Wie Dynamische Softwareanalyse die Zulassung von Medizingeräten vereinfachen kann

Autoren: Mark Pitchford, Field Application Engineer, LDRA
Chris Ault, Product Marketing Manager, Medical, QNX Software Systems
mark.pitchford@ldra.com, cault@qnx.com

„Vorsicht, in meinem Code könnten noch Bugs stecken – ich habe lediglich seine Korrektheit bewiesen, ihn aber nicht ausprobiert." – Donald Knuth

Einleitung

Alle Hersteller von Software für Medizingeräte haben die gleichen Probleme, wie jeder, der komplexe Systeme entwirft: Zeit, Qualität, Größe (Anzahl und Komplexität der Features) und Kosten. Dazu kommen noch Zulassungen nach MDD, bzw. durch die FDA, MHRA, Health Canada und den entsprechenden Institutionen in den Zuständigkeitsbereichen, in denen das Gerät eingesetzt werden wird.

In diesem Artikel beleuchten wir, wie a) dynamische Codeanalyse den Nachweis der Einhaltung von Sicherheitsanforderungen unterstützen kann und b) welche Fähigkeiten das Werkzeug zur dynamischen Codeanalyse besitzen sollte. Zudem bieten wir im Anhang für die Auswahl eines Werkzeuges Tabellen, die bestimmte Entwicklungstätigkeiten den Anforderungen des IEC 62304 Standards gegenüberstellen. Zuletzt beschreiben wir wichtige Charakteristika für die Auswahl eines Betriebssystems, welche Entwurf, Entwicklung und Zulassung einer sicherheitskritischen Software deutlich erleichtern können.

Kompetenz und Prozesse

Fachkompetenz und saubere Entwicklungsprozesse allein sind noch keine Garantie dafür, dass ein System einen geforderten Level an Zuverlässigkeit erreicht, noch nicht einmal, dass es ein gutes Produkt wird. Allerdings erhöhen sie die Chancen darauf erheblich.

Erfahrung und Können benötigt man vor allem für einen möglichst einfachen Entwurf des sicherheitskritischen Systems. Ein umfassendes Verständnis für Methoden der Softwarevalidierung, der zu untersuchenden Software und den Kontext der Evaluierung (inklusive der Validierung ähnlicher Systeme) sind erforderlich, um nachweisen zu können, dass die Software die Sicherheitsanforderungen erfüllt.

Es ist kein Zufall, dass sich IEC 62304 auf den Entwicklungsprozess konzentriert. Wir sollten darauf achten, unsere Software nicht nur in einer Umgebung zu entwickeln, die höchsten Qualitätssicherungsstandards entspricht, sondern auch Werkzeuge einzusetzen, die sicherstellen, dass wir den Standards entsprechen und dies den Zulassungsstellen auch nachweisen können.

Nachweis der Verlässlichkeit

Für eine Zulassung müssen die Hersteller nachweisen, dass ihr Gerät den Sicherheitsanforderungen entspricht. Für die Gerätesoftware bedeutet das den Nachweis der Verlässlichkeit (Zuverlässigkeit und Verfügbarkeit). Ob Zuverlässigkeit oder Verfügbarkeit wichtiger sind, hängt stark von der Verwendung des Gerätes ab. Präzise definierte

Anforderungen an die Zuverlässigkeit schaffen einen definierten Kontext und messbare Kriterien, gegen welche wir die Verlässlichkeit eines Softwaresystems validieren können.[1]

Akzeptables Risiko definieren

Kein Softwaresystem ist absolut zuverlässig – und wenn doch, könnten wir das nicht nachweisen. Alle bekannten Techniken können letztlich nicht beweisen, dass nie ein Fehler auftreten könnte. Sie können uns nur helfen, Fehler zu finden und zu beseitigen und die Wahrscheinlichkeit eines Versagens abzuschätzen. Somit gilt eine Software als „sicher", wenn die Wahrscheinlichkeit des Versagens niedrig genug ist, um kein inakzeptables Risiko darzustellen. Doch was genau unter „inakzeptablem" oder „akzeptablem" Risiko zu verstehen ist, wird je nach Geräteart und Gesetzeslage unterschiedlich gesehen. Mögliche Definitionen:

ALARP (*As Low as Reasonably Practical*): Die potentiellen Gefährdungen werden identifiziert und wie folgt klassifiziert: a) nicht akzeptabel, b) tolerierbar, sofern die Kosten zur Beseitigung untragbar sind und c) akzeptabel. Alle nicht akzeptablen Risiken müssen beseitigt werden, die tolerierbaren nur, sofern Zeit und Kosten sich rechtfertigen lassen.

GAMAB (*globalement au moins aussi bon*) oder GAME (*globalement au moins équivalent*): Das Gesamtrisiko des Systems darf das Gesamtrisiko vergleichbarer Systeme nicht überschreiten.

MEM (*Minimum Endogenous Mortality*): Das Risiko durch das neue System darf nicht ein Zehntel der normalen Sterblichkeit der Region übersteigen. Für Menschen zwischen 20 und 30 in westlichen Ländern wäre dieser Wert 0,0002.

Alle diese Verfahren müssen entsprechend der Anzahl der im Fall des Systemversagens gleichzeitig gefährdeten Personen angepasst werden.

Bei ALARP müssen wir für die Unterteilung in „nicht akzeptable", „tolerierbare" und „akzeptable" Risiken Werte für das Maximum der Wahrscheinlichkeit des Auftretens eines Versagens je Risiko ermitteln. Für GAMAB und MEM müssen wir diese Zahl als einen Gesamtwert ermitteln.

Techniken zum Nachweis der Verlässlichkeit

Es gibt nicht eine einzelne Technik, die für den Nachweis der Verlässlichkeit eines Softwaresystems ausreichen würde. Unser Nachweis der Verlässlichkeit muss deshalb mit einem ganzen Arsenal an Strategien und Techniken geführt werden. Dazu zählen unter anderem (aber nicht ausschließlich):

- Eine Entwicklungsumgebung entsprechend dem Standard IEC 62304 oder vergleichbaren Standards
- Genaue Anforderungsspezifikationen um sicherzustellen, dass alle sicherheitsrelevanten Anforderungen abgedeckt werden
- Formale Entwurfsmethoden und Werkzeuge zum mathematischen Nachweis der Korrektheit des Entwurfs
- Fehlerbaum-Analysen mit Mitteln wie z.B. bayesschen Netzen

[1] Siehe auch Chris Hobbs, et al. "Building Functional Safety into Complex Software Systems", Teil I und II. QNX Software Systems, 2011. www.qnx.com.

- Retrospektive Entwurfsvalidierung, welche das Systemdesign auf Basis der tatsächlichen Software evaluiert
- Statische Analysemethoden wie Modellprüfung und Datenflussanalyse
- Tests mit direkter Fehlersuche, wie dynamische Analysen zur Identifizierung von Programmfehlern über die Störungen und Systemausfälle, die sie verursachen

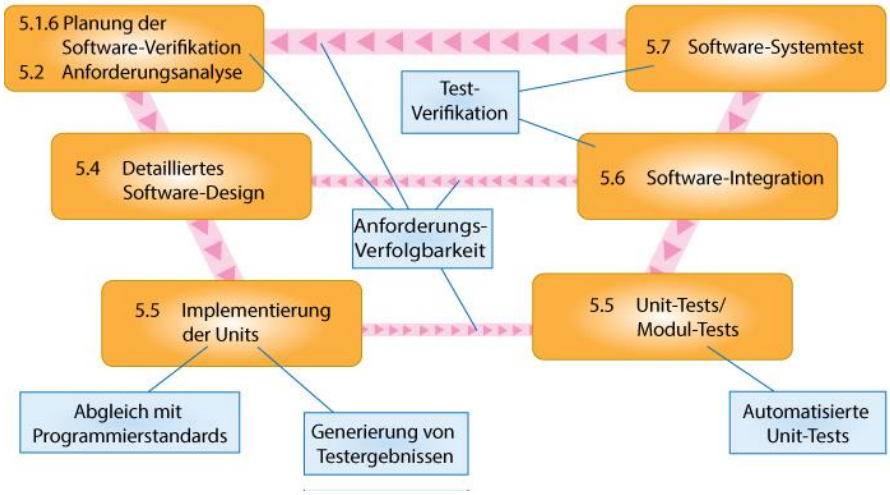

Abbildung 1. *Verschiedene Analysetechniken und die zugehörigen IEC 62304 Abschnitte in einem typischen „V-Modell“. Keine der abgebildeten Techniken hängt vom gewählten Prozess ab; man hätte eine äquivalente Darstellung auch mit einem anderen Entwicklungsmodell machen können, z.B. Wasserfall, Iterativ, Agil.*

IEC 62304

IEC 62304 hat sich inzwischen als globaler *de facto* Standard für Software-Lebenszyklusprozesse in der Medizintechnik etabliert. Die Entwicklung wurde vor allem durch die FDA vorangetrieben und inzwischen mit dem EU-Standard 93/42 EWG (MDD) abgestimmt.[2]

Wie die anderen Standards in Abbildung 2 setzt IEC 62304 auf in der Industrie etablierte Praktiken als Ergänzung zu den Richtlinien von IEC 61508. Beispielsweise definiert IEC 62304, anders als ISO 26262 oder IEC 61508, keine allgemeinen Zahlenwerte für akzeptierte Fehlerraten (Sicherheits-Integritätslevel, SIL). Stattdessen werden Sicherheitsklassifizierungen nach der Höhe des Schadens definiert, der durch ein Versagen einem Patienten, dem Bedienpersonal oder einer anderen Person zugefügt würde. Diese Klassifizierungen entsprechen den FDA-Klassifizierungen für Medizingeräte: A (keine Schädigung der Gesundheit), B (Möglichkeit einer leichten Schädigung) und C (Möglichkeit einer schwerwiegenden Schädigung oder Todesfolge).

[2] Cristoph Gerber, “Introduction into software lifecycle for medical devices”, Stryker Navigation: Presentation (4. September 2008)

Allgemein ähneln sich die von IEC 61508 abgeleiteten Standards darin, dass sie die Prozesse (inklusive eines Prozesses zum Risikomanagement), Aktivitäten und Aufgaben für den gesamten Lebenszyklus der Software festlegen. Dabei wird Wert darauf gelegt, dass der Lebenszyklus keineswegs mit der Produktfreigabe endet, sondern Wartung und Fehlerbehebung beinhaltet, solange die Software im Einsatz ist. Unabhängig davon, wie akzeptable und inakzeptable Risiken spezifiziert werden, letztlich dienen Standards wie IEC 62304 oder IEC 61508 als Leitfaden und Messlatte, die wir für den Nachweis verwenden müssen, dass unsere Software die Sicherheitsanforderungen erfüllt.

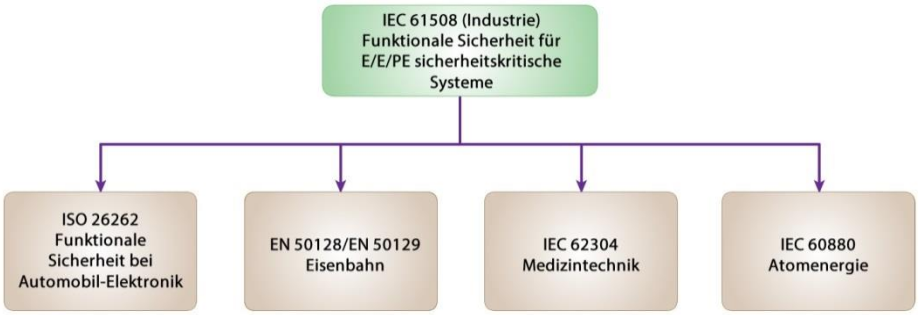

Abbildung 2. *IEC 62304 leitet sich von IEC 61508 ab und besitzt somit gemeinsame Wurzeln mit anderen Industriestandards. Im Standard IEC 62304 wird ausdrücklich darauf hingewiesen, dass er nicht vom Standard IEC 61508 abhängt, aber dass man bezüglich Werkzeuge und Techniken auf IEC 61508 zurückgreifen kann.*[3]

Dynamische Analyse

Dynamische Analysen werden zum Test des übersetzten Quellcodes verwendet – entweder für das gesamte Programm oder für einzelne Teile. Da bei der Analyse Code ausgeführt wird, testet man nicht nur den Quellcode, sondern auch Compiler, Linker, Entwicklungsumgebung und ggf. die Zielhardware. Allgemein umfasst dynamische Analyse strukturelle Überdeckungsanalysen und Unit-Tests. Gemeinsam stellen diese nicht nur eine effektive Methode zum Aufspüren von Fehlern dar, sondern auch einen Nachweis, welche Software wie ausgeführt und getestet wurde.

Strukturelle Überdeckungsanalyse ist übrigens das Fundament des Luftfahrtindustrie-Standards DO-178B/C. Auch wenn Flugunfälle häufig tragisch enden und es deshalb eher in die Nachrichten schaffen als Unfälle mit Medizingeräten, setzt die Luftfahrtindustrie doch die höchsten Standards für Sicherheit: Bezogen auf die Wegstrecke ist Fliegen eines der sichersten Transportmittel.

Strukturelle Überdeckungsanalyse

Dynamische Code-Analyse kann sowohl intrusiv als auch nicht-intrusiv erfolgen. Ein intrusiver Test fügt Prüfsoftware für die Analyse (Zähler oder Funktionsaufrufe) in den eigentlichen Code (Hochsprache oder Assembler) ein. Diese Codefragmente sammeln Daten über die Prozessausführung und legen Historien zum Programmablauf an.

[3] Anhang C.

Intrusive und nicht-intrusive Tests

Bei der Verwendung von intrusiven Testmethoden ist für die Gültigkeit der Analyse wichtig sicherzustellen, dass die Prüfsoftware die Funktionalität des instrumentierten Codes nicht verändert. Neben dem Nachweis, dass intrusive Tests den eigentlichen Code nicht verändern, muss man üblicherweise auch nachweisen, dass der eingefügte Testcode nicht zu Fehlern bei der Übersetzung führt. Dazu kann man eine Compiler Validation Suite (eine Sammlung von Quellcode-Beispielen, die extra zum Nachweis der korrekten Übersetzung durch den Compiler entworfen wurden) verwenden, um zu zeigen, dass die Integrität des Compilers nicht durch die Instrumentierung beeinträchtigt wird.

Ein nicht-intrusives System ermittelt die gleichen oder ähnliche Daten wie ein intrusives System, aber direkt vom Prozessor. Das Analyse-Werkzeug setzt dann diese Low-Level Informationen wieder in Beziehung zur ursprünglichen Darstellung des Programms (Hochsprache oder Assembler). Leider ist eine eindeutige Zuordnung aus vielerlei Gründen (wie z.B. Optimierung durch den Compiler) nicht immer möglich.

Allerdings ist es, wie bei allen Tests, für ein komplexes Softwaresystem nicht möglich, zu 100% nachzuweisen, dass der Prüfcode zur strukturellen Überdeckungsanalyse das Verhalten des Programms nicht verändert. Beispielsweise sind „Heisenbugs“ per Definition nicht reproduzierbar; sie werden häufig durch subtile Veränderungen im Laufzeitverhalten verursacht und können allein schon durch die Instrumentierung des Codes behoben (oder aber verursacht!) werden.

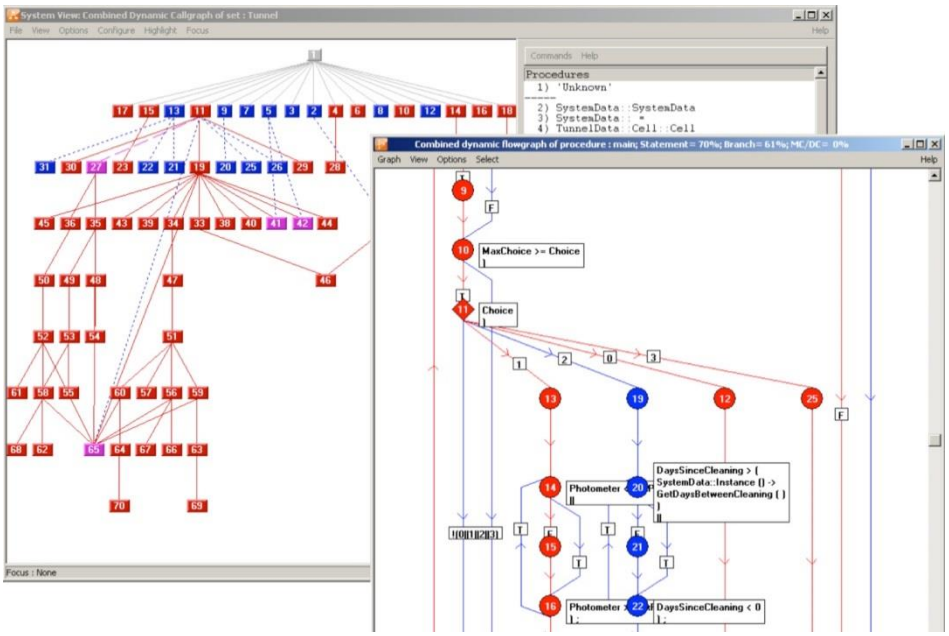

Abbildung 3. Bei diesen Screenshots eines LDRA-Werkzeuges zur Überdeckungsanalyse identifizieren farbige Graphen die nicht ausgeführten Codeteile.

Nachweis für die Abschätzung der Verlässlichkeit

Der Trick ist nun nicht nachweisen zu wollen, dass die Software komplett fehlerfrei ist (was formal unmöglich ist), sondern Nachweise für die Abschätzung der Verlässlichkeit unserer Software zu liefern. Wenn wir in unserem System beispielsweise SOUP (Software Of Unknown Pedigree, Software unbekannter Herkunft) verwenden, kann die strukturelle Überdeckungsanalyse nachweisen, dass es keinen nicht verwendeten oder überflüssigen Code gibt, wie es z.B. ANSI/AAMI/IEC TIR80002-1:2009[4] in Tabelle B.2 fordert: „Verwenden Sie nur die benötigten SOUP-Features und entfernen Sie alle übrigen“.

Unit-Tests

Unit-Tests überprüfen kleine Programmeinheiten und machen es relativ einfach, nicht korrektes Verhalten einzugrenzen und somit Fehler aufzuspüren. Bei Unit-Tests werden einzelne Funktionen oder eine Sammlung von Funktionen isoliert vom Gesamtsystem getestet, um nachzuweisen, dass bestimmte Anforderungen erfüllt werden.

Üblicherweise werden die Anforderungen sogar schärfer definiert als im Kontext des Projektes, damit man z.B. extreme Randbedingungen testen kann: Ein Test für das Zeichnen auf einem Display mit 750 x 1000 Pixel könnte z.B. mit einer Auflösung bis zu 1200 x 1600 Pixel durchgeführt werden. Die Schnittstelle jeder Funktion könnte für Eingabewerte getestet werden, die normalerweise von den aufrufenden Programmteilen gar nicht zugelassen werden. So kann man nachweisen, dass sich Funktionen immer wie gefordert verhalten.

Unit-Tests können auch gezielt Code testen, der in einem normalen Testszenario womöglich gar nicht aufgerufen wird, z.B. Funktionen zur Freigabe von Systemressourcen. Man kann auch Fälle aufspüren, in denen sich ein fehlerhaftes Programm zufällig korrekt verhält, z.B. wenn eine Funktion aufgerufen wird, die gar nicht aufgerufen werden sollte, der Beobachter aber den Eindruck hat, dass alles korrekt abläuft. Da man kleinere Komponenten testet, kann man inkorrektes Verhalten direkter beobachten und so Fehler leichter aufspüren.

Die Frage, wie man die Funktionsaufrufe für die zu testende Einheit gestaltet, hängt vom Ziel des jeweiligen Tests ab. Üblicherweise verwenden Unit-Tests eine Bottom-Up Teststrategie (auch Modul- oder Integrationstests genannt), bei der einzelne Einheiten getestet und schrittweise in weitere Units integriert werden. Werden aufgerufene Funktionen aus dem Test ausgenommen, so können sie durch „Funktions-Stubs“ ersetzt werden.

[4] Guidance on the application of ISO 14971 to medical device software

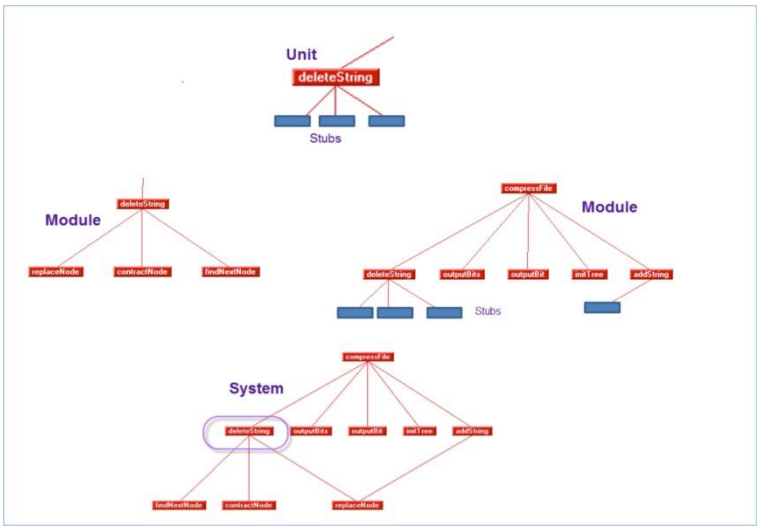

Abbildung 4. *Die Durchführung einer strukturellen Abdeckungsanalyse am Gesamtsystem und an Einzelkomponenten bietet maximale Flexibilität*

In Kombination mit struktureller Überdeckungsanalyse hat man somit die Flexibilität, so viele – oder so wenige – Teile des Aufrufbaums in die Tests zu integrieren wie gewünscht. Damit wird man in die Lage versetzt, Abdeckung selbst für strengste Zulassungen zu erreichen.

Auswertung der strukturellen Testabdeckung

Eine der schwierigsten Fragestellungen bei jeder Systemvalidierung ist zu entscheiden, wann man ausreichend getestet hat. Diese Entscheidung sollte man im Kontext mit den Anforderungen an die Verlässlichkeit des Systems treffen. Letztlich hängt es von der IEC 62304 und der Sicherheitsklassifizierung der Zulassungsstellen für das Gerät ab, das unsere Software verwendet.

Die Daten aus der Testabdeckung helfen bei der Abschätzung, wie umfassend die dynamischen Tests waren und wie viel Testarbeit noch zu tun bleibt. Diese Daten sind u.a.:

- Anweisungsüberdeckung: Diese einfachste Metrik ist der Anteil der ausgeführten Anweisungen am Gesamtsystem
- Bedingungs-/Entscheidungsüberdeckung: Der Anteil der abgedeckten Zweige in Kontrollstrukturen. Im Mittel wird jede Anweisung und jede Funktion doppelt so oft ausgeführt wie bei der Anweisungsüberdeckung.
- LCSAJ Coverage (strukturierter Pfadtest): LCSAJ (linear code sequence and jump) Coverage ist anspruchsvoller als die Entscheidungsüberdeckung und sollte für die kritischsten Teile der Software verwendet werden. LCSAJ wird von modernen Werkzeugen unterstützt.
- Modifizierter Bedingungs-/Entscheidungsüberdeckungstest (MC/DC Coverage): Volle MC/DC-Abdeckung ist erreicht, wenn jeder Einsprungs- und Aussprungs-Punkt des Programms einmal aufgerufen wurde und jede Verzweigung in einer Kontrollstruktur mindestens einmal erreicht wurde und die Abhängigkeit der Gesamtentscheidung von jeder einzelnen atomaren Teilentscheidung überprüft wurde.

Auswahl des Software-Analyse Werkzeuges

Alle Softwarehersteller wollen selbstverständlich ihre Produkte verkaufen und haben kein großes Interesse daran herauszustellen, was ihre Software eventuell nicht kann. Im Folgenden finden Sie einige wichtige Fragestellungen für die Auswahl eines Analysewerkzeuges:

Fehlerreporting

- Meldet das Werkzeuge häufig „falschen Alarm“ („False Positives“)? Sprich werden Fehler gemeldet, die gar keine Fehler sind?
- Meldet das Werkzeug eventuelle Fehler gar nicht („False Negatives“)? Sprich werden tatsächlich vorhandene Probleme nicht erkannt?

Projektkompatibilität

- Benötigt das Werkzeug gemessen am Gesamtnutzen zu viel Zeit für die Analyse? Die Arbeitszeit eines Software-Werkzeuges ist normalerweise nicht so kritisch, aber in Extremfällen kann sie zu einem Problem für das Projekt werden.
- Unterstützt das Werkzeug den für das Projekt verwendeten Dialekt der Programmiersprache? Die meisten Compiler definieren quasi ihre eigene Variante der Sprache, in der der Code geschrieben wurde. Somit ist es unabdingbar, dass das Analysewerkzeug mit dieser Sprachvariante zurechtkommt.
- Wie problemlos kann das Werkzeug in den Entwicklungsprozess integriert werden? Jedes Werkzeug nützt wenig, wenn der Integrationsaufwand in das Projekt überproportional groß ist.

Fähigkeiten und Grenzen

- Kann das Werkzeug für das gesamte System verwendet werden? Das ist eine sehr wichtige Frage, da manche Fehler nur durch die Analyse des Gesamtsystems erkannt werden können.
- Verträgt das Werkzeug Rekursion zwischen Funktionen? Auch bei nur einer Datei ist Rekursion zwischen Funktionen von Bedeutung, wenn eine Funktion nur vollständig analysiert werden kann, nachdem eine andere Funktion analysiert wurde.
- Wo liegen die Grenzen des Werkzeuges? Jedes Werkzeug hat seine Grenzen, sei es die Menge an Code, die untersucht werden kann, die Tiefe der analysierbaren Blöcke, die Anzahl verschachtelter Klammern, die Größe der Symboltabelle usw. Diese Grenzen und ihre Auswirkungen auf das Projekt sollten bekannt und verstanden sein.

Fazit

Angesichts der hochkomplexen Softwaresysteme in vielen Medizingeräten hängt der Erfolg dieser Geräte immer häufiger von der Fähigkeit des Herstellers ab, den geforderten Level an Verlässlichkeit für das System nachzuweisen. Institutionen wie FDA und MHRA geben ihre Zulassung stets für Gesamtsysteme, nicht deren Einzelteile. Je mehr Software im Gerät steckt, desto wichtiger ist deshalb der Nachweis für die Verlässlichkeit dieser Software (strukturierter Sicherheitsnachweis) für die Marktzulassung. Somit ist es für den Erfolg jedes Medizingeräteprojektes mit Software absolut notwendig, nicht nur den Entwurfs- und Entwicklungsmethoden höchste Aufmerksamkeit zu schenken, sondern auch der Auswahl an Validierungstechniken und den Werkzeugen, die diese Techniken unterstützen.

Anhang A: IEC 62304 und Entwicklungstätigkeiten

Die Tabelle in diesem Abschnitt bildet Paragraphen des Standards IEC 62304 auf Entwurfs-, Entwicklungs- und Validierungstätigkeiten für Software ab. Die Einhaltung von IEC 62304 garantiert zwar nicht, dass die Software die gestellten Anforderungen an die Verlässlichkeit erfüllt oder Marktzulassungen problemlos geschafft werden. Aber es wird sichergestellt, dass wohldefinierte Prozesse eingehalten werden, dass die Anforderungen auf allen Ebenen vollständig definiert sind und dass ein Sicherheitsnachweis für das gesamte Produkt erstellt werden kann.

Legende

"+" Diese Methode wird für diese Klasse empfohlen.
"✔" Testwerkzeuge werden für die Effizienz und Effektivität der Tests empfohlen.

5.2 Software Anforderungsanalyse		**Klasse**		
		A	B	C
5.2.1	Festlegung und Dokumentation der Anforderungen an die Software aus den SYSTEM-Anforderungen	+ ✔	+ ✔	+ ✔
5.2.2	Inhalt der Software-Anforderungen	+ ✔	+ ✔	+ ✔
5.2.3	Aufnahme von Maßnahmen zur Risikokontrolle in die Anforderungen an die Software		+ ✔	+ ✔
5.2.4	Neu-Evaluierung der Risikoanalyse für das Medizingerät	+ ✔	+ ✔	+ ✔
5.2.5	Aktualisierung der SYSTEM-Anforderungen	+ ✔	+ ✔	+ ✔
5.2.6	Verifikation der Software-Anforderungen	+ ✔	+ ✔	+ ✔

Tabelle A1: Funktionalität von Testwerkzeugen in Bezug auf IEC 62304 Abschnitt 5.2 „Software Anforderungsanalyse"

5.5 Software Unit Implementierung und Verifikation		**Klasse**		
		A	**B**	**C**
5.5.1	Implementierung einzelner Software-Einheiten (Units)	+ ✔	+ ✔	+ ✔
5.5.2	Aufbau eines SOFTWARE UNIT Verifikationsprozesses		+ ✔	+ ✔
5.5.3	SOFTWARE UNIT Akzeptranzkriterien		+ ✔	+ ✔
5.5.4	zusätzliche SOFTWARE UNIT Akzeptranzkriterien			+ ✔
5.5.5	SOFTWARE UNIT Verifikation		+ ✔	+ ✔

Tabelle A2: Funktionalität von Testwerkzeugen in Bezug auf Abschnitt 5.5 „Software Unit Implementierung und Verifikation"

5.5.4 Zusätzliche SOFTWARE UNIT Akzeptanzkriterien	Klasse C
a) Saubere Eventsequenz	+
b) Daten- und Kontrollfluss	+ ✔
c) Geplante Ressourcen-Allokation	+
d) Fehlerbehandlung (Fehler-Definition, Isolation und Wiederaufsetzen)	+
e) Initialisierung von Variablen	+ ✔
f) Selbstdiagnose	+
g) Speichermanagement und Speicherüberlauf	+ ✔
h) Randbedingungen	+ ✔

Tabelle A3: Funktionalität von Testwerkzeugen in Bezug auf IEC 62304 Abschnitt 5.5.4 „Zusätzliche SOFTWARE UNIT Akzeptanzkriterien“

5.7 Software Systemtests		Klasse		
		A	B	C
5.7.1	Erstellen von Tests für Softwareanforderungen		+ ✔	+ ✔
5.7.2	Einsatz eines Problemlösungsprozesses		+ ✔	+ ✔
5.7.3	Neuer Test nach Änderungen		+ ✔	+ ✔
5.7.4	Verifikation der Systemsoftware-Tests		+ ✔	+ ✔
5.7.5	Protokollierung der Systemsoftware-Tests		+ ✔	+ ✔

Tabelle A4: Funktionalität von Testwerkzeugen in Bezug auf IEC 62304 Abschnitt 5.7 „Software Systemtests“

Anhang B: Das Embedded-Betriebssystem

Egal wie gut die Validierungswerkzeuge auch sein mögen, letztlich ist es das Gerät mit der Software, das die Zulassung bekommen muss. In jedem durch Software gesteuerten Gerät hängt alles oberhalb der Hardware vom Betriebssystem ab. Somit kann jedes Gerät mit einer Softwarekomponente nur so verlässlich sein wie das Betriebssystem. Deshalb sollte das Betriebssystem bestimmte Eigenschaften mitbringen, welche die Aussagen zur Sicherheit des Gerätes unterstützen.

Eine ausführliche Diskussion über die Anforderungen an Betriebssysteme in sicheren Systemen würde mehrere Bücherregale füllen. Trotzdem möchten wir kurz einige der wichtigsten Anforderungen für die Auswahl eines Betriebssystems für ein sicheres System aufführen:

Echtzeitfähigkeit

Nur ein Echtzeitbetriebssystem ist dafür konzipiert, zeitlich deterministische Antwortzeiten sicherzustellen, die für jedes sichere Softwaresystem unabdingbar sind.

Architektur

Ein schwerer Fehler in einem monolithischen Betriebssystem zieht für gewöhnlich einen Neustart nach sich und kompromittiert somit die Verfügbarkeit des Systems. Bei einer Microkernel-Architektur liegen Anwendungen, Gerätetreiber, Dateisystem und die Netzwerkkomponenten außerhalb des Kernels in separaten Adressräumen. Ein Absturz einer Komponente führt somit nicht zum Versagen des Gesamtsystems.

Speicherschutz

Die Betriebssystemarchitektur sollte Anwendungen und kritischen Prozessen jeweils eigene Speicherbereiche zuteilen, damit sich Fehler nicht im System ausbreiten können.

Prioritätsvererbung

Als Schutz gegen Prioritätsinversion sollte das Betriebssystem die Möglichkeit bieten, einem blockierenden niedrig priorisierten Thread die Priorität eines höher priorisierten blockierten Threads zuzuweisen, bis die Blockierung aufgehoben ist.

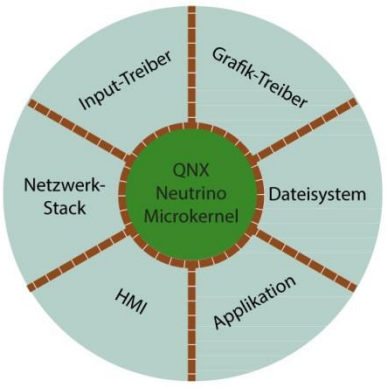

Abbildung 5. *In einem Microkernel-Betriebssystem laufen Systemdienste als normale Prozesse im User-Space. Ein Fehler in einem Speicherbereich ist in diesem isoliert. Der Kernel und die anderen Speicherbereiche sind geschützt.*

Zeitpartitionierung

Um hohe Verfügbarkeit zu garantieren, sollte das Betriebssystem eine feste oder vorzugsweise eine adaptive Zeitpartitionierung unterstützen. Damit lassen sich CPU-Zeitbudgets zuweisen. Bei einem adaptiven Ansatz können CPU-Zyklen von Partitionen, die sie aktuell nicht benötigen, anderen Partitionen zugewiesen werden, die ein Extra an Rechenleistung gebrauchen können.

Hohe Verfügbarkeit

Ein Software-Watchdog sollte Prozesse überwachen, stoppen und – sofern die Sicherheit garantiert werden kann – einzelne Prozesse neu starten können und somit einen Neustart des Gesamtsystems vermeiden. Wenn der Neustart eines Prozesses keine sichere Alternative ist, sollte das System in seinen Design Safe State (als sicher definierter Zustand) versetzt werden.

Über QNX Software Systems

QNX Software Systems Limited, eine Tochtergesellschaft von BlackBerry, ist ein führender Anbieter von Betriebssystemen, Entwicklungswerkzeugen und Dienstleistungen für vernetzte Embedded-Systeme. Weltmarktführer wie Audi, Cisco, General Electric, Lockheed Martin und Siemens nutzen Technologie von QNX zum Beispiel in Fahrzeug-Infotainment-Einheiten, Netzwerkroutern, medizintechnischen Geräten, industriellen Automatisierungsanlagen sowie Sicherheitssystemen und anderen missions- oder betriebskritischen Anwendungen. QNX Software Systems Limited wurde 1980 gegründet. Der Hauptsitz des Unternehmens befindet sich in Ottawa, Kanada; die deutsche Niederlassung befindet sich in Hannover. QNX-Produkte werden weltweit in über 100 Ländern vertrieben. Besuchen Sie www.qnx.de und https://www.facebook.com/QNXSoftwareSystems, und folgen Sie @QNX_News auf Twitter. Weitere Informationen zu unserer Tätigkeit im Automobilbereich finden Sie unter qnxauto.blogspot.com oder folgen Sie @QNX_Auto.

www.qnx.de