Wie verifiziert man eine Software-Unit nach IEC 62304?

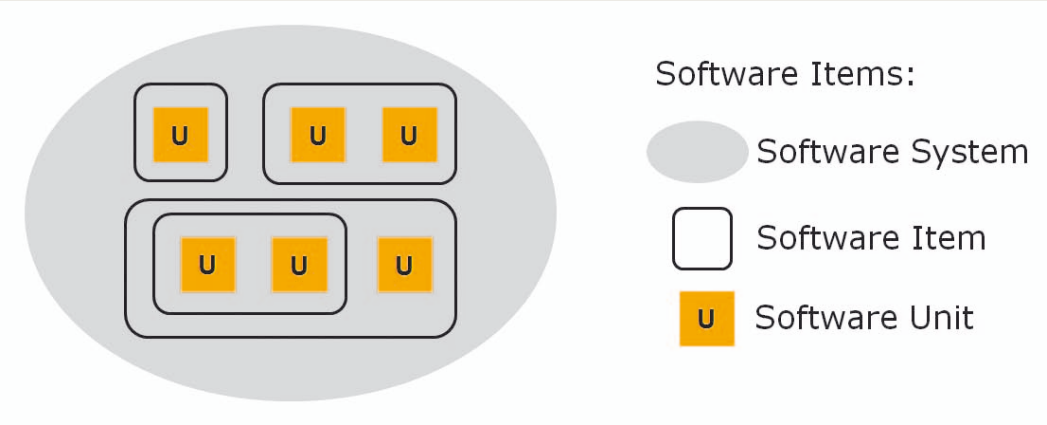

Bild 1: In IEC 62304 besteht die Software aus Elementen („items"); die Software-Elemente auf der untersten Ebene heißen Units

In Abschnitt 5.5.5 fordert die IEC 62304 [Ref1] für Software der Klassen B und C (siehe Kasten) kurz und knapp in einem einzigen Satz: „Der Hersteller soll Software-Unit-Verifikation durchführen und die Ergebnisse dokumentieren." Um diese Forderung erfüllen zu können, muss man die Norm durchforsten, um zunächst zwei Aspekte zu klären: Wie definiert die IEC 62304 eine Software-Unit und was versteht die Norm unter Verifikation?

Kurz gefasst:

Die Norm IEC 62304 fordert die Verifikation von Software-Units, erklärt aber nicht genauer, was exakt zu tun ist. Lediglich ein paar Beispiele für mögliche Akzeptanzkriterien werden aufgelistet. Dieser Artikel diskutiert, was Software-Unit-Verifikation bedeutet und wie Akzeptanzkriterien erfüllt werden könnten

Definition Software-Unit

In Abschnitt 3.25 wird eine Software-Unit als ein Software-Element („item") definiert, das sich auf der untersten Ebene des Software-Systems befindet und das nicht weiter unterteilt wird (Bild 1). In Abschnitt 3.28 findet sich sinngemäß die gleiche Aussage. Wohlgemerkt sagt die Norm nicht, dass eine Software-Unit nicht weiter unterteilt werden kann. Dadurch wird ein Widerspruch vermieden, denn in der Notiz in Abschnitt 3.28 überlässt die Norm dem Hersteller die Definition der Granularität einer Software-Unit. Die Begründung liefert Abschnitt B.3: Der Begriff wird absichtlich vage gehalten, damit die Norm für unterschiedlich große (Software-) Projekte und unterschiedliche Entwicklungsmethoden angewendet werden kann.

Eine weitere Charakterisierung für eine Software-Unit nennt Abschnitt B.5.4: Software-Units können separat getestet werden. Insgesamt hat man also die Merkmale „nicht weiter unterteilt" und „separat testbar". Konkreter kann man werden, wenn man die verwendete Programmiersprache heranzieht: Die kleinste Software-Unit in der Programmiersprache C ist eine Funktion im Sinne von C, in den objektorientierten Programmiersprachen wie C++ oder Java kann man auch Methoden als Units betrachten.

Software-Entwurf

Medizinische Software für eingebettete Systeme wird oftmals in der Programmiersprache C geschrieben. Deshalb betrachten wir im Folgenden diese Programmiersprache, um konkret werden zu können. Welche Units es gibt und welche Funktionalität sie haben sollen, wird für Software der Klassen B und C durch den ausführlichen Software-Entwurf („Software detailed design", Abschnitt 5.4) festgelegt. Der Software-Entwurf ergibt die Software-Architektur. Eine gute Architektur ist durch schmale Schnittstellen zwischen den Units und eine vernünftige Größe der Units gekennzeichnet. Dabei kann eine Unit auch mehrere C-Funktionen umfassen.

Zwei Funktionen zusammenfassen

In Bild 2 benutzt die Funktion sum_up() die Funktion signum(). Es ist vertretbar, die beiden Funktionen als eine Unit zu betrachten. Ein Grund dafür ist, dass ein sinnvoller Software-Unit-Test von sum_up() kaum bestanden werden kann, sollte signum() einen Fehler enthalten. Weiterhin ist der Aufwand für einen isolierten Test

Autor:
Dipl.-Inform. Frank Büchner
Principal Engineer
Software Quality
Hitex GmbH
www.hitex.com

```
// Sum up the first n array elements, but only if they are positive.
// (We assume the first n array elements exist.)
int sum_up(int a[], int n)
{
   int i, sum;
   for(i = 0, sum = 0; i < n; i++)
       sum += (1 + signum(a[i]))/2 * a[i];
   return sum;
}
```

```
int signum(int x)
{
   if (x > 0) return 1;
   if (x == 0) return 0;
   if (x < 0) return -1;
}
```

Bild 2: Es ist vertretbar, die beiden C-Funktionen sum_up() und signum() zu einer Software-Unit zusammen zu fassen

von sum_up() im Verhältnis zum erwarteten Erkenntnisgewinn zu hoch, weil beim isolierten Test der Aufruf von signum() durch eine Platzhalterfunktion (stub) ersetzt werden muss, die das Verhalten von signum() simuliert. Ein separater Software-Unit-Test von signum() ist technisch nicht aufwendig, und obwohl der Test voraussichtlich ohne Überraschungen verläuft, weil signum() sehr einfach ist, sollte der geringe Testaufwand erbracht werden, auch wenn er nicht unbedingt notwendig ist.

Definition Verifikation

Die IEC 62304 definiert in Abschnitt 3.33 Verifikation als „Bestätigung durch Bereitstellung eines objektiven Nachweises, dass spezifizierte Anforderungen erfüllt wurden". Diese Anforderungen werden beim Software-Entwurf spezifiziert, der laut Abschnitt 5.4.1 für Software der Klassen B und C durchgeführt und dokumentiert werden muss. Hierbei werden die Anforderungen an die Gesamt-Software auf die Units verteilt, was letztendlich die Beschreibung der Funktionalität einer Software-Unit ergibt. Für Software der Klassen B und C muss der Entwurf so ausführlich sein, dass die korrekte Implementierung der Software-Unit möglich ist. Laut Abschnitt 5.4.3 muss für Software der Klasse C ein so ausführlicher Entwurf auch für die Schnittstellen dokumentiert werden, sowohl für die Schnittstellen zu externen Komponenten (aus Hardware oder Software) als auch für die Schnittstellen zwischen Software-Units. Insbesondere für Software der Klasse C sind somit die Anforderungen ausreichend detailliert dokumentiert, dass man nun beginnen kann, den Nachweis Ihrer Erfüllung bereit zu stellen.

Diese Bestätigung bzw. Verifikation der Software-Units wird laut Abschnitt 5.5.2 der IEC 62304 durch einen Prozess aus Strategien, Methoden und Prozeduren erbracht. Testen ist offensichtlich in diesen Methoden enthalten, da dazu für Software der Klassen B und C die Angemessenheit der Testprozeduren beurteilt werden muss (ebenfalls laut Abschnitt 5.5.2). Drei Beispiele für Akzeptanzkriterien für Software-Units der Software-Klassen B und C gibt der folgende Abschnitt 5.5.3 der IEC 62304 in der Notiz. Diese drei Beispiele werden im Folgenden diskutiert.

Akzeptanzkriterium Anforderungen

Implementiert der Software-Code Anforderungen inklusive der Risikokontrollmaßnahmen? Dies prüft man typischerweise durch geeignete Testfälle. Gibt es zu einer Anforderung mindestens einen Testfall, der sie prüft und ist dieser Testfall durchgeführt und bestanden, so implementiert der Code diese Anforderung. Erstaunlicherweise fordert die IEC 62304 nicht explizit, dass alle Anforderungen durch den Software-Code implementiert sein müssen. Geforderte Risikokontrollmaßnahmen (beispielsweise die Vermeidung von NULL-Pointer-Zugriffen) im Software-Code kann man in diesem Zusammenhang wie Anforderungen prüfen.

Die Zuordnung von Anforderungen zu Testfällen ist eine manuelle/menschliche Tätigkeit, eine automatische Zuordnung ist im Allgemeinen nicht möglich. Hat man diese Zuordnung jedoch in einem geeigneten Werkzeug hergestellt, so kann man werkzeuggestützt ermitteln, ob alle Anforderungen Testfälle haben und ob diese bestanden sind.

Unit-Test-Werkzeug

Bild 3 zeigt eine Abdeckung der Anforderungen durch Testfälle im Werkzeug TESSY [Ref7]. TESSY ist ein Unit-Test-Werkzeug, das auch Anforderungen und ihre Beziehung zu Testfällen verwalten kann. Die Segmente des Kreises stehen für Anforderungen, die dazugehörige Zahl gibt die Anzahl der Anforderungen im jeweiligen Segment an. Der Status der Anforderungen in einem Segment wird farblich bzw. durch die Schraffur angezeigt. Beispielsweise gibt es zwei Anforderungen (im grau-rot-gestreiften Segment dargestellt), für die Testfälle bereits ausgeführt wurden, aber manche davon sind fehlgeschlagen.

Das Ziel ist ein vollständig grün ausgefüllter Kreis, der anzeigt, dass alle Anforderungen mit Testfällen assoziiert sind und dass alle diese Testfälle ausgeführt wurden und bestanden sind. Natürlich erzeugt TESSY die Information über die Überdeckung von

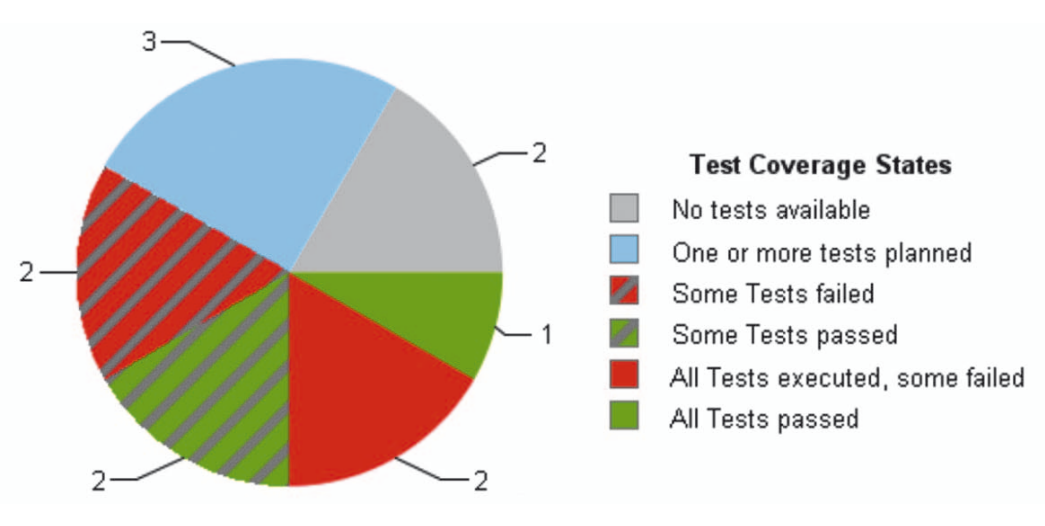

Bild 3: Darstellung der Überdeckung der Anforderungen durch Testfälle im Werkzeug TESSY [Ref7]

Interface of 'func'

Filter interface elements

	Passing
External Functions	
Local Functions	
External Variables	
• char c	IN
Global Variables	
• int m	OUT
• long s	INOUT
Parameter	
• int k	IN
Return	
• int	OUT

Bild 4: Interface der Software-Unit func

Anforderungen durch Testfälle auch als Liste.

Akzeptanzkriterium Interface-Design

Ist der Software-Code ohne Widersprüche zum Interface-Design der Software-Unit? TESSY kann das Interface einer Software-Unit aus dem Software-Code ermitteln und darstellen. Damit kann man prüfen, ob der Software-Code der Software-Unit Widersprüche zum Interface-Design der Software-Unit enthält.

In Bild 4 ist die Struktur des Interfaces der Software-Unit func() dargestellt. Es zeigt im Wesentlichen die Variablen und ihre Passierrichtung. Offensichtlich besteht das Interface aus einer externen Variablen c, deren Passierrichtung IN ist. IN bedeutet, dass die Variable von der Software-Unit nur gelesen wird, deshalb muss für diese Variable vor dem Test ein Eingabewert spezifiziert werden. Ferner gibt es die globale Variable m, deren Passierrichtung OUT ist. Bei dieser Passierrichtung muss für einen „guten" Test ein erwarteter Wert vorgegeben werden, der nach dem Test mit dem tatsächlichen Wert verglichen wird. Für die globale Variable s benötigt man sowohl einen Eingabewert vor dem Test als auch einen erwarteten Ergebniswert, denn diese Variable hat die Passierrichtung INOUT. Die Interface-Variablen können natürlich auch Parameter sein (im Bild 4 der Parameter k) und auch der Return-Wert, so es denn einen gibt, gehört zum Interface. Auch die Datentypen der Interface-Variablen gehören zum Interface.

Üblicherweise liegt die Interface-Beschreibung aus dem ausführlichen Software-Entwurf (Abschnitt 5.4.3 der IEC 62304) nicht in maschinenlesbarer Form vor, deshalb bleibt nur ein Review zur Prüfung ob das im Code implementierte Interface der Software-Unit mit dem Interface aus dem Entwurf übereinstimmt.

Akzeptanzkriterium Programmierverfahren und Codier-Standards

Ist der Software-Code konform zu Programmierverfahren und Codier-Standards? Abschnitt B.5.5 empfiehlt, Codier-Standards zu nutzen, um durchgehend die gewünschten Merkmale des Codes zu erreichen. Als Beispiele für Codier-Standards werden (a) Anforderungen für Verständlichkeit, (b) Regeln zur Benutzung der Programmiersprache oder Einschränkungen bei der Verwendung der Programmiersprache und (c) das Management der Komplexität genannt.

Regeln zur Benutzung der Programmiersprache ergeben einen (projekt-) einheitlichen Programmierstil, was die Verständlichkeit des Codes fördert, was wichtig ist, wenn unterschiedliche Personen den Code (weiter-) entwickeln und Reviews durchführen. Solche Regeln zur Benutzung der Programmiersprache betreffen üblicherweise Dinge wie Namensgebung (z. B. Höcker-Schrift, Unterstriche in Bezeichnern), Einrückungen, das Setzen von (geschweiften) Klammern und ähnliches. Leider gibt es keine allgemein anerkannte Vorgabe, und deswegen gibt es auch kein vorgefertigtes Werkzeug, das ohne Anpassung alle Regeln prüfen kann, die in irgendeinem bestimmten Projekt gelten. Verletzungen dieser Regeln (beispielsweise uneinheitlich Namensgebung) beeinflussen das Verhalten der Software nicht negativ. Es gibt aber auch vorgefertigte Sätze von Codier-Regeln, die die Programmiersprache einschränken. Deren Verletzung kann sehr wohl problematisch sein, weil durch diese Einschränkungen potentielles Fehlverhalten von Software verhindert werden soll.

CERT C Coding Standard

Der CERT C Coding Standard des Software Engineering Institute (SEI) der Carnegie Mellon Universität (Pittsburgh, PA), der Joint Strike Fighter Air Vehicle (JSF AV) Standard für C++ von Lockheed Martin oder die MISRA-Regeln [Ref5] sind Beispiele hierfür. Die MISRA-Regeln für die Programmiersprache C gibt es schon seit 1998 und aktuell gibt es bereits die dritte Version mit einem Zusatz. Mit den MISRA-Regeln soll der Programmiersprache C ihre Gefährlichkeit genommen werden, denn in C ist es relativ leicht etwas zu programmieren, was etwas anderes bewirkt als es den Anschein hat (siehe Bild 5). Das liegt an dem unspezifizierten, undefinierten und compiler-abhängigen Verhalten, das in der Programmiersprache C vorkommen kann und worauf statische Analysewerkzeuge hinweisen können.

Beispiel sizeof

Wer denkt, dass der Wert der Variablen v1 in Bild 5 nach Ausführung der Zeile 85 wegen dem Inkrement-Operator (++) vom Wert 0 auf den Wert 1 erhöht wurde, der irrt sich. Der Grund liegt darin, dass der sizeof-Operator zur Übersetzungszeit ausgewertet wird, denn da ist die Größe von v1, um die es hier geht, ja bereits bekannt. Der Inkrement-Operator fällt zur Ausführungszeit sozusagen unter den Tisch. Deshalb hat nach der Zeile 85 die Variable v1 immer noch den Wert 0 (und die Variable s1 hat den Wert 4). Solche Probleme sollen durch Einhalten der Regel 13.6 aus MISRA-C:2012 vermieden werden. Diese Regel besagt, dass der Operand des sizeof-Operators keinen Ausdruck mit möglichen Seiteneffekten haben darf. Denn solche Ausdrücke bewirken

```
84    uint32_t v1 = 0;
85    s1 = sizeof(v1++);
86
```

Bild 5: Das statische Analyse-Werkzeug Klocwork [Ref8] zeigt ein potentielles Problem in Zeile 85

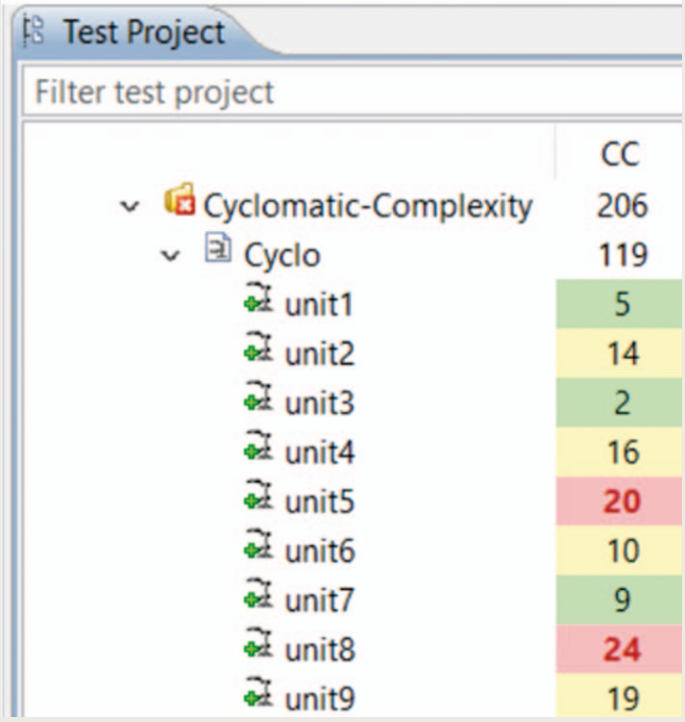

Bild 6: Die zyklomatische Komplexität für Software-Units im Werkzeug TESSY

keinen Seiteneffekt, obwohl es so aussieht. Auf die Verletzung dieser Regel wurde durch das statische Analysewerkzeug Klocwork [Ref8] hingewiesen.

Die Prüfung von MISRA-Regeln ist sehr zu empfehlen, sie ist allerdings nur mit einem Werkzeug sinnvoll. Jedoch können viele MISRA-Regeln problemlos eingehalten werden.

Komplexität

Als letztes Beispiel für Codier-Regeln nennt Abschnitt B.5.5 das Management der Komplexität. Zu diesem Zweck kann man sicherstellen, dass bestimmte Metriken bestimmte Werte nicht überschreiten. Beispielsweise kann man mit Hilfe der Metrik Lines-of-Code (LOC), die die Zahl der Zeilen in einer Software-Unit angibt, verhindern, dass zu große Software-Units entstehen. Eine Metrik, die häufig zur Messung der Komplexität eingesetzt wird, ist die zyklomatische Komplexität nach McCabe [Ref6]. Diese Metrik basiert auf dem Kontrollflussgraph der Software-Unit und gibt die Anzahl der vollständigen, linear unabhängigen Pfade durch sie an. Dies ist auch die Anzahl der binären Entscheidungen plus eins in einer Software-Unit. Leider berücksichtigt diese Metrik weder Berechnungen noch zusammengesetzte Entscheidungen, so dass man überraschend niedrige Werte bei umfangreichen Berechnungen und komplizierten Entscheidungen erhält.

Beispiel zyklomatische Komplexität

Bild 6 zeigt die zyklomatische Komplexität für Software-Units im Werkzeug TESSY. In der Spalte CC ist der Wert für neun Units dargestellt. Die Werte sind farblich unterlegt: Ist der Wert grün unterlegt, so ist er kleiner als der Schwellenwert für eine Warnung, der im vorliegenden Beispiel auf den Wert 10 eingestellt ist; rot unterlegte Werte sind größer oder gleich dem Schwellenwert für einen Fehler, der im vorliegenden Beispiel auf den Wert 20 eingestellt ist; gelb unterlegte Werte befinden sich zwischen den beiden Schwellen.

Weitere Akzeptanzkriterien

In Abschnitt 5.5.4 werden für Software der Klasse C noch weitere Akzeptanzkriterien vorgegeben, beispielsweise zum Daten- und Kontrollfluss oder zu Grenzwerten in Testfällen. Nähere Erläuterungen gibt es jedoch diesbezüglich nicht. Es fehlen darüber hinaus in IEC 62304 gänzlich Akzeptanzkriterien, die in anderen Normen wie IEC 61508 [Ref3] oder ISO 26262 [Ref4] wichtig sind, beispielsweise die Messung der durch die Tests der Software-Unit erreichten Codeüberdeckung oder die Methoden, wie man zu Testfällen für die Software-Unit-Verifikation kommt.

Literatur

[Ref1] IEC 62304, Edition 1.1, 2015-06, VDE Verlag GmbH

[Ref2] Prinz, T. [Hrsg.]: Entwicklung und Herstellung medizinischer Software, VDE-Schriftenreihe Normen verständlich, Nr. 171, VDE-Verlag, Berlin, 2017.

[Ref3] DIN EN 61508 (VDE 0803): 2011-02, DIN Deutsches Institut für Normung e.V. und VDE Verband der Elektrotechnik, Elektronik, Informationstechnik e.V. (IEC 61508:2010).

[Ref4] ISO 26262, International Standard, Road vehicles – Functional Safety, First edition, 2011

[Ref5] MISRA-C:2012, Guidelines for the use of the C language in critical systems, 2013, MIRA Limited, Nuneaton, UK.

[Ref6] Kan, Stephen H.: Metrics and Models in Software Quality Engineering, 1995, Addison-Wesley, Reading, Mass., USA.

[Ref7] http://www.hitex.de/tessy: Mehr über das Unit-Testwerkzeug TESSY

[Ref8] http://www.hitex.com/klocwork: Mehr über das statische Analysewerkzeug Klocwork ◄

Software-Sicherheits-Klassen

Aufgrund Abschnitt 4.3 der IEC 62304 muss der Hersteller dem Software-System eine Software-Sicherheits-Klasse zuordnen. Dabei geht es, vereinfacht gesagt, um die Frage, ob das Software-System zu einer gefährlichen Situation (für Personen) beitragen kann und wenn ja, ob das daraus entstehende Risiko - jeweils nach Risiko-Kontroll-Maßnahmen außerhalb der Software - akzeptabel ist und wenn nicht, welcher Schaden entstehen kann.

Klasse A:

Das Software-System kann nicht zu einer gefährlichen Situation beitragen oder die gefährliche Situation birgt kein inakzeptables Risiko.

Klasse B:

Das Software-System kann zu einer gefährlichen Situation beitragen und die gefährliche Situation birgt inakzeptables Risiko und der mögliche Schaden ist nicht-ernsthafte Verletzung.

Klasse C:

Das Software-System kann zu einer gefährlichen Situation beitragen und die gefährliche Situation birgt inakzeptables Risiko und der mögliche Schaden ist Tod oder ernsthafte Verletzung.