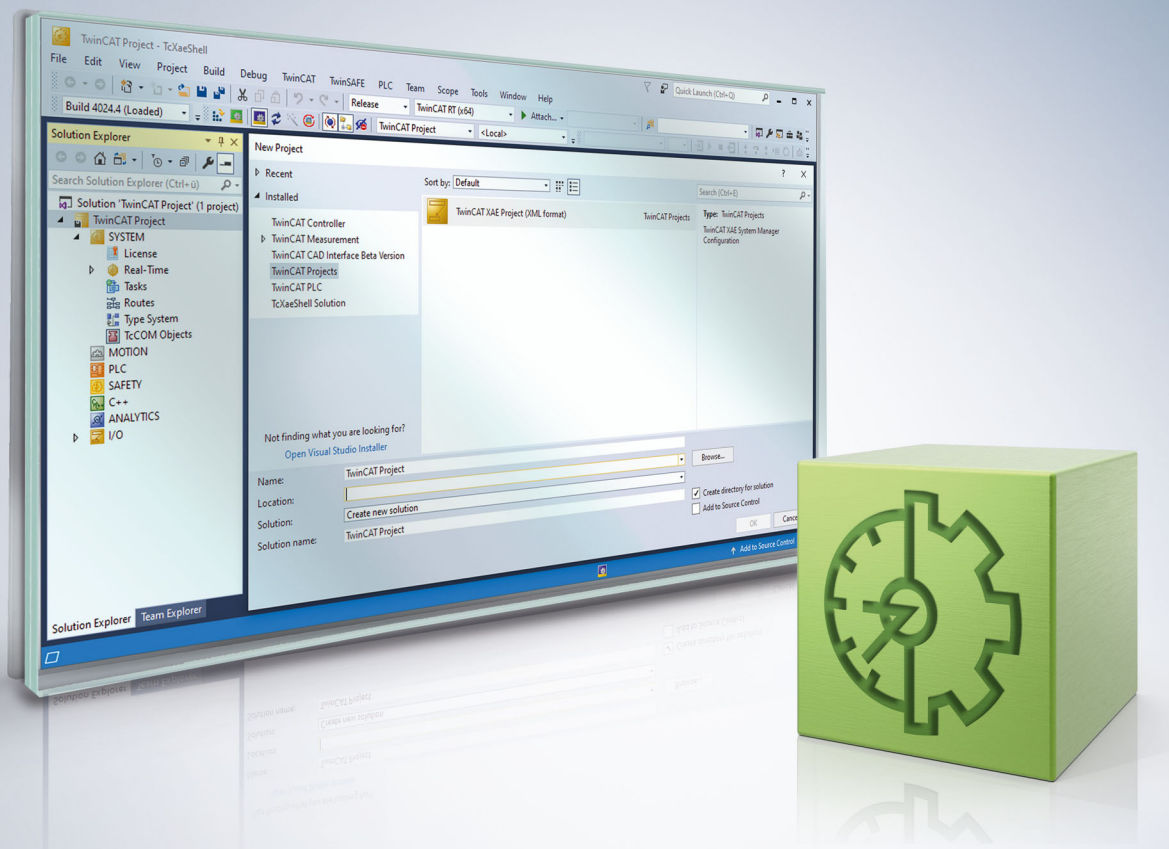
BECKHOFF New Automation Technology

Handbuch | DE

TE1200

TwinCAT 3 | PLC Static Analysis



19.02.2025 | Version: 2.8.0

Inhaltsverzeichnis

1 Vorwort

1.1 Hinweise zur Dokumentation

Diese Beschreibung wendet sich ausschließlich an ausgebildetes Fachpersonal der Steuerungs- und Automatisierungstechnik, das mit den geltenden nationalen Normen vertraut ist.
Zur Installation und Inbetriebnahme der Komponenten ist die Beachtung der Dokumentation und der nachfolgenden Hinweise und Erklärungen unbedingt notwendig.
Das Fachpersonal ist verpflichtet, stets die aktuell gültige Dokumentation zu verwenden.

Das Fachpersonal hat sicherzustellen, dass die Anwendung bzw. der Einsatz der beschriebenen Produkte alle Sicherheitsanforderungen, einschließlich sämtlicher anwendbaren Gesetze, Vorschriften, Bestimmungen und Normen erfüllt.

Disclaimer

Diese Dokumentation wurde sorgfältig erstellt. Die beschriebenen Produkte werden jedoch ständig weiterentwickelt.
Wir behalten uns das Recht vor, die Dokumentation jederzeit und ohne Ankündigung zu überarbeiten und zu ändern.
Aus den Angaben, Abbildungen und Beschreibungen in dieser Dokumentation können keine Ansprüche auf Änderung bereits gelieferter Produkte geltend gemacht werden.

Marken

Beckhoff®, TwinCAT®, TwinCAT/BSD®, TC/BSD®, EtherCAT®, EtherCAT G®, EtherCAT G10®, EtherCAT P®, Safety over EtherCAT®, TwinSAFE®, XFC®, XTS® und XPlanar® sind eingetragene und lizenzierte Marken der Beckhoff Automation GmbH.
Die Verwendung anderer in dieser Dokumentation enthaltenen Marken oder Kennzeichen durch Dritte kann zu einer Verletzung von Rechten der Inhaber der entsprechenden Bezeichnungen führen.

EtherCAT® ist eine eingetragene Marke und patentierte Technologie lizenziert durch die Beckhoff Automation GmbH, Deutschland

Copyright

Fremdmarken

In dieser Dokumentation können Marken Dritter verwendet werden. Die zugehörigen Markenvermerke finden Sie unter: https://www.beckhoff.com/trademarks.

1.2 Zu Ihrer Sicherheit

Sicherheitsbestimmungen

Lesen Sie die folgenden Erklärungen zu Ihrer Sicherheit.
Beachten und befolgen Sie stets produktspezifische Sicherheitshinweise, die Sie gegebenenfalls an den entsprechenden Stellen in diesem Dokument vorfinden.

Haftungsausschluss

Die gesamten Komponenten werden je nach Anwendungsbestimmungen in bestimmten Hard- und Software-Konfigurationen ausgeliefert. Änderungen der Hard- oder Software-Konfiguration, die über die dokumentierten Möglichkeiten hinausgehen, sind unzulässig und bewirken den Haftungsausschluss der Beckhoff Automation GmbH & Co. KG.

Qualifikation des Personals

Diese Beschreibung wendet sich ausschließlich an ausgebildetes Fachpersonal der Steuerungs-, Automatisierungs- und Antriebstechnik, das mit den geltenden Normen vertraut ist.

Signalwörter

Im Folgenden werden die Signalwörter eingeordnet, die in der Dokumentation verwendet werden. Um Personen- und Sachschäden zu vermeiden, lesen und befolgen Sie die Sicherheits- und Warnhinweise.

Warnungen vor Personenschäden

⚠ GEFAHR

Es besteht eine Gefährdung mit hohem Risikograd, die den Tod oder eine schwere Verletzung zur Folge hat.

⚠ WARNUNG

Es besteht eine Gefährdung mit mittlerem Risikograd, die den Tod oder eine schwere Verletzung zur Folge haben kann.

⚠ VORSICHT

Es besteht eine Gefährdung mit geringem Risikograd, die eine mittelschwere oder leichte Verletzung zur Folge haben kann.

Warnung vor Umwelt- oder Sachschäden

HINWEIS

Es besteht eine mögliche Schädigung für Umwelt, Geräte oder Daten.

Information zum Umgang mit dem Produkt

Diese Information beinhaltet z. B.:
Handlungsempfehlungen, Hilfestellungen oder weiterführende Informationen zum Produkt.

1.3 Hinweise zur Informationssicherheit

Die Produkte der Beckhoff Automation GmbH & Co. KG (Beckhoff) sind, sofern sie online zu erreichen sind, mit Security-Funktionen ausgestattet, die den sicheren Betrieb von Anlagen, Systemen, Maschinen und Netzwerken unterstützen. Trotz der Security-Funktionen sind die Erstellung, Implementierung und ständige Aktualisierung eines ganzheitlichen Security-Konzepts für den Betrieb notwendig, um die jeweilige Anlage, das System, die Maschine und die Netzwerke gegen Cyber-Bedrohungen zu schützen. Die von Beckhoff verkauften Produkte bilden dabei nur einen Teil des gesamtheitlichen Security-Konzepts. Der Kunde ist dafür verantwortlich, dass unbefugte Zugriffe durch Dritte auf seine Anlagen, Systeme, Maschinen und Netzwerke verhindert werden. Letztere sollten nur mit dem Unternehmensnetzwerk oder dem Internet verbunden werden, wenn entsprechende Schutzmaßnahmen eingerichtet wurden.

Zusätzlich sollten die Empfehlungen von Beckhoff zu entsprechenden Schutzmaßnahmen beachtet werden. Weiterführende Informationen über Informationssicherheit und Industrial Security finden Sie in unserem https://www.beckhoff.de/secguide.

Die Produkte und Lösungen von Beckhoff werden ständig weiterentwickelt. Dies betrifft auch die Security-Funktionen. Aufgrund der stetigen Weiterentwicklung empfiehlt Beckhoff ausdrücklich, die Produkte ständig auf dem aktuellen Stand zu halten und nach Bereitstellung von Updates diese auf die Produkte aufzuspielen. Die Verwendung veralteter oder nicht mehr unterstützter Produktversionen kann das Risiko von Cyber-Bedrohungen erhöhen.

Um stets über Hinweise zur Informationssicherheit zu Produkten von Beckhoff informiert zu sein, abonnieren Sie den RSS Feed unter https://www.beckhoff.de/secinfo.

1.4 Informationen zur Security-Risikoanalyse

Wenn Sie dieses Produkt installiert haben, stellt Beckhoff Ihnen folgende Informationen für eine Security-Risikoanalyse Ihres Systems bereit.

Engineering

Workload: TwinCAT.Standard.XAE

Um die Function TE1200 nutzen zu können, ist lediglich die Installation des genannten Standard-Workloads erforderlich.

Sehen Sie auch:

- Informationen zur Security-Risikoanalyse des TwinCAT-Standard-XAE-Workloads

Runtime

Es werden keine Runtime-Komponenten installiert bzw. benötigt.

2 Überblick

Mit der Integration der statischen Codeanalyse steht in TwinCAT 3.1 ein weiteres Programmierwerkzeug zur Verfügung, das den Entwicklungsprozess von SPS-Software unterstützt. Das Tool ist in TwinCAT 3 SPS integriert und als Ergänzung des Compilers zu sehen.

Funktionsüberblick

Im Static Analysis sind mehr als 100 zum Teil parametrisierbare Kodierregeln implementiert, die Sie zu individuellen Regelwerken kombinieren können. In einigen Regeln werden die in "PLCopen Coding Guidelines" definierten Regelsätze berücksichtigt. Beispielsweise kann gemeldet werden, wenn eine Zeigervariable vor einer Dereferenzierung nicht auf ungleich 0 überprüft worden ist. Dadurch wird der Anwender auf möglicherweise unbeabsichtigte und fehlerhafte Implementierungen hingewiesen, sodass diese Programmstellen frühzeitig optimiert werden können.

Außerdem können Sie für jeden möglichen Datentyp eine Namenskonvention definieren, auf dessen Einhaltung dann geprüft wird. Darüber hinaus stehen Ihnen über 20 Metriken zur Verfügung, die den zugrundeliegenden Quellcode analysieren und charakterisieren. Bei regelmäßiger Berechnung können die Metriken Hinweise auf negative Trends und Abweichungen von Qualitätszielen geben. Die Kennzahlen stellen somit ein Indiz zur Beurteilung der Softwarequalität dar. In der tabellarischen Ausgabe sind beispielsweise Metriken für die Anzahl an Anweisungen oder den Anteil an Kommentaren zu finden.

Das Static Analysis kann manuell angestoßen oder automatisch mit der Codeerzeugung durchgeführt werden. Das Ergebnis der Analyse, also Meldungen bezüglich Abweichungen der Vorgaben und Regeln, gibt TwinCAT im Meldungsfenster aus. In den SPS-Projekteigenschaften definieren Sie, was im Einzelnen geprüft werden soll. Beim Konfigurieren der Regeln können Sie außerdem jeweils definieren, ob eine Regelverletzung als Fehler oder Warnung ausgegeben werden soll. Mit Hilfe von Pragma-Anweisungen können Sie einzelne Teile des Codes von der Prüfung ausnehmen. Für Fehler, die vom Static Analysis auf Basis von Precompile-Informationen gemeldet werden, gibt es im ST-Editor eine Unterstützung zur unmittelbaren Fehlerbehebung (QuickFix/Precompile [▶ 134]).

Nutzen

Das Static Analysis hilft dabei, einen besser lesbaren Code zu schreiben sowie bereits während der Programmierung potenzielle Fehlerquellen aufzudecken. Dadurch steigt zum einen die Codequalität und zum anderen kann bei der Entwicklung von Applikationen und bei der Fehlersuche viel Zeit eingespart werden.

Die Missachtung einer Kodierregel deutet im Allgemeinen auf eine Implementierungsschwachstelle hin, deren Korrektur eine frühzeitige Fehlerbehebung bzw. -vermeidung ermöglicht. Die automatische Kontrolle der anwenderspezifischen Namenskonventionen sorgt darüber hinaus dafür, dass die Steuerungsprogramme hinsichtlich der Typ- und Variablennamen standardisiert entwickelt werden können. Dadurch erhalten unterschiedliche SPS-Projekte, die auf Basis der gleichen Namenskonventionen implementiert wurden, ein einheitliches Look-and-feel, was die Lesbarkeit der Programme stark verbessert. Ergänzend dazu stellen die Metriken ein Indiz zur Beurteilung der Softwarequalität dar.

Funktionalitäten

Nachfolgend sind die Funktionalitäten von "TwinCAT 3 PLC Static Analysis" als Übersicht dargestellt:

- Statische Analyse:
  - Funktion: Die Statische Analyse überprüft den Quellcode eines Projekts auf Abweichungen von bestimmten Kodierregeln, Namenskonventionen und unzulässigen Symbolen. Die Ausgabe des Ergebnisses erfolgt im Meldungsfenster.
  - Konfiguration: Die gewünschten Kodierregeln, Namenskonventionen und unzulässigen Symbole konfigurieren Sie in den Projekteigenschaften des SPS-Projekts in den Registerkarten Regeln [▶ 17], Namenskonventionen [▶ 83] und Unzulässige Symbole [▶ 114].
- Standard-Metriken:
  - Funktion: Auf Ihren Quellcode werden Metriken angewendet, die die Eigenschaften der Software in Kennzahlen ausdrücken (z. B. Anzahl der Anweisungen oder Anteil an Kommentaren). Sie stellen ein Indiz zur Beurteilung der Softwarequalität dar. Die Ausgabe der Standard-Metriken erfolgt in der Ansicht **Standard-Metriken**.

- Konfiguration: Die gewünschten Metriken konfigurieren Sie in den Projekteigenschaften des SPS-Projekts in der Registerkarte Metriken [▶ 97].

Alternativ besteht die Möglichkeit, eine lizenzfreie Variante vom Static Analysis zu verwenden, welche einen stark reduzierten Funktionsumfang zur Verfügung stellt. Einen detaillierten Vergleich der Funktionen der lizenzfreien und der lizenzpflichtigen Variante vom Static Analysis finden Sie im Kapitel Installation [▶ 11].

Weiterführende Informationen zur Installation, Konfiguration und Ausführung des "Static Analysis" finden Sie auf den folgenden Seiten:

- Installation [▶ 11]
- Konfiguration der Einstellungen, Regeln, Namenskonventionen, Metriken und unzulässigen Symbole [▶ 14]
- Befehl 'Statische Analyse durchführen' [▶ 115]
- Befehl 'Statische Analyse durchführen [Überprüfe alle Objekte]' [▶ 117]
- Befehl 'Standard-Metriken anzeigen' [▶ 118]
- Befehl 'Standard-Metriken anzeigen [Überprüfe alle Objekte]' [▶ 120]
- Pragmas und Attribute [▶ 125]
- Beispiele [▶ 139]
- Automation Interface Unterstützung [▶ 136]

Bibliotheken

TwinCAT analysiert nur den Applikationscode des aktuellen SPS-Projekts, die referenzierten Bibliotheken bleiben unbeachtet!

Wenn Sie das Bibliotheksprojekt hingegen geöffnet haben, können Sie die enthaltenen Elemente mit Hilfe des Befehls Befehl 'Statische Analyse durchführen [Überprüfe alle Objekte]' [▶ 117] überprüfen.

Punktuelle Ausschaltung von Prüfungen

Mit Hilfe von Pragmas und Attributen [▶ 125] können Sie Prüfungen für bestimmte Codeteile ausschalten.

Static Analysis übers Automation Interface

Bitte beachten Sie die Möglichkeit, das Static Analysis teilweise über das Automation Interface bedienen zu können (siehe Automation Interface Unterstützung [▶ 136]).

3 Installation

Die Function "TE1200 | TwinCAT 3 PLC Static Analysis" wird bereits mit der Installation der TwinCAT-3-Entwicklungsumgebung installiert. Entsprechend existiert kein separates TE1200-Setup/-Paket. Die zusätzliche Engineering-Komponente TE1200 muss lediglich lizenziert werden. Informationen zu einem lizenzfreien Testmodus finden Sie unter Lizenzierung [▶ 13].

TwinCAT Package Manager: Installation (TwinCAT 3.1 Build 4026)

Eine ausführliche Anleitung zur Installation von Produkten finden Sie im Kapitel Workloads installieren in der Installationsanleitung TwinCAT 3.1 Build 4026.

Installieren Sie den folgenden Workload, um das Produkt nutzen zu können:

- TwinCAT.Standard.XAE

TwinCAT Setup: Installation (TwinCAT 3.1 Build 4024 und früher)

Installieren Sie das folgende Setup, um das Produkt nutzen zu können:

- TwinCAT 3.1 eXtended Automation Engineering (XAE) (Vollinstallation)

Eine ausführliche Anleitung zur Installation finden Sie im Kapitel Installation bis TwinCAT 3.1 Build 4024.

3.1 Funktionsumfang: Light vs. Full

Ohne die Engineering-Lizenz für TE1200 können Sie die lizenzfreie Variante des Static Analysis (Static Analysis Light) nutzen, welche einige Einschränkungen beinhaltet (siehe Tabelle unten). Anhand der kostenfreien Light-Variante können Sie sich – auf Basis eines stark reduzierten Funktionsumfangs – beispielsweise mit dem prinzipiellen Handling des Produkts vertraut machen.

Static Analysis Light vs. Static Analysis Full

Nachfolgend finden Sie eine Übersicht über den unterschiedlichen Funktionsumfang der lizenzfreien und der lizenzpflichtigen Variante vom Static Analysis.

Funktionsaspekt	Static Analysis Light (ohne TE1200-Lizenz)	Static Analysis Full (mit TE1200-Lizenz)
Lizenz erforderlich	Nein, kostenfrei nutzbar	Ja, TE1200-Lizenz erforderlich
(Regel-) Konfiguration speichern/ exportieren und laden/importieren	Nicht möglich, gekoppelt an SPS-Projekteigenschaften	Möglich (mit Hilfe der Schaltflächen **Laden/ Speichern** in den Einstellungen [▶ 14])
Kopplung der Ausführung an den Übersetzungsprozess	Ja, nicht konfigurierbar	Konfigurierbar (mit Hilfe der Option **Statische Analyse automatisch durchführen** in den Einstellungen [▶ 14]; Manuelle Ausführung mit Hilfe des Befehls Befehl 'Statische Analyse durchführen' [▶ 115])
Überprüfung von nicht genutzten Objekten (z.B. innerhalb eines Bibliotheksprojekts)	Nicht möglich	Möglich (mit Hilfe des Befehls Befehl 'Statische Analyse durchführen [Überprüfe alle Objekte]' [▶ 117])
Maximale Anzahl an berichteten Fehlern	500 (nicht konfigurierbar) (Weiterführende Informationen zu der Bedeutung von 500 als maximale Fehleranzahl finden Sie in den Einstellungen [▶ 14])	Konfigurierbar (mit Hilfe der Einstellung **Maximale Anzahl an Fehlern** in den Einstellungen [▶ 14])
Maximale Anzahl an berichteten Warnungen	Ausgabe von Warnungen nicht möglich (siehe folgende Zeile)	Konfigurierbar (mit Hilfe der Einstellung **Maximale Anzahl an Warnungen** in den Einstellungen [▶ 14])
Regeln: Aktivierungsmöglichkeiten [▶ 17]	• Aktiv und Ausgabe als Fehler • Inaktiv	• Aktiv und Ausgabe als Fehler • Aktiv und Ausgabe als Warnung • Inaktiv
Regeln: Umfang [▶ 18]	7 Kodierregeln • SA0033: Nicht verwendete Variablen • SA0028: Überlappende Speicherbereiche • SA0006: Schreibzugriff auf mehrere Tasks • SA0004: Mehrfacher Schreibzugriff auf Ausgang • SA0027: Mehrfachverwendung des Namens • SA0167: Temporäre Funktionsbausteininstanzen • SA0175: Verdächtige Operation auf String	Mehr als 100 Kodierregeln
Regeln: Precompile-Unterschlängelung, QuickFix [▶ 134]	Nicht verfügbar	Verfügbar
Namenskonventionen [▶ 83]	Nicht verfügbar	Verfügbar
Metriken [▶ 97]	Nicht verfügbar	Verfügbar

Unzulässige Symbole [▶ 114]	Nicht verfügbar	Verfügbar
Pragmas und Attribute [▶ 125] zur temporären Deaktivierung von Regeln	Ja, im Light-Umfang verfügbar: • Pragma {analysis ...} • Attribut {attribute 'no-analysis'} • Attribut {attribute 'analysis' := '...'}	Ja, im Full-Umfang verfügbar: • Pragma {analysis ...} • Attribut {attribute 'no-analysis'} • Attribut {attribute 'analysis' := '...'} • Attribut {attribute 'naming' := '...'} • Attribut {attribute 'nameprefix' := '...'} • Attribut {attribute 'analysis:report-multiple-instance-calls'}

3.2 Lizenzierung

Für Informationen zur Lizenzierung der Engineering-Komponente TE1200 lesen Sie bitte die Dokumentation zur Lizenzierung.

Testmodus

Bitte beachten Sie, dass für dieses Produkt keine 7-Tage-Testlizenz verfügbar ist. Ohne die Engineering-Lizenz für TE1200 können Sie die lizenzfreie Variante des Static Analysis (Static Analysis Light) nutzen, welche einige Einschränkungen beinhaltet. Anhand der kostenfreien Light-Variante können Sie sich – auf Basis eines stark reduzierten Funktionsumfangs – beispielsweise mit dem prinzipiellen Handling des Produkts vertraut machen.

Sehen Sie dazu auch: Funktionsumfang: Light vs. Full [▶ 11]

3.3 Systemvoraussetzungen

Engineering (XAE)

Technische Daten	Voraussetzungen
Betriebssystem	• Windows 7 • Windows 10 • Windows 11
Zielplattform	• x86 • x64
TwinCAT-Version	TwinCAT 3.1 Build 4022 und höher
Erforderliche TwinCAT-Lizenz	TE1200 Engineering-Lizenz

4 Konfiguration

Nach der Installation [▶ 11] und Lizenzierung der "TE1200 | TwinCAT 3 PLC Static Analysis" wird die Kategorie **Static Analysis** in den Projekteigenschaften des SPS-Projekts um die zusätzlichen Regeln und Konfigurationsmöglichkeiten erweitert.

In den Projekteigenschaften finden Sie dann Registerkarten für die Basiskonfiguration und für die Zusammenstellung der Regeln, Konventionen, Metriken und unzulässigen Symbole, die bei der Code-Überprüfung beachtet werden.

Die Projekteigenschaften eines SPS-Projekts können Sie über das Kontextmenü des SPS-Projektobjekts oder über das Menü **Projekt** öffnen, falls ein SPS-Projekt im Projektbaum fokussiert ist.

Die aktuellen Einstellungen bzw. Änderungen werden gespeichert, sobald Sie die SPS-Projekteigenschaften speichern. Über die Schaltfläche **Speichern**, die Sie in der Registerkarte **Einstellungen** finden, können Sie die aktuelle Konfiguration der Static Analysis zusätzlich in eine externe Datei speichern. Eine solche Konfigurationsdatei können Sie über die Schaltfläche **Laden** wieder in die Entwicklungsumgebung laden.

Auf den folgenden Seiten finden Sie weiterführende Informationen zu den einzelnen Registerkarten der Projekteigenschaftskategorie **Static Analysis**.

- Einstellungen [▶ 14]
- Regeln [▶ 17]
- Namenskonventionen [▶ 83]
- Namenskonventionen (2) [▶ 94]
- Metriken [▶ 97]
- Unzulässige Symbole [▶ 114]

Geltungsbereich der "Static Analysis"-Konfiguration

Die Einstellungen, die Sie in der Kategorie **Static Analysis** der SPS-Projekteigenschaften vornehmen, sind sogenannte **Solution options** und wirken sich daher nicht nur auf das SPS-Projekt aus, dessen Eigenschaften Sie aktuell bearbeiten. Die konfigurierten Einstellungen, Regeln, Namenskonventionen, Metriken und unzulässigen Symbole werden für alle SPS-Projekte übernommen, die sich in der Entwicklungsumgebung befinden.

4.1 Einstellungen

In der Registerkarte **Einstellungen** können Sie konfigurieren, ob die statische Codeanalyse automatisch mit der Codeerzeugung durchgeführt soll. Des Weiteren können Sie die aktuelle Konfiguration des **Static Analysis** in einer externen Datei speichern bzw. eine Konfiguration aus einer externen Datei laden.

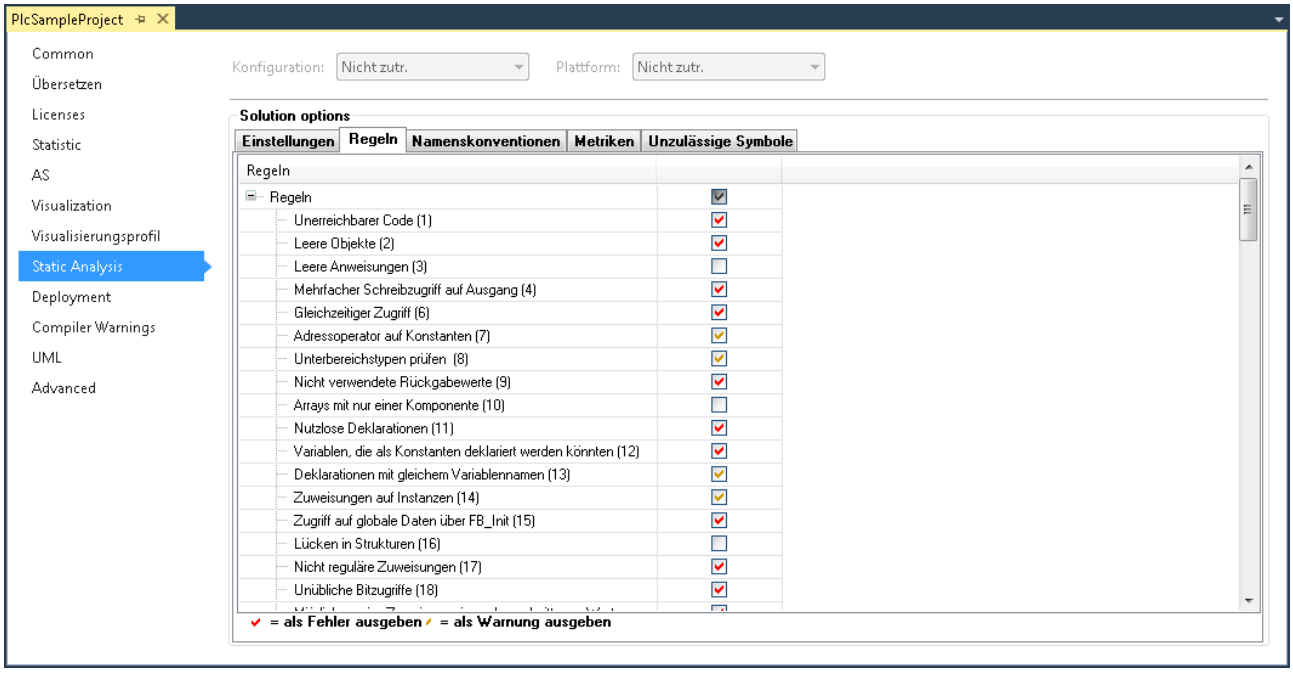
Statische Analyse automatisch nach dem Kompilieren durchführen	Wenn diese Option aktiviert ist, führt TwinCAT die Statische Analyse im Anschluss an jede fehlerfreie Codegenerierung durch (beispielsweise bei Befehl **Projekt erstellen**). Unabhängig von der Konfiguration dieser Option können Sie die Prüfung manuell über den Befehl 'Statische Analyse durchführen' [▶ 115] starten.
Laden	Diese Schaltfläche öffnet den Standarddialog zum Suchen nach einer Datei. Wählen Sie die gewünschte Konfigurationsdatei *.csa für die Statische Analyse aus, die möglicherweise zuvor über **Speichern** angelegt wurde (siehe unten). Da es sich bei den Eigenschaften der Static Analysis um "Solution options" handelt, werden die Projekteigenschaften für die Statische Analyse, wie sie in der csa-Datei beschrieben sind, für alle SPS-Projekte übernommen die sich in der Entwicklungsumgebung befinden.
Speichern	Diese Schaltfläche dient dem Speichern der aktuellen Projekteigenschaften für die Statische Analyse in einer xml-Datei. Der Standarddialog zum Speichern einer Datei erscheint und der Dateityp "Static analysis files" (*.csa) ist bereits eingestellt. Eine solche Datei können Sie später über die Schaltfläche **Laden** wieder im Projekt anwenden (siehe oben). Bitte beachten Sie, dass die Einstellung des Fehlerlimits „Maximale Anzahl an Fehlern“ nicht in dieser Datei gespeichert wird.

Maximale Anzahl an Fehlern	Voreinstellung: 500 In diesem Feld können Sie das gewünschte Fehlerlimit eingeben, das während der Ausführung des Static Analysis überprüft wird. Wenn entweder das **Fehlerlimit oder** das **Warnungslimit (siehe unten) erreicht** ist, wird die **Ausführung** des Static Analysis **abgebrochen** und das **bisherige Analyseergebnis** wird **ausgegeben**. **Performance vs. Vollständigkeit:** Bitte beachten Sie: Je mehr Objekte vom Static Analysis überprüft werden, desto länger dauert die Ausführung des Static Analysis. Und je mehr Fehler in dem Ausgabefenster eingetragen werden, desto länger dauert die Ergebnisausgabe vom Static Analysis. Für den angenommenen Fall, dass in einem SPS-Projekt mehr als 500 Static Analysis Fehler vorhanden sind, ergeben sich daher die folgenden Anwendungsfälle. • Verwendung eines kleinen Fehlerlimits (z. B. 500): Sie möchten die ausgegebenen Fehler nach und nach abarbeiten, indem Sie den entsprechenden Programmcode korrigieren und das Static Analysis zur Überprüfung der Korrektur erneut ausführen. Dann wäre es nicht erforderlich, alle Objekte auf einmal zu prüfen und alle Fehler auf einmal anzuzeigen. Sondern in diesem Fall genügt es i.d.R., eine Teilmenge als Static Analysis Ergebnis angezeigt zu bekommen, wobei die Ausführung des Static Analysis performant abläuft. • Verwendung eines großen Fehlerlimits (z. B. 5000): Sie möchten einen Gesamtbericht vom Static Analysis ausgegeben bekommen, um den Gesamtaufwand zur Korrektur des Programmcodes ungefähr abschätzen zu können. Dieses Ziel können Sie durch Erhöhung des Fehlerlimits erreichen. Bitte beachten Sie, dass die Ausführung des Static Analysis je nach Projektsituation (deutlich) länger dauert, je höher das Fehlerlimit konfiguriert ist. **Detaillierte Verhaltenserläuterung:** Wenn in einem Projekt mehr als 500 Static Analysis Fehler vorhanden sind, dann bedeutet eine Konfiguration von 500 als Fehlerlimit nicht, dass das Static Analysis exakt 500 Fehler ausgibt. Während der Ausführung des Static Analysis passiert vielmehr folgendes: Vor der Überprüfung einer weiteren POU wird überprüft, ob die bisher gefundenen Static Analysis Fehler bereits das konfigurierte Limit überschreiten. Wenn dies der Fall ist, wird die weitere Ausführung des Static Analysis abgebrochen und das bisherige Analyseergebnis wird ausgegeben. Wenn das Limit hingegen noch nicht überschritten wurde, wird diese POU vom Static Analysis geprüft und die in dieser POU gefundenen Fehler werden zum Analyseergebnis hinzugefügt. Wenn diese neu gebildete Fehlersumme (z. B. 530) das konfigurierte Fehlerlimit übersteigt, wird die Ausführung des Static Analysis vor der Überprüfung der nächsten POU abgebrochen und die bisher gefundenen Fehler (z. B. 530) werden ausgegeben.
Maximale Anzahl an Warnungen	Voreinstellung: 500 In diesem Feld können Sie das gewünschte Warnungslimit eingeben, das während der Ausführung des Static Analysis überprüft wird. Wenn entweder das **Fehlerlimit** (siehe oben) **oder** das **Warnungslimit erreicht** ist, wird die **Ausführung** des Static Analysis **abgebrochen** und das **bisherige Analyseergebnis** wird **ausgegeben**. Weiterführende Informationen zu den Anwendungsfällen können Sie der Beschreibung der Option „Maximale Anzahl an Fehlern“ entnehmen (siehe oben).

4.2 Regeln

In der Registerkarte **Regeln** können Sie die Regeln konfigurieren, die bei der Durchführung der Statischen Analyse [▶ 115] berücksichtigt werden. Die Regeln werden in den Projekteigenschaften als Baumstruktur angezeigt. Teilweise sind einige Regeln unterhalb von organisatorischen Knoten angeordnet.

Mithilfe der Regeln wird der Anwender auf möglicherweise unbeabsichtigte und fehlerhafte Implementierungen hingewiesen, sodass diese Programmstellen frühzeitig optimiert werden können.



Standardeinstellungen

Mit Ausnahme von SA0016, SA0024, SA0073, SA0101, SA0105-SA0107, SA0111-SA0125, SA0133, SA0134, SA0145, SA0147, SA0148, SA0150, SA0162-SA0167 und den "Strikten IEC-Regeln" sind standardmäßig alle Regeln aktiviert.

Konfiguration der Regeln

Sie können die einzelnen Regeln über das Kontrollkästchen der jeweiligen Zeile aktivieren oder deaktivieren. Wenn Sie das Kontrollkästchen eines Unterknotens auswählen, wirkt sich dies auf alle Einträge unterhalb dieses Knotens aus. Wenn Sie das Kontrollkästchen des obersten Regeln-Knotens auswählen, wirkt sich dies auf alle Einträge der Liste aus. Die Einträge unterhalb eines Knotens werden durch Mausklick auf das Minus- bzw. Pluszeichen vor dem Knotentitel zu- bzw. aufgeklappt.

Die Nummer in Klammern hinter jeder Regel, zum Beispiel "Unerreichbarer Code (1)", ist die Regelnummer, die bei einer Nichteinhaltung der Regel ausgegeben wird.

Es gibt folgende drei Einstellungen, zwischen denen Sie durch wiederholtes Klicken auf das Kontrollkästchen wechseln können:

-  : Die Regel wird nicht geprüft.
-  : Eine Regelverletzung wird als Fehler im Meldungsfenster ausgegeben.
-  : Eine Regelverletzung wird als Warnung im Meldungsfenster ausgegeben.

Syntax von Regelverletzungen im Meldungsfenster

Jede Regel besitzt eine eindeutige Nummer (in der Konfigurationsansicht der Regeln in runden Klammern hinter der Regel dargestellt). Wenn während der Statischen Analyse die Verletzung einer Regel festgestellt wird, wird die Nummer zusammen mit einer Fehler- bzw. Warnungsbeschreibung gemäß folgender Syntax im Meldungsfenster ausgegeben. Die Abkürzung "SA" weist dabei auf "Static Analysis" hin.

Syntax: **"SA<Regelnummer>: <Regelbeschreibung>"**

Beispiel für Regelnummer 33 (Nicht verwendete Variablen): "SA0033: Nicht verwendet: Variable 'bSample'"

Temporäre Deaktivierung von Regeln

Regeln, die hier im Dialog aktiviert sind, können über ein Pragma im Projekt temporär abgeschaltet werden. Weiterführende Informationen hierzu finden Sie unter Pragmas und Attribute [▶ 125].

Übersicht und Beschreibung der Regeln

Eine Übersicht der Regeln sowie eine detaillierte Beschreibung der Regeln finden Sie unter Regeln - Übersicht und Beschreibung [▶ 18].

4.2.1 Regeln - Übersicht und Beschreibung

Strikte IEC-Regeln prüfen

Die Prüfungen unterhalb des Knotens "Strikte IEC-Regeln prüfen" ermitteln Funktionalitäten und Datentypen, die in TwinCAT in Erweiterung der IEC61131-3 erlaubt sind.

Gleichzeitigen/Konkurrierenden Zugriff prüfen

Zu diesem Thema existierenden die folgenden Regeln:

SA0006: Schreibzugriff aus mehreren Tasks [▶ 24]
Ermittelt Variablen, auf die von mehr als einer Task geschrieben wird.

SA0103: Gleichzeitiger Zugriff auf nicht-atomare Daten [▶ 61]
Ermittelt nicht-atomare Variablen (zum Beispiel mit Datentyp STRING, WSTRING, ARRAY, STRUCT, FB-Instanzen, 64-Bit Datentypen), die in mehr als einer Task verwendet werden.

Bitte beachten Sie, dass nur direkte Zugriffe erkannt werden können. Indirekte Zugriffe, beispielsweise per Pointer/Referenz, werden nicht aufgelistet.

Bitte beachten Sie außerdem die Dokumentation zum Thema "Multitask-Datenzugriffs-Synchronisation in der SPS", in der einige Hinweise zur Notwendigkeit und den Möglichkeiten einer Datenzugriffs-Synchronisation enthalten sind.

Übersicht

Detaillierte Beschreibung

SA0001: Unerreichbarer Code

Funktion	Ermittelt Code, der nicht ausgeführt wird, beispielweise wegen einer RETURN oder CONTINUE Anweisung.
Begründung	Unerreichbarer Code sollte in jedem Fall vermieden werden. Häufig weist die Prüfung darauf hin, dass noch Testcode enthalten ist, der wieder entfernt werden sollte.
Wichtigkeit	Hoch
PLCopen-Regel	CP2

Beispiel 1 – RETURN:

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

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

Beispiel 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: Leere Objekte

Funktion	Ermittelt POUs, GVLs oder Datentypdeklarationen, die keinen Code enthalten.
Begründung	Leere Objekte sollten vermieden werden. Sie sind oft ein Zeichen dafür, dass ein Objekt nicht vollständig implementiert ist. Ausnahme: In manchen Fällen wird dem Rumpf eines Funktionsblocks kein Code geben, wenn dieser nur über Schnittstellen verwendet werden soll. In anderen Fällen wird eine Methode nur anlgelegt, weil sie von einer Schnittstelle gefordert wird, ohne dass für die Methode eine sinnvolle Implementierung möglich ist. In jedem Fall sollte eine solche Situation kommentiert werden.
Wichtigkeit	Mittel

SA0003: Leere Anweisungen

Funktion	Ermittelt Codezeilen, die ein Semikolon (;), aber keine Anweisung enthalten.
Begründung	Eine leere Anweisung kann ein Anzeichen für fehlenden Code sein.
Ausnahme	Es gibt sinnvolle Verwendungen leerer Anweisungen. Beispielsweise kann es sinnvoll sein, in einer CASE-Anweisung alle Fälle explizit auszuprogrammieren, auch die, in denen nichts zu tun ist. Wenn eine solche leere CASE-Anweisung mit einem Kommentar versehen ist, erzeugt die statische Codeanalyse keine Fehlermeldung.
Wichtigkeit	Niedrig

Beispiele:

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

Das folgende Beispiel erzeugt für den Zustand 2 den Fehler "SA0003: Empty statement".

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

Das folgende Beispiel erzeugt keinen SA0003-Fehler.

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

SA0004: Mehrfacher Schreibzugriff auf Ausgang

Funktion	Ermittelt Ausgänge, die an mehr als einer Position geschrieben werden.
Begründung	Die Wartbarkeit leidet, wenn ein Ausgang an verschiedenen Stellen im Code geschrieben wird. Es ist dann unklar, welcher Schreibzugriff derjenige ist, der tatsächlich Auswirkungen im Prozess hat. Gute Praxis ist es, die Berechnung der Ausgangsvariablen in Hilfsvariablen durchzuführen und an einer Stelle am Ende des Zyklus den berechneten Wert zuzuweisen.
Ausnahme	Es wird kein Fehler ausgegeben, wenn eine Ausgangsvariable in verschiedenen Zweigen von IF- bzw. CASE-Anweisungen geschrieben wird.
Wichtigkeit	Hoch
PLCopen-Regel	CP12

Diese Regel kann **nicht** über ein Pragma oder Attribut abgeschaltet werden!
Weitere Informationen zu Attributen finden Sie unter Pragmas und Attribute [▶ 125].

Beispiel:

Globale Variablenliste:

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

Programm 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: Schreibzugriff aus mehreren Tasks

Funktion	Ermittelt Variablen, auf die von mehr als einer Task geschrieben wird.
Begründung	Eine Variable, die in mehreren Tasks geschrieben wird, kann unter Umständen ihren Wert unerwartet ändern. Das kann zu verwirrenden Situationen führen. Stringvariablen und auf einigen 32-Bit-Systemen auch 64-Bit-Integer-Variablen können sogar einen inkonsistenten Zustand bekommen, wenn die Variable gleichzeitig in zwei Tasks geschrieben wird.
Ausnahme	In bestimmten Fällen kann es nötig sein, dass mehrere Tasks eine Variable schreiben. Stellen Sie dann sicher, beispielsweise durch die Verwendung von Semaphoren, dass der Zugriff nicht zu einem inkonsistenten Zustand führt.
Wichtigkeit	Hoch
PLCopen-Regel	CP10

Sehen Sie auch die Regel SA0103 [▶ 61].

Aufruf entspricht Schreibzugriff

Bitte beachten Sie, dass Aufrufe als Schreibzugriff interpretiert werden. Beispielsweise wird der Aufruf einer Methode für eine Funktionsbausteininstanz als Schreibzugriff auf die Funktionsbausteininstanz angesehen. Eine genauere Analyse der Zugriffe und Aufrufe ist z.B. aufgrund von virtuellen Aufrufen (Zeiger, Interface) nicht möglich.

Wenn Sie die Regel SA0006 für eine Variable (z.B. für eine Funktionsbausteininstanz) deaktivieren möchten, können Sie das folgende Attribut oberhalb der Variablendeklaration einfügen: {attribute 'analysis' := '-6'}

Beispiele:

Die beiden globalen Variablen nVar und bVar werden von zwei Tasks geschrieben.

Globale Variablenliste:

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

Programm MAIN_Fast, aufgerufen von der Task PlcTaskFast:

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

Programm MAIN_Slow, aufgerufen von der Task PlcTaskSlow:

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

SA0007: Adressoperator auf Konstanten

Funktion	Ermittelt Stellen, an denen der ADR-Operator bei einer Konstanten angewendet wird.
Begründung	Durch einen Pointer auf eine konstante Variable hebt man die CONSTANT-Eigenschaft der Variable auf. Über den Pointer kann die Variable verändert werden, ohne dass der Compiler dies meldet.
Ausnahme	In seltenen Fällen kann es sinnvoll sein, einen Pointer auf eine Konstante an eine Funktion zu übergeben. Es muss dann allerdings gewährleistet sein, dass diese Funktion den übergebenen Wert nicht ändert. Verwenden Sie in diesem Fall wenn möglich VAR_IN_OUT CONSTANT.
Wichtigkeit	Hoch

Wenn die Option **Konstanten ersetzen** in den Compiler-Optionen der SPS-Projekteigenschaften aktiviert ist, ist der Adressoperator für skalare Konstanten (Integer, BOOL, REAL) nicht erlaubt und ein Übersetzungsfehler wird ausgegeben. (Konstante Strings, Strukturen und Arrays haben immer eine Adresse.)

Beispiel:

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

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

SA0008: Unterbereichstypen prüfen

Funktion	Ermittelt Bereichsüberschreitungen von Unterbereichstypen. Zugewiesene Literale werden bereits vom Compiler geprüft. Wenn Konstanten zugeordnet sind, müssen die Werte innerhalb des definierten Bereichs liegen. Wenn Variablen zugeordnet sind, müssen die Datentypen identisch sein.
Begründung	Wenn Unterbereichstypen verwendet werden, dann sollte sichergestellt werden, dass dieser Unterbereich nicht verlassen wird. Der Compiler überprüft solche Unterbereichsverletzungen nur für Zuweisungen von Konstanten.
Wichtigkeit	Niedrig

Die Prüfung wird nicht für CFC-Objekte durchgeführt, da die Codestruktur dies nicht zulässt.

Beispiel:

```
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: Nicht verwendete Rückgabewerte

Funktion	Ermittelt Funktions-, Methoden- und Eigenschaftenaufrufe, bei denen der Rückgabewert nicht verwendet wird.
Begründung	Wenn eine Funktion oder eine Methode einen Rückgabewert liefert, dann sollte dieser auch ausgewertet werden. Häufig wird im Rückgabewert mitgeliefert, ob die Funktion erfolgreich ausgeführt werden konnte. Wenn keine Auswertung erfolgt, kann man später nicht mehr erkennen, ob der Rückgabewert übersehen wurde, oder ob er tatsächlich nicht benötigt wird.
Ausnahme	Wenn ein Rückgabewert beim Aufruf nicht von Interesse ist, sollte dies dokumentiert und die Zuweisung weglassen werden. Fehlerrückgaben sollten nie ignoriert werden!
Wichtigkeit	Mittel
PLCopen-Regel	CP7/CP17

Beispiel:

Funktion F_ReturnBOOL:

```
FUNCTION F_ReturnBOOL : BOOL
```

```
F_ReturnBOOL := TRUE;
```

Programm MAIN:

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

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

SA0010: Arrays mit nur einer Komponente

Funktion	Ermittelt Arrays, die nur eine einzige Komponente enthalten.
Begründung	Ein Array mit einer Komponente kann durch eine Variable vom Basistyp ersetzt werden. Der Zugriff auf diese Variable ist deutlich schneller als der Zugriff mit Index auf eine Variable.
Ausnahme	Häufig wird die Länge eines Arrays über eine Konstante festgelegt und ist ein Parameter für ein Programm. Das Programm kann dann mit Arrays von verschiedener Länge arbeiten und muss nicht geändert werden, wenn die Länge nur 1 beträgt. Eine solche Situation sollte entsprechend dokumentiert werden.
Wichtigkeit	Niedrig

Beispiele:

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

SA0011: Nutzlose Deklarationen mit nur einer einzigen Komponente

Funktion	Ermittelt Strukturen, Unions oder Enumerationen mit nur einer einzigen Komponente.
Begründung	Es sollten keine Strukturen, Unions oder Enumerationen mit nur einer einzigen Komponente deklariert werden. Solche Deklarationen können für Leser verwirrend sein. Eine Struktur mit nur einem Element kann durch einen Aliastyp ersetzt werden. Eine Enumeration mit einem Element kann durch eine Konstante ersetzt werden.
Wichtigkeit	Niedrig
PLCopen-Regel	CP22/CP24

Beispiel 1 – Struktur:

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

Beispiel 2 – Union:

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

Beispiel 3 – Enumeration:

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

SA0012: Variablen, die als Konstanten deklariert werden könnten

Funktion	Ermittelt Variablen, auf die nicht schreibend zugegriffen wird und die deshalb als Konstante deklariert werden könnten.
Begründung	Wenn eine Variable nur an der Deklarationsstelle geschrieben und sonst nur lesend verwendet wird, dann nimmt die statische Analyse an, dass die Variable auch nicht geändert werden soll. Eine Deklaration als Konstante führt dann erstens dazu, dass auch bei Programmänderungen überprüft wird, dass die Variable nicht verändert wird. Zweitens führt die Deklaration als Konstante unter Umständen zu schnellerem Code.
Wichtigkeit	Niedrig

Beispiel:

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

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

SA0013: Deklarationen mit gleichem Variablennamen

Funktion	Ermittelt Variablen, die den gleichen Namen haben wie andere Variablen (Beispiel: globale und lokale Variablen mit gleichen Namen), oder wie Funktionen, Aktionen, Methoden oder Eigenschaften (Properties) innerhalb des gleichen Zugriffsbereichs.
Begründung	Gleiche Namen können beim Lesen des Codes verwirrend sein und sie können zu Fehlern führen, wenn unbeabsichtigt auf das falsche Objekt zugegriffen wird. Es wird empfohlen, Namenskonventionen zu verwenden, deren Einhaltung solche Situationen vermeidet.
Wichtigkeit	Mittel
PLCopen-Regel	N5/N9

Beispiele:

Globale Variablenliste GVL_App:

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

Programm MAIN, welches eine Methode mit dem Namen Sample beinhaltet:

```
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: Zuweisungen auf Instanzen

Funktion	Ermittelt Zuweisungen auf Funktionsbausteininstanzen. Bei Instanzen mit Pointer- oder Referenzvariablen können diese Zuweisungen riskant sein.
Begründung	Dies ist eine Performance-Warnung. Wenn eine Instanz einer anderen Instanz zugewiesen wird, dann werden alle Elemente und Unterelemente von der einen Instanz in die andere kopiert. Pointer auf Daten werden mitkopiert, jedoch nicht deren referenzierte Daten, so dass die Zielinstanz und die Quellinstanz nach der Zuweisung die gleichen Daten enthalten. Je nach Größe der Instanzen kann eine solche Zuweisung sehr lange dauern. Wenn eine Instanz beispielsweise zur Bearbeitung an eine Funktion übergeben werden soll, dann ist es sehr viel performanter, einen Pointer auf die Instanz zu übergeben. Um selektiv Werte von einer Instanz in eine andere zu kopieren, kann eine Kopiermethode sinnvoll sein: `fb2.CopyFrom(fb1)`
Wichtigkeit	Mittel

Beispiel:

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

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

SA0015: Zugriff auf globale Daten über FB_init

Funktion	Ermittelt Zugriffe eines Funktionsbausteins auf globale Daten über die FB_init-Methode. Der Wert dieser Variablen hängt von der Reihenfolge der Initialisierungen ab!
Begründung	Je nach Deklarationsstelle der Instanz eines Bausteins kann es sein, dass bei Verletzung der Regel auf eine nicht-initialisierte Variable zugegriffen wird.
Wichtigkeit	Hoch

Beispiel:

Globale Variablenliste GVL_App:

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

Funktionsbaustein FB_Sample:

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

Methode 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
```

Programm MAIN:

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

SA0016: Lücken in Strukturen

Funktion	Ermittelt Lücken in Strukturen oder Funktionsbausteinen, verursacht durch die Alignment-Anforderungen des aktuell eingestellten Zielsystems. Wenn möglich, sollten Sie die Lücken durch Umsortieren der Strukturelemente oder durch Auffüllen mit einem Dummy-Elemententfernen. Wenn dies nicht möglich ist, dann können Sie die Regel für die betroffenen Strukturen durch das Attribut {attribute 'analysis' := '...'} [▶ 127] deaktivieren.
Begründung	Durch unterschiedliche Alignment-Anforderungen auf verschiedenen Plattformen, kann es für solche Strukturen zu einem unterschiedlichen Layout im Speicher kommen. Der Code kann sich dann je nach Plattform unterschiedlich verhalten.
Wichtigkeit	Niedrig

Beispiele:

```
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: Nicht-reguläre Zuweisungen auf Pointer-Variable

Funktion	Ermittelt Zuweisungen auf Pointer, die keine Adresse (ADR-Operator, Zeigervariablen) oder Konstante 0 sind.
Begründung	Wenn ein Pointer einen Wert zugewiesen bekommt, der keine gültige Adresse ist, dann führt die Dereferenzierung des Pointers zu einer „Access Violation Exception“ (Exception bei Zugriffsverletzung).
Wichtigkeit	Hoch

Beispiel:

```
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: Unübliche Bitzugriffe

Funktion	Ermittelt Bitzugriffe auf vorzeichenbehaftete Variablen. Die Norm IEC 61131-3 erlaubt allerdings nur Bitzugriffe auf Bitfelder. Sehen Sie hierzu auch die strikte Regel SA0148 [▶ 69].
Begründung	Vorzeichenbehaftete Datentypen sollten nicht als Bitfelder verwendet werden und umgekehrt. Die Norm IEC 61131-3 sieht solche Zugriffe nicht vor. Diese Regel muss eingehalten werden, wenn der Code protierbar sein soll.
Ausnahme	Ausnahme für Flag-Enumerationen: Wenn eine Enumeration mit Hilfe des Pragmaattributs {attribute 'flags'} als Flag deklariert ist, wird für Bitzugriffe mit den Operationen OR, AND oder NOT der Fehler SA0018 nicht ausgegeben.
Wichtigkeit	Mittel

Beispiele:

```
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: Möglicherweise Zuweisung eines abgeschnittenen Werts an REAL-Variable

Funktion	Ermittelt Operationen auf Integer-Variablen, bei denen möglicherweise ein abgeschnittener Wert an eine Variable vom Datentyp REAL zugewiesen wird.

Begründung	Die statische Codeanalyse gibt einen Fehler aus, wenn das Ergebnis einer Integerberechnung einer REAL- oder LREAL-Variablen zugewiesen wird. Der Programmierer soll dabei auf eine möglicherweise fehlerhafte Interpretation einer solchen Zuweisung aufmerksam gemacht werden:
	`fLEAL := nDINT1 * nDINT2.`
	Da der Wertebereich von LREAL größer ist als der von DINT, könnte angenommen werden, dass das Ergebnis der Rechnung in jedem Fall in LREAL dargestellt wird. Das ist aber nicht der Fall. Der Prozessor berechnet das Ergebnis der Multiplikation als Integer und castet anschließend das Ergebnis nach LREAL. Ein Überlauf in der Integer-Berechnung würde verloren gehen. Um das Problem zu umgehen, muss die Rechnung bereits als REAL-Operation erfolgen:
	`fLREAL := TO_LREAL(nDINT1) * TO_LREAL(nDINT2)`

Wichtigkeit	Hoch

Beispiel:

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

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

SA0021: Weitergabe der Adresse einer temporären Variablen

Funktion	Ermittelt Zuweisungen von Adressen von temporären Variablen (Variablen auf dem Stack) zu nicht-temporären Variablen.
Begründung	Lokale Variablen einer Funktion oder einer Methode werden auf dem Stack angelegt und existieren nur während der Abarbeitung der Funktion oder Methode. Zeigt ein Pointer nach Abarbeitung der Methode oder Funktion auf eine solche Variable, dann kann über diesen Pointer in undefinierten Speicher gegriffen, oder auf eine falsche Variable in einer anderen Funktion zugegriffen werden. Diese Situation ist in jedem Fall zu vermeiden.
Wichtigkeit	Hoch

Beispiel:

Methode FB_Sample.SampleMethod:

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

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

Programm MAIN:

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

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

SA0022: (Möglicherweise) nicht zugewiesene Rückgabewerte

Funktion	Ermittelt alle Funktionen und Methoden, die einen Ausführungsstrang ohne Zuweisung auf den Rückgabewert enthalten.
Begründung	Ein nicht zugewiesener Rückgabewert in einer Funktion oder Methode deutet auf fehlenden Code hin. Auch wenn der Rückgabewert in jedem Fall einen Standardwert hat, ist es immer sinnvoll, diesen nochmal explizit zuzuweisen, um Unklarheiten zu vermeiden.
Wichtigkeit	Mittel

Beispiel:

```
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: Komplexe Rückgabewerte

Funktion	Ermittelt komplexe Rückgabewerte, die mit einer einfachen Registerkopie des Prozessors nicht zurückgegeben werden können. Dazu gehören Strukturen und Arrays sowie Rückgabewerte vom Typ STRING (unabhängig von der Größe des belegten Speicherplatzes).
Begründung	Dies ist eine Performance-Warnung. Wenn große Werte als Ergebnis einer Funktion, Methode oder einer Eigenschaft zurückgeliefert werden, dann werden diese vom Prozessor bei der Ausführung des Codes mehrfach umkopiert. Das kann zu Laufzeitproblemen führen und sollte wenn möglich vermieden werden. Eine bessere Performance wird erreicht, wenn ein strukturierter Wert als VAR_IN_OUT an eine Funktion oder Methode übergeben wird und in der Funktion oder Methode gefüllt wird.
Wichtigkeit	Mittel

Beispiel:

Struktur ST_Sample:

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

Beispielfunktionen mit Rückgabewert:

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

SA0024: Nicht typisierte Literale

Funktion	Ermittelt nicht typisierte Literale, die Teil einer Operation sind.
Begründung	Nicht typisierte Literale werden je nach ihrer Verwendung automatisch typisiert. In einigen Fällen wie beispielsweise `nDWORD := ROL(DWORD#1, i);` kann dies zu unerwarteten Situationen führen, in denen es besser ist, eine eindeutige Klärung durch Verwendung eines typisierten Literals zu erreichen.
Wichtigkeit	Niedrig

Beispiel:

```
PROGRAM MAIN
VAR
    nINT    : INT := 10;         // no error as no part of operation
    nDINT   : DINT;
    nLINT   : LINT;
    fREAL   : REAL;
    fLREAL  : LREAL;
END_VAR
```

```
nINT   := nINT + 34;             // => SA0024
nINT   := nINT + INT#34;         // no error

nDINT  := nDINT + 23;            // => SA0024
nDINT  := nDINT + DINT#23;       // no error

nLINT  := nLINT + 124;           // => SA0024
fREAL  := fREAL + 1.1;           // => SA0024
fLREAL := fLREAL + 3.4;          // => SA0024
```

SA0025: Unqualifizierte Enumerationskonstanten

Funktion	Ermittelt Aufzählungskonstanten, die nicht mit einem qualifizierten Namen verwendet werden, d.h. ohne dass der Name der Enumeration vorangestellt ist.
Begründung	Qualifizierte Zugriffe machen den Code besser lesbar und besser wartbar. Ohne das Erzwingen qualifizierter Variablennamen könnte bei Erweiterung des Programms eine weitere Enumeration eingefügt werden, die eine gleichnamige Konstante wie eine bereits existierende Enumeration enthält (siehe im Beispiel unten: “eRed”). Dann käme es zu einem nicht-eindeutigen Zugriff in diesem Codestück. Wir empfehlen in jedem Fall nur Enumerationen zu verwenden, die das {attribute ‘qualified-only’} tragen.
Wichtigkeit	Mittel

Beispiel:

Enumeration E_Color:

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

Programm MAIN:

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

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

SA0026: Möglicherweise Abschneiden von Strings

Funktion	Ermittelt String-Zuweisungen und -Initialisierungen, die keine ausreichende String-Länge verwenden.
Begründung	Wenn Strings unterschiedlicher Länge zugewiesen werden, dann wird möglicherweise ein String abgeschnitten. Das Ergebnis ist dann nicht das erwartete.
Wichtigkeit	Mittel

Beispiele:

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

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

SA0027: Mehrfachverwendung des Namens

Funktion	Ermittelt die Mehrfachverwendung eines Namens/Bezeichners einer Variable oder eines Objekts (POU) innerhalb des Gültigkeitsbereichs eines Projekts. Die folgenden Fälle werden abgedeckt: • Der Name einer Enumerationskonstanten ist identisch mit dem Namen in einer anderen Enumeration innerhalb der Applikation oder in einer eingebundenen Bibliothek. • Der Name einer Variablen ist identisch mit dem Namen eines anderen Objekts in der Applikation oder in einer eingebundenen Bibliothek. • Der Name einer Variablen ist identisch mit dem Namen einer Enumerationskonstanten in einer Enumeration in der Applikation oder in einer eingebundenen Bibliothek. • Der Name eines Objekts ist identisch mit dem Namen eines anderen Objekts in der Applikation oder in einer eingebundenen Bibliothek.
Begründung	Gleiche Namen können beim Lesen des Codes verwirrend sein. Sie können zu Fehlern führen, wenn unbeabsichtigt auf das falsche Objekt zugegriffen wird. Definieren und befolgen Sie deshalb Namenskonventionen zur Vermeidung solcher Situationen.
Ausnahme	Enumerationen, die mit dem Attribut 'qualified_only' deklariert sind, sind von der SA0027-Prüfung ausgenommen, da auf ihre Elemente nur qualifiziert zugegriffen werden kann.
Wichtigkeit	Mittel

Beispiel:

Das folgende Beispiel erzeugt Fehler/Warnung SA0027, da die Bibliothek Tc2_Standard im Projekt eingebunden ist, welche den Funktionsbaustein TON zur Verfügung stellt.

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

SA0028: Überlappende Speicherbereiche

Funktion	Ermittelt die Stellen, durch die zwei oder mehr Variablen denselben Speicherplatz belegen.
Begründung	Wenn zwei Variablen auf dem gleichen Speicherplatz liegen, dann kann sich der Code sehr unerwartet verhalten. Dies ist in jedem Fall zu vermeiden. Wenn es unumgänglich ist, einen Wert in verschiedenen Interpretationen zu verwenden, zum Beispiel einmal als DINT und einmal als REAL, dann sollten Sie eine UNION definieren. Auch über einen Pointer können Sie auf einen Wert anders getypt zugreifen, ohne dass der Wert umgewandelt wird.
Wichtigkeit	Hoch

Beispiel:

In dem folgenden Beispiel verwenden beide Variablen Byte 21, d.h. die Speicherbereiche der Variablen überlappen.

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

SA0029: Notation in Implementierung und Deklaration unterschiedlich

Funktion	Ermittelt die Codestellen (in der Implementierung), an denen sich die Notation eines Bezeichners zur Notation in dessen Deklaration unterscheidet.
Begründung	Die Norm IEC 61131-3 definiert Bezeichner als nicht case-sensitiv. Das heißt, eine Variable die als “varx” deklariert wurde, kann im Code auch als “VaRx” verwendet werden. Dies ist jedoch verwirrend und irreführend und sollte daher vermieden werden.
Wichtigkeit	Mittel

Beispiele:

Funktion F_TEST:

```
FUNCTION F_TEST : BOOL
…
```

Programm MAIN:

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

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

SA0031: Nicht verwendete Signaturen

Funktion	Ermittelt Programme, Funktionsbausteine, Funktionen, Datentypen, Schnittstellen, Methoden, Eigenschaften, Aktionen etc., die innerhalb des kompilierten Programmcodes nicht aufgerufen werden.
Begründung	Nicht verwendete Objekte vergrößern das Projekt unnötig und können beim Lesen des Codes verwirren.
Wichtigkeit	Niedrig
PLCopen-Regel	CP2

SA0032: Nicht verwendete Aufzählungskonstante

Funktion	Ermittelt Enumerationskonstanten, die nicht im kompilierten Programmcode verwendet werden.
Begründung	Nicht verwendete Enumerationskonstanten vergrößern die Enumerationsdefinition unnötig und können beim Lesen des Programms verwirren.
Wichtigkeit	Niedrig
PLCopen-Regel	CP24

Beispiel:

Enumeration E_Sample:

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

Programm MAIN:

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

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

SA0033: Nicht verwendete Variablen

Funktion	Ermittelt Variablen, die deklariert sind, aber innerhalb des kompilierten Programmcodes nicht verwendet werden.
Begründung	Nicht verwendete Variablen machen ein Programm weniger gut lesbar und wartbar. Nicht verwendete Variablen belegen unnötig Speicher und kosten bei der Initialisierung unnötig Laufzeit.
Wichtigkeit	Mittel
PLCopen-Regel	CP22/CP24

SA0035: Nicht verwendete Eingabevariablen

Funktion	Ermittelt Eingangsvariablen, die innerhalb des jeweiligen Funktionsbausteins nicht zugewiesen werden.
Begründung	Nicht verwendete Variablen machen ein Programm weniger gut lesbar und wartbar. Nicht verwendete Variablen belegen unnötig Speicher und kosten bei der Initialisierung unnötig Laufzeit.
Wichtigkeit	Mittel
PLCopen-Regel	CP24

Beispiel:

Funktionsbaustein FB_Sample:

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

```
bOut1 := bIn1;
```

SA0036: Nicht verwendete Ausgabevariablen

Funktion	Ermittelt Ausgangsvariablen, die innerhalb des jeweiligen Funktionsbausteins nicht zugewiesen werden.
Begründung	Nicht verwendete Variablen machen ein Programm weniger gut lesbar und wartbar. Nicht verwendete Variablen belegen unnötig Speicher und kosten bei der Initialisierung unnötig Laufzeit.
Wichtigkeit	Mittel
PLCopen-Regel	CP24

Beispiel:

Funktionsbaustein FB_Sample:

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

```
bOut1 := bIn1;
```

SA0034: Enumerationsvariablen mit falscher Zuweisung

Funktion	Ermittelt Werte, die einer Enumerationsvariablen zugewiesen sind. Einer Enumerationsvariablen dürfen nur definierte Enumerationskonstanten zugewiesen werden.
Begründung	Eine Variable vom Typ einer Enumeration sollte auch nur die vorgesehenen Werte haben, anderfalls funktioniert Code, der diese Variable verwendet möglicherweise nicht richtig. Wir empfehlen, Enumerationen immer mit dem {attribute 'strict'} zu verwenden. Dann prüft bereits der Compiler die korrekte Verwendung der Enumerationskomponenten.
Wichtigkeit	Hoch

Beispiel:

Enumeration E_Color:

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

Programm MAIN:

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

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

SA0037: Schreibzugriff auf Eingabevariable

Funktion	Ermittelt Eingangsvariablen (VAR_INPUT), auf die innerhalb der POU schreibend zugegriffen wird.
Begründung	Nach der Norm IEC 61131-3 darf eine Eingabevariable nicht innerhalb eines Bausteins verändert werden. Ein solcher Zugriff ist außerdem eine Fehlerquelle und macht den Code schlecht wartbar. Es weist daraufhin, dass eine Variable als Eingang und gleichzeitig als Hilfsvariable verwendet wird. Eine solche Doppelverwendung sollte vermieden werden.
Wichtigkeit	Mittel

Beispiel:

Funktionsbaustein FB_Sample:

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

Methode FB_Sample.SampleMethod:

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

SA0038: Lesezugriff auf Ausgabevariable

Funktion	Ermittelt Ausgangsvariablen (VAR_OUTPUT), auf die innerhalb der POU lesend zugegriffen wird.
Begründung	Nach 61131-3 ist es verboten, einen Ausgang innerhalb eines Bausteins zu lesen. Es weist darauf hin, dass der Ausgang nicht nur als Ausgang sondern gleichzeitig als temporäre Variable für Zwischenergebnisse verwendet wird. Eine solche Doppelverwendung sollte vermieden werden.
Wichtigkeit	Niedrig

Beispiel:

Funktionsbaustein FB_Sample:

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

Methode FB_Sample.SampleMethod:

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

SA0040: Mögliche Division durch Null

Funktion	Ermittelt Codestellen, an denen möglicherweise durch Null dividiert wird.
Begründung	Eine Division durch 0 ist nicht erlaubt. Eine Variable durch die dividiert wird, sollte immer vorher auf 0 überprüft werden. Andernfalls kann es zu einer "Divide by Zero"-Exception zur Laufzeit kommen.
Wichtigkeit	Hoch

Beispiel:

```
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: Möglicherweise schleifeninvarianter Code

Funktion	Ermittelt Zuweisungen in (FOR-, WHILE-, REPEAT-) Schleifen, die bei jedem Schleifendurchlauf den gleichen Wert berechnen. Solche Codezeilen könnten außerhalb der Schleife eingefügt werden.
Begründung	Dies ist eine Performance-Warnung. Code, der in einer Schleife ausgeführt wird, aber in jedem Schleifendurchlauf das Gleiche tut, kann außerhalb der Schleife durchgeführt werden.
Wichtigkeit	Mittel

Beispiel:

Im folgenden Beispiel wird SA0041 als Fehler/Warnung ausgegeben, da die Variablen nTest1 und nTest2 in der Schleife nicht verwendet werden.

```
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: Verwendung unterschiedlicher Zugriffspfade

Funktion	Ermittelt die Verwendung unterschiedlicher Zugriffspfade für die gleiche Variable.
Begründung	Unterschiedlicher Zugriff auf das gleiche Element reduziert die Lesbarkeit und Wartbarkeit eines Programms. Wir empfehlen die konsequente Verwendung von {attribute 'qualified-only'} für Bibliotheken, globale Variablenlisten und Enumerationen. Dadurch wird der vollqualifizierte Zugriff erzwungen.
Wichtigkeit	Niedrig

Beispiele:

Im folgenden Beispiel wird SA0042 als Fehler/Warnung ausgegeben, da auf die globale Variable nGlobal einmal direkt und einmal über den GVL-Namensraum zugegriffen wird und da auf die Funktion CONCAT einmal direkt und einmal über den Bibliotheksnamensraum zugegriffen wird.

Globale Variablen:

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

Programm 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: Verwendung einer globalen Variablen in nur einer POU

Funktion	Ermittelt globale Variablen, die nur in einer einzigen POU verwendet werden.
Begründung	Eine globale Variable, die nur an einer Stelle verwendet wird, sollte auch an dieser einen Stelle deklariert sein.
Wichtigkeit	Mittel
PLCopen-Regel	CP26

Beispiel:

Die globale Variable nGlobal1 wird nur im Programm MAIN verwendet.

Globale Variablen:

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

Programm SubProgram:

```
nGlobal2 := 123;
```

Programm MAIN:

```
SubProgram();

nGlobal1 := nGlobal2;
```

SA0044: Deklarationen mit Schnittstellenreferenz

Funktion	Ermittelt Deklarationen mit REFERENCE TO <Schnittstelle> und Deklarationen von VAR_IN_OUT-Variablen mit dem Typ einer Schnittstelle (implizit über REFERENCE TO realisiert).
Begründung	Ein Schnittstellentyp ist immer implizit eine Referenz auf eine Instanz eines Funktionsbausteins, der diese Schnittstelle implementiert. Eine Referenz auf eine Schnittstelle ist demnach eine Referenz auf eine Referenz und kann zu sehr unerwünschtem Verhalten führen.
Wichtigkeit	Hoch

Beispiele:

I_Sample ist eine im Projekt definierte Schnittstelle.

Funktionsbaustein 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
```

Programm MAIN:

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

SA0019: Implizite Pointer-Konvertierungen

Funktion	Ermittelt implizit erzeugte Pointer-Datentyp-Konvertierungen.
Begründung	Pointer sind in TwinCAT nicht streng getypt und können einander beliebig zugewiesen werden. Dies wird häufig genutzt und deswegen auch nicht vom Compiler gemeldet. Allerdings kann es dadurch auch ungewollt zu unerwarteten Zugriffen kommen. Wenn ein POINTER TO BYTE einem POINTER TO DWORD zugewiesen wird, ist es möglich, dass über den letzten Pointer ungewollt Speicher überschrieben wird. Lassen Sie diese Regel deshalb in jedem Fall prüfen und unterdrücken Sie die Meldung nur in den Fällen, in denen Sie bewusst anders getypt auf einen Wert zugreifen wollen. Implizite Datentyp-Konvertierungen werden mit einer anderen Meldung gemeldet.
Ausnahme	BOOL ↔ BIT
Wichtigkeit	Hoch
PLCopen-Regel	CP25

Beispiele:

```
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: Implizite erweiternde Konvertierungen

Funktion	Ermittelt Codestellen, an denen bei arithmetischen Operationen implizit Konvertierungen von kleineren in größere Datentypen durchgeführt werden.

Begründung	Der Compiler erlaubt jegliche Zuweisungen von unterschiedlichen Typen, wenn der Wertebereich des Quelltyps vollständig im Wertebereich des Zieltyps enthalten ist. Allerdings baut der Compiler eine Konvertierung so spät wie möglich in den Code ein. Bei einer Zuweisung der folgenden Art:
	`nLINT := nDINT * nDINT;`
	führt der Compiler die implizite Konvertierung erst nach der Multiplikation durch:
	`nLINT := TO_LINT(nDINT * nDINT);`
	Ein Überlauf wird daher abgeschnitten. Wenn Sie das verhindern wollen, können Sie die Konvertierung bereits für die Elemente durchführen lassen:
	`nLINT := TO_LINT(nDINT) * TO_LINT(nDINT);`
	Es kann daher sinnvoll sein, sich Stellen melden zu lassen, an denen der Compiler implizite Konvertierungen einbaut, um zu prüfen, ob diese genau so gewollt sind. Außerdem können explizite Konvertierungen zu besserer Portierbarkeit auf andere Systeme dienen, wenn diese restriktivere Typprüfungen haben.

Ausnahme	BOOL ↔ BIT
Wichtigkeit	Niedrig

Beispiele:

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

```
nDINT  := UINT_TO_DINT(nUINT) * UINT_TO_DINT(nUINT);  // no error
nDINT  := nUINT * nUINT;                              // => SA0130

nLINT  := nDINT * nDINT;                              // => SA0130
nULINT := nUSINT * nUSINT;                            // => SA0130
nLWORD := nUDINT * nUDINT;                            // => SA0130
fLREAL := nBYTE * nBYTE;                              // => SA0130
```

SA0133: Explizite einschränkende Konvertierungen

Funktion	Ermittelt explizit durchgeführte Konvertierungen von einem größeren auf einen kleineren Datentyp.
Begründung	Eine große Zahl von Typkonvertierungen kann bedeuten, dass falsche Datentypen für Variablen gewählt wurden. Es gibt daher Programmierrichtlinien, die eine explizite Begründung für Datentypkonvertierungen fordern.
Wichtigkeit	Niedrig

Beispiele:

```
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: Explizite vorzeichenbehaftete/vorzeichenlose Konvertierungen

Funktion	Ermittelt explizit durchgeführte Konvertierungen von vorzeichenbehafteten auf vorzeichenlose Datentypen oder umgekehrt.
Begründung	Ein übermäßiger Gebrauch von Typkonvertierungen kann bedeuten, dass falsche Datentypen für Variablen gewählt wurden. Es gibt daher Programmierrichtlinien, die eine explizite Begründung für Datentypkonvertierungen fordern.
Wichtigkeit	Niedrig

Beispiele:

```
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: Ungültige Adressen und Datentypen

Funktion	Ermittelt ungültige Adress- und Datentypspezifikationen. Für Adressen sind die folgenden Größenpräfixe gültig. Abweichungen davon führen zu einer ungültigen Adressspezifikation. • X für BOOL • B für 1-Byte-Datentypen • W für 2-Byte-Datentypen • D für 4-Byte-Datentypen
Begründung	Variablen, die auf direkten Adressen liegen, sollten möglichst mit einer Adresse assoziiert werden, die ihrer Datentypbreite entspricht. Es kann für den Leser des Codes zur Verwirrung führen, wenn beispielsweise ein DWORD auf eine BYTE-Adresse gelegt wird.
Wichtigkeit	Niedrig

Mit den empfohlenen Platzhaltern %I* oder %Q* wird eine flexible und optimierte Adressierung von TwinCAT automatisch durchgeführt.

Beispiele:

```
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: Zugriff auf direkte Adressen

Funktion	Ermittelt direkte Adresszugriffe im Implementierungscode.
Begründung	Symbolische Programmierung ist immer zu bevorzugen: Eine Variable hat einen Namen, der auch eine Bedeutung tragen kann. Einer Adresse ist nicht ansehbar, wofür diese verwendet wird.
Wichtigkeit	Hoch
PLCopen-Regel	N1/CP1

Beispiele:

```
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-Deklarationen auf direkte Adressen

Funktion	Ermittelt AT-Deklarationen auf direkte Adressen.
Begründung	Die Verwendung von direkten Adressen im Code ist eine Fehlerquelle und führt zu schlechterer Lesbarkeit und Wartbarkeit des Codes. Daher wird die Verwendung der Platzhalter %I* oder %Q* empfohlen, bei denen TwinCAT eine flexible und optimierte Adressierung automatisch durchführt.
Wichtigkeit	Hoch
PLCopen-Regel	N1/CP1

Beispiele:

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

SA0051: Vergleichsoperationen auf BOOL-Variablen

Funktion	Ermittelt Vergleichsoperationen auf Variablen vom Typ BOOL.
Begründung	TwinCAT erlaubt solche Vergleiche, diese sind aber zumindest sehr unüblich und können verwirrend sein. Die Norm IEC-61131-3 sieht diese Vergleiche nicht vor, daher sollten Sie sie vermeiden.
Wichtigkeit	Mittel

Beispiel:

```
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: Unübliche Schiebeoperation

Funktion	Ermittelt Schiebeoperationen (Bit-Shift) auf vorzeichenbehaftete Variablen. Die Norm IEC 61131-3 erlaubt allerdings nur Schiebeoperationen auf Bitfelder. Sehen Sie hierzu auch die strikte Regel SA0147 [▶ 68].
Begründung	TwinCAT erlaubt Schiebeoperationen auf vorzeichenbehafteten Datentypen. Diese Operationen sind aber unüblich und können verwirrend sein. Die Norm IEC-61131-3 sieht solche Operationen nicht vor, daher sollten Sie sie vermeiden.
Ausnahme	Im Falle von Schiebeoperationen auf Bitfeld-Datentypen (Byte, DWORD, LWORD, WORD) wird kein Fehler SA0052 ausgegeben.
Wichtigkeit	Mittel

Beispiele:

```
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: Zu große bitweise Verschiebung

Funktion	Ermittelt bei bitweiser Verschiebung (Bitverschiebung/Bit-Shift) von Operanden, ob dessen Datentyp-Breite überschritten wurde.
Begründung	Wenn eine Verschiebeoperation über die Datentypbreite hinausgeht, wird eine Konstante 0 erzeugt. Wenn eine Rotationsverschiebung über die Datentypbreite hinausgeht, dann ist das schwer zu lesen und der Rotationswert sollte deswegen gekürzt werden.
Wichtigkeit	Hoch

Beispiele:

```
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: Vergleich von REAL/LREAL auf Gleichheit/Ungleichheit

Funktion	Ermittelt, wo die Vergleichsoperatoren = (Gleichzeit) und <> (Ungleichheit) Operanden vom Typ REAL oder LREAL vergleichen.

Begründung	REAL/LREAL-Werte werden als Gleitpunktzahlen nach dem Standard IEEE 754 implementiert. Dieser Standard bringt es mit sich, dass bestimmte scheinbar einfache Dezimalzahlen nicht exakt dargestellt werden können. Das hat zur Folge, dass es für dieselbe Dezimalzahl unterschiedliche Repräsentationen als LREAL geben kann.
	Beispiel:
	`fLREAL_11 := 1.1;`
	`fLREAL_33 := 3.3;`
	`fLREAL_a := fLREAL_11 + fLREAL_11;`
	`fLREAL_b := fLREAL_33 - fLREAL_11;`
	`bTest := fLREAL_a = fLREAL_b;`
	bTest wird in diesem Fall FALSE liefern, auch wenn die Variablen fLREAL_a und fLREAL_b beide den Monitoring-Wert “2.2” liefern. Das ist kein Fehler des Compilers, sondern eine Eigenschaft der Gleitpunkteinheiten aller üblichen Prozessoren. Vermeiden können Sie das, indem Sie einen Mindestwert angeben, um den sich die Werte unterscheiden dürfen:
	`bTest := ABS(fLREAL_a - fLREAL_b) < 0.1;`

Ausnahme	Ein Vergleich mit 0.0 wird nicht von dieser Analyse gemeldet. Für die 0 gibt es im Standard IEEE 754 eine exakte Darstellung und daher funktioniert der Vergleich üblicherweise wie erwartet. Für eine bessere Performance ist es daher sinnvoll, hier einen direkten Vergleich zuzulassen.
Wichtigkeit	Hoch
PLCopen-Regel	CP54

Beispiele:

```
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: Unnötige Vergleichsoperationen von vorzeichenlosen Operanden

Funktion	Ermittelt unnötige Vergleiche mit vorzeichenlosen Operanden. Ein vorzeichenloser Datentyp ist nie kleiner Null.
Begründung	Ein mit dieser Prüfung aufgedeckter Vergleich liefert ein konstantes Ergebnis und das deutet auf einen Fehler im Code hin.
Wichtigkeit	Hoch

Beispiele:

```
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: Konstante außerhalb des gültigen Bereichs

Funktion	Ermittelt Literale (Konstanten) außerhalb des für den Operator gültigen Bereichs.
Begründung	Die Meldung wird für Fälle ausgegeben, in denen eine Variable mit einer Konstanten verglichen wird, die außerhalb des Wertebereichs dieser Variablen liegt. Der Vergleich liefert dann konstant TRUE oder FALSE. Dies deutet auf einen Programmierfehler hin.
Wichtigkeit	Hoch

Beispiele:

```
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: Möglicher Verlust von Nachkommastellen

Funktion	Ermittelt Anweisungen mit möglichem Verlust von Dezimalstellen.

Begründung	Ein Codestück der folgenden Art:
	`nDINT := 1;`
	`fREAL := TO_REAL(nDINT / DINT#2);`
	kann zu einer Fehlinterpretation führen. Diese Codezeile kann zu der Annahme führen, die Division würde als REAL-Operation durchgeführt und das Ergebnis würde in diesem Fall REAL#0.5 sein. Dies ist jedoch nicht der Fall, die Operation wird als Integer-Operation durchgeführt, das Ergebnis wird auf REAL gecastet und fREAL erhält den Wert REAL#0. Um dies zu vermeiden, sollten Sie durch einen Cast dafür sorgen, dass die Operation als REAL-Operation durchgeführt wird:
	`fREAL := TO_REAL(nDINT) / REAL#2;`

Wichtigkeit	Mittel

Beispiele:

```
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: Operation auf Enumerationsvariablen

Funktion	Ermittelt Operationen auf Variablen vom Typ einer Enumeration. Zuweisungen sind erlaubt.
Begründung	Enumerationen sollten nicht als normale Integer-Werte verwendet werden. Alternativ kann ein Alias-Datentyp definiert oder ein Unterbereichstyp verwendet werden.
Ausnahme	Wenn eine Enumeration mit dem Attribut {attribute 'strict'} gekennzeichnet ist, dann meldet bereits der Compiler eine solche Operation. Wenn eine Enumeration mit Hilfe des Pragmaattributs {attribute 'flags'} als Flag deklariert ist, wird für Operationen mit AND, OR, NOT, XOR kein Fehler SA0058 ausgegeben.
Wichtigkeit	Mittel

Beispiel 1:

Enumeration E_Color:

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

Programm MAIN:

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

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

Beispiel 2:

Enumeration E_State mit Attribut 'flags':

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

Programm 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: Vergleichsoperationen, die immer TRUE oder FALSE liefern

Funktion	Ermittelt Vergleiche mit Literalen, deren Ergebnis immer TRUE oder FALSE ist und die bereits während der Kompilierung ausgewertet werden können.
Begründung	Eine Operation, die konstant TRUE oder FALSE liefert, ist ein Hinweis auf einen Programmierfehler.
Wichtigkeit	Hoch

Beispiele:

```
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: Null als ungültiger Operand

Funktion	Ermittelt Operationen, in denen ein Operand mit dem Wert 0 zu einer ungültigen oder unsinnigen Operation führt.
Begründung	Ein solcher Ausdruck kann auf einen Programmierfehler hindeuten. In jedem Fall kostet er unnötig Laufzeit.
Wichtigkeit	Mittel

Beispiele:

```
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: Unübliche Operation auf Pointer

Funktion	Ermittelt Operationen auf Variablen vom Typ POINTER TO, die nicht = (Gleichheit), <> (Ungleichheit), + (Addition) oder ADR sind.
Begründung	In TwinCAT ist Pointer-Arithmetik grundsätzlich erlaubt und kann auch sinnvoll eingesetzt werden. Als übliche Operation auf Pointer wird daher die Addition eines Pointers mit einem Integerwert eingestuft. Damit ist es möglich, mit Hilfe eines Pointers ein Array mit variabler Länge zu bearbeiten. Alle anderen (unüblichen) Operationen mit Pointer werden mit SA0061 gemeldet.
Wichtigkeit	Hoch
PLCopen-Regel	E2/E3

Beispiele:

```
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: Ausdruck ist konstant

Funktion	Ermittelt konstante Ausdrücke, die immer TRUE oder FALSE liefern, unabhängig von den Werten eventuell verwendeter Variablen.
Begründung	Ein solcher Ausdruck ist offensichtlich unnötig und kann auf einen Fehler hindeuten. In jedem Fall belastet ein solcher Ausdruck unnötig die Lesbarkeit und ggf. auch die Laufzeit.
Wichtigkeit	Mittel

Beispiele:

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

```
IF MAX(nVar,1) >= 1 THEN          // => SA0062
    ;
END_IF

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

IF (bVar1 = FALSE) THEN           // => SA0062
    ;
END_IF

IF NOT bVar1 THEN                 // => no error
    ;
END_IF

nVar := 0;
IF nVar <> 0 THEN                 // => SA0062
    ;
END_IF
```

SA0063: Möglicherweise nicht 16-bitkompatible Operationen

Funktion	Ermittelt 16-Bit-Operationen mit Zwischenergebnissen. Hintergrund: Auf 16-Bit-Systemen können 32-Bit-Zwischenergebnisse abgeschnitten werden.
Begründung	Diese Meldung soll in dem sehr seltenen Fall vor Problemen schützen, wenn Code geschrieben wird, der sowohl auf einem 16-Bit-Prozessor als auch auf einem 32-Bit-Prozessor laufen soll.
Wichtigkeit	Niedrig

Beispiel:

(nVar+10) kann 16 Bit überschreiten.

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

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

SA0064: Addition eines Pointers

Funktion	Ermittelt alle Additionen von Pointern.
Begründung	In TwinCAT ist eine Pointerarithmetik grundsätzlich erlaubt und kann sinnvoll eingesetzt werden. Diese stellt aber auch eine Fehlerquelle dar. Deswegen gibt es Programmiervorschriften, die eine Pointerarithmetik grundsätzlich verbieten. Eine solche Vorschrift kann mit diesem Test überprüft werden.
Wichtigkeit	Mittel

Beispiele:

```
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: Pointer-Addition passt nicht zur Basisgröße

Funktion	Ermittelt Pointeradditionen, bei denen der zu addierende Wert nicht zur Basis-Datengröße des Pointers passt. Fehlerfrei können nur Literale der Basisdatengröße und Vielfache davon addiert werden.

Begründung	In TwinCAT (im Gegensatz zu C und C++) wird bei einer Addition eines Pointers mit einem Integerwert nur dieser Integerwert als Anzahl der Bytes addiert, und nicht der Integerwert mit der Basisgröße multipliziert.
	`pINT := ADR(array_of_int[0]);`
	`pINT := pINT + 2 ; // in TwinCAT zeigt pINT anschließend auf array_of_int[1]`
	Dieser Code würde in C anders funktionieren:
	`short* pShort`
	`pShort = &(array_of_short[0])`
	`pShort = pShort + 2; // in C zeigt pShort anschließend auf array_of_short[2]`
	Daher sollte in TwinCAT immer ein Vielfaches der Basisgröße des Pointers zu einem Pointer addiert werden. Ansonsten zeigt der Pointer möglicherweise auf einen nicht-ausgerichteten (not aligned) Speicher, was (je nach Prozessor) beim Zugriff zu einer Alignment-Exception führen kann.

Wichtigkeit	Hoch

Beispiele:

```
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: Verwendung von Zwischenergebnissen

Funktion	Ermittelt Verwendungen von Zwischenergebnissen in Anweisungen mit einem Datentyp, der kleiner als die Registergröße ist. In diesem Fall führt der implizite Cast gegebenenfalls zu unerwünschten Ergebnissen.

Begründung	TwinCAT führt aus Performancegründen Operationen auf der Registerbreite des Prozessors aus. Zwischenergebnisse werden nicht abgeschnitten! Das kann zu Fehlinterpretationen führen, wie im folgenden Fall:
	`usintTest := 0;`
	`bError := usintTest - 1 <> 255;`
	In TwinCAT ist bError in diesem Fall TRUE, weil die Operation usintTest - 1 typischerweise als 32-Bit-Operation ausgeführt wird und das Ergebnis nicht auf die Größe von Byte gecastet wird. Im Register steht dann der Wert 16#ffffffff und dieser ist ungleich 255.Um dies zu umgehen müssen Sie das Zwischenergebnis explizit casten:
	`bError := TO_USINT(usintTest - 1) <> 255;`

Wichtigkeit	Niedrig

Wenn diese Meldung aktiviert ist, werden sehr viele eher unproblematische Stellen im Code gemeldet werden. Ein Problem kann zwar nur entstehen, wenn die Operation einen Überlauf oder Unterlauf im Datentyp produziert, die statische Analyse kann dies aber für die einzelnen Stellen nicht differenziert erkennen.

Wenn Sie an allen gemeldeten Stellen einen expliziten Typcast einbauen, dann wird der Code deutlich langsamer und unleserlicher!

Beispiel:

```
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: Ungültige Verwendung einer Zählervariablen

Funktion	Ermittelt Schreibzugriffe auf eine Zählervariablen innerhalb einer FOR-Schleife.
Begründung	Eine Manipulation der Zählervariablen in einer FOR-Schleife kann leicht zu einer Endlosschleife führen. Um die Ausführung der Schleife für bestimmte Werte der Zählervariablen zu unterbinden, arbeiten Sie mit CONTINUE oder einfach mit einem IF.
Wichtigkeit	Hoch
PLCopen-Regel	L12

Beispiel:

```
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: Verwendung einer nicht-temporären Zählervariablen

Funktion	Ermittelt die Verwendung von nicht-temporären Variablen in FOR-Schleifen.
Begründung	Dies ist eine Performance-Warnung. Eine Zählervariable wird in jedem Fall bei jedem Aufruf eines Programmierbausteins initialisiert. Sie können eine solche Variable als temporäre Variable (VAR_TEMP) anlegen, ein Zugriff darauf ist unter Umständen schneller und die Variable belegt keinen dauerhaften Speicherplatz.
Wichtigkeit	Mittel
PLCopen-Regel	CP21/L13

Beispiel:

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

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

SA0081: Obergrenze ist kein konstanter Wert

Funktion	Ermittelt FOR-Anweisungen, bei denen die Obergrenze nicht mit einem konstanten Wert definiert ist.
Begründung	Wenn die Obergrenze einer Schleife ein variabler Wert ist, dann lässt sich nicht mehr erkennen, wie oft eine Schleife ausgeführt wird. Dies kann zur Laufzeit zu gravierenden Problemen führen, im schlimmsten Fall zu einer Endlosschleife.
Wichtigkeit	Hoch

Beispiele:

```
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: Fehlendes ELSE

Funktion	Ermittelt CASE-Anweisungen ohne ELSE-Zweig.
Begründung	Defensive Programmierung fordert das Vorhandensein eines ELSE in jeder CASE-Anweisung. Wenn im ELSE-Fall nichts zu tun ist, dann sollten Sie dies durch einen Kommentar kennzeichnen. Dem Leser des Codes ist dann klar, dass der Fall nicht einfach vergessen wurde.
Ausnahme	Ein fehlender ELSE-Zweig wird nicht als fehlend berichtet, wenn in der CASE-Anweisung eine Enumeration verwendet wird, die mit dem Attribut 'strict' deklariert ist, und wenn in dieser CASE-Anweisung alle Enumerationskonstanten aufgeführt sind.
Wichtigkeit	Niedrig
PLCopen-Regel	L17

Beispiel:

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

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

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

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

SA0076: Fehlende Aufzählungskonstante

Funktion	Ermittelt, ob in CASE-Anweisungen jede Enumerationskonstante als Bedingung verwendet und in einem CASE-Zweig abgefragt wird.
Begründung	Defensive Programmierung erfordert die Bearbeitung aller möglichen Werte einer Enumeration. Wenn für einen bestimmten Enumerationswert keine Aktion nötig ist, dann sollten Sie dies explizit durch einen Kommentar kennzeichnen. Damit wird deutlich, dass der Wert nicht einfach vergessen wurde.
Wichtigkeit	Niedrig

Beispiel:

Im folgenden Beispiel wird der Enumerationswert eYellow nicht als CASE-Zweig behandelt.

Enumeration E_Color:

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

Programm 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: Datentypdiskrepanz bei CASE-Ausdruck

Funktion	Ermittelt Codepositionen, wo der Datentyp einer Bedingung nicht mit dem des CASE-Zweigs übereinstimmt.
Begründung	Wenn die Datentypen zwischen der CASE-Variable und dem CASE-Fall nicht übereinstimmen, dann könnte das auf einen Fehler hindeuten.
Wichtigkeit	Niedrig

Beispiel:

Enumeration E_Sample:

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

Programm 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: CASE-Anweisungen ohne CASE-Zweig

Funktion	Ermittelt CASE-Anweisungen ohne Fälle, d.h. mit nur einer ELSE-Anweisung.
Begründung	Eine CASE-Anweisung ohne Fälle kostet nur Zeit in der Ausführung und ist schwer zu lesen.
Wichtigkeit	Mittel

Beispiel:

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

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

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

SA0090: POUs sollen eine einzige Exit-Stelle haben

Funktion	Erkennt Codestellen, an denen die RETURN-Anweisung nicht die letzte Anweisung in einer Funktion, Methode, Eigenschaft oder einem Programm ist. Es werden auch Stellen erkannt, an denen ein RETURN innerhalb einer IF-Verzweigung steht.
Begründung	Ein RETURN im Code führt zu schlechterer Wartbarkeit, Testbarkeit und Lesbarkeit des Codes. Ein RETURN im Code wird leicht übersehen. Sie müssen Code, der auf alle Fälle beim Austritt einer Funktion ausgeführt werden sollte, vor jedem RETURN einfügen und das wird oft vergessen.
Wichtigkeit	Mittel
PLCopen-Regel	CP14

Beispiel:

```
FUNCTION F_TestFunction : DINT
VAR_INPUT
    bTest   : BOOL;
END_VAR
```

```
IF bTest THEN
    RETURN;                      // => SA0090
END_IF

F_TestFunction := 99;
```

SA0095: Zuweisung in Bedingung

Funktion	Ermittelt Zuweisungen in Bedingungen von IF-, CASE-, WHILE- oder REPEAT-Konstrukten.
Begründung	Ein Zuweisung (:=) und ein Vergleich (=) kann leicht verwechselt werden. Eine Zuweisung in einer Bedingung kann daher leicht unabsichtlich erfolgt sein und wird deswegen gemeldet. Auch der Leser des Codes kann dadurch verwirrt werden.
Wichtigkeit	Hoch

Beispiele:

```
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: Variablen größer als <n> Bytes

Funktion	Ermittelt Variablen, die mehr als n Bytes verwenden, wobei n durch die aktuelle Konfiguration vorgegeben ist. Den Parameter, der bei dieser Prüfung berücksichtigt wird, können Sie konfigurieren, indem Sie innerhalb der Regelkonfiguration auf die Zeile von Regel 100 doppelklicken (SPS-Projekteigenschaften > Kategorie "Static Analysis" > Registerkarte "Regeln" > Regel 100). In dem aufgehenden Dialog können Sie folgende Einstellungen vornehmen: • Obergrenze in Bytes (Standardwert: 1024)
Begründung	Manche Programmierrichtlinien legen eine maximale Größe für eine einzelne Variable fest. Dies kann hiermit überprüft werden.
Wichtigkeit	Niedrig

Beispiel:

Im folgenden Beispiel ist die Variable aSample größer als 1024 Bytes.

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

SA0101: Namen mit unzulässiger Länge

Funktion	Ermittelt Namen mit unzulässiger Länge. Der Name von Objekten muss eine definierte Länge haben. Die Parameter, die bei dieser Prüfung berücksichtigt werden, können Sie konfigurieren, indem Sie innerhalb der Regelkonfiguration auf die Zeile von Regel 101 doppelklicken (SPS-Projekteigenschaften > Kategorie "Static Analysis" > Registerkarte "Regeln" > Regel 101). In dem aufgehenden Dialog können Sie folgende Einstellungen vornehmen: • Minimale Anzahl an Zeichen (Standardwert: 5) • Maximale Anzahl an Zeichen (Standardwert: 30) • Ausnahmen
Begründung	In manchen Programmierrichtlinien wird eine Mindestlänge für Variablennamen festgelegt. Die Einhaltung kann mit dieser Analyse überprüft werden.
Wichtigkeit	Niedrig
PLCopen-Regel	N6

Beispiele:

Regel 101 ist mit folgenden Parametern konfiguriert:

- Minimale Anzahl an Zeichen: 5
- Maximale Anzahl an Zeichen: 30
- Ausnahmen: MAIN, i

Programm PRG1:

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

Programm 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: Zugriff von außen auf lokale Variablen

Funktion	Ermittelt lesende Zugriffe von außen auf lokale Variablen von Programmen oder Funktionsbausteinen.
Begründung	TwinCAT ermittelt Schreibzugriffe von außen auf lokale Variablen von Programmen oder Funktionsbausteinen als Kompilierfehler. Da Lesezugriffe auf lokale Variablen nicht vom Compiler abgefangen werden und dies mit dem Grundsatz der Datenkapselung (Verbergen von Daten) bricht und nicht der Norm IEC 61131-3 entspricht, kann diese Regel verwendet werden, um Lesezugriffe auf lokale Variablen zu ermitteln.
Wichtigkeit	Mittel

Beispiele:

Funktionsbaustein FB_Base:

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

Methode FB_Base.SampleMethod:

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

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

Funktionsbaustein FB_Sub:

```
FUNCTION_BLOCK FB_Sub EXTENDS FB_Base
```

Methode FB_Sub.SampleMethod:

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

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

Programm PRG_1:

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

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

Programm 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: Gleichzeitiger Zugriff auf nicht-atomare Daten

Funktion	Ermittelt nicht-atomare Variablen (zum Beispiel mit Datentyp STRING, WSTRING, ARRAY, STRUCT, FB-Instanzen, 64-Bit Datentypen), die in mehr als einer Task verwendet werden.
Begründung	Wenn keine Synchronisation beim Zugriff erfolgt, dann kann es bei gleichzeitigem Lesen in einer Task und Schreiben in einer anderen Task dazu kommen, dass inkonsistente Werte gelesen werden.
Ausnahme	In folgenden Fällen greift diese Regel nicht: • Wenn das Zielsystem eine FPU (Floating Point Unit) besitzt, wird der Zugriff mehrerer Tasks auf LREAL-Variablen nicht ermittelt und gemeldet. • Wenn das Zielsystem ein 64-Bit Prozessor ist bzw. "TwinCAT RT (x64)" als Solution-Plattform ausgewählt ist, greift die Regel nicht für 64-Bit Datentypen.
Wichtigkeit	Mittel

Sehen Sie auch die Regel SA0006 [▶ 24].

Beispiele:

Struktur ST_Sample:

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

Funktionsbaustein 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
```

Programm MAIN1, aufgerufen von der 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;
```

Programm MAIN2, aufgerufen von der 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: Mehrfache Instanzaufrufe

Funktion	Ermittelt und meldet Instanzen von Funktionsbausteinen, die mehr als einmal aufgerufen werden. Damit eine Fehlermeldung für eine mehrfach aufgerufene Instanz eines Funktionsbausteins generiert wird, muss im Deklarationsteil des Funktionsbausteins das Attribut {attribute 'analysis:report-multiple-instance-calls'} [▶ 129] hinzugefügt werden.
Begründung	Einige Funktionsbausteine sind so designt, dass sie nur einmal im Zyklus aufgerufen werden können. Dieser Test prüft, ob ein Aufruf an mehreren Stellen erfolgt.
Wichtigkeit	Niedrig
PLCopen-Regel	CP16/CP20

Beispiel:

Im folgenden Beispiel wird die Statische Analyse einen Fehler für fb2 ausgeben, weil die Instanz mehr als einmal aufgerufen wird und der Funktionsbaustein mit dem benötigten Attribut deklariert ist.

Funktionsbaustein FB_Test1 ohne Attribut:

```
FUNCTION_BLOCK FB_Test1
```

Funktionsbaustein FB_Test2 mit Attribut:

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

Programm MAIN:

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

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

SA0106: Virtuelle Methodenaufrufe in FB_init

Funktion	Ermittelt Methodenaufrufe in der Methode FB_init eines Basis-Funktionsbausteins, die von einem vom Basis-FB abgeleiteten Funktionsbaustein überschrieben werden.
Begründung	In solchen Fällen kann es sein, dass die Variablen in überschriebenen Methoden im Basis-FB nicht initialisiert sind.
Wichtigkeit	Hoch

Beispiel:

- Funktionsbaustein FB_Base hat die Methoden FB_init und MyInit. FB_init ruft MyInit zur Initialisierung auf.
- Funktionsbaustein FB_Sub ist von FB_Base abgeleitet.
- FB_Sub.MyInit überschreibt bzw. erweitert FB_Base.MyInit.
- MAIN instanziiert FB_Sub auf und verwendet dabei aufgrund der Aufrufreihenfolge während der Initialisierung die Instanzvariable nSub, bevor sie initialisiert wurde.

Funktionsbaustein FB_Base:

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

Methode FB_Base.FB_init:

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

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

Methode FB_Base.MyInit:

```
METHOD MyInit : DINT
```

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

Funktionsbaustein FB_Sub:

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

Methode FB_Sub.MyInit:

```
METHOD MyInit : DINT
```

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

Programm MAIN:

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

Die Instanz MAIN.fbBase besitzt nach der Initialisierung folgende Variablenwerte:

- nBase ist 123

Die Instanz MAIN.fbSub besitzt nach der Initialisierung folgende Variablenwerte:

- nBase ist 123
- nSub ist 0

Die Variable MAIN.fbSub.nSub ist nach der Initialisierung 0, da während der Initialisierung von fbSub die folgende Aufrufreihenfolge durchgeführt wird:

- Initialisierung des Basisbausteins:
  - implizite Initialisierung
  - explizite Initialisierung: FB_Base.FB_init
  - FB_Base.FB_init ruft FB_Sub.MyInit auf → **SA0106**

- FB_Sub.MyInit ruft FB_Base.MyInit auf (über SUPER-Zeiger)
- Initialisierung des abgeleiteten Bausteins:
  - implizite Initialisierung

SA0107: Fehlen von formalen Parametern

Funktion	Ermittelt, wo formale Parameter fehlen.
Begründung	Code wird lesbarer, wenn die formalen Parameter beim Aufruf angegeben werden.
Wichtigkeit	Niedrig

Beispiel:

Funktion F_Sample:

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

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

Programm 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: Zeigervariablen

Funktion	Ermittelt Variablen des Typs POINTER TO.
Begründung	Die Norm IEC 61131-3 erlaubt keine Pointer.
Wichtigkeit	Niedrig

Beispiel:

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

SA0112: Referenzvariablen

Funktion	Ermittelt Variablen des Typs REFERENCE TO.
Begründung	Die Norm IEC 61131-3 erlaubt keine Referenzen.
Wichtigkeit	Niedrig

Beispiel:

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

SA0113: Variablen mit Datentyp WSTRING

Funktion	Ermittelt Variablen des Typs WSTRING.
Begründung	Nicht alle Systeme unterstützen WSTRING. Der Code wird leichter portierbar, wenn auf WSTRING verzichtet wird.
Wichtigkeit	Niedrig

Beispiel:

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

SA0114: Variablen mit Datentyp LTIME

Funktion	Ermittelt Variablen des Typs LTIME.
Begründung	Nicht alle Systeme unterstützen LTIME. Der Code wird portierbarer, wenn auf LTIME verzichtet wird.
Wichtigkeit	Niedrig

Beispiel:

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

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

SA0115: Deklarationen mit Datentyp UNION

Funktion	Ermittelt Deklarationen eines UNION-Datentyps und Deklarationen von Variablen vom Typ einer UNION.
Begründung	Die Norm IEC-61131-3 kennt keine Unions. Der Code wird leichter portierbar, wenn auf Unions verzichtet wird.
Wichtigkeit	Niedrig

Beispiele:

Union U_Sample:

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

Programm MAIN:

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

SA0117: Variablen mit Datentyp BIT

Funktion	Ermittelt Deklarationen von Variablen des Typs BIT (möglich innerhalb von Struktur- und Funktionsbausteindefinitionen).
Begründung	Die IEC-61131-3 kennt keinen Datentyp BIT. Der Code wird leichter portierbar, wenn auf BIT verzichtet wird.
Wichtigkeit	Niedrig

Beispiele:

Struktur ST_Sample:

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

Funktionsbaustein FB_Sample:

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

SA0119: Objektorientierte Funktionalität

Funktion	Ermittelt die Verwendung objektorientierter Funktionalitäten, wie beispielsweise: • Funktionsbaustein-Deklarationen mit EXTENDS oder IMPLEMENTS • Eigenschaften- und Schnittstellendeklarationen • Verwendung des THIS- oder SUPER-Zeigers
Begründung	Nicht alle Systeme unterstützen Objektorientierte Programmierung. Der Code wird leichter portierbar, wenn auf Objektorientierung verzichtet wird.
Wichtigkeit	Niedrig

Beispiele:

Schnittstelle I_Sample:

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

Funktionsbaustein FB_Base:

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

Funktionsbaustein FB_Sub:

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

Methode FB_Sub.SampleMethod:

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

Get-Funktion der Eigenschaft FB_Sub.SampleProperty:

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

Set-Funktion der Eigenschaft FB_Sub.SampleProperty:

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

SA0120: Programmaufrufe

Funktion	Ermittelt Programmaufrufe.
Begründung	Nach der Norm IEC 61131-3 können Programme nur in der Taskkonfiguration aufgerufen werden. Der Code wird leichter portierbar, wenn auf Programmaufrufe an anderen Stellen verzichtet wird.
Wichtigkeit	Niedrig

Beispiel:

Programm SubProgram:

```
PROGRAM SubProgram
```

Programm MAIN:

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

SA0121: Fehlende VAR_EXTERNAL-Deklarationen

Funktion	Ermittelt die Verwendung einer globalen Variablen im Funktionsbaustein, ohne dass sie dort als VAR_EXTERNAL deklariert ist (erforderlich laut Norm).
Begründung	Nach der Norm IEC 61131-3 ist der Zugriff auf globale Variablen nur über einen expliziten Import durch eine VAR_EXTERNAL-Deklaration erlaubt.
Wichtigkeit	Niedrig
PLCopen-Regel	CP18

In TwinCAT 3 PLC ist es nicht notwendig, Variablen als extern zu deklarieren. Das Schlüsselwort existiert, um die Kompatibilität zu IEC 61131-3 zu wahren.

Beispiel:

Globale Variablen:

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

Programm Prog1:

```
PROGRAM Prog1
VAR
    nVar    : INT;
END_VAR
nVar := nGlobal;                 // => SA0121
```

Programm Prog2:

```
PROGRAM Prog2
VAR
    nVar    : INT;
END_VAR
VAR_EXTERNAL
    nGlobal : INT;
END_VAR
nVar := nGlobal;                 // no error
```

SA0122: Als Ausdruck definierter Arrayindex

Funktion	Ermittelt die Verwendung von Ausdrücken bei der Deklaration von Arraygrenzen.
Begründung	Nicht alle Systeme erlauben Ausdrücke als Arraygrenzen.
Wichtigkeit	Niedrig

Beispiel:

```
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: Verwendung von INI, ADR oder BITADR

Funktion	Ermittelt die Verwendung der (TwinCAT spezifischen) Operatoren INI, ADR, BITADR.
Begründung	TwinCAT-spezifische Operatoren verhindern die Portierbarkeit des Codes.
Wichtigkeit	Niedrig

Beispiel:

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

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

SA0147: Unübliche Schiebeoperation - strikt

Funktion	Ermittelt Bitshift-Operationen, die nicht auf Bitfeld-Datentypen (BYTE, WORD, DWORD, LWORD) erfolgen.
Begründung	Die Norm IEC 61131-3 erlaubt nur Bitzugriffe auf Bitfeld-Datentypen. Der TwinCAT 3 Compiler erlaubt jedoch auch Bitshift-Operationen mit nicht vorzeichenbehafteten Datentypen.
Wichtigkeit	Niedrig

Sehen Sie auch die nicht strikte Regel SA0052 [▶ 45].

Beispiele:

```
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: Unüblicher Bitzugriff - strikt

Funktion	Ermittelt Bitzugriffe, die nicht auf Bitfeld-Datentypen (BYTE, WORD, DWORD, LWORD) erfolgen.
Begründung	Die Norm IEC 61131-3 erlaubt nur Bitzugriffe auf Bitfeld-Datentypen. Der TwinCAT 3 Compiler erlaubt jedoch auch Bitzugriffe auf nicht vorzeichenbehaftete Datentypen.
Wichtigkeit	Niedrig

Sehen Sie auch die nicht strikte Regel SA0018 [▶ 30].

Beispiele:

```
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: Initialisierungen nicht mit Konstanten

Funktion	Ermittelt Initialisierungen, die nicht Konstanten zuweisen.
Begründung	Initialisierungen sollten möglichst konstant sein und sich nicht auf andere Variablen beziehen. Insbesondere sollten Sie Funktionsaufrufe in der Initialisierung vermeiden, weil es dadurch zu einem Zugriff auf nicht initialisierte Daten kommen kann.
Wichtigkeit	Mittel

Beispiele:

Funktion F_ReturnDWORD:

```
FUNCTION F_ReturnDWORD : DWORD
```

Programm 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: Dereferenzierungszugriff in Initialisierungen

Funktion	Ermittelt alle Codestellen, an denen dereferenzierte Pointer im Deklarationsteil von POUs verwendet werden.
Begründung	Pointer und Referenzen sollten nicht für Initialisierungen verwendet werden, weil es dadurch zur Laufzeit zu Zugriffsverletzungen ("Access Violations") kommen kann, wenn der Pointer nicht initialisiert worden ist.
Wichtigkeit	Mittel

Beispiele:

```
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
```

Überblick über die Regeln zum Thema „Dereferenzierung“

Pointer:

- Dereferenzierung von Pointern im Deklarationsteil => SA0124 [▶ 70]
- Mögliche Null-Pointer-Dereferenzierung im Implementierungsteil => SA0039 [▶ 71]

Referenzen:

- Verwendung von Referenzen im Deklarationsteil => SA0125 [▶ 70]
- Mögliche Verwendung nicht initialisierter Referenzen im Implementierungsteil => SA0145 [▶ 73]

Schnittstellen:

- Mögliche Verwendung nicht initialisierter Schnittstellen im Implementierungsteil => SA0046 [▶ 72]

SA0125: Referenzen in Initialisierungen

Funktion	Ermittelt alle Referenzvariablen, die zur Initialisierung im Deklarationsteil von POUs verwendet werden.
Begründung	Pointer und Referenzen sollten nicht für Initialisierungen verwendet werden, weil es dadurch zur Laufzeit zu Zugriffsverletzungen (“Access Violations”) kommen kann, wenn der Pointer nicht initialisiert worden ist.
Wichtigkeit	Mittel

Beispiele:

```
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
```

Überblick über die Regeln zum Thema „Dereferenzierung“

Pointer:

- Dereferenzierung von Pointern im Deklarationsteil => SA0124 [▶ 70]
- Mögliche Null-Pointer-Dereferenzierung im Implementierungsteil => SA0039 [▶ 71]

Referenzen:

- Verwendung von Referenzen im Deklarationsteil => SA0125 [▶ 70]
- Mögliche Verwendung nicht initialisierter Referenzen im Implementierungsteil => SA0145 [▶ 73]

Schnittstellen:

- Mögliche Verwendung nicht initialisierter Schnittstellen im Implementierungsteil => SA0046 [▶ 72]

SA0039: Mögliche Null-Pointer-Dereferenzierung

Funktion	Ermittelt Codestellen, an denen möglicherweise ein Null-Pointer dereferenziert wird.
Begründung	Ein Pointer sollte vor jeder Dereferenzierung daraufhin geprüft werden, ob er ungleich 0 ist. Ansonsten kann es zu Zugriffsverletzungen (“Access Violation”) zur Laufzeit kommen.
Wichtigkeit	Hoch

Beispiel 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
```

Beispiel 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
```

Überblick über die Regeln zum Thema „Dereferenzierung“

Pointer:

- Dereferenzierung von Pointern im Deklarationsteil => SA0124 [▶ 70]
- Mögliche Null-Pointer-Dereferenzierung im Implementierungsteil => SA0039 [▶ 71]

Referenzen:

- Verwendung von Referenzen im Deklarationsteil => SA0125 [▶ 70]
- Mögliche Verwendung nicht initialisierter Referenzen im Implementierungsteil => SA0145 [▶ 73]

Schnittstellen:

- Mögliche Verwendung nicht initialisierter Schnittstellen im Implementierungsteil => SA0046 [▶ 72]

SA0046: Mögliche Verwendung nicht initialisierter Schnittstellen

Funktion	Ermittelt die Verwendung von Schnittstellen, die vor der Verwendung möglicherweise nicht initialisiert wurden.
Begründung	Eine Interface-Referenz sollte vor ihrer Verwendung auf <> 0 geprüft werden, weil es sonst beim Zugriff zur Laufzeit zu einer "Access Violation" kommen kann.
Wichtigkeit	Hoch

Beispiele:

Schnittstelle I_Sample:

```
INTERFACE I_Sample
```

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

Funktionsbaustein FB_Sample:

```
FUNCTION_BLOCK FB_Sample IMPLEMENTS I_Sample
```

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

Programm 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
```

Überblick über die Regeln zum Thema „Dereferenzierung“

Pointer:

- Dereferenzierung von Pointern im Deklarationsteil => SA0124 [▶ 70]
- Mögliche Null-Pointer-Dereferenzierung im Implementierungsteil => SA0039 [▶ 71]

Referenzen:

- Verwendung von Referenzen im Deklarationsteil => SA0125 [▶ 70]
- Mögliche Verwendung nicht initialisierter Referenzen im Implementierungsteil => SA0145 [▶ 73]

Schnittstellen:

- Mögliche Verwendung nicht initialisierter Schnittstellen im Implementierungsteil => SA0046 [▶ 72]

SA0145: Mögliche Verwendung nicht initialisierter Referenzen

Funktion	Ermittelt alle verwendeten Referenzvariablen, die möglicherweise vor der Verwendung nicht initialisiert werden und nicht durch den Operator __ISVALIDREF überprüft wurden. Diese Regel wird im Implementierungsteil von POUs angewendet.
Begründung	Eine Referenz sollte vor dem Zugriff auf Gültigkeit geprüft werden, weil es sonst beim Zugriff zur Laufzeit zu einer Zugriffsverletzung ("Access Violation") kommen kann.
Wichtigkeit	Hoch

Beispiele:

```
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
```

Überblick über die Regeln zum Thema „Dereferenzierung“

Pointer:

- Dereferenzierung von Pointern im Deklarationsteil => SA0124 [▶ 70]
- Mögliche Null-Pointer-Dereferenzierung im Implementierungsteil => SA0039 [▶ 71]

Referenzen:

- Verwendung von Referenzen im Deklarationsteil => SA0125 [▶ 70]
- Mögliche Verwendung nicht initialisierter Referenzen im Implementierungsteil => SA0145 [▶ 73]

Schnittstellen:

- Mögliche Verwendung nicht initialisierter Schnittstellen im Implementierungsteil => SA0046 [▶ 72]

SA0140: Auskommentierte Anweisungen

Funktion	Ermittelt Anweisungen, die auskommentiert sind.
Begründung	Code wird oft zu Debugging-Zwecken auskommentiert. Wenn ein solcher Kommentar freigegeben wird, dann ist zu einem späteren Zeitpunkt nicht mehr klar, ob der Code gelöscht werden sollte oder ob er nur zu Debug-Zwecken auskommentiert wurde und versehentlich nicht mehr einkommentiert wurde.
Wichtigkeit	Hoch
PLCopen-Regel	C4

Beispiel:

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

SA0150: Verletzung von Unter- oder Obergrenzen der Metriken

Funktion	Ermittelt die Bausteine, die die aktivierten Metriken an der Unter- oder Obergrenze verletzen.
Begründung	Code, der bestimmte Metriken einhält, ist leichter lesbar, leichter wartbar und leichter zu testen.
Wichtigkeit	Hoch
PLCopen-Regel	CP9

Beispiel:

Die Metrik "Anzahl Aufrufe" ist in der Metriken-Konfiguration aktiviert und konfiguriert (SPS-Projekteigenschaften > Kategorie "Static Analysis" > Registerkarte "Metriken").

- Untergrenze: 0
- Obergrenze: 3
- Baustein Prog1 wird jedoch 5x aufgerufen

Beim Ausführen der Statischen Analyse wird die Verletzung von SA0150 als Fehler bzw. Warnung im Meldungsfenster ausgegeben.

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

SA0160: Rekursive Aufrufe

Funktion	Ermittelt rekursive Aufrufe von Programmen, Aktionen, Methoden und Eigenschaften. Ermittelt auch mögliche Rekursionen durch virtuelle Funktionsaufrufe und Schnittstellenaufrufe.
Begründung	Rekursionen führen zu nicht-deterministischem Verhalten und sind daher eine Fehlerquelle.
Wichtigkeit	Mittel
PLCopen-Regel	CP13

Beispiel 1:

Methode FB_Sample.SampleMethod1:

```
METHOD SampleMethod1
VAR_INPUT
END_VAR
```

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

Methode FB_Sample.SampleMethod2:

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

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

Programm MAIN:

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

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

Beispiel 2:

Bitte beachten Sie bei Eigenschaften:

Für eine Eigenschaft wird implizit eine lokale Eingangsvariable mit dem Namen der Eigenschaft erzeugt. Die folgende Set-Funktion einer Eigenschaft weist somit den Wert der impliziten lokalen Eingangsvariablen der Eigenschaft einer FB-Variablen zu.

Funktionsbaustein FB_Sample:

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

Set-Funktion der Eigenschaft SampleProperty:

```
nParameter := SampleProperty;
```

Bei der folgenden Set-Funktion wird somit die implizite Eingangsvariable der Eigenschaft sich selbst zugewiesen. Die Zuweisung einer Variablen auf sich selbst bedeutet keine Rekursion, sodass diese Set-Funktion keinen Fehler SA0160 erzeugt.

Set-Funktion der Eigenschaft SampleProperty:

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

Der Zugriff auf eine Eigenschaft über den THIS-Zeiger ist hingegen qualifiziert. Durch Verwendung des THIS-Zeigers wird auf die Instanz und somit auf die Eigenschaft und nicht auf die implizite lokale Eingangsvariable zugegriffen. Das bedeutet, dass die Verschattung von impliziter lokaler Eingangsvariablen und der Eigenschaft selbst aufgehoben wird. Bei der folgenden Set-Funktion wird daher ein erneuter Aufruf der Eigenschaft erzeugt, der zu einer Rekursion und damit zum Fehler SA0160 führt.

Set-Funktion der Eigenschaft SampleProperty:

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

SA0161: Ungepackte Struktur in gepackter Struktur

Funktion	Ermittelt ungepackte Strukturen, die in gepackten Strukturen verwendet werden.
Begründung	Eine ungepackte Struktur legt der Compiler normalerweise auf eine Adresse, die einen ausgerichteten (alignten) Zugriff auf alle Elemente innerhalb der Struktur erlaubt. Wenn Sie diese Struktur in einer gepackten Struktur anlegen, dann ist ein alignter Zugriff nicht mehr möglich, und ein Zugriff auf ein Element in der ungepackten Struktur kann zur Laufzeit zu einer “Misalignment Exception” führen.
Wichtigkeit	Hoch

Beispiel:

Die Struktur ST_SingleDataRecord ist gepackt, enthält jedoch Instanzen der ungepackten Strukturen ST_4Byte und ST_9Byte. Dies resultiert jeweils in einer Fehlermeldung SA0161.

```
{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
```

Struktur ST_9Byte:

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

Struktur ST_4Byte:

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

SA0162: Fehlende Kommentare

Funktion	Ermittelt Stellen im Programm, die nicht kommentiert sind. Kommentare sind erforderlich bei: • der Deklaration von Variablen. Die Kommentare stehen darüber oder rechts davon. • der Deklaration von POUs, DUTs, GVLs oder Schnittstellen. Die Kommentare stehen über der Deklaration (in der ersten Zeile).
Begründung	Eine vollständige Kommentierung wird von vielen Programmierrichtlinien gefordert und erhöht die Lesbarkeit und Wartbarkeit des Codes.
Wichtigkeit	Niedrig
PLCopen-Regel	C2

Beispiele:

Das folgende Beispiel erzeugt für die Variable b1 den Fehler: "SA0162: Missing comment for '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: Verschachtelte Kommentare

Funktion	Ermittelt Codestellen, an denen verschachtelte Kommentare sind.
Begründung	Verschachtelte Kommentare sind schwer zu lesen und sollten deswegen vermieden werden.
Wichtigkeit	Niedrig
PLCopen-Regel	C3

Beispiele:

Die im folgenden Beispiel entsprechend bezeichneten vier verschachtelten Kommentare führen jeweils zu dem Fehler: "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: Mehrzeilige Kommentare

Funktion	Ermittelt Codestellen, an denen der Mehrzeilenkommentar-Operator (* *) verwendet wird. Erlaubt sind nur die beiden Einzeilenkommentar-Operatoren // für Standardkommentare und /// für Dokumentationskommentare.
Begründung	Einige Programmierrichtlinien verbieten mehrzeilige Kommentare im Code, weil Anfang und Ende eines Kommentars aus dem Blickfeld geraten könnten und die schließende Kommentarklammer durch einen Fehler gelöscht werden könnte.
Wichtigkeit	Niedrig
PLCopen-Regel	C5

Sie können diese Prüfung mit dem Pragma {analysis …} [▶ 126] deaktivieren, auch für Kommentare im Deklarationsteil.

Beispiele:

```
(*
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: Maximale Anzahl an Eingabe-/Ausgabe-/VAR_IN_OUT-Variablen

Funktion	Die Prüfung ermittelt, ob eine definierte Anzahl an Eingabevariablen (VAR_INPUT), Ausgabevariablen (VAR_OUTPUT) oder VAR_IN_OUT-Variablen in einem Baustein überschritten wird. Die Parameter, die bei dieser Prüfung berücksichtigt werden, können Sie konfigurieren, indem Sie innerhalb der Regelkonfiguration auf die Zeile von Regel 166 doppelklicken (SPS-Projekteigenschaften > Kategorie "Static Analysis" > Registerkarte "Regeln" > Regel 166). In dem aufgehenden Dialog können Sie folgende Einstellungen vornehmen: • Maximale Anzahl an Eingängen (Standardwert: 10) • Maximale Anzahl an Ausgängen (Standardwert: 10) • Maximale Anzahl an Ein-/Ausgängen (Standardwert: 10)
Begründung	Es geht um die Überprüfung von individuellen Programmierrichtlinien. Viele Programmierrichtlinien sehen für Bausteine eine maximale Anzahl an Parametern vor. Zu viele Parameter machen den Code unleserlich und die Bausteine schwer zu testen.
Wichtigkeit	Mittel
PLCopen-Regel	CP23

Entsprechende Metriken verfügbar

Zur Berechnung der Eingabe- und Ausgabevariablen als Bestandteil der Metriken-Tabelle stehen die folgenden Metriken zur Verfügung:

Anzahl Eingabevariablen [▶ 102]

Anzahl Ausgabevariablen [▶ 102]

Beispiel:

Regel 166 ist mit folgenden Parametern konfiguriert:

- Maximale Anzahl an Eingängen: 0
- Maximale Anzahl an Ausgängen: 10
- Maximale Anzahl an Ein-/Ausgängen: 1

Für den folgenden Funktionsbaustein werden somit zwei Fehler SA0166 gemeldet, da zu viele Eingänge (> 0) und zu viele Ein-/Ausgänge (> 1) deklariert sind.

Funktionsbaustein 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: Temporäre Funktionsbausteininstanzen

Funktion	Ermittelt Funktionsbausteininstanzen, die als temporäre Variable deklariert sind. Dies betrifft Instanzen, die in einer Methode oder in einer Funktion oder als VAR_TEMP deklariert sind, und die deshalb in jedem Abarbeitungszyklus bzw. bei jedem Bausteinaufruf neu initialisiert werden.
Begründung	• Funktionsbausteine haben einen Zustand, der meist über mehrere SPS-Zyklen hinweg erhalten bleibt. Eine Instanz auf dem Stack existiert nur für die Dauer des Funktionsaufrufs. Es ist daher nur selten sinnvoll, eine Instanz als temporäre Variable anzulegen. • Zweitens sind Funktionsbausteininstanzen häufig groß und verbrauchen sehr viel Platz auf dem Stack (der auf Steuerungen meist begrenzt ist). • Drittens kann die Initialisierung und häufig auch die Terminierung eines Funktionsbausteins ziemlich viel Zeit in Anspruch nehmen.
Wichtigkeit	Mittel

Beispiele:

Methode FB_Sample.SampleMethod:

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

Funktion F_Sample:

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

Programm MAIN:

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

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

SA0168: Unnötige Zuweisungen

Funktion	Ermittelt Zuweisungen auf Variablen, die keine Auswirkungen im Code haben.
Begründung	Wenn einer Variablen mehrfach Werte zugewiesen werden, ohne dass die Variable zwischen den Zuweisungen ausgewertet wird, wirken sich die ersten Zuweisungen nicht auf das Programm aus.
Wichtigkeit	Niedrig

Beispiel:

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

```
nVar1 := 1;

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

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

SA0169: Ignorierte Ausgänge

Funktion	Ermittelt die Ausgänge von Methoden und Funktionen, die beim Aufruf der Methode oder Funktion nicht angegeben werden.
Begründung	Ignorierte Ausgänge können ein Hinweis auf nicht behandelte Fehler oder sinnlose Funktionsaufrufe sein, da Ergebnisse nicht verwendet werden.
Wichtigkeit	Mittel

Beispiel:

Funktion F_Sample:

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

Programm MAIN:

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

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

SA0170: Adresse einer Ausgabevariablen sollte nicht verwendet werden

Funktion	Ermittelt Codestellen, an denen die Adresse einer Ausgabevariablen (VAR_OUTPUT, VAR_IN_OUT) eines Funktionsbausteins verwendet wird.
Begründung	Es ist nicht erlaubt, in folgender Weise die Adresse eines Funktionsbausteinausgangs zu verwenden: • Via ADR-Operator • Via REF=
Ausnahme	Es wird kein Fehler gemeldet, wenn die Ausgabevariable innerhalb desselben Funktionsbausteins verwendet wird.
Wichtigkeit	Mittel

Beispiel:

Funktionsbaustein FB_Sample:

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

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

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

Zugriffe innerhalb eines anderen Bausteins, in diesem Fall im Programm MAIN:

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

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

SA0171: Enumerationen sollten das Attribut 'strict' haben

Funktion	Ermittelt Deklarationen von Enumerationen, die nicht mit dem Attribut {attribute 'strict'} versehen sind.
Begründung	Das Attribut {attribute 'strict'} bewirkt, dass Compilerfehler ausgegeben werden, wenn der Code gegen strikte Programmierregeln für Enumerationen verstößt. Standardmäßig wird beim Anlegen einer neuen Enumeration die Deklaration automatisch mit dem Attribut 'strict' versehen.
Wichtigkeit	Hoch

Für weitere Informationen siehe: PLC > Referenz Programmierung > Pragmas > Attribut-Pragmas > Attribut 'strict'

Beispiel:

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

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

SA0172: Möglicher Versuch eines Zugriffs außerhalb der Arraygrenzen

Funktion	Ermittelt mögliche Zugriffe auf einen Arrayindex außerhalb der Arraygrenzen.
Begründung	Häufig wird der Bereich des Arrayindex in FOR-Schleifen überschritten, bei denen die Indexvariable für den Zugriff auf einen Arrayindex verwendet wird.
Wichtigkeit	Hoch

Beispiel:

```
PROGRAM MAIN
VAR_TEMP
    nIndex       : INT;
END_VAR
VAR
    aSample      : ARRAY[0..10] OF INT;
END_VAR
```

```
FOR nIndex := INT#0 TO INT#50 DO
    aSample[nIndex] := 0;                       // => SA0172
END_FOR
```

SA0175: Verdächtige Operation auf Zeichenkette

Funktion	Ermittelt Codestellen, die bei einer UTF-8-Kodierung verdächtig sind.

Erfasste Konstrukte	1. Indexzugriff auf einen Single-Byte-String
	◦ Beispiel: `sVar[2]`
	◦ Meldung: Verdächtige Operation auf String: Indexzugriff '<expression>'
	2. Adresszugriff auf einen Single-Byte-String
	◦ Beispiel: `ADR(sVar)`
	◦ Meldung: Verdächtige Operation auf String: Möglicher Indexzugriff '<expression>'
	3. Aufruf einer String-Funktion der Tc2_Standard-Bibliothek außer CONCAT und LEN
	◦ Beispiel: `FIND(sVar, 'a');`
	◦ Meldung: Verdächtige Operation auf String: Möglicher Indexzugriff '<expression>'
	4. Einzelnes Byte-Literal, das Nicht-ASCII-Zeichen enthält
	◦ Beispiele:
	`sVar := '99€';`
	`sVar := 'Ä';`
	◦ Meldung: Verdächtige Operation auf String: Literal '<literal>' enthält Nicht-ASCII-Zeichen

Wichtigkeit	Mittel

Beispiele:

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

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

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

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

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

SA0178: Kognitive Komplexität

Funktion	Die Prüfung ermittelt, ob ein definiertes Limit der kognitiven Komplexität in einem Baustein überschritten wird. Den Parameter, der bei dieser Prüfung berücksichtigt wird, können Sie konfigurieren, indem Sie innerhalb der Regelkonfiguration auf die Zeile von Regel 178 doppelklicken (SPS-Projekteigenschaften > Kategorie "Static Analysis" > Registerkarte "Regeln" > Regel 178). In dem aufgehenden Dialog können Sie folgende Einstellungen vornehmen: • Komplexitätslimit (Standardwert: 20)
Begründung	Es geht um die Überprüfung von individuellen Programmierrichtlinien. Manche Programmierrichtlinien sehen für Bausteine einen maximalen Wert für die kognitive Komplexität vor. Eine zu hohe kognitive Komplexität macht den Code schwer lesbar und wartbar.
Wichtigkeit	Mittel

Entsprechende Metrik verfügbar

Zur Berechnung der kognitiven Komplexität als Bestandteil der Metriken-Tabelle steht die folgende Metrik zur Verfügung:

Kognitive Komplexität [▶ 105]

Beispiel:

Siehe Metrik: Kognitive Komplexität [▶ 105]

SA0179: Kopplung zwischen Objekten

Funktion	Die Prüfung ermittelt, ob ein definiertes Limit der Kopplung zwischen Objekten in einem Baustein überschritten wird. Den Parameter, der bei dieser Prüfung berücksichtigt wird, können Sie konfigurieren, indem Sie innerhalb der Regelkonfiguration auf die Zeile von Regel 179 doppelklicken (SPS-Projekteigenschaften > Kategorie "Static Analysis" > Registerkarte "Regeln" > Regel 179). In dem aufgehenden Dialog können Sie folgende Einstellungen vornehmen: • Kopplungslimit (Standardwert: 30)
Begründung	Es geht um die Überprüfung von individuellen Programmierrichtlinien. Manche Programmierrichtlinien sehen für Bausteine einen maximalen Wert für die Kopplung zwischen Objekten vor. Eine zu hohe Kopplung zwischen Objekten macht den Code schwer wartbar.
Wichtigkeit	Mittel
Synonym	CBO: **C**oupling **B**etween **O**bjects

Entsprechende Metrik verfügbar

Zur Berechnung der kognitiven Komplexität als Bestandteil der Metriken-Tabelle steht die folgende Metrik zur Verfügung:

Koppeln zwischen Objekten (CBO) [▶ 109]

Beispiel:

Siehe Metrik: Koppeln zwischen Objekten (CBO) [▶ 109]

SA0180: Indexbereich deckt nicht das gesamte Array ab

Funktion	Ermittelt Arrays mit nicht vollständig abgedecktem Indexbereich.
Begründung	Arrays werden oftmals in Schleifen behandelt, wobei der Schleifenindex das Array so indiziert, dass alle Komponenten des Arrays lückenlos angesprungen werden. Das ist dann gegeben, wenn der Schleifenindex und der Arrayindex in allen Dimensionen gleich sind. Wenn der Indexbereich das Array nicht vollständig abdeckt, weist das auf nicht behandelte Komponenten im Array hin.
Wichtigkeit	Mittel

Beispiel:

```
PROGRAM MAIN
VAR_TEMP
    nIndex       : INT;
END_VAR
VAR
    aSample      : ARRAY[0..10] OF INT;
END_VAR
```

```
FOR nIndex := INT#1 TO INT#10 DO
    aSample[nIndex] := 0;                    // => SA0180
END_FOR
```

4.3 Namenskonventionen

In der Registerkarte **Namenskonventionen** können Sie Namenskonventionen definieren, deren Einhaltung bei der Durchführung der Statischen Analyse [▶ 115] berücksichtigt wird. Dabei definieren Sie für die verschiedenen Datentypen von Variablen sowie für unterschiedliche Gültigkeitsbereiche, Bausteintypen und

Datentypdeklarationen obligatorische Präfixe. Die Namen aller Objekte, für die eine Konvention vorgegeben werden kann, werden in den Projekteigenschaften als Baumstruktur angezeigt. Die Objekte sind dabei unterhalb von organisatorischen Knoten angeordnet.

Zudem finden Sie in der Registerkarte **Namenskonventionen** Optionen [▶ 94], die die Konfiguration der Präfixe erweitern. Mithilfe dieser Optionen können Sie konfigurieren, wie sich das erwartete Gesamtpräfix für Variablen/Deklarationen zusammensetzen soll.

Konfiguration der Namenskonventionen

Namen	Knotenpunkte und Elemente, für die ein Präfix definiert werden kann Die Nummer in Klammern hinter jedem Element, zum Beispiel "PROGRAM (102)", ist die Präfix-Konventionsnummer, die bei einer Nichteinhaltung der Namenskonvention ausgegeben wird.
Präfix	Sie können die Namenskonventionen definieren, indem Sie in dieser Spalte das gewünschte Präfix eingeben. Bitte beachten Sie dabei die folgenden Hinweise und Möglichkeiten: • Mehrere mögliche Präfixe pro Zeile ◦ Mehrere Präfixe können Sie kommasepariert eingeben. ◦ Beispiel: "x, b" als Präfixe für Variablen vom Datentyp BOOL. Es darf sowohl "x" als auch "b" als Präfix für Boolesche Variablen verwendet werden. • Reguläre Ausdrücke ◦ Sie können für das Präfix auch reguläre Ausdrücke (RegEx) verwenden. Dazu müssen Sie ein @ voranstellen. ◦ Beispiel: "@b[a-dA-D]" als Präfix für Variablen vom Datentyp BOOL. Der Name der Booleschen Variablen muss mit einem "b" beginnen und darf dann noch ein Zeichen im Bereich von "a-dA-D" enthalten. • Datentyp-Platzhalter ◦ Für Variablen vom Datentyp Alias und für Eigenschaften können Sie den Datentyp-Platzhalter "{datatype}" als Präfix verwenden. ◦ Beispiel: Präfix für den Variablen-Datentyp Alias (33) = "{datatype}"
Präfixe für Variablen	Organisatorischer Knotenpunkt für alle Variablen, für die ein von ihrem Datentyp oder Gültigkeitsbereich abhängiges Präfix definiert werden kann
Präfixe für POUs	Organisatorischer Knotenpunkt für alle POU-Typen und Methodengültigkeitsbereiche, für die ein Präfix definiert werden kann
Präfixe für DUTs	Organisatorischer Knotenpunkt für die DUT-Datentypen Struktur, Enumeration, Alias oder Union, für die ein Präfix definiert werden kann
Präfixe für benutzerdefinierte Typen (NC0160)	Verfügbar ab TC3.1 Build 4026 Organisatorischer Knotenpunkt für spezielle benutzerdefinierte Typen, insbesondere solche aus Bibliotheken oder für schreibgeschützte Typen (z.B. PVOID, HRESULT) • Sie können die Liste mit Konventionen erweitern: Klicken Sie auf die Leerzeile darunter. Geben Sie dann den Namen eines benutzerdefinierten Typen ein oder wählen Sie einen benutzerdefinierten Typ im Dialog „Eingabehilfe“ aus. • Sie können eine Konvention löschen, indem Sie diese selektieren und die Taste [Entf] wählen. Hinweis: Diese Konventionen haben Vorrang vor den Präfixen, die mit dem Attribut {attribute 'nameprefix' := '<prefix>'} definiert sind. Beispiel: • In der Spalte „Name“ tragen Sie in einer Leerzeile unterhalb der Präfixe für benutzerdefinierte Typen den schreibgeschützten Systemdatentyp „PVOID“ ein. In der gleichen Zeile in der Spalte „Präfix“ tragen Sie den gewünschten Präfix ein, z.B. „p“. Variablen vom Typ PVOID werden bei Ausführung der Statischen Analyse auf dieses Präfix überprüft. • Weitere Beispiele für benutzerdefinierte Typen, deren gewünschten Präfix Sie an dieser Stelle konfigurieren können: ◦ Systemdatentyp HRESULT ◦ Funktionsbaustein TON aus der Bibliothek Tc2_System

Bildung des erwarteten Präfixes

Je nach Konfiguration der Optionen, die Sie im Dialog Optionen [▶ 94] finden, wird das Präfix gebildet, das für die verschiedenen Deklarationen erwartet wird.

Auf der Seite Optionen [▶ 94] finden Sie auch Erklärungen dazu, wie das erwartete Präfix gebildet wird, sowie einige Beispiele.

Platzhalter {datatype} bei Alias-Variablen und Eigenschaften

Bitte beachten Sie auch die Möglichkeiten des Platzhalters {datatype} [▶ 97], den Sie für die Präfixdefinition von Alias-Variablen und Eigenschaften verwenden können.

Lokale Präfixdefinition für strukturierte Typen

Für Variablen strukturierter Typen können Sie ein Präfix auch lokal in der Deklaration des Datentyps über das Attribut 'nameprefix' [▶ 128] vorschreiben.

Syntax von Konventionsverletzungen im Meldungsfenster

Jede Namenskonvention besitzt eine eindeutige Nummer (in der Konfigurationsansicht der Namenskonventionen in runden Klammern hinter der Konvention dargestellt). Wenn während der Statischen Analyse die Verletzung einer Konvention bzw. einer Vorgabe festgestellt wird, wird die Nummer zusammen mit einer Fehlerbeschreibung gemäß folgender Syntax in der Fehlerliste ausgegeben. Die Abkürzung "NC" weist dabei auf "Naming Convention" hin.

Syntax: **"NC<Präfix-Konventionsnummer>: <Konventionsbeschreibung>"**

Beispiel für Konventionsnummer 151 (DUTs vom Typ Struktur): "NC0151: Ungültiger Typname 'STR_Sample'. Erwartetes Präfix 'ST_'"

Temporäre Deaktivierung von Namenskonventionen

Es besteht die Möglichkeit, einzelne Konventionen vorübergehend, also für bestimmte Codezeilen, zu deaktivieren. Dazu können Sie ein Pragma oder ein Attribut im Deklarations- oder Implementierungsteil des Codes einfügen. Für Variablen strukturierter Typen können Sie ein Präfix auch lokal über ein Attribut in der Deklaration des Datentyps vorschreiben. Weiterführende Informationen hierzu finden Sie unter Pragmas und Attribute [▶ 125].

Übersicht der Namenskonventionen

Eine Übersicht der Namenskonventionen finden Sie unter Namenskonventionen – Übersicht und Beschreibung [▶ 86].

4.3.1 Namenskonventionen – Übersicht und Beschreibung

Übersicht

- **Präfixe für Variablen**
    - **Präfixe für Typen**
        - NC0003: BOOL [▶ 89]
        - NC0004: BIT [▶ 89]
        - NC0005: BYTE [▶ 89]
        - NC0006: WORD [▶ 89]
        - NC0007: DWORD [▶ 89]

- NC0008: LWORD [▶ 89]
- NC0013: SINT [▶ 89]
- NC0014: INT [▶ 89]
- NC0015: DINT [▶ 89]
- NC0016: LINT [▶ 89]
- NC0009: USINT [▶ 89]
- NC0010: UINT [▶ 89]
- NC0011: UDINT [▶ 89]
- NC0012: ULINT [▶ 89]
- NC0017: REAL [▶ 89]
- NC0018: LREAL [▶ 89]
- NC0019: STRING [▶ 89]
- NC0020: WSTRING [▶ 89]
- NC0021: TIME [▶ 89]
- NC0022: LTIME [▶ 89]
- NC0023: DATE [▶ 89]
- NC0024: DATE_AND_TIME [▶ 89]
- NC0025: TIME_OF_DAY [▶ 89]
- NC0026: POINTER [▶ 89]
- NC0027: REFERENCE [▶ 90]
- NC0028: SUBRANGE [▶ 90]
- NC0030: ARRAY [▶ 90]
- NC0031: Funktionsbausteininstanz [▶ 90]
- NC0036: Schnittstelle [▶ 91]
- NC0032: Struktur [▶ 91]
- NC0029: ENUM [▶ 91]
- NC0033: Alias [▶ 91]
- NC0034: Union [▶ 92]
- NC0035: __XWORD [▶ 89]
- NC0037: __UXINT [▶ 89]
- NC0038: __XINT [▶ 89]

- **Präfixe für Gültigkeitsbereiche**
  - NC0051: VAR_GLOBAL [▶ 92]

- NC0070: VAR_GLOBAL CONSTANT [▶ 92]
- NC0071: VAR_GLOBAL RETAIN [▶ 92]
- NC0072: VAR_GLOBAL PERSISTENT [▶ 92]
- NC0073: VAR_GLOBAL RETAIN PERSISTENT [▶ 92]
- **VAR**
  - NC0053: Programmvariablen [▶ 92]
  - NC0054: Funktionsbausteinvariablen [▶ 92]
  - NC0055: Funktions-/Methodenvariablen [▶ 92]
- NC0056: VAR_INPUT [▶ 92]
- NC0057: VAR_OUTPUT [▶ 92]
- NC0058: VAR_IN_OUT [▶ 92]
- NC0059: VAR_STAT [▶ 92]
- NC0061: VAR_TEMP [▶ 92]
- NC0062: VAR CONSTANT [▶ 92]
- NC0063: VAR PERSISTENT [▶ 92]
- NC0064: VAR RETAIN [▶ 92]
- NC0065: E-/A-Variablen [▶ 93]

- **Präfixe für POUs**
  - **Präfixe für POU-Typ**
    - NC0102: PROGRAM [▶ 93]
    - NC0103: FUNCTIONBLOCK [▶ 93]
    - NC0104: FUNCTION [▶ 93]
    - NC0105: METHOD [▶ 93]
    - NC0106: ACTION [▶ 93]
    - NC0107: PROPERTY [▶ 93]
    - NC0108: INTERFACE [▶ 93]
  - **Methoden-/Eigenschaftengültigkeitsbereich**
    - NC0121: PRIVATE [▶ 93]
    - NC0122: PROTECTED [▶ 93]
    - NC0123: INTERNAL [▶ 93]
    - NC0124: PUBLIC [▶ 93]
- **Präfixe für DUTs**
  - NC0151: Struktur [▶ 93]
  - NC0152: Aufzählung [▶ 93]

- NC0153: Union [▶ 93]

- NC0154: Alias [▶ 93]

- **Präfixe für benutzerdefinierte Typen**

- NC0160: Benutzerdefinierter Typ [▶ 94]

Detaillierte Beschreibung

Im Folgenden finden Sie Erläuterungen und Beispiele, bei welchen Deklarationen (d.h. an welcher Stelle im Projekt) die einzelnen Namenskonventionen verwendet werden. Bei den Beispielen handelt es sich um beispielhafte Deklarationen, bei denen das entsprechende Präfix erwartet werden würde, falls ein Präfix bei der zugehörigen Namenskonvention definiert wäre. Es soll deutlich werden, wo und wie ein Typ oder eine Variable deklariert sein kann, damit die Namenskonvention NC<xxxx> an dieser Stelle geprüft wird. Die Beispiele zeigen hingegen nicht, welches konkrete Präfix für die einzelnen Namenskonventionen definiert ist und somit bei den beispielhaften Deklarationen erwartet werden würde. Es gibt somit keine OK-/NOK-Gegenüberstellung.

Für konkrete Beispiele mit definiertem Präfix sehen Sie bitte die Seite Optionen [▶ 94].

Basisdatentypen:

NC0003: BOOL

Konfiguration eines Präfixes für eine Variablendeklaration vom Typ BOOL.

Beispieldeklarationen:

Bei folgenden Variablendeklarationen wird u.a. das für NC0003 konfigurierte Präfix zur Bildung des Gesamtpräfixes verwendet, dessen Einhaltung bei Durchführung der Statischen Analyse [▶ 115] überprüft wird.

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

Die Beschreibung von „NC0003: BOOL“ ist auf die anderen Basisdatentypen übertragbar:

- 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

Geschachtelte Datentypen:

NC0026: POINTER

Konfiguration eines Präfixes für eine Variablendeklaration vom Typ POINTER TO.

Beispieldeklaration:

Bei folgender Variablendeklaration wird u.a. das für NC0026 konfigurierte Präfix zur Bildung des Gesamtpräfixes verwendet, dessen Einhaltung bei Durchführung der Statischen Analyse [▶ 115] überprüft wird.

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

NC0027: REFERENCE

Konfiguration eines Präfixes für eine Variablendeklaration vom Typ REFERENCE TO.

Beispieldeklaration:

Bei folgender Variablendeklaration wird u.a. das für NC0027 konfigurierte Präfix zur Bildung des Gesamtpräfixes verwendet, dessen Einhaltung bei Durchführung der Statischen Analyse [▶ 115] überprüft wird.

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

NC0028: SUBRANGE

Konfiguration eines Präfixes für eine Variablendeklaration von einem Unterbereichstyp. Ein Unterbereichstyp ist ein Datentyp, dessen Wertebereich eine Untermenge eines Basistypen umfasst.

Mögliche Basisdatentypen für einen Unterbereichstyp: SINT, USINT, INT, UINT, DINT, UDINT, BYTE, WORD, DWORD, LINT, ULINT, LWORD.

Beispieldeklarationen:

Bei folgender Variablendeklaration wird u.a. das für NC0028 konfigurierte Präfix zur Bildung des Gesamtpräfixes verwendet, dessen Einhaltung bei Durchführung der Statischen Analyse [▶ 115] überprüft wird.

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

NC0030: ARRAY

Konfiguration eines Präfixes für eine Variablendeklaration vom Typ ARRAY[…] OF.

Beispieldeklaration:

Bei folgender Variablendeklaration wird u.a. das für NC0030 konfigurierte Präfix zur Bildung des Gesamtpräfixes verwendet, dessen Einhaltung bei Durchführung der Statischen Analyse [▶ 115] überprüft wird.

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

Instanz-basierte Datentypen:

NC0031: Funktionsbausteininstanz

Konfiguration eines Präfixes für eine Variablendeklaration vom Typ eines Funktionsbausteins.

Beispieldeklaration:

Deklaration eines Funktionsbausteins:

```
FUNCTION_BLOCK FB_Sample
…
```

Bei folgender Variablendeklaration wird u.a. das für NC0031 konfigurierte Präfix zur Bildung des Gesamtpräfixes verwendet, dessen Einhaltung bei Durchführung der Statischen Analyse [▶ 115] überprüft wird.

```
fbSample  : FB_Sample;
```

NC0036: Schnittstelle

Konfiguration eines Präfixes für eine Variablendeklaration vom Typ einer Schnittstelle.

Beispieldeklaration:

Deklaration einer Schnittstelle:

```
INTERFACE I_Sample
```

Bei folgender Variablendeklaration wird u.a. das für NC0036 konfigurierte Präfix zur Bildung des Gesamtpräfixes verwendet, dessen Einhaltung bei Durchführung der Statischen Analyse [▶ 115] überprüft wird.

```
iSample  : I_Sample;
```

NC0032: Struktur

Konfiguration eines Präfixes für eine Variablendeklaration vom Typ einer Struktur.

Beispieldeklaration:

Deklaration einer Struktur:

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

Bei folgender Variablendeklaration wird u.a. das für NC0032 konfigurierte Präfix zur Bildung des Gesamtpräfixes verwendet, dessen Einhaltung bei Durchführung der Statischen Analyse [▶ 115] überprüft wird.

```
stSample  : ST_Sample;
```

NC0029: ENUM

Konfiguration eines Präfixes für eine Variablendeklaration vom Typ einer Enumeration.

Beispieldeklaration:

Deklaration einer Enumeration:

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

Bei folgender Variablendeklaration wird u.a. das für NC0029 konfigurierte Präfix zur Bildung des Gesamtpräfixes verwendet, dessen Einhaltung bei Durchführung der Statischen Analyse [▶ 115] überprüft wird.

```
eSample  : E_Sample;
```

NC0033: Alias

Konfiguration eines Präfixes für eine Variablendeklaration vom Typ eines Alias.

Beispieldeklaration:

Deklaration eines Alias:

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

Bei folgender Variablendeklaration wird u.a. das für NC0033 konfigurierte Präfix zur Bildung des Gesamtpräfixes verwendet, dessen Einhaltung bei Durchführung der Statischen Analyse [▶ 115] überprüft wird.

```
tMessage  : T_Message;
```

NC0034: Union

Konfiguration eines Präfixes für eine Variablendeklaration vom Typ einer Union.

Beispieldeklaration:

Deklaration einer Union:

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

Bei folgender Variablendeklaration wird u.a. das für NC0034 konfigurierte Präfix zur Bildung des Gesamtpräfixes verwendet, dessen Einhaltung bei Durchführung der Statischen Analyse [▶ 115] überprüft wird.

```
uSample   : U_Sample;
```

Gültigkeitsbereiche von Variablendeklarationen:

NC0051: VAR_GLOBAL

Konfiguration eines Präfixes für eine Variablendeklaration zwischen den Schlüsselwörtern VAR_GLOBAL und END_VAR.

Beispieldeklaration:

Bei folgender Deklaration einer globalen Variablen wird u.a. das für NC0051 konfigurierte Präfix zur Bildung des Gesamtpräfixes verwendet, dessen Einhaltung bei Durchführung der Statischen Analyse [▶ 115] überprüft wird.

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

Die Beschreibung von „NC0051: VAR_GLOBAL“ ist auf die anderen Gültigkeitsbereiche von Variablendeklarationen übertragbar:

- NC0070: VAR_GLOBAL CONSTANT
- NC0071: VAR_GLOBAL RETAIN
- NC0072: VAR_GLOBAL PERSISTENT
- NC0073: VAR_GLOBAL RETAIN PERSISTENT
- NC0053: Programmvariablen (VAR innerhalb eines Programms)
- NC0054: Funktionsbausteinvariablen (VAR innerhalb eines Funktionsbausteins)
- NC0055: Funktions-/Methodenvariablen (VAR innerhalb einer Funktion/Methode)
- 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: E-/A-Variablen

Konfiguration eines Präfixes für eine Variablendeklaration mit AT-Deklaration.

Beispieldeklarationen:

Bei folgenden Variablendeklarationen mit AT-Deklaration wird u.a. das für NC0065 konfigurierte Präfix zur Bildung des Gesamtpräfixes verwendet, dessen Einhaltung bei Durchführung der Statischen Analyse [▶ 115] überprüft wird.

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

POU-Typen:

NC0102: PROGRAM

Konfiguration eines Präfixes für die Deklaration eines Programms (Name des Programms im Projektbaum).

Die Beschreibung von „NC0102: PROGRAM“ ist auf die anderen POU-Typen übertragbar:

- NC0103: FUNCTIONBLOCK

- NC0104: FUNCTION

- NC0105: METHOD

- NC0106: ACTION

- NC0107: PROPERTY

- NC0108: INTERFACE

Gültigkeitsbereiche von Methoden und Eigenschaften:

NC0121: PRIVATE

Konfiguration eines Präfixes für die Deklaration einer Methode oder einer Eigenschaft (Name der Methode/ Eigenschaft im Projektbaum), deren Zugriffsmodifizierer PRIVATE ist.

Die Beschreibung von „NC121: PRIVATE“ ist auf die anderen Gültigkeitsbereiche von Methoden und Eigenschaften übertragbar:

- NC0122: PROTECTED

- NC0123: INTERNAL

- NC0124: PUBLIC

DUTs:

NC0151: Struktur

Konfiguration eines Präfixes für die Deklaration einer Struktur (Name der Struktur im Projektbaum).

Die Beschreibung von „NC0151: Struktur“ ist auf die anderen DUT-Typen übertragbar:

- NC0152: Aufzählung

- NC0153: Union

- NC0154: Alias

Benutzerdefinierte Typen:

NC0160: Benutzerdefinierter Typ

Konfiguration eines Präfixes für einen benutzerdefinierten Typen, z.B. für Variablen vom Typ PVOID oder für Instanzen vom Bibliotheksbaustein Tc2_System.TON.

Weitere Informationen zu den Eingabemöglichkeiten in diesem Bereich finden Sie unter Namenskonventionen [▶ 83].

4.3.2 Optionen

In der Registerkarte **Namenskonventionen** finden Sie Optionen, die die Konfiguration der Präfixe erweitern. Mithilfe dieser Optionen können Sie konfigurieren, wie sich das erwartete Gesamtpräfix für Variablen/ Deklarationen zusammensetzen soll.

1) Erstes Zeichen nach Präfix soll ein Großbuchstabe sein

- Wenn aktiviert: Die statische Codeanalyse meldet einen Fehler für eine Variable, wenn das erste Zeichen des Variablennamens nach dem definierten Präfix kein Großbuchstabe ist.
- Wenn deaktiviert: Es findet keine Prüfung der Groß-/Kleinschreibung statt.
- Standardeinstellung: deaktiviert

Beispiele:

- Variable "bvar" mit dem erwarteten Präfix "b"
- Funktionsbaustein "FB_sample" mit dem erwarteten Präfix "FB_"

Option	Zustand	Ergebnis der statischen Analyse
Erstes Zeichen nach Präfix soll ein Großbuchstabe sein	Aktiviert	Für die oben genannten Definitionen wird jeweils ein Fehler gemeldet, dass der erste Buchstabe nach dem Präfix groß sein müsse. Korrekte Bezeichner wären "bVar" und "FB_Sample".
	Deaktiviert	Die Bezeichner "bvar" und "FB_sample" sind zulässig. Es wird kein Fehler bzgl. der Groß-/Kleinschreibung ausgegeben.

2) Rekursive Präfixe für kombinierbare Datentypen

- Wenn aktiviert: Variablen von kombinierbaren Datentypen (POINTER, REFERENCE, ARRAY, SUBRANGE) müssen ein zusammengesetztes **Datentyppräfix** haben. Das zusammengesetzte Präfix wird aus den Einzelpräfixen gebildet, die für die einzelnen Bestandteile des kombinierten Datentyps konfiguriert sind.
- Wenn deaktiviert: Es wird nur das Präfix des äußersten Datentyps als **Datentyppräfix** erwartet.
- Standardeinstellung: aktiviert
- Beispiele: siehe unten

3) Namensraumpräfix mit Datentyppräfix kombinieren

(Namensraum = Gültigkeitsbereich = Scope)

- Wenn aktiviert: Eine Variable muss das in den Namenskonventionen definierte **Präfix für ihren Namensraum** gefolgt von ihrem **Datentyppräfix** haben.
- Wenn deaktiviert: Das erwartete Gesamtpräfix hängt davon ab, ob für eine Variable ein Namensraumpräfix definiert ist oder nicht.
  - Falls für eine Variable das zugehörige **Namensraumpräfix definiert** ist, muss die Variable **nur** das in den Namenskonventionen definierte **Präfix für ihren Namensraum** haben. Das **Datentyppräfix** wird im Anschluss an das Namensraumpräfix **nicht** erwartet.
  - Falls für eine Variable das zugehörige **Namensraumpräfix nicht definiert** ist, muss die Variable **nur** das für sie definierte **Datentyppräfix** haben.

- Standardeinstellung: aktiviert
- Beispiele: siehe unten

Beispiele

- Präfixkonfiguration für Datentypen:
  - POINTER (26) = "p"
  - ARRAY (30) = "a"
  - INT (14) = "n"
  - BOOL (3) = "b"
- Präfixkonfiguration für Namensraum/Gültigkeitsbereich/Scope
  - Fall 1: Funktionsbausteinvariablen (54) = "_local_"
  - Fall 2: Funktionsbausteinvariablen (54) = leeres Feld/nicht konfiguriert
  - **INFO**: Weitere Beispiele für einen Namensraum/Gültigkeitsbereich/Scope sind u.a. VAR_GLOBAL (51), VAR_INPUT (56) und VAR CONSTANT (62).
- Deklaration:

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

Optionsszenario 1:

Option	Zustand	Erwartetes Gesamtpräfix für Fall 1 (NC0054 = "_local_")	Erwartetes Gesamtpräfix für Fall 2 (NC0054 = leer)
Rekursive Präfixe für kombinierbare Datentypen	Aktiviert	Für var1: '_local_pan' Für var2: '_local_aab'	Für var1: 'pan' Für var2: 'aab'
Namensraumpräfix mit Datentyppräfix kombinieren	Aktiviert		

Erklärung:

- Da die Option „Rekursive Präfixe für kombinierbare Datentypen“ aktiviert ist, wird als **Datentyppräfix** das zusammengesetzte Präfix aus den Einzelpräfixen erwartet. Folglich werden die Teilpräfixe "p" für POINTER, "a" für ARRAY und "n" für INT zum Datentyppräfix "pan" bzw. die Teilpräfixe "a" für ARRAY, nochmal "a" für ARRAY und "b" für BOOL zum Datentyppräfix "aab" zusammengesetzt.
- Da die Option „Namensraumpräfix mit Datentyppräfix kombinieren“ ebenfalls aktiviert ist, wird als **Gesamtpräfix** für Variablen die Kombination aus **Namensraumpräfix** und **Datentyppräfix** erwartet.
  - Fall 1: _local_ + pan = _local_pan
  - Fall 2: <leer> + pan = pan

Optionsszenario 2:

Option	Zustand	Erwartetes Gesamtpräfix für Fall 1 (NC0054 = "_local_")	Erwartetes Gesamtpräfix für Fall 2 (NC0054 = leer)
Rekursive Präfixe für kombinierbare Datentypen	Deaktiviert	Für var1: '_local_p' Für var2: '_local_a'	Für var1: 'p' Für var2: 'a'
Namensraumpräfix mit Datentyppräfix kombinieren	Aktiviert		

Erklärung:

- Da die Option „Rekursive Präfixe für kombinierbare Datentypen“ deaktiviert ist, wird als **Datentyppräfix** nur das Präfix des äußersten Datentyps erwartet. Das erwartete Datentyppräfix ist somit "p" bzw. "a".

- Da die Option „Namensraumpräfix mit Datentyppräfix kombinieren“ aktiviert ist, wird als **Gesamtpräfix** für Variablen die Kombination aus **Namensraumpräfix** und **Datentyppräfix** erwartet.
  - Fall 1: _local_ + p = _local_p
  - Fall 2: <leer> + p = p

Optionsszenario 3:

Option	Zustand	Erwartetes Gesamtpräfix für Fall 1 (NC0054 = "_local_")	Erwartetes Gesamtpräfix für Fall 2 (NC0054 = leer)
Rekursive Präfixe für kombinierbare Datentypen	Aktiviert	Für var1: '_local_' Für var2: '_local_'	Für var1: 'pan' Für var2: 'aab'
Namensraumpräfix mit Datentyppräfix kombinieren	Deaktiviert		

Erklärung:

- Siehe Optionsszenario 1: Da die Option „Rekursive Präfixe für kombinierbare Datentypen“ aktiviert ist, wird als **Datentyppräfix** das zusammengesetzte Präfix aus den Einzelpräfixen erwartet. Dies ergibt "pan" bzw. "aab" als Datentyppräfix.
- Da die Option „Namensraumpräfix mit Datentyppräfix kombinieren“ deaktiviert ist, hängt das erwartete **Gesamtpräfix** davon ab, ob für eine Variable ein Namensraumpräfix definiert ist oder nicht.
  - Falls **Namensraumpräfix definiert** (Fall 1): Die Variable muss **nur** das **Namensraumpräfix** haben. Das Datentyppräfix wird im Anschluss an das Namensraumpräfix nicht erwartet. Dies ergibt für beide Variablen "_local_" als erwartetes Gesamtpräfix.
  - Falls **Namensraumpräfix nicht definiert** (Fall 2): Die Variable muss nur das Datentyppräfix haben. Dies ergibt "pan" bzw. "aab" als erwartetes Gesamtpräfix.

Optionsszenario 4:

Option	Zustand	Erwartetes Gesamtpräfix für Fall 1 (NC0054 = "_local_")	Erwartetes Gesamtpräfix für Fall 2 (NC0054 = leer)
Rekursive Präfixe für kombinierbare Datentypen	Deaktiviert	Für var1: '_local_' Für var2: '_local_'	Für var1: 'p' Für var2: 'a'
Namensraumpräfix mit Datentyppräfix kombinieren	Deaktiviert		

Erklärung:

- Siehe Optionsszenario 2: Da die Option „Rekursive Präfixe für kombinierbare Datentypen“ deaktiviert ist, wird als **Datentyppräfix** nur das Präfix des äußersten Datentyps erwartet. Dies ergibt "p" bzw. "a" als Datentyppräfix.
- Da die Option „Namensraumpräfix mit Datentyppräfix kombinieren“ deaktiviert ist, hängt das erwartete **Gesamtpräfix** davon ab, ob für eine Variable ein Namensraumpräfix definiert ist oder nicht.
  - Falls **Namensraumpräfix definiert** (Fall 1): Die Variable muss **nur** das **Namensraumpräfix** haben. Das Datentyppräfix wird im Anschluss an das Namensraumpräfix nicht erwartet. Dies ergibt für beide Variablen "_local_" als erwartetes Gesamtpräfix.
  - Falls **Namensraumpräfix nicht definiert** (Fall 2): Die Variable muss nur das Datentyppräfix haben. Dies ergibt "p" bzw. "a" als erwartetes Gesamtpräfix.

Weitere Hinweise/Beispiele:

Für POUs mit einem Zugriffsmodifizierer (also Methoden oder Eigenschaften) wird als **Gesamtpräfix** die Kombination aus dem **Präfix für den Gültigkeitsbereich** (NC0121-NC0124: PRIVATE/PROTECTED/INTERNAL/PUBLIC) und dem **Präfix für den POU-Typ** (NC0105 bei Methode, NC0107 bei Eigenschaft) erwartet. Beispiele:

- Wenn für PRIVATE (121) das Präfix "priv_" und für METHOD (105) das Präfix "M_" konfiguriert wurde, dann wird für eine PRIVATE-Methode das **Gesamtpräfix** "priv_M_" erwartet.
- Wenn für METHOD (105) weiterhin das Präfix "M_" konfiguriert ist, aber für PRIVATE (121) kein Präfix konfiguriert wurde, d.h. wenn das Feld in den Namenskonventionen leer ist, dann wird für eine PRIVATE-Methode das **Gesamtpräfix** "M_" erwartet.

4.3.3 Platzhalter {datatype}

Für Variablen vom Typ Alias und für Eigenschaften kann in der Registerkarte „Namenskonventionen“ der Platzhalter "{datatype}" als Präfix verwendet werden. Dabei wird der Platzhalter {datatype} durch das Präfix ersetzt, das für den Datentyp des Alias bzw. für den Datentyp der Eigenschaft definiert ist. Somit meldet das Static Analysis Fehler für alle Alias-Variablen, die nicht das Präfix für den Datentyp des Alias besitzen, bzw. für alle Eigenschaften, die nicht das Präfix für den Datentyp der Eigenschaft besitzen.

Der Platzhalter "{datatype}" kann bei der Präfixdefinition auch mit weiteren Präfixen kombiniert werden, z.B. zu "P_{datatype}_".

Beispiel 1 für eine Alias-Variable:

- Im Projekt gibt es ein Alias "TYPE MyMessageType : STRING; END_TYPE" sowie eine Variable von diesem Typ (var : MyMessageType;).
- Präfixdefinitionen
  - Präfix für den Variablen-Datentyp Alias (33) = "{datatype}"
  - Präfix für den Variablen-Datentyp STRING (19) = "s"
- Bei den genannten Präfixdefinitionen wird für eine Variablen vom Aliastyp "MyMessageType" (z.B. für die Variable "var") das Datentyppräfix "s" erwartet.

Beispiel 2 für eine Alias-Variable:

- Gleiche Situation wie bei Beispiel 1 für eine Alias-Variable, einziger Unterschied:
  - Präfix für den Variablen-Datentyp Alias (33) = "al_{datatype}"
- Dann wird für eine Variablen vom Aliastyp "MyMessageType" das Datentyppräfix "al_s" erwartet.

Beispiel für eine Eigenschaft:

- Präfixdefinitionen
  - Präfix für den Methoden-/Eigenschaftengültigkeitsbereich PRIVATE (121) = "priv_"
  - Präfix für den POU-Typ PROPERTY (107) = "P_{datatype}"
  - Präfix für den Variablen-Datentyp LREAL (18) = "f"
- Hinweis: Für POUs mit einem Zugriffsmodifizierer (also Methoden oder Eigenschaften) wird als Gesamtpräfix die Kombination aus dem Präfix für den Gültigkeitsbereich (NC0121-NC0124: PRIVATE/PROTECTED/INTERNAL/PUBLIC) und dem Präfix für den POU-Typ (NC0105 bei Methode, NC0107 bei Eigenschaft) erwartet.
- Bei den genannten Präfixdefinitionen wird für eine Eigenschaft mit dem Zugriffsmodifizierer PRIVATE und dem Datentyp LREAL somit das Gesamtpräfix "priv_P_f" erwartet.

4.4 Metriken

In der Registerkarte **Metriken** können Sie die Metriken auswählen und konfigurieren, die bei Ausführung des Befehls 'Standard-Metriken anzeigen' [▶ 118] in der Ansicht **Standard-Metriken** für jeden Baustein angezeigt werden sollen.

Es stehen Ihnen über 20 Metriken zur Verfügung, die den zugrundeliegenden Quellcode analysieren und charakterisieren. Bei regelmäßiger Berechnung können die Metriken Hinweise auf negative Trends und Abweichungen von Qualitätszielen geben. Die Kennzahlen stellen somit ein Indiz zur Beurteilung der Softwarequalität dar. In der tabellarischen Ausgabe sind beispielsweise Metriken für die Anzahl an Anweisungen oder den Anteil an Kommentaren zu finden.

Analyse von Bibliotheken

Die folgenden Metriken werden auch für die im Projekt eingebundenen Bibliotheken ausgegeben: Code-Größe, Variablengröße, Stack-Größe und Anzahl Aufrufe.

Übersetzungsfehler bei Verletzungen der Ober-/Untergrenzen

Sie können Verletzungen der Ober- und Untergrenzen der aktivierten Metriken über die Regel SA0150 der statischen Codeanalyse als Übersetzungsfehler ausgeben lassen.

Konfiguration der Metriken

Aktiv	Sie können die einzelnen Metriken über das Kontrollkästchen der jeweiligen Zeile aktivieren oder deaktivieren. Nach Ausführung des Befehls 'Standard-Metriken anzeigen' [▶ 118] werden in der Ansicht **Standard-Metriken** für jeden Programmierbaustein die Metriken angezeigt, die Sie in dieser Konfiguration aktiviert haben. • ☐ : Die Metrik ist deaktiviert und wird nach Ausführung des Befehls **Standard-Metriken anzeigen** in der Ansicht **Standard-Metriken** nicht angezeigt. • ☑ : Die Metrik ist aktiviert und wird nach Ausführung des Befehls **Standard-Metriken anzeigen** in der Ansicht **Standard-Metriken** angezeigt.
Untergrenze Obergrenze	Für jede Metrik können Sie eine individuelle Ober- und Untergrenze definieren, indem Sie die gewünschte Zahl in der jeweiligen Metrikzeile eintragen. Falls eine Metrik nur in eine Richtung begrenzt ist, können Sie die Konfiguration der anderen Richtung leer lassen. Sie geben somit nur die Untergrenze oder nur die Obergrenze an.

Auswertung der Ober- und Untergrenzen

Die eingestellten Ober- und Untergrenzen können Sie auf zwei Arten auswerten.

- Ansicht **Standard-Metriken**:
  - Aktivieren Sie die Metrik, dessen konfigurierten Ober- und Untergrenzen Sie auswerten möchten.
  - Führen Sie den Befehl 'Standard-Metriken anzeigen' [▶ 118] aus.
  - In der tabellarischen Ansicht **Standard-Metriken** zeigt TwinCAT für jeden Programmierbaustein die aktivierten Metriken.
  - Liegt ein Wert außerhalb des Bereichs, der in der Konfiguration durch eine Ober- und/oder Untergrenze definiert ist, erscheint das Tabellenfeld rot hinterlegt.
- Statische Analyse:
  - Aktivieren Sie Regel 150 in der Registerkarte Regeln [▶ 17] als Fehler oder Warnung.
  - Führen Sie die Statische Analyse aus (siehe: Befehl 'Statische Analyse durchführen' [▶ 115]).
  - Verletzungen der Ober- und/oder Untergrenzen werden im Meldungsfenster als Fehler oder Warnung ausgegeben.

Übersicht und Beschreibung der Metriken

Eine Übersicht der Metriken sowie eine detaillierte Beschreibung der Regeln finden Sie im nächsten Kapitel.

4.4.1 Metriken - Übersicht und Beschreibung

Titel in der Ansicht „Standard-Metriken“	Beschreibung
Codegröße	Codegröße [Anzahl Bytes] [▶ 99]
Variablengröße	Variablengröße [Anzahl Bytes] [▶ 99]
Stack-Größe	Stack-Größe [Anzahl Bytes] [▶ 100]
Aufrufe	Anzahl Aufrufe [▶ 101]
Tasks	Anzahl der Aufrufe aus Tasks [▶ 101]
Globale	Anzahl verwendeter globaler Variablen [▶ 101]
EAs	Anzahl Adresszugriffe [▶ 101]
Lokale	Anzahl lokaler Variablen [▶ 102]
Eingänge	Anzahl Eingabevariablen [▶ 102]
Ausgänge	Anzahl Ausgabevariablen [▶ 102]
NOS	Anzahl Anweisungen (NOS) [▶ 103]
Kommentare	Prozentsatz Kommentare [▶ 104]
McCabe	Komplexität (McCabe) [▶ 104]
Komplexität	Kognitive Komplexität [▶ 105]
DIT	Tiefe des Vererbungsbaums (DIT) [▶ 107]
NOC	Anzahl Kindobjekte (NOC) [▶ 107]
RFC	Antwort auf Klasse (RFC) [▶ 108]
CBO	Koppeln zwischen Objekten (CBO) [▶ 109]
Elshof	Referenzierungskomplexität (Elshof) [▶ 109]
LCOM	Mangelnder Zusammenhalt in Methoden (LCOM) [▶ 110]
AS-Verzweigungen (englisch: "SFC Branches")	Anzahl AS-Verzweigungen [▶ 111]
AS-Schritte (englisch: "SFC Steps")	Anzahl AS-Schritte [▶ 111]

Detaillierte Beschreibung

Codegröße [Anzahl Bytes]

Titel Kurzform	Codegröße
Kategorien	Informativ, Effizienz
Definition	Anzahl der Bytes, die ein Baustein zum Applikationscode beiträgt
Weitere Informationen	Die Anzahl hängt auch vom Codegenerator ab. Beispielsweise erzeugt der Codegenerator für Arm®-Prozessoren im Allgemeinen mehr Bytes als der Codegenerator für x86-Prozessoren.

Variablengröße [Anzahl Bytes]

Titel Kurzform	Variablengröße
Kategorien	Informativ, Effizienz
Definition	Größe des statischen Speichers, der von dem Objekt verwendet wird
Weitere Informationen	Bei Funktionsbausteinen ist dies die Größe, die für eine Instanz dieses Funktionsbausteins verwendet wird (und die je nach Speicher-Alignment auch Speicherlücken enthalten kann). Bei Programmen, Funktionen und globalen Variablenlisten ist dies die Summe der Größe aller statischen Variablen.

Beispiel:

```
FUNCTION F_Sample : INT
VAR_INPUT
    a,b   : INT;
END_VAR
VAR
    c,d   : INT;
END_VAR
VAR_STAT
    f,g,h : INT;
END_VAR
```

Die Funktion hat drei statische Variablen vom Typ INT (`f, g, h`), die jeweils 2 Byte Speicherplatz benötigen. `F_Sample` hat folglich eine Variablengröße von 6 Byte.

Stack-Größe [Anzahl Bytes]

Titel Kurzform	Stack-Größe
Kategorien	Informativ, Effizienz, Verlässlichkeit
Definition	Anzahl der Bytes, die für den Aufruf einer Funktion oder eines Funktionsbausteins benötigt werden
Weitere Informationen	Eingangsvariablen und Ausgangsvariablen werden am Speicher ausgerichtet. Dadurch kann eine Lücke zwischen diesen Variablen und den lokalen Variablen entstehen. Diese Lücke wird mitgezählt. Rückgabewerte von aufgerufenen Funktionen, die nicht in ein Register passen, werden auf den Stack geschoben. Der größte dieser Werte bestimmt den zusätzlich zugewiesenen Speicher, der ebenfalls mitzählt. Funktionen oder Funktionsbausteine, die innerhalb der betrachteten POUs aufgerufen werden, haben ihren eigenen Stack-Frame. Deshalb zählt der Speicher für solche Aufrufe nicht mit. Je nach verwendetem Codegenerator verwenden auch Zwischenergebnisse von Berechnungen den Stapel. Diese Ergebnisse werden nicht gezählt.

Beispiel:

```
FUNCTION F_Sample : INT
VAR_INPUT
    a,b   : INT;
END_VAR
VAR
    c,d,e : INT;
END_VAR
VAR_STAT
    f,g,h : INT;
END_VAR
```

```
c := b;
d := a;
e := a + b;
```

Annahme: Für die Berechnung wird die Solution-Plattform „TwinCAT RT (x86)“ verwendet. Das Gerät besitzt ein Stack-Alignment von 4 Byte, wodurch zwischen den Variablen Lücken entstehen können.

Die gesamte Stack-Größe von `F_Sample` beträgt 16 Byte und setzt sich zusammen aus:

- 2 Eingangsvariablen mit je 2 Byte = 4 Byte
- Keine Füllbytes

- Rückgabewert INT = 2 Byte
- Füllbytes für das Stack-Alignment = 2 Byte

- 3 lokale Variablen mit je 2 Byte = 6 Byte
- Füllbytes für das Stack-Alignment = 2 Byte

VAR_STAT wird nicht auf dem Stack gespeichert und erhöht daher nicht die Stack-Größe einer POU.

Anzahl Aufrufe

Titel Kurzform	Aufrufe
Kategorien	Informativ
Definition	Anzahl der Aufrufe der Programmiereinheit (POU) innerhalb der Applikation
Weitere Informationen	Falls ein Programm in einer Task aufgerufen wird, wird dieser Aufruf auch mitgezählt.

Anzahl der Aufrufe aus Tasks

Titel Kurzform	Tasks
Kategorien	Wartbarkeit, Verlässlichkeit
Definition	Anzahl der Tasks, in der die Programmiereinheit (POU) aufgerufen wird
Weitere Informationen	Bei Funktionsbausteinen wird die Anzahl der Tasks gezählt, in denen der Funktionsbaustein selbst oder ein beliebiger Funktionsbaustein im Vererbungsbaum des Funktionsbausteins aufgerufen wird. Bei Methoden und Aktionen wird die Anzahl der Tasks angezeigt, in denen der (übergeordnete) Funktionsbaustein aufgerufen wird.

Beispiel:

```
FUNCTION_BLOCK FB1
FUNCTION_BLOCK FB2 EXTENDS FB1
FUNCTION_BLOCK FB3 EXTENDS FB2
```

Jeder Funktionsbaustein wird in einem eigenen Programm instanziiert und aufgerufen. Zudem wird jedes Programm in einer eigenen Task aufgerufen.

Die Metrik **Anzahl der Aufrufe aus Tasks** ergibt somit:

- Für `FB3`: 1
- Für `FB2`: 2, da die Aufrufe von `FB2` und `FB3` (`EXTENDS FB2`) gezählt werden
- Für `FB1`: 3, da die Aufrufe von `FB1`, `FB2` und `FB3` gezählt werden

Anzahl verwendeter globaler Variablen

Titel Kurzform	Globale
Kategorien	Wartbarkeit, Wiederverwendbarkeit
Definition	Anzahl der verwendeten, unterschiedlichen globalen Variablen, die in der Programmiereinheit (POU) verwendet werden

Anzahl Adresszugriffe

Titel Kurzform	EAs
Kategorien	Wiederverwendbarkeit, Wartbarkeit
Definition	Anzahl der Adresszugriffe in der Implementierung des Objekts

Beispiel:

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

```
    bIn   AT%I* : BOOL;
    bOut  AT%Q* : BOOL;
END_VAR
```

```
bVar := TRUE;
bOut := bIn;
bOut := NOT bOut AND bIn;
```

Die Anzahl der Adresszugriffe für das Programm `MAIN` beträgt 5 und setzt sich aus 2 schreibenden und 3 lesenden Zugriffen zusammen.

Anzahl lokaler Variablen

Titel Kurzform	Lokale
Kategorien	Informativ, Effizienz
Definition	Anzahl der Variablen, die im VAR-Bereich der Programmiereinheit (POU) deklariert sind
Weitere Informationen	Geerbte lokale Variablen werden nicht gezählt.

Anzahl Eingabevariablen

Anzahl der Eingangsvariablen des Bausteins (VAR_INPUT).

Titel Kurzform	Eingänge
Kategorien	Wartbarkeit, Wiederverwendbarkeit
Definition	Anzahl der Variablen, die im VAR_INPUT-Bereich der Programmeinheit (POU) deklariert sind
Weitere Informationen	Geerbte Eingabevariablen werden nicht gezählt.
Standardobergrenze für die zugehörige Regel SA0166 [▶ 78]	10

Anzahl Ausgabevariablen

Anzahl der Ausgangsvariablen des Bausteins (VAR_OUTPUT).

Titel Kurzform	Ausgänge
Kategorien	Wartbarkeit, Wiederverwendbarkeit
Definition	Anzahl der Variablen, die im VAR_OUTPUT-Bereich der Programmeinheit (POU) deklariert sind
Weitere Informationen	Bei Funktionsbausteinen ist dies die Anzahl der benutzerdefinierten Ausgabevariablen (VAR_OUTPUT). Bei Methoden und Funktionen ist dies die Anzahl der benutzerdefinierten Ausgabevariablen (VAR_OUTPUT) plus eins, wenn sie einen Rückgabewert haben. Der Rückgabewert wird mitgezählt. Geerbte Ausgabevariablen werden nicht gezählt. Eine hohe Anzahl an Ausgabevariablen ist ein Zeichen für die Verletzung des Prinzips der eindeutigen Verantwortlichkeit.
Standardobergrenze für die zugehörige Regel SA0166 [▶ 78]	10

Beispiel:

```
METHOD METH : BOOL
VAR_OUTPUT
    a : INT;
    b : LREAL;
END_VAR
```

Die Methode `METH` hat drei Ausgänge:

- Rückgabewert `METH`
- `a`
- `b`

```
METHOD METH1
VAR_OUTPUT
    a : ARRAY[0..10] OF INT;
    b : LREAL;
END_VAR
```

Die Methode `METH1` hat zwei Ausgänge:

- `a`
- `b`

Anzahl Anweisungen (NOS)

Titel Kurzform	NOS
Kategorien	Informativ
Definition	Anzahl der ausführbaren Anweisungen in der Implementierung eines Funktionsbausteines, einer Funktion oder einer Methode
Weitere Informationen	NOS = **N**umber **O**f executable **S**tatements Anweisungen in der Deklaration, leere Anweisungen oder Pragmas werden nicht gezählt.

Beispiel:

```
FUNCTION_BLOCK FB_Sample
VAR_OUTPUT
    nTest  : INT;
    i      : INT;
END_VAR
VAR
    bVar   : BOOL;
    c      : INT := 100;    // statements in the declaration are not counted
END_VAR
```

```
IF bVar THEN                //if statement: +1
    nTest := 0;             // +1
END_IF

WHILE nTest = 1 DO          //while statement: +1
    ;                       // empty statements do not add to the statement count
END_WHILE

FOR c := 0 TO 10 BY 2 DO    //for statement: +1
    i := i+i;               // +1
END_FOR

{text 'simple text pragma'} //pragmas are not counted
nTest := 2;                 //+1
```

Das Beispiel hat sechs Anweisungen.

Prozentsatz Kommentare

Titel Kurzform	Kommentare
Kategorien	Wartbarkeit
Definition	Prozentualer Anteil Kommentare im Quellcode Diese Zahl wird nach der folgenden Formel berechnet: *Prozentsatz = 100 * <Buchstaben in Kommentaren> / <Buchstaben in Quellcode und Kommentaren zusammen>*
Weitere Informationen	Mehrere aufeinander folgende Leerzeichen im Quellcode werden als ein Leerzeichen gezählt, was eine hohe Gewichtung von eingerücktem Quellcode verhindert. Für leere Objekte (kein Quellcode und keine Kommentare) wird ein Prozentsatz von 0 zurückgegeben. Zu den Anweisungen gehören beispielsweise auch Deklarationsanweisungen.

Komplexität (McCabe)

Titel Kurzform	McCabe
Kategorien	Testbarkeit
Definition	Anzahl der Binärverzweigungen im Kontrollfluss der POU (beispielsweise die Anzahl an Verzweigungen bei IF- und CASE-Anweisungen sowie Schleifen)
Weitere Informationen	Die zyklomatische Komplexität nach McCabe ist ein Maß für die Lesbarkeit und Testbarkeit von Quellcode. Sie wird durch Zählen der Anzahl der Binärverzweigungen im Kontrollfluss der POU berechnet. Die zyklomatische Komplexität bestraft eine hohe Verzweigung, da eine hohe Verzweigung die Anzahl der für eine hohe Testabdeckung benötigten Testfälle erhöht.
Empfohlene Obergrenze	10

Die folgenden Beispiele zeigen, wie die Komplexität nach McCabe berechnet wird.

Beispiel: IF-Anweisung

```
// every POU has an initial cyclomatic complexity of 1, since it has at least 1 branch
IF b1 THEN                // +1 for the THEN branch
    ;
ELSIF b2 THEN             // +1 for the THEN branch of the IF inside the ELSE
    ;
ELSE
    IF b3 OR b4 THEN      // +1 for the THEN branch
        ;
    END_IF
END_IF
```

Der Codeschnipsel hat eine zyklomatische Komplexität von 4.

Beispiel: CASE-Anweisung

```
// every POU has an initial cyclomatic complexity of 1, since it has at least 1 branch
CASE a OF
    1:     ;              // +1
    2:     ;              // +1
    3,4,5: ;              // +1
ELSE                      // the ELSE statement does not increase the cyclomatic complexity
    ;
END_CASE
```

Der Codeschnipsel hat eine zyklomatische Komplexität von 4.

Beispiel: Schleifenanweisung

```
// every POU has an initial cyclomatic complexity of 1, since it has at least 1 branch
WHILE b1 DO               // +1 for the WHILE loop
    ;
END_WHILE
```

```
REPEAT                    // +1 for the REPEAT loop
    ;
UNTIL b2
END_REPEAT

FOR a := 0 TO 100 BY 2 DO // +1 for the REPEAT loop
    ;
END_FOR
```

Der Codeschnipsel hat eine zyklomatische Komplexität von 4.

Beispiel: Andere Anweisungen

Auch die folgenden Anweisungen führen zu einer Erhöhung der zyklomatischen Komplexität:

```
FUNCTION FUN : STRING
VAR_INPUT
    bReturn  : BOOL;
    bJump    : BOOL;
END_VAR
```

```
// every POU has an initial cyclomatic complexity of 1, since it has at least 1 branch
JMP(bJump) lbl;           //Conditional jumps increase the cyclomatic complexity by 1

FUN := 'u';
RETURN(condition_return); //Conditional returns increase the cyclomatic complexity by 1, too

lbl:
    FUN := 't';
```

Der Codeschnipsel hat eine zyklomatische Komplexität von 3.

Kognitive Komplexität

Titel Kurzform	Kognitive Komplexität
Kategorien	Wartbarkeit
Definition	Summe der Teilkomplexitäten, die sich beispielsweise durch Verzweigungen im Kontrollfluss der POU und durch komplexe boolesche Ausdrücke ergeben
Weitere Informationen	Die kognitive Komplexität ist ein Maß für die Lesbarkeit und Verständlichkeit von Quellcode, das von Sonarsource™ im Jahr 2016 eingeführt wurde. Sie bestraft eine starke Verschachtelung des Kontrollflusses und komplexe boolesche Ausdrücke. Die kognitive Komplexität wird nur für strukturierte Textimplementierungen berechnet.
Standardobergrenze für die zugehörige Regel SA0178 [▶ 82]	20

Tipp

Mit dem Befehl 'Kognitive Komplexität für aktuellen Editor anzeigen' [▶ 123] können Sie die Inkremente für strukturierten Text zusätzlich direkt im Editor anzeigen.

Die folgenden Beispiele zeigen, wie die kognitive Komplexität berechnet wird.

Beispiel: Kontrollfluss

Anweisungen, die den Kontrollfluss manipulieren, erhöhen die kognitive Komplexität um 1.

```
IF TRUE THEN              //+1 cognitive complexity
    ;
END_IF

WHILE TRUE DO             //+1 cognitive complexity
    ;
END_WHILE

FOR i := 0 TO 10 BY 1 DO  //+1 cognitive complexity
    ;
END_FOR
```

```
REPEAT                    //+1 cognitive complexity
    ;
UNTIL TRUE
END_REPEAT
```

Der Codeschnipsel hat eine kognitive Komplexität von 4.

Beispiel: Verschachtelung des Kontrollflusses

Bei der Verschachtelung des Kontrollflusses wird für jede Stufe der Verschachtelung ein Inkrement von 1 hinzugefügt.

```
IF TRUE THEN                      //+1 cognitive complexity
    WHILE TRUE DO                 //+2 (+1 for the loop itself, +1 for the nesting inside the IF)
        FOR i := 0 TO 10 BY 1 DO  //+3 (+1 for the FOR loop itself, +2 for the nesting inside the
WHILE and the IF)
            ;
        END_FOR
    END_WHILE

    REPEAT                        //+2 (+1 for the loop itself, +1 for the nesting inside the IF)
        ;
    UNTIL TRUE
    END_REPEAT
END_IF
```

Der Codeschnipsel hat eine kognitive Komplexität von 8.

Beispiel: Boolescher Ausdruck

Da boolesche Ausdrücke eine große Rolle beim Verständnis von Quellcode spielen, werden sie auch bei der Berechnung der kognitiven Komplexität berücksichtigt.

Das Verständnis von booleschen Ausdrücken, die mit demselben booleschen Operator verbunden sind, ist nicht so schwierig wie das Verständnis eines booleschen Ausdrucks, der alternierende boolesche Operatoren enthält. Daher erhöht jede Kette von gleichen booleschen Operatoren in einem Ausdruck die kognitive Komplexität.

```
b := b1;                          //+0: a simple expression, containing no operators, has no
increment
```

Der einfache Ausdruck ohne Operator hat ein Inkrement von 0.

```
b := b1 AND b2;                   //+1: one chain of AND operators
```

Der Ausdruck mit einer AND-Verknüpfung hat ein Inkrement von 1.

```
b := b1 AND b2 AND b3;            //+1: one more AND, but the number of chains of operators does
not change
```

Der Ausdruck hat ein AND mehr. Aber da es der gleiche Operator ist, ändert sich die Anzahl der mit identischen Operatoren gebildeten Kette nicht.

```
b := b1 AND b2 OR b3;             //+2: one chain of AND operators and one chain of OR operators
```

Der Ausdruck hat eine Kette von AND-Operatoren und eine Kette von OR-Operatoren. Das ergibt ein Inkrement von 2.

```
b := b1 AND b2 OR b3 AND b4 AND b5;  //+3: one chain of AND operators, one chain of OR operators and
another chain of AND operators
```

Der Codeschnipsel hat ein Inkrement von 3.

```
b := b1 AND NOT b2 AND b3;          //+1: the unary NOT operator is not considered in the cognitive
complexity
```

Der unäre Operator NOT wird bei der kognitiven Komplexität nicht berücksichtigt.

Beispiel: Weitere Anweisungen mit Inkrement

Strukturierter Text hat zusätzliche Anweisungen und Ausdrücke, die den Kontrollfluss verändern.

Die folgenden Anweisungen werden mit einem Inkrement der kognitiven Komplexität bestraft:

```
aNewLabel:
    x := MUX(i, a,b,c);  //+1 for MUX operator
    y := SEL(b, i,j);    //+1 for SEL operator
JMP aNewLabel;           //+1 for JMP to label
```

EXIT- und RETURN-Anweisungen erhöhen nicht die kognitive Komplexität.

Tiefe des Vererbungsbaums (DIT)

Titel Kurzform	DIT
Kategorien	Wartbarkeit
Definition	Anzahl der Vererbungen, bis ein Funktionsbaustein erreicht ist, der keinen anderen Funktionsbaustein erweitert
Weitere Informationen	DIT = **D**epth of **I**nheritance **T**ree

Beispiel:

```
FUNCTION_BLOCK FB_Base
FUNCTION_BLOCK FB_Sub EXTENDS FB_Base
FUNCTION_BLOCK FB_SubSub EXTENDS FB_Sub
```

Die Metrik **Tiefe des Vererbungsbaum** beträgt:

- Für `FB_Base`: 0, da er selbst ein Funktionsbaustein ist, der keinen anderen Funktionsblock erweitert.
- Für `FB_Sub`: 1, da ein Schritt erforderlich ist, um zu `FB_Base` zu gelangen.
- Für `FB_SubSub`: 2, da ein Schritt zu `FB_Sub` und ein weiterer Schritt zu `FB_Base` benötigt wird.

Anzahl Kindobjekte (NOC)

Titel Kurzform	NOC
Kategorien	Wiederverwendbarkeit, Wartbarkeit
Definition	Anzahl der Funktionsbausteine, die den gegebenen Basisfunktionsbaustein erweitern. Dabei werden Funktionsbausteine, die einen Basisfunktionsbaustein indirekt erweitern, nicht mitgezählt.
Weitere Informationen	NOC = **N**umber **O**f **C**hildren

Beispiel:

```
FUNCTION_BLOCK FB_Base
FUNCTION_BLOCK FB_Sub EXTENDS FB_Base
FUNCTION_BLOCK FB_SubSub1 EXTENDS FB_Sub
FUNCTION_BLOCK FB_SubSub2 EXTENDS FB_Sub
```

Die Metrik **Anzahl Kindobjekte** beträgt:

- Für `FB_Base`: 1 Kindobjekt (FB_Sub)
- Für `FB_Sub`: 2 Kindobjekte (FB_SubSub1, FB_SubSub2)
- Für `FB_SubSub1`: 0 Kindobjekte
- Für `FB_SubSub2`: 0 Kindobjekte

Antwort auf Klasse (RFC)

Titel Kurzform	RFC
Kategorien	Wartbarkeit, Wiederverwendbarkeit
Definition	Anzahl unterschiedlicher POUs, Methoden oder Aktionen, die von einer POU aufgerufen werden können
Weitere Informationen	RFC = **R**esponse **F**or **C**lass Der Wert dient der Messung der Komplexität (im Hinblick auf Testbarkeit und Wartbarkeit). Dabei werden alle möglichen direkten und indirekten über Assoziationen erreichbaren Methodenaufrufe gewertet. Diese können verwendet werden, um auf eine eingegangene Nachricht zu antworten oder auf ein eingetretenes Ereignis zu reagieren.

Beispiel:

Funktionsbaustein FB1:

```
FUNCTION_BLOCK FB1
VAR
    d,x,y : INT;
END_VAR
```

```
x := METH(d+10);
y := FUN(42, 0.815);
```

Methode FB1.METH:

```
METHOD METH : INT
VAR_INPUT
    i     : INT;
END_VAR
```

```
METH := FUN(CUBE(i), 3.1415);
```

Funktion Cube:

```
FUNCTION CUBE : INT
VAR_INPUT
    i     : INT;
END_VAR
```

```
CUBE := i*i*i;
```

Funktion FUN:

```
FUNCTION FUN : INT
VAR_INPUT
    a     : INT;
    f     : LREAL;
END_VAR
```

```
FUN := LREAL_TO_INT(f*10)*a;
```

- `FUN`, `CUBE`: Diese Funktionen haben einen RFC von 0, denn keine der beiden Funktionen rufen andere Funktionen, Funktionsbausteine oder Methoden für ihre Berechnungen auf.
- `FB1.METH`: Die Methode verwendet `FUN` und `CUBE`, was einen RFC von 2 ergibt.
- `FB1`:
  - Der Funktionsbaustein `FB1` ruft `METH` und `FUN` auf, was seinen RFC um 2 erhöht.
  - Bei `FB1` muss auch seine Methode `METH` berücksichtigt werden. `METH` verwendet `FUN` und `CUBE`. `FUN` ist bereits zum RFC von `FB1` hinzugefügt (siehe vorheriger Stichpunkt). Somit erhöht nur die Verwendung von `CUBE` in `METH` den RFC für `FB1` auf 3.

Koppeln zwischen Objekten (CBO)

Titel Kurzform	CBO
Kategorien	Wartbarkeit, Wiederverwendbarkeit
Definition	Anzahl weiterer Funktionsbausteine, die in einem Funktionsbaustein instanziiert und verwendet werden
Weitere Informationen	CBO = **C**oupling **B**etween **O**bjects Ein Funktionsbausteine mit einer hohen Kopplung zwischen Objekten ist wahrscheinlich an vielen verschiedenen Aufgaben beteiligt und verstößt daher gegen das Prinzip der eindeutigen Verantwortlichkeit.
Standardobergrenze für die zugehörige Regel SA0179 [▶ 83]	30

Beispiel:

```
FUNCTION_BLOCK FB_Base
VAR
    fb3  : FB3;  // +1 instantiated here
END_VAR
```

```
FUNCTION_BLOCK FB_Sub EXTENDS FB_Base   // +0 for EXTENDS
VAR
    fb1  : FB1;  // +1: instantiated here
    fb2  : FB2;  // +1: instantiated here
END_VAR
```

```
fb3();           // +0: instantiated in FB_Base, no increment for call
```

- Die Erweiterung eines Funktionsbausteine erhöht nicht die Kopplung zwischen Objekten.
- fb3 wird in der Implementierung von `FB_Base` instanziiert und an `FB_Sub` vererbt. Der Aufruf in `FB_Sub` erhöht nicht die Kopplung zwischen den Objekten.
- Somit beträgt die Metrik **Koppeln zwischen Objekten** für `FB_Sub`: 2

Referenzierungskomplexität (Elshof)

Referenzkomplexität = Referenzierte Daten (Anzahl Variablen) / Anzahl der Datenreferenzen

Titel Kurzform	Elshof
Kategorien	Effizienz, Wartbarkeit, Wiederverwendbarkeit
Definition	Komplexität des Datenflusses einer POU Die Referenzierungskomplexität wird nach der folgenden Formel berechnet: *<Anzahl verwendeter Variablen> / <Anzahl Variablenzugriffe>*
Weitere Informationen	Es werden nur Variablenzugriffe im Implementierungsteil der POU berücksichtigt.

Beispiel:

```
PROGRAM MAIN
VAR
    i, j  : INT;
    k     : INT := GVL.m;
    b, c  : BOOL;
    fb    : FB_Sample;
END_VAR
```

```
fb(paramA := b);   // +3 accesses (fb, paramA and b)
i := j;            // +2 accesses (i and j)
j := GVL.d;        // +2 accesses (j and GVL.d)
```

Für die Metrik **Referenzierungskomplexität (Elshof)** ergibt sich für `MAIN`:

- Anzahl verwendeter Variablen = 6
- Anzahl Variablenzugriffe = 7

- Referenzierungskomplexität (Elshof) = Anzahl verwendeter Variablen/Anzahl Variablenzugriffe = 6/7 = 0.85

Achtung:

- c und k werden nicht verwendet und zählen daher nicht als "verwendete Variablen".
- Die Zuweisung `k : INT := GVL.m;` wird nicht gezählt, da sie Teil der Deklaration des Programms ist.

Mangelnder Zusammenhalt in Methoden (LCOM)

Titel Kurzform	LCOM
Kategorien	Wartbarkeit, Wiederverwendbarkeit
Definition	Zusammenhalt/Kohäsion = Paare von Methoden ohne gemeinsame Instanzvariablen abzüglich Paare von Methoden mit gemeinsamen Instanzvariablen Die Metrik wird nach folgender Formel berechnet: *MAX(0, <Anzahl Objektpaare ohne Kohäsion> - <Anzahl Objektpaare mit Kohäsion>)*
Weitere Informationen	LCOM: **L**ack of **Co**hesion in **M**ethods Der Zusammenhalt bzw. die Kohäsion zwischen Funktionsbausteinen, ihren Aktionen, Transitionen und Methoden beschreibt, ob sie auf die gleichen Variablen zugreifen. Der Mangel an Kohäsion von Methoden beschreibt, wie stark die Objekte eines Funktionsbausteins miteinander verbunden sind. Je geringer der Kohäsionsmangel, desto stärker ist die Verbindung zwischen den Objekten. Funktionsbausteine mit einem hohen Mangel an Kohäsion sind wahrscheinlich an vielen verschiedenen Aufgaben beteiligt und verletzen daher das Prinzip der eindeutigen Verantwortlichkeit.

Beispiel:

Funktionsbaustein FB:

```
FUNCTION_BLOCK FB
VAR_INPUT
    a    : BOOL;
END_VAR
VAR
    i,b  : BOOL;
END_VAR
```

Aktion FB.ACT:

```
i := FALSE;
```

Methode FB.METH:

```
METHOD METH : BOOL
VAR_INPUT
    c    : BOOL;
END_VAR
```

```
METH := c;
i := TRUE;
```

Methode FB.METH2:

```
METHOD METH2 : INT
VAR_INPUT
END_VAR
```

```
METH2 := SEL(b,3,4);
```

Für die Metrik **Mangelnder Zusammenhalt in Methoden (LCOM)** ergibt sich für `FB`:

- Objektpaare ohne Verbindung/Kohäsion (5 Paare):
  - `FB, FB.ACT`
  - `FB, FB.METH`

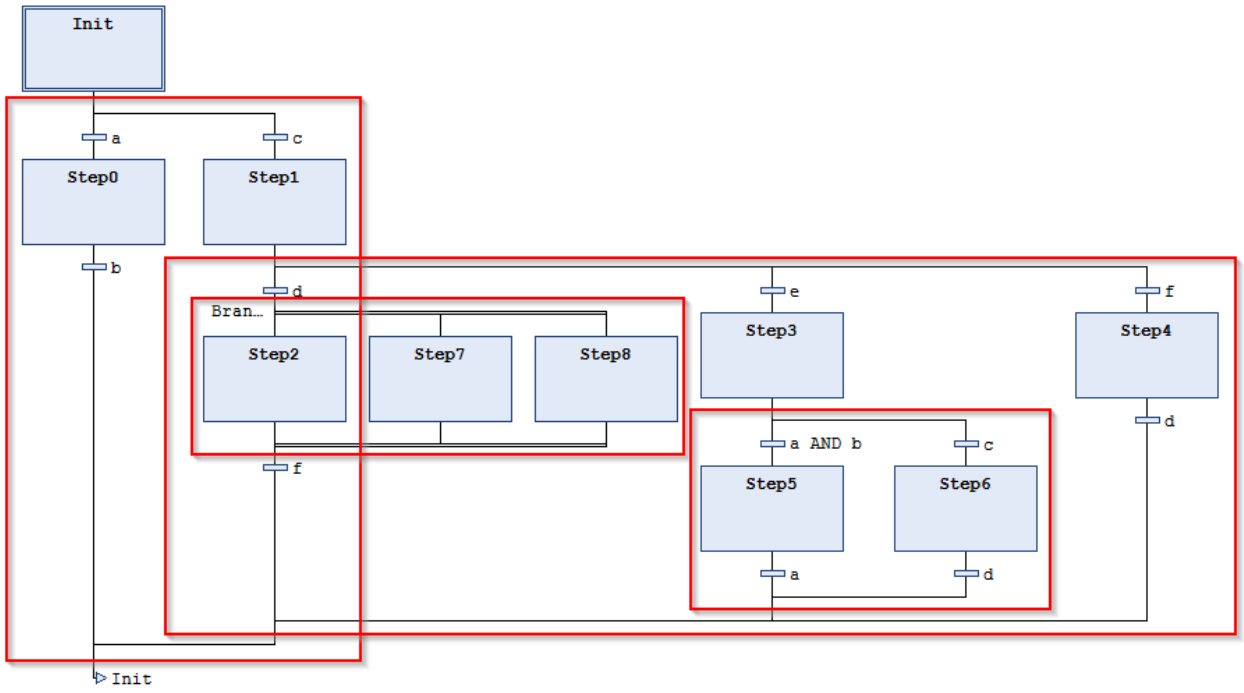
- `FB`, `FB.METH2`
- `FB.ACT`, `FB.METH2`
- `FB.METH`, `FB.METH2`

- Objektpaare mit Verbindung/Kohäsion (1 Paar):
  - `FB.ACT`, `FB.METH` (beide verwenden `i`)
- LCOM = Anzahl Objektpaare ohne Kohäsion - Anzahl Objektpaare mit Kohäsion = 5 – 1 = 4

Anzahl AS-Verzweigungen

Titel Kurzform	AS-Verzweigungen
Kategorien	Testbarkeit, Wartbarkeit
Definition	Anzahl alternativer und paralleler Verzweigungen einer POU der Implementierungssprache AS (Ablaufsprache)

Beispiel:



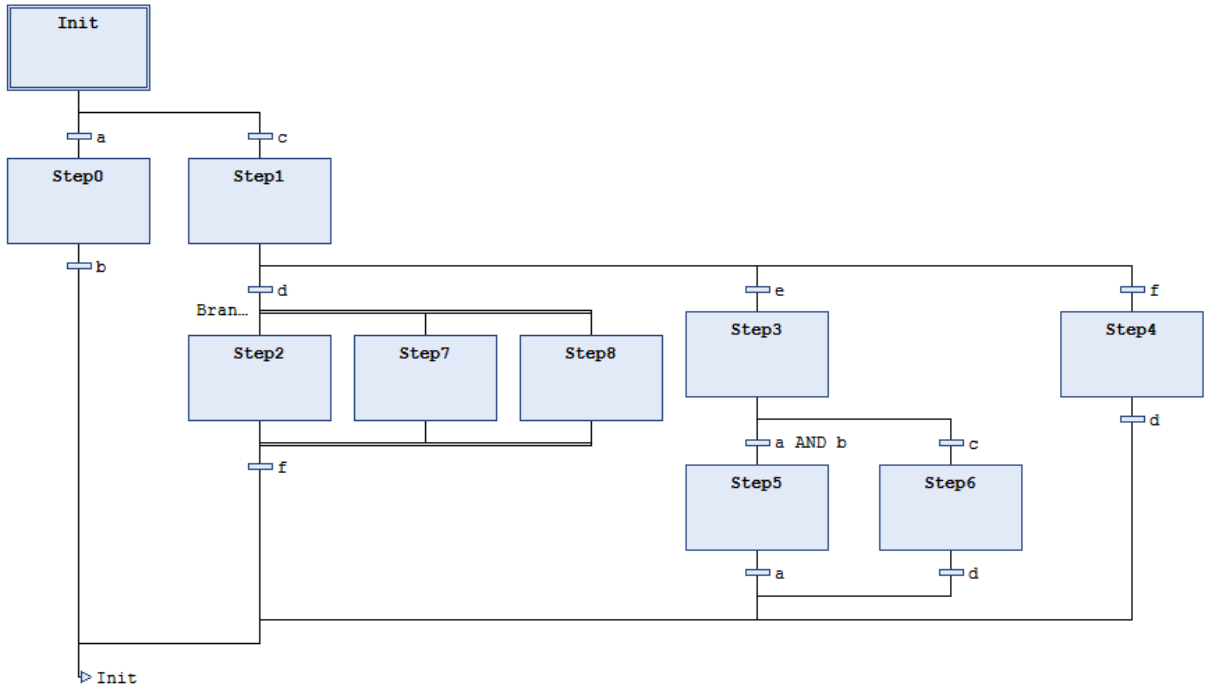
Das obige Codeschnipsel in AS hat 4 Verzweigungen: 3 alternative und 1 parallele Verzweigung.

Anzahl AS-Schritte

Wenn der Funktionsbaustein in Ablaufsprache (AS, englisch: SFC) implementiert ist, gibt diese Codemetrik die Anzahl an Schritten im Funktionsbaustein an.

Titel Kurzform	AS-Schritte
Kategorien	Wartbarkeit
Definition	Anzahl der Schritte in einer POU der Implementierungssprache AS (Ablaufsprache)
Weitere Informationen	Es werden nur die Schritte gezählt, die in der in AS programmierten POU enthalten sind. Schritte in den Implementierungen von in POUs aufgerufenen Aktionen oder Transitionen werden nicht gezählt.

Beispiel:



Das obige Codeschnipsel in AS hat 10 Schritte.

Metriken, die in Versionen < TwinCAT 3.1.4026.14 verfügbar sind:

Spaltenabkürzung in der Ansicht „Standard-Metriken“	Beschreibung
Prather	Verschachtelungskomplexität (Prather) [▶ 112]
n1 (Halstead)	Halstead – Anzahl unterschiedlicher verwendeter Operatoren (n1) [▶ 112]
N1 (Halstead)	Halstead – Anzahl Operatoren (N1) [▶ 112]
n2 (Halstead)	Halstead – Anzahl unterschiedlicher verwendeter Operanden (n2) [▶ 112]
N2 (Halstead)	Halstead – Anzahl Operanden (N2) [▶ 112]
HL (Halstead)	Halstead – Länge (HL) [▶ 112]
HV (Halstead)	Halstead – Volumen (HV) [▶ 112]
D (Halstead)	Halstead – Schwierigkeit (D) [▶ 112]

Verschachtelungskomplexität (Prather)

Verschachtelungsgewicht = Anweisungen * Verschachtelungstiefe

Verschachtelungskomplexität = Verschachtelungsgewicht / Anzahl Anweisungen

Verschachtelung beispielsweise durch IF/ELSEIF- oder CASE/ELSE-Anweisungen.

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

Die folgenden Metriken gehören zu dem Bereich "Halstead":

- Anzahl unterschiedlicher verwendeter Operatoren - Halstead (n1)

- Anzahl Operatoren - Halstead (N1)

- Anzahl unterschiedlicher verwendeter Operanden - Halstead (n2)

- Anzahl Operanden - Halstead (N2)

- Länge - Halstead (HL)

- Volumen - Halstead (HV)

- Schwierigkeit - Halstead (D)

Hintergrundinformationen:

- Verhältnis von Operatoren und Operanden (Anzahl, Komplexität, Testaufwand)
- Basiert auf der Annahme, dass sich ausführbare Programme aus Operatoren und Operanden zusammensetzen.
- Operanden in TwinCAT: Variablen, Konstanten, Komponenten, Literale und IEC-Adressen.
- Operatoren in TwinCAT: Schlüsselwörter, logische und Vergleichsoperatoren, Zuweisungen, IF, FOR, BY, ^, ELSE, CASE, Caselabel, BREAK, RETURN, SIN, +, Labels, Aufrufe, Pragmas, Konvertierungen, SUPER, THIS, Indexzugriff, Komponentenzugriff etc.

Für jedes Programm werden die folgenden Basismaße gebildet:

- **Anzahl unterschiedlicher verwendeter Operatoren - Halstead (n1),**
  Anzahl unterschiedlicher verwendeter Operanden - Halstead (n2):
  - Anzahl der verwendeten unterschiedlichen Operatoren (h_1) und Operanden (h_2), zusammen die Vokabulargröße h.
- **Anzahl Operatoren - Halstead (N1),**
  Anzahl Operanden - Halstead (N2):
  - Anzahl der insgesamt verwendeten Operatoren (N_1) und Operanden (N_2), zusammen die Implementierungslänge N.
- (Sprachkomplexität = Operatoren/Operatorenvorkommnisse * Operanden/Operandenvorkommnisse)

Hieraus werden dann die Größen Halstead-Länge (HL) und Halstead-Volumen (HV) errechnet:

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

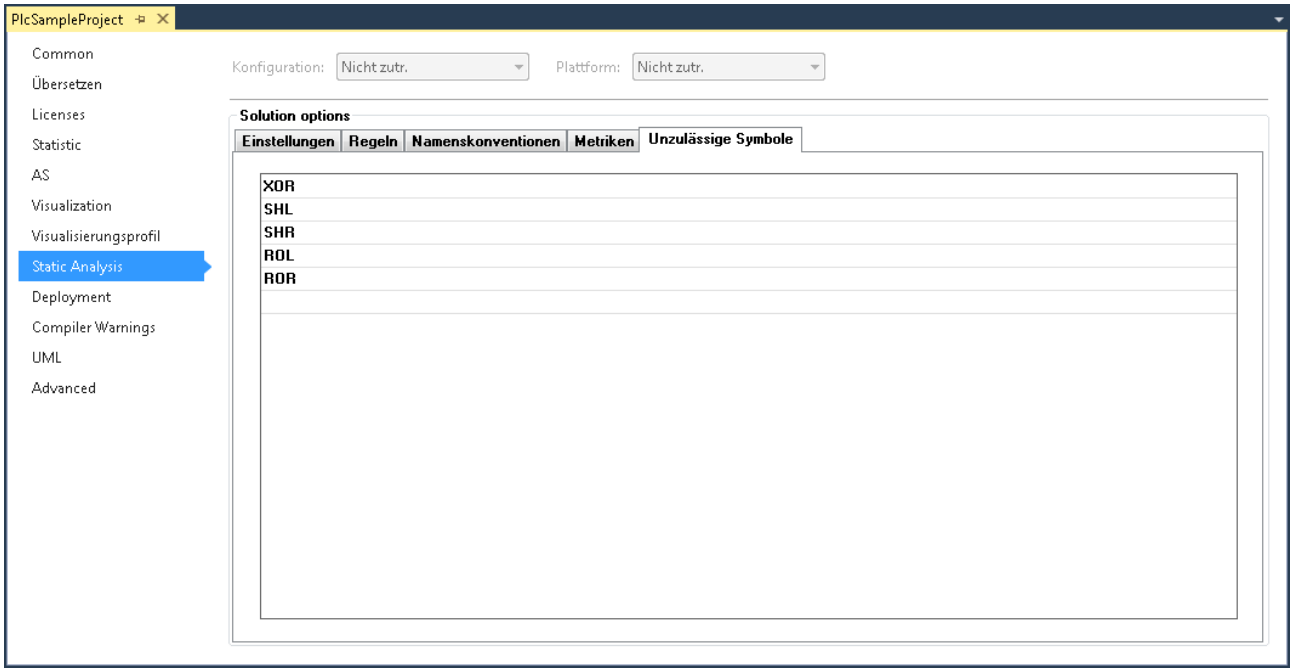
Aus den Basisgrößen kann man verschiedene Kennzahlen berechnen:

- **Schwierigkeit - Halstead (D)**:
  - Schwierigkeit, ein Programm zu schreiben bzw. zu verstehen (beispielsweise bei einem Code-Review)
  - $D = h_1/2 * N2/h_2$
- Aufwand:
  - E = D*V

Die Kennzahlen stimmen meist sehr gut mit tatsächlich gemessenen Werten überein. Der Nachteil ist, dass die Methode nur einzelne Funktionen betrifft und ausschließlich lexikalische/textuelle Komplexität misst.

4.5 Unzulässige Symbole

In der Registerkarte **Unzulässige Symbole** können Sie die Schlüsselwörter, Symbole und Bezeichner festlegen, die im Code des Projekts nicht verwendet werden dürfen. Die Unzulässigen Symbole werden bei der Durchführung der Statischen Analyse [▶ 115] überprüft.



Konfiguration der unzulässigen Symbole

Sie können diese Symbole direkt in der Zeile eingeben oder über die Eingabehilfe auswählen. Bei Ausführung der Static Analysis wird der Code auf das Vorhandensein dieser Begriffe geprüft. Bei Treffern wird ein Fehler im Meldungsfenster ausgegeben.

Syntax von Symbolverletzungen im Meldungsfenster

Wenn im Code ein Symbol verwendet wird, welches als unzulässiges Symbol konfiguriert ist, wird nach Ausführung der Statischen Analyse ein Fehler im Meldungsfenster ausgegeben.

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

Beispiel für das Symbol XOR: "Forbidden symbol 'XOR'“

5 Befehle

5.1 Befehl 'Statische Analyse durchführen'

Symbol: 

Funktion: Der Befehl startet die statische Codeanalyse für das gerade aktive SPS-Projekt und gibt die Ergebnisse im Meldungsfenster aus.

Aufruf: Menü **Erstellen** oder Kontextmenü des SPS-Projektobjekts

Bei Durchführung der Statischen Analyse wird die Einhaltung der Kodierregeln, der Namenskonventionen und der unzulässigen Symbole überprüft. Die Statische Analyse kann über diesen Befehl manuell angestoßen werden (explizite Durchführung) oder sie kann automatisch mit der Codeerzeugung durchgeführt werden (implizite Durchführung, weitere Informationen siehe unten).

Das Ergebnis der Statischen Analyse, also Meldungen bezüglich Regelverletzungen, gibt TwinCAT im Meldungsfenster aus. Welche Regeln [▶ 17], Namenskonventionen [▶ 83] und unzulässigen Symbole [▶ 114] bei der Durchführung der Statischen Analyse berücksichtigt werden sollen, können Sie in den SPS-Projekteigenschaften konfigurieren [▶ 14]. Des Weiteren können Sie jeweils definieren, ob die Verletzung einer Kodierregel als Fehler oder Warnung im Meldungsfenster erscheint (siehe: Regeln [▶ 17]).

Siehe auch: Syntax im Meldungsfenster [▶ 116]

Bitte beachten Sie, dass das selektierte SPS-Projekt vor der Durchführung dieses Befehls erstellt wird, und die Prüfung durch die Statische Analyse nur gestartet wird, wenn die Codegenerierung erfolgreich war, d.h. wenn der Compiler keine Kompilierfehler festgestellt hat.

Bitte beachten Sie auch den Befehl 'Statische Analyse durchführen [Überprüfe alle Objekte]' [▶ 117] und die Unterschiede zwischen den beiden Befehlen, welche in der folgenden Tabelle beschrieben sind.

Unterschiede	Befehl 'Statische Analyse durchführen'	Befehl 'Statische Analyse durchführen [Überprüfe alle Objekte]'
Geltungsbereich/ Funktionsweise	**Genutzte Objekte:** Die aktivierten Regeln werden auf die Objekte angewandt, die in dem SPS-Projekt verwendet werden. **Ungenutzte Objekte:** Ungenutzte Objekte werden bei diesem Befehl nicht überprüft.	**Genutzte Objekte:** Die aktivierten Regeln werden auf die Objekte angewandt, die in dem SPS-Projekt verwendet werden. **Ungenutzte Objekte:** Auf die ungenutzten Objekte werden die Regeln angewandt, die aktiviert sind und welche im Precompile überprüft werden können. Sehen Sie dazu auch: QuickFix/Precompile [▶ 134]
Hinweis	Falls Sie auch die ungenutzten Objekte von der Statischen Analyse überprüfen lassen möchten, können Sie den **Befehl 'Statische Analyse durchführen [Überprüfe alle Objekte]'** verwenden.	Der Befehl ist in erster Linie bei der Erstellung von Bibliotheken bzw. bei der Bearbeitung von Bibliotheksprojekten nützlich.
Ausführungsmöglichkeiten des Befehls	Die Statische Analyse kann sowohl explizit über den Befehl als auch implizit ausgeführt werden. Die implizite Durchführung der Statischen Analyse bei jeder Codegenerierung können Sie in den SPS-Projekteigenschaften (Registerkarte Einstellungen [▶ 14]) ein- bzw. ausschalten. Wenn Sie die Option **Statische Analyse automatisch durchführen** aktiviert haben, führt TwinCAT die Statische Analyse im Anschluss an die erfolgreiche Codegenerierung durch (wie beispielsweise bei Befehl **Projekt erstellen**). Zudem kann der Befehl per Automation Interface aufgerufen werden. Sehen Sie dazu auch: Automation Interface Unterstützung [▶ 136]	Die "Überprüfe alle Objekte"-Variante kann nicht implizit, sondern nur explizit über den Befehl ausgeführt werden. Zudem kann der Befehl per Automation Interface aufgerufen werden. Sehen Sie dazu auch: Automation Interface Unterstützung [▶ 136]

5.1.1 Syntax im Meldungsfenster

Syntax von Regelverletzungen im Meldungsfenster

Jede Regel besitzt eine eindeutige Nummer (in der Konfigurationsansicht der Regeln in runden Klammern hinter der Regel dargestellt). Wenn während der Statischen Analyse die Verletzung einer Regel festgestellt wird, wird die Nummer zusammen mit einer Fehler- bzw. Warnungsbeschreibung gemäß folgender Syntax im Meldungsfenster ausgegeben. Die Abkürzung "SA" weist dabei auf "Static Analysis" hin.

Syntax: **"SA<Regelnummer>: <Regelbeschreibung>"**

Beispiel für Regelnummer 33 (Nicht verwendete Variablen): "SA0033: Nicht verwendet: Variable 'bSample'"

Syntax von Konventionsverletzungen im Meldungsfenster

Jede Namenskonvention besitzt eine eindeutige Nummer (in der Konfigurationsansicht der Namenskonventionen in runden Klammern hinter der Konvention dargestellt). Wenn während der Statischen Analyse die Verletzung einer Konvention bzw. einer Vorgabe festgestellt wird, wird die Nummer zusammen mit einer Fehlerbeschreibung gemäß folgender Syntax in der Fehlerliste ausgegeben. Die Abkürzung "NC" weist dabei auf "Naming Convention" hin.

Syntax: **"NC<Präfix-Konventionsnummer>: <Konventionsbeschreibung>"**

Beispiel für Konventionsnummer 151 (DUTs vom Typ Struktur): "NC0151: Ungültiger Typname 'STR_Sample'. Erwartetes Präfix 'ST_'"

Syntax von Symbolverletzungen im Meldungsfenster

Wenn im Code ein Symbol verwendet wird, welches als unzulässiges Symbol konfiguriert ist, wird nach Ausführung der Statischen Analyse ein Fehler im Meldungsfenster ausgegeben.

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

Beispiel für das Symbol XOR: "Forbidden symbol 'XOR'“

5.2 Befehl 'Statische Analyse durchführen [Überprüfe alle Objekte]'

Symbol: 

Funktion: Der Befehl startet die statische Codeanalyse für alle Objekte des gerade aktiven SPS-Projekts und gibt die Ergebnisse im Meldungsfenster aus.

Aufruf: Menü **Erstellen** oder Kontextmenü des SPS-Projektobjekts

Bei Durchführung der Statischen Analyse wird die Einhaltung der Kodierregeln, der Namenskonventionen und der unzulässigen Symbole überprüft. Die Statische Analyse kann über diesen Befehl manuell angestoßen werden (explizite Durchführung).

Das Ergebnis der Statischen Analyse, also Meldungen bezüglich Regelverletzungen, gibt TwinCAT im Meldungsfenster aus. Welche Regeln [▶ 17], Namenskonventionen [▶ 83] und unzulässigen Symbole [▶ 114] bei der Durchführung der Statischen Analyse berücksichtigt werden sollen, können Sie in den SPS-Projekteigenschaften konfigurieren [▶ 14]. Des Weiteren können Sie jeweils definieren, ob die Verletzung einer Kodierregel als Fehler oder Warnung im Meldungsfenster erscheint (siehe: Regeln [▶ 17]).

Siehe auch: Syntax im Meldungsfenster [▶ 116]

Bitte beachten Sie, dass das selektierte SPS-Projekt vor der Durchführung dieses Befehls erstellt wird, und die Prüfung durch die Statische Analyse nur gestartet wird, wenn die Codegenerierung erfolgreich war, d.h. wenn der Compiler keine Kompilierfehler festgestellt hat.

Bitte beachten Sie auch den Befehl 'Statische Analyse durchführen' [▶ 115] und die Unterschiede zwischen den beiden Befehlen, welche in der folgenden Tabelle beschrieben sind.

Unterschiede	Befehl 'Statische Analyse durchführen'	Befehl 'Statische Analyse durchführen [Überprüfe alle Objekte]'
Geltungsbereich/ Funktionsweise	**Genutzte Objekte:** Die aktivierten Regeln werden auf die Objekte angewandt, die in dem SPS-Projekt verwendet werden. **Ungenutzte Objekte:** Ungenutzte Objekte werden bei diesem Befehl nicht überprüft.	**Genutzte Objekte:** Die aktivierten Regeln werden auf die Objekte angewandt, die in dem SPS-Projekt verwendet werden. **Ungenutzte Objekte:** Auf die ungenutzten Objekte werden die Regeln angewandt, die aktiviert sind und welche im Precompile überprüft werden können. Sehen Sie dazu auch: QuickFix/Precompile [▶ 134]
Hinweis	Falls Sie auch die ungenutzten Objekte von der Statischen Analyse überprüfen lassen möchten, können Sie den **Befehl 'Statische Analyse durchführen [Überprüfe alle Objekte]'** verwenden.	Der Befehl ist in erster Linie bei der Erstellung von Bibliotheken bzw. bei der Bearbeitung von Bibliotheksprojekten nützlich.
Ausführungsmöglichkeiten des Befehls	Die Statische Analyse kann sowohl explizit über den Befehl als auch implizit ausgeführt werden. Die implizite Durchführung der Statischen Analyse bei jeder Codegenerierung können Sie in den SPS-Projekteigenschaften (Registerkarte Einstellungen [▶ 14]) ein- bzw. ausschalten. Wenn Sie die Option **Statische Analyse automatisch durchführen** aktiviert haben, führt TwinCAT die Statische Analyse im Anschluss an die erfolgreiche Codegenerierung durch (wie beispielsweise bei Befehl **Projekt erstellen**). Zudem kann der Befehl per Automation Interface aufgerufen werden. Sehen Sie dazu auch: Automation Interface Unterstützung [▶ 136]	Die "Überprüfe alle Objekte"-Variante kann nicht implizit, sondern nur explizit über den Befehl ausgeführt werden. Zudem kann der Befehl per Automation Interface aufgerufen werden. Sehen Sie dazu auch: Automation Interface Unterstützung [▶ 136]

5.3 Befehl 'Standard-Metriken anzeigen'

Symbol:

Funktion: Der Befehl startet die statische Metrik-Codeanalyse für das gerade aktive SPS-Projekt und stellt die Metriken für die genutzten Programmierbausteine in einer Tabelle dar.

Aufruf: Menü **Erstellen** oder Kontextmenü des SPS-Projektobjekts

Der Befehl startet für das ausgewählte SPS-Projekt zunächst die Codegenerierung (wie beispielsweise bei Befehl **Projekt erstellen**). In einer tabellarischen Ansicht **Standard-Metriken** zeigt TwinCAT dann für jeden genutzten Programmierbaustein die gewünschten Metriken (Kennzahlen). Die Metriken, die angezeigt werden sollen, werden in den Projekteigenschaften aktiviert (siehe Konfiguration der Metriken [▶ 97]).

Liegt ein Wert außerhalb des Bereichs, der in der Konfiguration durch eine Unter- und Obergrenze definiert ist, erscheint das Tabellenfeld rot hinterlegt.

Mit einem Mausklick auf eine Spaltenüberschrift können Sie die Tabelle gemäß der Spalte sortieren.

Bitte beachten Sie, dass das selektierte SPS-Projekt vor der Durchführung dieses Befehls erstellt wird, und die Erstellung der Standard-Metriken nur gestartet wird, wenn die Codegenerierung erfolgreich war, d.h. wenn der Compiler keine Kompilierfehler festgestellt hat.

Bitte beachten Sie auch den Befehl 'Standard-Metriken anzeigen [Überprüfe alle Objekte]' [▶ 120] und die Unterschiede zwischen den beiden Befehlen, welche in der folgenden Tabelle beschrieben sind.

Unterschiede	**Befehl 'Standard-Metriken anzeigen'**	**Befehl 'Standard-Metriken anzeigen [Überprüfe alle Objekte]'**
Geltungsbereich	Die Standard-Metriken werden für die Objekte erstellt, die in dem SPS-Projekt verwendet werden. Die ungenutzten Objekte werden bei diesem Befehl nicht beachtet. Der Geltungsbereich dieses Befehls deckt sich somit mit den Erstellungsbefehlen **Projekt/ Projektmappe erstellen** bzw. **neu erstellen**. Falls Sie die Standard-Metriken auch für die ungenutzten Objekte erstellen lassen möchten, was z.B. bei der Bearbeitung von Bibliotheksprojekten nützlich ist, können Sie den **Befehl 'Standard-Metriken anzeigen [Überprüfe alle Objekte]'** verwenden.	Die Standard-Metriken werden für alle Objekte erstellt, die sich im Projektbaum des SPS-Projekts befinden. Dies ist in erster Linie bei der Erstellung von Bibliotheken bzw. bei der Bearbeitung von Bibliotheksprojekten nützlich. Der Geltungsbereich dieses Befehls deckt sich somit mit dem Erstellungsbefehl **Überprüfe alle Objekte**.

5.3.1 Befehle im Kontextmenü der Ansicht 'Standard-Metriken'

Über einen Rechtsklick in der Ansicht **Standard-Metriken** gelangen Sie zum Kontextmenü, welches einige Befehle zur Verfügung stellt.

Über das Kontextmenü können Sie die Tabelle der Metriken aktualisieren, drucken, exportieren und in die Zwischenablage kopieren. Außerdem gelangen Sie über das Kontextmenü in eine Ansicht, um die Metriken – genau wie in den SPS-Projekteigenschaften – zu konfigurieren. Des Weiteren können Sie für die ausgewählten Bausteine ein Kiviat-Diagramm erzeugen oder den Baustein im entsprechenden Editor öffnen. Voraussetzung für die Generierung eines Kiviat-Diagramms ist, dass mindestens drei Metriken mit einem definierten Wertebereich konfiguriert sind (Unter- und Obergrenze).

Die Befehle dazu lauten wie folgt:

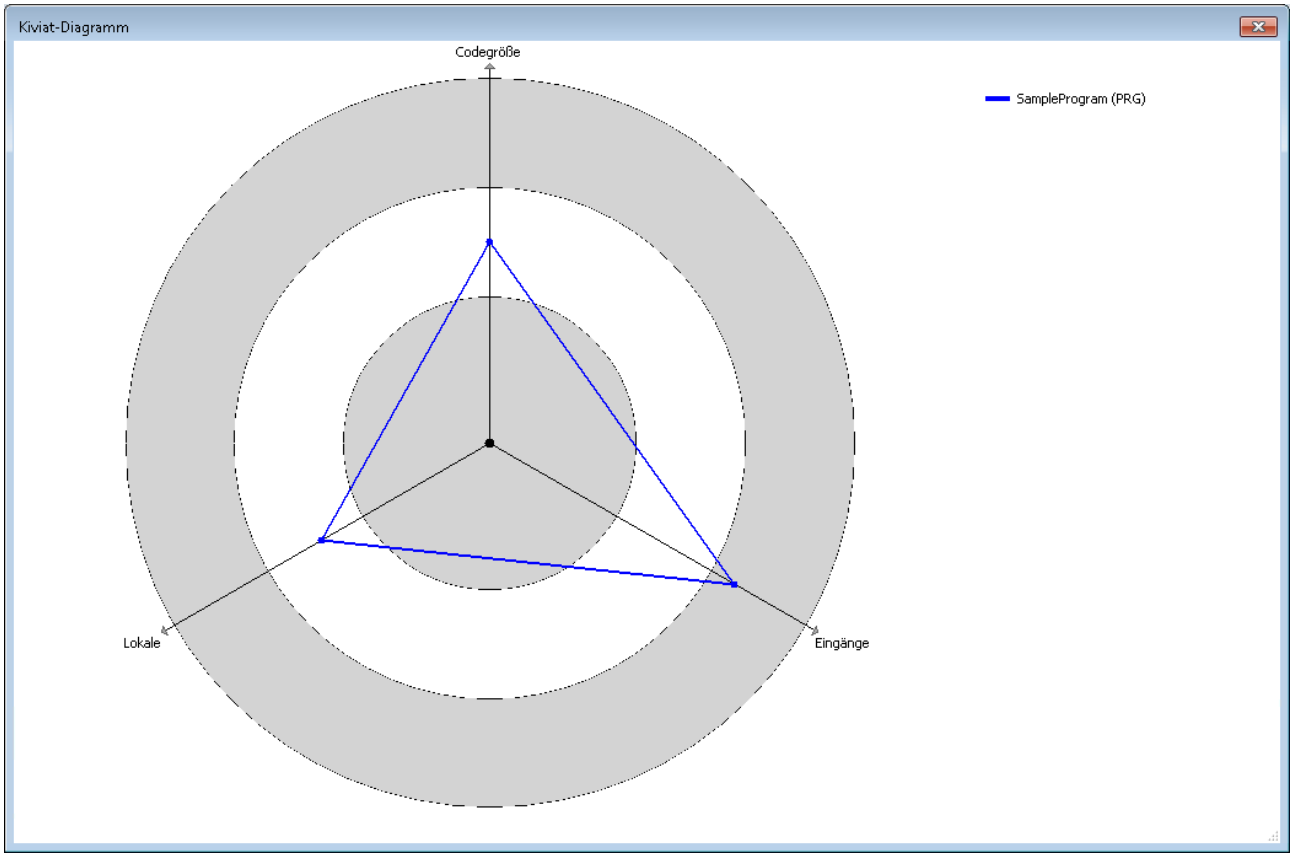
- **Berechnen**: Die Werte werden aktualisiert.
- **Tabelle drucken**: Der Standarddialog zum Einrichten des Druckauftrags erscheint.
- **Tabelle kopieren**: Die Daten werden mit Tabstopps getrennt in die Zwischenablage kopiert. Von dort können Sie die Tabelle dann beispielsweise direkt in eine Tabellenkalkulation oder Textverarbeitung einfügen.
- **Tabelle exportieren**: Die Daten werden als durch Strichpunkt getrennte Werte in eine Textdatei (*.csv) exportiert.
- **Kiviat-Diagramm**: Für den ausgewählten Baustein wird ein Spinnennetzdiagramm erstellt. Dies ist eine grafische Darstellung für die Bausteine, für die Metriken eine Unter- und Obergrenze definieren. Damit können Sie visualisieren, wie gut der Code der Programmiereinheit bezüglich einer vorgegebenen Norm ist.
  Jede Metrik wird in einem Kreis als Achse dargestellt, die ausgehend vom Mittelpunkt (Wert 0) nach außen drei Ringzonen des Kreises durchläuft. Die innere Ringzone repräsentiert den Wertebereich

unterhalb der für die Metrik definierten Untergrenze, die äußere den oberhalb der Obergrenze. Die Achsen der betroffenen Metriken sind gleichmäßig auf den Kreis verteilt.
Die aktuellen Werte der einzelnen Metriken auf ihren Achsen werden mit Linien verbunden und idealerweise liegt die Gesamtlinie in der mittleren Kreiszone.

Voraussetzung für die Nutzung eines Kiviat-Diagramms

Es müssen mindestens drei Metriken mit einem definierten Wertebereich konfiguriert sein.

Sehen Sie in der folgenden Abbildung ein Beispiel für ein Diagramm für 3 Metriken mit definiertem Wertebereich (Name der Metrik jeweils am Ende der Achse, Name des Bausteins rechts oben):



- **Konfigurieren**: Es öffnet sich eine Tabelle, in der die Metriken konfiguriert werden können. Die Ansicht, Funktionsweise und Einstellungen entsprechen der Metriken-Konfiguration [▶ 97] in den SPS-Projekteigenschaften. Wenn Sie in dieser Tabelle eine Änderung vornehmen, wird diese automatisch in die SPS-Projekteigenschaften übernommen.
- **POU öffnen**: Der Programmierbaustein wird im entsprechenden Editor geöffnet.

5.4 Befehl 'Standard-Metriken anzeigen [Überprüfe alle Objekte]'

Symbol:

Funktion: Der Befehl startet die statische Metrik-Codeanalyse für das gerade aktive SPS-Projekt und stellt die Metriken für alle Programmierbausteine in einer Tabelle dar.

Aufruf: Menü **Erstellen** oder Kontextmenü des SPS-Projektobjekts

Der Befehl startet für das ausgewählte SPS-Projekt zunächst die Codegenerierung (wie beispielsweise bei Befehl **Projekt erstellen**). In einer tabellarischen Ansicht **Standard-Metriken** zeigt TwinCAT dann für jeden Programmierbaustein die gewünschten Metriken (Kennzahlen). Die Metriken, die angezeigt werden sollen, werden in den Projekteigenschaften aktiviert (siehe Konfiguration der Metriken [▶ 97]).

Liegt ein Wert außerhalb des Bereichs, der in der Konfiguration durch eine Unter- und Obergrenze definiert ist, erscheint das Tabellenfeld rot hinterlegt.

Mit einem Mausklick auf eine Spaltenüberschrift können Sie die Tabelle gemäß der Spalte sortieren.

Bitte beachten Sie, dass das selektierte SPS-Projekt vor der Durchführung dieses Befehls erstellt wird, und die Erstellung der Standard-Metriken nur gestartet wird, wenn die Codegenerierung erfolgreich war, d.h. wenn der Compiler keine Kompilierfehler festgestellt hat.

Berechnung der Metrik „Codegröße“ über diesen Befehl nicht möglich

Die Berechnung der Metrik Codegröße [Anzahl Bytes] [▶ 99] ist lediglich über den Befehl 'Standard-Metriken anzeigen' [▶ 118] möglich. Bei Ausführung des Befehls **Standard-Metriken anzeigen [Überprüfe alle Objekte]** bleibt das Feld **Codegröße** leer.

Bitte beachten Sie auch den Befehl 'Standard-Metriken anzeigen' [▶ 118] und die Unterschiede zwischen den beiden Befehlen, welche in der folgenden Tabelle beschrieben sind.

Unterschiede	**Befehl 'Standard-Metriken anzeigen'**	**Befehl 'Standard-Metriken anzeigen [Überprüfe alle Objekte]'**
Geltungsbereich	Die Standard-Metriken werden für die Objekte erstellt, die in dem SPS-Projekt verwendet werden. Die ungenutzten Objekte werden bei diesem Befehl nicht beachtet. Der Geltungsbereich dieses Befehls deckt sich somit mit den Erstellungsbefehlen **Projekt/ Projektmappe erstellen** bzw. **neu erstellen**. Falls Sie die Standard-Metriken auch für die ungenutzten Objekte erstellen lassen möchten, was z.B. bei der Bearbeitung von Bibliotheksprojekten nützlich ist, können Sie den **Befehl 'Standard-Metriken anzeigen [Überprüfe alle Objekte]'** verwenden.	Die Standard-Metriken werden für alle Objekte erstellt, die sich im Projektbaum des SPS-Projekts befinden. Dies ist in erster Linie bei der Erstellung von Bibliotheken bzw. bei der Bearbeitung von Bibliotheksprojekten nützlich. Der Geltungsbereich dieses Befehls deckt sich somit mit dem Erstellungsbefehl **Überprüfe alle Objekte**.

5.4.1 Befehle im Kontextmenü der Ansicht 'Standard-Metriken'

Über einen Rechtsklick in der Ansicht **Standard-Metriken** gelangen Sie zum Kontextmenü, welches einige Befehle zur Verfügung stellt.

Über das Kontextmenü können Sie die Tabelle der Metriken aktualisieren, drucken, exportieren und in die Zwischenablage kopieren. Außerdem gelangen Sie über das Kontextmenü in eine Ansicht, um die Metriken – genau wie in den SPS-Projekteigenschaften – zu konfigurieren. Des Weiteren können Sie für die ausgewählten Bausteine ein Kiviat-Diagramm erzeugen oder den Baustein im entsprechenden Editor öffnen. Voraussetzung für die Generierung eines Kiviat-Diagramms ist, dass mindestens drei Metriken mit einem definierten Wertebereich konfiguriert sind (Unter- und Obergrenze).

Die Befehle dazu lauten wie folgt:

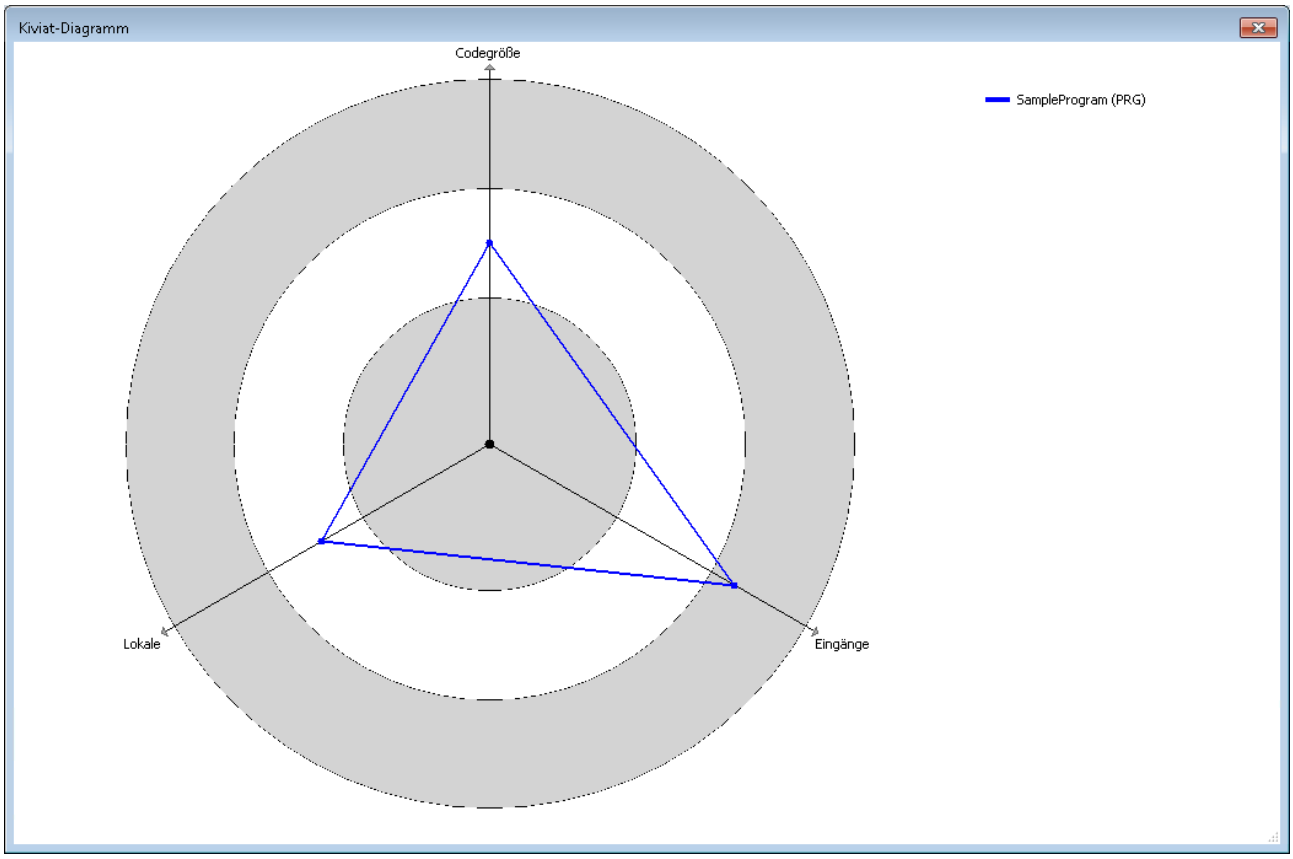
- **Berechnen**: Die Werte werden aktualisiert.
- **Tabelle drucken**: Der Standarddialog zum Einrichten des Druckauftrags erscheint.

- **Tabelle kopieren**: Die Daten werden mit Tabstopps getrennt in die Zwischenablage kopiert. Von dort können Sie die Tabelle dann beispielsweise direkt in eine Tabellenkalkulation oder Textverarbeitung einfügen.
- **Tabelle exportieren**: Die Daten werden als durch Strichpunkt getrennte Werte in eine Textdatei (*.csv) exportiert.
- **Kiviat-Diagramm**: Für den ausgewählten Baustein wird ein Spinnennetzdiagramm erstellt. Dies ist eine grafische Darstellung für die Bausteine, für die Metriken eine Unter- und Obergrenze definieren. Damit können Sie visualisieren, wie gut der Code der Programmiereinheit bezüglich einer vorgegebenen Norm ist.
  Jede Metrik wird in einem Kreis als Achse dargestellt, die ausgehend vom Mittelpunkt (Wert 0) nach außen drei Ringzonen des Kreises durchläuft. Die innere Ringzone repräsentiert den Wertebereich unterhalb der für die Metrik definierten Untergrenze, die äußere den oberhalb der Obergrenze. Die Achsen der betroffenen Metriken sind gleichmäßig auf den Kreis verteilt.
  Die aktuellen Werte der einzelnen Metriken auf ihren Achsen werden mit Linien verbunden und idealerweise liegt die Gesamtlinie in der mittleren Kreiszone.

Voraussetzung für die Nutzung eines Kiviat-Diagramms

Es müssen mindestens drei Metriken mit einem definierten Wertebereich konfiguriert sein.

Sehen Sie in der folgenden Abbildung ein Beispiel für ein Diagramm für 3 Metriken mit definiertem Wertebereich (Name der Metrik jeweils am Ende der Achse, Name des Bausteins rechts oben):



- **Konfigurieren**: Es öffnet sich eine Tabelle, in der die Metriken konfiguriert werden können. Die Ansicht, Funktionsweise und Einstellungen entsprechen der Metriken-Konfiguration [▶ 97] in den SPS-Projekteigenschaften. Wenn Sie in dieser Tabelle eine Änderung vornehmen, wird diese automatisch in die SPS-Projekteigenschaften übernommen.
- **POU öffnen**: Der Programmierbaustein wird im entsprechenden Editor geöffnet.

5.5 Befehl 'Werte der Konstantenpropagation für aktuellen Editor anzeigen'

Verfügbar ab TwinCAT 3.1.4026.14

Symbol:

Funktion: Der Befehl startet die statische Codeanalyse und berechnet eine Maßzahl für die Konstantenpropagation des Codes im aktuellen Editors. Der sich öffnende Dialog visualisiert das Ergebnis. Der analysierte Code wird gelistet und die ermittelten Maßzahlen werden angezeigt.

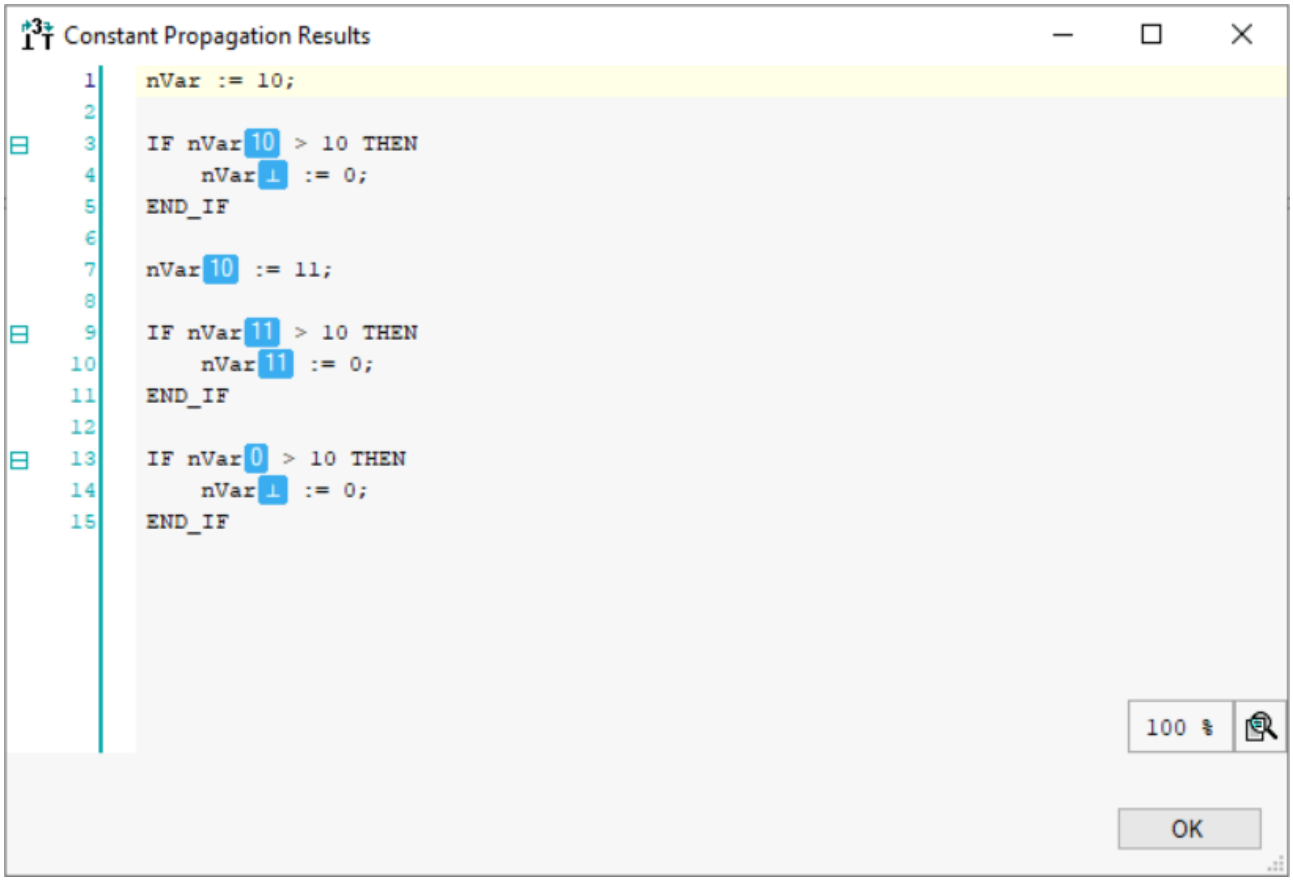
Aufruf: Menü **Erstellen** oder Kontextmenü des ST-Editors

Voraussetzung: Im Editor ist ein Programmierobjekt in der Implementierungssprache ST geöffnet.

Für weitere Informationen siehe: Konstantenpropagation [▶ 130]

Dialog: Ergebnisse der Konstantenpropagation

Beispiel:



5.6 Befehl 'Kognitive Komplexität für aktuellen Editor anzeigen'

Verfügbar ab TwinCAT 3.1.4026.14

Symbol:

Funktion: Der Befehl startet die statische Codeanalyse und berechnet ein Inkrement für die kognitive Komplexität des Codes im aktuellen Editor. Der sich öffnende Dialog visualisiert das Ergebnis und gibt im Titel die aufsummierte Maßzahl an. Der analysierte Code wird gelistet und mit den erkannten Komplexitäten angezeigt.

Aufruf: Menü **Erstellen** oder Kontextmenü des ST-Editors

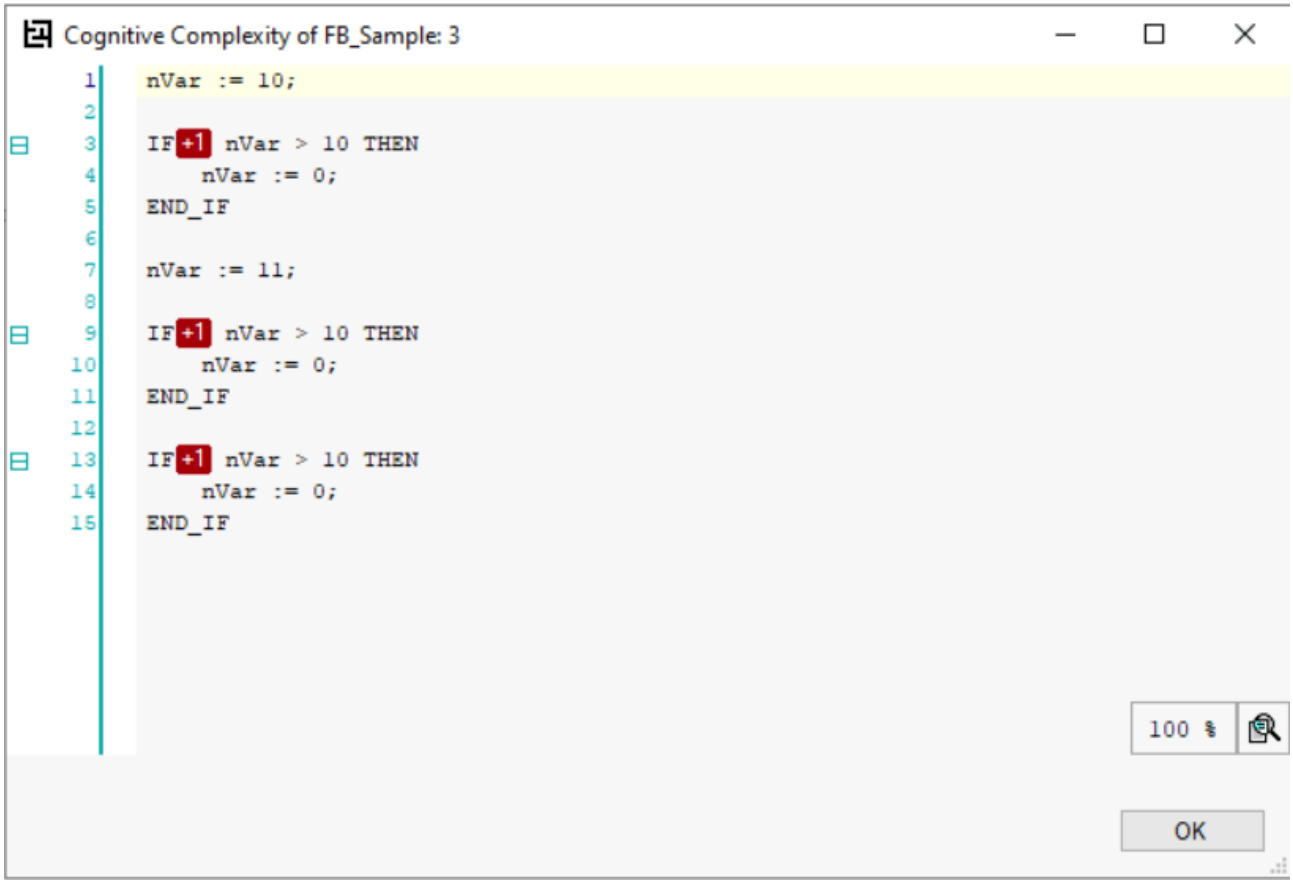
Voraussetzung: Im Editor ist ein Programmierobjekt in der Implementierungssprache ST geöffnet.

Für weitere Informationen sehen Sie auch die Dokumentation der Metrik Kognitive Komplexität [▶ 105].

Dialog: Kognitive Komplexität

Beispiel:

„Kognitive Komplexität von FB_Sample: 3“



6 Pragmas und Attribute

Ein Pragma und verschiedene Attribute sind verfügbar, um beispielsweise einzelne Regeln oder Namenskonventionen für die Statische Analyse temporär auszuschalten, also um bestimmte Codezeilen oder Programmeinheiten aus der Prüfung auszuklammern.

Voraussetzung: Sie haben die Regeln oder Konventionen in den SPS-Projekteigenschaften aktiviert bzw. definiert. Sehen Sie hierzu:

- Regeln [▶ 17]
- Namenskonventionen [▶ 83]

Attribute werden im Deklarationsteil eines Programmierbausteins eingefügt, um bestimmte Regeln für ein gesamtes Programmierobjekt abzuschalten.

Pragmas werden im Implementierungsteil eines Programmierbausteins verwendet, um bestimmte Regeln für einzelne Codezeilen abzuschalten. Ausnahme ist die Regel SA0164, die auch im Deklarationsteil per Pragma abgeschaltet werden kann.

Regeln, die in den Projekteigenschaften deaktiviert sind, können Sie auch nicht über ein Pragma oder Attribut aktivieren.

Regel SA0004 kann nicht über ein Pragma oder Attribut deaktiviert werden.

Pragmas im Implementierungseditor

Wenn Sie ein Pragma im Implementierungseditor verwenden möchten, ist dies aktuell im ST- sowie im FUP/KOP/AWL-Editor möglich.

In FUP/KOP/AWL muss das gewünschte Pragma in eine Sprungmarke eingetragen werden.

Sehen Sie im Folgenden eine Übersicht sowie eine detaillierte Beschreibung der verfügbaren Pragmas und Attribute.

Übersicht

- Pragma {analysis ...} [▶ 126]
  - zur Ausschaltung von Kodierregeln im Implementierungsteil
  - für einzelne Codezeilen anwendbar
- Attribut {attribute 'no-analysis'} [▶ 126]
  - zum Ausschluss von Programmierobjekten (z.B. POU, GVL, DUT) von der Statischen Analyse (Kodierregeln, Namenskonventionen, unzulässige Symbole)
  - nur für ganze Programmierobjekte anwendbar
- Attribut {attribute 'analysis' := '...'} [▶ 127]
  - zur Ausschaltung von Kodierregeln im Deklarationsteil
  - für einzelne Deklarationen oder für ganze Programmierobjekte anwendbar
- Attribut {attribute 'naming' := '...'} [▶ 127]
  - zur Ausschaltung von Namenskonventionen im Deklarationsteil
  - für einzelne Deklarationen oder für ganze Programmierobjekte anwendbar
- Attribut {attribute 'nameprefix' := '...'} [▶ 128]
  - zur Definition von Präfixen für Instanzen eines strukturierten Datentyps
  - im Deklarationsteil eines strukturierten Datentyps anwendbar
- Attribut {attribute 'analysis:report-multiple-instance-calls'} [▶ 129]

- zur Festlegung, dass eine Funktionsbaustein-Instanz nur einmal aufgerufen werden soll
- im Deklarationsteil eines Funktionsbausteins anwendbar

Detaillierte Beschreibung

Pragma {analysis ...}

Das Pragma {analysis -/+<Regelnummer>} können Sie im Implementierungsteil eines Programmierbausteins verwenden, um einzelne Kodierregeln für die nachfolgenden Codezeilen auszuschalten. Sie deaktivieren Codierregeln durch die Angabe der Regelnummern und einem vorangestellten Minuszeichen (“-”). Zur Aktivierung wird ein Pluszeichen (“+”) vorangestellt. Mit Hilfe einer Kommaseparierung können Sie im Pragma beliebig viele Regeln angeben.

Einfügeort:

- Deaktivierung von Regeln: Im Implementierungsteil vor der ersten Codezeile, ab der die Codeanalyse deaktiviert wird, mit {analysis - ...}.
- Aktivierung von Regeln: Nach der letzten Zeile der Deaktivierung mit {analysis + ...}.
- Für die Regel SA0164 kann das Pragma auch im Deklarationsteil vor einem Kommentar eingefügt werden.

Syntax:

- Deaktivierung von Regeln:
  - eine Regel: {analysis -<Regelnummer>}
  - mehrere Regeln: {analysis -<Regelnummer>, -<weitere Regelnummer>, -<weitere Regelnummer>}
- Aktivierung von Regeln:
  - eine Regel: {analysis +<Regelnummer>}
  - mehrere Regeln: {analysis +<Regelnummer>, +<weitere Regelnummer>, +<weitere Regelnummer>}

Beispiele:

Sie möchten Regel 24 (nur getypte Literale erlaubt) für eine Zeile deaktivieren (d.h. es ist in diesen Zeilen nicht nötig, "nTest := DINT#99" zu schreiben) und danach wieder aktivieren:

```
{analysis -24}
nTest := 99;
{analysis +24}
nVar := INT#2;
```

Angabe mehrerer Regeln:

```
{analysis -10, -24, -18}
```

Attribut {attribute 'no-analysis'}

Das Attribut {attribute 'no-analysis'} können Sie verwenden, um ein gesamtes Programmierobjekt von der Prüfung durch die Statische Analyse auszuschließen. Für dieses Programmierobjekt wird die Prüfung der Kodierregeln, der Namenskonventionen und der unzulässigen Symbole nicht durchgeführt.

Einfügeort:

oberhalb der Deklaration eines Programmierobjekts

Syntax:

{attribute 'no-analysis'}

Beispiele:

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

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

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

Das Attribut {attribute 'analysis' := '-<Regelnummer>'} können Sie verwenden, um bestimmte Regeln für einzelne Deklarationen oder für ein ganzes Programmierobjekt abzuschalten. Sie deaktivieren die Kodierregel durch die Angabe der Regelnnummer(n) und einem vorangestellten Minuszeichen. Sie können im Attribut beliebig viele Regeln angeben.

Einfügeort:

oberhalb der Deklaration eines Programmierobjekts oder in der Zeile oberhalb einer Variablendeklaration

Syntax:

- eine Regel: {attribute 'analysis' := '-<Regelnummer>'}
- mehrere Regeln: {attribute 'analysis' := '-<Regelnummer>, -<weitere Regelnummer>, -<weitere Regelnummer>'}

Beispiele:

Sie möchten Regel 33 (Nicht verwendete Variablen) für alle Variablen der Struktur ausschalten.

```
{attribute 'analysis' := '-33'}
TYPE ST_Sample :
STRUCT
    bMember  : BOOL;
    nMember  : INT;
END_STRUCT
END_TYPE
```

Sie möchten die Prüfung von Regel 28 (Überlappende Speicherbereiche) und von Regel 33 (Nicht verwendete Variablen) für die Variable nVar1 ausschalten.

```
PROGRAM MAIN
VAR
    {attribute 'analysis' := '-28, -33'}
    nVar1 AT%QB21  : INT;
    nVar2 AT%QD5   : DWORD;

    nVar3 AT%QB41  : INT;
    nVar4 AT%QD10  : DWORD;
END_VAR
```

Sie möchten Regel 6 (Gleichzeitiger Zugriff) für eine globale Variable ausschalten, sodass keine Fehlermeldung generiert wird, wenn die Variable von mehr als einer Task geschrieben wird.

```
VAR_GLOBAL
    {attribute 'analysis' := '-6'}
    nVar  : INT;
    bVar  : BOOL;
END_VAR
```

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

Das Attribut {attribute 'naming' := '...'} können Sie im Deklarationsteil verwenden, um einzelne Deklarationszeilen von der Prüfung auf Einhaltung der aktuell gültigen Namenskonventionen auszuschließen.

Einfügeort:

- Deaktivierung: im Deklarationsteil oberhalb der betreffenden Zeilen
- Aktivierung: Nach der letzten Zeile der Deaktivierung

Syntax:

{attribute 'naming' := '<off|on|omit>'}

- off, on: die Prüfung wird für alle Zeilen zwischen der "off" und der "on" Anweisung ausgeschaltet
- omit: nur die nachfolgende Zeile wird von der Prüfung ausgenommen

Beispiel:

Angenommen wird, dass folgende Namenskonventionen definiert sind:

- Die Kennzeichner von INT-Variablen müssen mit einem Präfix "n" versehen sein (Namenskonvention NC0014), beispielsweise "nVar1".
- Funktionsbausteinnamen müssen mit "FB_" beginnen (Namenskonvention NC0103), beispielsweise "FB_Sample".

Für den unten dargestellte Code gibt die statische Analyse dann nur Meldungen zu folgenden Variablen aus: 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;
    …
```

Attribut {attribute 'nameprefix' := '...'}

Das Attribut {attribute 'nameprefix' := '...'} definiert ein Präfix für Variablen eines strukturierten Datentyps. Dann gilt die Namenskonvention, dass Bezeichner von Instanzen dieses Typs dieses Präfix besitzen müssen.

Einfügeort:

oberhalb der Deklaration eines strukturierten Datentyps

Syntax:

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

Beispiel:

In der Kategorie Namenskonventionen [▶ 83] der SPS-Projekteigenschaften sind die folgenden Namenskonventionen definiert:

- Variablen vom Typ einer Struktur (NC0032): st
- Strukturen (NC0151): ST_

Variablen des Typs "ST_Point" sollen hingegen nicht mit dem Präfix "st", sondern mit dem Präfix "pt" beginnen.

Im folgenden Beispiel wird die Statische Analyse eine Meldung für "a1" und "st1" vom Typ "ST_Point" ausgeben, weil die Variablennamen nicht mit "pt" beginnen. Für die Variablen vom Typ "ST_Test" wird hingegen der Präfix "st" erwartet.

```
TYPE ST_Test :
STRUCT
    …
END_STRUCT
END_TYPE
```

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

```
PROGRAM MAIN
VAR
    a1   : ST_Point;     // => Invalid variable name 'a1'. Expect prefix 'pt'
    st1  : ST_Point;     // => Invalid variable name 'st1'. Expect prefix 'pt'
    pt1  : ST_Point;

    a2   : ST_Test;      // => Invalid variable name 'a2'. Expect prefix 'st'
    st2  : ST_Test;
    pt2  : ST_Test;      // => Invalid variable name 'st2'. Expect prefix 'st'
END_VAR
```

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

Das Attribut {attribute 'analysis:report-multiple-instance-calls'} kennzeichnet einen Funktionsbaustein für eine Prüfung auf Regel 105: Nur bei Funktionsbausteinen mit diesem Attribut wird geprüft, ob die Instanzen des Funktionsbausteins mehrfach aufgerufen werden. Wenn die Regel 105 in der Kategorie Regeln [▶ 17] in den SPS-Projekteigenschaften deaktiviert ist, hat das Attribut keine Auswirkung.

Einfügeort:

oberhalb der Deklaration eines Funktionsbausteins

Syntax:

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

Beispiel:

Im folgenden Beispiel wird die Statische Analyse einen Fehler für fb2 ausgeben, weil die Instanz mehr als einmal aufgerufen wird.

Funktionsbaustein FB_Test1 ohne Attribut:

```
FUNCTION_BLOCK FB_Test1
…
```

Funktionsbaustein FB_Test2 mit Attribut:

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

Programm 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 Konstantenpropagation

Verfügbar ab TwinCAT 3.1.4026.14

Die statische Codeanalyse basiert auf Konstantenpropagation, deren Ergebnisse für verschiedene Prüfungen genutzt werden. So wird beispielsweise überprüft, ob Pointer ungleich 0 sind, oder ob Arrayindizes außerhalb des gültigen Bereichs liegen.

Sie können die statische Analyse effektiv unterstützen, allein wenn sie wissen, wie diese Analyse funktioniert und wo ihre Grenzen liegen.

Sehen Sie auch: Befehl 'Werte der Konstantenpropagation für aktuellen Editor anzeigen' [▶ 123]

Konstantenpropagation

Bei der statischen Analyse wird versucht, den Wert einer Variable anhand ihrer Verwendung zu bestimmen.

Beispiel:

```
PROGRAM MAIN
VAR
    x     : INT;
    bTest : BOOL;
END_VAR
```

```
x := 99;

IF x < 100 THEN
    bTest := TRUE;
END_IF
```

In der Implementierung in Zeile 1 zeichnet die Konstantenpropagation den Wert 99 für die Variable `x`, um diesen Wert für weitere Analysen zu verwenden. Die Analyse erkennt dann, dass der Ausdruck in der nachfolgenden IF-Anweisung konstant TRUE ist.

Lokal durchgeführte Konstantenpropagation

Ein Wert wird nur lokal im Baustein ermittelt. Es ist unerheblich, wie eine Eingabe übergeben wird. Auch die Ergebnisse von Funktionsaufrufen sind irrelevant.

Beispiel:

```
FUNCTION Func : BOOL
VAR_INPUT
    bTest : BOOL;
END_VAR
```

```
IF bTest THEN
    Func := OtherFunc(TRUE);
END_IF
```

Wenn der Parameter `bTest` bei jedem Aufruf auf TRUE gesetzt wird, hat dies keine Auswirkung auf die Konstantenpropagation. Auch wenn `OtherFunc(TRUE)` immer TRUE zurück gibt, hat dies keine Auswirkung auf die Konstantenpropagation.

Nur temporäre Variablen haben Initialwerte

Statische lokale Variablen in Programmen und Funktionsbausteinen haben keinen angenommenen Initialwert. Die Variablen behalten ihre Werte des letzten Aufrufes und können daher prinzipiell "alles" sein.

Lokale Variablen in Funktionen und temporäre Variablen haben einen Initialwert bei jedem Aufruf. Die Konstantenpropagation rechnet mit diesem Initialwert.

Beispiel:

```
PROGRAM MAIN
VAR
    x     : INT := 6;
    bTest : BOOL;
```

```
END_VAR
VAR_TEMP
    y     : INT := 8;
END_VAR
```

```
bTest := x < y;
```

Die Variable `y` wird bei jeder Ausführung von MAIN den Wert 8 haben. Die Variable `x` jedoch nicht unbedingt. Daher wird die Konstantenpropagation nur für `y` einen Wert annehmen, nicht aber für `x`.

Es empfiehlt sich, Variablen, die immer zuerst geschrieben und dann gelesen werden, als temporäre Variablen zu deklarieren.

Konstantenpropagation ermittelt Wertebereiche für numerische Datentypen

Um die Komplexität zu reduzieren, wird für jede Variable ein Wertebereich mit Ober- und Untergrenze ermittelt.

Beispiel:

```
PROGRAM MAIN
VAR
    x     : INT := 6;
    bTest : BOOL;
    y     : INT;
END_VAR
```

```
IF bTest THEN
    x := 1;
ELSE
    x := 100;
END_IF

IF x = 77 THEN
    y := 13;
END_IF
```

Hier wird für die Variable `x` der Wertebereich `[1..100]` ermittelt. Infolgedessen wird in Zeile 7 der Vergleich `x = 77` nicht als konstanter Ausdruck erkannt, da 77 innerhalb des Wertebereichs liegt.

Wiederkehrende komplexe Ausdrücke werden nicht als die gleiche Variable erkannt

Komplexe Ausdrücke haben unter Umständen keinen Wert zugeordnet. Wenn solche Ausdrücke mehrfach vorkommen, ist es hilfreich, eine Hilfsvariable einzuführen.

Beispiel:

```
PROGRAM MAIN
VAR
    x         : DINT;
    py        : POINTER TO INT;
    y         : INT;
    testArray : ARRAY [0..4] OF DINT;
END_VAR
```

```
IF py <> 0 THEN
    IF py^ >= 0 AND py^<= 4 THEN
        x := testArray[py^];
    END_IF

    y := py^;

    IF y <= 0 AND y <=4 THEN
        x := testArray[y];
    END_IF
END_IF
```

In Zeile 3 wird ein Fehler ausgegeben für einen möglichen Zugriff über Pointer auf einen Wert, obwohl der Bereich, auf den der Pointer zeigt, überprüft wird. Wird der Wert zuerst in eine lokale Variable kopiert und deren Bereich überprüft, dann kann die Konstantenpropagation den Wertebereich für diese Variable ermitteln und erlaubt den Zugriff in den Array in Zeile 9.

Verzweigungen

Bei Verzweigungen werden einzelne Zweige getrennt berechnet. Wertebereiche aus den einzelnen Bereichen werden anschließend zu einem neuen Wertebereich vereinigt.

Beispiel:

```
IF func(TRUE) THEN
    x := 1;
ELSE
    x := 10;
END_IF

IF func(FALSE) THEN
    y := x;
ELSE
    y := 2*x;
END_IF
```

In Zeile 6 hat `x` den Bereich `[1..10]`. Nach Zeile 11 hat `y` den Wertebereich `[1..20]`. Dies ergibt sich aus der Vereinigung der beiden Wertebereiche `[1..10]` und `[2..20]`.

Bedingungen

Beispiel:

Bedingungen können den Wertebereich einer Variablen in einem Codeblock einschränken. Mehrere Bedingungen können kombiniert werden. Einander ausschließende Bedingungen können auch zu einem leeren Wertebereich führen.

```
IF y > 0 AND y < 10 THEN
    x := y;
ELSE
    x:= 0;
END_IF

IF x < 0 THEN
    i := 99;
END_IF
```

`y` hat in Zeile 2 den Wertebereich [1..9]. Daraus ergibt sich für `x` in Zeile 6 der Wertebereich `[0..9]`. Kombiniert mit der Bedingung `x < 0` ergibt das in Zeile 9 für `x` eine leere Menge an möglichen Werten. Der Code ist nicht erreichbar. Die statische Analyse wird melden, dass die Bedingung `x < 0` an dieser Stelle immer FALSE ergibt.

Schleifen

Die Konstantenpropagation wird Schleifen im Code so lange ausführen, bis sich die Werte der Variablen in der Schleife nicht mehr ändern. Dabei wird angenommen, dass eine Schleife beliebig oft durchlaufen werden kann. Die bisher ermittelten Werte werden mit den vorhergehenden Werten vereint. Variablen, die innerhalb der Schleife geändert werden, haben einen sukzessiv wachsenden Bereich. Dabei nimmt die Konstantenpropagation nicht alle möglichen Werte für Bereiche an, sondern verwendet nur im Code vorkommende Grenzen und außerdem die Werte 0, 1, 2, 3 und 10, da diese häufig eine Rolle spielen.

Am einfachsten wird das Vorgehen an einem Beispiel deutlich.

Beispiel:

```
PROGRAM MAIN
VAR
    x : DINT;
    i : DINT;
    y : DINT;
END_VAR
```

```
x := 0;
y := 0;

FOR i := 0 TO 5 DO
    x := x + 1;
    y := i;
END_FOR
```

Die Konstantenpropagation weiß folgendes über die Schleife:

`i`, `x`, und `y` sind zu Beginn der ersten Ausführung der Schleife 0. Für den Code in der Schleife gilt die Bedingung `i <= 5`. Für den Code nach der Schleife gilt die Bedingung `i > 5`.

Für die Werte der Variablen in der Schleife ermittelt die Konstantenpropagation folgende Werte:

	i	x	y	
	`[0..5]`	`[0..MAXDINT]`	`[0..5]`	

Im Detail werden folgende Zwischenschritte durchlaufen:

Durchlauf	i	x	y	
1	`0`	`[0..1]`	`0`	`i` wurde mit 0 initialisiert, `y` bekommt immer die gleichen Werte wie `i`
2	`[0..1]`	`[0..2]`	`[0..1]`	
6	`[0..5]`	`[0..6]`	`[0..5]`	Zunächst wird tatsächlich der Bereich `[0..6]` für `i` berechnet. Es ist aber bekannt, dass `i < 5` eine Bedingung ist. Daher wird der Wert für den Code in der Schleife auf diesen Wert begrenzt.
7	`[0..5]`	`[0..7]`	`[0..5]`	
10	`[0..5]`	`[0..10]`	`[0..5]`	`x` wird immer weiter hochgezählt. Ab 10 wird allerdings der Wert auf `MAXDINT` "aufgerundet".
11	`[0..5]`	`[0..MAXDINT]`	`[0..5]`	`MAXDINT + 1` ergibt `MAXDINT`
ab 11				Ab dem 11. Durchlauf werden sich die Werte in der Schleife nicht mehr ändern. Die Propagation wird beendet.

Darüber hinaus gilt für den nach dieser Schleife folgenden Code: `i = 6`. Es wird in der Schleife der Bereich [0..6] ermittelt und dieser mit der Bedingung `i > 5` kombiniert, was exakt den Wert 6 ergibt.

8 QuickFix/Precompile

Verfügbar ab TwinCAT 3.1 Build 4026

Einige Regeln vom Static Analysis können bereits während der Vorkompilierung geprüft werden. Für das Auffinden solcher Regelverletzungen ist dabei keine explizite Ausführung der Statischen Analyse notwendig, sondern die Überprüfung findet bereits auf Basis der Precompile-Informationen während des Editierens statt. Die Überprüfung einer Regel während der Vorkompilierung findet nur statt, falls die Regel in den Einstellungen vom Static Analysis aktiviert ist.

Precompile: Unterschlängelung und Anzeige im Meldungsfenster

Wenn eine Regelverletzung auftritt, wird diese sofort durch Unterschlängelung im Deklarationseditor oder im ST-Editor angezeigt. Zusätzlich erscheinen – solange der Editor geöffnet ist – im Meldungsfenster in der Kategorie „IntelliSense“ Fehlermeldungen oder Warnungen. Diese enthalten vor der Regelnummer den Hinweis „(precompile)“.

QuickFix-Befehle

Darüber hinaus gibt es für einige Regeln, die während der Vorkompilierung geprüft werden können, im Deklarationseditor und im ST-Editor die Möglichkeit einer schnellen Fehlerbehebung (QuickFix). Sie können direkt an den betroffenen Codestellen eine automatische, unmittelbare Fehlerbehandlung ausführen. Zur schnellen Fehlerbehandlung gelangen Sie im Editor mit Klick auf den unterschlängelten Code und dann mit Klick auf das Glühbirnensymbol.

Je nach Fehler werden die folgenden QuickFix-Befehle angeboten:

- Fehlermeldung/Warnung ignorieren:
  Der Befehl bewirkt, dass automatisch Pragmas oder Attribute in den Code eingefügt werden, die eine Prüfung der dazugehörigen Regel für diese Codezeile ausschließen.
- Fehlermeldung/Warnung global für die POU ignorieren:
  Der Befehl bewirkt, dass automatisch ein Attribut an den Beginn des Deklarationsteils des Programmierobjekts eingefügt wird. Dann wird eine Prüfung der dazugehörigen Regel für dieses Programmierobjekt ausgeschlossen.
- Prüfung ausschalten:
  Der Befehl bewirkt, dass die Überprüfung der dazugehörigen Regel in den Einstellungen deaktiviert wird.
- Fehler durch Vorschlag zur Änderung des ST-Codes beheben:
  Beispiel für „SA0033: Nicht verwendete Variablen“: Die Deklaration der nicht verwendeten Variablen wird aus dem Deklarationseditor entfernt.

Verfügbare Regeln

Nicht verfügbar:

Bitte beachten Sie, dass die folgenden Regeln **nicht** während der Vorkompilierung geprüft werden können.

- SA0004
- SA0006
- SA0013
- SA0016
- SA0027
- SA0028
- SA0042
- SA0100
- SA0103
- SA0105

- SA0150
- SA0160
- SA0161
- SA0175

Verfügbar:

Alle anderen Regeln werden auf Basis der Precompile-Informationen überprüft.

9 Automation Interface Unterstützung

Das Static Analysis kann teilweise über das Automation Interface (AI) bedient werden. Die AI-Unterstützung umfasst folgende Befehle/Aktionen:

- Explizite Durchführung der Statischen Analyse per Automation Interface [▶ 136]
- Implizite Durchführung der Statischen Analyse per Automation Interface [▶ 136]
- Einstellungen/Konfiguration per Automation Interface speichern [▶ 136]
- Einstellungen/Konfiguration per Automation Interface laden [▶ 137]
- Metriken exportieren [▶ 137]

Bitte beachten Sie hierzu auch die Automation Interface Dokumentation:
Produktbeschreibung

Explizite Durchführung der Statischen Analyse per Automation Interface

Die beiden folgenden Befehle können explizit per Automation Interface aufgerufen werden:

- Befehl 'Statische Analyse durchführen' [▶ 115]
- Befehl 'Statische Analyse durchführen [Überprüfe alle Objekte]' [▶ 117]

Für die Methode `RunStaticAnalysis()` kann bCheckAll als optionaler Parameter angegeben werden. Die Methode kann aber auch parameterlos aufgerufen werden.

Parameter	Aufruf
RunStaticAnalysis()	Ausführung des Befehls **Statische Analyse durchführen [Überprüfe alle Objekte]**
RunStaticAnalysis(bCheckAll = TRUE)	
RunStaticAnalysis(bCheckAll = FALSE)	Ausführung des Befehls **Statische Analyse durchführen**

PowerShell-Beispiel:

```
$p = $sysMan.LookupTreeItem("TIPC^MyPlcProject^MyPlcProject Project")
$p.RunStaticAnalysis()
```

C#-Beispiel für TC3.1 Version >= Build 4024:

```
ITcPlcIECProject3 plcIec3 = sysMan.LookupTreeItem("TIPC^Untitled1^Untitled1
Project") as ITcPlcIECProject3;
plcIec3.RunStaticAnalysis();
```

C#-Beispiel für TC3.1 Version >= Build 4026:

```
ITcPlcIECProject4 plcIec4 = sysMan.LookupTreeItem("TIPC^Untitled1^Untitled1
Project") as ITcPlcIECProject4;
plcIec4.RunStaticAnalysis();
```

Implizite Durchführung der Statischen Analyse per Automation Interface

Alternativ besteht die Möglichkeit, die Einstellung [▶ 14] **Statische Analyse automatisch durchführen** zu aktivieren und das Projekt per Automation Interface zu erstellen, sodass bei diesem Erstellungsvorgang implizit die Statische Analyse durchgeführt wird.

Einstellungen/Konfiguration per Automation Interface speichern

Verfügbar ab TwinCAT 3.1 Build 4026

Die Einstellungen [▶ 14] vom Static Analysis können per Automation Interface in eine *.csa-Datei gespeichert bzw. exportiert werden.

Für die Methode `SaveStaticAnalysisSettings(string bstrFilename)` muss der Zielpfad der Datei als Übergabeparameter angegeben werden.

> Die Methode `RunStaticAnalysis` ist ab dem Interface `ITcPlcIECProject3` verfügbar. Die Methoden `SaveStaticAnalysisSettings` und `LoadStaticAnalysisSettings` werden ab dem Interface `ITcPlcIECProject4` angeboten.

C#-Beispiel für TC3.1 Version >= Build 4026:

```
// Path to the location to export the SAN configuration
string saveCsaPath = @"C:\Users\UserName\Desktop\SaveTest.csa";
[…]
// Navigate to PLC project
ITcPlcIECProject4 plcIec4 = sysMan.LookupTreeItem("TIPC^Untitled1^Untitled1
Project") as ITcPlcIECProject4;
// Save SAN configuration
plcIec4.SaveStaticAnalysisSettings(saveCsaPath);
```

Einstellungen/Konfiguration per Automation Interface laden

> Verfügbar ab TwinCAT 3.1 Build 4026

Eine vorgefertigte Static Analysis Konfiguration (*.csa-Datei) kann per Automation Interface ins PLC-Projekt geladen werden. Die hierdurch geladenen Einstellungen [▶ 14] können anschließend per AI überprüft werden, indem das Static Analysis ausgeführt wird (s.o.).

Für die Methode `LoadStaticAnalysisSettings(string bstrFilename)` muss der Pfad der zu ladenden Datei als Übergabeparameter angegeben werden.

> Die Methode `RunStaticAnalysis` ist ab dem Interface `ITcPlcIECProject3` verfügbar. Die Methoden `SaveStaticAnalysisSettings` und `LoadStaticAnalysisSettings` werden ab dem Interface `ITcPlcIECProject4` angeboten.

C#-Beispiel für TC3.1 Version >= Build 4026:

```
// Path to load a SAN configuration
string loadCsaPath = @"C:\Users\UserName\Desktop\LoadTest.csa";
[…]
// Navigate to PLC project
ITcPlcIECProject4 plcIec4 = sysMan.LookupTreeItem("TIPC^Untitled1^Untitled1
Project") as ITcPlcIECProject4;
// Load SAN configuration
plcIec4.LoadStaticAnalysisSettings(loadCsaPath);
// Optionally run SAN afterwards
plcIec4.RunStaticAnalysis();
```

Metriken exportieren

> Verfügbar ab TwinCAT 3.1 Build 4026.4

Die Standard-Metriken können per Automation Interface in eine Textdatei (*.csv) exportiert werden. Dabei findet implizit eine aktuelle Berechnung der Metriken statt. Eine manuelle Ausführung dieses Vorgangs würde die beiden folgenden Befehle beinhalten:

- Befehl 'Standard-Metriken anzeigen' [▶ 118]
- Befehl **Tabelle exportieren**, siehe Befehle im Kontextmenü der Ansicht 'Standard-Metriken' [▶ 119]

Für die Methode `ExportStandardMetrics(string bstrFilename)` muss der Pfad, an dem die Exportdatei gespeichert werden soll, als Übergabeparameter angegeben werden.

Die Methode `ExportStandardMetrics` ist ab dem Interface `ITcPlcIECProject5` verfügbar.

C#-Beispiel für TC3.1 Version >= Build 4026.4:

```
// Path to save the csv file
string savePath = @"C:\Users\UserName\Desktop\Metrics.csv";
[…]
// Navigate to PLC project
ITcPlcIECProject5 plcIec5 = sysMan.LookupTreeItem("TIPC^Untitled1^Untitled1
Project") as ITcPlcIECProject5;
// Export standard metrics
plcIec5.ExportStandardMetrics(savePath);
```

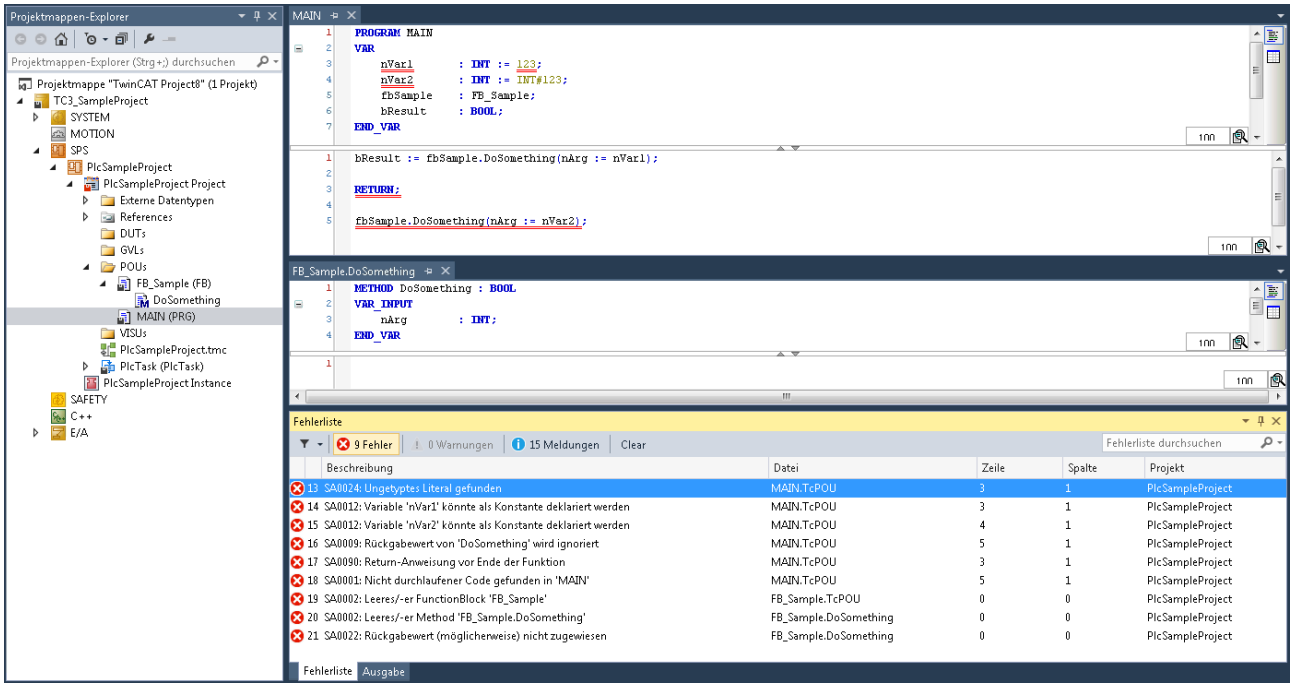
10 Beispiele

10.1 Statische Analyse

Bei Durchführung der Statischen Analyse [▶ 115] wird die Einhaltung der Kodierregeln [▶ 17], der Namenskonventionen [▶ 83] und der unzulässigen Symbole [▶ 114] überprüft. Im Folgenden finden Sie zu jedem dieser Aspekte ein separates Beispiel.

1) Kodierregeln

Im diesem Beispiel sind einige Kodierregeln als Fehler konfiguriert. Die Verletzungen dieser Kodierregeln werden nach Ausführung der Statischen Analyse folglich als Fehler gemeldet. Nähere Informationen können Sie dem folgenden Bild entnehmen.

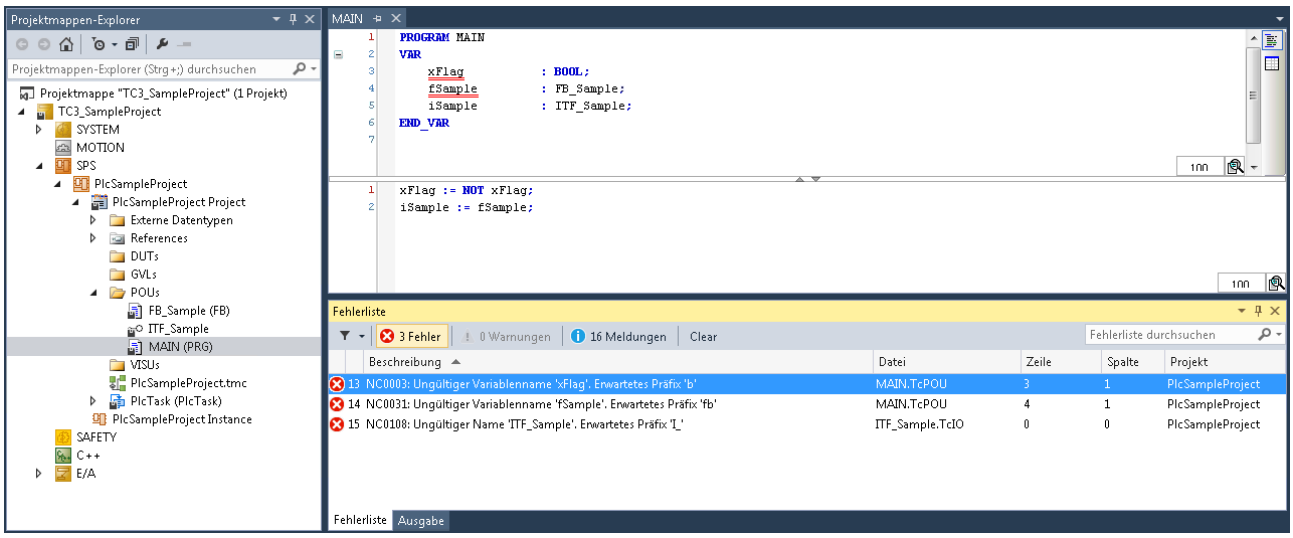


2) Namenskonventionen

Es sind folgende Namenskonventionen konfiguriert:

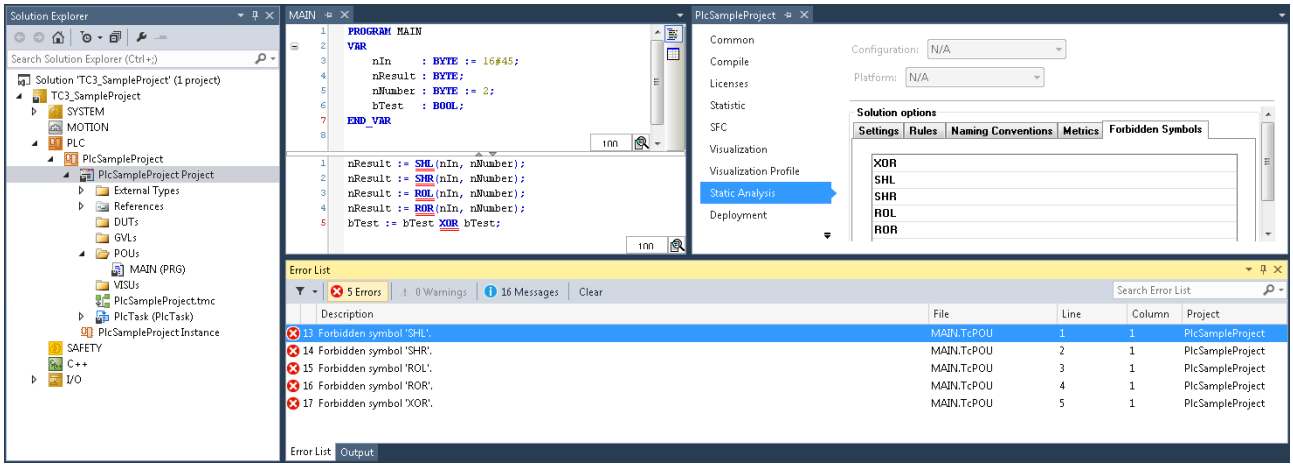
- Präfix "b" für Variablen vom Typ BOOL (NC0003)
- Präfix "fb" für Funktionsbausteininstanzen (NC0031)
- Präfix "FB_" für Funktionsbausteine (NC0103)
- Präfix "I_" für Schnittstellen (NC0108)

Bei der Deklaration der Booleschen Variablen ("x"), der Instanziierung des Funktionsbausteins ("f") sowie bei der Deklaration des Schnittstellentyps ("ITF_") werden diese Namenskonventionen nicht eingehalten. Diese Codestellen werden nach Ausführung der Statischen Analyse jeweils als Fehler gemeldet.



3) Unzulässige Symbole

Der Bitstring-Operator XOR sowie die Bit-Shift-Operatoren SHL, SHR, ROL und ROR sind als unzulässige Symbole konfiguriert. Diese Operatoren sollen im Code nicht verwendet werden.
Entsprechend wird die Verwendung dieser Operatoren nach Ausführung der Statischen Analyse jeweils als Fehler gemeldet.

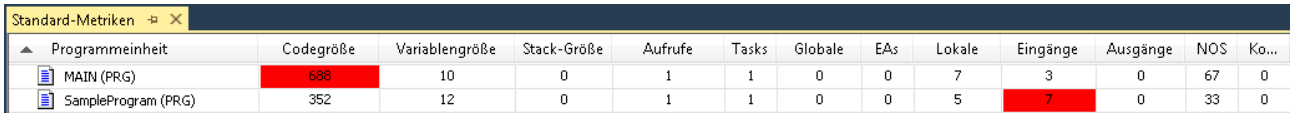


10.2 Standard-Metriken

Nachfolgend finden Sie ein Beispiel zum Umgang mit den Standard-Metriken.

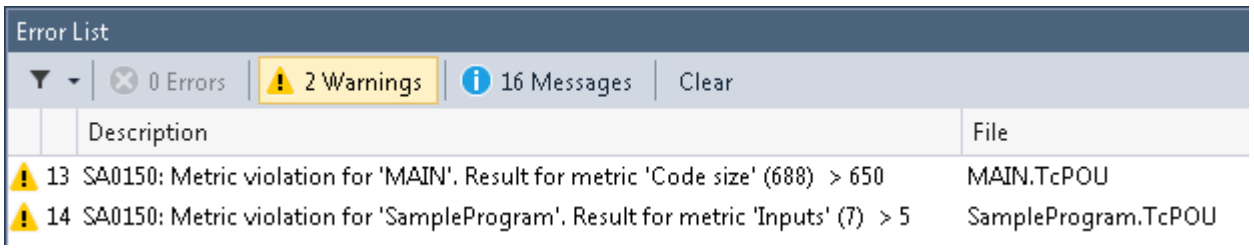
In diesem Beispiel sind "650" (= 650 Bytes) als Obergrenze für die Metrik "Codegröße" und "5" als Obergrenze für die Metrik "Anzahl Eingangsvariablen" definiert (siehe: Konfiguration der Metriken [▶ 97]). Des Weiteren wird Regel 150 (SA0150: Verletzung von Unter- oder Obergrenzen der Metriken) aktiviert und als Warnung konfiguriert.

Nach Ausführen des Befehls 'Standard-Metriken anzeigen' [▶ 118] wird die Metrik-Ansicht geöffnet und die ermittelten Kennzahlen werden tabellarisch angezeigt. Da die Größe des Programms MAIN 688 Bytes beträgt und das Programm SampleProgram 7 Eingangsvariablen besitzt, überschreiten diese Kennzahlen jeweils die definierten Obergrenzen, sodass die entsprechenden Tabellenfelder rot hinterlegt sind.



Programmeinheit	Codegröße	Variablengröße	Stack-Größe	Aufrufe	Tasks	Globale	EAs	Lokale	Eingänge	Ausgänge	NOS	Ko...
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

Dass die definierten Obergrenzen in dem Beispiel überschritten werden, ist in diesem Beispiel nicht nur in der Metrik-Ansicht erkennbar. Da Regel 150 als Warnung konfiguriert ist, wird die Verletzung von Unter- und Obergrenzen der Metriken durch die Statische Analyse überprüft. Nach Durchführung der Statischen Analyse [▶ 115] wird die Verletzung der beiden Obergrenzen somit als Warnung im Meldungsfester ausgegeben.


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

11 Support und Service

Beckhoff und seine weltweiten Partnerfirmen bieten einen umfassenden Support und Service, der eine schnelle und kompetente Unterstützung bei allen Fragen zu Beckhoff Produkten und Systemlösungen zur Verfügung stellt.

Downloadfinder

Unser Downloadfinder beinhaltet alle Dateien, die wir Ihnen zum Herunterladen anbieten. Sie finden dort Applikationsberichte, technische Dokumentationen, technische Zeichnungen, Konfigurationsdateien und vieles mehr.

Die Downloads sind in verschiedenen Formaten erhältlich.

Beckhoff Niederlassungen und Vertretungen

Wenden Sie sich bitte an Ihre Beckhoff Niederlassung oder Ihre Vertretung für den lokalen Support und Service zu Beckhoff Produkten!

Die Adressen der weltweiten Beckhoff Niederlassungen und Vertretungen entnehmen Sie bitte unserer Internetseite: www.beckhoff.com

Dort finden Sie auch weitere Dokumentationen zu Beckhoff Komponenten.

Beckhoff Support

Der Support bietet Ihnen einen umfangreichen technischen Support, der Sie nicht nur bei dem Einsatz einzelner Beckhoff Produkte, sondern auch bei weiteren umfassenden Dienstleistungen unterstützt:

- Support
- Planung, Programmierung und Inbetriebnahme komplexer Automatisierungssysteme
- umfangreiches Schulungsprogramm für Beckhoff Systemkomponenten

Hotline:	+49 5246 963-157
E-Mail:	support@beckhoff.com

Beckhoff Service

Das Beckhoff Service-Center unterstützt Sie rund um den After-Sales-Service:

- Vor-Ort-Service
- Reparaturservice
- Ersatzteilservice
- Hotline-Service

Hotline:	+49 5246 963-460
E-Mail:	service@beckhoff.com

Beckhoff Unternehmenszentrale

Beckhoff Automation GmbH & Co. KG

Hülshorstweg 20
33415 Verl
Deutschland

Telefon:	+49 5246 963-0
E-Mail:	info@beckhoff.com
Internet:	www.beckhoff.com

Trademark statements

Beckhoff®, TwinCAT®, TwinCAT/BSD®, TC/BSD®, EtherCAT®, EtherCAT G®, EtherCAT G10®, EtherCAT P®, Safety over EtherCAT®, TwinSAFE®, XFC®, XTS® and XPlanar® are registered trademarks of and licensed by Beckhoff Automation GmbH.

Third-party trademark statements

Arm, Arm9 and Cortex are trademarks or registered trademarks of Arm Limited (or its subsidiaries or affiliates) in the US and/or elsewhere.

Intel, the Intel logo, Intel Core, Xeon, Intel Atom, Celeron and Pentium are trademarks of Intel Corporation or its subsidiaries.

Microsoft, Microsoft Azure, Microsoft Edge, PowerShell, Visual Studio, Windows and Xbox are trademarks of the Microsoft group of companies.

Mehr Informationen:
www.beckhoff.com/te1200

Beckhoff Automation GmbH & Co. KG
Hülshorstweg 20
33415 Verl
Deutschland
Telefon: +49 5246 9630
info@beckhoff.com
www.beckhoff.com