

JmetrikaC

Zusatzinformationen für die Programmiersprache C

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 Beschreibung der verwendeten Metriken	4
2.1 Halstead	4
2.2 McCabe	6
2.3 Krell.....	9
2.4 Maurer.....	11
3 Beschreibung der Qualitätskriterien	11
3.1 Testbarkeit.....	11
3.2 Einfachheit	12
3.3 Lesbarkeit.....	12
3.4 Selbstbeschreibung	13
3.5 Globale Qualität.....	13

1 Allgemeines

In der Implementierungsphase eines Softwareentwicklungsprojekts, also der Codierung in einer Programmiersprache, können viele Fehler gemacht werden. Obwohl die typischen Fehler auf dieser Ebene relativ klar definiert sind, können sie nicht immer erkannt werden. Die Implementierungsfehler sind meistens spezifisch für eine bestimmte Programmiersprache oder ein Betriebssystem. Die Wahl einer „sicheren“ Programmiersprache ist daher eine Möglichkeit, Implementierungsfehler zu vermeiden. Bei der Auswahl der Programmiersprache sollte auf die unterschiedlichen Schwerpunkte verschiedener Sprachen im Hinblick auf die Funktionalität, Effizienz oder Zuverlässigkeit geachtet werden. So hat die Programmiersprache C ihre Stärken in der Laufzeiteffizienz und Hardwarenähe. C ist aber grundsätzlich keine sichere Programmiersprache. Es müssen Codierungsregeln beachtet werden, die man mit Metriken überprüfen kann.

Es ist möglich, Assemblercode in C-Programme einzubinden. Dadurch kann eine statische Analyse erschwert werden. Die Teile des Quellcodes in unterschiedlichen Programmiersprachen müssen getrennt analysiert werden, da der Parser zum Einlesen von C-Quellcode nicht in der Lage ist, Assemblercode einzulesen. Vor der Analyse muss also jeglicher Inline-Assemblercode entfernt werden.

Seitdem C standardisiert wurde, sind neue, immer leistungsfähigere Compiler ein Entwicklungsschwerpunkt. Die Entwickler von C-Softwareprojekten rechnen damit, dass die von ihnen eingesetzten Compiler fehlerfrei funktionieren, d. h., dass nach der Zwischencodeerzeugung und der Optimierung durch den Compiler keine Verschlechterung der Qualität des Quellcodes entsteht. Deswegen ist es für den Gutachter wichtig zu wissen, dass mit einer statischen Codeanalyse nur die Qualität des Quellcodes überprüft werden kann mit dem Ziel, die Qualität des Programms zu sichern oder zu verbessern. Inwieweit sich die Qualität des Endproduktes durch die Optimierung des Compilers verändert, hängt vom eingesetzten Compiler ab. Unabhängig vom Quellcode muss deshalb der Compiler einer Überprüfung unterzogen werden.

Eine weitere Besonderheit der Programmiersprache C ist – im Zusammenhang mit der Beurteilung der Qualität – die Möglichkeit, Bibliotheken zu benutzen. Bibliotheken sind Quellcodedateien, die bereits implementierte und nützliche Standardfunktionen beinhalten. Der Programmierer kann solche Funktionen selbst implementieren, deren Prototypen als „*.h“ Dateien gespeichert werden. Beim Kompilieren bindet der Compiler diese Dateien ein. Die Qualität der Bibliothekskomponenten hat auch Einfluss auf die Qualität des zu erstellenden Programms.

Wichtig: Da jmetrikaC solche Bibliotheken nicht selbstständig in den Quelltext einbindet, wird der Parser bei Verwendung von in den Bibliotheken neu definierten Datentypen eine Fehlermeldung ausgeben. Um diese Problematik zu umgehen, kann entweder vor der Analyse der Präcompiler auf die Quelltexte angewendet werden (was allerdings dazu führt, dass jegliche Funktionen in der Bibliothek analysiert werden) oder der Anwender ersetzt für die Analyse im Quelltext alle neu definierten Datentypen durch primitive Datentypen wie z. B. „int“.

2 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 *Maurer*.

2.1 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.

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. n^2 -Ideal ist die Anzahl der geringstmöglichen Ein- und Ausgaben. n^1 -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:

$$V^* = (N^2 + 2) 2 \log (N^2 + 2).$$

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 – $\hat{PR_LVL}$

$$\hat{PR_LVL} = 2 / n1 * n2 / N2$$

$\hat{PR_LVL}$ 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 \hat{L}

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

Information Komponente – INTELL

$$INTELL = V * \hat{PR_LVL}$$

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

Programmkomplexität – PR_CPXTY

$$PR_CPXTY = 1 / PR_LVL$$

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

$$EFFORT = V / PR_LVL$$

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

Fehlerwahrscheinlichkeit – N_ERRORS

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

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

2.2 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, Ausprünge, Verzweigungen, 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. Ausprünge 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 Ausprung 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 Ausprung 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 bei C-Quelltexten immer den Wert 1, da es immer nur einen Eingang in eine Funktion 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 („If“- , „Switch“- , „Do“- , „While“- , „For“-Anweisungen) und einfache Sprünge, die von keiner Bedingung abhängen („Goto“-Anweisungen).

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.

Verschachtelungstiefe – MAX_LVL

Diese Metrik zählt die maximale Verschachtelungstiefe in einer C-Funktion. Die Verschachtelungstiefe erhöht sich bei jeder „Do“- , „For“- , „While“- , „If“- und „Switch“-Anweisung um jeweils 1 und wird beim Verlassen einer solchen Struktur wieder um 1 reduziert. MAX_LVL entspricht dem größten jemals erreichten Wert der Verschachtelungstiefe.

2.3 Krell

Durchschnittliche Verwendung der Operatoren – N1_F

$$N1_F = N1 / n1$$

N1_F ermittelt die Wiederholungsrate der Operatoren .

Durchschnittliche Verwendung der Operanden – N2_F

$$N2_F = N2 / n2$$

N2_F ermittelt die Wiederholungsrate der Operanden. Im Normalfall sollte ein Operand

1. definiert werden,
2. initialisiert werden,
3. berechnet werden und
4. abgefragt oder verglichen werden.

Ein hohes Ergebnis kann darauf hinweisen, dass ein Operand mehrfach für verschiedene Zwecke eingesetzt wird oder dass der Operand sehr häufig benutzt wird. Eine häufige Nutzung von Operanden könnte durch Einführung einer Schleifenstruktur minimiert werden.

Anzahl unterschiedlicher Aussprünge – NP_NODES

Diese Metrik ermittelt die Anzahl der unterschiedlichen Aussprünge. Da für die Aussprünge in C-Quelltexten 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 Modulende) 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

N_DEAD_LABELS zählt die Sprungmarken, die niemals angesprungen werden.

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.

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. Hierzu zählen neben „Goto“-Anweisungen auch alle Schleifen („For“-, „Do“- und „While“-Anweisungen).

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.

2.4 Maurer

Kommentarzeilen pro Statement in der Komponente – **INLINE_COM**

$$\text{INLINE_COM} = (\text{L_COM} - \text{L_COB}) / \text{N_STMTS}$$

INLINE_COM berechnet die Anzahl der Kommentarzeilen pro Statement in einem Modul. Dabei werden nur die Kommentare im Modul selbst betrachtet, nicht aber die Kommentare, die vor dem Modul stehen und zu dessen Beschreibung dienen.

Kommentarzeilen im Kommentar vor der Komponente – **L_COB**

L_COB zählt die Kommentarzeilen im Kommentar vor der Komponente.

Anzahl von Zeichen pro Kommentarzeile – **C/COM**

Zur Berechnung der Anzahl der Zeichen pro Kommentarzeile werden nur Buchstaben und Ziffern als Zeichen gezählt, um möglichst sinnvolle Kommentare besser zu bewerten. Dies ist eine ganz grobe Methode, da die Semantik eines Kommentars nicht durch eine statische Analyse bewertet werden kann.

Anzahl Kommentarzeilen gesamt – **L_COM**

L_COM misst die Anzahl der Kommentarzeilen vor und in der Komponente.

3 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 auch ebenfalls die Werte 0 oder 1 erhält.

Ein Beispiel:

$$\text{ACCEPTED} = \text{VG} \ \& \ \text{N_IO} \ \& \ \text{VG_STMTS} \ \& \ \text{N_LOOP}$$

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

3.1 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 einem Modul. 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 in Verbindung setzt. Grundlage dafür ist, dass sich ein großes, aber wenig komplexes Modul oder aber ein komplexes, aber dafür sehr kleines Modul möglicherweise trotzdem gut testen lassen. Die Metrik N_IO zählt die Anzahl der Ein- und Ausgänge. Sie fließt mit in das Qualitätskriterium Testbarkeit ein, da es schwieriger ist, Testfälle zu entwickeln, wenn der Kontrollfluss im Modul öfter durch Aussprünge unterbrochen wird.

Akzeptiert wird ein Modul bzgl. seiner Testbarkeit nur, wenn all seine die Testbarkeit betreffenden Metriken in ihren Grenzen liegen. Wenn die Metrik N_IO außerhalb ihrer Grenzen liegt, wird die Aussage TO_INSPECT getroffen. Das Modul sollte auf ihre Ein- und Ausgänge inspiziert werden. Wenn entweder die Metrik N_LOOP oder VG außerhalb ihrer Grenzen liegt, sollte das Modul besser strukturiert werden. Gekürzt werden sollte das Modul hinsichtlich seiner Testbarkeit genau dann, wenn die Metrik VG_STMTS nicht in ihren Grenzen liegt. Damit ergeben sich folgende Formeln:

ACCEPTED = VG & N_IO & VG_STMTS & N_LOOP
 TO_INSPECT = !N_IO
 TO_STRUCTURE = !N_LOOP | !VG
 TO_CUT = !VG_STMTS

3.2 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 ist hat die Metrik AVG_S (durchschnittliche Größe der Instruktionen) eine wichtige Bedeutung für das Qualitätskriterium Einfachheit, da sehr lange und verschachtelte Instruktionen schwer zu verstehen sind.

Akzeptiert wird ein Modul bzgl. der Einfachheit, wenn die Metriken VG_NEU, VG_STMTS und AVG_S innerhalb ihrer Grenzen liegen. Für den Fall, dass alle Metriken ihre Grenze überschreiten, sollte das Modul komplett neu geschrieben werden. Wenn die obere Grenze von VG_STMTS überschritten wird, dann bedeutet das, dass das Modul in Bezug auf seine Komplexität zu groß ist. Aus diesem Grund wird in diesem Fall empfohlen, das Modul zu verkleinern. Wenn die Metriken VG_NEU und/oder AVG_S außerhalb ihrer Grenzen liegen, sollte das Modul hinsichtlich seiner Komplexität untersucht werden. Damit ergeben sich folgende Formeln:

ACCEPTED = VG_NEU & VG_STMTS & AVG_S
 TO INSPECT = !TO_REWRITE & (!VG_NEU | !AVG_S)
 TO CUT = !VG_STMTS
 TO REWRITE = !VG_NEU & !VG_STMTS & !AVG_S

3.3 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 dem Modul. VG_STMTS betrachtet den Umfang eines Moduls im Zusammenspiel mit der Komplexität. Eine großes Modul, das eine niedrige Komplexität hat, kann durchaus noch lesbar sein. Im Gegensatz dazu kann eine kleines Modul, das eine sehr große Komplexität hat, unter Umständen schon nicht mehr lesbar sein. Die Metrik MAX_LVL (Verschachtelungstiefe) spielt für die Lesbarkeit eine Rolle, da ein sehr stark verschachteltes Modul schwerer lesbar ist als ein wenig verschachteltes. Ebenfalls schwer zu lesen sind sehr lange Anweisungen, sodass auch die Metrik AVG_S mit in das Qualitätskriterium Lesbarkeit einfließt.

Wenn alle bzgl. der Lesbarkeit betrachteten Metriken ihre Grenzen erfüllen, ist das Modul akzeptiert. Wenn eine der Metriken VG_NEU oder VG_STMTS oder beide ihre Grenzen überschreiten, wird empfohlen, das Modul im Hinblick auf seine Lesbarkeit zu inspizieren. Besser strukturiert werden sollte das Modul, wenn eine der Metriken

MAX_LVL_S oder AVG_S oder beide ihre Grenzen nicht erfüllen. Damit ergeben sich folgende Formeln:

ACCEPTED = VG_NEU & VG_STMTS & MAX_LVL_S & AVG_S
TO_INSPECT = !VG_NEU | !VG_STMTS
TO_STRUCTURE = !MAX_LVL_S | !AVG_S

3.4 Selbstbeschreibung

Ein Modul hat die Eigenschaft einer guten Selbstbeschreibung dann, wenn es gut kommentiert ist. Mit der Metrik NEU_COM_R wird die Kommentarthäufigkeit in Bezug auf die Komplexität eines Moduls betrachtet. Ein Modul muss weniger kommentiert werden, je einfacher es ist, bzw. im Umkehrschluss mehr kommentiert werden, je komplexer es ist. Die Metrik L_COB betrachtet die Anzahl der Kommentarzeilen im Kommentar vor der Komponente und sagt somit etwas über die allgemeine Beschreibung der Komponente aus. Die Metrik C/COM zählt die Anzahl der Zeichen pro Kommentarzeilen. Liegt die Metrik außerhalb ihrer Grenzen, so sind die Kommentare zu kurz und somit vermutlich nicht aussagekräftig genug.

Akzeptiert wird ein Modul bzgl. der Selbstbeschreibung dann, wenn die Metriken NEU_COM_R, C/COM und L_COB innerhalb ihrer Grenzen liegen. Liegt die Metrik L_COB außerhalb ihrer Grenzen, wird empfohlen, das Modul besser zu beschreiben. Liegt die Metrik C/COM außerhalb ihrer Grenzen, sollten die Kommentare etwas ausführlicher sein. Wenn die Metrik NEU_COM_R nicht in ihren Grenzen liegt, dann scheint das Modul aufgrund seiner Komplexität allgemein nicht ausreichend kommentiert zu sein. Damit ergeben sich folgende Formeln:

ACCEPTED = NEU_COM_R & C/COM & L_COB
DESCRIBE_COMPONENT = L_COB
LONGER_COMMENTS = !C/COM
TO_DOCUMENT = !NEU_COM_R

3.5 Globale Qualität

Die Aussagen der globalen Qualität umfassen alle Aussagen der einzelnen Qualitätskriterien. Sobald für ein Modul 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 ein Modul 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 ein Modul neu strukturiert werden muss. Nur wenn für alle Module 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