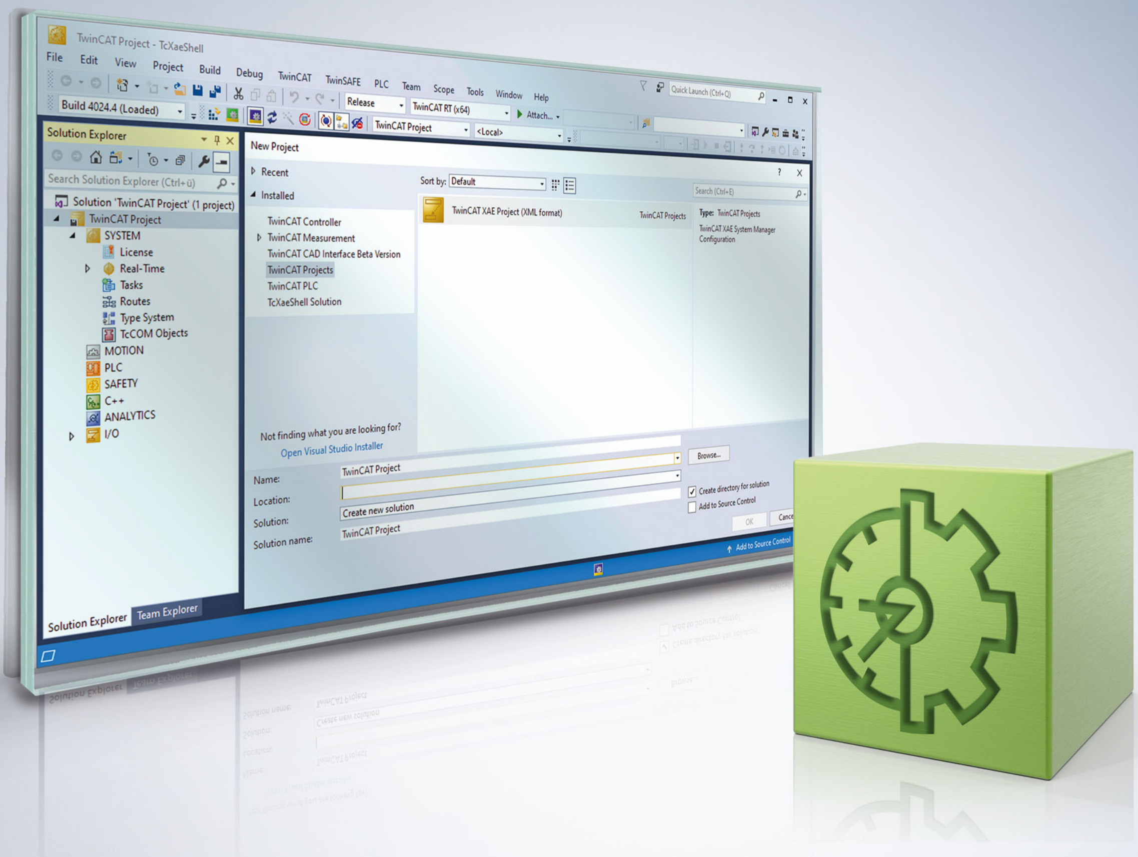
BECKHOFF New Automation Technology

Manual | EN

TE1200

TwinCAT 3 | PLC Static Analysis



2024-04-10 | Version: 2.6.4

Table of contents

1 Foreword

1.1 Notes on the documentation

This description is intended exclusively for trained specialists in control and automation technology who are familiar with the applicable national standards.
For installation and commissioning of the components, it is absolutely necessary to observe the documentation and the following notes and explanations.
The qualified personnel is obliged to always use the currently valid documentation.

The responsible staff must ensure that the application or use of the products described satisfies all requirements for safety, including all the relevant laws, regulations, guidelines, and standards.

Disclaimer

The documentation has been prepared with care. The products described are, however, constantly under development.
We reserve the right to revise and change the documentation at any time and without notice.
No claims to modify products that have already been supplied may be made on the basis of the data, diagrams, and descriptions in this documentation.

Trademarks

Beckhoff®, TwinCAT®, TwinCAT/BSD®, TC/BSD®, EtherCAT®, EtherCAT G®, EtherCAT G10®, EtherCAT P®, Safety over EtherCAT®, TwinSAFE®, XFC®, XTS® and XPlanar® are registered and licensed trademarks of Beckhoff Automation GmbH.
If third parties make use of designations or trademarks used in this publication for their own purposes, this could infringe upon the rights of the owners of the said designations.

Patents

The EtherCAT Technology is covered, including but not limited to the following patent applications and patents:
EP1590927, EP1789857, EP1456722, EP2137893, DE102015105702
and similar applications and registrations in several other countries.

EtherCAT® is registered trademark and patented technology, licensed by Beckhoff Automation GmbH, Germany

Copyright

1.2 For your safety

Safety regulations

Read the following explanations for your safety.
Always observe and follow product-specific safety instructions, which you may find at the appropriate places in this document.

Exclusion of liability

All the components are supplied in particular hardware and software configurations which are appropriate for the application. Modifications to hardware or software configurations other than those described in the documentation are not permitted, and nullify the liability of Beckhoff Automation GmbH & Co. KG.

Personnel qualification

This description is only intended for trained specialists in control, automation, and drive technology who are familiar with the applicable national standards.

Signal words

The signal words used in the documentation are classified below. In order to prevent injury and damage to persons and property, read and follow the safety and warning notices.

Personal injury warnings

⚠ DANGER
Hazard with high risk of death or serious injury.

⚠ WARNING
Hazard with medium risk of death or serious injury.

⚠ CAUTION
There is a low-risk hazard that could result in medium or minor injury.

Warning of damage to property or environment

NOTICE
The environment, equipment, or data may be damaged.

Information on handling the product

This information includes, for example:
recommendations for action, assistance or further information on the product.

1.3 Notes on information security

The products of Beckhoff Automation GmbH & Co. KG (Beckhoff), insofar as they can be accessed online, are equipped with security functions that support the secure operation of plants, systems, machines and networks. Despite the security functions, the creation, implementation and constant updating of a holistic security concept for the operation are necessary to protect the respective plant, system, machine and networks against cyber threats. The products sold by Beckhoff are only part of the overall security concept. The customer is responsible for preventing unauthorized access by third parties to its equipment, systems, machines and networks. The latter should be connected to the corporate network or the Internet only if appropriate protective measures have been set up.

In addition, the recommendations from Beckhoff regarding appropriate protective measures should be observed. Further information regarding information security and industrial security can be found in our https://www.beckhoff.com/secguide.

Beckhoff products and solutions undergo continuous further development. This also applies to security functions. In light of this continuous further development, Beckhoff expressly recommends that the products are kept up to date at all times and that updates are installed for the products once they have been made available. Using outdated or unsupported product versions can increase the risk of cyber threats.

To stay informed about information security for Beckhoff products, subscribe to the RSS feed at https://www.beckhoff.com/secinfo.

2 Overview

With the integration of the static code analysis, a further programming tool is available in TwinCAT 3.1 that supports the PLC software development process.

Static code analysis should be regarded as a supplement to the compiler. It facilitates writing clearer code and can help uncover potential sources of error during programming. For example, it can report if a pointer variable has not been checked for nonzero before dereferencing. As a result, the user's attention is drawn to possibly inadvertent and erroneous implementations, so that these program points can be optimized at an early stage.

Static Analysis is integrated into TwinCAT 3 PLC as a programming tool. It checks the source code of a PLC project for deviations from certain coding rules, naming conventions or unauthorized symbols. The rule set defined in the PLCopen Coding Guidelines is used as the basis, complemented by additional checking options.

The Static Analysis can be triggered manually or performed automatically during the code generation. TwinCAT outputs the result of the analysis, i.e. messages regarding deviations from the specifications and rules, in the message window. In the PLC project properties you can define the parameters to be checked in detail. When configuring the rules, you can also define whether a rule violation is to be output as an error or a warning. You can use pragma instructions statements to exclude particular parts of the code from the check. For errors reported by Static Analysis based on precompile information, there is support in the ST Editor for immediate troubleshooting (QuickFix/Precompile [▸ 119]).

In addition, you can display selected metrics to assess code quality in a separate view. Static Analysis determines these metrics from your program code. Key parameters are calculated that characterize the various program parts or express the properties of the software. They therefore provide an indication of the software quality. For example, the tabular output contains metrics for the number of statements or the proportion of comments.

Advantage

"Static code analysis" facilitates writing clearer code and can help uncover potential sources of error during programming.

Failure to observe a coding rule generally indicates an implementation weakness; correcting it enables early troubleshooting or error avoidance. The automatic control of the user-specific naming conventions also ensures that the control programs can be developed in a standardized manner with regard to type and variable names. This gives different PLC projects implemented on the basis of the same naming conventions a uniform look and feel, which greatly improves the readability of programs. In addition, the metrics provide an indication of the software quality.

Functionalities

An overview of the functionalities of "TwinCAT 3 PLC Static Analysis" is provided below:

- Static Analysis:
    - Function: The Static Analysis checks the source code of a project for deviations from certain coding rules and naming conventions, as well as for forbidden symbols. The result is output in the message window.
    - Configuration: The required coding rules, naming conventions and forbidden symbols can be configured in the Rules [▸ 15], Naming conventions [▸ 80] and [▸ 98] tabs of the PLC project properties.
- Standard metrics:
    - Function: Certain metrics are applied to your source code, which express the software properties in the form of indicators (e.g. the number of statements or the proportion of comments). They provide an indication of the software quality. The results are output in the **Standard Metrics** view.
    - Configuration: The required metrics can be configured in the Metrics [▸ 93] tab of the PLC project properties.

Alternatively, there is an option to use a license-free version of Static Analysis that provides a very much reduced range of functions. A detailed comparison of the functions of the license-free and licensed versions of Static Analysis can be found in the chapter Installation [▶ 10].

Further information on installation, configuration and execution of the "Static Analysis" can be found on the following pages:

- Installation [▶ 10]
- Configuration of the settings, rules, naming conventions, metrics and forbidden symbols [▶ 13]
- Command 'Run static analysis' [▶ 100]
- Command 'Run static analysis [Check all objects]' [▶ 102]
- Command 'View Standard Metrics' [▶ 103]
- Command 'View Standard Metrics [Check all objects]' [▶ 105]
- Pragmas and attributes [▶ 108]
- Examples [▶ 113]
- Automation Interface support [▶ 116]

Libraries

TwinCAT only analyzes the application code of the current PLC project; the referenced libraries are ignored!

If you have opened the library project, however, you can check the elements it contains with the help of the command Command 'Run static analysis [Check all objects]' [▶ 102].

Punctual disablement of checks

Pragmas and attributes [▶ 108] can be used to disable checks for certain parts of the code.

Static Analysis via the Automation Interface

Static Analysis can be operated via the Automation Interface (see Automation Interface support [▶ 116]).

3 Installation

The TE1200 | TwinCAT 3 PLC Static Analysis function is installed together with the TwinCAT 3 development environment and has been included as a release version since TwinCAT version 3.1 build 4022.0. Therefore, only the additional TE1200 engineering component needs to be licensed.

Licensing

For information on licensing the TE1200 engineering component, please read the licensing documentation.

Test mode

Please note that there is no 7-day trial license available for this product. If you do not have an Engineering license for TE1200 you can use the license-free version of Static Analysis (Static Analysis Light), which has some restrictions (see below). The free Light version enables you to familiarize yourself with the basic handling of the product, for example, based on a heavily reduced set of functions.

See also: Functionality: Light vs. full [▶ 10]

TwinCAT Package Manager: Installation (TwinCAT 3.1 Build 4026)

Detailed instructions on installing products can be found in the chapter Installing workloads in the TwinCAT 3.1 Build 4026 installation instructions.

Install the following workload to be able to use the product:

- TwinCAT.Standard.XAE (contains TwinCAT.XAE.PLC)

or

- TwinCAT.XAE.PLC

TwinCAT setup: Installation (TwinCAT 3.1 build 4024 and earlier)

Install the following setup in order to be able to use the product:

- TwinCAT 3.1 eXtended Automation Engineering (XAE) (full installation)

Detailed installation instructions can be found in the Installation TwinCAT 3.1 Build 4024 chapter.

3.1 Functionality: Light vs. full

If you do not have an Engineering license for TE1200 you can use the license-free version of Static Analysis (Static Analysis Light), which has some restrictions (see table below). The free Light version enables you to familiarize yourself with the basic handling of the product, for example, based on a heavily reduced set of functions.

Static Analysis Light vs. Static Analysis Full

An overview of the different features of the license-free and license-managed variants of Static Analysis is provided below.

Functional aspect	**Static Analysis Light (without TE1200 license)**	**Static Analysis Full (with TE1200 license)**
License required	No, usable free of charge	Yes, TE1200 license required
Save/export and load/import (rule) configuration	Not possible, coupled to PLC project properties	Possible (using the **Load/Save** buttons in the Settings [▶ 13])
Execution is coupled to the compilation process	Yes, not configurable	Configurable (using the **Perform static analysis automatically** option in the Settings [▶ 13]; Manual execution with the help of the command Command 'Run static analysis' [▶ 100])
Checking for unused objects (e.g. within a library project)	Not possible	Possible (with the help of the command Command 'Run static analysis [Check all objects]' [▶ 102])
Maximum number of reported errors	500 (not configurable) (Further information on the significance of 500 as the maximum number of errors can be found in the Settings [▶ 13])	Configurable (using the setting **Maximum number of errors** in the Settings [▶ 13])
Maximum number of reported warnings	Output of warnings not possible (see following line)	Configurable (using the setting **Maximum number of warnings** in the Settings [▶ 13])
Rules: Activation options [▶ 15]	• Active and output as error • Inactive	• Active and output as error • Active and output as warning • Inactive
Rules: scope [▶ 16]	7 coding rules • SA0033: Unused variables • SA0028: Overlapping memory areas • SA0006: Write access to multiple tasks • SA0004: Multiple writes access on output • SA0027: Multiple usage of name • SA0167: Report temporary FunctionBlock instances • SA0175: Suspicious operation on string	More than 100 coding rules
Rules: Precompile wavy underline, QuickFix [▶ 119]	Not available	Available
Naming conventions [▶ 80]	Not available	Available
Metrics [▶ 93]	Not available	Available
Forbidden symbols [▶ 98]	Not available	Available

Pragmas and attributes [▶ 108] for temporary deactivation of rules	Yes, available in the Light scope: • Pragma {analysis ...} • Attribute {attribute 'no-analysis'} • Attribute {attribute 'analysis' := '...'}	Yes, available in full scope: • Pragma {analysis ...} • Attribute {attribute 'no-analysis'} • Attribute {attribute 'analysis' := '...'} • Attribute {attribute 'naming' := '...'} • Attribute {attribute 'nameprefix' := '...'} • Attribute {attribute 'analysis:report-multiple-instance-calls'}

4 Configuration

After the installation [▸ 10] and licensing of "TE1200 | TwinCAT 3 PLC Static Analysis", the category **Static Analysis** in the properties of the PLC project is extended by the additional rules and configuration options.

In the project properties you will then find tabs for the basic configuration and for configuring the rules, conventions, metrics and forbidden symbols, which have to be taken into account in the code analysis.

The properties of a PLC project can be opened via the context menu of PLC project object or via the **Project** menu, if the focus is on a PLC project in the project tree.

The current settings or modifications are saved when you save the PLC project properties. The **Save** button, which can be found in the **Settings** tab, can be used to save the current Static Analysis configuration additionally in an external file. Such a configuration file can be loaded into the development environment via the **Load** button.

The following pages contain further information on the individual tabs of the **Static Analysis** project properties category.

- Settings [▸ 13]
- Rules [▸ 15]
- Naming conventions [▸ 80]
- Naming conventions (2) [▸ 90]
- Metrics [▸ 93]
- Forbidden symbols [▸ 98]

Scope of the "Static Analysis" configuration

The parameters you set in the category **Static Analysis** of the PLC project properties are referred to as **Solution options** and therefore affect not only the PLC project whose properties you currently edit. The configured settings, rules, naming conventions, metrics and forbidden symbols are applied to all PLC projects in the development environment.

4.1 Settings

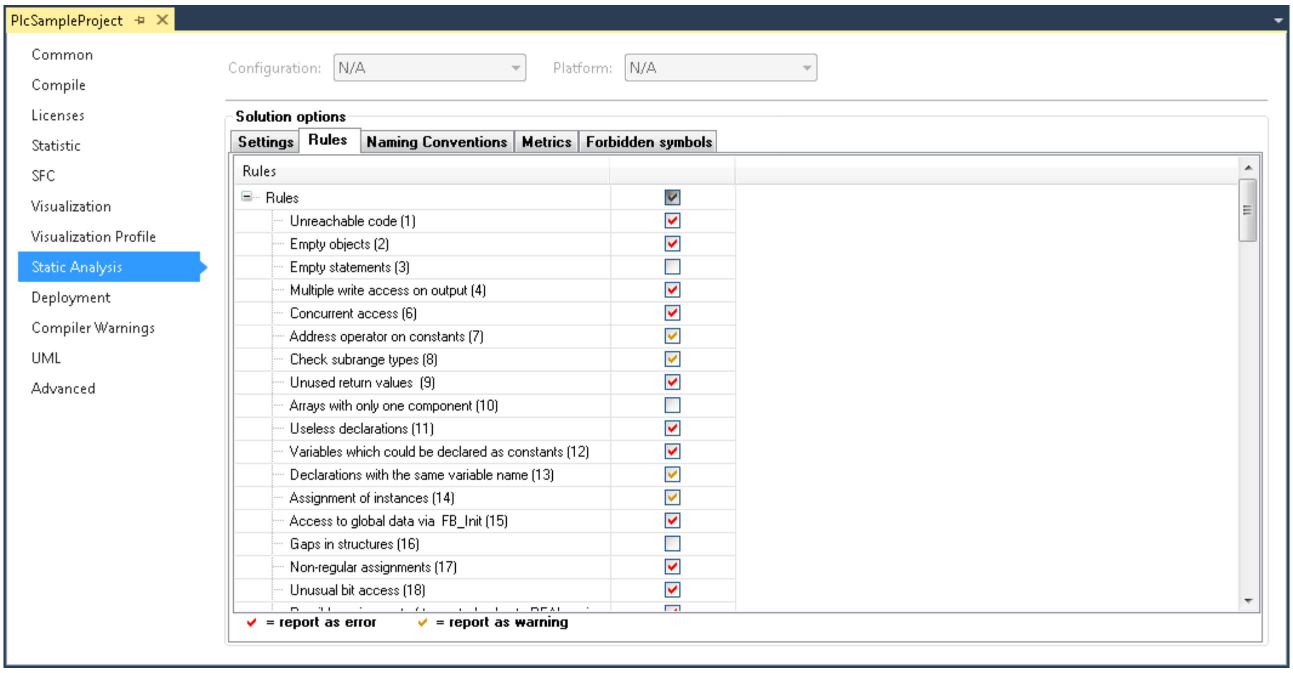
The **Settings** tab can be used to configure whether the static code analysis is automatically performed when the code is generated. The current configuration of the **Static Analysis** can be saved in an external file, or a configuration can be loaded from an external file.

Perform Static Analysis automatically after compilation	If this option is enabled, TwinCAT performs the Static Analysis whenever code is generated without error (e.g. when the command **Build Project** is executed). The analysis can be started manually via the command Command 'Run static analysis' [▶ 100], irrespective of the configuration of this option.
Load	This button opens the standard dialog for a locating of a file. Select the required configuration file *.csa for the Static Analysis, which may previously have been created via **Save** (see below). Since the Static Analysis properties are "solution options", the project properties for the Static Analysis, as described in the csa file, are applied to all PLC projects in the development environment.
Save	This button is used to save the current project properties for the Static Analysis in an xml file. The standard dialog for saving a file appears, and the file type is preset to "Static analysis files" (*.csa). Such a file can later be applied to the project via the **Load** button (see above). Please note that the setting of the error limit "Maximum number of errors" is not saved in this file.
Maximum number of errors	Preset: 500 In this box you can enter the desired error limit, which is checked during the execution of the Static Analysis. If either the **error limit or** the **warning limit** (see below) is reached, **execution** of the Static Analysis is **canceled** and the **previous analysis result** is **output**. **Performance vs. completeness:** Please note: The more objects are checked by the Static Analysis, the longer the execution of the Static Analysis takes. And the more errors are entered in the output window, the longer the results output of the Static Analysis takes. In the assumed case that there are more than 500 Static Analysis errors in a PLC project, the following use cases arise. • Use of a small error limit (e.g. 500): You wish to gradually process the output errors by correcting the respective program code and executing the Static Analysis again to check the correction. In this case it wouldn't be necessary to check all the objects at once and to display all the errors at once. Instead, it is usually sufficient in this case to display a subset as the Static Analysis result, wherein the Static Analysis is executed with a good performance. • Use of a large error limit (e.g. 5000): You wish to output a total report from the Static Analysis in order to be able to roughly estimate the total work required for the correction of the program code. You can attain this goal by increasing the error limit. Please note that, depending on the project situation, the execution of the Static Analysis takes (much) longer the higher the error limit is set. **Detailed explanation of the behavior:** If there are more than 500 Static Analysis errors in a project, then configuring the error limit to 500 does not mean that the Static Analysis outputs exactly 500 errors. In fact, the following happens during the execution of the Static Analysis: Before checking a further POU, a check is performed to see whether the Static Analysis errors found so far already exceed the configured limit. If this is the case, the execution of the Static Analysis is aborted and the analysis result so far is output. If on the other hand the limit has not been reached, this POU is checked by the Static Analysis and the errors found in this POU are added to the analysis result. If this newly formed error total (e.g. 530) exceeds the configured error limit, the execution of the Static Analysis is aborted before the checking of the next POU and the errors found so far (e.g. 530) are output.

Maximum number of warnings	Preset: 500 In this field, you can enter the desired warning limit, which is checked during the execution of Static Analysis. If either the **error limit** (see above) **or** the **warning limit** is reached, **execution** of the Static Analysis is **canceled** and the **previous analysis result** is **output**. Further information on the use cases can be found in the description of the "Maximum number of errors" option (see above).

4.2 Rules

In the **Rules** tab you can configure the rules that are taken into account when the static analysis is performed [▶ 100]. The rules are displayed as a tree structure in the project properties. Some rules are arranged below organizational nodes.



Configuration: N/A Platform: N/A

Solution options

Rules	
Rules	■
Unreachable code (1)	✓
Empty objects (2)	✓
Empty statements (3)	☐
Multiple write access on output (4)	✓
Concurrent access (6)	✓
Address operator on constants (7)	☑
Check subrange types (8)	☑
Unused return values (9)	✓
Arrays with only one component (10)	☐
Useless declarations (11)	✓
Variables which could be declared as constants (12)	✓
Declarations with the same variable name (13)	☑
Assignment of instances (14)	☑
Access to global data via FB_Init (15)	✓
Gaps in structures (16)	☐
Non-regular assignments (17)	✓
Unusual bit access (18)	✓

✓ = report as error ☑ = report as warning

Default settings

All rules are enabled by default, with the exception of SA0016, SA0024, SA0073, SA0101, SA0105-SA0107, SA0111-SA0125, SA0133, SA0134, SA0145, SA0147, SA0148, SA0150, SA0162-SA0167 and the "strict" IEC rules.

Configuring the rules

Individual rules can be enabled or disabled via the checkbox for the respective row. Ticking the checkbox for a subnode affects all entries below this node. Ticking the checkbox for the top node affects all list entries. The entries below a node can be collapsed or expanded by clicking on the minus or plus sign to the left of the node name.

The number in brackets after each rule, for example "Unreachable code (1)", is the rule number that is issued if the rule is not observed.

The following three settings are available, which can be accessed by repeated clicking on the checkbox:

- ☐ : The rule is not checked.
- ✓ : A rule violation results in an error being reported in the message window.
- ☑ : A rule violation results in a warning being reported in the message window.

Syntax of rule violations in the message window

Each rule has a unique number (shown in parentheses after the rule in the rule configuration view). If a rule violation is detected during the static analysis, the number together with an error or warning description is issued in the message window, based on the following syntax. The abbreviation "SA" stands for "Static Analysis".

Syntax: **"SA<rule number>: <rule description>"**

Sample for rule number 33 (unused variables): "SA0033: Not used: variable 'bSample'"

Temporary deactivation of rules

Rules that are enabled in this dialog can be temporarily disabled in the project via a pragma. For further information please refer to Pragmas and attributes [▸ 108].

Overview and description of the rules

An overview and a detailed description of the rules can be found under Rules - overview and description [▸ 16].

4.2.1 Rules - overview and description

Check strict IEC rules

The checks under the node "Check strict IEC rules" determine functionalities and data types that are allowed in TwinCAT, in extension of IEC61131-3.

Checking concurrent/competing access

The following rules exist on this topic:

SA0006: Write access from multiple tasks [▸ 22]
Determines variables that are written to by more than one task.

SA0103: Concurrent access on not atomic data [▸ 59]
Determines non-atomic variables (for example with data types STRING, WSTRING, ARRAY, STRUCT, FB instances, 64-bit data types) that are used in more than one task.

Please note that only direct access can be recognized. Indirect access operations, for example via pointer/reference, are not listed.

Please also refer to the documentation on the subject "Multi-task data access synchronization in the PLC", which contains several notes on the necessity and options for data access synchronization.

Overview

- SA0001: Unreachable code [▸ 20]

- SA0002: Empty objects [▸ 21]

- SA0003: Empty statements [▸ 21]

- SA0004: Multiple writes access on output [▸ 21]

- SA0006: Write access from several tasks [▸ 22]

- SA0007: Address operators on constants [▸ 23]

- SA0008: Check subrange types [▸ 23]

- SA0009: Unused return values [▶ 24]
- SA0010: Arrays with only one component [▶ 24]
- SA0011: Useless declarations [▶ 25]
- SA0012: Variables which could be declared as constants [▶ 25]
- SA0013: Declarations with the same variable name [▶ 25]
- SA0014: Assignments of instances [▶ 26]
- SA0015: Access to global data via FB_init [▶ 26]
- SA0016: Gaps in structures [▶ 27]
- SA0017: Non-regular assignments [▶ 27]
- SA0018: Unusual bit access [▶ 28]
- SA0020: Possibly assignment of truncated value to REAL variable [▶ 28]
- SA0021: Transporting the address of a temporary variable [▶ 29]
- SA0022: (Possibly) non-rejected return values [▶ 29]
- SA0023: Complex return values [▶ 30]
- SA0024: Untyped literals/constants [▶ 30]
- SA0025: Unqualified enumeration constants [▶ 30]
- SA0026: Possible truncated strings [▶ 31]
- SA0027: Multiple usage of name [▶ 31]
- SA0028: Overlapping memory areas [▶ 32]
- SA0029: Notation in code different to declaration [▶ 32]
- **List unused objects**
    - SA0031: Unused signatures [▶ 32]
    - SA0032: Unused enumeration constants [▶ 33]
    - SA0033: Unused variables [▶ 33]
    - SA0035: Unused input variables [▶ 33]
    - SA0036: Unused output variables [▶ 34]
- SA0034: Enumeration variables with incorrect assignment [▶ 34]
- SA0037: Write access to input variable [▶ 35]
- SA0038: Read access to output variable [▶ 35]
- SA0040: Possible division by zero [▶ 36]
- SA0041: Possibly loop-invariant code [▶ 36]
- SA0042: Usage of different access paths [▶ 36]
- SA0043: Use of a global variable in only one POU [▶ 37]

- SA0044: Declarations with reference to interface [▸ 37]
- **Conversions**
    - SA0019: Implicit pointer conversions [▸ 38]
    - SA0130: Implicit expanding conversions [▸ 39]
    - SA0131: Implicit narrowing conversions [▸ 39]
    - SA0132: Implicit signed/unsigned conversions [▸ 40]
    - SA0133: Explicit narrowing conversions [▸ 40]
    - SA0134: Explicit signed/unsigned conversions [▸ 41]
- **Usage of direct addresses**
    - SA0005: Invalid addresses and data types [▸ 41]
    - SA0047: Access to direct addresses [▸ 42]
    - SA0048: AT declarations on direct addresses [▸ 42]
- **Rules for operators**
    - SA0051: Comparison operators on BOOL variables [▸ 42]
    - SA0052: Unusual shift operation [▸ 43]
    - SA0053: Too big bitwise shift [▸ 43]
    - SA0054: Comparisons of REAL/LREAL for equality/inequality [▸ 44]
    - SA0055: Unnecessary comparison operations of unsigned operands [▸ 44]
    - SA0056: Constant out of valid range [▸ 45]
    - SA0057: Possible loss of decimal points [▸ 46]
    - SA0058: Operations of enumeration variables [▸ 46]
    - SA0059: Comparison operations always returning TRUE or FALSE [▸ 47]
    - SA0060: Zero used as invalid operand [▸ 48]
    - SA0061: Unusual operation on pointer [▸ 48]
    - SA0062: Using TRUE and FALSE in expressions [▸ 48]
    - SA0063: Possibly not 16-bit-compatible operations [▸ 49]
    - SA0064: Addition of pointer [▸ 49]
    - SA0065: Incorrect pointer addition to base size [▸ 50]
    - SA0066: Use of temporary results [▸ 51]
- **Rules for statements**
    - **FOR statements**
        - SA0072: Invalid uses of counter variable [▸ 52]
        - SA0073: Use of non-temporary counter variable [▸ 52]
        - SA0080: Loop index variable for array index exceeds array range [▸ 52]

- SA0081: Upper border is not a constant [▸ 53]
- **CASE statements**
    - SA0075: Missing ELSE [▸ 54]
    - SA0076: Missing enumeration constant [▸ 54]
    - SA0077: Type mismatches with CASE expression [▸ 55]
    - SA0078: Missing CASE branches [▸ 55]
- SA0090: Return statement before end of function [▸ 56]

- SA0095: Assignments in conditions [▸ 56]
- SA0100: Variables greater than <n> bytes [▸ 57]
- SA0101: Names with invalid length [▸ 58]
- SA0102: Access to program/fb variables from the outside [▸ 58]
- SA0103: Concurrent access on not atomic data [▸ 59]
- SA0105: Multiple instance calls [▸ 60]
- SA0106: Virtual method calls in FB_init [▸ 61]
- SA0107: Missing formal parameters [▸ 62]
- **Check strict IEC rules**
    - SA0111: Pointer variables [▸ 63]
    - SA0112: Reference variables [▸ 63]
    - SA0113: Variables with data type WSTRING [▸ 63]
    - SA0114: Variables with data type LTIME [▸ 63]
    - SA0115: Declarations with data type UNION [▸ 64]
    - SA0117: Variables with data type BIT [▸ 64]
    - SA0119: Object-oriented features [▸ 64]
    - SA0120: Program calls [▸ 65]
    - SA0121: Missing VAR_EXTERNAL declarations [▸ 65]
    - SA0122: Array index defined as expression [▸ 66]
    - SA0123: Usages of INI, ADR or BITADR [▸ 66]
    - SA0147: Unusual shift operation - strict [▸ 66]
    - SA0148: Unusual bit access - strict [▸ 67]
- **Rules for initializations**
    - SA0118: Initializations not using constants [▸ 67]
    - SA0124: Dereference access in initializations [▸ 68]
    - SA0125: References in initializations [▸ 69]
- SA0140: Statements commented out [▸ 72]

- Possible use of uninitialized variables

- SA0039: Possible null pointer dereferences [▸ 69]
- SA0046: Possible use of not initialized interface [▸ 70]
- SA0145: Possible use of not initialized reference [▸ 71]

- SA0150: Violations of lower or upper limits of the metrics [▸ 72]

- SA0160: Recursive calls [▸ 72]

- SA0161: Unpacked structure in packed structure [▸ 74]

- SA0162: Missing comments [▸ 74]

- SA0163: Nested comments [▸ 75]

- SA0164: Multi-line comments [▸ 75]

- SA0166: Maximum number of input/output/VAR_IN_OUT variables [▸ 76]

- SA0167: Report temporary FunctionBlock instances [▸ 76]

- SA0168: Unnecessary assignments [▸ 77]

- SA0169: Ignored outputs [▸ 77]

- SA0170: Address of an output variable should not be used [▸ 78]

- SA0171: Enumerations should have the 'strict' attribute [▸ 78]

- SA0175: Suspicious operation on string [▸ 79]

Detailed description

SA0001: Unreachable code

Function	Determines code that is not executed, for example due to a RETURN or CONTINUE statement.
Reason	Unreachable code should be avoided in any case. The check often indicates the presence of test code, which should be removed.
Importance	High
PLCopen rule	CP2

Sample 1 – RETURN:

```
PROGRAM MAIN
VAR
    bReturnBeforeEnd : BOOL;
END_VAR
```

```
bReturnBeforeEnd := FALSE;
RETURN;
bReturnBeforeEnd := TRUE;        // => SA0001
```

Sample 2 – CONTINUE:

```
FUNCTION F_ContinueInLoop : BOOL
VAR
    nCounter  : INT;
END_VAR
```

```
F_ContinueInLoop := FALSE;

FOR nCounter := INT#0 TO INT#5 BY INT#1 DO
```

```
    CONTINUE;
    F_ContinueInLoop := FALSE;    // => SA0001
END_FOR
```

SA0002: Empty objects

Function	Determines POUs, GVLs or data type declarations that do not contain code.
Reason	Empty objects should be avoided. They are often a sign that an object is not fully implemented. Exception: In some cases, the body of a function block will not assigned code if it is only to be used via interfaces. In other cases, a method is only created because it is required by an interface, without scope for meaningful implementation of the method. In any case, a comment should be included in such a situation.
Importance	Medium

SA0003: Empty statements

Function	Determines lines of code containing a semicolon (;) but no statement.
Reason	An empty statement can be an indication of missing code.
Exception	Although there are meaningful uses for empty statements. For example, it may be useful to explicitly program all cases in a CASE statement, including cases in which no action is required. If such an empty CASE statement is commented, the statistical code analysis does not generate an error message.
Importance	Low

Samples:

```
;                                 // => SA0003
(* comment *);                    // => SA0003
nVar;                             // => SA0003
```

The following sample generates the error "SA0003: Empty statement" for State 2.

```
CASE nVar OF
    1: DoSomething();
    2: ;
    3: DoSomethingElse();
END_CASE
```

The following sample does not generate an SA0003 error.

```
CASE nVar OF
    1: DoSomething();
    2: ;                          // nothing to do
    3: DoSomethingElse();
END_CASE
```

SA0004: Multiple writes access on output

Function	Determines outputs that are written at more than one position.
Reason	The maintainability suffers if an output is written in various places in the code. It is then unclear which write access is actually affecting the process. It is good practice to perform the calculation of the output variables in auxiliary variables and to assign the calculated value to a point at the end of the cycle.
Exception	No error is issued if an output variable is written in different branches of IF or CASE statements.
Importance	High
PLCopen rule	CP12

This rule **cannot** be disabled via a pragma or attribute!
For more information on attributes, see Pragmas and attributes [▶ 108].

Sample:

Global variable list:

```
VAR_GLOBAL
    bVar     AT%QX0.0 : BOOL;
    nSample  AT%QW5   : INT;
END_VAR
```

Program MAIN:

```
PROGRAM MAIN
VAR
    nCondition        : INT;
END_VAR
```

```
IF nCondition < INT#0 THEN
    bVar    := TRUE;              // => SA0004
    nSample := INT#12;            // => SA0004
END_IF

CASE nCondition OF
    INT#1:
        bVar := FALSE;            // => SA0004

    INT#2:
        nSample := INT#11;        // => SA0004

ELSE
     bVar    := TRUE;             // => SA0004
     nSample := INT#9;            // => SA0004
END_CASE
```

SA0006: Write access from several tasks

Function	Determines variables with write access from more than one task.
Reason	A variable that is written in several tasks may change its value unexpectedly under certain circumstances. This can lead to confusing situations. String variables and, on some 32-bit systems, 64-bit integer variables also may even assume an inconsistent state if the variable is written in two tasks at the same time.
Exception	In certain cases it may be necessary for several tasks to write a variable. Make sure, for example through the use of semaphores, that the access does not lead to an inconsistent state.
Importance	High
PLCopen rule	CP10

See also rule SA0103 [▶ 59].

Call corresponds to write access

Please note that calls are interpreted as write access. For example, calling a method for a function block instance is regarded as a write access to the function block instance. A more detailed analysis of accesses and calls is not possible, e.g. due to virtual calls (pointers, interface).

To deactivate rule SA0006 for a variable (e.g. for a function block instance), the following attribute can be inserted above the variable declaration: {attribute 'analysis' := '-6'}

Samples:

The two global variables nVar and bVar are written by two tasks.

Global variable list:

```
VAR_GLOBAL
    nVar   : INT;
    bVar   : BOOL;
END_VAR
```

Program MAIN_Fast, called from the task PlcTaskFast:

```
nVar := nVar + 1;                 // => SA0006
bVar := (nVar > 10);              // => SA0006
```

Program MAIN_Slow, called from the task PlcTaskSlow:

```
nVar := nVar + 2;                 // => SA0006
bVar := (nVar < -50);             // => SA0006
```

SA0007: Address operators on constants

Function	Determines locations at which the ADR operator is used for a constant.
Reason	A pointer to a constant variable cancels the CONSTANT property of the variable. The variable can be changed via the pointer without the compiler reporting this.
Exception	In rare cases, it may make sense for pointer to a constant to be passed to a function. If this option is used, measures must be implemented to ensure that the function does not change the value that was passed to it. In this case, use VAR_IN_OUT CONSTANT if possible.
Importance	High

If the option **Replace constants** is enabled in the compiler options of the PLC project properties, the address operator for scalar constants (Integer, BOOL, REAL) is not allowed and a compilation error is issued. (Constant strings, structures and arrays always have an address.)

Sample:

```
PROGRAM MAIN
VAR CONSTANT
    cValue   : INT := INT#15;
END_VAR
VAR
    pValue   : POINTER TO INT;
END_VAR
```

```
pValue := ADR(cValue);            // => SA0007
```

SA0008: Check subrange types

Function	Determines range exceedances of subrange types. Assigned literals are checked at an early stage by the compiler. If constants are assigned, the values must be within the defined range. If variables are assigned, the data types must be identical.
Reason	If subrange types are used, make sure that the function remains within the respective subrange. The compiler checks such subrange violations only for assignments of constants.
Importance	Low

The check is not performed for CFC objects, because the code structure does not allow this.

Sample:

```
PROGRAM MAIN
VAR
    nSub1   : INT (INT#1..INT#10);
    nSub2   : INT (INT#1..INT#1000);
    nVar    : INT;
END_VAR
```

```
nSub1  := nSub2;                  // => SA0008
nSub1  := nVar;                   // => SA0008
```

SA0009: Unused return values

Function	Determines function, method and property calls for which the return value is not used.
Reason	If a function or method returns a return value, the value should be evaluated. In many cases the return value contains information to indicate whether the function was executed successfully. If no evaluation is performed, it is subsequently not possible to determine whether the return value was overlooked or whether it is in fact not required.
Exception	If a return value is of no interest during a call, this can be documented and the assignment can be omitted. Error returns should never be ignored!
Importance	Medium
PLCopen rule	CP7/CP17

Sample:

Function F_ReturnBOOL:

```
FUNCTION F_ReturnBOOL : BOOL
```

```
F_ReturnBOOL := TRUE;
```

Program MAIN:

```
PROGRAM MAIN
VAR
    bVar   : BOOL;
END_VAR
```

```
F_ReturnBOOL();                   // => SA0009
bVar  := F_ReturnBOOL();
```

SA0010: Arrays with only one component

Function	Determines arrays containing only a single component.
Reason	An array with a component can be replaced by a Base Type variable. Access to such a variable is much faster than access to a variable via an index.
Exception	The length of an array is often determined by a constant and used as a parameter for a program. The program can then work with arrays of different lengths and does not have to be changed if the length is only 1. Such a situation should be documented accordingly.
Importance	Low

Samples:

```
PROGRAM MAIN
VAR
    aEmpty1   : ARRAY [0..0] OF INT;                // => SA0010
    aEmpty2   : ARRAY [15..15] OF REAL;             // => SA0010
END_VAR
```

SA0011: Useless declarations

Function	Determines structures, unions or enumerations with only a single component.
Reason	Such a declaration can be confusing for a reader. A structure with only one element can be replaced by an alias type. An enumeration with an element can be replaced by a constant.
Importance	Low
PLCopen rule	CP22/CP24

Sample 1 – Structure:

```
TYPE ST_SingleStruct :              // => SA0011
STRUCT
    nPart   : INT;
END_STRUCT
END_TYPE
```

Sample 2 – Union:

```
TYPE U_SingleUnion :                // => SA0011
UNION
    fVar   : LREAL;
END_UNION
END_TYPE
```

Sample 3 – Enumeration:

```
TYPE E_SingleEnum :                 // => SA0011
(
    eOnlyOne := 1
);
END_TYPE
```

SA0012: Variables which could be declared as constants

Function	Determines variables that are not subject to write access and therefore could not be declared as constants.
Reason	If a variable is only written at the declaration point and is otherwise only used in read mode, the static analysis assumes that the variable is to remain unchanged. Declaration as a constant means that the variable is checked for changes in the event of program modifications. Plus, declaration as a constant may lead to faster code.
Importance	Low

Sample:

```
PROGRAM MAIN
VAR
    nSample   : INT := INT#17;
    nVar      : INT;
END_VAR
```

```
nVar := nVar + nSample;             // => SA0012
```

SA0013: Declarations with the same variable name

Function	Determines variables with the same name as other variables (example: global and local variables with the same name), or the same name as functions, actions, methods or properties within the same access range.
Reason	Identical names can be confusing when the code is read and can lead to errors if the wrong object is accessed accidentally. We therefore recommend using naming conventions that avoid such situations.
Importance	Medium
PLCopen rule	N5/N9

Samples:

Global variable list GVL_App:

```
VAR_GLOBAL
    nVar   : INT;
END_VAR
```

MAIN program, containing a method with the name Sample:

```
PROGRAM MAIN
VAR
    bVar     : BOOL;
    nVar     : INT;                // => SA0013
    Sample   : DWORD;              // => SA0013
END_VAR
```

```
.nVar := 100;                    // Writing global variable "nVar"
nVar  := 500;                    // Writing local variable "nVar"
```

```
METHOD Sample
VAR_INPUT
…
```

SA0014: Assignments of instances

Function	Determines assignments to function block instances. For instances with pointer or reference variables such assignments may be risky.
Reason	This is a performance warning. When an instance is assigned to another instance, all elements and subelements are copied from one instance to the other. Pointers to data are also copied, but not their referenced data, so that the target instance and the source instance contain the same data after the assignment. Depending on the size of the instances, such an assignment may take a long time. If, for example, an instance is to be passed to a function for processing, it is much better to pass a pointer to the instance. A copy method can be useful for selectively copying values from one instance to another: `fb2.CopyFrom(fb1)`
Importance	Medium

Sample:

```
PROGRAM MAIN
VAR
    fb1   : FB_Sample;
    fb2   : FB_Sample;
END_VAR
```

```
fb1();
fb2 := fb1;                      // => SA0014
```

SA0015: Access to global data via FB_init

Function	Determines access of a function block to global data via the FB_init method. The value of this variables depends on the order of the initializations!
Reason	Depending on the declaration location of the instance of a function block, a non-initialized variable may be accessed if the rule is violated.
Importance	High

Sample:

Global variable list GVL_App:

```
VAR_GLOBAL
    nVar      : INT;
END_VAR
```

Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR
    nLocal    : INT;
END_VAR
```

Method FB_Sample.FB_init:

```
METHOD FB_init : BOOL
VAR_INPUT
    bInitRetains  : BOOL;        // if TRUE, the retain variables are initialized (warm start / cold
start)
    bInCopyCode   : BOOL;        // if TRUE, the instance afterwards gets moved into the copy code
(online change)
END_VAR
```

```
nLocal := 2*nVar;                // => SA0015
```

Program MAIN:

```
PROGRAM MAIN
VAR
    fbSample   : FB_Sample;
END_VAR
```

SA0016: Gaps in structures

Function	Determines gaps in structures or function blocks, caused by the alignment requirements of the currently selected target system. If possible, you should remove gaps by rearranging the structure elements or by filling them with dummy elements. If this is not possible, you can disable the rule for the affected structures using the attribute {attribute 'analysis' := '...'} [▶ 110].
Reason	Due to different alignment requirements on different platforms, such structures may have a different layout in the memory. The code may behave differently, depending on the platform.
Importance	Low

Samples:

```
TYPE ST_UnpaddedStructure1 :
STRUCT
    bBOOL   : BOOL;
    nINT    : INT;               // => SA0016
    nBYTE   : BYTE;
    nWORD   : WORD;
END_STRUCT
END_TYPE
```

```
TYPE ST_UnpaddedStructure2 :
STRUCT
    bBOOL   : WORD;
    nINT    : INT;
    nBYTE   : BYTE;
    nWORD   : WORD;              // => SA0016
END_STRUCT
END_TYPE
```

SA0017: Non-regular assignments

Function	Determines assignments to pointers, which are not an address (ADR operator, pointer variables) or constant 0.
Reason	If a pointer contains a value that is not a valid address, an access violation exception occurs when dereferencing the pointer.
Importance	High

Sample:

```
PROGRAM MAIN
VAR
    nVar      : INT;
    pInt      : POINTER TO INT;
    nAddress  : XWORD;
END_VAR
```

```
nAddress := nAddress + 1;

pInt     := ADR(nVar);            // no error
pInt     := 0;                    // no error
pInt     := nAddress;             // => SA0017
```

SA0018: Unusual bit access

Function	Determines bit access to signed variables. However, the IEC 61131-3 standard only permits bit access to bit fields. See also strict rule SA0148 [▶ 67].
Reason	Signed data types should not be used as bit fields and vice versa. The IEC 61131-3 standard does not provide for such access. This rule must be observed if the code is to be portable.
Exception	Exception for flag enumerations: If an enumeration is declared as flag via the pragma attribute {attribute 'flags'}, the error SA0018 is not issued for bit access with OR, AND or NOT operations.
Importance	Medium

Samples:

```
PROGRAM MAIN
VAR
    nINT    : INT;
    nDINT   : DINT;
    nULINT  : ULINT;
    nSINT   : SINT;
    nUSINT  : USINT;
    nBYTE   : BYTE;
END_VAR
```

```
nINT.3     := TRUE;               // => SA0018
nDINT.4    := TRUE;               // => SA0018
nULINT.18  := FALSE;              // no error because this is an unsigned data type
nSINT.2    := FALSE;              // => SA0018
nUSINT.3   := TRUE;               // no error because this is an unsigned data type
nBYTE.5    := FALSE;              // no error because BYTE is a bit field
```

SA0020: Possibly assignment of truncated value to REAL variable

Function	Determines operations on integer variables, during which a truncated value may be assigned to a variable of data type REAL.

Reason	The static code analysis returns an error when the result of an integer calculation is assigned to a REAL or LREAL variable. The programmer should be made aware of a possibly incorrect interpretation of such an assignment:
	`fLEAL := nDINT1 * nDINT2.`
	Since the value range of LREAL is greater than that of DINT, it could be assumed that the result of the calculation is always displayed in LREAL. But this is not the case. The processor calculates the result of the multiplication as an integer and then casts the result to LREAL. An overflow in the integer calculation would be lost. To avoid this problem, the calculation should be performed as a REAL operation:
	`fLREAL := TO_LREAL(nDINT1) * TO_LREAL(nDINT2)`

Importance	High

Sample:

```
PROGRAM MAIN
VAR
    nVar1   : DWORD;
    nVar2   : DWORD;
    fVar    : REAL;
END_VAR
```

```
nVar1  := nVar1 + DWORD#1;
nVar2  := nVar2 + DWORD#2;
fVar   := nVar1 * nVar2;          // => SA0020
```

SA0021: Transporting the address of a temporary variable

Function	Determines assignments of addresses of temporary variables (variables on the stack) to non-temporary variables.
Reason	Local variables of a function or method are created on the stack and exist only while the function or method is processed. If a pointer points to such a variable after processing the method or function, then this pointer can be used to access undefined memory or an incorrect variable in another function. This situation must be avoided in any case.
Importance	High

Sample:

Method FB_Sample.SampleMethod:

```
METHOD SampleMethod : XWORD
VAR
    fVar   : LREAL;
END_VAR
```

```
SampleMethod := ADR(fVar);
```

Program MAIN:

```
PROGRAM MAIN
VAR
    nReturn    : XWORD;
    fbSample   : FB_Sample;
END_VAR
```

```
nReturn  := fbSample.SampleMethod();                 // => SA0021
```

SA0022: (Possibly) unassigned return value

Function	Determines all functions and methods containing an execution thread without assignment to the return value.
Reason	An unassigned return value in a function or method indicates missing code. Even if the return value always has a default value, it is useful to explicitly assign it again in any case, in order to avoid ambiguities.
Importance	Medium

Sample:

```
FUNCTION F_Sample : DWORD
VAR_INPUT
    nIn     : UINT;
END_VAR
VAR
    nTemp   : INT;
END_VAR
```

```
nIn := nIn + UINT#1;

IF (nIn > UINT#10) THEN
    nTemp    := 1;                 // => SA0022
ELSE
    F_Sample := DWORD#100;
END_IF
```

SA0023: Complex return values

Function	Determines complex return values that cannot be returned with a simple register copy of the processor. These include structures and arrays as well as return values of the type STRING (irrespective of the size of storage space occupied).
Reason	This is a performance warning. If large values are returned as a result of a function, method, or property, the processor copies them repeatedly when the code is executed. This can lead to runtime problems and should be avoided if possible. Better performance is achieved if a structured value is passed to a function or method as VAR_IN_OUT and filled in the function or method.
Importance	Medium

Sample:

Structure ST_sample:

```
TYPE ST_Sample :
STRUCT
    n1  : INT;
    n2  : BYTE;
END_STRUCT
END_TYPE
```

Example of functions with return value:

```
FUNCTION F_MyFunction1 : I_MyInterface              // no error
FUNCTION F_MyFunction2 : ST_Sample                  // => SA0023
FUNCTION F_MyFunction3 : ARRAY[0..1] OF BOOL        // => SA0023
```

SA0024: Untyped literals/constants

Function	Determines untyped literals/constants (e.g. `nCount : INT := 10;`).
Reason	TwinCAT assigns the types for literals according to their use. In some cases, this can lead to unexpected situations that are better clarified by a typed literal. Example: `nDWORD := ROL(DWORD#1, i);`
Importance	Low

Sample:

```
PROGRAM MAIN
VAR
    nVar  : INT;
    fVar  : LREAL;
END_VAR
```

```
nVar := 100;                     // => SA0024
nVar := INT#100;                 // no error

fVar := 12.5;                    // => SA0024
fVar := LREAL#12.5;              // no error
```

SA0025: Unqualified enumeration constants

Function	Determines enumeration constants that are not used with a qualified name, i.e. without preceding enumeration name.
Reason	Qualified access makes the code more readable and easier to maintain. Without forcing qualified variable names, an additional enumeration could be inserted when the program is extended. This enumeration contains a constant with the same name as an existing enumeration (see the sample below: "eRed"). In this case there would be an ambiguous access in this piece of code. We recommend using only enumerations that have the {attribute 'qualified-only'}.
Importance	Medium

Sample:

Enumeration E_Color:

```
TYPE E_Color :
(
    eRed,
    eGreen,
    eBlue
);
END_TYPE
```

Program MAIN:

```
PROGRAM MAIN
VAR
    eColor  : E_Color;
END_VAR
```

```
eColor := E_Color.eGreen;        // no error
eColor := eGreen;                // => SA0025
```

SA0026: Possible truncated strings

Function	Determines string assignments and initializations that do not use an adequate string length.
Reason	If strings of different lengths are assigned, a string may be truncated. The result is then not the expected one.
Importance	Medium

Samples:

```
PROGRAM MAIN
VAR
    sVar1  : STRING[10];
    sVar2  : STRING[6];
    sVar3  : STRING[6]  := 'abcdefghi';              // => SA0026
END_VAR
```

```
sVar2 := sVar1;                                      // => SA0026
```

SA0027: Multiple usage of name

Function	Determines multiple use of a variable name/identifier or object name (POU) within the scope of a project. The following cases are covered: • The name of an enumeration constant is identical to the name in another enumeration within the application or in an included library. • The name of a variable is identical to the name of another object in the application or in an included library. • The name of a variable is identical to the name of an enumeration constant in an enumeration in the application or in an included library. • The name of an object is identical to the name of another object in the application or in an included library.
Reason	Identical names can be confusing when reading the code. They can lead to errors if the wrong object is accessed inadvertently. Therefore, define and follow naming conventions in order to avoid such situations.
Exception	Enumerations declared with the 'qualified_only' attribute are exempt from SA0027 checking because their elements can only be accessed in a qualified manner.
Importance	Medium

Sample:

The following sample generates error/warning SA0027, since the library Tc2_Standard is referenced in the project, which provides the function block TON.

```
PROGRAM MAIN
VAR
    ton   : INT;                  // => SA0027
END_VAR
```

SA0028: Overlapping memory areas

Function	Determines the points due to which two or more variables occupy the same storage space.
Reason	If two variables occupy the same storage space, the code may behave very unexpectedly. This must be avoided in all cases. If the use of a value in different interpretations is unavoidable, for example once as DINT and once as REAL, you should define a UNION. Also, via a pointer you can access a value typed otherwise without converting the value.
Importance	High

Sample:

In the following sample both variables use byte 21, i.e. the memory areas of the variables overlap.

```
PROGRAM MAIN
VAR
    nVar1 AT%QB21  : INT;         // => SA0028
    nVar2 AT%QD5   : DWORD;       // => SA0028
END_VAR
```

SA0029: Notation in code different to declaration

Function	Determines the code positions (in the implementation) at which the notation of an identifier differs from the notation in its declaration.
Reason	The IEC 61131-3 standard defines identifiers as not case-sensitive. This means that a variable declared as "varx" can also be used as "VaRx" in the code. However, this can be confusing and misleading and should therefore be avoided.
Importance	Medium

Samples:

Function F_TEST:

```
FUNCTION F_TEST : BOOL
…
```

Program MAIN:

```
PROGRAM MAIN
VAR
    nVar     : INT;
    bReturn  : BOOL;
END_VAR
```

```
nvar    := nVar + 1;              // => SA0029
bReturn := F_Test();              // => SA0029
```

SA0031: Unused signatures

Function	Determines programs, function blocks, functions, data types, interfaces, methods, properties, actions etc., which are not called within the compiled program code.
Reason	Unused objects result in unnecessary project bloat and confusion when the code is read.
Importance	low
PLCopen rule	CP2

SA0032: Unused enumeration constants

Function	Determines enumeration constants that are not used in the compiled program code.
Reason	Unused enumeration constants result in unnecessary enumeration definition bloat and confusion when the program is read.
Importance	Low
PLCopen rule	CP24

Sample:

Enumeration E_Sample:

```
TYPE E_Sample :
(
    eNull,
    eOne,                           // => SA0032
    eTwo
);
END_TYPE
```

Program MAIN:

```
PROGRAM MAIN
VAR
    eSample   : E_Sample;
END_VAR
```

```
eSample  := E_Sample.eNull;
eSample  := E_Sample.eTwo;
```

SA0033: Unused variables

Function	Determines variables that are declared but not used within the compiled program code.
Reason	Unused variables make a program less easy to read and maintain. Unused variables occupy unnecessary memory space and take up unnecessary runtime during the initialization.
Importance	medium
PLCopen rule	CP22/CP24

SA0035: Unused input variables

Function	Determines input variables that are not assigned within the respective function block.
Reason	Unused variables make a program less easy to read and maintain. Unused variables occupy unnecessary memory space and take up unnecessary runtime during the initialization.
Importance	Medium
PLCopen rule	CP24

Sample:

Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR_INPUT
    bIn1   : BOOL;
    bIn2   : BOOL;                  // => SA0035
END_VAR
VAR_OUTPUT
    bOut1 : BOOL;
    bOut2 : BOOL;                   // => SA0036
END_VAR
```

```
bOut1 := bIn1;
```

SA0036: Unused output variables

Function	Determines output variables that are not assigned within the respective function block.
Reason	Unused variables make a program less easy to read and maintain. Unused variables occupy unnecessary memory space and take up unnecessary runtime during the initialization.
Importance	Medium
PLCopen rule	CP24

Sample:

Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR_INPUT
    bIn1  : BOOL;
    bIn2  : BOOL;                // => SA0035
END_VAR
VAR_OUTPUT
    bOut1 : BOOL;
    bOut2 : BOOL;                // => SA0036
END_VAR
```

```
bOut1 := bIn1;
```

SA0034: Enumeration variables with incorrect assignment

Function	Determines values that are assigned to an enumeration variable. Only defined enumeration constants may be assigned to an enumeration variable.
Reason	An enumeration type variable should only have the intended values, otherwise code that uses that variable may not work correctly. We recommend using only enumerations that have the {attribute 'strict'}. In this case the compiler checks the correct use of the enumeration components.
Importance	High

Sample:

Enumeration E_Color:

```
TYPE E_Color :
(
    eRed   := 1,
    eBlue  := 2,
    eGreen := 3
);
END_TYPE
```

Program MAIN:

```
PROGRAM MAIN
VAR
    eColor : E_Color;
END_VAR
```

```
eColor := E_Color.eRed;
eColor := eBlue;
eColor := 1;                     // => SA0034
```

SA0037: Write access to input variable

Function	Determines input variables (VAR_INPUT) that are subject to write access within the POU.
Reason	According to the IEC 61131-3 standard, an input variable may not be changed within a function block. Such access is also an error source and makes the code more difficult to maintain. It indicates that a variable is used as an input and simultaneously as an auxiliary variable. Such dual use should be avoided.
Importance	Medium

Sample:

Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR_INPUT
    bIn    : BOOL := TRUE;
    nIn    : INT := 100;
END_VAR
VAR_OUTPUT
    bOut   : BOOL;
END_VAR
```

Method FB_Sample.SampleMethod:

```
IF bIn THEN
    nIn   := 500;                 // => SA0037
    bOut  := TRUE;
END_IF
```

SA0038: Read access to output variable

Function	Determines output variables (VAR_OUTPUT) that are subject to read access within the POU.
Reason	The IEC-61131-3 standard prohibits reading an output within a function block. It indicates that the output is not only used as an output but also as a temporary variable for intermediate results. Such dual use should be avoided.
Importance	Low

Sample:

Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR_OUTPUT
    bOut     : BOOL;
    nOut     : INT;
END_VAR
VAR
    bLocal   : BOOL;
    nLocal   : INT;
END_VAR
```

Method FB_Sample.SampleMethod:

```
IF bOut THEN                      // => SA0038
    bLocal := (nOut > 100);       // => SA0038
    nLocal := nOut;               // => SA0038
    nLocal := 2*nOut;             // => SA0038
END_IF
```

SA0040: Possible division by zero

Function	Determines code positions at which division by zero may occur.
Reason	Division by 0 is not allowed. A variable that is used as a divisor should always be checked for 0 first. Otherwise, a "Divide by Zero" exception may occur at runtime.
Importance	High

Sample:

```
PROGRAM MAIN
VAR CONSTANT
    cSample     : INT := 100;
END_VAR
VAR
    nQuotient1  : INT;
    nDividend1  : INT;
    nDivisor1   : INT;

    nQuotient2  : INT;
    nDividend2  : INT;
    nDivisor2   : INT;
END_VAR
```

```
nDivisor1  := cSample;
nQuotient1 := nDividend1/nDivisor1;                 // no error

nQuotient2 := nDividend2/nDivisor2;                 // => SA0040
```

SA0041: Possibly loop-invariant code

Function	Determines code that may be loop-invariant, i.e. code within a FOR, WHILE or REPEAT loop that returns the same result in each loop, in which case repeated execution would be unnecessary. Only calculations are taken into account, no simple assignments.
Reason	This is a performance warning. Code that is executed in a loop but performs a repetitive action in each loop can be executed outside the loop.
Importance	Medium

Sample:

In the following sample SA0041 is output as error/warning, since the variables nTest1 and nTest2 are not used in the loop.

```
PROGRAM MAIN
VAR
    nTest1    : INT := 5;
    nTest2    : INT := nTest1;
    nTest3    : INT;
    nTest4    : INT;
    nTest5    : INT;
    nTest6    : INT;
    nCounter  : INT;
END_VAR
```

```
FOR nCounter := 1 TO 100 BY 1 DO
    nTest3 := nTest1 + nTest2;   // => SA0041
    nTest4 := nTest3 + nCounter; // no loop-invariant code, because nTest3 and nCounter are used 
within loop
    nTest6 := nTest5;            // simple assignments are not regarded
END_FOR
```

SA0042: Usage of different access paths

Function	Determines the usage of different access paths for the same variable.
Reason	Different access to the same element reduces the readability and maintainability of a program. We recommend consistent use of {attribute 'qualified-only'} for libraries, global variable lists and enumerations. This forces fully qualified access.
Importance	Low

Samples:

In the following sample SA0042 is output as error/warning, because the global variable nGlobal is accessed directly and via the GVL namespace, and because the function CONCAT is accessed directly and via the library namespace.

Global variables:

```
VAR_GLOBAL
    nGlobal    : INT;
END_VAR
```

Program MAIN:

```
PROGRAM MAIN
VAR
    sVar       : STRING;
END_VAR
```

```
nGlobal       := INT#2;                                // => SA0042
GVL.nGlobal   := INT#3;                                // => SA0042

sVar := CONCAT('ab', 'cd');                            // => SA0042
sVar := Tc2_Standard.CONCAT('ab', 'cd');               // => SA0042
```

SA0043: Use of a global variable in only one POU

Function	Determines global variables that are only used in a single POU.
Reason	A global variable that is only used at one point should also be declared at this point.
Importance	Medium
PLCopen rule	CP26

Sample:

The global variable nGlobal1 is only used in the MAIN program.

Global variables:

```
VAR_GLOBAL
    nGlobal1  : INT;              // => SA0043
    nGlobal2  : INT;
END_VAR
```

Program SubProgram:

```
nGlobal2 := 123;
```

Program MAIN:

```
SubProgram();

nGlobal1 := nGlobal2;
```

SA0044: Declarations with reference to interface

Function	Determines declarations with REFERENCE TO <interface> and declarations of VAR_IN_OUT variables with the type of an interface (realized implicitly via REFERENCE TO).
Reason	An interface type is always implicitly a reference to an instance of a function block that implements this interface. A reference to an interface is therefore a reference to a reference and can lead to unwanted behavior.
Importance	High

Samples:

I_Sample is an interface defined in the project.

Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR_INPUT
    iInput      : I_Sample;
END_VAR
VAR_OUTPUT
    iOutput     : I_Sample;
END_VAR
VAR_IN_OUT
    iInOut1     : I_Sample;              // => SA0044

    {attribute 'analysis' := '-44'}
    iInOut2     : I_Sample;              // no error SA0044 because rule is deactivated via
attribute
END_VAR
```

Program MAIN:

```
PROGRAM MAIN
VAR
    fbSample    : FB_Sample;
    iSample     : I_Sample;
    refItf      : REFERENCE TO I_Sample; // => SA0044
END_VAR
```

SA0019: Implicit pointer conversions

Function	Determines implicitly generated pointer data type conversions.
Reason	Pointers are not strictly typed in TwinCAT and can be assigned to each other as required. This is a commonly used option and therefore not reported by the compiler. However, it can also unintentionally lead to unexpected access. If a POINTER TO BYTE is assigned to a POINTER TO DWORD, it is possible that the last pointer will unintentionally overwrite memory. Therefore, always check this rule and suppress the message only in cases where you deliberately want to access a value with a different type. Implicit data type conversions are reported with a different message.
Exception	BOOL ↔ BIT
Importance	High
PLCopen rule	CP25

Samples:

```
PROGRAM MAIN
VAR
    nInt    : INT;
    nByte   : BYTE;

    pInt    : POINTER TO INT;
    pByte   : POINTER TO BYTE;
END_VAR
```

```
pInt  := ADR(nInt);
pByte := ADR(nByte);

pInt  := ADR(nByte);             // => SA0019
pByte := ADR(nInt);              // => SA0019

pInt  := pByte;                  // => SA0019
pByte := pInt;                   // => SA0019
```

SA0130: Implicit expanding conversions

Function	Determines implicitly performed conversions from smaller to larger data types.
Reason	The compiler allows any assignment of different types if the range of the source type is fully within the range of the target type. However, the compiler will build a conversion into the code as late as possible. For an assignment of the following type: `nLINT := nDINT * nDINT;` the compiler performs the implicit conversion only after the multiplication: `nLINT := TO_LINT(nDINT * nDINT);` An overflow is therefore truncated. If you want to prevent this, you can have the conversion performed earlier for the elements: `nLINT := TO_LINT(nDINT) * TO_LINT(nDINT);` Therefore, it may be useful to report points where the compiler implements implicit conversions in order to check whether these are exactly what is intended. In addition, explicit conversions can serve to improve portability to other systems if they have more restrictive type checks.
Exception	BOOL ↔ BIT
Importance	Low

Samples:

```
PROGRAM MAIN
VAR
    nBYTE     : BYTE;
    nUSINT    : USINT;
    nUINT     : UINT;
    nINT      : INT;
    nUDINT    : UDINT;
    nDINT     : DINT;
    nULINT    : ULINT;
    nLINT     : LINT;
    nLWORD    : LWORD;
    fLREAL    : LREAL;
END_VAR
```

```
nLINT    := nINT;                  // => SA0130
nULINT   := nBYTE;                 // => SA0130
nLWORD   := nUDINT;                // => SA0130
fLREAL   := nBYTE;                 // => SA0130
nDINT    := nUINT;                 // => SA0130

nBYTE.5  := FALSE;                 // no error (BIT-BOOL-conversion)
```

SA0131: Implicit narrowing conversions

Function	Determines implicitly performed conversions from larger to smaller data types.
Exception	BOOL ↔ BIT
Importance	Low

This message is now obsolete because it is already reported as a warning by the compiler.

Sample:

```
PROGRAM MAIN
VAR
    fREAL     : REAL;
    fLREAL    : LREAL;
END_VAR
```

```
fREAL    := fLREAL;                // => SA0131

nBYTE.5  := FALSE;                 // no error (BIT-BOOL-conversion)
```

SA0132: Implicit signed/unsigned conversions

Function	Determines implicitly performed conversions from signed to unsigned data types or vice versa.
Importance	Low

This message is now obsolete because it is already reported as a warning by the compiler.

Samples:

```
PROGRAM MAIN
VAR
    nBYTE    : BYTE;
    nUDINT   : UDINT;
    nULINT   : ULINT;
    nWORD    : WORD;
    nLWORD   : LWORD;
    nSINT    : SINT;
    nINT     : INT;
    nDINT    : DINT;
    nLINT    : LINT;
END_VAR
```

```
nLINT    := nULINT;                // => SA0132
nUDINT   := nDINT;                 // => SA0132
nSINT    := nBYTE;                 // => SA0132
nWORD    := nINT;                  // => SA0132
nLWORD   := nSINT;                 // => SA0132
```

SA0133: Explicit narrowing conversions

Function	Determines explicitly performed conversions from a larger to a smaller data type.
Reason	A large number of type conversions can mean that incorrect data types have been selected for variables. There are therefore programming guidelines that require an explicit justification for data type conversions.
Importance	Low

Samples:

```
PROGRAM MAIN
VAR
    nSINT    : SINT;
    nDINT    : DINT;
    nLINT    : LINT;
    nBYTE    : BYTE;
    nUINT    : UINT;
    nDWORD   : DWORD;
    nLWORD   : LWORD;
    fREAL    : REAL;
    fLREAL   : LREAL;
END_VAR
```

```
nSINT := LINT_TO_SINT(nLINT);     // => SA0133
nBYTE := DINT_TO_BYTE(nDINT);     // => SA0133
nSINT := DWORD_TO_SINT(nDWORD);   // => SA0133
nUINT := LREAL_TO_UINT(fLREAL);   // => SA0133
fREAL := LWORD_TO_REAL(nLWORD);   // => SA0133
```

SA0134: Explicit signed/unsigned conversions

Function	Determines explicitly performed conversions from signed to unsigned data types or vice versa.
Reason	Excessive use of type conversions may mean that incorrect data types have been selected for variables. There are therefore programming guidelines that require an explicit justification for data type conversions.
Importance	Low

Samples:

```
PROGRAM MAIN
VAR
    nBYTE     : BYTE;
    nUDINT    : UDINT;
    nULINT    : ULINT;
    nWORD     : WORD;
    nLWORD    : LWORD;
    nSINT     : SINT;
    nINT      : INT;
    nDINT     : DINT;
    nLINT     : LINT;
END_VAR
```

```
nLINT   := ULINT_TO_LINT(nULINT); // => SA0134
nUDINT  := DINT_TO_UDINT(nDINT);  // => SA0134
nSINT   := BYTE_TO_SINT(nBYTE);   // => SA0134
nWORD   := INT_TO_WORD(nINT);     // => SA0134
nLWORD  := SINT_TO_LWORD(nSINT);  // => SA0134
```

SA0005: Invalid addresses and data types

Function	Determines invalid address and data type specifications. The following size prefixes are valid for addresses. Deviations from this lead to an invalid address specification. • X for BOOL • B for 1-byte data types • W for 2-byte data types • D for 4-byte data types
Reason	Variables that lie on direct addresses should, if possible, be associated with an address that corresponds to their data type width. It can be confusing for the reader of the code if, for example, a DWORD is placed on a BYTE address.
Importance	Low

If the recommended placeholders %I* or %Q* are used, TwinCAT automatically performs flexible and optimized addressing.

Samples:

```
PROGRAM MAIN
VAR
    nOK   AT%QW0   : INT;
    bOK   AT%QX5.0 : BOOL;

    nNOK  AT%QD10  : INT;         // => SA0005
    bNOK  AT%QB15  : BOOL;        // => SA0005
END_VAR
```

SA0047: Access to direct addresses

Function	Determines direct address access operations in the implementation code.
Reason	Symbolic programming is always preferred: A variable has a name that can also have a meaning. An address does not provide an indication of what it is used for.
Importance	High
PLCopen rule	N1/CP1

Samples:

```
PROGRAM MAIN
VAR
    bBOOL  : BOOL;
    nBYTE  : BYTE;
    nWORD  : WORD;
    nDWORD : DWORD;
END_VAR
```

```
bBOOL  := %IX0.0;                // => SA0047
%QX0.0 := bBOOL;                 // => SA0047
%QW2   := nWORD;                 // => SA0047
%QD4   := nDWORD;                // => SA0047
%MX0.1 := bBOOL;                 // => SA0047
%MB1   := nBYTE;                 // => SA0047
%MD4   := nDWORD;                // => SA0047
```

SA0048: AT declarations on direct addresses

Function	Determines AT declarations on direct addresses.
Reason	The use of direct addresses in the code is an error source and leads to poorer readability and maintainability of the code. We therefore recommend using the placeholders %I* or %Q*, for which TwinCAT automatically carries out flexible and optimized addressing.
Importance	High
PLCopen rule	N1/CP1

Samples:

```
PROGRAMM MAIN
VAR
    b1     AT%IX0.0 : BOOL;       // => SA0048
    b2     AT%I*    : BOOL;       // no error
END_VAR
```

SA0051: Comparison operations on BOOL variables

Function	Determines comparison operations on variables of type BOOL.
Reason	TwinCAT allows such comparisons, but they are rather unusual and can be confusing. The IEC-61131-3 standard does not provide for these comparisons, so you should avoid them.
Importance	Medium

Sample:

```
PROGRAM MAIN
VAR
    b1        : BOOL;
    b2        : BOOL;
    bResult   : BOOL;
END_VAR
```

```
bResult := (b1 > b2);            // => SA0051
bResult := NOT b1 AND b2;
bResult := b1 XOR b2;
```

SA0052: Unusual shift operation

Function	Determines shift operation (bit shift) on signed variables. However, the IEC 61131-3 standard only permits shift operations to bit fields. See also strict rule SA0147 [▶ 66].
Reason	TwinCAT allows shift operations on signed data types. However, such operations are unusual and can be confusing. The IEC-61131-3 standard does not provide for such operations, so you should avoid them.
Exception	Shift operation on bit array data types (byte, DWORD, LWORD, WORD) do not result in a SA0052 error.
Importance	Medium

Samples:

```
PROGRAM MAIN
VAR
    nINT   : INT;
    nDINT  : DINT;
    nULINT : ULINT;
    nSINT  : SINT;
    nUSINT : USINT;
    nLINT  : LINT;

    nDWORD : DWORD;
    nBYTE  : BYTE;
END_VAR
```

```
nINT   := SHL(nINT, BYTE#2);      // => SA0052
nDINT  := SHR(nDINT, BYTE#4);     // => SA0052
nULINT := ROL(nULINT, BYTE#1);    // no error because this is an unsigned data type
nSINT  := ROL(nSINT, BYTE#2);     // => SA0052
nUSINT := ROR(nUSINT, BYTE#3);    // no error because this is an unsigned data type
nLINT  := ROR(nLINT, BYTE#2);     // => SA0052

nDWORD := SHL(nDWORD, BYTE#3);    // no error because DWORD is a bit field data type
nBYTE  := SHR(nBYTE, BYTE#1);     // no error because BYTE is a bit field data type
```

SA0053: Too big bitwise shift

Function	Determines whether the data type width was exceeded in bitwise shift of operands.
Reason	If a shift operation exceeds the data type width, a constant 0 is generated. If a rotation shift exceeds the data type width, it is difficult to read and the rotation value should therefore be shortened.
Importance	High

Samples:

```
PROGRAM MAIN
VAR
    nBYTE  : BYTE;
    nWORD  : WORD;
    nDWORD : DWORD;
    nLWORD : LWORD;
END_VAR
```

```
nBYTE  := SHR(nBYTE, BYTE#8);     // => SA0053
nWORD  := SHL(nWORD, BYTE#45);    // => SA0053
nDWORD := ROR(nDWORD, BYTE#78);   // => SA0053
nLWORD := ROL(nLWORD, BYTE#111);  // => SA0053

nBYTE  := SHR(nBYTE, BYTE#7);     // no error
nWORD  := SHL(nWORD, BYTE#15);    // no error
```

SA0054: Comparisons of REAL/LREAL for equality/inequality

Function	Determines where the comparison operators = (equality) and <> (inequality) compare operands of type REAL or LREAL.

Reason	REAL/LREAL values are implemented as floating-point numbers according to the IEEE 754 standard. This standard implies that certain seemingly simple decimal numbers cannot be represented exactly. As a result, the same decimal number may have different LREAL representations.
	Example:
	`fLREAL_11 := 1.1;`
	`fLREAL_33 := 3.3;`
	`fLREAL_a := fLREAL_11 + fLREAL_11;`
	`fLREAL_b := fLREAL_33 - fLREAL_11;`
	`bTest := fLREAL_a = fLREAL_b;`
	bTest will return FALSE in this case, even if the variables fLREAL_a and fLREAL_b both return the monitoring value "2.2". This is not a compiler error, but a property of the floating-point units of all common processors. You can avoid this by specifying a minimum value by which the values may differ:
	`bTest := ABS(fLREAL_a - fLREAL_b) < 0.1;`

Exception	A comparison with 0.0 is not reported by this analysis. For 0 there is an exact representation in the IEEE 754 standard and therefore the comparison normally works as expected. For better performance, it therefore makes sense to allow a direct comparison here.
Importance	High
PLCopen rule	CP54

Samples:

```
PROGRAM MAIN
VAR
    fREAL1  : REAL;
    fREAL2  : REAL;
    fLREAL1 : LREAL;
    fLREAL2 : LREAL;
    bResult : BOOL;
END_VAR
```

```
bResult := (fREAL1 = fREAL1);    // => SA0054
bResult := (fREAL1 = fREAL2);    // => SA0054
bResult := (fREAL1 <> fREAL2);   // => SA0054
bResult := (fLREAL1 = fLREAL1);  // => SA0054
bResult := (fLREAL1 = fLREAL2);  // => SA0054
bResult := (fLREAL2 <> fLREAL2); // => SA0054

bResult := (fREAL1 > fREAL2);    // no error
bResult := (fLREAL1 < fLREAL2);  // no error
```

SA0055: Unnecessary comparison operations of unsigned operands

Function	Determines unnecessary comparisons with unsigned operands. An unsigned data type is never less than zero.
Reason	A comparison revealed by this check provides a constant result and indicates an error in the code.
Importance	High

Samples:

```
PROGRAM MAIN
VAR
    nBYTE   : BYTE;
    nWORD   : WORD;
    nDWORD  : DWORD;
    nLWORD  : LWORD;
    nUSINT  : USINT;
    nUINT   : UINT;
    nUDINT  : UDINT;
    nULINT  : ULINT;

    nSINT   : SINT;
    nINT    : INT;
    nDINT   : DINT;
    nLINT   : LINT;

    bResult : BOOL;
END_VAR
```

```
bResult := (nBYTE >= BYTE#0);    // => SA0055
bResult := (nWORD < WORD#0);     // => SA0055
bResult := (nDWORD >= DWORD#0);  // => SA0055
bResult := (nLWORD < LWORD#0);   // => SA0055
bResult := (nUSINT >= USINT#0);  // => SA0055
bResult := (nUINT < UINT#0);     // => SA0055
bResult := (nUDINT >= UDINT#0);  // => SA0055
bResult := (nULINT < ULINT#0);   // => SA0055

bResult := (nSINT < SINT#0);     // no error
bResult := (nINT < INT#0);       // no error
bResult := (nDINT < DINT#0);     // no error
bResult := (nLINT < LINT#0);     // no error
```

SA0056: Constant out of valid range

Function	Determines literals (constants) outside the valid operator range.
Reason	The message is output in cases where a variable is compared with a constant that lies outside the value range of this variable. The comparison then returns a constant TRUE or FALSE. This indicates a programming error.
Importance	High

Samples:

```
PROGRAM MAIN
VAR
    nBYTE   : BYTE;
    nWORD   : WORD;
    nDWORD  : DWORD;
    nUSINT  : USINT;
    nUINT   : UINT;
    nUDINT  : UDINT;

    bResult : BOOL;
END_VAR
```

```
bResult := nBYTE >= 355;                         // => SA0056
bResult := nWORD > UDINT#70000;                  // => SA0056
bResult := nDWORD >= ULINT#4294967300;           // => SA0056
bResult := nUSINT > UINT#355;                    // => SA0056
bResult := nUINT >= UDINT#70000;                 // => SA0056
bResult := nUDINT > ULINT#4294967300;            // => SA0056
```

SA0057: Possible loss of decimal points

Function	Determines statements with possible loss of decimal points.

Reason	A piece of code of the following type:
	`nDINT := 1;`
	`fREAL := TO_REAL(nDINT / DINT#2);`
	can lead to misinterpretation. This line of code can lead to the assumption that the division would be performed as a REAL operation and the result in this case would be REAL#0.5. However, this is not the case, i.e. the operation is performed as an integer operation, the result is cast to REAL, and fREAL is assigned the value REAL#0. To avoid this, you should use a cast to ensure that the operation is performed as a REAL operation:
	`fREAL := TO_REAL(nDINT) / REAL#2;`

Importance	Medium

Samples:

```
PROGRAM MAIN
VAR
    fREAL : REAL;
    nDINT : DINT;
    nLINT : LINT;
END_VAR
```

```
nDINT := nDINT + DINT#11;
fREAL := DINT_TO_REAL(nDINT / DINT#3);              // => SA0057
fREAL := DINT_TO_REAL(nDINT) / 3.0;                 // no error
fREAL := DINT_TO_REAL(nDINT) / REAL#3.0;            // no error

nLINT := nLINT + LINT#13;
fREAL := LINT_TO_REAL(nLINT / LINT#7);              // => SA0057
fREAL := LINT_TO_REAL(nLINT) / 7.0;                 // no error
fREAL := LINT_TO_REAL(nLINT) / REAL#7.0;            // no error
```

SA0058: Operations of enumeration variables

Function	Determines operations on variables of type enumeration. Assignments are permitted.
Reason	Enumerations should not be used as normal integer values. Alternatively, an alias data type can be defined or a subrange type can be used.
Exception	If an enumeration is marked with the attribute {attribute 'strict'}, the compiler reports such an operation. If an enumeration is declared as a flag via the pragma attribute {attribute 'flags'}, no SA0058 error is issued for operations with AND, OR, NOT, XOR.
Importance	Medium

Sample 1:

Enumeration E_Color:

```
TYPE E_Color :
(
    eRed   := 1,
    eBlue  := 2,
    eGreen := 3
);
END_TYPE
```

Program MAIN:

```
PROGRAM MAIN
VAR
    nVar   : INT;
    eColor : E_Color;
END_VAR
```

```
eColor := E_Color.eGreen;                           // no error
eColor := E_Color.eGreen + 1;                       // => SA0058
nVar   := E_Color.eBlue / 2;                        // => SA0058
nVar   := E_Color.eGreen + E_Color.eRed;            // => SA0058
```

Sample 2:

Enumeration E_State with attribute 'flags':

```
{attribute 'flags'}
TYPE E_State :
(
    eUnknown := 16#00000001,
    eStopped := 16#00000002,
    eRunning := 16#00000004
) DWORD;
END_TYPE
```

Program MAIN:

```
PROGRAM MAIN
VAR
    nFlags : DWORD;
    nState : DWORD;
END_VAR
```

```
IF (nFlags AND E_State.eUnknown) <> DWORD#0 THEN    // no error
    nState := nState AND E_State.eUnknown;          // no error

ELSIF (nFlags OR E_State.eStopped) <> DWORD#0 THEN  // no error
    nState := nState OR E_State.eRunning;           // no error
END_IF
```

SA0059: Comparison operations always returning TRUE or FALSE

Function	Determines comparisons with literals that always have the result TRUE or FALSE and can already be evaluated during compilation.
Reason	An operation that consistently returns TRUE or FALSE is an indication of a programming error.
Importance	High

Samples:

```
PROGRAM MAIN
VAR
    nBYTE   : BYTE;
    nWORD   : WORD;
    nDWORD  : DWORD;
    nLWORD  : LWORD;
    nUSINT  : USINT;
    nUINT   : UINT;
    nUDINT  : UDINT;
    nULINT  : ULINT;
    nSINT   : SINT;
    nINT    : INT;
    nDINT   : DINT;
    nLINT   : LINT;
    bResult : BOOL;
END_VAR
```

```
bResult := nBYTE  <= 255;                           // => SA0059
bResult := nBYTE  <= BYTE#255;                      // => SA0059
bResult := nWORD  <= WORD#65535;                    // => SA0059
bResult := nDWORD <= DWORD#4294967295;              // => SA0059
bResult := nLWORD <= LWORD#18446744073709551615;    // => SA0059
bResult := nUSINT <= USINT#255;                     // => SA0059
bResult := nUINT  <= UINT#65535;                    // => SA0059
bResult := nUDINT <= UDINT#4294967295;              // => SA0059
bResult := nULINT <= ULINT#18446744073709551615;    // => SA0059
bResult := nSINT  >= -128;                          // => SA0059
bResult := nSINT  >= SINT#-128;                     // => SA0059
bResult := nINT   >= INT#-32768;                    // => SA0059
bResult := nDINT  >= DINT#-2147483648;              // => SA0059
bResult := nLINT  >= LINT#-9223372036854775808;     // => SA0059
```

SA0060: Zero used as invalid operand

Function	Determines operations in which an operand with value 0 results in an invalid or meaningless operation.
Reason	Such an expression may indicate a programming error. In any case, it causes unnecessary runtime.
Importance	Medium

Samples:

```
PROGRAM MAIN
VAR
    nBYTE   : BYTE;
    nWORD   : WORD;
    nDWORD  : DWORD;
    nLWORD  : LWORD;
END_VAR
```

```
nBYTE  := nBYTE  + 0;            // => SA0060
nWORD  := nWORD  - WORD#0;       // => SA0060
nDWORD := nDWORD * DWORD#0;      // => SA0060
nLWORD := nLWORD / 0;            // Compile error: Division by zero
```

SA0061: Unusual operation on pointer

Function	Determines operations on variables of type POINTER TO, which are not = (equality), <> (inequality), + (addition) or ADR.
Reason	Pointer arithmetic is generally permitted in TwinCAT and can be used in a meaningful way. The addition of a pointer with an integer value is therefore classified as a common operation on pointers. This makes it possible to process an array of variable length using a pointer. All other (unusual) operations with pointers are reported with SA0061.
Importance	High
PLCopen rule	E2/E3

Samples:

```
PROGRAM MAIN
VAR
    pINT   : POINTER TO INT;
    nVar   : INT;
END_VAR
```

```
pINT  := ADR(nVar);              // no error
pINT  := pINT * DWORD#5;         // => SA0061
pINT  := pINT / DWORD#2;         // => SA0061
pINT  := pINT MOD DWORD#3;       // => SA0061
pINT  := pINT + DWORD#1;         // no error
pINT  := pINT - DWORD#1;         // => SA0061
```

SA0062: Using TRUE and FALSE in expressions

Function	Determines the use of the literal TRUE or FALSE in expressions (e.g. `b2 := b1 AND NOT TRUE` or `IF (bVar = FALSE) THEN`).
Reason	Such an expression is obviously unnecessary and may indicate an error. In any case, the expression unnecessarily affects the readability and possibly also the runtime.
Importance	Medium

Samples:

```
PROGRAM MAIN
VAR
    bVar1   : BOOL;
    bVar2   : BOOL;
END_VAR
```

```
bVar1 := bVar1 AND NOT TRUE;      // => SA0062
bVar2 := bVar1 OR TRUE;           // => SA0062
bVar2 := bVar1 OR NOT FALSE;      // => SA0062
bVar2 := bVar1 AND FALSE;         // => SA0062

IF (bVar = FALSE) THEN            // => SA0062
    ;
END_IF

IF NOT bVar THEN                  // => no error
    ;
END_IF
```

SA0063: Possibly not 16-bit-compatible operations

Function	Determines 16-bit operations with intermediate results. Background: On 16-bit systems, 32-bit temporary results can be truncated.
Reason	This message is intended to protect against problems in the very rare case when code is written that is intended to run on both a 16-bit processor and a 32-bit processor.
Importance	Low

Sample:

(nVar+10) can exceed 16 bits.

```
PROGRAM MAIN
VAR
    nVar  : INT;
END_VAR
```

```
nVar := (nVar + 10) / 2;          // => SA0063
```

SA0064: Addition of pointer

Function	Determines all pointer additions.
Reason	Pointer arithmetic is generally permitted in TwinCAT and can be used in a meaningful way. However, it is also a source of error. Therefore, there are programming rules that prohibit pointer arithmetic. Such a requirement can be verified with this test.
Importance	Medium

Samples:

```
PROGRAM MAIN
VAR
    aTest : ARRAY[0..10] OF INT;
    pINT  : POINTER TO INT;
    nIdx  : INT;
END_VAR
```

```
pINT  := ADR(aTest[0]);
pINT^ := 0;
pINT  := ADR(aTest) + SIZEOF(INT);                  // => SA0064
pINT^ := 1;
pINT  := ADR(aTest) + 6;                            // => SA0064
pINT  := ADR(aTest[10]);

FOR nIdx := 0 TO 10 DO
    pINT^ := nIdx;
    pINT  := pINT + 2;                              // => SA0064
END_FOR
```

SA0065: Incorrect pointer addition to base size

Function	Determines pointer additions in which the value to be added does not match the basic data size of the pointer. Only literals with the basic size may be added. No multiples of the basic size may be added.

Reason	In TwinCAT (in contrast to C and C++), when a pointer with an integer value is added, only this integer value is added as the number of bytes, not the integer value multiplied by the base size. Sample in ST:
	`pINT := ADR(array_of_int[0]);`
	`pINT := pINT + 2 ; // in TwinCAT zeigt pINT anschließend auf array_of_int[1]`
	This code would work differently in C:
	`short* pShort`
	`pShort = &(array_of_short[0])`
	`pShort = pShort + 2; // in C zeigt pShort anschließend auf array_of_short[2]`
	In TwinCAT, a multiple of the basic size of the pointer should therefore always be added to a pointer. Otherwise the pointer may point to a non-aligned memory, which (depending on the processor) can lead to an alignment exception during access.

Importance	High

Samples:

```
PROGRAM MAIN
VAR
    pUDINT : POINTER TO UDINT;
    nVar   : UDINT;
    pREAL  : POINTER TO REAL;
    fVar   : REAL;
END_VAR
```

```
pUDINT := ADR(nVar) + 4;
pUDINT := ADR(nVar) + (2 + 2);
pUDINT := ADR(nVar) + SIZEOF(UDINT);
pUDINT := ADR(nVar) + 3;                              // => SA0065
pUDINT := ADR(nVar) + 2*SIZEOF(UDINT);                // => SA0065
pUDINT := ADR(nVar) + (3 + 2);                        // => SA0065

pREAL  := ADR(fVar);
pREAL  := pREAL + 4;
pREAL  := pREAL + (2 + 2);
pREAL  := pREAL + SIZEOF(REAL);
pREAL  := pREAL + 1;                                  // => SA0065
pREAL  := pREAL + 2;                                  // => SA0065
pREAL  := pREAL + 3;                                  // => SA0065
pREAL  := pREAL + (SIZEOF(REAL) - 1);                 // => SA0065
pREAL  := pREAL + (1 + 4);                            // => SA0065
```

SA0066: Use of temporary results

Function	Determines applications of intermediate results in statements with a data type that is smaller than the register size. In this case the implicit cast may lead to undesirable results.
Reason	For performance reasons, TwinCAT carries out operations across the register width of the processor. Intermediate results are not truncated. This can lead to misinterpretations, as in the following case: `usintTest := 0;` `bError := usintTest - 1 <> 255;` In TwinCAT, bError is TRUE in this case, because the operation usintTest - 1 is typically executed as a 32-bit operation and the result is not cast to the size of bytes. In the register the value 16#ffffffff is then displayed and this is not equal to 255. To avoid this, you have to explicitly cast the intermediate result: `bError := TO_USINT(usintTest - 1) <> 255;`
Importance	Low

If this message is enabled, a large number of rather unproblematic situations in the code will be reported. Although a problem can only arise if the operation produces an overflow or underflow in the data type, the Static Analysis cannot differentiate between the individual situations.

If you include an explicit typecast in all reported situations, the code will be much slower and less readable!

Sample:

```
PROGRAM MAIN
VAR
    nBYTE   : BYTE;
    nDINT   : DINT;
    nLINT   : LINT;
    bResult : BOOL;
END_VAR
```

```
//
====================================================================================================
=
// type size smaller than register size
// use of temporary result + implicit casting => SA0066
bResult := ((nBYTE - 1) <> 255);                           // => SA0066

// correcting this code by explicit cast so that the type size is equal to or bigger than register
size
bResult := ((BYTE_TO_LINT(nBYTE) - 1) <> 255);             // no error
bResult := ((BYTE_TO_LINT(nBYTE) - LINT#1) <> LINT#255);   // no error

//
====================================================================================================
=
// result depends on solution platform
bResult := ((nDINT - 1) <> 255);                           // no error on x86 solution platform
                                                           // => SA0066 on x64 solution platform

// correcting this code by explicit cast so that the type size is equal to or bigger than register
size
bResult := ((DINT_TO_LINT(nDINT) - LINT#1) <> LINT#255);   // no error

//
====================================================================================================
=
// type size equal to or bigger than register size
// use of temporary result and no implicit casting => no error
bResult := ((nLINT - 1) <> 255);                           // no error

//
====================================================================================================
```

SA0072: Invalid uses of counter variable

Function	Determines write access operations to a counter variable within a FOR loop.
Reason	Manipulating the counter variable in a FOR loop can easily lead to an infinite loop. To prevent the execution of the loop for certain values of the counter variables, use CONTINUE or simply IF.
Importance	High
PLCopen rule	L12

Sample:

```
PROGRAM MAIN
VAR_TEMP
    nIndex  : INT;
END_VAR
VAR
    aSample : ARRAY[1..10] OF INT;
    nLocal  : INT;
END_VAR
```

```
FOR nIndex := 1 TO 10 BY 1 DO
    aSample[nIndex]  := nIndex;                      // no error
    nLocal           := nIndex;                      // no error

    nIndex           := nIndex - 1;                  // => SA0072
    nIndex           := nIndex + 1;                  // => SA0072
    nIndex           := nLocal;                      // => SA0072
END_FOR
```

SA0073: Use of non-temporary counter variable

Function	Determines the use of non-temporary variables in FOR loops.
Reason	This is a performance warning. A counter variable is always initialized each time a programming block is called. You can create such a variable as a temporary variable (VAR_TEMP). This may result in faster access, and the variable does not occupy permanent storage space.
Importance	Medium
PLCopen rule	CP21/L13

Sample:

```
PROGRAM MAIN
VAR
    nIndex  : INT;
    nSum    : INT;
END_VAR
```

```
FOR nIndex := 1 TO 10 BY 1 DO    // => SA0073
    nSum := nSum + nIndex;
END_FOR
```

SA0080: Loop index variable for array index exceeds array range

Function	Determines FOR statements in which the index variable is used for access to an array index and exceeds the array index range.
Reason	Arrays are typically processed in FOR loops. The start and end value of the counter variable should typically match or at least not exceed the lower and upper limits of the array. Here a typical cause of error is detected if array boundaries are changed and constants are not handled carefully, or if a different value is used by mistake in the FOR loop than in the array declaration.
Importance	High

Samples:

```
PROGRAM MAIN
VAR CONSTANT
    c1      : INT := 0;
END_VAR
VAR
    nIndex1 : INT;
    nIndex2 : INT;
    nIndex3 : INT;
    a1      : ARRAY[1..100] OF INT;
    a2      : ARRAY[1..9,1..9,1..9] OF INT;
    a3      : ARRAY[0..99] OF INT;
END_VAR
```

```
// 1 violation of the rule (lower range is exeeded) => 1 error SA0080
FOR nIndex1 := c1 TO INT#100 BY INT#1 DO
    a1[nIndex1] := nIndex1;                          // => SA0080
END_FOR

// 6 violations (lower and upper range is exeeded for each array dimension) => 3 errors SA0080
FOR nIndex2 := INT#0 TO INT#10 BY INT#1 DO
    a2[nIndex2, nIndex2, nIndex2] := nIndex2;        // => SA0080
END_FOR

// 1 violation (upper range is exeeded by the end result of the index), expressions on index are not
evaluated => no error
FOR nIndex3 := INT#0 TO INT#50 BY INT#1 DO
    a3[nIndex3 * INT#2] := nIndex3;                  // no error
END_FOR
```

SA0081: Upper border is not a constant

Function	Determines FOR statements in which the upper limit is not defined with a constant value.
Reason	If the upper limit of a loop is a variable value, it is no longer possible to see how often a loop is executed. This can lead to serious problems at runtime, in the worst case to an infinite loop.
Importance	High

Samples:

```
PROGRAM MAIN
VAR CONSTANT
    cMax   : INT := 10;
END_VAR
VAR
    nIndex : INT;
    nVar   : INT;
    nMax1  : INT := 10;
    nMax2  : INT := 10;
END_VAR
```

```
FOR nIndex := 0 TO 10 DO         // no error
    nVar := nIndex;
END_FOR

FOR nIndex := 0 TO cMax DO       // no error
    nVar := nIndex;
END_FOR

FOR nIndex := 0 TO nMax1 DO      // => SA0081
    nVar := nIndex;
END_FOR

FOR nIndex := 0 TO nMax2 DO      // => SA0081
    nVar := nIndex;

    IF nVar = 10 THEN
        nMax2 := 50;
    END_IF
END_FOR
```

SA0075: Missing ELSE

Function	Determines CASE statements without ELSE branch.
Reason	Defensive programming requires the presence of an ELSE in every CASE statement. If no action is required in the ELSE case, you should indicate this with a comment. The reader of the code is then aware that the case was not simply overlooked.
Exception	A missing ELSE branch is not reported as missing if an enumeration declared with the 'strict' attribute is used in the CASE statement, and if all enumeration constants are listed in that CASE statement.
Importance	Low
PLCopen rule	L17

Sample:

```
{attribute 'qualified_only'}
{attribute 'strict'}
{attribute 'to_string'}
TYPE E_Sample :
(
    eNull,
    eOne,
    eTwo
);
END_TYPE
```

```
PROGRAM MAIN
VAR
    eSample : E_Sample;
    nVar    : INT;
END_VAR
```

```
CASE eSample OF
    E_Sample.eNull: nVar := 0;
    E_Sample.eOne:  nVar := 1;
    E_Sample.eTwo:  nVar := 2;
END_CASE

CASE eSample OF                  // => SA0075
    E_Sample.eNull: nVar := 0;
    E_Sample.eTwo:  nVar := 2;
END_CASE
```

SA0076: Missing enumeration constant

Function	Determines code positions where an enumeration variable is used as condition and not all enumeration values are treated as CASE branches.
Reason	Defensive programming requires the processing of all possible values of an enumeration. If no action is required for a particular enumeration value, you should indicate this explicitly with a comment. This makes it clear that the value was not simply overlooked.
Importance	Low

Sample:

In the following sample the enumeration value eYellow is not treated as a CASE branch.

Enumeration E_Color:

```
TYPE E_Color :
(
    eRed,
    eGreen,
    eBlue,
    eYellow
);
END_TYPE
```

Program MAIN:

```
PROGRAM MAIN
VAR
    eColor : E_Color;
    bVar   : BOOL;
END_VAR
```

```
eColor := E_Color.eYellow;

CASE eColor OF                   // => SA0076
    E_Color.eRed:
         bVar := FALSE;

    E_Color.eGreen,
    E_Color.eBlue:
         bVar := TRUE;

ELSE
    bVar := NOT bVar;
END_CASE
```

SA0077: Type mismatches with CASE expression

Function	Determines code positions where the data type of a condition does not match that of the CASE branch.
Reason	If the data types between the CASE variable and the CASE case do not match, this could indicate an error.
Importance	Low

Sample:

Enumeration E_Sample:

```
TYPE E_Sample :
(
    eNull,
    eOne,
    eTwo
) DWORD;
END_TYPE
```

Program MAIN:

```
PROGRAM MAIN
VAR
    nDINT  : DINT;
    bVar   : BOOL;
END_VAR
```

```
nDINT := nDINT + DINT#1;

CASE nDINT OF
    DINT#1:
         bVar := FALSE;

    E_Sample.eTwo,               // => SA0077
    DINT#3:
         bVar := TRUE;

ELSE
    bVar := NOT bVar;
END_CASE
```

SA0078: Missing CASE branches

Function	Determines CASE statements without cases, i.e. with only a single ELSE statement.
Reason	A CASE statement without cases wastes execution time and is difficult to read.
Importance	Medium

Sample:

```
PROGRAM MAIN
VAR
    nVar   : DINT;
    bVar   : BOOL;
END_VAR
```

```
nVar := nVar + INT#1;

CASE nVar OF                      // => SA0078
ELSE
    bVar := NOT bVar;
END_CASE
```

SA0090: Return statement before end of function

Function	Determines code positions where the RETURN statement is not the last statement in a function, method, property or program.
Reason	A RETURN in the code leads to poorer maintainability, testability and readability of the code. A RETURN in the code is easily overlooked. You must insert code, which should be executed in any case when a function exits, before each RETURN. This is often overlooked.
Importance	Medium
PLCopen rule	CP14

Sample:

```
FUNCTION F_TestFunction : BOOL
```

```
F_TestFunction := FALSE;
RETURN;                           // => SA0090
F_TestFunction := TRUE;
```

SA0095: Assignments in conditions

Function	Determines assignments in conditions of IF, CASE, WHILE or REPEAT constructs.
Reason	An assignment (:=) and a comparison (=) can easily be confused. An assignment in a condition can therefore easily be unintentional and is therefore reported. This can also confuse readers of the code.
Importance	High

Samples:

```
PROGRAM MAIN
VAR
    bTest   : BOOL;
    bResult : BOOL;
    bValue  : BOOL;

    b1      : BOOL;
    n1      : INT;
    n2      : INT;

    nCond1  : INT  := INT#1;
    nCond2  : INT  := INT#2;
    bCond   : BOOL := FALSE;
    nVar    : INT;
    eSample : E_Sample;
END_VAR
```

```
// IF constructs
IF (bTest := TRUE) THEN                                    // => SA0095
    DoSomething();
END_IF

IF (bResult := F_Sample(bInput := bValue)) THEN            // => SA0095
    DoSomething();
END_IF

b1 := ((n1 := n2) = 99);                                   // => SA0095
```

```
IF INT_TO_BOOL(nCond1 := nCond2) THEN                      // => SA0095
    DoSomething();
ELSIF (nCond1 := 11) = 11 THEN                             // => SA0095
    DoSomething();
END_IF

IF bCond := TRUE THEN                                      // => SA0095
    DoSomething();
END_IF

IF (bCond := FALSE) OR (nCond1 := nCond2) = 12 THEN        // => SA0095
    DoSomething();
END_IF

IF (nVar := nVar + 1) = 120 THEN                           // => SA0095
    DoSomething();
END_IF

// CASE construct
CASE (eSample := E_Sample.eMember0) OF                     // => SA0095
    E_Sample.eMember0:
            DoSomething();

    E_Sample.eMember1:
            DoSomething();
END_CASE

// WHILE construct
WHILE (bCond = TRUE) OR (nCond1 := nCond2) = 12 DO         // => SA0095
    DoSomething();
END_WHILE

// REPEAT construct
REPEAT
    DoSomething();
UNTIL
     (bCond = TRUE) OR ((nCond1 := nCond2) = 12)           // => SA0095
END_REPEAT
```

SA0100: Variables greater than <n> bytes

Function	Determines variables that use more than n bytes; n is defined by the current configuration. You can configure the parameter that is taken into account in the check by double-clicking on the row for rule 100 in the rule configuration (PLC Project Properties > category "Static Analysis" > "Rules" tab > Rule 100). You can make the following settings in the dialog that appears: • Upper limit in bytes (default value: 1024)
Reason	Some programming guidelines specify a maximum size for a single variable. This function facilitates a corresponding check.
Importance	Low

Sample:

In the following sample the variable aSample is greater than 1024 bytes.

```
PROGRAM MAIN
VAR
    aSample : ARRAY [0..1024] OF BYTE;              // => SA0100
END_VAR
```

SA0101: Names with invalid length

Function	Determines names with invalid length. The object names must have a defined length. You can configure the parameters that are taken into account in the check by double-clicking on the row for rule 101 in the rule configuration (PLC Project Properties > category "Static Analysis" > "Rules" tab > Rule 101). You can make the following settings in the dialog that appears: • Minimum number of characters (default value: 5) • Maximum number of characters (default value: 30) • Exceptions
Reason	Some programming guidelines specify a minimum length for variable names. Compliance can be verified with this analysis.
Importance	Low
PLCopen rule	N6

Samples:

Rule 101 is configured with the following parameters:

- Minimum number of characters: 5
- Maximum number of characters: 30
- Exceptions: MAIN, i

Program PRG1:

```
PROGRAM PRG1                      // => SA0101
VAR
END_VAR
```

Program MAIN:

```
PROGRAM MAIN                      // no error due to configured exceptions
VAR
    i     : INT;                  // no error due to configured exceptions
    b     : BOOL;                 // => SA0101
    nVar1 : INT;
END_VAR
```

```
PRG1();
```

SA0102: Access to program/fb variables from the outside

Function	Determines external access to local variables of programs or function blocks.
Reason	TwinCAT determines external write access operations to local variables of programs or function blocks as compilation errors. Since read access operations to local variables are not intercepted by the compiler and this violates the basic principle of data encapsulation (concealing of data) and contravenes the IEC 61131-3 standard, this rule can be used to determine read access to local variables.
Importance	Medium

Samples:

Function block FB_Base:

```
FUNCTION_BLOCK FB_Base
VAR
    nLocal : INT;
END_VAR
```

Method FB_Base.SampleMethod:

```
METHOD SampleMethod : INT
VAR_INPUT
END_VAR
```

```
nLocal := nLocal + 1;
```

Function block FB_Sub:

```
FUNCTION_BLOCK FB_Sub EXTENDS FB_Base
```

Method FB_Sub.SampleMethod:

```
METHOD SampleMethod : INT
VAR_INPUT
END_VAR
```

```
nLocal := nLocal + 5;
```

Program PRG_1:

```
PROGRAM PRG_1
VAR
    bLocal : BOOL;
END_VAR
```

```
bLocal := NOT bLocal;
```

Program MAIN:

```
PROGRAM MAIN
VAR
    bRead     : BOOL;
    nReadBase : INT;
    nReadSub  : INT;
    fbBase    : FB_Base;
    fbSub     : FB_Sub;
END_VAR
```

```
bRead     := PRG_1.bLocal;       // => SA0102
nReadBase := fbBase.nLocal;      // => SA0102
nReadSub  := fbSub.nLocal;       // => SA0102
```

SA0103: Concurrent access on not atomic data

Function	Determines non-atomic variables (for example with data types STRING, WSTRING, ARRAY, STRUCT, FB instances, 64-bit data types) that are used in more than one task.
Reason	If no synchronization occurs during access, inconsistent values may be read when reading in one task and writing in another task at the same time.
Exception	This rule does not apply in the following cases: • If the target system has an FPU (floating point unit), the access of several tasks to LREAL variables is not determined and reported. • If the target system is a 64-bit processor or "TwinCAT RT (x64)" is selected as the solution platform, the rule does not apply for 64-bit data types.
Importance	Medium

See also rule SA0006 [▸ 22].

Samples:

Structure ST_sample:

```
TYPE ST_Sample :
STRUCT
    bMember : BOOL;
    nTest   : INT;
END_STRUCT
END_TYPE
```

Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR_INPUT
    fInput  : LREAL;
END_VAR
```

GVL:

```
{attribute 'qualified_only'}
VAR_GLOBAL
    fTest    : LREAL;              // => no error SA0103: Since the target system has a FPU, SA0103
does not apply.
    nTest    : LINT;               // => error reporting depends on the solution platform:
                                      // - SA0103 error if solution platform is set to "TwinCAT
RT(x86)"
                                      // - no error SA0103 if solution platform is set to "TwinCAT
(x64)"
    sTest    : STRING;             // => SA0103
    wsTest   : WSTRING;            // => SA0103
    aTest    : ARRAY[0..2] OF INT; // => SA0103
    aTest2   : ARRAY[0..2] OF INT; // => SA0103
    fbTest   : FB_Sample;          // => SA0103
    stTest   : ST_Sample;          // => SA0103
END_VAR
```

Program MAIN1, called by task PlcTask1:

```
PROGRAM MAIN1
VAR
END_VAR
```

```
GVL.fTest         := 5.0;
GVL.nTest         := 123;
GVL.sTest         := 'sample text';
GVL.wsTest        := "sample text";
GVL.aTest         := GVL.aTest2;
GVL.fbTest.fInput := 3;
GVL.stTest.nTest  := GVL.stTest.nTest + 1;
```

Program MAIN2, called by task PlcTask2:

```
PROGRAM MAIN2
VAR
    fLocal   : LREAL;
    nLocal   : LINT;
    sLocal   : STRING;
    wsLocal  : WSTRING;
    aLocal   : ARRAY[0..2] OF INT;
    aLocal2  : ARRAY[0..2] OF INT;
    fLocal2  : LREAL;
    nLocal2   : INT;
END_VAR
```

```
fLocal   := GVL.fTest + 1.5;
nLocal   := GVL.nTest + 10;
sLocal   := GVL.sTest;
wsLocal  := GVL.wsTest;
aLocal   := GVL.aTest;
aLocal2  := GVL.aTest2;
fLocal2  := GVL.fbTest.fInput;
nLocal2  := GVL.stTest.nTest;
```

SA0105: Multiple instance calls

Function	Determines and reports instances of function blocks that are called more than once. To ensure that an error message for a repeatedly called function block instance is generated, the attribute {attribute 'analysis:report-multiple-instance-call'} [▶ 112] must be added in the declaration part of the function block.
Reason	Some function blocks are designed such that they can only be called once in a cycle. This test checks whether a call is made at several points.
Importance	Low
PLCopen rule	CP16/CP20

Sample:

In the following sample the Static Analysis will issue an error for fb2, since the instance is called more than once, and the function block is declared with the required attribute.

Function block FB_Test1 without attribute:

```
FUNCTION_BLOCK FB_Test1
```

Function block FB_Test2 with attribute:

```
{attribute 'analysis:report-multiple-instance-calls'}
FUNCTION_BLOCK FB_Test2
```

Program MAIN:

```
PROGRAM MAIN
VAR
    fb1   : FB_Test1;
    fb2   : FB_Test2;
END_VAR
```

```
fb1();
fb1();
fb2();                           // => SA0105
fb2();                           // => SA0105
```

SA0106: Virtual method calls in FB_init

Function	Determines method calls in the method FB_init of a basic function block, which are overwritten by a function block derived from the basic FB.
Reason	In such cases it may happen that the variables in overwritten methods are not initialized in the base FB.
Importance	High

Sample:

- Function block FB_Base has the methods FB_init and MyInit. FB_init calls MyInit for initialization.
- Function block FB_Sub is derived from FB_Base.
- FB_Sub.MyInit overwrites or extends FB_Base.MyInit.
- MAIN instantiates FB_Sub. During this process it uses the instance variable nSub before it was initialized, due to the call sequence during the initialization.

Function block FB_Base:

```
FUNCTION_BLOCK FB_Base
VAR
    nBase         : DINT;
END_VAR
```

Method FB_Base.FB_init:

```
METHOD FB_init : BOOL
VAR_INPUT
    bInitRetains : BOOL;
    bInCopyCode  : BOOL;
END_VAR
VAR
    nLocal       : DINT;
END_VAR
```

```
nLocal := MyInit();              // => SA0106
```

Method FB_Base.MyInit:

```
METHOD MyInit : DINT
```

```
nBase  := 123;                   // access to member of FB_Base
MyInit := nBase;
```

Function block FB_Sub:

```
FUNCTION_BLOCK FB_Sub EXTENDS FB_Base
VAR
    nSub         : DINT;
END_VAR
```

Method FB_Sub.MyInit:

```
METHOD MyInit : DINT
```

```
nSub   := 456;                  // access to member of FB_Sub
SUPER^.MyInit();                // call of base implementation
MyInit := nSub;
```

Program MAIN:

```
PROGRAM MAIN
VAR
    fbBase       : FB_Base;
    fbSub        : FB_Sub;
END_VAR
```

The instance MAIN.fbBase has the following variable values after the initialization:

- nBase is 123

The instance MAIN.fbSub has the following variable values after the initialization:

- nBase is 123
- nSub is 0

The variable MAIN.fbSub.nSub is 0 after the initialization, because the following call sequence is used during the initialization of fbSub:

- Initialization of the basic function block:
  - implicit initialization
  - explicit initialization: FB_Base.FB_init
  - FB_Base.FB_init calls FB_Sub.MyInit → **SA0106**
  - FB_Sub.MyInit calls FB_Base.MyInit (via SUPER pointer)
- Initialization of the derived function block:
  - implicit initialization

SA0107: Missing formal parameters

Function	Determines where formal parameters are missing.
Reason	Code becomes more readable if the formal parameters are specified when the code is called.
Importance	Low

Sample:

Function F_Sample:

```
FUNCTION F_Sample : BOOL
VAR_INPUT
    bIn1 : BOOL;
    bIn2 : BOOL;
END_VAR
```

```
F_Sample := bIn1 AND bIn2;
```

Program MAIN:

```
PROGRAM MAIN
VAR
    bReturn : BOOL;
END_VAR
```

```
bReturn := F_Sample(TRUE, FALSE);                   // => SA0107
bReturn := F_Sample(TRUE, bIn2 := FALSE);           // => SA0107
bReturn := F_Sample(bIn1 := TRUE, bIn2 := FALSE);   // no error
```

SA0111: Pointer variables

Function	Determines variables of type POINTER TO.
Reason	The IEC 61131-3 standard does not allow pointers.
Importance	Low

Sample:

```
PROGRAM MAIN
VAR
    pINT   : POINTER TO INT;      // => SA0111
END_VAR
```

SA0112: Reference variables

Function	Determines variables of type REFERENCE TO.
Reason	The IEC 61131-3 standard does not allow references.
Importance	Low

Sample:

```
PROGRAM MAIN
VAR
    refInt : REFERENCE TO INT;    // => SA0112
END_VAR
```

SA0113: Variables with data type WSTRING

Function	Determines variables of type WSTRING.
Reason	Not all systems support WSTRING. The code becomes easier to port if WSTRING is not used.
Importance	Low

Sample:

```
PROGRAM MAIN
VAR
    wsVar  : WSTRING;             // => SA0113
END_VAR
```

SA0114: Variables with data type LTIME

Function	Determines variables of type LTIME.
Reason	Not all systems support LTIME. The code becomes more portable if LTIME is not used.
Importance	Low

Sample:

```
PROGRAM MAIN
VAR
    tVar   : LTIME;               // => SA0114
END_VAR
```

```
// no error SA0114 for the following code line:
tVar := tVar + LTIME#1000D15H23M12S34MS2US44NS;
```

SA0115: Declarations with data type UNION

Function	Determines declarations of a UNION data type and declarations of variables of the type of a UNION.
Reason	The IEC-61131-3 standard has no provision for unions. The code becomes easier to port if there are no unions.
Importance	Low

Samples:

Union U_Sample:

```
TYPE U_Sample :                   // => SA0115
UNION
    fVar    : LREAL;
    nVar    : LINT;
END_UNION
END_TYPE
```

Program MAIN:

```
PROGRAM MAIN
VAR
    uSample : U_Sample;           // => SA0115
END_VAR
```

SA0117: Variables with data type BIT

Function	Determines declarations of variables of type BIT (possible within structure and function block definitions).
Reason	The IEC-61131-3 has no provision for data type BIT. The code becomes easier to port if BIT is not used.
Importance	Low

Samples:

Structure ST_sample:

```
TYPE ST_Sample :
STRUCT
    bBIT  : BIT;                  // => SA0117
    bBOOL : BOOL;
END_STRUCT
END_TYPE
```

Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR
    bBIT  : BIT;                  // => SA0117
    bBOOL : BOOL;
END_VAR
```

SA0119: Object-oriented features

Function	Determines the use of object-oriented features such as: • Function block declarations with EXTENDS or IMPLEMENTS • Property and interface declarations • Use of the THIS or SUPER pointer
Reason	Not all systems support object-oriented programming. The code becomes easier to port if object orientation is not used.
Importance	Low

Samples:

Interface I_Sample:

```
INTERFACE I_Sample                                    // => SA0119
```

Function block FB_Base:

```
FUNCTION_BLOCK FB_Base IMPLEMENTS I_Sample            // => SA0119
```

Function block FB_Sub:

```
FUNCTION_BLOCK FB_Sub EXTENDS FB_Base                 // => SA0119
```

Method FB_Sub.SampleMethod:

```
METHOD SampleMethod : BOOL                            // no error
```

Get function of the property FB_Sub.SampleProperty:

```
VAR                                                   // => SA0119
END_VAR
```

Set function of the property FB_Sub.SampleProperty:

```
VAR                                                   // => SA0119
END_VAR
```

SA0120: Program calls

Function	Determines program calls.
Reason	According to the IEC 61131-3 standard, programs can only be called in the task configuration. The code becomes easier to port if program calls elsewhere are avoided.
Importance	Low

Sample:

Program SubProgram:

```
PROGRAM SubProgram
```

Program MAIN:

```
PROGRAM MAIN
```

```
SubProgram();                    // => SA0120
```

SA0121: Missing VAR_EXTERNAL declarations

Function	Determines the use of a global variable in the function block, without it being declared as VAR_EXTERNAL (required according to the standard).
Reason	According to the IEC 61131-3 standard, access to global variables is only permitted via an explicit import through a VAR_EXTERNAL declaration.
Importance	Low
PLCopen rule	CP18

In TwinCAT 3 PLC it is not necessary for variables to be declared as external. The keyword exists in order to maintain compatibility with IEC 61131-3.

Sample:

Global variables:

```
VAR_GLOBAL
    nGlobal : INT;
END_VAR
```

Program Prog1:

```
PROGRAM Prog1
VAR
    nVar     : INT;
END_VAR
```

```
nVar := nGlobal;                 // => SA0121
```

Program Prog2:

```
PROGRAM Prog2
VAR
    nVar     : INT;
END_VAR
VAR_EXTERNAL
    nGlobal : INT;
END_VAR
```

```
nVar := nGlobal;                 // no error
```

SA0122: Array index defined as expression

Function	Determines the use of expressions in the declaration of array boundaries.
Reason	Not all systems allow expressions as array boundaries.
Importance	Low

Sample:

```
PROGRAM MAIN
VAR CONSTANT
    cSample   : INT := INT#15;
END_VAR
VAR
    aSample1 : ARRAY[0..10] OF INT;
    aSample2 : ARRAY[0..10+5] OF INT;                // => SA0122
    aSample3 : ARRAY[0..cSample] OF INT;
    aSample4 : ARRAY[0..cSample + 1] OF INT;         // => SA0122
END_VAR
```

SA0123: Usages of INI, ADR or BITADR

Function	Determines the use of the (TwinCAT-specific) operators INI, ADR, BITADR.
Reason	TwinCAT-specific operators prevent portability of the code.
Importance	Low

Sample:

```
PROGRAM MAIN
VAR
    nVar : INT;
    pINT : POINTER TO INT;
END_VAR
```

```
pINT := ADR(nVar);               // => SA0123
```

SA0147: Unusual shift operation - strict

Function	Determines bit shift operations that are not performed on bit field data types (BYTE, WORD, DWORD, LWORD).
Reason	The IEC 61131-3 standard only allows bit access to bit field data types. However, the TwinCAT 3 compiler also allows bit shift operations with unsigned data types.
Importance	Low

See also non-strict rule SA0052 [▸ 43].

Samples:

```
PROGRAM MAIN
VAR
    nBYTE      : BYTE := 16#45;
    nWORD      : WORD := 16#0045;
    nUINT      : UINT;
    nDINT      : DINT;
    nResBYTE   : BYTE;
    nResWORD   : WORD;
    nResUINT   : UINT;
    nResDINT   : DINT;
    nShift     : BYTE := 2;
END_VAR
```

```
nResBYTE := SHL(nByte,nShift);   // no error because BYTE is a bit field
nResWORD := SHL(nWORD,nShift);   // no error because WORD is a bit field
nResUINT := SHL(nUINT,nShift);   // => SA0147
nResDINT := SHL(nDINT,nShift);   // => SA0147
```

SA0148: Unusual bit access - strict

Function	Determines bit access operations that are not performed on bit field data types (BYTE, WORD, DWORD, LWORD).
Reason	The IEC 61131-3 standard only allows bit access to bit field data types. However, the TwinCAT 3 compiler also allows bit access to unsigned data types.
Importance	Low

See also non-strict rule SA0018 [▸ 28].

Samples:

```
PROGRAM MAIN
VAR
    nINT       : INT;
    nDINT      : DINT;
    nULINT     : ULINT;
    nSINT      : SINT;
    nUSINT     : USINT;
    nBYTE      : BYTE;
END_VAR
```

```
nINT.3    := TRUE;               // => SA0148
nDINT.4   := TRUE;               // => SA0148
nULINT.18 := FALSE;              // => SA0148
nSINT.2   := FALSE;              // => SA0148
nUSINT.3  := TRUE;               // => SA0148
nBYTE.5   := FALSE;              // no error because BYTE is a bitfield
```

SA0118: Initializations not using constants

Function	Determines initializations that do not assign constants.
Reason	Initializations should be as consistent as possible and should not refer to other variables. In particular, you should avoid function calls during initialization, since this can lead to access to uninitialized data.
Importance	Medium

Samples:

Function F_ReturnDWORD:

```
FUNCTION F_ReturnDWORD : DWORD
```

Program MAIN:

```
PROGRAM MAIN
VAR CONSTANT
    c1 : DWORD := 100;
END_VAR
VAR
    n1 : DWORD := c1;
    n2 : DWORD := F_ReturnDWORD();                  // => SA0118
    n3 : DWORD := 150;
    n4 : DWORD := n3;                               // => SA0118
END_VAR
```

SA0124: Dereference access in initializations

Function	Determines all code locations where dereferenced pointers are used in the declaration part of POUs.
Reason	Pointers and references should not be used for initializations, because this can lead to access violations at runtime if the pointer has not been initialized.
Importance	Medium

Samples:

```
FUNCTION_BLOCK FB_Test
VAR_INPUT
    pStruct    : POINTER TO ST_Test;
    refStruct  : REFERENCE TO ST_Test;
END_VAR
VAR
    bPointer   : BOOL := pStruct^.bTest;  // => SA0124: Dereference access in initialization
    bRef       : BOOL := refStruct.bTest; // => SA0125: Reference used in initialization
END_VAR
```

```
bPointer := pStruct^.bTest;               // => SA0039: Possible null pointer dereference 'pStruct^'
bRef     := refStruct.bTest;              // => SA0145: Possible use of not initialized reference
'refStruct'

IF pStruct <> 0 THEN
    bPointer := pStruct^.bTest;           // no error SA0039 as the pointer is checked for unequal 0
END_IF

IF __ISVALIDREF(refStruct) THEN
    bRef     := refStruct.bTest;          // no error SA0145 as the reference is checked via
__ISVALIDREF
END_IF
```

Overview of the rules on "dereferencing".

Pointers:

- Dereferencing of pointers in the declaration part => SA0124 [▸ 68]
- Possible null pointer dereferences in the implementation part => SA0039 [▸ 69]

References:

- Use of references in the declaration part => SA0125 [▸ 69]
- Possible use of not initialized reference in the implementation part => SA0145 [▸ 71]

Interfaces:

- Possible use of not initialized interface in the implementation part => SA0046 [▸ 70]

SA0125: References in initializations

Function	Determines all reference variables used for initialization in the declaration part of POUs.
Reason	Pointers and references should not be used for initializations, because this can lead to access violations at runtime if the pointer has not been initialized.
Importance	Medium

Samples:

```
FUNCTION_BLOCK FB_Test
VAR_INPUT
    pStruct    : POINTER TO ST_Test;
    refStruct  : REFERENCE TO ST_Test;
END_VAR
VAR
    bPointer   : BOOL := pStruct^.bTest;  // => SA0124: Dereference access in initialization
    bRef       : BOOL := refStruct.bTest; // => SA0125: Reference used in initialization
END_VAR
```

```
bPointer := pStruct^.bTest;                 // => SA0039: Possible null pointer dereference 'pStruct^'
bRef     := refStruct.bTest;                // => SA0145: Possible use of not initialized reference
'refStruct'

IF pStruct <> 0 THEN
    bPointer := pStruct^.bTest;             // no error SA0039 as the pointer is checked for unequal 0
END_IF

IF __ISVALIDREF(refStruct) THEN
    bRef     := refStruct.bTest;            // no error SA0145 as the reference is checked via
__ISVALIDREF
END_IF
```

Overview of the rules on "dereferencing".

Pointers:

- Dereferencing of pointers in the declaration part => SA0124 [▸ 68]
- Possible null pointer dereferences in the implementation part => SA0039 [▸ 69]

References:

- Use of references in the declaration part => SA0125 [▸ 69]
- Possible use of not initialized reference in the implementation part => SA0145 [▸ 71]

Interfaces:

- Possible use of not initialized interface in the implementation part => SA0046 [▸ 70]

SA0039: Possible null pointer dereferences

Function	Determines code positions at which a NULL-pointer may be dereferenced.
Reason	A pointer should be checked before each dereferencing to see if it is not equal to 0. Otherwise an access violation may occur at runtime.
Importance	High

Sample 1:

```
PROGRAM MAIN
VAR
    pInt1     : POINTER TO INT;
    pInt2     : POINTER TO INT;
    pInt3     : POINTER TO INT;
    nVar1     : INT;
    nCounter  : INT;
END_VAR
```

```
nCounter := nCounter + INT#1;

pInt1    := ADR(nVar1);
pInt1^   := nCounter;            // no error
```

```
pInt2^   := nCounter;             // => SA0039
nVar1    := pInt3^;               // => SA0039
```

Sample 2:

```
FUNCTION_BLOCK FB_Test
VAR_INPUT
    pStruct     : POINTER TO ST_Test;
    refStruct   : REFERENCE TO ST_Test;
END_VAR
VAR
    bPointer    : BOOL := pStruct^.bTest;  // => SA0124: Dereference access in initialization
    bRef        : BOOL := refStruct.bTest; // => SA0125: Reference used in initialization
END_VAR
```

```
bPointer := pStruct^.bTest;                // => SA0039: Possible null pointer dereference 'pStruct^'
bRef     := refStruct.bTest;               // => SA0145: Possible use of not initialized reference
'refStruct'

IF pStruct <> 0 THEN
    bPointer := pStruct^.bTest;            // no error SA0039 as the pointer is checked for unequal 0
END_IF

IF __ISVALIDREF(refStruct) THEN
    bRef     := refStruct.bTest;           // no error SA0145 as the reference is checked via
__ISVALIDREF
END_IF
```

Overview of the rules on "dereferencing".

Pointers:

- Dereferencing of pointers in the declaration part => SA0124 [▶ 68]
- Possible null pointer dereferences in the implementation part => SA0039 [▶ 69]

References:

- Use of references in the declaration part => SA0125 [▶ 69]
- Possible use of not initialized reference in the implementation part => SA0145 [▶ 71]

Interfaces:

- Possible use of not initialized interface in the implementation part => SA0046 [▶ 70]

SA0046: Possible use of not initialized interfaces

Function	Determines the use of interfaces that may not have been initialized before the use.
Reason	An interface reference should be checked for <> 0 before it is used, otherwise an access violation may occur at runtime.
Importance	High

Samples:

Interface I_Sample:

```
INTERFACE I_Sample
```

```
METHOD SampleMethod : BOOL
VAR_INPUT
    nInput  : INT;
END_VAR
```

Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample IMPLEMENTS I_Sample
```

```
METHOD SampleMethod : BOOL
VAR_INPUT
    nInput  : INT;
END_VAR
```

Program MAIN:

```
PROGRAM MAIN
VAR
    fbSample       : FB_Sample;
    iSample        : I_Sample;
    iSampleNotSet : I_Sample;
    nParam         : INT;
    bReturn        : BOOL;
END_VAR
```

```
iSample  := fbSample;
bReturn  := iSample.SampleMethod(nInput := nParam);         // no error

bReturn  := iSampleNotSet.SampleMethod(nInput := nParam);   // => SA0046
```

Overview of the rules on "dereferencing".

Pointers:

- Dereferencing of pointers in the declaration part => SA0124 [▶ 68]
- Possible null pointer dereferences in the implementation part => SA0039 [▶ 69]

References:

- Use of references in the declaration part => SA0125 [▶ 69]
- Possible use of not initialized reference in the implementation part => SA0145 [▶ 71]

Interfaces:

- Possible use of not initialized interface in the implementation part => SA0046 [▶ 70]

SA0145: Possible use of not initialized references

Function	Determines all reference variables that may not be initialized before they are used and were not checked by the __ISVALIDREF operator. This rule is applied in the implementation part of POUs.
Reason	A reference should be checked for validity before it is accessed, otherwise an access violation may occur at runtime.
Importance	High

Samples:

```
FUNCTION_BLOCK FB_Test
VAR_INPUT
    pStruct    : POINTER TO ST_Test;
    refStruct  : REFERENCE TO ST_Test;
END_VAR
VAR
    bPointer   : BOOL := pStruct^.bTest;  // => SA0124: Dereference access in initialization
    bRef       : BOOL := refStruct.bTest; // => SA0125: Reference used in initialization
END_VAR
```

```
bPointer := pStruct^.bTest;               // => SA0039: Possible null pointer dereference 'pStruct^'
bRef     := refStruct.bTest;              // => SA0145: Possible use of not initialized reference
'refStruct'

IF pStruct <> 0 THEN
    bPointer := pStruct^.bTest;           // no error SA0039 as the pointer is checked for unequal 0
END_IF

IF __ISVALIDREF(refStruct) THEN
    bRef     := refStruct.bTest;          // no error SA0145 as the reference is checked via
__ISVALIDREF
END_IF
```

Overview of the rules on "dereferencing".

Pointers:

- Dereferencing of pointers in the declaration part => SA0124 [▶ 68]
- Possible null pointer dereferences in the implementation part => SA0039 [▶ 69]

References:

- Use of references in the declaration part => SA0125 [▶ 69]
- Possible use of not initialized reference in the implementation part => SA0145 [▶ 71]

Interfaces:

- Possible use of not initialized interface in the implementation part => SA0046 [▶ 70]

SA0140: Statements commented out

Function	Determines statements that are commented out.
Reason	Code is often commented out for debugging purposes. When such a comment is enabled, it is not clear at a later point in time whether the code should be deleted or whether it was only commented out for debugging purposes and was inadvertently not commented in again.
Importance	High
PLCopen rule	C4

Sample:

```
//bStart := TRUE;                // => SA0140
```

SA0150: Violations of lower or upper limits of the metrics

Function	Determines function blocks that violate the enabled metrics at the lower or upper limit.
Reason	Code that adheres to certain metrics is easier to read, easier to maintain and easier to test.
Importance	High
PLCopen rule	CP9

Sample:

The metric "Number of calls" is enabled and configured in the metrics configuration enabled (PLC Project Properties > category "Static Analysis" > "Metrics" tab).

- Lower limit: 0
- Upper limit: 3
- Function block Prog1 is called 5 times

During the execution of the Static Analysis the violation of SA0150 is issued as an error or warning in the message window.

```
// => SA0150: Metric violation for 'Prog1'. Result for metric 'Calls' (5) > 3"
```

SA0160: Recursive calls

Function	Determines recursive calls of programs, actions, methods and properties. Determines possible recursions through virtual function calls and interface calls.
Reason	Recursions lead to non-deterministic behavior and are therefore a source of errors.
Importance	Medium
PLCopen rule	CP13

Sample 1:

Method FB_Sample.SampleMethod1:

```
METHOD SampleMethod1
VAR_INPUT
END_VAR
```

```
SampleMethod1(); (* => SA0160: Recursive call:
                              'MAIN -> FB_Sample.SampleMethod1 -> FB_Sample.SampleMethod1' *)
```

Method FB_Sample.SampleMethod2:

```
METHOD SampleMethod2 : BOOL
VAR_INPUT
END_VAR
```

```
SampleMethod2 := THIS^.SampleMethod2();(* => SA0160: Recursive call:
                                                    'MAIN -> FB_Sample.SampleMethod2 ->
FB_Sample.SampleMethod2' *)
```

Program MAIN:

```
PROGRAM MAIN
VAR
    fbSample : FB_Sample;
    bReturn  : BOOL;
END_VAR
```

```
fbSample.SampleMethod1();
bReturn := fbSample.SampleMethod2();
```

Sample 2:

Please note regarding properties:

For a property, a local input variable is implicitly created with the name of the property. The following Set function of a property thus assigns the value of the implicit local input variables to the property of an FB variable.

Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR
    nParameter : INT;
END_VAR
```

Set function of the property SampleProperty:

```
nParameter := SampleProperty;
```

In the following Set function, the implicit input variable of the property is assigned to itself. The assignment of a variable to itself does not constitute a recursion, so that this Set function does not generate an SA0160 error.

Set function of the property SampleProperty:

```
SampleProperty := SampleProperty;              // no error SA0160
```

However, access to a property using the THIS pointer is qualified. By using the THIS pointer, the instance and thus the property is accessed, rather than the implicit local input variable. This means that the shading of implicit local input variables and the property itself is lifted. In the following Set function, a new call to the property is generated, which leads to a recursion and thus to error SA0160.

Set function of the property SampleProperty:

```
THIS^.SampleProperty := SampleProperty;        // => SA0160
```

SA0161: Unpacked structure in packed structure

Function	Determines unpacked structures that are used in packed structures.
Reason	An unpacked structure is usually placed by the compiler on an address that allows aligned access to all elements within the structure. If you create this structure in a packed structure, aligned access is no longer possible, and access to an element in the unpacked structure can lead to a misalignment exception at runtime.
Importance	High

Sample:

The structure ST_SingleDataRecord is packed but contains instances of the unpacked structures ST_4Byte and ST_9Byte. This results in a SA0161 error message.

```
{attribute 'pack_mode' := '1'}
TYPE ST_SingleDataRecord :
STRUCT
    st9Byte          : ST_9Byte; // => SA0161
    st4Byte          : ST_4Byte; // => SA0161
    n1               : UDINT;
    n2               : UDINT;
    n3               : UDINT;
    n4               : UDINT;
END_STRUCT
END_TYPE
```

Structure ST_9Byte:

```
TYPE ST_9Byte :
STRUCT
    nRotorSlots      : USINT;
    nMaxCurrent      : UINT;
    nVelocity        : USINT;
    nAcceleration    : UINT;
    nDeceleration    : UINT;
    nDirectionChange : USINT;
END_STRUCT
END_TYPE
```

Structure ST_4Byte:

```
TYPE ST_4Byte :
STRUCT
    fDummy           : REAL;
END_STRUCT
END_TYPE
```

SA0162: Missing comments

Function	Detects uncommented locations in the program. Comments are required for: • the declaration of variables. The comments are shown above or to the right. • the declaration of POUs, DUTs, GVLs or interfaces. The comments are shown above the declaration (in the first row).
Reason	Full commentary is required by many programming guidelines. It increases the readability and maintainability of the code.
Importance	Low
PLCopen rule	C2

Samples:

The following sample generates the error "SA0162: Missing comment for 'b1'" for variable b1.

```
// Comment for MAIN program
PROGRAM MAIN
VAR
    b1  : BOOL;
    // Comment for variable b2
    b2  : BOOL;
    b3  : BOOL;                  // Comment for variable b3
END_VAR
```

SA0163: Nested comments

Function	Determines code positions with nested comments.
Reason	Nested comments are difficult to read and should be avoided.
Importance	Low
PLCopen rule	C3

Samples:

The four nested comments identified accordingly in the following sample each result in the error: "SA0163: Nested comment '<…>'".

```
(* That is
(* nested comment number 1 *)
*)
PROGRAM MAIN
VAR
    (* That is
    // nested comment
    number 2 *)
    a         : DINT;
    b         : DINT;

    (* That is
    (* nested comment number 3 *) *)
    c         : BOOL;
    nCounter : INT;
END_VAR
```

```
(* That is // nested comment number 4 *)

nCounter := nCounter + 1;

(* This is not a nested comment *)
```

SA0164: Multi-line comments

Function	Determines code positions at which the multi-line comment operator (* *) is used. Only the two single-line comment operators are allowed: // for standard comments, /// for documentation comments.
Reason	Some programming guidelines prohibit multi-line comments in the code, because the beginning and end of a comment could get out of sight and the closing comment bracket could be deleted by mistake.
Importance	Low
PLCopen rule	C5

You can disable this check with the pragma {analysis ...} [▶ 109], including for comments in the declaration part.

Samples:

```
(*
This comment leads to error:
"SA0164 …"
*)
PROGRAM MAIN
VAR
    /// Documentation comment not reported by SA0164
    nCounter1: DINT;
    nCounter2: DINT;              // Standard single-line comment not reported by SA0164
END_VAR
```

```
(* This comment leads to error: "SA0164 …" *)
nCounter1 := nCounter1 + 1;
nCounter2 := nCounter2 + 1;
```

SA0166: Maximum number of input/output/VAR_IN_OUT variables

Function	The check determines whether a defined number of input variables (VAR_INPUT), output variables (VAR_OUTPUT) or VAR_IN_OUT variables is exceeded in a function block. You can configure the parameters that are taken into account in the check by double-clicking on the row for rule 166 in the rule configuration (PLC Project Properties > category "Static Analysis" > "Rules" tab > Rule 166). You can make the following settings in the dialog that appears: • Maximum number of inputs (default value: 10) • Maximum number of outputs (default value: 10) • Maximum number of inputs/outputs (default value: 10)
Reason	This is about checking individual programming guidelines. Many programming guidelines stipulate a maximum number of parameters for function blocks. Too many parameters make the code unreadable and the function blocks difficult to test.
Importance	Medium
PLCopen rule	CP23

Sample:

Rule 166 is configured with the following parameters:

- Maximum number of inputs: 0
- Maximum number of outputs: 10
- Maximum number of inputs/outputs: 1

The following function block therefore reports two SA0166 errors, since too many inputs (> 0) and too many inputs/outputs (> 1) are declared.

Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample         // => SA0166
VAR_INPUT
    bIn      : BOOL;
END_VAR
VAR_OUTPUT
    bOut     : BOOL;
END_VAR
VAR_IN_OUT
    bInOut1 : BOOL;
    bInOut2 : BOOL;
END_VAR
```

SA0167: Report temporary FunctionBlock instances

Function	Determines function block instances that are declared as temporary variables. This applies to instances that are declared in a method, in a function or as VAR_TEMP, and which are reinitialized in each processing cycle or each function block call.
Reason	Function blocks have a state that is usually retained over several PLC cycles. An instance on the stack exists only for the duration of the function call. It is therefore only rarely useful to create an instance as a temporary variable. Secondly, function block instances are frequently large and require a great deal of space on the stack (which is usually limited on controllers). Thirdly, the initialization and often also the scheduling of the function block can take up quite a lot of time.
Importance	Medium

Samples:

Method FB_Sample.SampleMethod:

```
METHOD SampleMethod : INT
VAR_INPUT
END_VAR
VAR
    fbTrigger : R_TRIG;           // => SA0167
END_VAR
```

Function F_Sample:

```
FUNCTION F_Sample : INT
VAR_INPUT
END_VAR
VAR
    fbSample  : FB_Sample;        // => SA0167
END_VAR
```

Program MAIN:

```
PROGRAM MAIN
VAR_TEMP
    fbSample  : FB_Sample;        // => SA0167
    nReturn   : INT;
END_VAR
```

```
nReturn := F_Sample();
```

SA0168: Unnecessary assignments

Function	Determines assignments to variables that have no effects in the code.
Reason	If several values are assigned to a variable without the variable being evaluated between the assignments, the first assignments do not have any effect on the program.
Importance	Low

Sample:

```
PROGRAM MAIN
VAR
    nVar1   : DWORD;
    nVar2   : DWORD;
END_VAR
```

```
nVar1 := 1;

IF nVar2 > 100 THEN
    nVar2 := 0;
    nVar2 := nVar2 + 1;
END_IF

nVar1 := 2;                       // => SA0168
```

SA0169: Ignored outputs

Function	Determines the outputs of methods and functions that are not specified when calling the method or function.
Reason	Ignored outputs can be an indication of unhandled errors or nonsensical function calls, as the results are not used.
Importance	Medium

Sample:

Function F_Sample:

```
FUNCTION F_Sample : BOOL
VAR_INPUT
    bIn     : BOOL;
END_VAR
VAR_OUTPUT
    bOut    : BOOL;
END_VAR
```

Program MAIN:

```
PROGRAM MAIN
VAR
    bReturn : BOOL;
    bFunOut : BOOL;
END_VAR
```

```
bReturn := F_Sample(bIn := TRUE , bOut => bFunOut);
bReturn := F_Sample(bIn := TRUE);                          // => SA0169
```

SA0170: Address of an output variable should not be used

Function	Detects code locations where the address of an output variable (VAR_OUTPUT, VAR_IN_OUT) of a function block is determined.
Reason	It is not allowed to use the address of a function block output in the following way: • By means of the ADR operator • By means of REF=
Exception	No error is reported if the output is used within the same function block.
Importance	Medium

Sample:

Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR_INPUT
    nIn         : INT;
END_VAR
VAR_OUTPUT
    nOut        : INT;
END_VAR
VAR
    pFB         : POINTER TO FB_Sample;
    pINT        : POINTER TO INT;
END_VAR
```

```
IF pFB <> 0 THEN
    pINT := ADR(pFB^.nOut);                     // => SA0170
END_IF

nOut := nIn;
pINT := ADR(THIS^.nOut);                        // no error due to internal usage
pINT := ADR(nOut);                              // no error due to internal usage
```

Accesses within another function block, in this case in the MAIN program:

```
PROGRAM MAIN
VAR
    fbSample     : FB_Sample;
    pExternal    : POINTER TO INT;
    refExternal  : REFERENCE TO INT;
END_VAR
```

```
pExternal   := ADR(fbSample.nOut);              // => SA0170
refExternal REF= fbSample.nOut;                 // => SA0170
```

SA0171: Enumerations should have the 'strict' attribute

Function	Detects declarations of enumerations which are not provided with the {attribute 'strict'} attribute.
Reason	The {attribute 'strict'} attribute causes compiler errors to be issued if the code violates strict programming rules for enumerations. By default, when a new enumeration is created, the declaration is automatically assigned the 'strict' attribute.
Importance	High

For more information see: PLC > Reference programming > Pragmas > Attribute pragmas > Attribute 'strict'

Sample:

```
{attribute 'qualified_only'}
{attribute 'strict'}
TYPE E_TrafficLight :
(
    eRed := 0,
    eYellow,
    eGreen
);
END_TYPE
```

```
{attribute 'qualified_only'}
TYPE E_MachineStates :            // => SA0171
(
    eStopped := 0,
    eRunning,
    eError
);
END_TYPE
```

SA0175: Suspicious operation on string

Function	Determines code positions that are suspicious for UTF-8 encoding.

Captured constructs	1. Index access to a single-byte string
	◦ Sample: `sVar[2]`
	◦ Message: Suspicious operation on string: index access '<expression>'
	2. Address access to a single-byte string
	◦ Sample: `ADR(sVar)`
	◦ Message: Suspicious operation on string: Possible index access '<expression>'
	3. Call of a string function of the Tc2_Standard library except CONCAT and LEN
	◦ Sample: `FIND(sVar, 'a');`
	◦ Message: Suspicious operation on string: Possible index access '<expression>'
	4. Single byte literal containing non-ASCII characters
	◦ Samples:
	`sVar := '99€';`
	`sVar := 'Ä';`
	◦ Message: Suspicious operation on string: literal '<literal>' contains non-ASCII characters

Importance	Medium

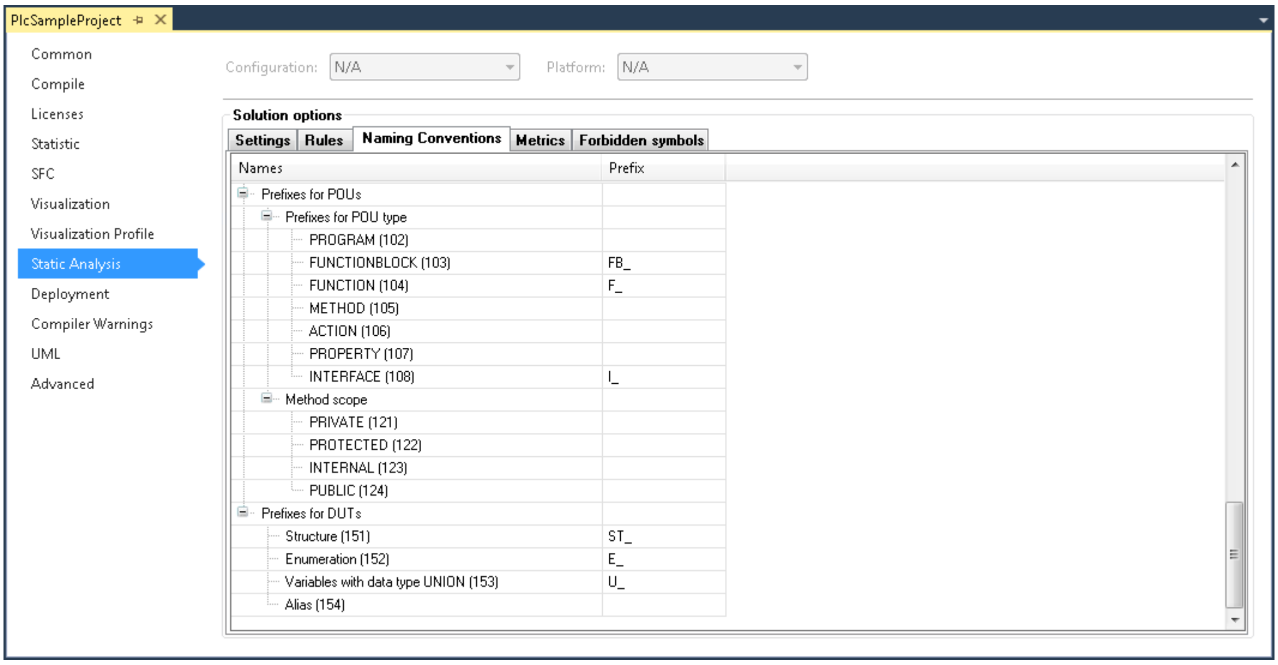
Samples:

```
VAR
    sVar  : STRING;
    pVar  : POINTER TO STRING;
    nVar  : INT;
END_VAR
```

```
// 1) SA0175: Suspicious operation on string: Index access
sVar[2];                          // => SA0175

// 2) SA0175: Suspicious operation on string: Possible index access
pVar := ADR(sVar);                // => SA0175

// 3) SA0175: Suspicious operation on string: Possible index access
nVar := FIND(sVar, 'a');          // => SA0175

// 4) SA0175: Suspicious operation on string: Literal '<...>' contains Non-ASCII character
sVar := '99€';                    // => SA0175
sVar := 'Ä';                      // => SA0175
```

4.3 Naming conventions

In the **Naming Conventions** tab you can define naming conventions. Their compliance is accounted for in the Static Analysis execution [▶ 100]. You define mandatory prefixes for the different data types of variables as well as for different scopes, function block types, and data type declarations. The names of all objects for which a convention can be specified are displayed in the project properties as a tree structure. The objects are arranged below organizational nodes.



Names	Prefix
Prefixes for POUs	
Prefixes for POU type	
PROGRAM (102)	
FUNCTIONBLOCK (103)	FB_
FUNCTION (104)	F_
METHOD (105)	
ACTION (106)	
PROPERTY (107)	
INTERFACE (108)	I_
Method scope	
PRIVATE (121)	
PROTECTED (122)	
INTERNAL (123)	
PUBLIC (124)	
Prefixes for DUTs	
Structure (151)	ST_
Enumeration (152)	E_
Variables with data type UNION (153)	U_
Alias (154)	

Configuration of the naming conventions

Names	Nodes and elements for which a prefix can be defined The number in brackets after each element, for example "PROGRAM (102)", is the prefix convention number that is output if the naming convention is not followed.
Prefix	You can define the naming conventions by entering the required prefix in this column. Please note the following notes and options: • Several possible prefixes per line ◦ Multiple prefixes can be entered separated by commas. ◦ Example: "x, b" as prefixes for variables of data type BOOL. "x" and "b" may be used as prefix for Boolean variables. • Regular expressions ◦ You can also use regular expressions (RegEx) for the prefix. In this case you have to use @ as additional prefix. ◦ Example: "@b[a-dA-D]" as prefix for variables of data type BOOL. The name of the boolean variable must start with "b", and may be followed by a character in the range "a-dA-D". • Data type placeholder ◦ For variables of the Alias data type and for properties you can use the data type placeholder "{datatype}" as prefix. ◦ Example: Prefix for the variable data type Alias (33) = "{datatype}"
Prefixes for variables	Organizational node for all variables for which a prefix dependent on their data type or scope can be defined
Prefixes for POUs	Organizational node for all POU types and method validity ranges for which a prefix can be defined
Prefixes for DUTs	Organizational node for the DUT data types Structure, Enumeration, Alias or Union for which a prefix can be defined
Prefixes for user-defined types (NC0160)	Available from TC3.1 Build 4026 Organizational node for special user-defined types, especially those from libraries or for read-only types (e.g. PVOID, HRESULT) • You can expand the list with conventions: click the blank line below. Then enter the name of a user-defined type or select a user-defined type in the "Input Assistant" dialog. • You can delete a convention by selecting it and choosing the [Del] key. Note: These conventions take priority over the prefixes defined with the {attribute 'nameprefix' := '<prefix>'} attribute. Example: • In the "Name" column, enter the read-only system data type "PVOID" in an empty line below the prefixes for user-defined types. In the same line in the "Prefix" column, enter the desired prefix, e.g. "p". Variables of type PVOID are checked for this prefix when running Static Analysis. • More examples of user-defined types whose desired prefix you can configure at this point: ◦ System data type HRESULT ◦ TON function block from the Tc2_System library

Formation of the expected prefix

The prefix expected for the different declarations is formed depending on the configuration of the options found in the Naming conventions (2) [▸ 90] dialog.

On the Naming conventions (2) [▸ 90] page you will also find explanations on how the expected prefix is formed, as well as some samples.

Placeholder {datatype} with alias variables and properties

Please also note the possibilities of the placeholder {datatype} [▸ 89], which you can use for the prefix definition of alias variables and properties.

Local prefix definition for structured types

For variables of structured types, you can specify a prefix locally in the data type declaration using the 'nameprefix' attribute [▸ 111].

Syntax of convention violations in the message window

Each naming convention has a unique number (shown in parentheses after the convention in the naming convention configuration view). If a violation of a convention or a preset is detected during the static analysis, the number is output in the error list together with an error description based on the following syntax. The abbreviation "NC" stands for "Naming Convention".

Syntax: **"NC<prefix convention number>: <convention description>"**

Sample for convention number 151 (DUTs of type Structure): "NC0151: Invalid type name 'STR_Sample'. Expected prefix 'ST_'"

Temporary deactivation of naming conventions

Individual conventions can be disabled temporarily, i.e. for particular code lines. To this end you can add a pragma or an attribute in the declaration or implementation part of the code. For variables of structured types you may specify a prefix locally via an attribute in the data type declaration. For more information see: Pragmas and attributes [▸ 108].

Overview of naming conventions

For an overview of naming conventions, see Naming conventions – overview and description [▸ 82].

4.3.1 Naming conventions – overview and description

Overview

- Prefixes for variables

- Prefixes for types

- NC0003: BOOL [▸ 85]
- NC0004: BIT [▸ 85]
- NC0005: BYTE [▸ 85]
- NC0006: WORD [▸ 85]
- NC0007: DWORD [▸ 85]
- NC0008: LWORD [▸ 85]
- NC0013: SINT [▸ 85]

  - NC0014: INT [▶ 85]
  - NC0015: DINT [▶ 85]
  - NC0016: LINT [▶ 85]
  - NC0009: USINT [▶ 85]
  - NC0010: UINT [▶ 85]
  - NC0011: UDINT [▶ 85]
  - NC0012: ULINT [▶ 85]
  - NC0017: REAL [▶ 85]
  - NC0018: LREAL [▶ 85]
  - NC0019: STRING [▶ 85]
  - NC0020: WSTRING [▶ 85]
  - NC0021: TIME [▶ 85]
  - NC0022: LTIME [▶ 85]
  - NC0023: DATE [▶ 85]
  - NC0024: DATE_AND_TIME [▶ 85]
  - NC0025: TIME_OF_DAY [▶ 85]
  - NC0026: POINTER [▶ 85]
  - NC0027: REFERENCE [▶ 85]
  - NC0028: SUBRANGE [▶ 86]
  - NC0030: ARRAY [▶ 86]
  - NC0031: Function block instance [▶ 86]
  - NC0036: Interface [▶ 86]
  - NC0032: Structure [▶ 87]
  - NC0029: ENUM [▶ 87]
  - NC0033: Alias [▶ 87]
  - NC0034: Union [▶ 87]
  - NC0035: __XWORD [▶ 85]
  - NC0037: __UXINT [▶ 85]
  - NC0038: __XINT [▶ 85]
- **Prefixes for scopes**
  - NC0051: VAR_GLOBAL [▶ 88]
  - NC0070: VAR_GLOBAL CONSTANT [▶ 88]
  - NC0071: VAR_GLOBAL RETAIN [▶ 88]

- NC0072: VAR_GLOBAL PERSISTENT [▶ 88]
- NC0073: VAR_GLOBAL RETAIN PERSISTENT [▶ 88]
- **VAR**
    - NC0053: Program variables [▶ 88]
    - NC0054: Function block variables [▶ 88]
    - NC0055: Function/method variables [▶ 88]
- NC0056: VAR_INPUT [▶ 88]
- NC0057: VAR_OUTPUT [▶ 88]
- NC0058: VAR_IN_OUT [▶ 88]
- NC0059: VAR_STAT [▶ 88]
- NC0061: VAR_TEMP [▶ 88]
- NC0062: VAR CONSTANT [▶ 88]
- NC0063: VAR PERSISTENT [▶ 88]
- NC0064: VAR RETAIN [▶ 88]
- NC0065: I/O variables [▶ 88]

- **Prefixes for POUs**
    - **Prefixes for POU type**
        - NC0102: PROGRAM [▶ 88]
        - NC0103: FUNCTIONBLOCK [▶ 88]
        - NC0104: FUNCTION [▶ 88]
        - NC0105: METHOD [▶ 88]
        - NC0106: ACTION [▶ 88]
        - NC0107: PROPERTY [▶ 88]
        - NC0108: INTERFACE [▶ 88]
    - **Method/property scope**
        - NC0121: PRIVATE [▶ 89]
        - NC0122: PROTECTED [▶ 89]
        - NC0123: INTERNAL [▶ 89]
        - NC0124: PUBLIC [▶ 89]
- **Prefixes for DUTs**
    - NC0151: Structure [▶ 89]
    - NC0152: Enumeration [▶ 89]
    - NC0153: Union [▶ 89]
    - NC0154: Alias [▶ 89]

- Prefixes for user-defined types

- NC0160: User-defined type [▸ 89]

Detailed description

The following sections contain explanations and examples of which declarations (i.e. at which point in the project) use the individual naming conventions. The declarations samples illustrate cases for which the corresponding prefix would be expected if a prefix was defined with the corresponding naming convention. It should become clear where and how a type or variable can be declared so that the naming convention NC<xxxx> is checked at this point. However, the samples do not show which concrete prefix is defined for the individual naming conventions and would therefore be expected in the sample declarations. There is therefore no OK/NOK comparison.

For concrete examples with a defined prefix, please refer to the page Naming conventions (2) [▸ 90].

Basic data types:

NC0003: BOOL

Configuration of a prefix for a variable declaration of type BOOL.

Sample declarations:

For the following variable declarations the prefix configured for NC0003 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [▸ 100].

```
bStatus          : BOOL;
abVar            : ARRAY[1..2] OF BOOL;
IbInput   AT%I*  : BOOL;
```

The description of "NC0003: BOOL" is transferrable to the other basic data types:

- NC0004: BIT, NC0005: BYTE

- NC0006: WORD, NC0007: DWORD, NC0008: LWORD

- NC0013: SINT, NC0014: INT, NC0015: DINT, NC0016: LINT, NC0009: USINT, NC0010: UINT, NC0011: UDINT, NC0012: ULINT

- NC0017: REAL, NC0018: LREAL

- NC0019: STRING, NC0020: WSTRING

- NC0021: TIME, NC0022: LTIME, NC0023: DATE, NC0024: DATE_AND_TIME, NC0025: TIME_OF_DAY

- NC0035: __XWORD, NC0037: __UXINT, NC0038: __XINT

Nested data types:

NC0026: POINTER

Configuration of a prefix for a variable declaration of type POINTER TO.

Sample declaration:

For the following variable declaration the prefix configured for NC0026 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [▸ 100].

```
pnID : POINTER TO INT;
```

NC0027: REFERENCE

Configuration of a prefix for a variable declaration of type REFERENCE TO.

Sample declaration:

For the following variable declaration the prefix configured for NC0027 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [▸ 100].

```
reffCurrentPosition  : REFERENCE TO REAL;
```

NC0028: SUBRANGE

Configuration of a prefix for a variable declaration of a subrange type. A subrange type is a data type whose value range only covers a subset of a base type.

Possible basic data types for a subrange type: SINT, USINT, INT, UINT, DINT, UDINT, BYTE, WORD, DWORD, LINT, ULINT, LWORD.

Sample declarations:

For the following variable declaration the prefix configured for NC0028 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [▸ 100].

```
subiRange   : INT(3..5);
sublwRange : LWORD(100..150);
```

NC0030: ARRAY

Configuration of a prefix for a variable declaration of type ARRAY[…] OF.

Sample declaration:

For the following variable declaration the prefix configured for NC0030 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [▸ 100].

```
anTargetPositions  : ARRAY[1..10] OF INT;
```

Instance-based data types:

NC0031: Function block instance

Configuration of a prefix for a variable declaration of a function block type.

Sample declaration:

Declaration of a function block:

```
FUNCTION_BLOCK FB_Sample
…
```

For the following variable declaration the prefix configured for NC0031 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [▸ 100].

```
fbSample  : FB_Sample;
```

NC0036: Interface

Configuration of a prefix for a variable declaration of an interface type.

Sample declaration:

Interface declaration:

```
INTERFACE I_Sample
```

For the following variable declaration the prefix configured for NC0036 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [▸ 100].

```
iSample  : I_Sample;
```

NC0032: Structure

Configuration of a prefix for a variable declaration of a structure type.

Sample declaration:

Declaration of a structure:

```
TYPE ST_Sample :
STRUCT
    bVar   : BOOL;
    sVar   : STRING;
END_STRUCT
END_TYPE
```

For the following variable declaration the prefix configured for NC0032 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [▶ 100].

```
stSample   : ST_Sample;
```

NC0029: ENUM

Configuration of a prefix for a variable declaration of an enumeration type.

Sample declaration:

Declaration of an enumeration:

```
TYPE E_Sample :
(
    eMember1 := 1,
    eMember2
);
END_TYPE
```

For the following variable declaration the prefix configured for NC0029 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [▶ 100].

```
eSample   : E_Sample;
```

NC0033: Alias

Configuration of a prefix for a variable declaration of an alias type.

Sample declaration:

Declaration of an alias:

```
TYPE T_Message : STRING; END_TYPE
```

For the following variable declaration the prefix configured for NC0033 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [▶ 100].

```
tMessage   : T_Message;
```

NC0034: Union

Configuration of a prefix for a variable declaration of a union type.

Sample declaration:

Declaration of a union:

```
TYPE U_Sample :
UNION
    n1   : WORD;
    n2   : INT;
END_UNION
END_TYPE
```

For the following variable declaration the prefix configured for NC0034 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [▶ 100].

```
uSample   : U_Sample;
```

Scopes of variable declarations:

NC0051: VAR_GLOBAL

Configuration of a prefix for a variable declaration between the keywords VAR_GLOBAL and END_VAR.

Sample declaration:

For the following declaration of a global variable, the prefix configured for NC0051 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [▸ 100].

```
VAR_GLOBAL
    gbErrorAcknowledge : BOOL;
END_VAR
```

The description of "NC0051: VAR_GLOBAL" is transferrable to other scopes of variable declarations:

- NC0070: VAR_GLOBAL CONSTANT

- NC0071: VAR_GLOBAL RETAIN

- NC0072: VAR_GLOBAL PERSISTENT

- NC0073: VAR_GLOBAL RETAIN PERSISTENT

- NC0053: Program variables (VAR within a program)

- NC0054: Function block variables (VAR within a function block)

- NC0055: Function/method variables (VAR within a function/method)

- NC0056: VAR_INPUT

- NC0057: VAR_OUTPUT

- NC0058: VAR_IN_OUT

- NC0059: VAR_STAT

- NC0061: VAR_TEMP

- NC0062: VAR CONSTANT

- NC0063: VAR PERSISTENT

- NC0064: VAR RETAIN

NC0065: I/O variables

Configuration of a prefix for a variable declaration with AT declaration.

Sample declarations:

For the following variable declarations with AT declaration, the prefix configured for NC0065 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [▸ 100].

```
ioVar1    AT%I*      : INT;
ioVar2    AT%IX1.0   : BOOL;
ioVar3    AT%Q*      : INT;
ioVar4    AT%QX2.0   : BOOL;
```

POU types:

NC0102: PROGRAM

Configuration of a prefix for the declaration of a program (name of the program in the project tree).

The description of "NC0102: PROGRAM" is transferrable to the other POU types:

- NC0103: FUNCTIONBLOCK
- NC0104: FUNCTION
- NC0105: METHOD
- NC0106: ACTION
- NC0107: PROPERTY
- NC0108: INTERFACE

Scopes of methods and properties:

NC0121: PRIVATE

Configuration of a prefix for the declaration of a method or a property (name of the method/property in the project tree), whose access modifier is PRIVATE.

The description of "NC121: PRIVATE" is transferrable to the other scopes of methods and properties:

- NC0122: PROTECTED
- NC0123: INTERNAL
- NC0124: PUBLIC

DUTs:

NC0151: Structure

Configuration of a prefix for the declaration of a structure (name of the structure in the project tree).

The description of "NC0151: Structure" is transferrable to the other DUT types:

- NC0152: Enumeration
- NC0153: Union
- NC0154: Alias

User-defined types:

NC0160: User defined type

Configuration of a prefix for a user-defined type, e.g. for variables of type PVOID or for instances of the library function block Tc2_System.TON.

For more information on the input options in this area, visit Naming conventions [▸ 80].

4.3.2 Placeholder {datatype}

For variables of type Alias and for properties, the placeholder "{datatype}" can be defined as a prefix in the "Naming Conventions" tab. The placeholder {datatype} is thereby replaced by the prefix that is defined for the data type of the alias or for the data type of the property. The static analysis thus reports errors for all alias variables that do not possess the prefix for the data type of the alias or for all properties that do not possess the prefix for the data type of the property.

The placeholder "{datatype}" can also be combined with further prefixes in the prefix definition, e.g. to "P_{datatype}_".

Example 1 for an alias variable:

- In the project there is an alias "TYPE MyMessageType : STRING; END_TYPE" as well as a variable of this type (var : MyMessageType;).
- Prefix definitions
  - Prefix for the variable data type alias (33) = "{datatype}"
  - Prefix for the variable data type STRING (19) = "s"
- In the prefix definitions mentioned the data type prefix "s" is expected for a variable of the alias type "MyMessageType" (e.g. for the variable "var").

Example 2 for an alias variable:

- Same situation as in example 1 for an alias variable, the only difference being:
  - Prefix for the variable data type alias (33) = "al_{datatype}"
- In this case the data type prefix "al_s" is expected for a variable of the alias type "MyMessageType".

Example of a property:

- Prefix definitions
  - Prefix for the method/property scope PRIVATE (121) = "priv_"
  - Prefix for the POU type PROPERTY (107) = "P_{datatype}"
  - Prefix for the variable data type LREAL (18) = "f"
- Note: For POUs with an access modifier (methods or properties), the combination of the prefix for the scope (NC0121-NC0124: PRIVATE/PROTECTED/INTERNAL/PUBLIC) and the prefix for the POU type (NC0105 for method, NC0107 for property) is expected as the overall prefix.
- With the prefix definitions mentioned the overall prefix "priv_P_f" is thus expected for a property with the access modifier PRIVATE and the data type LREAL.

4.4 Naming conventions (2)

The **Naming Conventions (2)** tab contains options that extend the settings of the Naming conventions [▶ 80] tab. You can use these options to configure how the expected overall prefix for variables/declarations is to be composed.

The observance of the naming conventions is checked during the execution of the Static Analysis [▶ 100].

1) First character after prefix should be an upper case letter

- If enabled: The static code analysis reports an error for a variable if the first character of the variable name after the defined prefix is not an upper case letter.
- If disabled: Upper case/lower case spelling is not checked.
- Default setting: disabled

Examples:

- Variable "bvar" with the expected prefix "b"
- Function block "FB_sample" with the expected prefix "FB_"

Option	State	Result of the static analysis
First character after prefix should be an upper case letter	Enabled	For the definitions mentioned above, an error will be reported in each case that the first letter after the prefix must be an upper case letter. Correct identifiers would be "bVar" and "FB_Sample".
	Disabled	The identifiers "bvar" and "FB_sample" are permissible. No upper/ lower case error is output.

2) Recursive prefixes for combinable data types

- If enabled: Variables of combinable data types (POINTER, REFRENCE, ARRAY, SUBRANGE) must have a composite **data type prefix**. The composite prefix is formed from the individual prefixes configured for the individual components of the combined data type.

- If disabled: Only the prefix of the outermost data type is expected as the **date type prefix**.
- Default setting: enabled
- Examples: see below

3) Combine scope prefix with data type prefix

(namespace = scope)

- If enabled: A variable must have the **prefix for its scope** defined in the naming conventions, followed by its **data type prefix**.
- If disabled: The expected overall prefix depends on whether or not a scope prefix is defined for a variable.
    - If the associated **scope prefix is defined** for a variable, the variable must have **only** the **prefix for its scope** defined in the naming conventions. The **data type prefix** is **not** expected after the scope prefix.
    - If the associated **scope prefix is not defined** for a variable, the variable must have **only** the **data type prefix** defined for it.
- Default setting: enabled
- Examples: see below

Examples

- Prefix configuration for data types:
    - POINTER (26) = "p"
    - ARRAY (30) = "a"
    - INT (14) = "n"
    - BOOL (3) = "b"
- Prefix configuration for scope
    - Case 1: Function block variables (54) = "_local_"
    - Case 2: Function block variables (54) = empty field/not configured

      INFO: Further examples of a scope include VAR_GLOBAL (51), VAR_INPUT (56) and VAR CONSTANT (62).
- Declaration:

```
FUNCTION_BLOCK FB_Sample
VAR
    var1   : POINTER TO ARRAY[1..3] OF INT;
    var2   : ARRAY[10..20] OF ARRAY[3..5] OF BOOL;
END_VAR
```

Option scenario 1:

Option	State	Expected overall prefix for case 1 (NC0054 = "_local_")	Expected overall prefix for case 2 (NC0054 = empty)
Recursive prefixes for combinable data types	Enabled	For var1: '_local_pan' For var2: '_local_aab'	For var1: 'pan' For var2: 'aab'
Combine scope prefix with data type prefix	Enabled		

Explanation:

- As the option "Recursive prefixes for combinable data types" is enabled, the prefix composed of the individual prefixes is expected as the **data type prefix**. Consequently, the sub-prefixes "p" for POINTER, "a" for ARRAY and "n" for INT are combined to form the data type prefix "pan", or the sub-prefixes "a" for ARRAY, "a" for ARRAY again and "b" for BOOL are combined to form the data type prefix "aab".
- As the option "Combine scope prefix with data type prefix" is also enabled, the combination of **scope prefix** and **data type prefix** is expected as the **overall prefix**.

- Case 1: _local_ + pan = _local_pan
- Case 2: <empty> + pan = pan

Option scenario 2:

Option	State	Expected overall prefix for case 1 (NC0054 = "_local_")	Expected overall prefix for case 2 (NC0054 = empty)
Recursive prefixes for combinable data types	Disabled	For var1: '_local_p' For var2: '_local_a'	For var1: 'p' For var2: 'a'
Combine scope prefix with data type prefix	Enabled		

Explanation:

- As the option "Recursive prefixes for combinable data types" is disabled, only the prefix of the outermost data type is expected as the **data type prefix**. The expected data type prefix is therefore "p" or "a".
- As the option "Combine scope prefix with data type prefix" is enabled, the combination of **scope prefix** and **data type prefix** is expected as the **overall prefix** for variables.
  - Case 1: _local_ + p = _local_p
  - Case 2: <empty> + p = p

Option scenario 3:

Option	State	Expected overall prefix for case 1 (NC0054 = "_local_")	Expected overall prefix for case 2 (NC0054 = empty)
Recursive prefixes for combinable data types	Enabled	For var1: '_local_' For var2: '_local_'	For var1: 'pan' For var2: 'aab'
Combine scope prefix with data type prefix	Disabled		

Explanation:

- See option scenario 1: As the option "Recursive prefixes for combinable data types" is enabled, the prefix composed of the individual prefixes is expected as the **data type prefix**. This results in "pan" or "aab" as the data type prefix.
- As the option "Combine scope prefix with data type prefix" is disabled, the expected **overall prefix** depends on whether or not a scope prefix is defined for a variable.
  - If **scope prefix is defined** (case 1): The variable must **only** have the **scope prefix**. The data type prefix is not expected after the scope prefix. This results for both variables in "_local_" as the expected overall prefix.
  - If **scope prefix is not defined** (case 2): The variable must only have the data type prefix. This results in "pan" or "aab" as the expected overall prefix.

Option scenario 4:

Option	State	Expected overall prefix for case 1 (NC0054 = "_local_")	Expected overall prefix for case 2 (NC0054 = empty)
Recursive prefixes for combinable data types	Disabled	For var1: '_local_' For var2: '_local_'	For var1: 'p' For var2: 'a'
Combine scope prefix with data type prefix	Disabled		

Explanation:

- See option scenario 2: As the option "Recursive prefixes for combinable data types" is disabled, only the prefix of the outermost data type is expected as the **data type prefix**. This results in "p" or "a" as the data type prefix.

- As the option "Combine scope prefix with data type prefix" is disabled, the expected **overall prefix** depends on whether or not a scope prefix is defined for a variable.
    - If **scope prefix is defined** (case 1): The variable must **only** have the **scope prefix**. The data type prefix is not expected after the scope prefix. This results for both variables in "_local_" as the expected overall prefix.
    - If **scope prefix is not defined** (case 2): The variable must only have the data type prefix. This results in "p" or "a" as the expected overall prefix.

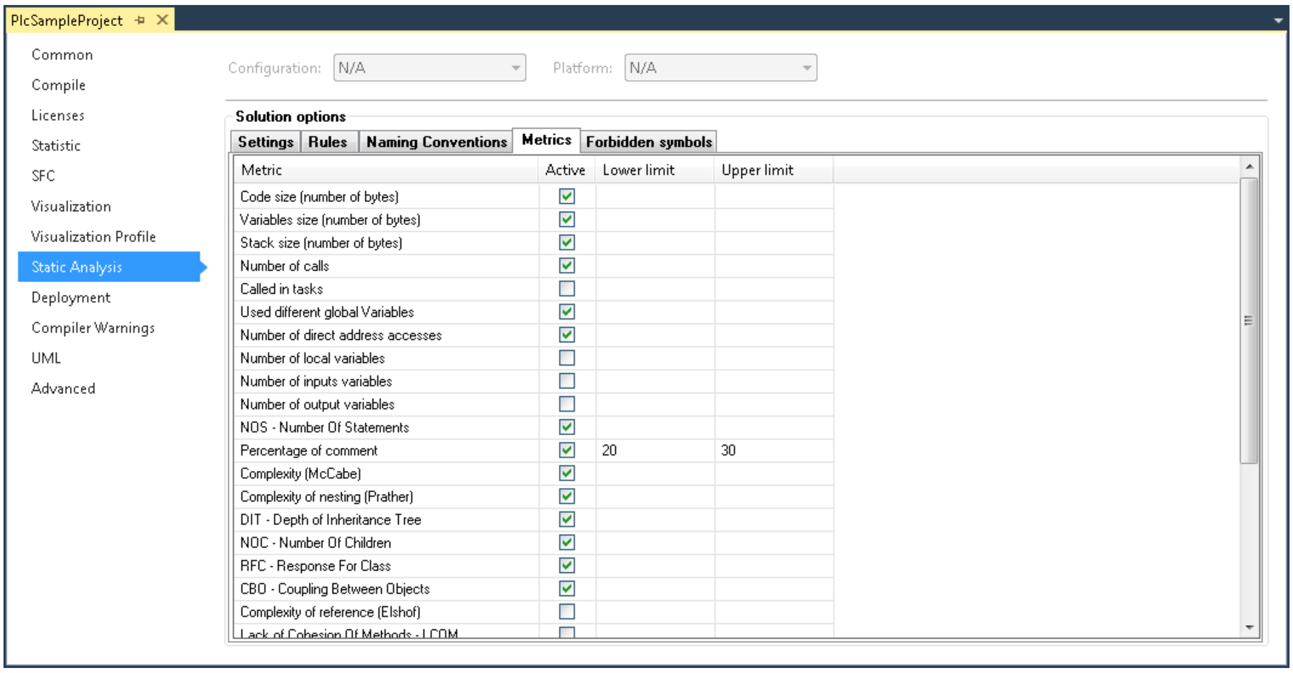
Further notes/examples:

For POUs with an access modifier (methods or properties), the combination of the **prefix for the scope** (NC0121-NC0124: PRIVATE/PROTECTED/INTERNAL/PUBLIC) and the **prefix for the POU type** (NC0105 for method, NC0107 for property) is expected as the **overall prefix**. Examples:

- If the prefix "priv_" has been configured for PRIVATE (121) and the prefix "M_" for METHOD (105), the **overall prefix** "priv_M_" is expected for a PRIVATE method.
- If the prefix "M_" is still configured for METHOD (105), but no prefix has been configured for PRIVATE (121), that is, if the field is empty in the naming conventions, the **overall prefix** "M_" is expected for a PRIVATE method.

4.5 Metrics

In the **Metrics** tab you can select and configure the metrics to be displayed for each function block in the **Standard Metrics** view when the command 'View Standard Metrics' [▸ 103] is executed.



Configuration: N/A Platform: N/A

Solution options

Metric	Active	Lower limit	Upper limit
Code size (number of bytes)	✓		
Variables size (number of bytes)	✓		
Stack size (number of bytes)	✓		
Number of calls	✓		
Called in tasks	☐		
Used different global Variables	✓		
Number of direct address accesses	✓		
Number of local variables	☐		
Number of inputs variables	☐		
Number of output variables	☐		
NOS - Number Of Statements	✓		
Percentage of comment	✓	20	30
Complexity (McCabe)	✓		
Complexity of nesting (Prather)	✓		
DIT - Depth of Inheritance Tree	✓		
NOC - Number Of Children	✓		
RFC - Response For Class	✓		
CBO - Coupling Between Objects	✓		
Complexity of reference (Elshof)	☐		
Lack of Cohesion Of Methods - LCOM	☐		

Analysis of libraries

The following metrics are also output for the libraries integrated in the project: code size, variable size, stack size and number of calls.

Compilation errors for violations of upper/lower limits

You can use rule SA0150 of the static code analysis to output violations of the upper and lower limits of the activated metrics as compilation errors.

Configuration of the metrics

Active	You can enable or disable the individual metrics using the checkbox for the respective row. When command 'View Standard Metrics' [▸ 103] is executed, the metrics that are enabled in the respective configuration are shown for each programming block in the **Standard Metrics** view. • ☐ : The metric is disabled and is not displayed in the **Standard Metrics** view when the command **View Standard Metrics** is executed. • ☑ : The metric is enabled and is displayed in the **Standard Metrics** view when the command **View Standard Metrics** is executed.
Lower limit Upper limit	For each metric you can define an individual upper and lower limit by entering the required number in the respective metric row. If a metric is only limited in one direction, you can leave the configuration for the other direction blank. In other words, you may specify either only the lower limit or only the upper limit.

Evaluation of the upper and lower limits

The set upper and lower limits you can be evaluated in two ways.

- **Standard Metrics** view:
  - Enable the metric whose configured upper and lower limits you want to evaluate.
  - Execute the Command 'View Standard Metrics' [▸ 103].
  - TwinCAT shows the enabled metrics for each programming block in the tabular **Standard Metrics** view.
  - If a value is outside the range defined by an upper and/or lower limit in the configuration, the table cell is shown in red.
- Static Analysis:
  - Enable rule 150 as error or warning in the Rules [▸ 15] tab.
  - Run the Static Analysis (see: Command 'Run static analysis' [▸ 100]).
  - Violations of the upper and/or lower limits are issued as error or warning in the message window.

Overview and description of the metrics

An overview of the metrics and a detailed description of the rules can be found in the next chapter.

4.5.1 Metrics - overview and description

Column abbreviation in Standard Metrics view	Description
Code size	Code size [number of bytes] ("code size") [▸ 95]
Variables size	Variables size [number of bytes] ("variable size") [▸ 95]
Stack size	Stack size [number of bytes] ("stack size") [▸ 96]
Calls	Number of calls ("calls") [▸ 96]
Tasks	Called in tasks ("tasks") [▸ 96]
Globals	Used different global variables ("Globals") [▸ 96]
IOs	Number of direct address accesses ("IOs") [▸ 96]
Locals	Number of local variables ("local") [▸ 96]
Inputs	Number of input variables ("inputs") [▸ 96]
Outputs	Number of output variables ("outputs") [▸ 96]
NOS	NOS - Number Of Statements ("NOS") [▸ 96]
Comments	Percentage of comments ("comments") [▸ 96]
McCabe	Complexity (McCabe) ("McCabe") [▸ 96]
Prather	Complexity of nesting (Prather) ("Prather") [▸ 96]
DIT	DIT - depth of inheritance tree ("DIT") [▸ 97]
NOC	NOC - number of children ("NOC") [▸ 97]
RFC	RFC - response for class ("RFC") [▸ 97]
CBO	CBO - coupling between objects ("CBO") [▸ 97]
Elshof	Complexity of reference (Elshof) ("Elshof") [▸ 97]
LCOM	Lack of cohesion of methods (LCOM) ("LCOM") [▸ 97]
n1 (Halstead)	Halstead – number of different used operators (n1) [▸ 97]
N1 (Halstead)	Halstead – number of operators (N1) [▸ 97]
n2 (Halstead)	Halstead – number of different used operands (n2) [▸ 97]
N2 (Halstead)	Halstead – number of operands (N2) [▸ 97]
HL (Halstead)	Halstead – length (HL) [▸ 97]
HV (Halstead)	Halstead – volume (HV) [▸ 97]
D (Halstead)	Halstead – difficulty (D) [▸ 97]
SFC branches	Number of SFC branches [▸ 98]
SFC steps	Number of SFC steps [▸ 98]

Detailed description

Code size [number of bytes] ("code size")

Code size as number of bytes.

Variables size [number of bytes] ("variable size")

Variables size as number of bytes.

Stack size [number of bytes] ("stack size")

Stack size as number of bytes.

Number of calls ("calls")

Number of function block calls within the application.

Called in tasks ("tasks")

Number of tasks calling the function block.

Used different global variables ("Globals")

Number of different global variables used in the function block.

Number of direct address accesses ("IOs")

Number of IO access operations in the function block = number of all read and write access operations to a direct address.

Example:

The number of direct address access operations for the MAIN program is 2.

```
PROGRAM MAIN
VAR
    OnOutput AT%QB1 : INT;
    nVar            : INT;
END_VAR
```

```
OnOutput := 123;
nVar     := OnOutput;
```

Number of local variables ("local")

Number of local variables in the function block (VAR).

Number input variables ("inputs")

Number of input variables in the function block (VAR_INPUT).

Number output variables ("outputs")

Number of output variables in the function block (VAR_OUTPUT).

Number of statements ("NOS")

NOS: **N**umber **O**f executable **S**tatements

NOS = number of executable statements in the function block

Percentage of comments ("comments")

Comment proportion = number of comments / number of statements in a function block

For the purpose of this definition, statements also include declaration statements, for example.

Complexity (McCabe) ("McCabe")

Complexity = number of binary branches in the control flow graph for the function block (e.g. the number of branches in IF and CASE statements and loops)

Complexity of nesting (Prather) ("Prather")

Nesting weight = statements * nesting depth

Complexity of nesting = nesting weight / number statements

Nesting through IF/ELSEIF or CASE/ELSE statements, for example.

Depth of inheritance tree ("DIT")

DIT: **D**epth of **I**nheritance **T**ree

DIT = inheritance depth or maximum path length from the root to the class under consideration

Number of children ("NOC")

NOC: **N**umber **O**f **C**hildren

NOC = number of child classes or number of direct class specializations

Response for class ("RFC")

RFC: **R**esponse **F**or **C**lass

RFC = number of methods that can potentially be executed, if an object of the class under consideration responds to a received message

The value is used for measuring the complexity (in terms of testability and maintainability). All possible direct and indirect method calls can be reached via associations are taken into account.

Coupling between objects ("CBO")

CBO: **C**oupling **B**etween **O**bjects

CBO = number of classes coupled with the class under consideration

The value is used to indicate the coupling between object classes. Coupling refers to a situation where a class uses instance variables (variables of an instantiated class) and the methods of another class.

Complexity of reference (Elshof) ("Elshof")

Complexity of reference = referenced data (number of variables) / number of data references

Lack of cohesion of methods (LCOM) ("LCOM")

Cohesion = pairs of methods without common instance variables minus pairs of methods with common instance variables

This cohesion value is a measure for the encapsulation of a class. The higher the value, the poorer the encapsulation. Reciprocal method and property calls (without init or exit) are also taken into account.

Halstead ("n1","N1","n2","N2", "HL", "HV", "D")

The following metrics are part of the "Halstead" range:

- Number of different used operators - Halstead (n1)

- Number of operators - Halstead (N1)

- Number of different used operands - Halstead (n2)

- Number of operands - Halstead (N2)

- Length - Halstead (HL)

- Volume - Halstead (HV)

- Difficulty - Halstead (D)

Background information:

- Relationship between operators and operands (number, complexity, test effort)
- Based on the assumption that executable programs consist of operators and operands.
- Operands in TwinCAT: Variables, constants, components, literals and IEC addresses.
- Operators in TwinCAT: keywords, logical and comparison operators, assignments, IF, FOR, BY, ^, ELSE, CASE, case label, BREAK, RETURN, SIN, +, labels, calls, pragmas, conversions, SUPER, THIS, index access, component access etc.

For each program the following basic parameters are formed:

- **Number of different used operators - Halstead (n1),**
  Number of different used operands - Halstead (n2):
  - Number of different used operators (h_1) and operands (h_2); together they form the vocabulary size h.
- **Number of operators - Halstead (N1),**
  Number of operands - Halstead (N2):
  - Number of total used operators (N_1) and operands (N_2); together they form the implementation class N.
- (Language complexity = operators/operator occurrences * operands/operand occurrences)

These parameters are used to calculate the Halstead length (HL) and Halstead volume (HV):

- **Length - Halstead (HL)**,
  Volume - Halstead (HV):
  - $HL = h_1 * \log_2 h_1 + h_2 * \log_2 h_2$
  - $HV = N * \log_2 h$

Various indicators are calculated from the basic parameters:

- **Difficulty - Halstead (D)**:
  - Describes the difficulty to write or understand a program (during a code review, for example)
  - $D = h_1/2 * N2/h_2$
- Effort:
  - E = D*V

The indicators usually match the actual measured values very well. The disadvantage is that the method only applies to individual functions and only measures lexical/textual complexity.

Number of SFC branches

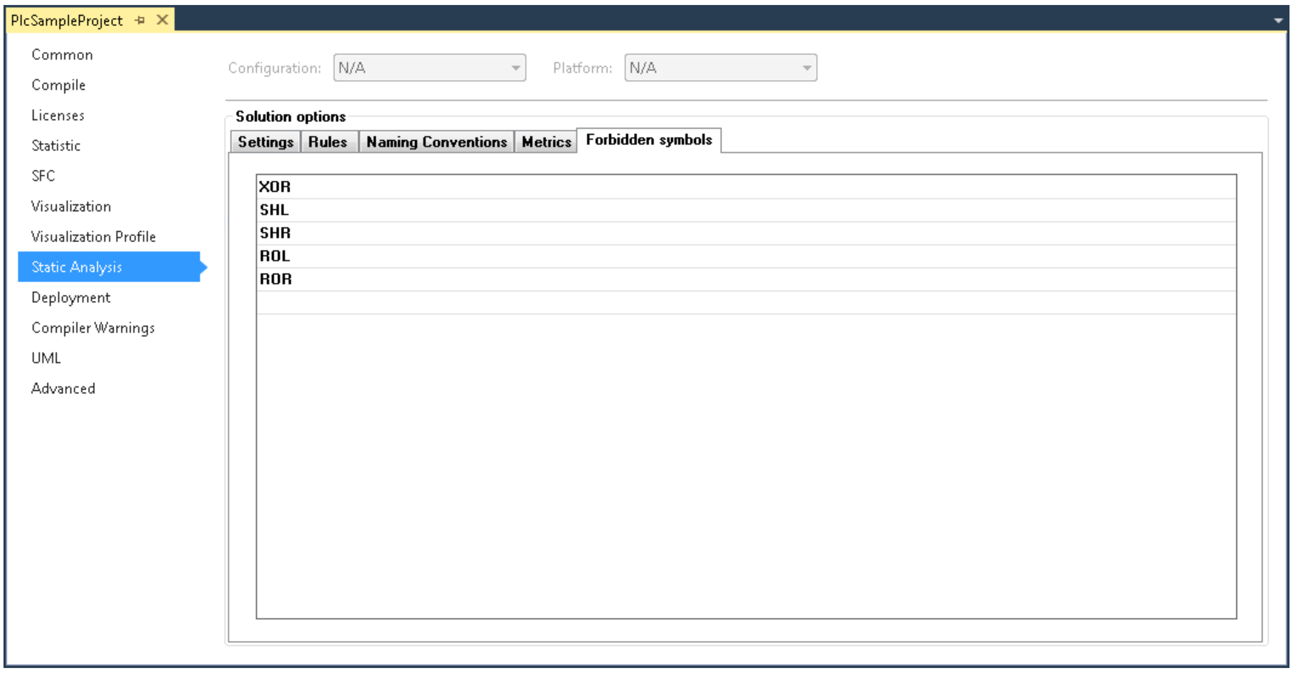
If the function block is implemented in the Sequential Function Chart language (SFC), this code metric indicates the number of branches in the function block.

Number of SFC steps

If the function block is implemented in the Sequential Function Chart language (SFC), this code metric indicates the number of steps in the function block.

4.6 Forbidden symbols

In the **Forbidden symbols** tab, you can specify the keywords, symbols and identifiers that must not be used in the project code. The forbidden symbols are checked during the static analysis [▸ 100].



XOR
SHL
SHR
ROL
ROR

Configuration of forbidden symbols

You can enter these symbols directly in the row or select them via the input assistant. During the static analysis the code is checked for the presence of these terms. Any hits result in an error being issued in the message window.

Syntax of symbol violations in the message window

If a symbol is used in the code that is configured as a forbidden symbol, an error is issued in the message window after the static analysis has been performed.

Syntax: **"Forbidden symbol '<symbol>'"**

Sample for the symbol XOR: "Forbidden symbol 'XOR'"

5 Execution

5.1 Command 'Run static analysis'

Symbol: §

Function: The command starts the static code analysis for the currently active PLC project and outputs the results in the message window.

Call: Build menu or context menu of the PLC project object

During execution of the static analysis, compliance with the coding rules, naming conventions and forbidden symbols is checked. This command can be used to trigger a static analysis manually (explicit execution), or the analysis can be performed automatically during code generation (implicit execution, see below for more information).

TwinCAT issues the result of the static analysis, i.e. messages relating to rule violations, in the message window. The rules [▸ 15], naming conventions [▸ 80] and forbidden symbols [▸ 98] to be taken into account in the static analysis can be configured [▸ 13] in the PLC project properties. You can also define whether the violation of a coding rule should appear as an error or a warning in the message window (see: Rules [▸ 15]).

See also: Syntax in the message window [▸ 101]

Please note that the selected PLC project is created before this command is executed. Checking via the static analysis is only started if the code generation was successful, i.e. if the compiler did not detect any compilation errors.

Please also note the Command 'Run static analysis [Check all objects]' [▸ 102] and the differences between the two commands described in the following table.

Differences	Command 'Run static analysis'	Command 'Run static analysis [Check all objects]'
Scope	The objects used in the PLC project are checked. Objects that are not used are not checked with this command. The scope of this command thus corresponds to the build commands **Build Project/Solution** or **Rebuild Project/Solution** respectively. If you also wish to have the unused objects checked by the static analysis, which is useful, for example, when processing library projects, you can use the command **Run static analysis [check all objects]**.	All objects in the project tree of the PLC project are checked. This is primarily useful when creating libraries or when processing library projects. The scope of this command thus corresponds to the build command **Check all objects**.
Execution options for the command	Static analysis can be performed either explicitly using the command or implicitly. Implicit execution of the static analysis during each code generation can be enabled or disabled in the PLC project properties (Settings [▶ 13] tab). If the option **Perform static analysis automatically** is enabled, TwinCAT performs the static analysis after each successful code generation (with the command **Build project**, for example).	The "Check all objects" variant cannot be executed implicitly. It can only be executed explicitly via the command.

5.1.1 Syntax in the message window

Syntax of rule violations in the message window

Each rule has a unique number (shown in parentheses after the rule in the rule configuration view). If a rule violation is detected during the static analysis, the number together with an error or warning description is issued in the message window, based on the following syntax. The abbreviation "SA" stands for "Static Analysis".

Syntax: **"SA<rule number>: <rule description>"**

Sample for rule number 33 (unused variables): "SA0033: Not used: variable 'bSample'"

Syntax of convention violations in the message window

Each naming convention has a unique number (shown in parentheses after the convention in the naming convention configuration view). If a violation of a convention or a preset is detected during the static analysis, the number is output in the error list together with an error description based on the following syntax. The abbreviation "NC" stands for "Naming Convention".

Syntax: **"NC<prefix convention number>: <convention description>"**

Sample for convention number 151 (DUTs of type Structure): "NC0151: Invalid type name 'STR_Sample'. Expected prefix 'ST_'"

Syntax of symbol violations in the message window

If a symbol is used in the code that is configured as a forbidden symbol, an error is issued in the message window after the static analysis has been performed.

Syntax: **"Forbidden symbol '<symbol>'“**

Sample for the symbol XOR: "Forbidden symbol 'XOR'“

5.2 Command 'Run static analysis [Check all objects]'

Symbol: §

Function: The command starts the static code analysis for all objects of the currently active PLC project and outputs the results in the message window.

Call: Build menu or context menu of the PLC project object

During execution of the static analysis, compliance with the coding rules, naming conventions and forbidden symbols is checked. This command can be used to trigger the static analysis manually (explicit execution).

TwinCAT issues the result of the static analysis, i.e. messages relating to rule violations, in the message window. The rules [▸ 15], naming conventions [▸ 80] and forbidden symbols [▸ 98] to be taken into account in the static analysis can be configured [▸ 13] in the PLC project properties. You can also define whether the violation of a coding rule should appear as an error or a warning in the message window (see: Rules [▸ 15]).

See also: Syntax in the message window [▸ 101]

Please note that the selected PLC project is created before this command is executed. Checking via the static analysis is only started if the code generation was successful, i.e. if the compiler did not detect any compilation errors.

Please also note the Command 'Run static analysis' [▸ 100] and the differences between the two commands described in the following table.

Differences	Command 'Run static analysis'	Command 'Run static analysis [Check all objects]'
Scope	The objects used in the PLC project are checked. Objects that are not used are not checked with this command. The scope of this command thus corresponds to the build commands **Build Project/Solution** or **Rebuild Project/Solution** respectively. If you also wish to have the unused objects checked by the static analysis, which is useful, for example, when processing library projects, you can use the command **Run static analysis [check all objects]**.	All objects in the project tree of the PLC project are checked. This is primarily useful when creating libraries or when processing library projects. The scope of this command thus corresponds to the build command **Check all objects**.
Execution options for the command	Static analysis can be performed either explicitly using the command or implicitly. Implicit execution of the static analysis during each code generation can be enabled or disabled in the PLC project properties (Settings [▸ 13] tab). If the option **Perform static analysis automatically** is enabled, TwinCAT performs the static analysis after each successful code generation (with the command **Build project**, for example).	The "Check all objects" variant cannot be executed implicitly. It can only be executed explicitly via the command.

5.3 Command 'View Standard Metrics'

Symbol: ▤

Function: The command starts the static metric code analysis for the currently active PLC project and represents the metrics for the programming blocks used in a table.

Call: Build menu or context menu of the PLC project object

The command starts the code generation for the selected PLC project (with the command **Build project**, for example). In a tabular view, **Standard Metrics**, TwinCAT then displays the desired metrics (parameters) for each programming block used. The metrics to be displayed are activated in the project properties (see Configuration of the metrics [▸ 93]).

If a value is outside the range defined by a lower and/or upper limit in the configuration, the table cell is shown in red.

The table can be sorted by columns by clicking on the respective column header.

Please note that the selected PLC project is created before this command is executed. Creation of the standard metrics is only started if the code generation was successful, i.e. if the compiler did not detect any compilation errors.

Please also note the Command 'View Standard Metrics [Check all objects]' [▸ 105] and the differences between the two commands are described in the following table.

Differences	**Command 'View Standard Metrics'**	**Command 'View Standard Metrics [Check all objects]'**
Scope	The standard metrics are created for the objects used in the PLC project. Objects that are not used are not considered with this command. The scope of this command thus corresponds to the build commands **Build Project/Solution** or **Rebuild Project/Solution** respectively. If you want to create default metrics for unused objects, which is useful when editing library projects, you can use the **command 'View Standard Metrics [Check all objects]'**.	The standard metrics are created for all objects located in the project tree of the PLC project. This is primarily useful when creating libraries or when processing library projects. The scope of this command thus corresponds to the build command **Check all objects**.

5.3.1 Commands in the context menu of the 'Standard Metrics' view

Right-click in the **Standard Metrics** view to open a context menu that offers several commands.

The context menu offers options for updating, printing or exporting the metrics table, or to copy to the clipboard. Via the context menu you can also navigate to a view for configuring the metrics – just like in the PLC project properties. In addition, you can generate a Kiviat diagram for the selected function blocks or open the block in the corresponding editor. A prerequisite for generating a Kiviat diagram is that at least three metrics are configured with a defined value range (lower and upper limit).

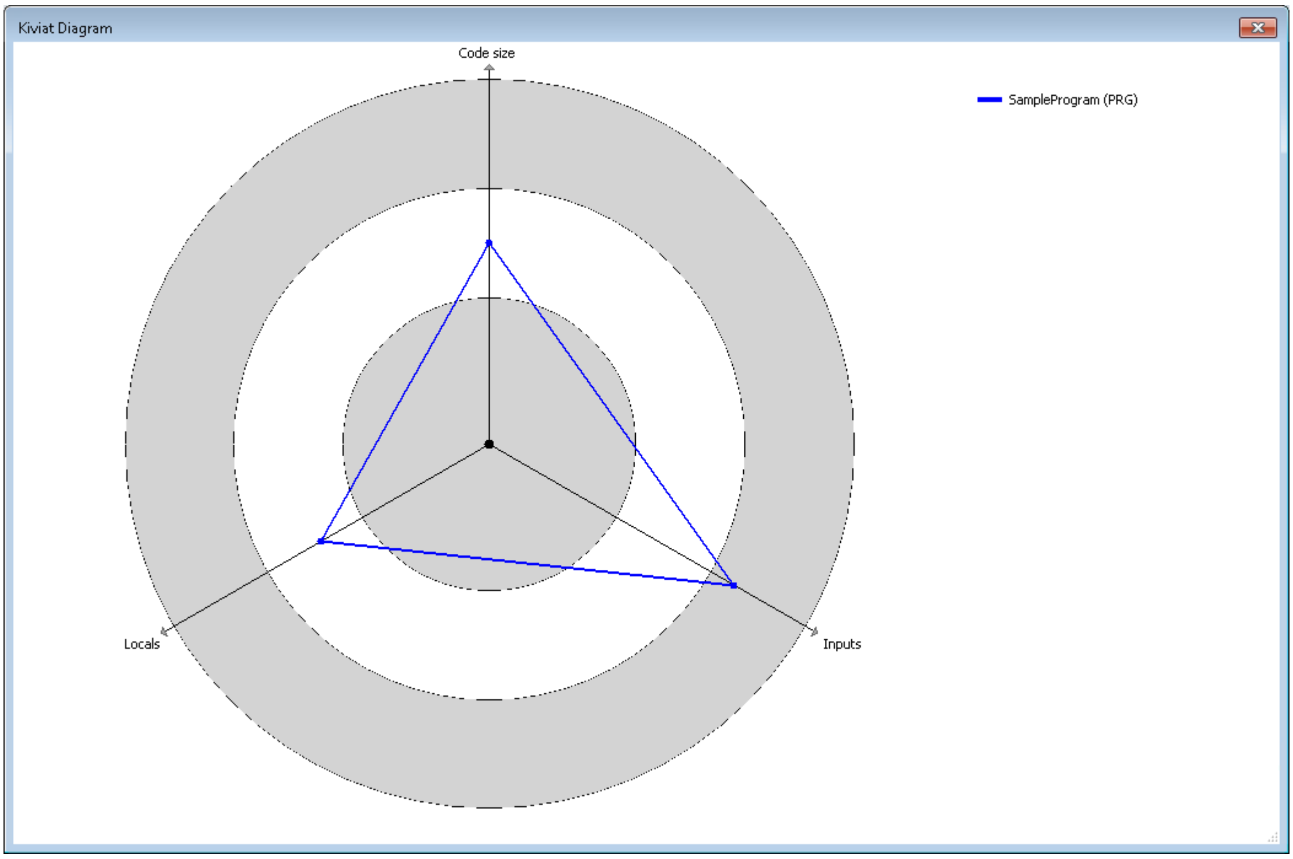
The following commands are available:

- **Calculate**: The values are updated.
- **Print table**: The standard dialog for setting up the print job appears.
- **Copy table**: The data are copied to the clipboard, separated by tabs. From there you can paste the table directly in a spreadsheet or a word processor.
- **Export table**: The data are exported into a text file (*.csv), separated by semicolons.
- **Kiviat diagram**: A radar chart is created for the selected function block. This is a graphical representation of the function blocks, for which the metrics define a lower and upper limit. It is used to visualize how well the code for the programming unit matches a particular standard.
  Each metric is shown as an axis in a circle, which starts in the center (value 0) and runs through three ring zones. The inner ring zone represents the range below the lower limit defined for the metric, the outer ring zone represents the range above the upper limit. The axes for the respective metrics are distributed evenly around the circle.
  The current values for the individual metrics on their axes are linked with lines. Ideally, the whole line should be within the central ring zone.

Prerequisite for using a Kiviat diagram

At least three metrics with a define value range must be configured.

The following diagram shows an example for 3 metrics with defined ranges (the name of the metric is shown at the end of each axis, the name of the function block at the top right):



- **Configure:** A table opens in which the metrics can be configured. The view, functionality and settings correspond to the metrics configuration [▸ 93] in the PLC project properties. If you make a change in this table, it is automatically applied to the PLC project properties.
- **Open POU**: The programming block opens in the corresponding editor.

5.4 Command 'View Standard Metrics [Check all objects]'

Symbol: ▤

Function: The command starts the static metric code analysis for the currently active PLC project and displays the metrics for all programming blocks in a table.

Call: Build menu or context menu of the PLC project object

The command starts the code generation for the selected PLC project (with the command **Build project**, for example). TwinCAT shows the selected metrics for each programming block in the tabular **Standard Metrics** view. The metrics to be displayed are activated in the project properties (see Configuration of the metrics [▸ 93]).

If a value is outside the range defined by a lower and/or upper limit in the configuration, the table cell is shown in red.

The table can be sorted by columns by clicking on the respective column header.

Please note that the selected PLC project is created before this command is executed. Creation of the standard metrics is only started if the code generation was successful, i.e. if the compiler did not detect any compilation errors.

Please also note the Command 'View Standard Metrics' [▸ 103] and the differences between the two commands, which are described in the following table.

Differences	Command 'View Standard Metrics'	Command 'View Standard Metrics [Check all objects]'
Scope	The standard metrics are created for the objects used in the PLC project. Objects that are not used are not considered with this command. The scope of this command thus corresponds to the build commands **Build Project/Solution** or **Rebuild Project/Solution** respectively. If you want to create default metrics for unused objects, which is useful when editing library projects, you can use the **command 'View Standard Metrics [Check all objects]'**.	The standard metrics are created for all objects located in the project tree of the PLC project. This is primarily useful when creating libraries or when processing library projects. The scope of this command thus corresponds to the build command **Check all objects**.

5.4.1 Commands in the context menu of the 'Standard Metrics' view

Right-click in the **Standard Metrics** view to open a context menu that offers several commands.

The context menu offers options for updating, printing or exporting the metrics table, or to copy to the clipboard. Via the context menu you can also navigate to a view for configuring the metrics – just like in the PLC project properties. In addition, you can generate a Kiviat diagram for the selected function blocks or open the block in the corresponding editor. A prerequisite for generating a Kiviat diagram is that at least three metrics are configured with a defined value range (lower and upper limit).

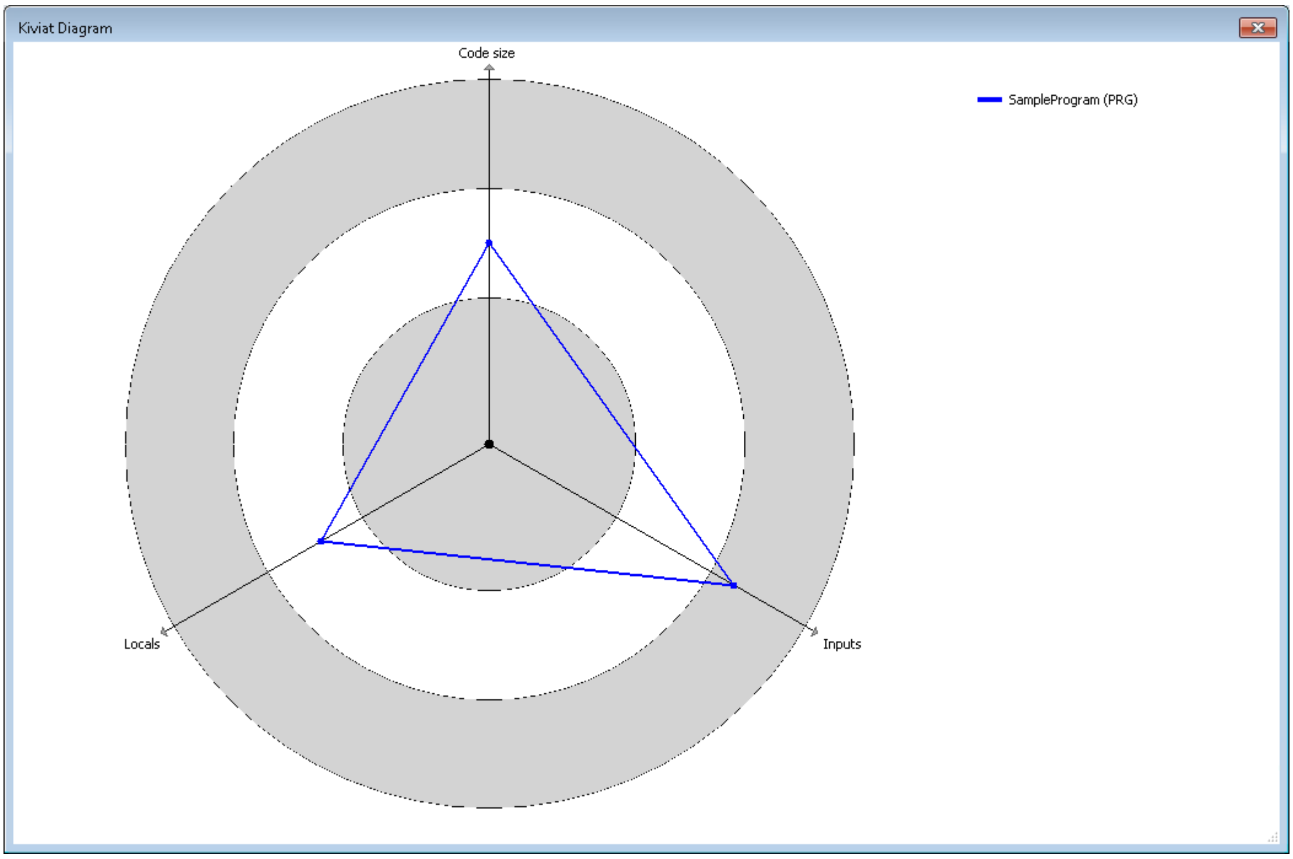
The following commands are available:

- **Calculate**: The values are updated.
- **Print table**: The standard dialog for setting up the print job appears.
- **Copy table**: The data are copied to the clipboard, separated by tabs. From there you can paste the table directly in a spreadsheet or a word processor.
- **Export table**: The data are exported into a text file (*.csv), separated by semicolons.
- **Kiviat diagram**: A radar chart is created for the selected function block. This is a graphical representation of the function blocks, for which the metrics define a lower and upper limit. It is used to visualize how well the code for the programming unit matches a particular standard.
  Each metric is shown as an axis in a circle, which starts in the center (value 0) and runs through three ring zones. The inner ring zone represents the range below the lower limit defined for the metric, the outer ring zone represents the range above the upper limit. The axes for the respective metrics are distributed evenly around the circle.
  The current values for the individual metrics on their axes are linked with lines. Ideally, the whole line should be within the central ring zone.

Prerequisite for using a Kiviat diagram

At least three metrics with a define value range must be configured.

The following diagram shows an example for 3 metrics with defined ranges (the name of the metric is shown at the end of each axis, the name of the function block at the top right):



- **Configure:** A table opens in which the metrics can be configured. The view, functionality and settings correspond to the metrics configuration [▸ 93] in the PLC project properties. If you make a change in this table, it is automatically applied to the PLC project properties.
- **Open POU**: The programming block opens in the corresponding editor.

6 Pragmas and attributes

A pragma and various attributes are available to temporarily disable individual rules or naming conventions for the static analysis, i.e. to exclude certain code lines or program units from the evaluation.

Requirement: The rules or conventions are enabled or defined in the PLC-project properties. See also:

- Rules [▸ 15]
- Naming conventions [▸ 80]

Attributes are inserted in the declaration part of a programming block in order to deactivate certain rules for a complete programming object.

Pragmas are used in the implementation part of a programming block in order to deactivate certain rules for individual code lines. The exception to this is rule SA0164, which can also be deactivated in the declaration part by a pragma.

Rules that are disabled in the project properties cannot be activated by a pragma or attribute.

Rule SA0004 cannot be disabled by a pragma or an attribute.

Pragmas in the implementation editor

If you want to use a pragma in the implementation editor, this is currently possible in the ST and FBD/LD/IL editors.

In FBD/LD/IL the desired pragma must be entered in a label.

The following section provides an overview and a detailed description of the available pragmas and attributes.

Overview

- Pragma {analysis ...} [▸ 109]
  - for disabling coding rules in the implementation part
  - can be used for individual code lines
- Attribute {attribute 'no-analysis'} [▸ 109]
  - for excluding programming objects (e.g. POU, GVL, DUT) from the static analysis (coding rules, naming conventions, forbidden symbols)
  - can only be used for whole programming objects
- Attribute {attribute 'analysis' := '...'} [▸ 110]
  - for disabling coding rules in the declaration part
  - can be used for individual declarations or for whole programming objects
- Attribute {attribute 'naming' := '...'} [▸ 110]
  - for disabling naming conventions in the declaration part
  - can be used for individual declarations or for whole programming objects
- Attribute {attribute 'nameprefix' := '...'} [▸ 111]
  - for defining prefixes for instances of a structured data type
  - can be used in the declaration part of a structured data type
- Attribute {attribute 'analysis:report-multiple-instance-calls'} [▸ 112]
  - for specifying that a function block instance should only be called once

- can be used in the declaration part of a function block

Detailed description

Pragma {analysis ...}

You can use the pragma {analysis -/+<rule number>} in the implementation part of a programming block in order to disregard individual coding rules for the following code lines. Coding rules are deactivated by specifying the rule numbers preceded by a minus sign ("-"). For activation they are preceded by a plus sign ("+"). You can specify any number of rules in the pragma with the help of comma separation.

Insertion location:

- Deactivation of rules: In the implementation part of the first code line from which the code analysis is disabled with {analysis - ...}.
- Activation of rules: After the last line of the deactivation with {analysis + ...}.
- For rule SA0164, the pragma can also be inserted in the declaration part before a comment.

Syntax:

- Deactivation of rules:
  - one rule: {analysis -<rule number>}
  - several rules: {analysis -<rule number>, -<further rule number>, -<further rule number>}
- Activation of rules:
  - one rule: {analysis +<rule number>}
  - several rules: {analysis +<rule number>, +<further rule number>, +<further rule number>}

Samples:

Rule 24 (only typed literals permitted) is to be disabled for one line (i.e. in these lines it is not necessary to write "nTest := DINT#99") and then enabled again:

```
{analysis -24}
nTest := 99;
{analysis +24}
nVar := INT#2;
```

Specification of several rules:

```
{analysis -10, -24, -18}
```

Attribute {attribute 'no-analysis'}

You can use the {attribute 'no-analysis'} attribute to exclude an entire programming object from the static analysis check. For this programming object no checks are carried out for the coding rules, naming conventions and forbidden symbols.

Insertion location:

above the declaration of a programming object

Syntax:

{attribute 'no-analysis'}

Samples:

```
{attribute 'qualified_only'}
{attribute 'no-analysis'}
VAR_GLOBAL
    …
END_VAR
```

```
{attribute 'no-analysis'}
PROGRAM MAIN
VAR
    …
END_VAR
```

Attribute {attribute 'analysis' := '...'}

You can use the attribute {attribute 'analysis' := '-<rule number>'} to switch off certain rules for individual declarations or for a complete programming object. The code rule is deactivated by specifying the rule number(s) with a minus sign in front. You can specify any number of rules in the attribute.

Insertion location:

above the declaration of a programming object or in the line above a variable declaration

Syntax:

- one rule: {attribute 'analysis' := '-<rule number>'}
- several rules: {attribute 'analysis' := '-<rule number>, -<further rule number>, -<further rule number>'}

Samples:

Rule 33 (unused variables) is to be disabled for all variables of the structure.

```
{attribute 'analysis' := '-33'}
TYPE ST_Sample :
STRUCT
    bMember   : BOOL;
    nMember   : INT;
END_STRUCT
END_TYPE
```

Checking of rules 28 (overlapping memory areas) and 33 (unused variables) is to be disabled for variable nVar1.

```
PROGRAM MAIN
VAR
    {attribute 'analysis' := '-28, -33'}
    nVar1 AT%QB21  : INT;
    nVar2 AT%QD5   : DWORD;

    nVar3 AT%QB41  : INT;
    nVar4 AT%QD10  : DWORD;
END_VAR
```

Rule 6 (concurrent access) is to be disabled for a global variable, so that no error message is generated if write access to the variable occurs from more than one task.

```
VAR_GLOBAL
    {attribute 'analysis' := '-6'}
    nVar  : INT;
    bVar  : BOOL;
END_VAR
```

Attribute {attribute 'naming' := '...'}

The attribute {attribute 'naming' := '...'} can be used in the declaration part in order to exclude individual declaration lines from the check for compliance with the current naming conventions.

Insertion location:

- Deactivation: in the declaration part above the relevant lines
- Activation: after the last line of the deactivation

Syntax:

{attribute 'naming' := '<off|on|omit>'}

- off, on: the check is disabled for all rows between the "off" and "on" statements
- omit: only the next row is excluded from the check

Sample:

It is assumed that the following naming conventions are defined:

- The identifiers of INT variables must have a prefix "n" (naming convention NC0014), e.g. "nVar1".
- Function block names must start with "FB_" (naming convention NC0103), e.g. "FB_Sample".

For the code shown below, the static analysis then only issues messages for the following variables: cVar, aVariable, bVariable.

```
PROGRAM MAIN
VAR
  {attribute 'naming' := 'off'}
  aVar  : INT;
  bVar  : INT;
  {attribute 'naming' := 'on'}

  cVar  : INT;

  {attribute 'naming' := 'omit'}
  dVar  : INT;

  fb1   : SampleFB;
  fb2   : FB;
END_VAR
```

```
{attribute 'naming' := 'omit'}
FUNCTION_BLOCK SampleFB
…
```

```
{attribute 'naming' := 'off'}
FUNCTION_BLOCK FB
VAR
    {attribute 'naming' := 'on'}
    aVariable : INT;
    bVariable : INT;
    …
```

Attribute {attribute 'nameprefix' := '...'}

The attribute {attribute 'nameprefix' := '...'} defines a prefix for variables of a structured data type. A naming convention then applies to the effect that identifiers for instances of this type must have this prefix.

Insertion location:

above the declaration of a structured data type

Syntax:

{attribute ‘nameprefix’ := '<prefix>'}

Example:

The following naming conventions are defined in the category Naming conventions [▶ 80] in the PLC project properties:

- Variables of the type of a structure (NC0032): st
- Structures (NC0151): ST_

Conversely, variables of the type "ST_Point" should not begin with the prefix "st", but with the prefix "pt".

In the following sample, the statistic analysis will output a message for "a1" and "st1" of the type "ST_Point" because the variable names do not begin with "pt". For variables of the type "ST_Test", conversely, the prefix "st" is expected.

```
TYPE ST_Test :
STRUCT
    …
END_STRUCT
END_TYPE
```

```
{attribute 'nameprefix' := 'pt'}
TYPE ST_Point :
STRUCT
    x  : INT;
    y  : INT;
END_STRUCT
END_TYPE
```

```
PROGRAM MAIN
VAR
    a1   : ST_Point;     // => Invalid variable name 'a1'. Expect prefix 'pt'
    st1  : ST_Point;     // => Invalid variable name 'st1'. Expect prefix 'pt'
    pt1  : ST_Point;

    a2   : ST_Test;      // => Invalid variable name 'a2'. Expect prefix 'st'
    st2  : ST_Test;
    pt2  : ST_Test;      // => Invalid variable name 'st2'. Expect prefix 'st'
END_VAR
```

Attribute {attribute 'analysis:report-multiple-instance-calls'}

The attribute {attribute 'analysis:report-multiple-instance-calls'} identifies a function block for a check for rule 105: Only function blocks with this attribute are checked to ascertain whether the instances of the function block are called several times. The attribute has no effect if rule 105 is disabled in the Rules [▶ 15] category in the PLC project properties.

Insertion location:

above the declaration of a function block

Syntax:

{attribute 'analysis:report-multiple-instance-calls'}

Sample:

In the following sample the static analysis will issue an error for fb2, since the instance is called more than once.

Function block FB_Test1 without attribute:

```
FUNCTION_BLOCK FB_Test1
…
```

Function block FB_Test2 with attribute:

```
{attribute 'analysis:report-multiple-instance-calls'}
FUNCTION_BLOCK FB_Test2
…
```

Program MAIN:

```
PROGRAM MAIN
VAR
    fb1  : FB_Test1;
    fb2  : FB_Test2;
END_VAR
```

```
fb1();
fb1();
fb2();          // => SA0105: Instance 'fb2' called more than once
fb2();          // => SA0105: Instance 'fb2' called more than once
```

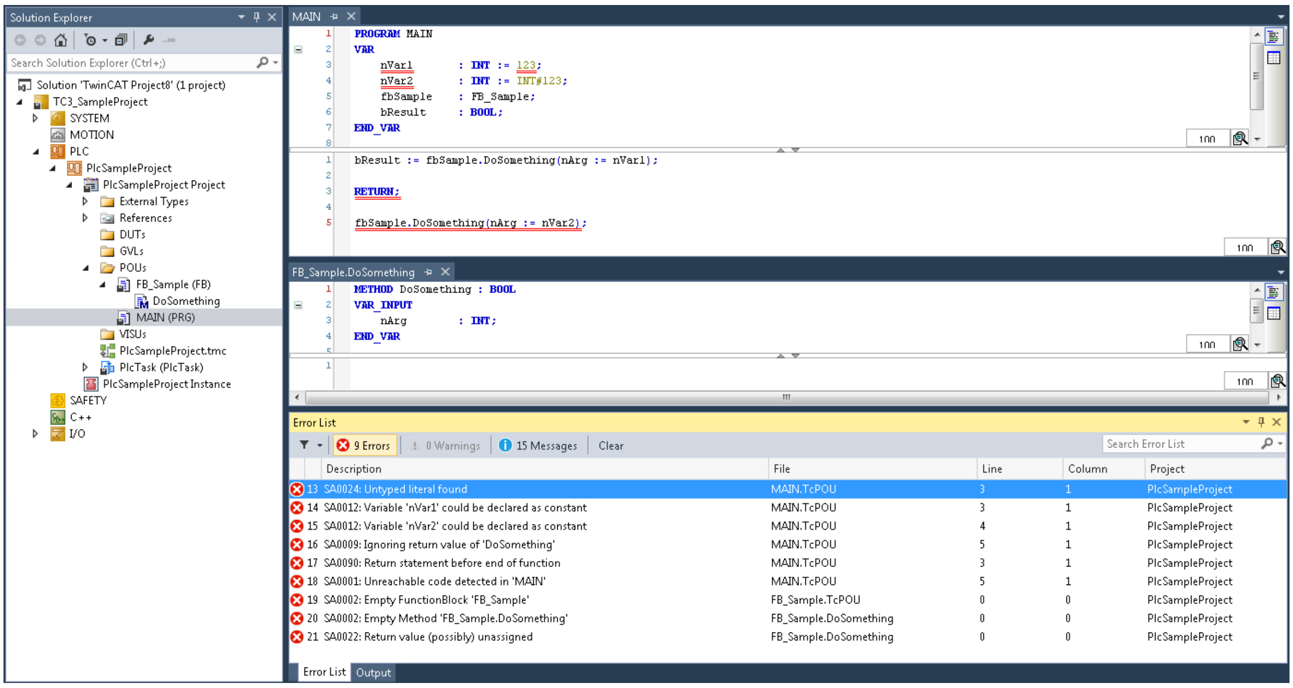
7 Examples

7.1 Static analysis

During execution of the static analysis [▶ 100], compliance with the coding rules [▶ 15], naming conventions [▶ 80] and forbidden symbols [▶ 98] is checked. The following section provides a sample for each of these aspects.

1) Coding rules

In this sample some coding rules are configured as error. The violations of this coding rules are therefore reported as an error after the static analysis has been performed. Further information is shown in the following diagram.



```
1  PROGRAM MAIN
2  VAR
3      nVar1       : INT := 123;
4      nVar2       : INT := INT#123;
5      fbSample    : FB_Sample;
6      bResult     : BOOL;
7  END_VAR
8
```

```
1  bResult := fbSample.DoSomething(nArg := nVar1);
2
3  RETURN;
4
5  fbSample.DoSomething(nArg := nVar2);
```

```
1  METHOD DoSomething : BOOL
2  VAR_INPUT
3      nArg        : INT;
4  END_VAR
5
```

```
1
```

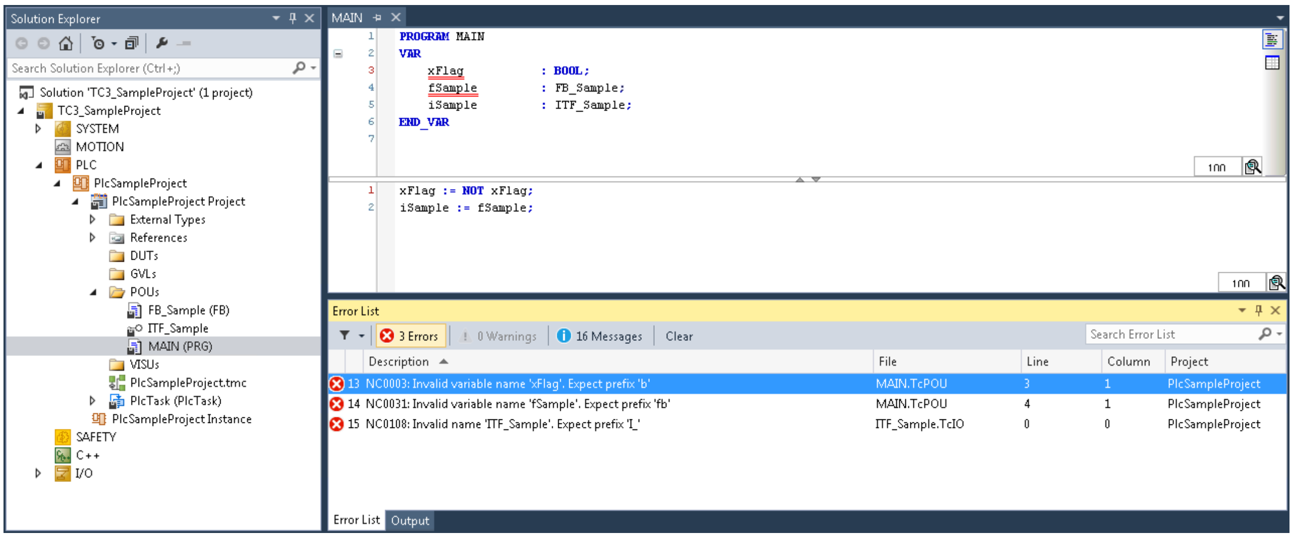
	Description	File	Line	Column	Project
13	SA0024: Untyped literal found	MAIN.TcPOU	3	1	PlcSampleProject
14	SA0012: Variable 'nVar1' could be declared as constant	MAIN.TcPOU	3	1	PlcSampleProject
15	SA0012: Variable 'nVar2' could be declared as constant	MAIN.TcPOU	4	1	PlcSampleProject
16	SA0009: Ignoring return value of 'DoSomething'	MAIN.TcPOU	5	1	PlcSampleProject
17	SA0090: Return statement before end of function	MAIN.TcPOU	3	1	PlcSampleProject
18	SA0001: Unreachable code detected in 'MAIN'	MAIN.TcPOU	5	1	PlcSampleProject
19	SA0002: Empty FunctionBlock 'FB_Sample'	FB_Sample.TcPOU	0	0	PlcSampleProject
20	SA0002: Empty Method 'FB_Sample.DoSomething'	FB_Sample.DoSomething	0	0	PlcSampleProject
21	SA0022: Return value (possibly) unassigned	FB_Sample.DoSomething	0	0	PlcSampleProject

2) Naming conventions

The following naming conventions are configured:

- Prefix "b" for variables of type BOOL (NC0003)
- Prefix "fb" for function block instances (NC0031)
- Prefix "FB_" for function blocks (NC0103)
- Prefix "I_" for interfaces (NC0108)

This naming conventions are not adhered to in the declaration of Boolean variables ("x"), the instantiation of function block ("f") and the declaration of the interface type ("ITF_"). These code positions are reported as an error after the static analysis has been performed.



```
PROGRAM MAIN
VAR
    xFlag        : BOOL;
    fSample      : FB_Sample;
    iSample      : ITF_Sample;
END_VAR
```

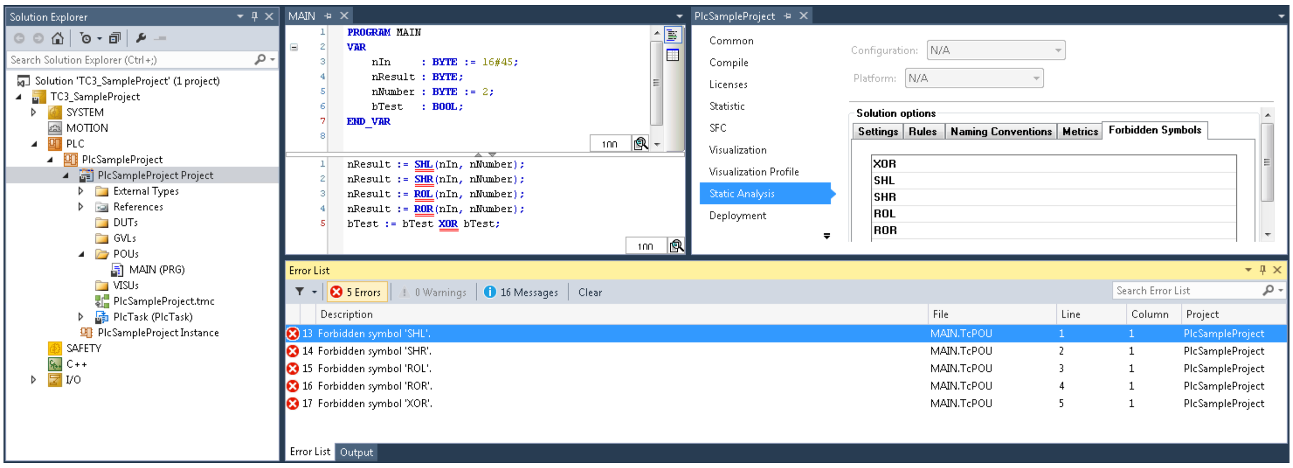
```
xFlag := NOT xFlag;
iSample := fSample;
```

Error List: 3 Errors | 0 Warnings | 16 Messages | Clear

	Description	File	Line	Column	Project
13	NC0003: Invalid variable name 'xFlag'. Expect prefix 'b'	MAIN.TcPOU	3	1	PlcSampleProject
14	NC0031: Invalid variable name 'fSample'. Expect prefix 'fb'	MAIN.TcPOU	4	1	PlcSampleProject
15	NC0108: Invalid name 'ITF_Sample'. Expect prefix 'I_'	ITF_Sample.TcIO	0	0	PlcSampleProject

3) Forbidden symbols

The bit string operator XOR and the bit shift-operators SHL, SHR, ROL and ROR are configured as forbidden symbols. These operators should not be used in the code.
Accordingly, any use of these operators is reported as an error after the static analysis has been performed.



```
PROGRAM MAIN
VAR
    nIn     : BYTE := 16#45;
    nResult : BYTE;
    nNumber : BYTE := 2;
    bTest   : BOOL;
END_VAR
```

```
nResult := SHL(nIn, nNumber);
nResult := SHR(nIn, nNumber);
nResult := ROL(nIn, nNumber);
nResult := ROR(nIn, nNumber);
bTest := bTest XOR bTest;
```

Forbidden Symbols:

XOR
SHL
SHR
ROL
ROR

Error List: 5 Errors | 0 Warnings | 16 Messages | Clear

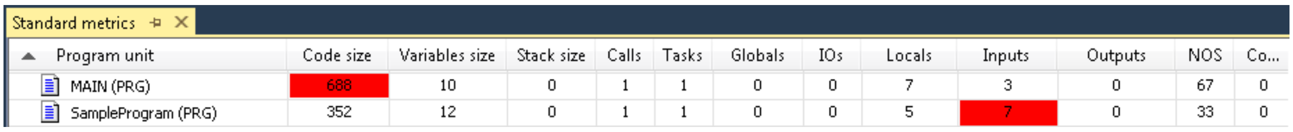
	Description	File	Line	Column	Project
13	Forbidden symbol 'SHL'.	MAIN.TcPOU	1	1	PlcSampleProject
14	Forbidden symbol 'SHR'.	MAIN.TcPOU	2	1	PlcSampleProject
15	Forbidden symbol 'ROL'.	MAIN.TcPOU	3	1	PlcSampleProject
16	Forbidden symbol 'ROR'.	MAIN.TcPOU	4	1	PlcSampleProject
17	Forbidden symbol 'XOR'.	MAIN.TcPOU	5	1	PlcSampleProject

7.2 Standard metrics

A sample for dealing with the standard metrics is provided below.

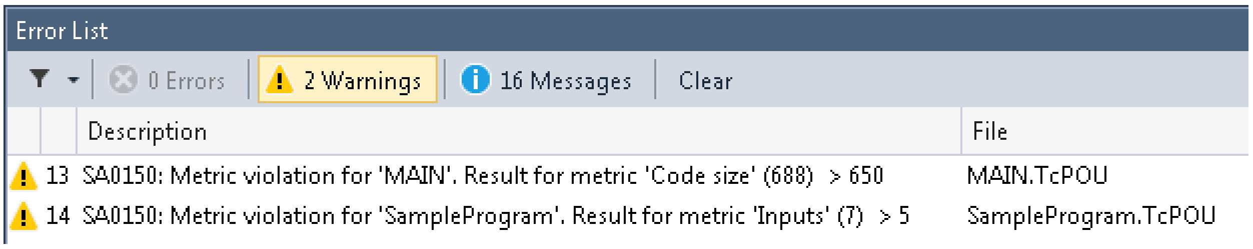
In this sample "650" (= 650 bytes) is defined as upper limit for the metric "code size" and "5" as upper limit for the metric "number of input variables" (see: Configuration of the metrics [▶ 93]). In addition, rule 150 (SA0150: Violation of lower or upper metrics limits) is enabled and configured as warning.

When the command 'View Standard Metrics' [▶ 103] is issued, the metric view opens and the indicators that were determined are displayed in tabular form. Since the size of the MAIN program is 688 bytes and the program SampleProgram has 7 input variables, these indicators exceed the defined upper limit in each case, so that the corresponding table cells are shown in red.


Standard metrics

Program unit	Code size	Variables size	Stack size	Calls	Tasks	Globals	IOs	Locals	Inputs	Outputs	NOS	Co...
MAIN (PRG)	688	10	0	1	1	0	0	7	3	0	67	0
SampleProgram (PRG)	352	12	0	1	1	0	0	5	7	0	33	0

In this sample, the fact that the defined upper limits are exceeded is not only apparent in the metric view. Since rule 150 is configured as warning, the static analysis checks for violations of lower and upper metric limits. After the static analysis [▶ 100] has been performed, the violation of the two upper limits is therefore reported as a warning in the message window.



		Description	File
⚠	13	SA0150: Metric violation for 'MAIN'. Result for metric 'Code size' (688) > 650	MAIN.TcPOU
⚠	14	SA0150: Metric violation for 'SampleProgram'. Result for metric 'Inputs' (7) > 5	SampleProgram.TcPOU

8 Automation Interface support

The Static Analysis can partly be operated via the Automation Interface (AI). AI support includes the following commands/actions:

- Explicit execution of Static Analysis via the Automation Interface [▸ 116]
- Implicit execution of Static Analysis via the Automation Interface [▸ 116]
- Save settings/configuration via Automation Interface [▸ 116]
- Load settings/configuration via Automation Interface [▸ 117]
- Export metrics [▸ 117]

Please also refer to the Automation Interface documentation:
Product description

Explicit execution of Static Analysis via the Automation Interface

The two following commands can be called explicitly via the Automation Interface:

- Command 'Run static analysis' [▸ 100]
- Command 'Run static analysis [Check all objects]' [▸ 102]

bCheckAll can be specified as optional parameter for the method `RunStaticAnalysis()`. However, the method can also be called without parameters.

Parameter	Call
RunStaticAnalysis()	Execution of the **Run static analysis** command
RunStaticAnalysis(bCheckAll = FALSE)	
RunStaticAnalysis(bCheckAll = TRUE)	Execution of the **Run static analysis [Check all objects]** command

PowerShell sample:

```
$p = $sysMan.LookupTreeItem("TIPC^MyPlcProject^MyPlcProject Project")
$p.RunStaticAnalysis()
```

C# sample:

```
ITcPlcIECProject4 plcIec4 = sysMan.LookupTreeItem("TIPC^Untitled1^Untitled1
Project") as ITcPlcIECProject4;
plcIec4.RunStaticAnalysis();
```

Implicit execution of Static Analysis via the Automation Interface

Alternatively, the setting [▸ 13] **Perform static analysis automatically** can be enabled, and the project can be created via the Automation Interface, so that the Static Analysis is implicitly performed during the project creation process.

Save settings/configuration via Automation Interface

Available from TwinCAT 3.1 Build 4026

The settings [▸ 13] from Static Analysis can be saved or exported to a *.csa file via Automation Interface.

For the method `SaveStaticAnalysisSettings(string bstrFilename)` the destination path of the file must be specified as a transfer parameter.

Note: The method `RunStaticAnalysis` is available from the interface `ITcPlcIECProject3`. The methods `SaveStaticAnalysisSettings` and `LoadStaticAnalysisSettings` are offered from the interface `ITcPlcIECProject4`.

C# sample:

```
// Path to the location to export the SAN configuration
string saveCsaPath = @"C:\Users\UserName\Desktop\SaveTest.csa";
[…]
// Navigate to PLC project
ITcPlcIECProject4 plcIec4 = sysMan.LookupTreeItem("TIPC^Untitled1^Untitled1
Project") as ITcPlcIECProject4;
// Save SAN configuration
plcIec4.SaveStaticAnalysisSettings(saveCsaPath);
```

Load settings/configuration via Automation Interface

Available from TwinCAT 3.1 Build 4026

A ready-made Static Analysis configuration (*.csa file) can be loaded into the PLC project via Automation Interface. The settings [▶ 13] loaded by this can then be checked by AI by running the Static Analysis (see above).

For the method `LoadStaticAnalysisSettings(string bstrFilename)` the path of the file to be loaded must be specified as a transfer parameter.

Note: The method `RunStaticAnalysis` is available from the interface `ITcPlcIECProject3`. The methods `SaveStaticAnalysisSettings` and `LoadStaticAnalysisSettings` are offered from the interface `ITcPlcIECProject4`.

C# sample:

```
// Path to load a SAN configuration
string loadCsaPath = @"C:\Users\UserName\Desktop\LoadTest.csa";
[…]
// Navigate to PLC project
ITcPlcIECProject4 plcIec4 = sysMan.LookupTreeItem("TIPC^Untitled1^Untitled1
Project") as ITcPlcIECProject4;
// Load SAN configuration
plcIec4.LoadStaticAnalysisSettings(loadCsaPath);
// Optionally run SAN afterwards
plcIec4.RunStaticAnalysis();
```

Export metrics

Available from TwinCAT 3.1 Build 4026.4

The standard metrics can be exported to a text file (*.csv) via the Automation Interface. A current calculation of the metrics is performed implicitly. If this process was executed manually, it would include the following two commands:

- Command 'View Standard Metrics' [▶ 103]
- **Export table** command, see Commands in the context menu of the 'Standard Metrics' view [▶ 104]

For the method `ExportStandardMetrics(string bstrFilename)`, the path that the export file is saved on must be specified as a parameter value.

The method `ExportStandardMetrics` is available from the interface `ITcPlcIECProject5`.

C# sample:

```
// Path to save the csv file
string savePath = @"C:\Users\UserName\Desktop\Metrics.csv";
[…]
// Navigate to PLC project
ITcPlcIECProject5 plcIec5 = sysMan.LookupTreeItem("TIPC^Untitled1^Untitled1
Project") as ITcPlcIECProject5;
// Export standard metrics
plcIec5.ExportStandardMetrics(savePath);
```

9 QuickFix/Precompile

Available from TwinCAT 3.1 Build 4026

Some rules from Static Analysis can already be checked during precompilation. For the detection of such rule violations no explicit execution of the Static Analysis is necessary, but the check already takes place on the basis of the precompile information during editing. The checking of a rule during precompilation takes place only if the rule is enabled in the Static Analysis settings.

Precompile: Wavy underline and display in the message window

When a rule violation occurs, it is immediately indicated by wavy underline in the declaration editor or in the ST editor. Additionally - as long as the editor is open - error messages or warnings appear in the message window in the category "IntelliSense". These contain the note "(precompile)" following the rule number.

QuickFix commands

In addition, for some rules that can be checked during precompilation, there is the possibility of a QuickFix in the declaration editor and the ST editor. You can perform automatic, immediate error handling directly at the affected code locations. For quick error handling, click on the wavy underlined code in the editor and then click on the light bulb icon.

Depending on the error, the following QuickFix commands are offered:

- Ignore error message/warning:
  The command causes pragmas or attributes to be automatically inserted into the code that exclude checking the associated rule for that line of code.
- Ignore error message/warning globally for the POU:
  The command causes an attribute to be automatically inserted at the beginning of the declaration part of the programming object. Then a check of the associated rule for this programming object is excluded.
- Disable checking:
  The command causes the checking of the associated rule to be disabled in the settings.
- Fix error by suggesting to change ST code:
  Example for "SA0033: Unused variables": The declaration of the unused variables is removed from the declaration editor.

Available rules

Please note that not all rules can be checked during precompilation. Based on the precompile information, the following rules are checked:
SA0001, SA0002, SA0011, SA0020, SA0022, SA0033, SA0034, SA0054, SA0090, SA0113, SA0114, SA0115, SA0117, SA0164, SA0168, SA0169, SA0170, SA0171

More Information:
www.beckhoff.com/te1200

Beckhoff Automation GmbH & Co. KG
Hülshorstweg 20
33415 Verl
Germany
Phone: +49 5246 9630
info@beckhoff.com
www.beckhoff.com