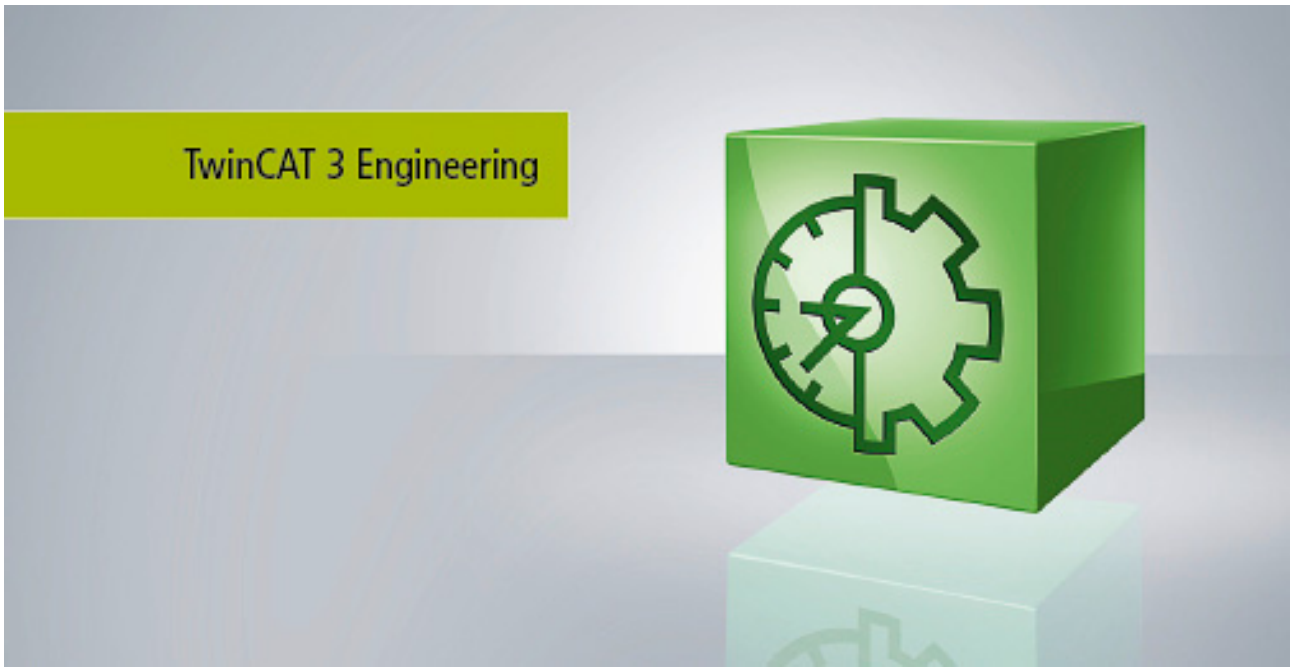


Manual

TC3 PLC Static Analysis

TwinCAT 3

Version: 1.9
Date: 2019-05-14
Order No.: TE1200

BECKHOFF

Table of contents

1 Foreword

1.1 Notes on the documentation

This description is only intended for the use of trained specialists in control and automation engineering who are familiar with the applicable national standards.
It is essential that the documentation and the following notes and explanations are followed when installing and commissioning the components.
It is the duty of the technical personnel to use the documentation published at the respective time of each installation and commissioning.

The responsible staff must ensure that the application or use of the products described satisfy all the 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 prior announcement.
No claims for the modification of products that have already been supplied may be made on the basis of the data, diagrams and descriptions in this documentation.

Trademarks

Beckhoff®, TwinCAT®, EtherCAT®, Safety over EtherCAT®, TwinSAFE®, XFC® and XTS® are registered trademarks of and licensed by Beckhoff Automation GmbH.
Other designations used in this publication may be trademarks whose use by third parties for their own purposes could violate the rights of the owners.

Patent Pending

The EtherCAT Technology is covered, including but not limited to the following patent applications and patents:
EP1590927, EP1789857, DE102004044764, DE102007017835
with corresponding applications or registrations in various other countries.

The TwinCAT Technology is covered, including but not limited to the following patent applications and patents:
EP0851348, US6167425 with corresponding applications or registrations in various other countries.

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

Copyright

1.2 Safety instructions

Safety regulations

Please note the following safety instructions and explanations!
Product-specific safety instructions can be found on following pages or in the areas mounting, wiring, commissioning etc.

Exclusion of liability

All the components are supplied in particular hardware and software configurations 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 engineering who are familiar with the applicable national standards.

Description of symbols

In this documentation the following symbols are used with an accompanying safety instruction or note. The safety instructions must be read carefully and followed without fail!

⚠ DANGER

Serious risk of injury!

Failure to follow the safety instructions associated with this symbol directly endangers the life and health of persons.

⚠ WARNING

Risk of injury!

Failure to follow the safety instructions associated with this symbol endangers the life and health of persons.

⚠ CAUTION

Personal injuries!

Failure to follow the safety instructions associated with this symbol can lead to injuries to persons.

NOTE

Damage to the environment or devices

Failure to follow the instructions associated with this symbol can lead to damage to the environment or equipment.

Tip or pointer

This symbol indicates information that contributes to better understanding.

2 Overview

"Static code analysis" is a programming tool that is integrated in TwinCAT 3 PLC. It checks the source code of a project for deviations from certain coding rules and naming conventions, before the project can be loaded onto the target system without compilation errors. To this end a set of rules and naming conventions, as well as a list of forbidden symbols, can be configured in the project properties, which are taken into account during the evaluation. The concept follows the basic idea of the "Lint" analysis tool. The static analysis can be triggered manually or performed automatically during the code generation. TwinCAT issues the result of the analysis, i.e. messages relating to rule violations, in the message window. When you configure the rules, you can define whether a violation appears as an error or a warning.

You can also configure various metrics to apply to your source 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.

Advantages

Static Analysis should be regarded as a supplement to the compiler. 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 applications implemented on the basis of the same naming conventions a uniform look and feel, which greatly improves the readability of the programs. In addition, the metrics provide an indication of the software quality.

Static Analysis thus helps avoid errors during programming and facilitates generating code that is easier to read.

Functionalities

An overview of the functionalities of "TC3 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, naming conventions and 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 [▶ 11], Naming conventions [▶ 57] and Forbidden symbols [▶ 75] 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 code lines). 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 [▶ 69] tab of the PLC project properties.

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

- Installation [▶ 9]
- Configuration of the settings, rules, naming conventions, metrics and forbidden symbols [▶ 10]
- Run static analysis [▶ 76]
- Run static analysis [check all objects] [▶ 77]
- View Standard Metrics [▶ 77]
- Pragmas and attributes [▶ 81]
- Examples [▶ 85]

Libraries

TwinCAT only analyzes the application code of the current 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 Run static analysis [check all objects] [▶ 77].

Punctual disablement of checks

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

3 Installation

The function "TE1200 | TC3 PLC Static Analysis" is installed together with the TwinCAT development environment (XAE setup) and has been included as release version since TwinCAT version 3.1 build 4022.0. All that is therefore required is licensing of the additional TE1200 engineering component. For further information please refer to the documentation on Licensing.

Please note that there is no 7-day trial license available for this product. Without an engineering license for TE1200 you can use the license-free version of Static Analysis (Static Analysis Light), which includes a few coding rules.

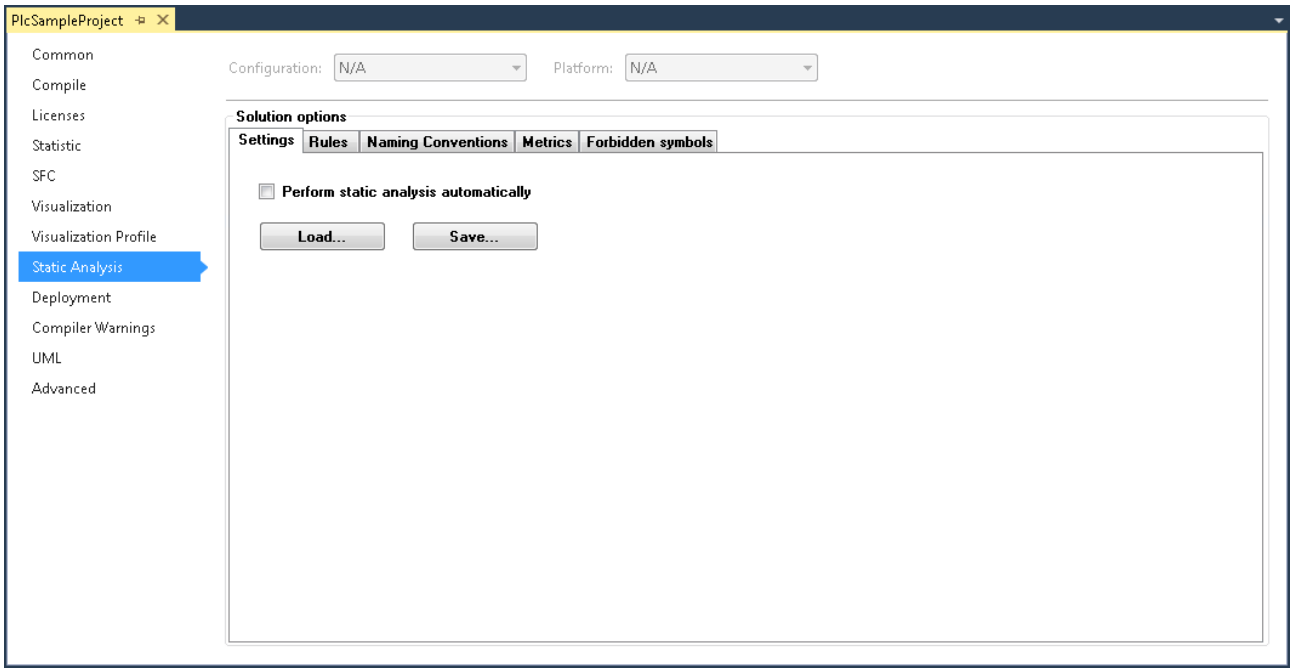
4 Configuration

After the installation [▶ 9] and licensing of "TE1200 | TC3 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.

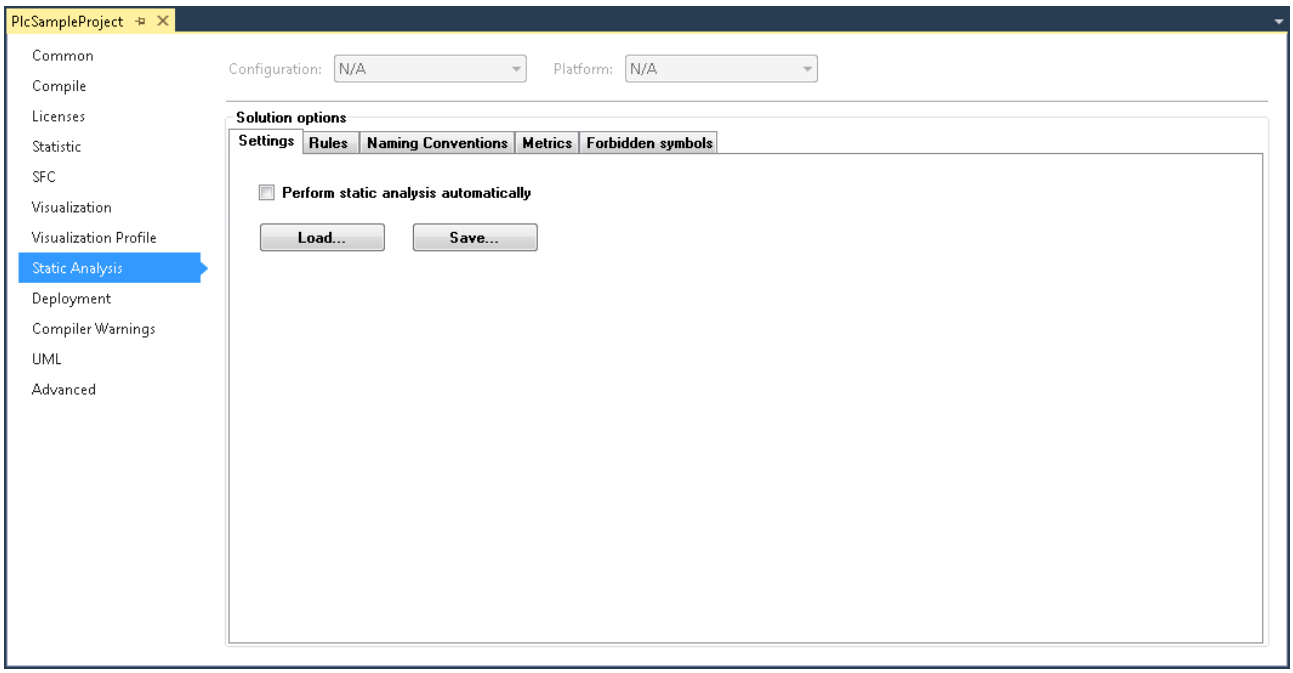
- Settings [▶ 10]
- Rules [▶ 11]
- Naming conventions [▶ 57]
- Metrics [▶ 69]
- Forbidden symbols [▶ 75]

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

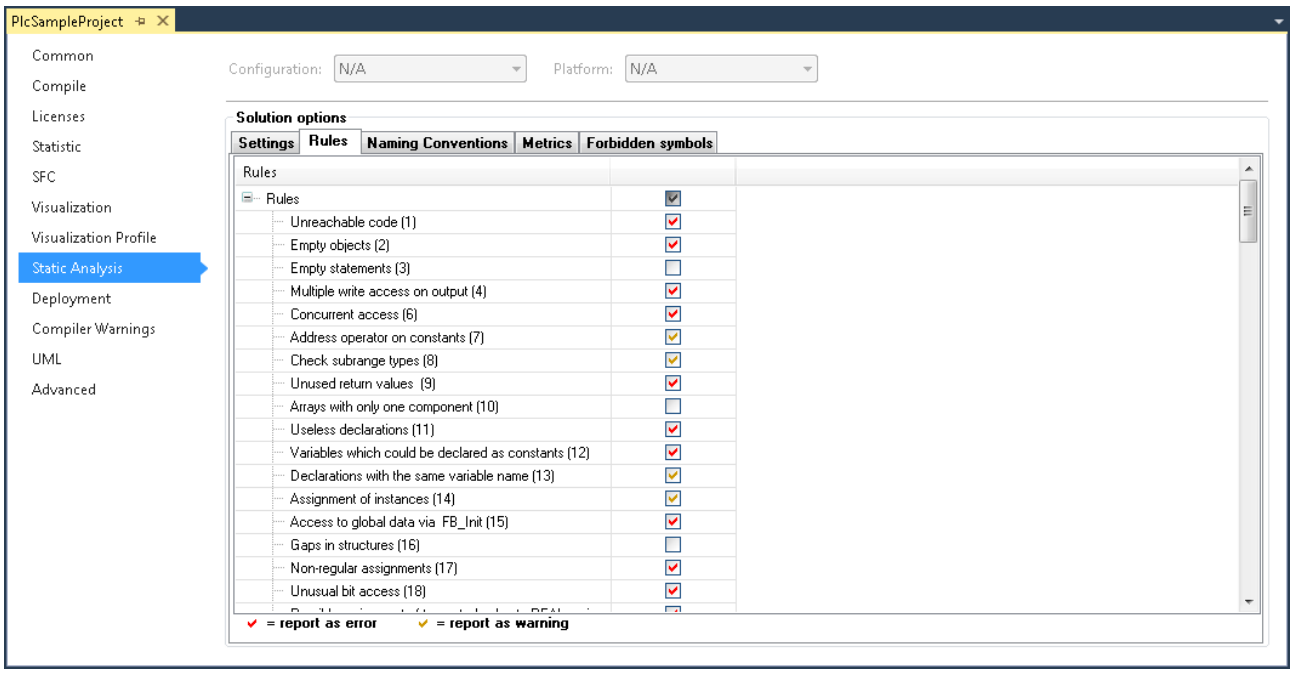
The **Settings** tab can be used to configure whether the static code analysis is automatically performed when the code is generated. The current configuration of the **Static Analysis** can be saved in an external file, or a configuration can be loaded from an external file.



- **Perform static analysis automatically**:
  If this option is enabled, TwinCAT performs the static analysis whenever code is generated (e.g. when the command **Build Project** is executed). The analysis can be started manually via the command Run static analysis [▶ 76], irrespective of the configuration of the automatic option.
- **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 below).
- **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 above). 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.

4.2 Rules

In the **Rules** tab you can configure the rules that are taken into account when the static analysis is performed [▶ 76]. The rules are displayed as a tree structure in the project properties. Some rules are arranged below organizational nodes.



Default settings:

All rules are enabled by default, with the exception of SA0016, SA0024, SA0073, SA0101, SA0105, SA0106, SA0107, SA0133, SA0134, SA0150, SA0162 to 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 following three settings are available, which can be accessed by repeated clicking on the checkbox:

-  : The rule is disabled.
-  : 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>"

Example 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 [▶ 81].

Overview and description of the rules:

An overview and a detailed description of the rules can be found under Rules - overview and description [▶ 13].

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: Concurrent access [▶ 18]
Determines variables with write access from more than one task.

SA0103: Concurrent access on not atomic data [▶ 43]
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 [▶ 17]
- SA0002: Empty objects [▶ 17]
- SA0003: Empty statements [▶ 17]
- SA0004: Multiple writes access on output [▶ 17]
- SA0006: Concurrent access [▶ 18]
- SA0007: Address operators on constants [▶ 18]
- SA0008: Check subrange types [▶ 18]
- SA0009: Unused return values [▶ 19]
- SA0010: Arrays with only one component [▶ 19]
- SA0011: Useless declarations [▶ 19]
- SA0012: Variables which could be declared as constants [▶ 20]
- SA0013: Declarations with the same variable name [▶ 20]
- SA0014: Assignments of instances [▶ 20]
- SA0015: Access to global data via FB_init [▶ 20]
- SA0016: Gaps in structures [▶ 21]
- SA0017: Non-regular assignments [▶ 21]
- SA0018: Unusual bit access [▶ 21]

- SA0167: Report temporary FunctionBlock instances [▶ 57]

Detailed description

SA0001: Unreachable code

Determines code that is not executed, for example due to a RETURN or CONTINUE statement.

Example 1 – RETURN:

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

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

Example 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

Determines POUs, GVLs or data type declarations that do not contain code.

SA0003: Empty statements

Determines rows containing a semicolon (;) but no statement.

Examples:

```
;                                // => SA0003
```

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

```
nVar;                            // => SA0003
```

SA0004: Multiple write access on output

Determines outputs that are written at more than one position.

Exception: No error is issued if an output variable is written in different branches of IF or CASE statements.

This rule **cannot** be switched off via pragma!
Further information on attributes can be found under Pragmas and attributes [▶ 81].
Please also note that this rule only checks allocated variables declared with a fixed address (e.g. X0.0). Allocated variables with a dynamic address (*) are not checked.

Example:

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: Concurrent access

Determines variables with write access from more than one task.

See also rule SA0103 [▶ 43].

Sample:

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

Determines locations at which the ADR operator is used for a constant.

Please note: If the option **Replace constants** is enabled in the compiler options of the PLC project properties, this is generally not allowed, and a compilation error is issued.

Example:

```
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

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.

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

Determines function, method and property calls for which the return value is not used.

Example:

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

Determines arrays containing only a single component.

Examples:

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

SA0011: Useless declarations

Determines structures, unions or enumerations with only a single component.

Example 1 – Structure:

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

Example 2 – Union:

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

Example 3 – Enumeration:

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

SA0012: Variables which could be declared as constants

Determines variables that are not subject to write access and therefore could be declared as constants.

Example:

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

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

SA0013: Declarations with the same variable name

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.

Examples:

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

Determines assignments to function block instances. For instances with pointer or reference variables such assignments may be risky.

Example:

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

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

SA0015: Access to global data via FB_init

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!

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

Determines gaps in structures or function blocks, caused by the alignment requirements of the currently selected target system. You can then fill the gaps.

Examples:

```
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

Determines assignments to pointers, which are not an address (ADR operator, pointer variables) or constant 0.

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

Determines bit access to signed variables. However, the IEC 61131-3 standard only permits bit access to bit fields. See also strict rule SA0148 [▶ 49].

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.

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

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

Example:

```
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

Determines assignments of addresses of temporary variables (variables on the stack) to non-temporary variables.

Example:

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

Determines all functions and methods containing an execution string without assignment to the return value.

Example:

```
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: Too big return values

Determines structured return values that occupy more than 4 bytes of memory and are therefore regarded as large. In addition, return values of type STRING are determined (irrespective of the memory space used).

Example:

Structure ST_Small:

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

Structure ST_Large:

```
TYPE ST_Large :
STRUCT
    n1  : LINT;
    b1  : BOOL;
END_STRUCT
END_TYPE
```

Function F_SmallReturnValue:

```
FUNCTION F_SmallReturnValue : ST_Small              // no error
```

Function F_LargeReturnValue:

```
FUNCTION F_LargeReturnValue : ST_Large              // => SA0023
```

SA0024: Untyped literals/constants

Determines untyped literals/constants (e.g. `nCount : INT := 10;`).

Example:

```
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

Determines enumeration constants that are not used with a qualified name, i.e. without preceding enumeration name.

Example:

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

Determines string assignments and initializations that do not use an adequate string length.

Examples:

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

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

SA0027: Multiple use of a name

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 the same as the name in another enumeration within the application or a referenced library.
- The name of a variable that is the same as the name of an object within the application or a referenced library.
- The name of a variable is the same as the name of an enumeration constant within the application or a referenced library.
- The name of an object is the same as the name of another object within the application or a referenced library.

Example:

The following example 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

Determines the locations at which 2 or more variables occupy the same memory space.

Example:

In the following example 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

Determines the code positions (in the implementation) at which the notation of an identifier differs from the notation in its declaration.

Examples:

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

Determines programs, function blocks, functions, data types, interfaces, methods, properties, actions etc., which are not called within the compiled program code.

SA0032: Unused enumeration constants

Determines enumeration constants that are not used in the compiled program code.

Example:

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

Determines variables that are declared but not used within the compiled program code.

SA0035: Unused input variables

Determines input variables that are not used by any function block instance.

Example:

Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR_INPUT
    nIn  : INT;
    bIn  : BOOL;                  // => SA0035
END_VAR
VAR_OUTPUT
    nOut : INT;                   // => SA0036
END_VAR
```

Program MAIN:

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

```
fbSample(nIn := 99);
```

SA0036: Unused output variables

Determines output variables that are not used by any function block instance.

Example:

Function block FB_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR_INPUT
    nIn  : INT;
    bIn  : BOOL;                  // => SA0035
```

```
END_VAR
VAR_OUTPUT
    nOut : INT;                 // => SA0036
END_VAR
```

Program MAIN:

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

```
fbSample(nIn := 99);
```

SA0034: Enumerations with incorrect assignment

Determines values that are assigned to an enumeration variable. Only defined enumeration constants may be assigned to an enumeration variable.

Example:

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

Determines input variables (VAR_INPUT) that are subject to write access within the POU.

Example:

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

Determines output variables (VAR_OUTPUT) that are subject to read access within the POU.

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

Determines code positions at which division by zero may occur.

Example:

```
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

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.

Example:

In the following example 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

Determines the usage of different access paths for the same variable.

Examples:

In the following example 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

Determines global variables that are only used in a single POU.

Example:

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

Global variables:

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

SubProgram:

```
nGlobal2 := 123;
```

Program MAIN:

```
SubProgram();

nGlobal1 := nGlobal2;
```

SA0044: Declarations with reference to interface

Determines declarations with REFERENCE TO <interface> and declarations of VAR_IN_OUT variables with the type of an interface (realized implicitly via REFERENCE TO).

Examples:

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

Determines implicitly generated pointer data type conversions.

Examples:

```
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

Determines implicitly performed conversions from smaller to larger data types.

Exception: BOOL ↔ BIT

Examples:

```
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

Determines implicitly performed conversions from larger to smaller data types.

Exception: BOOL ↔ BIT

Example:

```
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

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

Examples:

```
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

Determines explicitly performed conversions from a larger to a smaller data type.

Examples:

```
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: Implicit signed/unsigned conversions

Determines explicitly performed conversions from signed to unsigned data types or vice versa.

Examples:

```
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

Determines invalid address and data type specifications.

Valid size prefixes in addresses:

- X for BOOL
- B for 1-byte data types
- W for 2-byte data types
- D for 4-byte data types

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

Examples:

```
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

Determines direct address access operations in the implementation code.

Examples:

```
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

Determines AT declarations on direct addresses.

Examples:

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

SA0051: Comparison operations on BOOL variables

Determines comparison operations on variables of type BOOL.

Example:

```
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

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 [▶ 49].

Therefore, the following exception arises for this rule: Shift operation on bit array data types (byte, DWORD, LWORD, WORD) do not result in a SA0052 error.

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

Determines whether the data type width was exceeded in bitwise shift of operands.

Examples:

```
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

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

Examples:

```
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 comparisons of unsigned operands

Determines unnecessary comparisons with unsigned operands. An unsigned data type is never less than zero.

Examples:

```
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

Determines literals (constants) outside the valid operator range.

Examples:

```
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 places

Determines positions with possible loss of decimals.

Examples:

```
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

Determines operations on variables of type enumeration. Assignments are permitted.

Exception: 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.

Example 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.Green;                            // no error
eColor := E_Color.Green + 1;                        // => SA0058
nVar   := E_Color.Blue / 2;                         // => SA0058
nVar   := E_Color.Green + E_Color.Red;              // => SA0058
```

Example 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

Determines comparisons with literals that always have the result TRUE or FALSE and can already be evaluated during compilation.

Examples:

```
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

Determines operations in which an operand with value 0 results in an invalid or meaningless operation.

Examples:

```
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

Determines operations on variables of type POINTER TO, which are not = (equality), <> (inequality), + (addition) or ADR.

Examples:

```
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 or FALSE in expressions

Determines the use of the literal TRUE or FALSE in expressions (e.g. `b1 AND NOT TRUE`).

Examples:

```
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
```

SA0063: Possibly not 16-bit-compatible operations

Determines 16-bit operations with intermediate results. Background: 32-bit intermediate results may be truncated on 16-bit systems.

Example:

(nVar+10) may exceed 16 bits.

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

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

SA0064: Addition of pointer

Determines all pointer additions.

Examples:

```
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

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.

Examples:

```
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

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.

Example:

```
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

Determines write access operations to a counter variable within a FOR loop.

Example:

```
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

Determines the use of non-temporary variables in FOR loops.

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

Determines FOR statements in which the index variable is used for access to an array index and exceeds the array index range.

Examples:

```
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

Determines FOR statements in which the upper limit is not defined with a constant value.

Examples:

```
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

Determines CASE statements without ELSE branch.

Example:

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

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

CASE nVar OF                     // => SA0075
    INT#1:
        bVar := FALSE;

    INT#2:
        bVar := TRUE;
END_CASE
```

SA0076: Missing enumeration constant

Determines code positions where an enumeration variable is used as condition and not all enumeration values are treated as CASE branches.

Example:

In the following example 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

Determines code positions where the data type of a condition does not match that of the CASE branch.

Example:

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

Determines CASE statements without cases, i.e. with only a single ELSE statement.

Example:

```
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

Determines code positions where the RETURN statement is not the last statement in a function, method, property or program.

Example:

```
FUNCTION F_TestFunction : BOOL
```

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

SA0095: Assignments in conditions

Determines assignments in conditions of IF, CASE, WHILE or REPEAT constructs.

Examples:

```
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

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)

Example:

In the following example 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

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

Examples:

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

Determines external access to local variables of programs or function blocks.

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.

Examples:

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

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.

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.

See also rule SA0006 [▶ 18].

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

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 Pragmas and attributes [▶ 84] must be added in the declaration part of the function block.

Example:

In the following example 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

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. Background: In such cases it may happen that the variables that are used in overwritten methods of the basic FB are not yet initialized.

Example:

- 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

Determines where formal parameters are missing.

Example:

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

Determines variables of type POINTER TO.

Example:

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

SA0112: Reference variables

Determines variables of type REFERENCE TO.

Example:

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

SA0113: Variables with data type WSTRING

Determines variables of type WSTRING.

Example:

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

SA0114: Variables with data type LTIME

Determines variables of type LTIME.

Example:

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

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

SA0115: Variables with data type UNION

Determines declarations of a UNION data type and declarations of variables of the type of a UNION.

Examples:

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

Determines declarations of variables of type BIT (possible within structure and function block definitions).

Examples:

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

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

Examples:

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
```

Get function of the property FB_Sub.SampleProperty:

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

SA0120: Program calls

Determines program calls.

Example:

SubProgram:

```
PROGRAM SubProgram
```

Program MAIN:

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

SA0121: Missing VAR_EXTERNAL declarations

Determines the use of a global variable in the function block, without it being declared as VAR_EXTERNAL (required according to the standard).

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.

Example:

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

Determines the use of expressions in the declaration of array boundaries.

Example:

```
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

Determines the use of the (TwinCAT-specific) operators INI, ADR, BITADR.

Example:

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

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

SA0147: Unusual shift operation - strict

Determines bit shift operations that are not performed on bit field data types (BYTE, WORD, DWORD, LWORD). 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.

See also non-strict rule SA0052 [▶ 32].

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

Determines bit access operations that are not performed on bit field data types (BYTE, WORD, DWORD, LWORD). 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.

See also non-strict rule SA0018 [▶ 21].

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

Determines initializations that do not assign constants.

Examples:

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

Determines all code locations where dereferenced pointers are used in the declaration part of POUs.

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 [▶ 50]
- Possible null pointer dereferences in the implementation part => SA0039 [▶ 51]

References:

- Use of references in the declaration part => SA0125 [▶ 50]
- Possible use of not initialized reference in the implementation part => SA0145 [▶ 53]

Interfaces:

- Possible use of not initialized interface in the implementation part => SA0046 [▶ 52]

SA0125: References in initializations

Determines all reference variables used for initialization in the declaration part of POUs.

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 [▶ 50]
- Possible null pointer dereferences in the implementation part => SA0039 [▶ 51]

References:

- Use of references in the declaration part => SA0125 [▶ 50]
- Possible use of not initialized reference in the implementation part => SA0145 [▶ 53]

Interfaces:

- Possible use of not initialized interface in the implementation part => SA0046 [▶ 52]

SA0039: Possible null pointer dereferences

Determines code positions at which a NULL pointer may be dereferenced.

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 [▶ 50]
- Possible null pointer dereferences in the implementation part => SA0039 [▶ 51]

References:

- Use of references in the declaration part => SA0125 [▶ 50]
- Possible use of not initialized reference in the implementation part => SA0145 [▶ 53]

Interfaces:

- Possible use of not initialized interface in the implementation part => SA0046 [▶ 52]

SA0046: Possible use of not initialized interface

Determines the use of interfaces that may not have been initialized before the use.

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 [▶ 50]
- Possible null pointer dereferences in the implementation part => SA0039 [▶ 51]

References:

- Use of references in the declaration part => SA0125 [▶ 50]

- Possible use of not initialized reference in the implementation part => SA0145 [▶ 53]

Interfaces:

- Possible use of not initialized interface in the implementation part => SA0046 [▶ 52]

SA0145: Possible use of not initialized reference

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.

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 [▶ 50]
- Possible null pointer dereferences in the implementation part => SA0039 [▶ 51]

References:

- Use of references in the declaration part => SA0125 [▶ 50]
- Possible use of not initialized reference in the implementation part => SA0145 [▶ 53]

Interfaces:

- Possible use of not initialized interface in the implementation part => SA0046 [▶ 52]

SA0140: Statements commented out

Determines statements that are commented out.

Example:

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

SA0150: Violations of lower or upper limits of the metrics

Determines function blocks that violate the enabled metrics at the lower or upper limit.

Example:

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

Determines recursive calls in actions, methods and properties of function blocks. Determines possible recursions through virtual function calls and interface calls.

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

Determines unpacked structures that are used in packed structures.

Example:

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

Determines points in the program that are not commented. 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).

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

Determines code positions with nested comments.

Examples:

The four nested comments identified accordingly in the following example 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

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.

Examples:

```
(*
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/in-out variables

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)

Example:

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

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.

Examples:

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
```

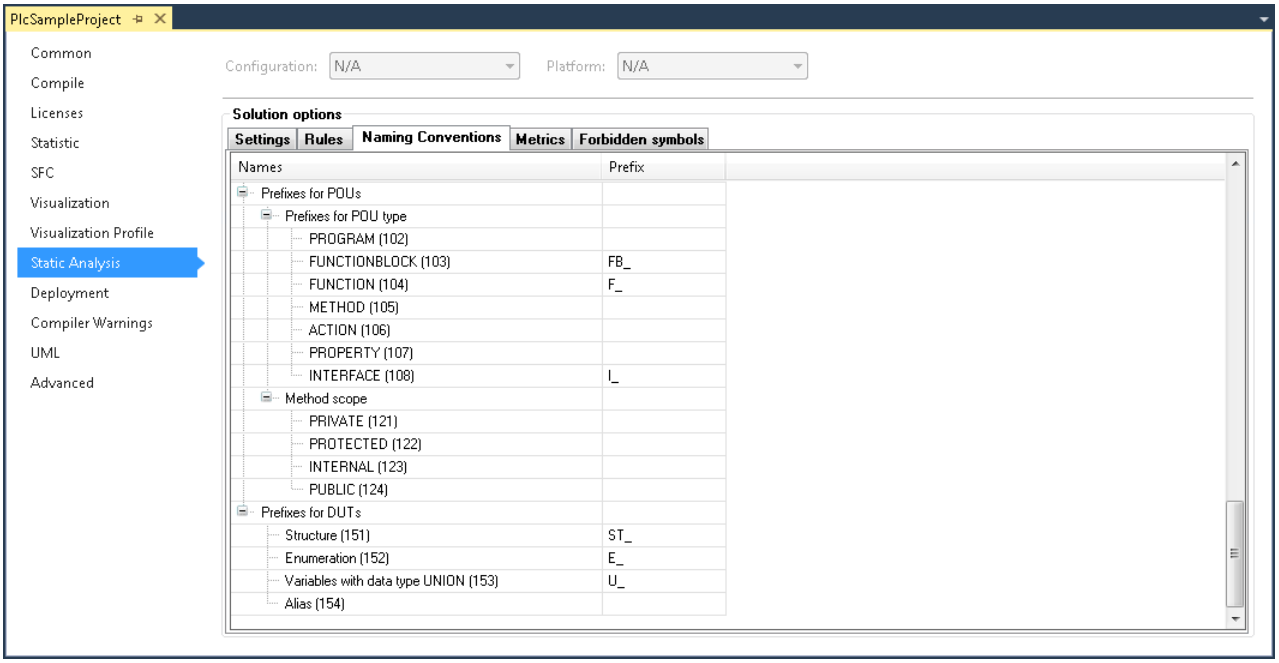
Program MAIN:

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

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

4.3 Naming conventions

In the **naming conventions** tab you can define naming conventions. Their compliance is accounted for in the static analysis execution [▶ 76]. 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.



Configuration of the naming conventions:

You can define the required conventions by entering the required prefix in the **Prefix** 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".

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) [▶ 67] dialog.

On the Naming conventions (2) [▶ 67] 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} [▶ 66], which you can use for the prefix definition of alias variables and properties.

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>"

Example 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 further information please refer to Pragmas and attributes [▶ 81].

Overview of naming conventions:

An overview of naming conventions can be found under Naming conventions – overview and description [▶ 59].

4.3.1 Naming conventions – overview and description

Overview

- **Prefixes for variables**
    - **Prefixes for types**
        - NC0003: BOOL [▶ 61]
        - NC0004: BIT [▶ 61]
        - NC0005: BYTE [▶ 61]
        - NC0006: WORD [▶ 61]
        - NC0007: DWORD [▶ 61]
        - NC0008: LWORD [▶ 61]
        - NC0013: SINT [▶ 61]
        - NC0014: INT [▶ 61]
        - NC0015: DINT [▶ 61]
        - NC0016: LINT [▶ 61]
        - NC0009: USINT [▶ 61]
        - NC0010: UINT [▶ 61]
        - NC0011: UDINT [▶ 61]
        - NC0012: ULINT [▶ 61]
        - NC0017: REAL [▶ 61]
        - NC0018: LREAL [▶ 61]
        - NC0019: STRING [▶ 61]
        - NC0020: WSTRING [▶ 61]
        - NC0021: TIME [▶ 61]
        - NC0022: LTIME [▶ 61]

- NC0023: DATE [▶ 61]
- NC0024: DATE_AND_TIME [▶ 61]
- NC0025: TIME_OF_DAY [▶ 61]
- NC0026: POINTER [▶ 62]
- NC0027: REFERENCE [▶ 62]
- NC0028: SUBRANGE [▶ 62]
- NC0030: ARRAY [▶ 62]
- NC0031: Function block instance [▶ 63]
- NC0036: Interface [▶ 63]
- NC0032: Structure [▶ 63]
- NC0029: ENUM [▶ 63]
- NC0033: Alias [▶ 64]
- NC0034: Union [▶ 64]
- NC0035: __XWORD [▶ 61]
- NC0037: __UXINT [▶ 61]
- NC0038: __XINT [▶ 61]

- **Prefixes for scopes**
  - NC0051: VAR_GLOBAL [▶ 64]
  - NC0070: VAR_GLOBAL CONSTANT [▶ 64]
  - NC0071: VAR_GLOBAL RETAIN [▶ 64]
  - NC0072: VAR_GLOBAL PERSISTENT [▶ 64]
  - NC0073: VAR_GLOBAL RETAIN PERSISTENT [▶ 64]
  - **VAR**
    - NC0053: Program variables [▶ 64]
    - NC0054: Function block variables [▶ 64]
    - NC0055: Function/method variables [▶ 64]
  - NC0056: VAR_INPUT [▶ 64]
  - NC0057: VAR_OUTPUT [▶ 64]
  - NC0058: VAR_IN_OUT [▶ 64]
  - NC0059: VAR_STAT [▶ 64]
  - NC0061: VAR_TEMP [▶ 64]
  - NC0062: VAR CONSTANT [▶ 64]
  - NC0063: VAR PERSISTENT [▶ 64]

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) [▶ 67].

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 [▶ 76].

```
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 [▶ 76].

```
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 [▶ 76].

```
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 [▶ 76].

```
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 [▶ 76].

```
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 [▶ 76].

```
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 [▶ 76].

```
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 [▶ 76].

```
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 [▶ 76].

```
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 [▶ 76].

```
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 [▶ 76].

```
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 [▶ 76].

```
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 [▶ 76].

```
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

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 [▶ 57] 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 [▶ 76].

Options

	Option	Functionality	Examples of enabled option	Examples of disabled option
1	First character after prefix should be an upper case letter	• If this option is enabled, the system checks whether the first character after the prefix is an upper-case character. • If this option is disabled, no such check takes place. Standard setting: disabled	If this option is enabled, an error is reported for the following declarations after the Static Analysis to indicate that the first character after the prefix must be upper-case. • Variable "bvar" with the expected prefix "b" (a correct identifier would be "bVar") • Function block "FB_sample" with the expected prefix "FB_" (a correct identifier would be "FB_Sample")	If this option is disabled, the identifiers "bvar" and "FB_sample" are permitted for the expected prefixes listed on the left. No upper/ lower case error is output.

2	Recursive prefixes for combinable data types	• If this option is 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 this option is disabled, only the prefix of the outermost data type is expected as the **data type prefix**. Standard setting: enabled	If this option is enabled, the following overall prefixes are expected when performing the Static Analysis. • For a variable of type "POINTER TO ARRAY[...] OF INT", the partial prefixes for POINTER (26), ARRAY (30) and INT (14) are expected as a composition. If the prefix "p" is configured for POINTER (26), the prefix "a" for ARRAY (30) and the prefix "n" for INT (14), the **data type prefix** "pan" is expected for a variable of type "POINTER TO ARRAY[...] OF INT". • For a variable of type "ARRAY[...] OF ARRAY[...] OF BOOL", the partial prefixes for ARRAY (30), ARRAY (30) and BOOL (3) are expected as a composition. If the prefix "a" has been configured for ARRAY (30) and the prefix "b" for BOOL (3), the **data type prefix** "aab" is expected for a variable of type "ARRAY[...] OF ARRAY[...] OF BOOL".	If this option is disabled, the following overall prefixes are expected when performing the Static Analysis. • Only the prefix for the outermost data type, in this case POINTER, is expected for a variable of type "POINTER TO ARRAY[...] OF INT". If the prefix "p" is configured for POINTER (26), the prefix "a" for ARRAY (30) and the prefix "n" for INT (14), the **data type prefix** "p" is expected for a variable of type "POINTER TO ARRAY[...] OF INT". • Only the prefix for the outermost data type, in this case ARRAY, is expected for a variable of type "ARRAY[...] OF ARRAY[...] OF BOOL". If the prefix "a" has been configured for ARRAY (30) and the prefix "b" for BOOL (3), the **data type prefix** "a" is expected for a variable of type "ARRAY[...] OF ARRAY[...] OF BOOL".

3	Combine **scope prefix** with **data type prefix** (namespace = scope)	• If this option is enabled, a variable must have the **prefix defined in the naming conventions** for its namespace followed by its **data type prefix**. • If this option is disabled, a variable must only have the prefix defined in the naming conventions **for its namespace**. The **data type prefix** is **not** expected after the namespace prefix. Standard setting: enabled	If this option is enabled, the following overall prefixes are expected when performing the Static Analysis. • For a variable of **type** "BOOL" declared within the **namespace** VAR_INPUT of a function block, the partial prefixes for VAR_INPUT (56) and BOOL (3) are expected as a composition. If the prefix "in_" has been configured for VAR_INPUT (56) and the prefix "b" for BOOL (3), the **overall prefix** "in_b" is expected for a corresponding variable. • For a variable of **type** "INT" declared within the VAR_GLOBAL **namespace** of a global variable list, the partial prefixes for VAR_GLOBAL (51) and INT (14) are expected as a composition. If the prefix "g" was configured for VAR_GLOBAL (51) and the prefix "n" for INT (14), the **overall prefix** "gn" is expected for a corresponding variable.	If this option is disabled, the following overall prefixes are expected when performing the Static Analysis. • Only the **prefix for namespace** VAR_INPUT (56) is expected for a variable declared within namespace VAR_INPUT of a function block. If the prefix "in_" has been configured for VAR_INPUT (56) and the prefix "b" for BOOL (3), the **overall prefix** "in_" is expected for a corresponding variable. • Only the **prefix for namespace** VAR_GLOBAL (51) is expected for a variable declared within namespace VAR_GLOBAL of a global variable list. If the prefix "g" was configured for VAR_GLOBAL (51) and the prefix "n" for INT (14), the **overall prefix** "g" is expected for a corresponding variable.

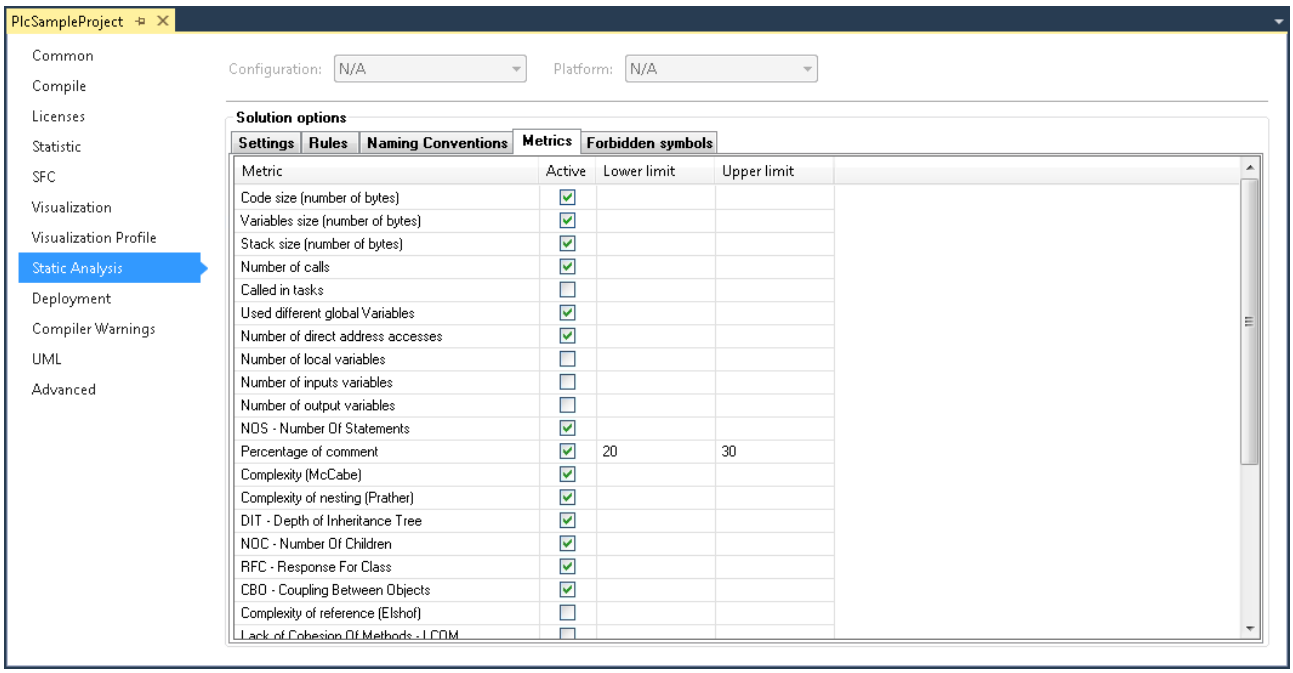
Further notes/samples:

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 [▶ 77] is executed.



Analysis of libraries

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

Configuration of the metrics:

You can enable or disable the individual metrics using the checkbox for the respective row. When command View Standard Metrics [▶ 77] is executed, the metrics that are enabled in the respective configuration are shown for each programming function 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.

Upper and lower limits:

For each metric you can define an individual upper and lower limit by entering the required number in the respective metric row (column **Lower limit** or **Upper limit**).
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 [▶ 77].
  - TwinCAT shows the enabled metrics for each programming function 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 [▶ 11] tab.
  - Perform the static analysis (see: Run Static Analysis [▶ 76]).

- 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 under Metrics - overview and description [▶ 71].

4.5.1 Metrics - overview and description

Overview

Column abbreviation in Standard Metrics view	Description
Code size	Code size [number of bytes] ("code size") [▶ 72]
Variables size	Variables size [number of bytes] ("variables size") [▶ 72]
Stack size	Stack size [number of bytes] ("stack size") [▶ 72]
Calls	Number of calls ("calls") [▶ 72]
Tasks	Called in tasks ("tasks") [▶ 72]
Globals	Used different global variables ("Globals") [▶ 72]
IOs	Number of direct address accesses ("IOs") [▶ 72]
Locals	Number of local variables ("locals") [▶ 72]
Inputs	Number input variables (inputs") [▶ 72]
Outputs	Number output variables ("outputs") [▶ 72]
NOS	Number of statements ("NOS") [▶ 72]
Comments	Percentage of comments ("comments") [▶ 72]
McCabe	Complexity (McCabe) ("McCabe") [▶ 73]
Prather	Complexitiy of nesting (Prather) ("Prather") [▶ 73]
DIT	Depth of inheritance tree ("DIT") [▶ 73]
NOC	Number of children ("NOC") [▶ 73]
RFC	Response for class ("RFC") [▶ 73]
CBO	Coupling between objects ("CBO") [▶ 73]
Elshof	Complexity of reference ("Elshof") [▶ 73]
LCOM	Lack of cohesion of methods ("LCOM") [▶ 73]
n1 (Halstead)	Halstead – number of different used operators (n1) [▶ 73]
N1 (Halstead)	Halstead – number of operators (N1) [▶ 73]
n2 (Halstead)	Halstead – number of different used operands (n2) [▶ 73]
N2 (Halstead)	Halstead – number of operands (N2) [▶ 73]
HL (Halstead)	Halstead – length (HL) [▶ 73]
HV (Halstead)	Halstead – volume (HV) [▶ 73]
D (Halstead)	Halstead – difficulty (D) [▶ 73]
SFC branches	Number of SFC branches [▶ 74]
SFC steps	Number of SFC steps [▶ 74]

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

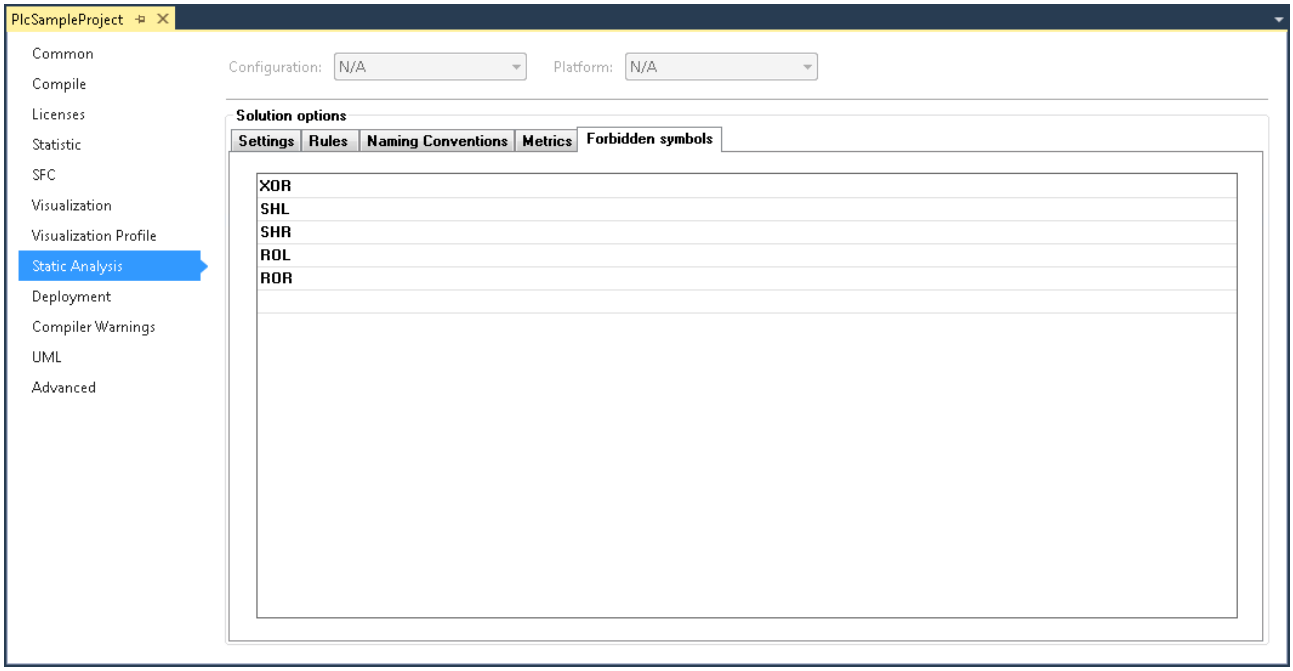
If the function block is implemented in the Sequential Function Chart language (SFC), this code metric indicates the number of branches in the function block.

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 configure the symbols that are taken into account when the static analysis is performed [▶ 76]. Examples of forbidden symbols are keywords or identifiers that must not be used in the code.



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>'“

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

5 Execution

5.1 Run Static Analysis

During execution of the static analysis, compliance with the coding rules, naming conventions and forbidden symbols is checked. The static analysis can be triggered manually (explicit) or performed automatically during the code generation (implicit).

TwinCAT issues the result of the static analysis, i.e. messages relating to rule violations, in the message window. The rules [▶ 11], naming conventions [▶ 57] and forbidden symbols [▶ 75] to be taken into account in the static analysis can be configured [▶ 10] 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 [▶ 11]).

Scope:

On execution of the static analysis using the **Run static analysis** command, the objects that are used in the application are checked. The scope of this command thus corresponds to the build commands **Build Project/Solution** or **Build new 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] [▶ 77].

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>"

Example 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>"

Example 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>'"

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

Please note that the code generation takes place before the static analysis. The static analysis only starts if the code generation was successful, i.e. if the compiler has not detected any compilation errors.

Implicit execution:

Implicit execution of the static analysis during each code generation can be enabled or disabled in the PLC project properties (Settings [▶ 10] tab). If you have activated the option **Perform static analysis automatically**, TwinCAT runs the static analysis directly after the successful code generation (as in the case of the command **Build Project**, for example).

Explicit execution:

Explicit execution of the static analysis can be initiated via the command **Run Static Analysis**, which can be found in the context menu of the PLC project or in the **Build** menu. You can also use this command for the explicit execution if you have activated implicit execution (option **Perform static analysis automatically**, see "Implicit execution" above).

The command first starts the code generation for the selected PLC project and then, if this is successful, the static analysis.

5.2 Run static analysis [check all objects]

Virtually the same information found on the documentation page for Run Static Analysis [▶ 76] also applies to the command **Run static analysis [check all objects]**. The two commands differ only in two points:

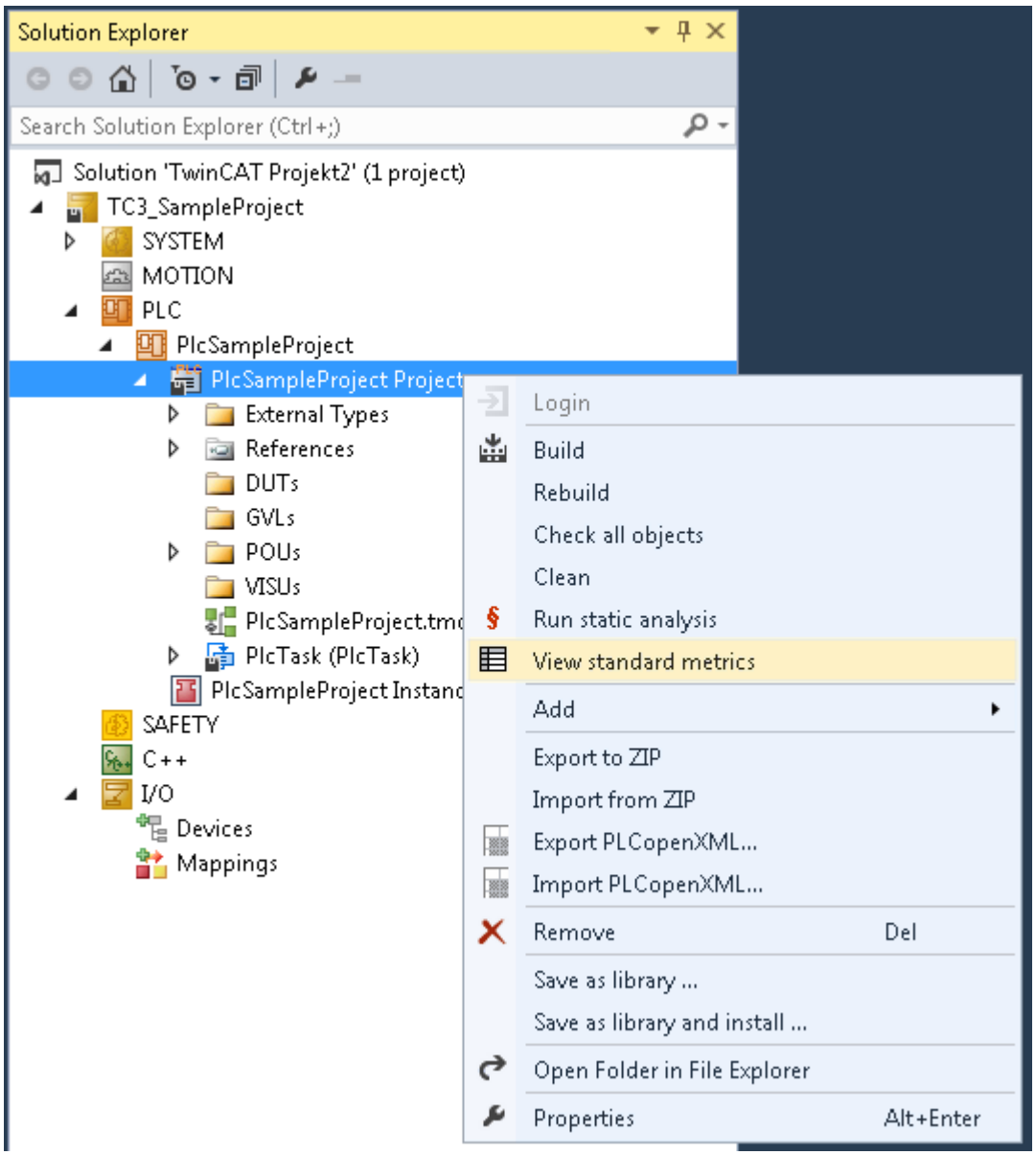
- firstly in the scope (see below)
- and secondly the "check all objects" variant cannot be executed implicitly, but only explicitly.

Scope:

On execution of the static analysis using the command **Run static analysis [check all objects]**, all objects located in the project tree of the PLC project are checked. This is primarily useful when creating libraries or when processing library projects. The area of application of this command thus corresponds to the build command **Check all objects**.

5.3 View Standard Metrics

The metrics can be displayed in a dedicated view by issuing the command **View Standard Metrics**, which can be found in the context menu of the PLC project or in the **Build** menu.



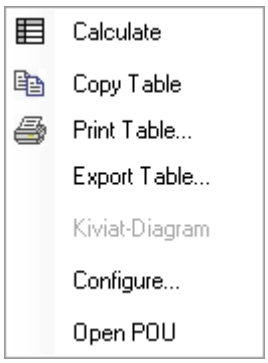
For the selected PLC project the command first starts the code generation (as with the command **Build** project, for example). For each programming function block TwinCAT then shows the metrics (indicators), which are enabled in the project properties, in a tabular **Standard Metrics** view (see Configuration of the metrics [▶ 69]). This configuration can also be accessed directly from **Standard Metrics** output window (see below: **Configure** as command in the context menu).

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.

Commands in the context menu

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).

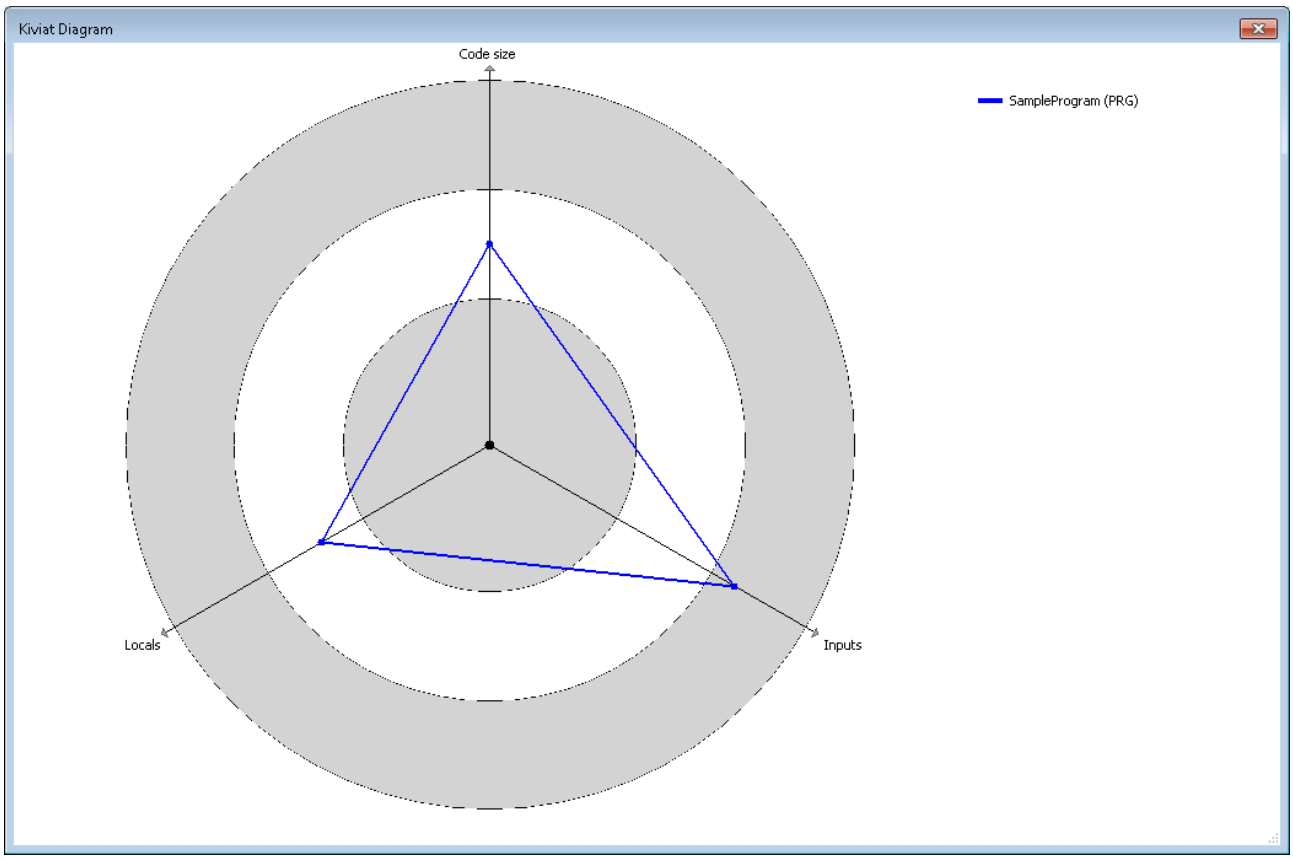
The following commands are available:

- **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 spider 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 metric configuration [▶ 69] 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 function 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 [▶ 11]
- Naming conventions [▶ 57]

Rules that are disabled in the project properties cannot be activated by a pragma or attribute.

Rule SA0004 cannot be disabled by a pragma.

The following section provides an overview and a detailed description of the available pragmas and attributes.

Overview

- Pragma {analysis ...} [▶ 81]
  - for disabling coding rules in the implementation part
  - can be used for individual code lines
- Attribute {attribute 'no-analysis'} [▶ 82]
  - 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' := '...'} [▶ 82]
  - for disabling coding rules in the declaration part
  - can be used for individual declarations or for whole programming objects
- Attribute {attribute 'naming' := '...'} [▶ 83]
  - for disabling naming conventions in the declaration part
  - can be used for individual declarations or for whole programming objects
- Attribute {attribute 'nameprefix' := '...'} [▶ 83]
  - 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'} [▶ 84]
  - 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 ...}

The pragma {analysis} can be used in the implementation part of a programming block in order to disable individual coding rules for the subsequent lines of code. It has to be entered twice: in the line above the respective code (rule is disabled) and in the line below (rule is enabled again). You have to specify the numbers of the respective rules: use a prefixed minus sign ("-") to disable, use a plus sign ("+") to enable again.

Syntax:

{analysis <sign><rule number>|,<further sign/rule number combinations, comma-separated>}

Examples:

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'}

The attribute {attributes 'no-analysis'} can be used above the declaration of a programming object, in order to exclude the whole programming object from the verification through the static analysis. For this programming object no checks are carried out for the coding rules, naming conventions and invalid symbols.

Syntax:

{attribute 'no-analysis'}

Examples:

```
{attribute 'qualified_only'}
{attribute 'no-analysis'}
VAR_GLOBAL
    …
END_VAR
```

```
{attribute 'no-analysis'}
PROGRAM MAIN
VAR
    …
END_VAR
```

Attribute {attribute 'analysis' := '...'}

The attribute {attribute 'analysis' := '<><rule number>'} can be added in the declaration part of a programming function block, in order to disable certain rules for individual declarations or for a whole programming object.

Syntax:

{attribute 'analysis' := '-<rule number>|,<further rule numbers, comma-separated>'}

Examples:

You want to disable rule 31 (unused signatures) for the structure.

```
{attribute 'analysis' := '-31'}
TYPE ST_Sample :
STRUCT
    bMember  : BOOL;
    nMember  : INT;
END_STRUCT
END_TYPE
```

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
```

You want to disable rule 100 (variable greater than <n> bytes) for the array "aNotReported":

```
{attribute 'analysis' := '-100'}
aNotReported  : ARRAY[1..10000] OF DWORD;
aReported     : ARRAY[1..10000] OF DWORD;
```

Attribute {attribute 'naming' := '...'}

The attribute {attribute 'naming' := '...'} can be used in the declaration part of POUs and DUTs, in order to exclude individual declaration lines from the check for compliance with the current naming conventions:

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

Example:

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' := '...'} can be added in the line before the declaration of a structured data type for defining a prefix. A naming convention then applies to the effect that identifiers for instances of this type must have this prefix.

Syntax:

{attribute ‘nameprefix’ := '<prefix>'}

Sample:

The prefix "ST_" is generally defined for structures in the **naming conventions** of the PLC project properties (NC0151). Instances of type "ST_Point" should start with the prefix "pt".

In the following static analysis sample a message is issued for "a" and "b", because the variable names do not start with "pt".

```
{attribute 'nameprefix' := 'pt'}
TYPE ST_Point :
STRUCT
    x  : INT;
    y  : INT;
END_STRUCT
END_TYPE
```

```
PROGRAM MAIN
VAR
    a    : ST_Point;         // => Invalid variable name 'a'. Expect prefix 'pt'
    b    : ST_Point;         // => Invalid variable name 'a'. Expect prefix 'pt'
    pt1  : ST_Point;
END_VAR
```

Attribute {attribute 'analysis:report-multiple-instance-calls'}

The attribute {attribute 'analysis:report-multiple-instance-call'} can be added in the declaration part of a function block whose instance should only be called once. In case that the global instance is called multiple times, the static analysis will generate a message.

Requirement: Rule SA0105 ("Multiple instance calls") is enabled in the Rules [▶ 11] category of the PLC project properties, i.e. the rule is configured as warning or error.

Syntax:

{attribute 'analysis:report-multiple-instance-calls'}

Example:

In the following example 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
```

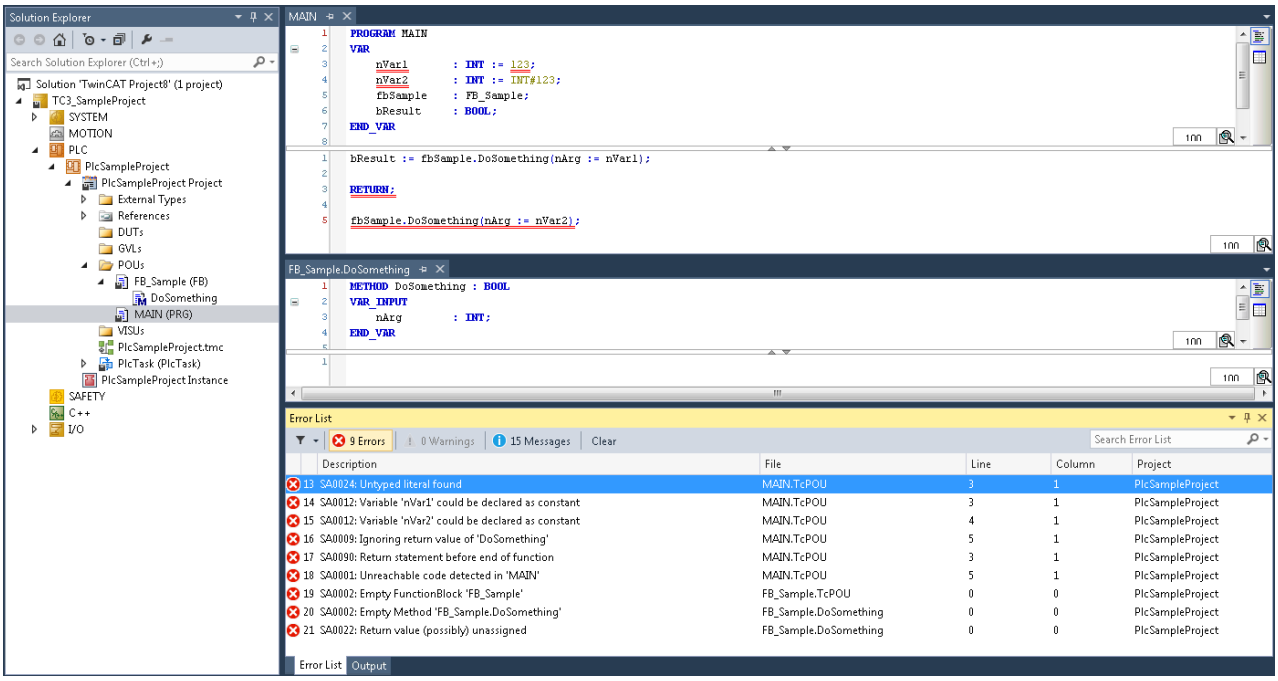
7 Examples

7.1 Static analysis

During execution of the static analysis [▶ 76], compliance with the coding rules [▶ 11], naming conventions [▶ 57] and forbidden symbols [▶ 75] is checked. The following section provides an example for each of these aspects.

1) Coding rules

In this example 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.

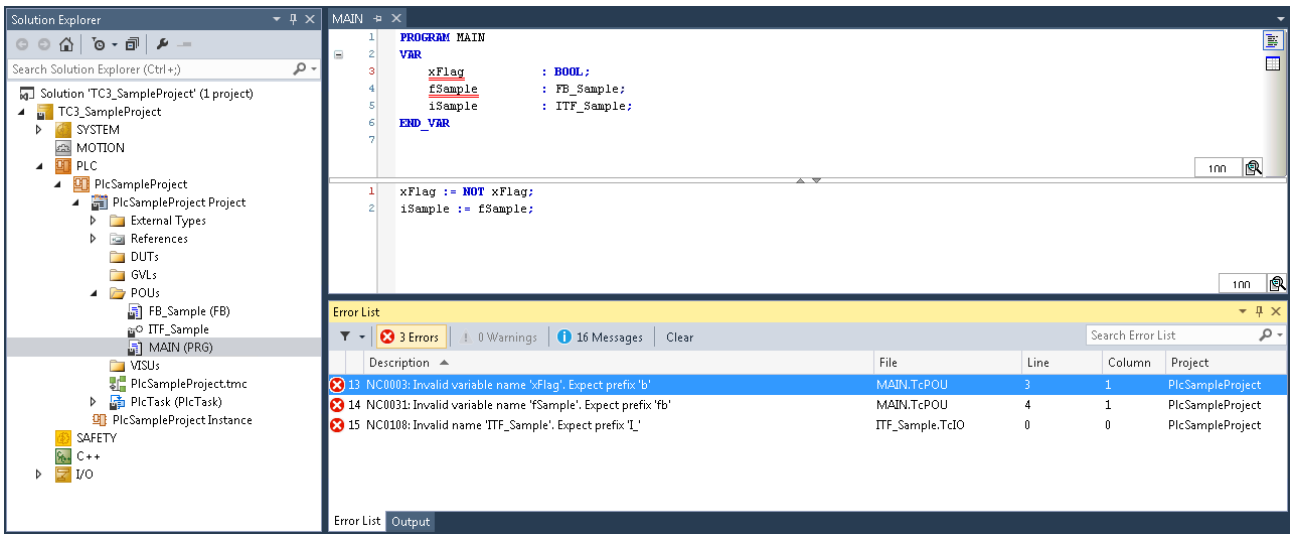


2) Naming conventions

The following naming conventions are configured:

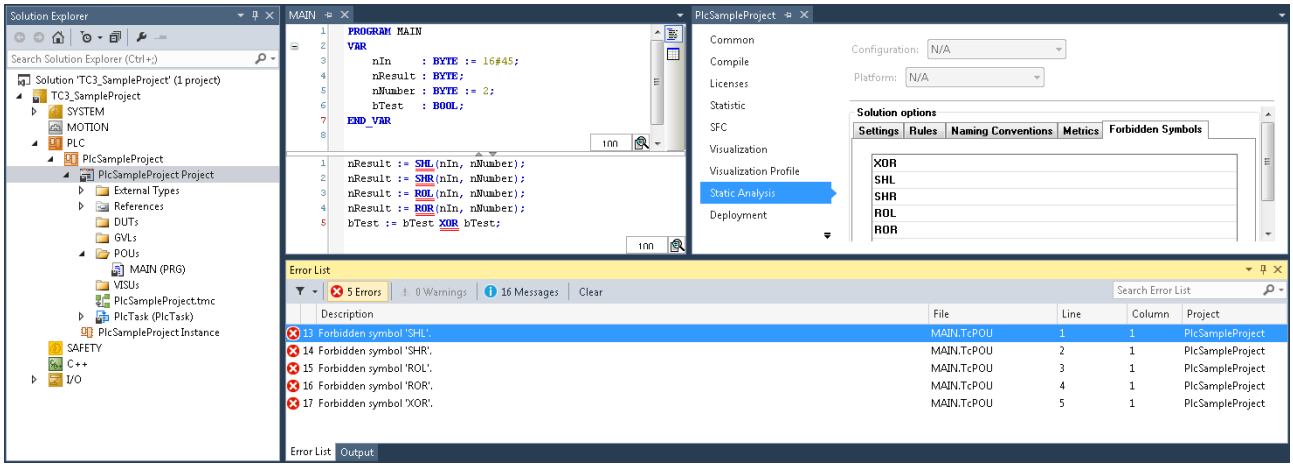
- Prefix "b" for variables of type BOOL (NC0003)
- Prefix "fb" for function block instances (NC0031)
- Prefix "FB_" for function blocks (NC0103)
- Prefix "I_" for interfaces (NC0108)

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.



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.



7.2 Standard metrics

An example for dealing with the standard metrics is provided below.

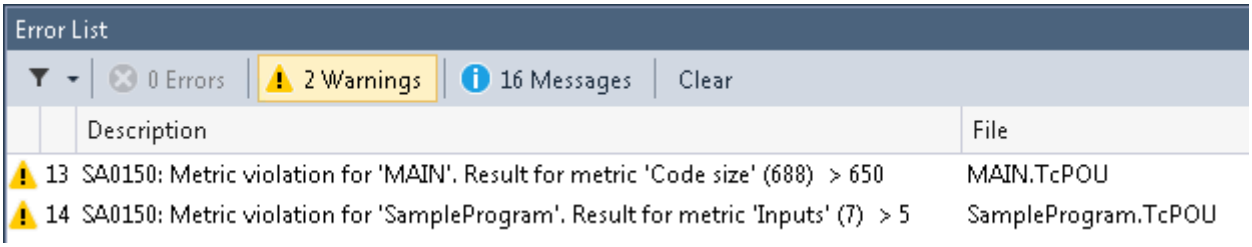
In this example "650" (= 650 bytes) is defined as upper limit for the metric "code size" and "5" as upper limit for the metric "number of input variables" (see: Configuration of the metrics [▶ 69]). In addition, rule 150 (SA0150: Violation of lower or upper metrics limits) is enabled and configured as warning.

When the command View Standard Metrics [▶ 77] 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 example, 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 [▶ 76] has been performed, the violation of the two upper limits is therefore reported as a warning in the message window.


Error List
0 Errors
2 Warnings
16 Messages
Clear
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