
   iID: INT;
   fbAxis: FB_MyAxis;
END_VAR

FB_MyAxis
VAR
END_VAR
```

Number Of Instances Results

```
Number Of Instances (FB_MyAlphaModule) = 1
Number Of Instances (FB_MySubModule) = 1
Number Of Instances (ST_MyAxisStructure) = 1
Number Of Instances (FB_MyAxis) = 2
```

Metric: Number Of Library References

User Description

The Number Of Library References metric is used to get information about how many libraries are directly referenced by an application or POU space.

Metric Calculation

Each reference from an application to a library or from a library to another library is considered.

Example

Number Of Library References calculation example:

```
Application
   --> Library A
      --> Library B
   --> Library B
   --> Library C
```

Number Of Library References Result (for the application)

```
Number Of Library References = 3
```

Metric: Number Of Messages

User Description

The Number Of Messages (information, advisory, detected error, unrecoverable error) metric is used to get information about how many messages are emitted to the **Message View** during compilation.

Metric Calculation

The majority of messages are associated to a programming object such as function blocks, functions, and so on. Each message that is not associated to a programming object is attached to the application (or to the POU space for analysis of POU space only).

NOTE: The code analysis is based on a compilable application without compile errors. Messages of category error and unrecoverable error emitted during compilation cannot occur in analysis data.

Metric: Number Of Methods

User Description

The Number of Methods metric is used to get information about how many methods are attached to a program or a function block.

Metric Calculation

Each Method attached to a program or function block is considered. Unused Methods are considered too.

Example

Number Of Methods calculation example:

```
FB_MyAlphaModule
  - ACT_InitAxis1 (Action)
  - ACT_InitAxis2 (Action)
  - UpdateStatus (Method)
  - Calculate (Method)
  - Enabled (Property)
     - Get (Property Get)
     - Set (Property Set)
  - Status (Property)
     - Get (Property Get)
     - Set (Property Set)
  - SwitchNextState (Transition)
```

Number Of Methods Result (for variable *FB_MyAlphaModule*)

```
Number Of Methods = 2
```

Metric: Number Of Multiline Comments

User Description

This metric counts the multiline comments in an object.

Do not use multiline comments because the start and end of such a comment could get lost while merging.

For example, a commented out source code may unintentionally become part of the program again.

Metric Calculation

Example

Multiline comments calculation example:

```
Declaration:
1: (*This is a multiline
2:  comment in header*)
3: PROGRAM SR_Main
4: VAR
5:     xCheck1: BOOL;(*not needed
6:     uiMyVariable2: UINT;*)
7:     xFlag: BOOL;
8: END_VAR
```

Multiline comments result: 2

Metric: Number Of Properties

User Description

The Number Of Properties metric is used to get information about how many properties are attached to a program or a function block.

Metric Calculation

Each Property attached to a program or function block is considered. Unused Properties are considered too.

Example

Number Of Properties calculation example:

```
FB_MyAlphaModule
  - ACT_InitAxis1 (Action)
  - ACT_InitAxis2 (Action)
  - UpdateStatus (Method)
  - Calculate (Method)
  - Enabled (Property)
     - Get (Property Get)
     - Set (Property Set)
  - Status (Property)
     - Get (Property Get)
     - Set (Property Set)
  - SwitchNextState (Transition)
```

Number Of Properties Result (for variable *FB_MyAlphaModule*)

```
Number Of Properties = 2
```

Metric: Number Of Reads

User Description

The Number of Reads metric is used to get information about which variables are read.

Metric Calculation

Each read of a variable in an implementation is considered, but if the same variable is read twice in an implementation, it is only counted once.

Example

Number Of Read calculation example:

```
METH(iMyVariable);

// Some other implementation

METH(iMyVariable);
```

Number Of Reads Result (for variable *iMyVariable*)

```
Number Of Reads = 1
```

Metric: Number Of Tasks

User Description

The Number Of Tasks metric is used to get information about how many tasks an application has.

Metric Calculation

Tasks can only be defined in applications. The number of defined tasks of an application is returned.

Example

Number Of Tasks calculation example:

```
Application
  - SR_Main (Program)
  - FB_Test1 (FunctionBlock)
  - FB_Test2 (FunctionBlock)
  - TaskConfiguration
    - TASK_SR_Main (Task)
    - TASK_Visu (Task
```

Number Of Tasks Result (for the application)

```
Number Of Tasks = 2
```

Metric: Number Of Transitions

User Description

The Number Of Transitions metric is used to get information about how many transitions are attached to a program or a function block.

Metric Calculation

Each transition attached to a program or function block is considered. Unused transitions are considered too.

Example

Number Of Transitions calculation example:

```
FB_MyAlphaModule
  - ACT_InitAxis1 (Action)
  - ACT_InitAxis2 (Action)
  - UpdateStatus (Method)
  - Calculate (Method)
  - Enabled (Property)
     - Get (Property Get)
     - Set (Property Set)
  - Status (Property)
     - Get (Property Get)
     - Set (Property Set)
  - SwitchNextState (Transition)
```

Number Of Transitions Result (for variable *FB_MyAlphaModule*)

```
Number Of Properties = 2
```

Metric: Number Of Variables

User Description

The Number Of Variables metric is used to get information about how many variables are defined in the declaration part of programs, function blocks, functions, methods, property *Get* or *Set*, transitions, global variable lists, and so on.

Metric Calculation

Each Variable defined in a declaration part is considered. Unused Variables are considered too.

Example

Number Of Variables calculation example:

```
FB_MyAlphaModule
VAR
   i: INT;
END_VAR
VAR_INPUT
   i_iCommand: INT;
   i_lrPosition: LREAL;
END_VAR
VAR_OUTPUT
   q_iStatus: INT;
END_VAR
```

Number Of Variables Result (for variable *FB_MyAlphaModule*)

```
Number Of Variables = 4
```

Metric: Number Of Writes

User Description

The Number of Writes metric is used to get information about which variables are written.

Metric Calculation

Each writing of a variable in an implementation is considered, but if the same variable is written twice in an implementation, it is only counted once.

Example

Number Of Write calculation example:

```
iMyVariable := 1;

// Some other implementation

iMyVariable := 2;
```

Number Of Writes Result (for variable *iMyVariable*)

```
Number Of Writes = 1
```

Metric: Number Of FBD Networks

User Description

The Number of Function Block Diagram (FBD) Networks metric is used to get information about how many networks are available in an FBD implemented program, function block, function, method, or property.

Metric Calculation

Each FBD network available in a program, function block, function, method, or property is considered.

Example

Number Of FBD Networks calculation example:

```
FB_MyAlphaModule (FB) - implemented in FBD
 - FBD Network 1
 - FBD Network 2
 - FBD Network 3
```

Number Of FBD Network Result (for variable *FB_MyAlphaModule)*

```
Number Of FBD Networks = 3
```

Metric: Source Code Comment Ratio

User Description

Comments can help developers to understand what the code is doing, for what it is and how it is working.

This metric calculates the ratio (Unit: %) between CLOC (Comment Lines Of Code) and SLOC (Source Lines Of Code) of the implementation part of an object.

CLOC: Number of comment lines, including lines that have source code and comment.

SLOC: Number of lines without comments and blank lines, including lines that have source code and comment.

Metric Calculation

Each line in a textual implemented object (function (FUN), function block (FB), Data Unit Type (DUT), program (PRG), and so on) is verified whether it contains a comment or source code.

The ratio between these two values is provided with this metric.

Example

Source code comment ratio calculation example:

```
1:
2: IF (x = TRUE) THEN
3:     DoSomething();//This is very important and hard to
understand
4: END_IF
5:
6: // A nice comment
7: SpecialMethod();
```

Source code comment ratio result: 50 %

Metric: Stack Size

User Description

An application or library is organized by complex types such as programs, function blocks, global variable lists, methods, actions, functions, structures, and so on. Inside each of these types, variables can be defined.

When a complex type like a function, method, action, property *Get*, property *Set*, or a transition is called, memory on the stack is needed to execute the method. The stack size information can be used to identify the complex type which is using too much memory of the stack.

NOTE: Stack memory that is available per task is limited and defined by the controller used. Large consumption of stack size can result in exceptions.

NOTE: If a function block type is used as method input variable type (call by value), the memory size of the complex type is needed (refer to Memory Size Data, page 74). Do not use call by value for complex types like method or function inputs.

Metric Calculation

For a complex type like function or method, the sizes of the variables are summed up. When the complex type is called, the size is allocated on top of the stack and the input values are copied the allocated memory. During the code execution of the function or method, these memory values are used.

NOTE: Each method or function call has its own memory and does not conflict if a method is called in parallel by another task.

Example

Stack Size calculation example:

```
FUNCTION_BLOCK FB_XXX
VAR
    fbComplex: FB_Test; // 20 byte
END_VAR

// method call of FB_XXX
Meth1(TRUE);

METHOD Meth1
VAR_INPUT
   xTest1: BOOL;
END_VARVAR
   iTest2: INT;
END_VAR


METHOD METH2
VAR_INPUT
    fbComp: FB_XXX;
END_VAR
```

Stack Size Results

```
Stack Size (METH1) = 8
Stack Size (METH2) = 32
```

Conventions

What's in This Chapter

Convention Queries

Convention Queries

The following queries are available by default for EcoStruxure Machine Expert installation.

Group: Code Smells

Name
Empty Implementation
Local Variable overwrites Global Variable
Useless DUT

Group: Comment Checks

Name
Complex POU With Low Comment Ratio
Multiline Comments Usage
No Header Comment
Uncommented Variable (All)
Uncommented Variable (In+Out+Global)

Group: Complex Type Name Checks

- Verify the user-defined objects to follow configured rules.
- Example: Enumerations must start with *ET_*. For example, *ET_MyTestEnumeration*.

Name	Prefix
Complex Type Name (Enumeration)	*ET_*
Complex Type Name (Function)	*FC_*
Complex Type Name (Function block)	*FB_*
Complex Type Name (Interface)	*IF_*
Complex Type Name (Program)	*SR_*
Complex Type Name (Struct)	*ST_*
Complex Type Name (TestCase)	*TC_*
Complex Type Name (TestResource)	*TR_*
Complex Type Name (TestSeries)	*TS_*
Complex Type Name (TestSet)	*TS_*
Complex Type Name (Union)	*UT_*

Group: Messages

Name
Compile Messages

Group: Variable Name Checks (Complex Types) and Variable Name Checks (Elementary Types)

Variable names should be prefixed by the scope (local, input, output, input/output, and so on) in combination with a prefix for the variable data type. For example, `q_` for input variables and `ar` for variables of type array -> `q_arMyVariable`.

Coding style guides suggest prefixing of variables.

Name	Prefix
Variable Name (Array)	`ar`
Variable Name (Enumeration)	`et`
Variable Name (FunctionBlock)	`fb`

Name	Prefix
Variable Name (Interface)	`if`
Variable Name (Pointer)	`p`
Variable Name (Reference)	`r`
Variable Name (Struct)	`st`
Variable Name (TestCase)	`tc`
Variable Name (Union)	`ut`

Name	Prefix
Variable Name (BIT)	`x`
Variable Name (BOOL)	`x`
Variable Name (BYTE)	`b`
Variable Name (DATE)	`dat`
Variable Name (DINT)	`di`
Variable Name (DT)	`dt`
Variable Name (DWORD)	`dw`
Variable Name (INT)	`i`
Variable Name (LINT)	`li`
Variable Name (LREAL)	`lr`
Variable Name (LTIME)	`ltim`
Variable Name (LWORD)	`lw`
Variable Name (REAL)	`r`
Variable Name (SINT)	`si`
Variable Name (STRING)	`s`
Variable Name (TOD)	`tod`
Variable Name (UDINT)	`udi`
Variable Name (UINT)	`ui`
Variable Name (ULINT)	`uli`
Variable Name (USINT)	`usi`
Variable Name (WORD)	`w`
Variable Name (WSTRING)	`ws`

Group: Threshold Checks

Name	Description
Inheritance Depth Limit	Description: Verify for each interface or function block that the inheritance depth (Extend edge chain) does not raise the limit. Use case: Verify inheritance depth limit to avoid too complex and too deep structures with non-operational maintainability. Example: FB1 extends FB2 extends FB3 extends FB4.
Variable Length Check	Description: Verify the length for each variable not to raise the suggested length limit. Use case: Verify variable name length to increase the readability of source code to improve maintainability. Example of a correct variable: `iCounterVariable1`. Example of an incorrect variable: `iAnotherToLengthCounter1Variable`.

Name	Description
Number Of Method Limit	Description: Verify the number of methods defined, for example for a function block not to raise the suggested length limit. Use case: Too many methods below a function block are an indicator to refactor the function block to implement only one feature.
Number Of Properties Limit	Description: Verify the number of properties defined, for example for a function block not to raise the suggested length limit. Use case: Too many properties below a function block are an indicator to refactor the function block to implement only one feature.

Group: Usage Checks

Name	Description
Input Variable Read Check	Description: Verify each input variable if it is read from outside the POU. Use case: Do not read an input variable of a POU.
Input Variable Type Check	Description: Verify each input variable if it is of type function block. Use case: Do not use input variables of type function block.
Input Variable Writes Check	Description: Verify each input variable if it is written from within the POU where it is defined in. Use case: Do not write an input variable from within a POU.
Output Variable Read Check	Description: Verify each output variable if it is read from inside the POU. Use case: Do not read an output variable from inside of a POU.
Output Variable Type Check	Description: Verify each output variable if it is of type function block. Use case: Do not use output variables of type function block.
Persistant Usage Check	Description: Verify each variable if it is PERSISTANT and if it is declared in a function bock. Use case: Do not use PERSISTANT variables in function blocks.
Retain Usage Check	Description: Verify each variable if it is RETAIN and if it is declared in a function bock. Use case: Do not use RETAIN variables in function blocks.
Unused Variables Check	Description: Verify the variables if not read and not written in the complete project (-> unused variable). Use case: Detect unused variables and report them, cleanup the project, and improve the code quality.

Convention: Access to Global Variable in FB_Init + FB_Exit

User Description

This convention detects a read or write access to a global variable from the *FB_INIT*, *FB_EXIT* or *FB_REINIT* method of a function block.

NOTE: Do not access global variables while executing these methods.

NOTE: Activate the option **Consider Implicit Methods** in the **Configuration** tab of the **Code Analysis Manager** to consider implicit methods during analysis. Refer to Configuration\Consider Implicit Methods, page 30.

Example

```
FUNCTION_BLOCK FB_TEST
```

```
VAR
END_VAR

METHOD FB_INIT: BOOL
VAR_INPUT
    bInitRetains: BOOL;
    bInCopyCode: BOOL;
END_VAR

GVL.g_iInitState := 3; // Global varible is accessed from
FB_INIT
```

Convention: Compile Messages

User Description

When compiling an application, the compiler reports detected unrecoverable errors, errors, advisories (often referred to as warnings), and information to the developer.

Detected unrecoverable errors and errors must be handled by the developer to get the application to run on a controller.

In addition, advisories are reported to the developer. Advisories should be handled by the developer. The number of advisories should be as close to zero as possible when an application is planned to be released.

Information messages are reported, for example, to inform the developer about the progress or needed memory sizes of the compiled application.

> **NOTE:** A code analysis run can only be made on an application for which no irreconcilable errors are detected. Only compile messages are considered during code analysis. Therefore, error messages are not supported.

Convention Verification Rule

For convention verification, the available compile messages (of the complete analysis data) are considered and reported. The compile message severity is used as severity of the convention violation.

Convention: Complex POU With Low Comment Ratio

User Description

This convention lists all complex objects with low comment ratio.

In contrast to other metrics where only a focus is set to the ratio between CLOC (Comment Lines Of Code) and SLOC (Source Lines Of Code), this convention allows you to filter to objects with high complexity and low comment ratio.

This convention has a focus on the implementation part of an object.

Convention: Complex Type Name Checks

User Description

Coding style is a set of rules or guidelines applied when writing source code. Following a specified coding style helps:

- To read and understand the source code
- To avoid and find programming issues
- To maintain the source code

Based on the Programming Guidelines (Naming Conventions (see Programming Guide), Prefixes (see Programming Guide)) for source code, complex type name convention queries are available to verify the suggested complex type names per type.

Convention Verification Rule

For convention verification, the complex type is combined with the object name.

Complex type name prefixes based on complex types:

- Function block: *FB_* as prefix
- Program: *SR_* as prefix
- Enumeration: *ET_* as prefix
- Structure: *ST_* as prefix
- etc.

Example

```
FB_MyAxisWrapper
ST_MyDataStruct
ET_MyEnumeration
SR_Main
etc.
```

Convention: Directly Referenced Library Not Signed

User Description

To help ensure the integrity of the project, directly referenced libraries should have a valid certificate from a trusted source.

Convention: Empty Implementation

User Description

This convention detects POUs that are implemented in structured text language and have no source code in the implementation part.

Convention: Global Variable Accessed Only in One POU

User Description

This convention detects the access of a global variable only in one POU.

Locally used variables should only be declared locally.

Convention: Inheritance Depth Limit

User Description

IEC-61131-3 provides language features to extend function blocks or implement interfaces. This is called inheritance and can result in a chain of inheritances. In theory, there is no limit for the inheritance depth, but nesting can become too complex to understand the inheritance tree of interfaces and function blocks.

For the application maintainability reasons, the inheritance depth limit can be verified and reported through convention violation rules.

Convention Verification Rule

The keyword *extends* between function blocks and function blocks, or interfaces and other interfaces is used to verify the chain length.

Example

The inheritance depth of the example is 6.

```
FB_Test1 extends FB_Test2
FB_Test2 extends FB_Test3
FB_Test3 extends FB_Test4
FB_Test4 extends FB_Test5
FB_Test5 extends FB_Test6
FB_Test6 extends FB_Test7
```

Convention: Input Variable Read Check

User Description

In the declaration part of a program, function block, method, or function, input variables can be defined. When objects of this type are called, input values must be specified. When a method or a function is called, the input values are copied to the stack, where they are stored in a dedicated memory area and used exclusively by the method or function call.

Compared to programs where exactly one instance exists in memory (or function blocks that are instantiated multiple times in memory), the programs or function blocks can be called multiple times, by multiple tasks, and the same memory location for the input variable is used in parallel.

For application execution stability, the input variable should not be read from outside the program or function block.

NOTE: Access to an input variable from a method defined below a function block is also considered as access from outside the function block. The convention violation of the input *Variable Read Check* could be a false positive, if the developer verifies, for example, by code review, that the input variable is initialized before using it in a method.

Convention Verification Rule

Each read access to an input variable from outside the function block (body) or program (body) implementation itself is reported as convention violation.

Example

```
SR_Main
VAR
   fbTest: FB_Test;
END_VAR

// call of FB method without calling FB (Body) before
fbTest.Meth();


FB_Test
VAR_INPUT
   i_lrCurrentAxisPosition: LREAL;
END_VAR


METHOD Meth()
VAR
   i_lrVar1: LREAL;
END_VAR

// potential access to not properly initialized variable
i_lrVar1 := i_lrCurrentAxisPosition;
```

Convention: Input Variable Type Check

User Description

In the declaration part of a POU, input variables can be defined. When this POU is called, input values must be specified.

These input values are copied and stored in a dedicated memory area.

For application execution stability, the input variable should not be of type function block.

Convention Verification Rule

Each input variable of type function block is reported as convention violation.

Example

```
SR_Main
VAR
```

```
   fbTest: FB_Test;
   fbArg: FB_MyArg;
END_VAR

// call of FB method without calling FB (Body) before
fbTest(i_fbMyArg := fbArg);

FB_MyArg
VAR_INPUT
END_VAR

FB_Test
VAR_INPUT
   i_fbMyArg: FB_MyArg;
END_VAR
```

Convention: Input Variable Write Check

User Description

In the declaration part of a program, function block, method, or function, input variables can be defined. When objects of this type are called, input values must be specified. When a method or a function is called, the input values are copied to the stack, where they are stored in a dedicated memory area and used exclusively by the method or function call.

Compared to programs where exactly one instance exists in memory (or function blocks that are instantiated multiple times in memory), the programs or function blocks can be called multiple times, by multiple tasks, and the same memory location for the input variable is used in parallel.

For application execution stability, the input variable should only be written by the caller from outside the program or function block.

> **NOTE:** Variables of category input means that data is transferred to that construct, be it a program, function block, method, or function. Results of the call are of category output. Input variables should be restricted to those passed to the construct, and input variables of the construct should not be read from outside the construct nor written to from inside the construct.

Convention Verification Rule

Each write access to an input variable from inside the function block is reported as convention violation.

Example

```
SR_Main
VAR
   fbTest: FB_Test;
   xTest: BOOL;
END_VAR

// call of FB method without calling FB (Body) before
fbTest(TRUE);

//potential violation reading outside the construct
xTest := fbTest.i_xEnable ;

FB_Test
VAR_INPUT
```

```
   i_xEnable: BOOL;
END_VAR

// potential violation. Now the input value has changed its
value.
i_xEnable := FALSE;
```

Convention: Library Does Not Specify Its Type

User Description

Libraries must specify their library type.

Each type has different rules and limitations that apply to them and make them suitable for different use cases.

The following library types are possible:

- Forward compatible libraries, page 94
- Container library, page 94
- Interface library, page 94
- Common library, page 94

Forward Compatible Library

A forward-compatible library (FCL) is designed to be compatible with future versions. A later library version can be used in already existing projects without any changes.

- The property `ForwardCompatibleLibrary(Bool) := True` must be set.
- The property `LanguageModelAttribute(Text) := 'qualified-access-only'` must be set.

Container Library

A container library is used to encapsulate a set of libraries. A container library publishes the symbols of its referenced libraries at its top-level.

- The property `IsContainerLibrary (Bool) := True` must be set.
- A container library must contain only one library manager and library documentation.

Interface Library

Interface libraries define the interface of a software component and provide only language elements that do not generate code (constants, structures and interfaces).

The property `IsInterfaceLibrary (Bool) := True` must be set.

Common Library

A common library is the right choice if none of the mentioned libraries applies.

- The property `Placeholder(Text) := <Placeholder name>` must be set.

- The property `LanguageModelAttribute(Text) := 'qualified-access-only'` must be set.

Convention: Library Referenced in an Incorrect Way

User Description

Libraries should be referenced according to their type. Libraries can either be referenced with a specific version or they can be set to always use the latest version.

- Interface libraries should be set to always use the latest version.
- Container libraries should be referenced with a specific version.
- Forward compatible libraries should be referenced with a specific version.

Convention: Local Variable Overwrites Global Variable

User Description

A programming issue arises when a local variable has the same name as a global variable. This can create confusion, because a software engineer might not notice that the qualifier (*GVL_...*) is missing from the code needed to use the global variable. In the following example, the local variable is used.

Example

```
PROGRAM SR_Main
VAR
    g_xAuthenticated: BOOL;
END_VAR

IF NOT g_xAuthenticated: THEN
    Authenticate();
END_IF

VAR_GLOBAL
    g_xAuthenticated: BOOL;
END_VAR
```

Convention: Multiline Comment Usage

User Description

This convention verifies if multiline comments are used in objects.

Do not use multiline comments because the start and end of such a comment could get lost while merging.

For example, a commented-out code may unintentionally become part of the program again.

Example

Multiline comments calculation example.

Declaration:

```
1: (*This is a multiline
2: comment in header*)
3: PROGRAM SR_Main
4: VAR
5:     xCheck1: BOOL;(*not needed
6:     uiMyVariable2: UINT;*)
7:     xFlag: BOOL;
8: END_VAR
```

Convention: No Header Comment

User Description

Many coding style guides suggest that a general description about what a POU is doing and how it is working be present in the header of the declaration part.

Example

All variants of comments are counted for this verification:

```
- //my comment
- ///my doc comment
- (*my multi line comment*)
```

Convention: Number of Methods Limit

User Description

For maintainability reasons of applications, there are design principles available how to organize your code. For example, only one job per function block or per method.

Applying code design rules can help to detect that, for example, too many methods are attached to one function block and is an indicator to split the function block itself into code pieces.

Convention Verification Rule

The number of methods attached to a function block or program is used to verify whether the limit is exceeded.

Convention: Number Of Pins Limit (Input/Output)

User Description

The number of input/output variables (*VAR_IN_OUT*) should be within a limited range set by your organization. Refer to your coding style guides.

Using a function block with too many pins affects readability (a limit could be around 10 for graphical programming languages).

If more input, output, or input/output variables are required, consider reducing the number of pins (input, output, input/output variables) of POUs by introducing a structure to group some input, output, or input/output variables.

Convention: Number Of Pins Limit (Input)

User Description

The number of input variables (*VAR_INPUT*) should be within a limited range set by your organization. Refer to your coding style guides.

Using a function block with too many pins affects readability (a limit could be around 10 for graphical programming languages).

If more input, output, or input/output variables are required, consider reducing the number of pins (input, output, input/output variables) of POUs by introducing a structure to group some input, output, or input/output variables.

Convention: Number Of Pins Limit (Output)

User Description

The number of output variables (*VAR_OUTPUT*) should be within a limited range set by your organization. Refer to your coding style guides.

Using a function block with too many pins affects readability (a limit could be around 10 for graphical programming languages).

If more input, output, or input/output variables are required, consider reducing the number of pins (input, output, input/output variables) of POUs by introducing a structure to group some input, output, or input/output variables.

Convention: Number of Properties Limit

User Description

IEC-61131-3 provides language features to organize an application in programs, function blocks, and Global Variable Lists (GVL). To reduce complexity and support object orientation, properties can be attached. Each property provides functional access to the information behind.

Too many attached properties:

- Are not easy to handle or not easy to understand by the developer.
- Can result in naming conflicts.
- Can be an indicator that a program or function block realizes multiple jobs.

The convention result can be used as an indicator to split a program/function block into several programs/function blocks, each with one job only.

Thus the maintainability of applications can be improved.

Convention Verification Rule

The number of properties attached to a function block, program, or Global Variable List (GVL) is used to verify whether the limit is exceeded.

Convention: Output Variable Read Check

User Description

In the declaration part of a program, function block, method, or function, output variables can be defined. When objects of this type are called, output value targets can be specified. When a method or a function is called, the output values are copied to the stack, where they are stored in a dedicated memory area and used exclusively by the method or function call and reused by its caller.

Compared to programs where exactly one instance exists in memory (or function blocks that are instantiated multiple times in memory), the programs or function blocks can be called multiple times, by multiple tasks, and the same memory location for the input variable is used in parallel.

For application execution stability, the output variable should only be read by the caller from outside the program or function block.

NOTE: Reading an output variable from within an implementation could be false positive, if the developer verifies, for example, by code review, that the output variable is written before using it later in the code.

Convention Verification Rule

Each read access of an output variable from inside the function block is reported as convention violation because it cannot be verified that a proper value was assigned before.

Example

```
SR_Main
VAR
   xResult: BOOL;
   fbTest: FB_Test;
END_VAR


// call of FB method without calling FB (Body) before
fbTest(q_xEnable => xResult);


FB_Test
VAR_OUTPUT
   q_xEnable: BOOL;
END_VAR


// potential violation. Now the input value has changed its
value.
IF (q_xEnable) THEN
     ; // Violation. Unclear value of output variable.
END_IF
```

Convention: Output Variable Type Check

User Description

In the declaration part of a POU, output variables can be defined. When this POU is called, output values can be assigned.

These output values are copied by value (memory copy).

For application execution stability, the output variable should not be of type function block.

Convention Verification Rule

Each output variable of type function block is reported as convention violation.

Example

```
SR_Main
VAR
    fbTest: FB_Test;
    fbArg: FB_MyArg;
END_VAR

// call of FB method without calling FB (Body) before
fbTest(q_fbMyArg => fbArg);

FB_MyArg
VAR_INPUT
END_VAR

FB_Test
VAR_OUTPUT
    q_fbMyArg: FB_MyArg;
END_VAR
```

Convention: Persistent Usage Check

User Description

If you declare a variable as PERSISTENT in a function block, then the entire instance of this function block is saved in the persistent range (all data of the block), but only the declared PERSISTENT variable is restored.

This increased memory consumption and additional handling of persistent variables can cause performance issues.

NOTE: The compiler treats a VAR PERSISTENT declaration just like a VAR PERSISTENT RETAIN or VAR RETAIN PERSISTENT declaration.

Convention: Referenced Library is not Signed

User Description

To help ensure the integrity of the project, all referenced libraries should have a valid certificate from a trusted source.

This query verifies both directly referenced libraries as well as libraries referenced indirectly by another library.

Convention: Retain Usage Check

User Description

If you declare a variable as RETAIN in a function block, then the entire instance of this function block is saved in the retain range (all data of the block), but only the declared RETAIN variable is restored.

This increased memory consumption and additional handling of retain variables can cause performance issues.

Convention: Uncommented Variable (All)

User Description

This convention verifies whether uncommented variables exist in an object.

Example

Declaration:

```
1: PROGRAM SR_Main
2: VAR
3:     xCheck1: BOOL;//flag to identify
4:     uiMyVariable2: UINT;
5:     xFlag: BOOL;
6: END_VAR
```

uiMyVariable2 and *xFlag* are not commented. Therefore a convention violation is created.

Convention: Uncommented Variable (In+Out+Global)

User Description

This convention verifies whether uncommented variables defined in *VAR_GLOBAL*, *VAR_INPUT*, *VAR_OUTPUT*, or *VAR_IN_OUT* exist in source code.

Example

Declaration:

```
1: PROGRAM SR_Main
2: VAR_IN
3:     i_xCheck1: BOOL;//flag to identify
3:     i_uiMyVariable2: UINT;
4: END_VAR
2: VAR
3:     xFlag: BOOL;
4: END_VAR
```

i_uiMyVariable2 is not commented. Therefore a convention violation is created.

Convention: Unused Enum Constants Check

User Description

This convention detects enumeration constants that are not used in your application.

Not used enumeration constants, for example, in state machines, may indicate that they are not used properly or are incomplete, or that outdated source code is present.

Convention: Unused Variables Check

User Description

In the declaration of a program, function block, method, or function variables can be defined. Normally, these variables are used inside the code. If a variable is defined but not used (read or written) in the code, it consumes memory.

Convention Verification Rule

Each variable without a read or write access is reported as convention violation.

Example

```
SR_Main
VAR
   xMyUnusedVariableResult: BOOL;
   fbTest: FB_Test;
END_VAR

// call of FB method but output is not assigned to intended
result variable.
fbTest(q_xEnable => );
```

Convention: Useless DUT

User Description

This convention detects DUTs (Data Unit Types) that consist only of one element.

This convention violation can indicate an incomplete refactoring activity or a feature that has not been completed.

Example

```
TYPE UT_MyUnion :
UNION
    xInit : BOOL; //only one element in union
END_UNION
END_TYPE

TYPE ST_MyStruct :
STRUCT
    xInit : BOOL; //only one element in struct
END_STRUCT
END_TYPE

TYPE ET_MyEnum :
(
    State := 1 //only one element in enum
);
END_TYPE
```

Convention: Variable Name Checks

User Description

Coding style is a set of rules or guidelines applied when writing source code. Following a specified coding style helps:

- To read and understand the source code
- To avoid and find programming issues
- To maintain the source code

Based on the Programming Guidelines (Naming Conventions (see Programming Guide), Prefixes (see Programming Guide)) for source code, variable name convention queries are available to verify the suggested variable name per data type and variable scope.

Convention Verification Rule

For convention verification, the variable name is combined with its linked data type and the scope where the variable is defined in.

Scopes:

- Local variable scope: No special scope prefix (*VAR* ... *END_VAR*)
- Input variable scope: *i_* as prefix (*VAR_INPUT* ... *END_VAR*)
- Output variable scope: *q_* as prefix (*VAR_OUTPUT* ... *END_VAR*)
- In-/Output variable scope: *iq_* as prefix (*VAR_IN_OUT_* ... *END_VAR*)
- Global variable scope: *G_* as prefix
- Global constants scope: *Gc_* as prefix
- etc.

Variable name prefixes based on data type:

- INT: *i* as prefix
- DINT: *di* as prefix

- UDINT: *udi* as prefix
- REAL: *r* as prefix
- LREAL: *lr* as prefix
- Function block: *fb* as prefix
- POINTER TO: *p* as prefix
- etc.

Example

```
VAR
    iMyVariable1: INT;
    uiMyVariable1: UINT;
    rMyVariable1: REAL;
    piMyVariable7: POINTER TO INT;
END_VAR
VAR_INPUT
    i_iMyVariable2: INT;
    i_uiMyVariable2: UINT;
    i_rMyVariable2: REAL;
END_VAR
VAR_IN_OUT
    iq_iMyVariable3: INT;
    iq_uiMyVariable3: UINT;
    iq_rMyVariable3: REAL;
END_VAR
VAR_OUTPUT
    iq_iMyVariable2: INT;
    iq_uiMyVariable2: UINT;
    iq_rMyVariable2: REAL;
END_VAR
```

Convention: Variable Name Length Check

User Description

Coding style is a set of rules or guidelines applied when writing source code. Following a specified coding style helps:

- To read and understand the source code
- To avoid and find programming issues
- To maintain the source code

For readability reasons, there are suggestions for variable names and length. The length of variables can be verified against a user-defined limit.

Convention Verification Rule

The length of a variable name is compared with a length threshold.

Example

```
iMyVariable                                 // length 11 -->
OK
iSpecialAndVeryLongAndHardToReadVariable    // length 20 -->
Not OK
```

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

U

V

Schneider Electric
35 rue Joseph Monier
92500 Rueil Malmaison
France

www.se.com

EIO0000002710.08