

JmetrikaAWL

Zusatzinformationen für die Programmiersprache AWL

Analyse sicherheitsrelevanter Software

Ansprechpartner:

Berufsgenossenschaftliches Institut
für Arbeitsschutz – BGIA
Zentralbereich
Prof. Dr. Dietmar Reinert
53754 Sankt Augustin
Tel.: 02241 231-2750
Fax: 02241 231-2234
Dietmar.Reinert@HVBG.de



Inhaltsverzeichnis

1 Allgemeines.....	3
2 Anpassung nicht normkonformer Quelltexte	3
3 Beschreibung der verwendeten Metriken	6
3.1 Halstead	7
3.2 McCabe	9
3.3 Krell	11
3.4 Breuer	13
4 Beschreibung der Qualitätskriterien	15
4.1 Testbarkeit.....	16
4.2 Einfachheit	16
4.3 Lesbarkeit.....	17
4.4 Selbstbeschreibung	18
4.5 Globale Qualität.....	18

1 Allgemeines

Bei zahlreichen Maschinen werden traditionell speicherprogrammierbare Steuerungen (SPS) zur Ablaufsteuerung der einzelnen Be- und Verarbeitungsvorgänge eingesetzt. Dabei kommen für die Verarbeitung sicherheitsbezogener Abläufe spezielle sichere SPS zur Anwendung. Auch wenn diese Steuerungen eine sichere Programmierung durch besondere Funktionsbausteine unterstützen, können umfangreiche Programme – insbesondere, wenn sie als Anweisungsliste (AWL) erstellt wurden – sehr unübersichtlich werden.

Auf Initiative des Fachausschusses Chemie (FA CH) wurde ein Werkzeug entwickelt, mit dem durch die Festlegung geeigneter Maßzahlen für die Softwarequalität (Metriken) umfangreiche SPS-Programme eine schnelle statische Analyse erfahren können. Zusätzlich lassen sich Qualitätskriterien wie Lesbarkeit, Selbstbeschreibung, Testbarkeit und Einfachheit bestimmen. Soweit uns bekannt, ist dies das erste Werkzeug, mit dem sich die Qualität von Software für SPS bestimmen lässt. Der Softwarecode kann ohne weitere Änderungen analysiert werden, wenn er dem Standard IEC 61131 entspricht oder in der Programmiersprache STEP 7 ausgeführt wurde.

2 Anpassung nicht normkonformer Quelltexte

Das Programm jmetrikaAWL ist nur in der Lage, mit der Norm IEC61131-3 konforme Quelltexte einzulesen. Hierbei spielt im Wesentlichen die Struktur des Programms eine Rolle und nicht der Befehlssatz (bis auf bedingte oder unbedingte Sprünge, Return-Anweisungen und direkte Aufrufe einer Komponente). Aufgrund der weiten Verbreitung wurde in jmetrikaAWL ebenfalls der relevante Befehlssatz von STEP7 implementiert, sodass auch Quelltexte von speicherprogrammierbaren Steuerungen von Siemens analysiert werden können. Alle anderen Quelltexte müssen vor einer möglichen Analyse mit jmetrikaAWL zuerst wie folgt beschrieben verändert werden:

Jede Programmorganisationseinheit muss von einem „PROGRAM“, „FUNCTION“ oder „FUNCTION_BLOCK“ Konstrukt umschlossen sein.

```
FUNCTION_BLOCK Name_des_Funktionsblockes
[ ... ]
END_FUNCTION_BLOCK
```

Der Code zwischen diesen Anweisungen lässt sich in einen Deklarationsteil und einen Anweisungsteil aufteilen. Im Deklarationsteil werden alle für die Programmorganisationseinheit (POE) benötigten Variablen definiert. Diese Variablendeklaration ist wiederum gegliedert in die verschiedenen Arten der Variablen. Ein mögliches Beispiel für die Variablendeklaration einer Eingangsvariablen, zweier lokalen, einer globalen und einer Ausgangsvariablen, kann so aussehen:

```
VAR_INPUT
    EINGANSVARIABLE : TYP ;
END_VAR
VAR
    LOKALE_VARIABLE1 : TYP ;
    LOKALE_VARIABLE2 : TYP ;
END_VAR
```

```

VAR_GLOBAL
    GLOBALE_VARIABLE:TYP;
END_VAR
VAR_OUTPUT
    AUSGANGSVARIABLE:TYP;
END_VAR

```

Manche Programme in AWL sehen eine Zuweisungsliste vor, in der den Ein- und Ausgängen einer SPS entsprechende Variablen zugewiesen werden. Eine solche Zuweisungsliste für Variablen könnte z. B. wie folgt aussehen (hier ein Beispiel einer Zuweisungsliste für eine Software für eine speicherprogrammierbare Steuerung von PILZ):

```

E 0.00 EINGANG_0 Beschreibung Funktion Eingang 0
A 0.00 AUSGANG_0 Beschreibung Funktion Ausgang 0

```

In diesem Fall müssten die benötigten Variablen in den Variablenteil der entsprechenden Programmorganisationseinheit wie folgt eingefügt werden:

```

VAR_INPUT
    EINGANG_0:TYP_EINGANG; // Beschreibung Funktion Eingang 0
END_VAR
VAR_OUTPUT
    AUSGANG_0:TYP_AUSGANG; // Beschreibung Funktion Ausgang 0
END_VAR

```

Wer auf die Auswertung der Metriken von *Breuer* verzichten möchte, muss trotzdem zumindest den Beginn und das Ende für einen Block von Variablendeklarationen direkt nach dem Beginn der Programmorganisationseinheit einfügen.

```

FUNCTION_BLOCK Name_des_Funktionsblockes
    VAR
        END_VAR
    [...]
END_FUNCTION_BLOCK

```

Dem Deklarationsteil folgt der Anweisungsteil, der die eigentlichen Anweisungen für die CPU enthält. Hier kommt es im Wesentlichen auf die Struktur an, beginnend mit einer Sprungmarke (optional), gefolgt von einem Operator, dem wiederum ein Operand bzw. eine Operandenliste folgt. An beliebiger Stelle können Anwenderkommentare stehen. Gewöhnliche Operatoren werden bei der Analyse nicht näher betrachtet, sodass ein anderer Befehlssatz für das Einlesen der Quelltexte keine Rolle spielt. Es kommt nur darauf an, Operatoren von Operanden unterscheiden zu können, was durch die gegebene Struktur einer Anweisung kein Problem ist.

Marke: Operator Operand (* Anwenderkommentar *)

Einige Operatoren müssen allerdings der Norm entsprechen, damit sie korrekt ausgewertet werden können. Dies ist bei bedingten oder unbedingten Sprüngen, Return-Anweisungen und direkten Aufrufen einer Komponente der Fall. Sollte sich die Befehle für die genannten Operationen von der Norm unterscheiden, so müssen sie durch folgende Befehle ersetzt werden:

- bedingter Sprung: JMPJC Sprungmarke
- unbedingter Sprung: JMP Sprungmarke
- bedingter Aussprung: RETC
- unbedingter Aussprung: RET
- bedingter Aufruf einer Komponente: CALC Name_der_aufgerufenen_Komponente
- unbedingter Aufruf einer Komponente: CAL Name_der_aufgerufenen_Komponente

Ebenfalls muss für die Auswertung der Anzahl der Timer und der Abfragehäufigkeit Folgendes beachtet werden: Timer müssen nach Norm in der Variablendeklaration vom Typ „TON“, „TOF“ oder „TP“ sein und mit dem Variablennamen des Timers, gefolgt von einem „.Q“, abgefragt werden:

```

VAR
    Timer_Schutztauer : TON;
END_VAR
LD Timer_Schutztauer.Q
JMPJC SprungmarkeTimerAbgelaufen

```

Als Timerabfrage werden auch folgende Operatoren gewertet (diese Befehle werden z. B. für speicherprogrammierbare Steuerungen von PILZ und Siemens verwendet):

```

„SE“, „SA“, „SI“, „SV“, „SS“, „FR“

```

Der Export von Quelltexten einer speicherprogrammierbaren Steuerung z. B. von PILZ kann sehr kryptisch aussehen und bedarf einiger Änderungen, die hier anhand eines kurzen Beispiels erläutert werden. Hier zu sehen ist ein (etwas konstruierter) Quellcode nach dem Export:

```

:
Sprungmarke1 : Netzwerk 00
:
:L E 0.01
:OR A 0.01
:= A 0.01
:
Sprungmarke2 : Netzwerk 01
:
:CAL irgendeinePOE 001
.HierStehtIrgendwas
E 0.01 .HierStehtIrgendwas
A 0.01 .HierStehtIrgendwas
*****
* E 0.01 .Eingang1 Beschreibender Kommentar E *
* A 0.01 .Ausgang1 Beschreibender Kommentar A *
*****

```

Anweisungen stehen immer hinter einem Doppelpunkt. Bis auf die Sprungmarken, bei denen alles nach dem Doppelpunkt entfernt werden muss (hier „Netzwerk 00/01“), müssen alle Zeilen, die nicht mit einem Doppelpunkt oder einem Punkt beginnen, entfernt werden. Bei allen Zeilen, die mit einem Punkt beginnen, kann der Punkt durch ein „//“ (Beginn Kommentar) ersetzt werden. Alle Anweisungen müssen z. B. von einem „FUNCTION_BLOCK“ umgeben werden.

Was von Sternchen umgeben ist, entspricht den verwendeten Variablen, die entsprechend definiert werden müssen. „E 0.01“ steht für den physikalischen Eingang 0.01 an der SPS. Darauf folgt der Name für den Eingang, gefolgt von einem beschreibenden Kommentar.

In allen Anweisungen sollten nun noch alle nummerierten Eingänge, Ausgänge, Merker und Timer durch die definierten symbolischen Namen ersetzt werden, damit die Metriken von *Breuer* korrekt berechnet werden können. Für den Fall, dass dies zu viel Aufwand bedeuten würde, ist zu beachten, dass in Operanden weder Leerzeichen noch Punkte erlaubt sind. Es müssen also zumindest alle Leerzeichen und Punkte in den Operanden z. B. durch einen Unterstrich ersetzt werden. Der Quelltext könnte nach den erläuterten Änderungen wie folgt aussehen:

```
FUNCTION_BLOCK Name_des_Funktionsblockes
    VAR_INPUT
        EINGANG1:TYP_E; // Beschreibender Kommentar E
    END_VAR
    VAR_OUTPUT
        AUSGANG1:TYP_A; // Beschreibender Kommentar A
    END_VAR
    VAR
        Aufruf:TYP_POE; // Typ entspricht einer POE
    END_VAR
    Sprungmarkel :
        L Eingang1 (* bzw. L E_0_01 *)
        OR Ausgang1 (* bzw. OR A_0_01 *)
        = Ausgang1 (* bzw. = A_0_01 *)
    Sprungmarke2 :
        CAL Aufruf
        // HierStehtIrgendwas
END_FUNCTION_BLOCK
```

3 Beschreibung der verwendeten Metriken

Die folgende Beschreibung der verwendeten Metriken wurde im Wesentlichen den im BGIA und in der Fachhochschule Bonn-Rhein-Sieg angefertigten Abschlussarbeiten von *Krell*, *Staron*, *Maurer*, *Ley* und *Breuer* entnommen und stellt nur eine Zusammenfassung dar. Für die entsprechenden Beschreibungen wird auf die in den Abschlussarbeiten referenzierten Quellen hingewiesen.

Einige der beschriebenen Metriken dienen nur als Zwischengröße für die Berechnung anderer Metriken und tauchen somit nicht im Bericht auf. Auf die Grenzwerte der Metriken wird an dieser Stelle nicht näher eingegangen. Sie wurden zum Teil aus der Literatur entnommen, zum Teil aber auch an die Besonderheiten sicherheitskritischer Anwendungen angepasst. Die Grenzwerte dienen als Anhaltspunkt zur Analyse einer Software, müssen aber ggf. noch an die eigenen Bedürfnisse angepasst werden. Um das Werkzeug jmetrika weiter optimieren zu können, sind die Autoren bzgl. der Grenzwerte an einer Rückmeldung aus der industriellen Praxis sehr interessiert. Dies gilt insbesondere für die Metriken von *Krell* und *Breuer*.

3.1 Halstead

Die Basis der von *Halstead* definierten Metriken sind die Operanden und Operatoren. Operanden sind alle Codeelemente, die Aktionen kennzeichnen. Zu den Operanden zählen in Abhängigkeit von der verwendeten Programmiersprache Schlüsselwörter wie z. B. WHILE, IF, CASE und mathematische Ausdrücke wie +, -, /. Zu den Operatoren zählen alle Ausdrücke, die Daten vertreten. Hauptsächlich sind das Variablennamen oder Werte, die direkt im Quellcode vorkommen. Von den Operanden und Operatoren werden jeweils zwei Größen ermittelt. Zum einen die Gesamtanzahl aller Operanden (**N2**) sowie Operatoren (**N1**) und zum anderen die Anzahl aller unterschiedlichen Operanden (**n2**) und Operatoren (**n1**). Weitere Größen, die *Halstead* in seine Berechnungen mit einfließen lässt, sind die Anzahl der Anweisungen (**N_STMTS**) und Kommentare (**N_COM**). Aus diesen sechs Werten werden alle folgenden Metriken berechnet.

Frequenz Kommentare – COM_R

$$\text{COM_R} = \text{N_COM} / \text{N_STMTS}$$

COM_R gibt das Verhältnis zwischen Kommentaren und Anweisungen wieder.

Programmlänge – PR_LGTH

$$\text{PR_LGHT} = \text{N1} + \text{N2}$$

Diese Metrik errechnet die Programmlänge, indem die Anzahl aller Operanden und aller Operatoren addiert werden. PR_LGHT entspricht der Anzahl der Wörter in einem Buch.

Umfang Vokabular – VOC_SZ

$$\text{VOC_SZ} = \text{n1} + \text{n2}$$

VOC_SZ bezeichnet den Umfang des verwendeten Vokabulars. VOC_SZ ist eine Teilmenge von PR_LGTH. VOC_SZ entspricht der Anzahl aller unterschiedlichen Wörter in einem Buch.

Frequenz Vokabular – VOC_F

$$\text{VOC_F} = \text{PR_LGHT} / \text{VOC_SZ}$$

Diese Metrik gibt Aufschluss über die Häufigkeit, mit der einzelne Operatoren bzw. Operanden benutzt werden. Aufgeschlüsselt berechnet sich die Metrik aus der Summe aller Operatoren und Operanden dividiert durch die Summe aller unterschiedlichen Operatoren und Operanden. Ein Ergebnis von „1“ bedeutet, dass es in dem Modul keine Wiederholungen von Variablennamen, Befehlen oder Berechnungen gibt. Wenn das Ergebnis „> 3“ ist, kann man daraus schließen, dass im Durchschnitt ein

Variablenname, ein Befehl oder eine Berechnungsart viermal verwendet wird. Durch die Wiederholungsrate des Vokabulars könnten je nach Programmiersprache Rückschlüsse auf den Wortschatz des Programmierers gezogen werden. Bei sehr hohen Ergebnissen ist es evtl. angebracht, Schleifen einzuführen, weil vermutlich Code dupliziert ist.

Größe der Instruktionen – AVG_S

$$AVG_S = PR_LGHT / N_STMTS$$

AVG_S ist ein Maß für die Länge bzw. Komplexität der einzelnen Anweisungen. Diese Metrik ist bei höheren Programmiersprachen sehr wertvoll, da es bei solchen Programmiersprachen möglich ist, einzelne Zeilen zu schachteln bzw. komplexe Berechnungen durchzuführen. So können sehr lange und meist komplexe Programmzeilen entstehen. In AWL ist es kaum möglich, Einfluss auf die Größe einer Instruktion zu nehmen, sodass diese Metrik für AWL nur eine geringe Bedeutung hat.

Programmvolumen – V

$$V = PR_LGTH * 2 \log VOC_SZ$$

Das Programmvolumen bezeichnet den minimalen Umfang an Bits, der benötigt wird, um dieses Modul zu erstellen. Die Maßzahl hängt von der Programmlänge PR_LGTH und dem Umfang des Vokabulars VOC_SZ ab. Genau genommen müsste $2 \log n$ aufgerundet werden, da es nur ganze Bits gibt.

Potenzielle Volumen – V*

Das potenzielle Volumen V* ist das geringstmögliche Volumen eines Algorithmus. Zur Berechnung wird ein ideales Vokabular angenommen. n2-Ideal ist die Anzahl der geringstmöglichen Ein- und Ausgaben. n1-Ideal ist der Funktionsname und die Anweisung für die Berechnung der Funktion. Das potenzielle Volumen einzelner Algorithmen ist für jede Programmiersprache konstant.

Der Sinn dieses Wertes liegt in der Vergleichsmöglichkeit zum errechneten Programmvolumen. Durch einen solchen Vergleich lässt sich bestimmen, wie nah die Implementierung am Ideal liegt. So könnten z. B. Programmierer oder Programme qualitativ bewertet werden. Da es kaum möglich ist, für jeden Algorithmus in jeder Programmiersprache einen Wert für V* festzulegen, wird folgender Schätzwert verwendet:

$$V^* = (N2 + 2) 2 \log (N2 + 2)$$

Programmlevel – PR_LVL

$$PR_LVL = V^* / V$$

PR_LVL ist das Verhältnis zwischen dem vorhandenen Programmvolumen und dem idealen Volumen. Durch diese Darstellung ist im Idealfall ersichtlich, wie weit der reale Code von einem (mathematisch) idealen Code entfernt ist. Achtung: Zur Berechnung dieser Metrik wird nur der Schätzwert von V* verwendet, sodass diese Metrik keine genauen Angaben machen kann.

Geschätztes Programmlevel – ^PR_LVL

$$\wedge PR_LVL = 2 / n1 * n2 / N2$$

$\wedge PR_LVL$ ist der Schätzwert für PR_LVL. Der Programmlevel kann nicht automatisch berechnet werden, da das ideale Volumen abhängig von der Programmiersprache

und den verwendeten Algorithmen ist. Die beiden Brüche können folgendermaßen formuliert werden:

$2 / n1$ -> je mehr Operatoren verwendet werden, desto geringer der Level von $\wedge L$

$n2 / N2$ -> je öfter Operanden wiederholt werden, desto geringer der Level von $\wedge L$.

Information Komponente – INTELL

$$\text{INTELL} = V * \wedge \text{PR_LVL}$$

INTELL soll die übermittelte Information des Moduls wiedergeben. Um dies zu erreichen, wird das Programmvolume multipliziert mit dem geschätzten Wert des Abstandes zwischen Programmvolume und dessen Idealvolume. Wäre die Schätzung von PR_LVL also $\wedge \text{PR_LVL}$ korrekt, dann entspräche INTELL dem idealen Programmvolume V^* , denn V^* ist gleich $\text{PR_LVL} * V$. INTELL ist also der Abstand zwischen dem Programmvolume und dem idealen Programmvolume.

Programmkomplexität – PR_CPXTY

$$\text{PR_CPXTY} = 1 / \text{PR_LVL}$$

Diese Metrik ist der Kehrwert von PR_LVL. Ideal wäre eine PR_CPXTY von 1, weil dann der Code dem idealen Code entsprechen würde. Je höher das Ergebnis, desto weiter ist der Code vom Ideal entfernt.

Aufwand – EFFORT

$$\text{EFFORT} = V / \text{PR_LVL}$$

EFFORT soll eine Einschätzung über den benötigten geistigen Aufwand darstellen. Um diesen Aufwand zu ermitteln, wird das Volume des Moduls multipliziert mit der Schwierigkeit des Moduls PR_CPXTY ($\text{PR_CPXTY} = 1 / \text{PR_LVL}$). Je höher also das Volume und die Schwierigkeit, desto höher der Aufwand geistiger Arbeit, die benötigt wird, um das Modul zu erstellen.

Fehlerwahrscheinlichkeit – N_ERRORS

$$\text{N_ERRORS} = \text{EFFORT}^{(2/3)} / 3000$$

N_ERRORS berechnet eine Fehlerwahrscheinlichkeit. Die Koeffizienten 2/3 und 3000 ergeben sich aus Erfahrungswerten.

3.2 McCabe

MCCabe bezieht sich mit seinen Metriken auf den Kontrollflussgraphen (KFG) des Programms. Grundlage hierfür ist die Graphentheorie. Ein Programm stellt durch seine Ablaufmöglichkeiten einen gerichteten Graphen dar. Der Graph enthält Knoten und Kanten. Bezogen auf den Quellcode ist jede Codezeile ein Knoten, also eine Aktion. Eine Kante verbindet die Knoten in einer definierten Richtung. Jeder Ein- und Ausgang aus dem Modul wird als Ein- bzw. Ausgangsknoten bezeichnet.

Zur Vereinfachung können mehrere Knoten, die aus linearem Programmcode bestehen, also keine Verzweigungen beinhalten, zu einem Knoten zusammengefasst werden. Diese Zusammenfassung wird als Blockung bezeichnet. Die Blockung hat zur Folge, dass der KFG übersichtlicher wird, sie verringert also die Anzahl der Knoten und Kanten. Im Normalfall ist jeder innere Knoten (alle bis auf Start- und Endknoten) des KFG mit zwei Kanten verbunden (einer, die zu ihm hinführt, und einer Kante, die von dem Knoten abgeht). Ausnahmen bilden tote Knoten, Aussprünge, Verzweigung-

gen, die von einer Bedingung abhängig sind, sowie Knoten, zu denen mehrere Kanten führen. Tote Knoten sind Stellen im Programmcode, die nicht ausgeführt werden, sie können im Assembler leicht entstehen, wenn sie mit einer Sprunganweisung übersprungen werden. Zu diesen toten Knoten führt keine Kante. Aussprünge haben das Verlassen des Moduls zur Folge und sind daher mit dem Endknoten vergleichbar. Von Verzweigungen, die von einer Bedingung abhängig sind, führen zwei Kanten weg. Wenn mehrere Verzweigungen in einem KFG vorhanden sind, ist es möglich, dass mehrere oder alle zusätzlichen Kanten von den Verzweigungen zu einem Knoten führen. Dieser Knoten hat dann mehrere Eingänge. Es gibt eine oder mehrere Möglichkeiten, diesen Graphen zu durchlaufen. Jede Möglichkeit, vom Startknoten zum Endknoten oder zu einem Aussprung zu gelangen, wird als Pfad des KFG bezeichnet. Wenn eine Kante in einem Pfad auf einen bereits durchlaufenen Knoten zurückverweist und sich somit ein Kreis ergibt, wächst die Zahl der möglichen Pfade ins Unendliche.

Anzahl weiterer Ausgänge – P_NODES

Diese Metrik zählt die Knoten eines Moduls, die neben dem Endknoten als Ausgänge dienen. Der Endpunkt wird nicht mitgezählt. Mit diesen Ausgängen ist kein Aufruf eines weiteren Moduls gemeint, sondern ein Aussprung ohne Rückkehr, sprich alle Return-Anweisungen.

Anzahl Ausgänge – N_OUT

$$N_OUT = P_NODES + 1$$

N_OUT zählt alle Ausgangsknoten eines Moduls. Die „+ 1“ steht stellvertretend für den Endknoten des Moduls.

Anzahl Eingänge – N_IN

Die Metrik N_IN hat in AWL immer den Wert 1, da es immer nur einen Eingang in eine POE gibt.

Anzahl Ein- und Ausgänge – N_IO

$$N_IO = N_OUT + N_IN$$

N_IO ist die Addition aller Ein- und Ausgänge eines Moduls.

Anzahl verarbeitende Knoten – N_SEQ

N_SEQ ist die Anzahl der Knoten, in denen Code ausgeführt wird. Verzweigungen sowie Start- oder Endpunkte werden dabei nicht gezählt. Diese Knoten beinhalten meistens mehrere Anweisungen aus seriellem Code. Durch die Blockung werden sie im KFG als ein Knoten dargestellt.

Anzahl Sprünge / Verzweigungen – N_JUMPS

N_JUMPS zählt die Anzahl der Verzweigungen und Sprünge in einem Modul. Zu diesen Sprüngen zählen Verzweigungen, die von Bedingungen abhängen und einfache Sprünge, die von keiner Bedingung abhängen.

Anzahl undefinierter Sprünge – UNDEF_JUMPS

UNDEF_JUMPS zählt die Anzahl aller Sprünge, deren Zielsprungmarken nicht existieren.

Anzahl unbedingter Sprünge – UNCOND_JUMPS

UNCOND_JUMPS sind die Sprünge in einem Modul, die bedingungslos sind. Durch den Einsatz von bedingungslosen Sprunganweisungen kann es zu dem so genannten Spaghetti-Code kommen. In diesem führen die einzelnen Pfade des Programms kreuz und quer durch den Programmcode.

Anzahl direkt aufgerufener Komponenten – DRCT_CALLS

DRCT_CALLS zählt alle Aufrufe von Modulen. Diese Aufrufe können als serieller Code angesehen werden, da nach dem Ablauf des aufgerufenen Moduls wieder an die aufrufende Stelle zurückgesprungen wird.

Anzahl Knoten – N_NODES

N_NODES zählt die Knoten nach der Blockung in einem Modul.

Anzahl Kanten – N_EDGES

N_EDGES stellt die Anzahl der Kanten nach der Blockung in einem Modul dar.

Zyklomatische Zahl – VG

$$VG = N_EDGES - N_NODES + N_IO$$

Die zyklomatische Zahl von *McCabe* misst die Anzahl der Verzweigungen in einem Modul. Für ein seriell ablaufendes Modul ergibt $VG = 1$. Für jede weitere Verzweigung wird eine weitere „1“ zu VG addiert. VG gibt die maximale Anzahl der Testfälle an, die benötigt wird, um jeden Pfad des Moduls einmal zu durchlaufen.

Kontrollflussdichte – C_DENS

$$C_DENS = (VG - 1) / N_NODES$$

Diese Metrik beschreibt das Verhältnis von Verzweigungen zu Knoten des KFG. Eine Verzweigung zeigt normalerweise auf zwei unterschiedliche Knoten. Das heißt allerdings nicht, dass alle Verzweigungen auf jeweils zwei unterschiedliche Knoten zeigen. Es ist möglich, dass mehrere Verzweigungen auf denselben Knoten zeigen. Wenn ein Modul viele Verzweigungen beinhaltet, die mehrfach auf den gleichen Knoten zeigen, steigt der Wert von C_DENS.

Maximaler Grad – MAX_DEG

MAX_DEG ist die maximale Anzahl von Kanten, die auf einen Knoten treffen und von ihm wegführen. Das Ergebnis steigt durch die Anzahl der Verzweigungen, die auf einen Knoten verweisen. MAX_DEG zählt auch den Zweig, der den Knoten verlässt, wenn es sich nicht um einen P_NODE handelt.

3.3 Krell

Durchschnittliche Verwendung der Operanden – N2_F

$$N2_F = N2 / n2$$

N2_F ermittelt die Wiederholungsrate der Operanden.

Anzahl unterschiedlicher Aussprünge – NP_NODES

Diese Metrik ermittelt die Anzahl der unterschiedlichen Aussprünge. Da für die Aussprünge in AWL nur Return-Anweisungen gezählt werden und bei einer statischen Analyse nicht feststellbar ist, in was für ein Modul ausgesprungen wird, liegt das

Ergebnis der Metrik immer zwischen 0 (nur Aussprung am Programmende) und 1 (ein oder mehrere Return-Anweisungen im Programm vorhanden).

Anzahl indirekter Aussprünge – IP_NODES

Bei dieser neuen Metrik werden alle Sprünge gezählt, die direkt oder indirekt zum Verlassen des Moduls führen. Indirekte Aussprünge entstehen z. B. durch die Einführung einer Sprungmarke am Ende des Moduls. Wenn von mehreren Stellen innerhalb dieses Moduls diese Sprungmarke am Ende des Moduls angesprungen wird, ist jeder Sprung gleichbedeutend einer P_NODE.

Summe aller maximaler Grade – S_MAX_DEG

Ziel dieser Metrik ist es, CASE-Strukturen zu finden und einen Eindruck von ihrer Größe zu bekommen. Das Ergebnis kann darüber hinaus auch benutzt werden, um VG zu relativieren, da CASE-Strukturen für einen Leser überschaubar sind.

S_MAX_DEG sucht nach Knoten im KFG, die von mehr als zwei bedingten oder unbedingten Sprüngen angesprungen werden. Die Summe der Sprünge, die zu diesen ausgewählten Knoten führen minus der Gesamtzahl der betroffenen Knoten ergibt S_MAX_DEG. Ohne die Subtraktion der betroffenen Knoten würden die CASE-Strukturen nicht in spätere Berechnungen einfließen, was die Ergebnisse verfälschen würde.

Die Untersuchung der Metrik hat ergeben, dass es sich bei den ermittelten Knoten nicht automatisch um CASE-Strukturen handelt. Es ist aber durchaus festzustellen, dass Knoten, die von mehr als fünf Sprüngen aufgesucht werden, generell die Struktur des KFG vereinfachen. In vielen der untersuchten Module, bei denen CASE-Strukturen vermutet werden, handelt es sich um Fehlerbehandlungen oder gesammelte Aussprünge.

VG ohne CASE-Strukturen – VG_S_MAX_DEG

$$VG_S_MAX_DEG = VG - S_MAX_DEG$$

CASE-Strukturen haben erheblichen Einfluss auf das Ergebnis von VG. Um diesen Einfluss zu minimieren, wird durch VG_S_MAX_DEG versucht, ein VG-Ergebnis ohne CASE-Strukturen bzw. größere vereinfachende Strukturen zu gewinnen.

VG ohne Fehlerverwaltung – VG_NEU

$$VG_NEU = VG - S_MAX_DEG - (I_P_NODES - N_P_NODES)$$

VG_NEU ist eine Erweiterung von VG_S_MAX_DEG. Zusätzlich zu den CASE- bzw. den vereinfachenden Strukturen werden dabei alle Aussprünge eines Moduls von VG abgezogen. Aussprünge in ein Fehlermodul können ebenso in einer CASE-Struktur zusammengefasst werden. Aus diesem Grund werden nur die unterschiedlichen Aussprungmarken gezählt.

Anzahl toter Sprungmarken – N_DEAD_LABELS

Bei der Analyse der Programmcodes fällt auf, dass in vielen Modulen Sprungmarken definiert sind, die nicht angesprungen werden. N_DEAD_LABELS zählt die Sprungmarken.

Anzahl der überflüssigen Sprünge – N_WASTE_JUMPS

Diese Metrik ermittelt alle Sprünge, die in die nächste Programmzeile führen. Da das Programm auch ohne einen Sprung in die nächste Zeile finden würde, sind diese Sprünge überflüssig. Besonders verheerend ist ein COND_JUMP, der auf die nächste

Zeile verweist, weil in einem solchen Fall das Ergebnis der Entscheidung keine Auswirkung hat.

Anzahl der einzeiligen Schleifen – N_LOOP_LINE

N_LOOP_LINE zählt in einem Modul die Schleifen, die aus einer Zeile bestehen. Derartige Schleifen könnten von einem Timer abhängig sein, was auf eine Synchronisation schließen lässt. N_LOOP_LINE soll diese feststellen. Eine Synchronisation ist in vielen Fällen notwendig und erwünscht.

Anzahl der Rücksprünge – N_LOOP

Hier werden alle Sprünge in einem Modul gezählt, die auf eine Sprungmarke zurückspringen. Da es bei AWL nicht üblich ist, den Inhalt eines Schleifenkörpers einzurücken, können Schleifen schnell unübersichtlich werden. Mit jeder Schleife nimmt die Komplexität des Moduls zu. Auf eine solche Komplexität kann mit dieser Metrik hingewiesen werden.

VG mal Anzahl Anweisungen – VG_STMTS

$$VG_STMTS = VG * N_STMTS$$

Die Metrik VG_STMTS ist das Produkt aus VG und N_STMTS. Ziel dieser Metrik ist es, Module zu erlauben, die zwar einen großen Umfang haben, allerdings nur eine geringe Komplexität aufweisen.

Erweiterte Frequenz Kommentare – NEU_COM_R

$$NEU_COM_R = N_COM * VOC_F / (N_STMTS + (VG_MAX_DEG - 1) * 2)$$

Um das Ergebnis von COM_R besser an die Eigenschaften des Moduls anzupassen, fließen zusätzlich zu N_COM und N_STMTS auch VOC_F und VG_MAX_DEG in NEU_COM_R ein. Ein hohes VG_MAX_DEG-Ergebnis erfordert nach der oberen Berechnung mehr Kommentierung, wohingegen ein hoher VOC_F-Wert die Anforderungen an die Kommentierung sinken lässt.

3.4 Breuer

Anzahl globaler Variablen – N_GLOB_VAR

Diese Metrik bestimmt die Anzahl der globalen Variablen einer POE. Variablen der Art „EXTERNAL“ und „ACCESS“ werden in diesem Kontext auch als globale Variablen betrachtet, da sie eine ähnliche Funktion wie globale Variablen haben. Um eine wohldefinierte Schnittstelle für eine POE zu bekommen, muss sie möglichst wenige globale Variablen besitzen.

Anzahl lokaler Variablen – N_LOC_VAR

Diese Metrik bestimmt die Anzahl der lokalen Variablen einer POE. Viele lokale Variablen deuten darauf hin, dass eine POE komplex ist.

Anzahl Eingangsvariablen – N_IN_VAR

Diese Metrik bestimmt die Anzahl aller Eingangsvariablen einer POE. Dazu werden alle Variablen gezählt, die sowohl unter VAR_INPUT, als auch unter VAR_IO deklariert werden.

Anzahl Ausgangsvariablen – N_OUT_VAR

Diese Metrik bestimmt die Anzahl aller Ausgangsvariablen einer POE. Dazu werden alle Variablen gezählt, die sowohl unter VAR_OUTPUT als auch unter VAR_IO deklariert werden. Wenn die obere Grenze überschritten wird, wird die POE möglicherweise zu komplex und muss aufgeteilt werden.

Gesamtanzahl aller Variablen – N_ALL_VAR

$$N_ALL_VAR = N_GLOB_VAR + N_LOC_VAR + N_IN_VAR + N_OUT_VAR$$

In dieser Metrik wird die Anzahl aller Variablen aufaddiert.

Gesamtlänge aller Variablennamen – L_ALL_VAR

Für diese Metrik werden die Längen aller Variablennamen addiert.

Durchschnittliche Länge aller Variablennamen – L_VAR

$$L_VAR = L_ALL_VAR / N_ALL_VAR$$

Variablen sollten den Quellcode schon durch ihre Namensgebung dokumentieren und verständlicher machen. Um diese Forderung zu erfüllen, muss die Variable eine gewisse Länge haben. Ein langer Variablenname wird in der Regel zum Verständnis des Programms beitragen.

Länge des Namens der Programmorganisationseinheit – L_POE

Für die Selbstbeschreibung und Lesbarkeit eines Programms ist es wichtig, schnell zu erkennen, welche Funktion eine Programmorganisationseinheit hat. Am leichtesten erreicht man dies, indem die Programmorganisationseinheit einen aussagekräftigen Namen erhält. Natürlich kann kein Computer die Bedeutung eines Namens einer Programmorganisationseinheit erkennen, allerdings wird ein sehr kurzer Name vermutlich auch wenig über den Baustein aussagen.

Anzahl beschriebener Eingangsvariablen – N_WR_IN

Eingangsvariablen sollten in keinem Fall beschrieben werden, da dies zu undefinierten Zuständen im Programm führen kann. Deswegen wird mit dieser Metrik kontrolliert, ob Eingangsvariablen beschrieben werden.

Anzahl gelesener Ausgangsvariablen – N_RD_OUT

Ausgangsvariablen sollten nicht in der POE, zu der sie gehören, gelesen werden.

Verhältnis IN/OUT-Variablen zu anderen Variablen – N_IO_TO_OTHER

$$N_IO_TO_OTHER = (N_GLOB_VAR + N_LOC_VAR) / (N_IN_VAR + N_OUT_VAR)$$

Hier wird das Verhältnis zwischen den globalen und lokalen Variablen und den Eingangs- und Ausgangsvariablen berechnet. Ein großes Verhältnis bedeutet, dass es im Gegensatz zu Eingangs- und Ausgangsvariablen sehr viele lokale und globale Variablen gibt. In diesem Fall werden viele lokale Variablen bzw. andere POEs benötigt, um wenige Ausgänge zu steuern. Dies bedeutet, dass die POE sehr komplex und damit schwer verständlich sein kann.

Anzahl gesetzter Ausgänge – N_SET_OUT

Hier wird die Anzahl von Zuweisungen auf Ausgangsvariablen gezählt.

Anzahl Zuweisungen pro Ausgang – N_ASS_OUT

$$N_ASS_OUT = N_SET_OUT / N_OUT_VAR$$

Ein Ausgang soll immer genau einmal innerhalb einer POE gesetzt werden, da sonst die Lesbarkeit und Verständlichkeit der POE stark erschwert wird. Aus diesem Grund sollte das Verhältnis zwischen gesetzten Ausgängen und der Anzahl von Ausgangsvariablen immer 1 sein.

Anzahl Zuweisungen – N_ASS

Diese Metrik dient als Zwischengröße für die Metrik „Durchschnittliche Häufigkeit der Veränderung der Variablen N_C_VAR“. Die Anzahl der Zuweisungen entspricht der Anzahl des Vorkommens von „ST“ nach DIN EN 61131-3 bzw. „=“ in STEP7.

Durchschnittliche Häufigkeit der Veränderung der Variablen – N_C_VAR

$$N_C_VAR = N_ASS / N_ALL_VAR$$

Variablen sollten nur an einer Stelle im Programm verändert werden.

Anzahl Timer – N_TIMER

Die Komplexität eines SPS-Programms wird durch Timer wesentlich beeinflusst. Mit dieser Metrik kann die Anzahl der verwendeten Timer kontrolliert werden.

Häufigkeit der Abfrage von Timern – N_QUERY_T

Jedes Abfragen von Timern (entspricht dem Verwenden von Variablen des Typs „TP“, „TON“ und „TOF“ nach DIN EN 61131-3, bzw. dem Verwenden der Befehle „SE“, „SA“, „SI“, „SV“, „SS“, „FR“ und „R“ in STEP7) erhöht die Komplexität und Nachvollziehbarkeit einer POE. Wenn es in einer POE zu viele Timerabfragen gibt, sollte die POE hinsichtlich einer Aufteilung in mehrere POEs untersucht werden.

Verhältnis der Anzahl der Timer zu deren Abfragehäufigkeit – N_T_TO_Q

Damit die Verwendung des Timers verständlich ist, sollte jeder Timer in einer POE genau einmal abgefragt werden.

Anzahl unterschiedlicher direkt aufgerufener Komponenten – N_DRCT_CALLS

Entsprechend *Krells* Metrik „Anzahl unterschiedlicher Aussprünge NP_NODES“, die unterschiedliche Aussprünge ohne Rückkehr betrachtet, werden mit dieser Metrik alle unterschiedlichen direkten Aufrufe gezählt. Diese Metrik relativiert *McCabes* Metrik „Anzahl direkt aufgerufener Komponenten – DRCT CALLS“, indem sie berücksichtigt, dass eine POE leichter zu verstehen ist, wenn aus einer POE heraus öfter dieselben POEs aufgerufen werden und nicht immer unterschiedliche.

4 Beschreibung der Qualitätskriterien

Den Metriken wird, wenn sie in ihren Grenzen liegen, jeweils eine 1 zugeordnet, andernfalls eine 0. Diese Werte werden dann jeweils für eine Aussage logisch miteinander verknüpft, sodass eine Aussage demnach ebenfalls die Werte 0 oder 1 erhält.

Ein Beispiel:

$$ACCEPTED = VG \& VG_STMTS \& N_LOOP \& N_QUERY_T$$

Dies entspricht der Formel für die Aussage ACCEPTED des Qualitätskriteriums Testbarkeit. Ein Modul wird hinsichtlich seiner Testbarkeit akzeptiert, wenn die Metriken

VG, VG_STMTS, N_LOOP und N_QUERY_T gleichzeitig in ihren Grenzen liegen. Damit wird die Aussage ACCEPTED gültig und erhält den Wert 1. Durch diesen Ansatz ist gewährleistet, dass eine Metrik nur auf die sie betreffenden Aussagen einen Einfluss hat. Des Weiteren kann ein Qualitätskriterium nun auch mehrere gültige Aussagen haben.

4.1 Testbarkeit

Mit VG spielt die Anzahl der Verzweigungen in einer Funktion eine Rolle, die bei einem Zweig- oder Pfadtest durchlaufen werden müssen. Die Metrik N_LOOP liefert einen Anhaltspunkt über die Anzahl möglicher Schleifen in einer POE. Die Anzahl der Schleifen ist z. B. bei einem „Boundary-Interior“-Test eine wichtige Größe. Außerdem wird die Metrik VG_STMTS betrachtet, die die Komplexität und die Anzahl der Anweisungen miteinander in Verbindung setzt. Grundlage dafür ist, dass sich eine große, aber wenig komplexe POE oder aber eine komplexe, aber dafür sehr kleine POE möglicherweise trotzdem gut testen lassen. Die Metrik N_QUERY_T zählt die Abfragehäufigkeit der verwendeten Timer in einer POE. Es ist schwierig, passende Testfälle zu entwickeln, wenn es in einer POE häufig Abfragen von Timern gibt.

Akzeptiert wird eine POE bzgl. ihrer Testbarkeit nur, wenn all ihre die Testbarkeit betreffenden Metriken in ihren Grenzen liegen. Wenn die Metrik N_QUERY_T außerhalb ihrer Grenzen liegt, wird die Aussage TO_INSPECT getroffen. Die POE sollte auf ihre Timerverwendung inspiziert werden. Wenn entweder die Metrik N_LOOP oder VG außerhalb ihrer Grenzen liegt, sollte die POE besser strukturiert werden. Gekürzt werden sollte die POE hinsichtlich ihrer Testbarkeit genau dann, wenn die Metrik VG_STMTS nicht in ihren Grenzen liegt. Damit ergeben sich folgende Formeln:

ACCEPTED

$$= VG \& VG_STMTS \& N_LOOP \& N_QUERY_T$$

TO_INSPECT

$$= !N_QUERY_T$$

TO_STRUCTURE

$$= !N_LOOP \mid !VG$$

TO_CUT

$$= !VG_STMTS$$

4.2 Einfachheit

Ein Programm gilt als einfach, wenn es wenig komplex ist. In Bezug auf die Komplexität spielen die Metriken VG_NEU und VG_STMTS eine wichtige Rolle. Außerdem kommt der sehr wichtige Aspekt der Verwendung von Timern für SPSen hinzu. Hierzu betrachten wir zum einen die Anzahl der verwendeten Timer (N_TIMER), zum anderen das Verhältnis der Anzahl der Timer zu deren Abfragehäufigkeit (N_T_TO_Q). Bei beiden Metriken bedeutet ein höherer Wert auch eine Vergrößerung der Komplexität, da es für den menschlichen Leser sehr schwierig ist, das genaue Ablaufverhalten von Timern im Kontext zu verstehen. Insbesondere sollte ein Timer in jeder POE genau einmal abgefragt werden. Für das Qualitätskriterium Einfachheit wird auch noch die Metrik „Verhältnis IN / OUT Variablen zu restlichen Variablen – N_IO_TO_OTHER“ betrachtet, da man bei einem sehr niedrigen Verhältnis der Eingangs- und Ausgangs-

variablen zu den restlichen Variablen von einer hohen Komplexität ausgehen kann. Wenn viele Variablen benötigt werden, um nur einen Ausgang zu setzen, dann bedeutet das, dass sich vermutlich eine komplizierte Berechnung innerhalb dieser POE verbirgt.

Akzeptiert wird eine POE bzgl. der Einfachheit, wenn die Metriken VG_NEU, VG_STMTS, N_IO_TO_OTHER, N_TIMER, N_T_TO_Q innerhalb ihrer Grenzen liegen. Für den Fall, dass alle Metriken ihre Grenze überschreiten, sollte die POE komplett neu geschrieben werden. Wenn die obere Grenze von VG_STMTS überschritten wird, bedeutet das, dass die POE in Bezug auf ihre Komplexität zu groß ist. Aus diesem Grund wird in diesem Fall empfohlen, die POE zu verkleinern. In allen anderen Fällen, bei denen eine oder mehrere Metriken ihre Grenzen überschreiten, sollte die POE hinsichtlich ihrer Einfachheit inspiziert werden. Damit ergeben sich folgende Formeln:

ACCEPTED

= VG_NEU & VG_STMTS & N_IO_TO_OTHER & N_TIMER & N_T_TO_Q

TO INSPECT

= !TO_REWRITE & (!VG_NEU | !N_IO_TO_OTHER | !N_TIMER | !N_T_TO_Q)

TO CUT

= !VG_STMTS

TO REWRITE

= !VG_NEU & !VG_STMTS & !N_IO_TO_OTHER & !N_TIMER & !N_T_TO_Q

4.3 Lesbarkeit

VG_NEU beschreibt die Komplexität ohne Fehlerverwaltung und CASE-Strukturen. Für das Qualitätskriterium Lesbarkeit wird VG_NEU verwendet, da CASE-Strukturen generell gut lesbar sind, ebenso wie eine Fehlerverwaltung, bestehend aus genau einem Aussprung aus der POE. VG_STMTS betrachtet den Umfang einer POE im Zusammenspiel mit der Komplexität. Eine große POE, die eine niedrige Komplexität hat, kann durchaus noch lesbar sein. Im Gegensatz dazu kann eine kleine POE, die eine sehr große Komplexität hat, unter Umständen schon nicht mehr lesbar sein. N_LOOP soll Schleifen in einer POE finden. Der Grund dafür ist, dass viele Schleifen den Code schwerer lesbar machen. Hinzu kommt die Überlegung, dass die durchschnittliche Häufigkeit der Veränderung der Variablen (N_C_VAR) ebenfalls eine entscheidende Rolle in Bezug auf die Lesbarkeit spielt. Wenn Operanden sehr häufig geändert werden, ist es für den Leser schwer zu erfassen, welche Bedeutung der Operand im jeweiligen Kontext hat.

Wenn alle bzgl. der Lesbarkeit betrachteten Metriken ihre Grenzen erfüllen, ist die POE akzeptiert. Wenn eine der Metriken VG_NEU oder VG_STMTS oder beide ihre Grenzen überschreiten, wird empfohlen, die POE im Hinblick auf ihre Lesbarkeit zu inspizieren. Besser strukturiert werden sollte die POE, wenn eine der Metriken N_LOOPS oder N_C_VAR oder beide ihre Grenzen nicht erfüllen. Damit ergeben sich folgende Formeln:

ACCEPTED

= VG_NEU & VG_STMTS & N_LOOP & N_C_VAR

TO_INSPECT

= !VG_NEU | !VG_STMTS

TO_STRUCTURE

= !N_LOOP | !N_C_VAR

4.4 Selbstbeschreibung

Eine POE hat die Eigenschaft einer guten Selbstbeschreibung dann, wenn sie gut kommentiert ist, aussagekräftige Variablennamen hat und auch einen aussagekräftigen Programmorganisationseinheitsnamen hat. Die Metrik NEU_COM_R prüft, ob das Modul hinsichtlich seiner Komplexität genug kommentiert ist. Wenn die Metrik L_VAR außerhalb ihrer Grenzen liegt, scheinen die Variablennamen zu kurz auszufallen und somit nicht besonders aussagekräftig zu sein. Liegt die Metrik L_POE nicht innerhalb der Grenzen, ist der Name der Programmorganisationseinheit ziemlich kurz geraten. Ein langer, aussagekräftiger Name für eine POE hat den Vorteil, dass bei Verwendung der POE auf den ersten Blick ersichtlich ist, welche Funktion sie erfüllt.

Wenn eine der Metriken L_VAR oder L_POE oder beide ihre Grenzen überschreiten, sollte die Namensgebung der Variablen und Programmorganisationseinheiten verbessert werden. Wenn die Metrik NEU_COM_R eine ihrer Grenzen überschreitet, wird die Verbesserung der Dokumentierung der POE gefordert. Damit eine POE hinsichtlich ihrer Selbstbeschreibung akzeptiert wird, müssen alle drei Metriken ihre Grenzen einhalten. Damit ergeben sich folgende Formeln:

ACCEPTED

= NEU_COM_R & L_V_AR & L_POE

TO_IMPROVE_NAMING

= !L_V_AR | !L_POE

TO_DOCUMENT

= !NEU_COM_R

4.5 Globale Qualität

Die Aussagen der globalen Qualität umfassen alle Aussagen der einzelnen Qualitätskriterien. Sobald für eine POE eine Aussage als gültig erklärt wurde, so wird diese Aussage auch für die globale Qualität als gültig erklärt. Dies gilt allerdings nicht für die Aussage ACCEPTED. Wenn für eine POE z. B. die Aussage TO_STRUCTURE getroffen wird, so wird auch für die globale Qualität die Aussage TO_STRUCTURE getroffen, damit der Gutachter sofort erkennt, dass mindestens eine POE neu strukturiert werden muss. Nur wenn für alle POEs jedes Qualitätskriterium die Aussage ACCEPTED erhält, gilt auch für die globale Qualität die Aussage ACCEPTED.

Damit ergeben sich für die globale Qualität folgende mögliche Aussagen:

- ACCEPTED
- TO_INSPECT
- TO_CUT
- TO_STRUCTURE

- TO_REWRITE
- TO_IMPROVE_NAMING
- TO_DOCUMENT