Evaluierung von Methoden zur Entwicklung sicherheitskritischer Softwaresysteme

Informatikprojekt
bei
Prof. Dr. Voos

von
Tobias Hondorf

29. April 2010

Studiengang Angewandte Informatik
Fakultät für Elektrotechnik und Informatik

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung - Was ist die Besonderheit an sicherheitskritischen Anwendungen?

Um die Fragestellung, was die Besonderheit an Sicherheitskritischen Systemen ist, zu klären, muss zunächst definiert werden, was ein sicherheitskritisches System ist. Der Begriff "sicherheitskritisches System" unterliegt keiner allgemeingültigen Definition. Das deutsche Wort "Sicherheit" hat zwei Bedeutungen, zum einen der Schutz der Umwelt und des Menschen vor einem Objekt, die sog. Betriebssicherheit (engl. Safety) und zum anderen der Schutz eines Objektes vor der Umwelt bzw. des Menschen hier spricht man von Angriffssicherheit (engl. Securtiy). In dieser Arbeit wird der Begriff Sicherheit im Sinne von Betriebssicherheit also Safety verwendet.

Safety vs. Security

Was ist nun also ein sicherheitskritisches System? Allgemein kann gesagt werden, dass ein System, von dem eine unmittelbare oder mittelbare Gefahr für den Mensch oder die Umwelt ausgeht, als sicherheitskritisch eingestuft wird. Die Sicherheitskritikalität eines Systems stellt besondere Anforderungen an seine Entwicklung. Und genau hierin liegt die Besonderheit sicherheitskritischer Anwendungen - die Anforderung nach Sicherheit. Ein sicherheitskritisches System muss nachweislich einen gewissen Grad an Robustheit und Sicherheit gewährleisten.

Diese Arbeit betrachtet Sicherheitskritische Anwendungen im Sinne von Software. Anders als Hardware kann Software nicht Ausfallen, sie ist frei von physikalischen Einflüssen. Ein Softwaresystem kann aber, auf Grund von Fehlern in der Software selbst, zu Fehlfunktionen führen. Durch einen Softwarefehler könnte das System mit falschen Werten oder Steueranweisungen versorgt werden. Im extremsten Fall stürzt die Software ab, was sich durch eine permanente Unterbrechung des Programmablaufes äußert. Eine solche Unterbrechung kann Auswirkung auf das gesamte System haben. Kann das System durch einen Softwarefehler in einen gefährlichen Zustand versetzt werden, so ist die Software als sicherheitskritisch zu sehen. Und hat die Anforderung nach Sicherheit, aus der sich die Anforderung nach Fehlerfreiheit ableitet.

Sicherheitskritische Software

Diese Anforderung soll an einem kleinen Beispiel verdeutlicht werden. Im Alltag kommt jeder unweigerlich mit vielen elektronischen Systemen in Kontakt in denen Software arbeitet. Ein einfaches Beispiel ist ein Fernsehgerät. An die Software auf einem solchen Gerät wird keine sicherheitskritische Anforderung gestellt. Einer der schwersten Fehler die auftreten können wäre der Totalausfall im operationellen Betrieb. Selbstverständlich soll dieser Fehler auch ohne sicherheitskritische Anforderungen nicht auftreten, denn wer kauft sich einen Fernseher der sich ständig abschaltet? Würde ein solcher Fehler jedoch in 10.000 Betriebsstunden einmal Auftreten, so wäre

Fehlertoleranz

diese Zahl tolerierbar. In jedem Fall würde von einem solchen Softwarefehler keine Gefahr für den Menschen ausgehen.
Ein Gegenbeispiel wäre, der Totalausfall des "Fly-by-wire" Systems eines modernen Passagierflugzeuges im operationellen Betrieb. Ein solcher Ausfall würde unweigerlich zum Tod vieler Menschen führen. Das Auftreten dieses Fehlers wird als nicht tolerierbar angesehen und muss deshalb ausgeschlossen werden.

Aus diesem Grund benötigen Systeme, die eine Gefahr für Mensch und Umwelt darstellen können, in der Regel eine Zulassung durch eine (meist staatliche) Behörde. So brauchen ein Flugzeug, ein Auto oder auch z.B. Industrieroboter oder medizinische Geräte eine Zulassung um eingesetzt werden zu dürfen. Hierfür muss der Behörde ein Nachweis darüber erbracht werden, dass das jeweilige System ein gewisses Maß an Sicherheit erfüllt. Und die somit von dem System ausgehende Gefahr minimiert ist.

Zertifizierung sicherheitskritischer Systeme

Um diese Nachweise einheitlich Erbringen zu können, wurden Standards festgelegt, deren Einhaltung darauf abzielt ein System sicher zu machen und dies einer Zulassungsbehörde gegenüber Nachweisen zu können. Ein allgemeingültiger Standard für die Softwareentwicklung in einem sicherheitskritischen System ist die Norm IEC 61508. Die Einhaltung dieser Norm zielt auf eine allgemeine Zertifizierung eines Systems unter sicherheitskritischen Anforderungen ab. Neben der allgemeingültigen IEC 61508 gibt es noch viele Branchenspezifische Normen. Diese sind dann an die Problemstellungen und Eigenheiten der jeweiligen Bereiche angepasst. So stellt die Entwicklung eines Flugzeuges andere Anforderungen an die Sicherheit, als z.B. ein chirurgisches Laserskalpell. Aus diesem Grund gibt es Standards für die Entwicklung in der Luftfahrt, als auch Standards zur Entwicklung von medizinischen Geräten.
Welcher Standard bei einer Entwicklung aber in letzter Konsequenz eingesetzt wird, hängt von den Forderungen der Zertifzierungsbehörde und/oder dem Kunde ab.

Zusammenfassung Die Besonderheit an sicherheitskritischer Software ist die Anforderung nach Sicherheit. Diese muss durch die Einhaltung von Standards bzw. Normen nachweisbar sein.

2 Übersicht sicherheitskritischer Entwicklungsstandards in der Embedded-Software

2.1 Allgemein

IEC 61508

Der allgemeinste Standard für die Zertifizierung sicherheitskritischer Systeme ist die Norm IEC/ISO 61508 "Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer elektronischer Systeme" [CEN01].
Dieser Standard legt sich nicht auf ein Einsatzgebiet fest, sondern behandelt die Sicherheitskritikalität im Allgemeinen. Er lässt sich somit auf die unterschiedlichsten Einsatzgebiete zuschneiden und ist stark anpassungsfähig und kann auch zur Entwicklung eines unkritischen Systems verwendet werden.

Die IEC 61508 besteht aus folgenden 7 Teilen:

- IEC 61508-1 Allgemeine Anforderungen
- IEC 61508-2 Hardware
- IEC 61508-3 Software
- IEC 61508-4 Definitionen
- IEC 61508-5 Richtlinien zu Teil 1, Beispiele zur Einstufung der Sicherheitsintegrität
- IEC 61508-6 Richtlinien zu Teil 2 und 3
- IEC 61508-7 Techniken, Beispiele

Wobei die Teile 1-4 normativ sind, also die dort beschriebenen Anforderungen als Prüf- und Zertifizierungskriterien verwendet werden können. Die Teile 5-7 haben rein informativen Inhalt. Sie enthalten Beispiele, Ansätze und Strukturen, die als Ergänzung zu den ersten 4 Teilen zu sehen sind.

Zertifizierung erfolgt immer für ein gesamtes System

Wie anhand der Teile der Norm zu sehen ist, beschäftigt sich diese nicht nur mit der Softwareentwicklung, sondern mit dem Gesamtsystem. Es gilt zu verstehen, dass es keine alleinstehende Softwarezertifizierung gibt. Eine Zertifizierung erfolgt immer für das Gesamtsystem. Dies liegt in der Tatsache begründet, dass ein System nur sicher sein kann, wenn "sichere" Software auf "sicherer" Hardware läuft. Die Software ist also ein Teil des Gesamtsystems, deshalb müssen für diese Nachweise erbracht werden. In der Praxis besteht zwar die Möglichkeit der Einzelzertifizierung, also der Zertifizierung eines Softwaresystems oder einer Hardwareplattform, jedoch

muss bei der Zertifizierung eines Produkts dennoch das Gesamtsystem geprüft werden. Teilprüfungszertifikate werden aber unterstützend anerkannt. Ein Beispiel hierfür sind COTS[1] Produkte die es im Hardware- sowie Softwaresektor gibt. Für diese existiert dann ggf. eine Vorabzertifizierung, die bei der Software in Form von Zertifizierungsartefakten vorliegt. Diese können dann zur Zertifizierung des Endproduktes, zusammen mit den selbst erstellten Zertifzierungsunterlagen, für das Projekt beim Zertifizierer eingereicht werden.

Allgemeine Anforderungen Der erste Teil der IEC 61508 Norm enthält die allgemeinen Anforderungen. Er beschreibt die Betrachtung des Gesamtlebenszyklus eines Projektes. Diese Betrachtung geht über die reine Softwareentwicklung hinaus und schließt das Projekt, Konfigurations- und Dokumentationsmanagement mit ein. In den Anforderungen sind Projektphasen definiert, bei deren Übergang die Norm eine Verifikation der Ergebnisse fordert. Es muss also ein Nachweis darüber erbracht werden, dass eine Phase korrekt und den Anforderungen entsprechend abgeschlossen wurde, bevor zur nächst übergegangen werden darf. Über den gesamten Projektverlauf muss die Dokumentation so erfolgen, dass jeder Vorgang durch dritte Vollständig rekonstruiert werden kann. Das heißt, dass eine detailierte Planung durchgeführt werden muss, die das Vorgehen und die geforderten Ergebnisse definiert. Durch die Dokumentation muss nachweisbar sein, dass das vordefinierte Vorgehen angewandt und die geforderten Ergebnisse erzielt wurden.

Der Projektablauf muss zu jedem Zeitpunkt nachweisbar sein

Sicherheitskritikalität Die IEC 61508 fordert, dass nach der Definition des gesamten Systems, eine Gefahren- und Risikoanalyse durchgeführt wird. Hierbei soll ermittelt werden, was es für potentielle Gefahren gibt und wie sich diese auf das System auswirken können. Für die Gefahren müssen entsprechende Gegenmaßnahmen ausgearbeitet werden. Hierzu kann Teil 5 der Norm unterstützend herangezogen werden. Das Ergebnis der Analyse sind Sicherheitsanforderungen an das Gesamtsystem, anhand derer sich das System in eines der vier Sicherheitsintegritätslevel (SIL) einordnen lässt. Je nach zunehmender Kritikalität werden die Anforderungen an ein SIL strenger (SIL 1 = gering, SIL 4 = hoch). Das Ergebnis der Risiko- und Gefahrenanalyse und die Einstufung in SIL, fließen direkt in die Software-Sicherheitsanforderungen (Beschrieben in IEC 61508 Teil 3) ein.

SIL: Sicherheitsintegritätslevel

Softwareentwicklung Je nach SIL einer Software müssen die Entwickler in der Norm zitierte Strategien verwenden. Über die mögliche Kombination der Strategien wird jedoch keine Aussage getroffen. Beispielsweise werden keine Anmerkung zu Codebeschränkungen gemacht. Oftmals wird in der aus der Automobilindustrie stammende MISRA Coding-Standard verwendet, dieser wird aber nicht explizit vorgeschrieben. Es ist ratsam Teil 7

[1] Commercial of-the-shelf

SIL	Fehlerhäufigkeit
4	10^{-5} - 10^{-4}
3	10^{-4} - 10^{-3}
2	10^{-3} - 10^{-2}
1	10^{-2} - 10^{-1}

Tabelle 1: IEC 61508 - Sicherheitsintegriätslevel
(Abgeleitet aus [CEN01])

des Standard als Unterstützung für die in Teil 3 beschriebenen Software-Anforderungen zu verwenden. Für die Softwareentwicklung wird auch das Softwarelebenszyklusmodell verfolgt. Generell ist für die Norm ein modellbasiertes Vorgehen verpflichtend, allerdings wird hier nur ein Vorschlag zum V-Modell gemacht. Die Verwendung dieses ist aber nicht zwingend. Dies rührt daher, dass viele Firmen ihre eigenen Vorgehensmodelle für Projekte entwickelt haben (diese sind oftmals stark an das V-Modell angelehnt.). Es soll hier Freiraum für diese individuellen Vorgehenslösungen gelassen werden.

Verwendung eines Vorgehensmodells

Zusammenfassung Die Norm beschreibt, dass der Geltungsbereich des Systems definiert werden muss. Es müssen Analysen durchgeführt werden, die Gefahren und Risiken genau festlegen. Die Softwareentwicklung muss einem in der Planung festgelegten Modell folgen, dass zwar frei wählbar ist, aber beim Übergang von einer in die nächste Phase Tests und Reviews verlangt. So soll sichergestellt werden, dass die Anforderungen einer Phase erfüllt wurden, bevor mit der nächsten weiter gemacht wird. Grundsätzlich kann diese Norm auch zur Erstellung von nicht sicherheitskritischen Systemen eingesetzt werden. Die Norm sieht es vor, das bereits definiertes Vorgehen von Vorgänger- bzw. ähnlichen Projekten, in ein neues Projekt übernommen werden kann. Somit soll versucht werden, den Aufwand für weitere Projekte möglich gering zu halten.

IEC 61508 auch für die Entwicklung von nicht-sicherheits-kritischen Systemen anwendbar

2.2 Luftfahrt, ist und Zukunft

Der Flugzeugbau wird im laufe seiner Entstehung und bis heute, als eine der größten ingeneurswissenschaftlichen Herausforderungen betrachtet. Waren die ersten Flugzeuge doch rein mechanisch und zählten zur Kategorie "Maschinenbau", so wäre heute die Elektronik nicht mehr wegzudenken. Große Passagierflugzeug sind zu fliegenden High-Tech Computern geworden. Die Steuerung des Flugzeugs wird schon längst nicht mehr mechanisch, sonder elektronisch umgesetzt. Zahlreiche elektronische Geräte unterstützen die Piloten bei Steuerung, Navigation und Kommunikation - Geräte in denen

Software arbeitet.

Am Beispiel der Flugsteuerung (Fly-by-Wire) ist gut zu sehen, wie wichtig fehlerfreie und funktionierende Systeme sind. Ein Ausfall des Systems hätte katastrophale Folgen, daher gilt es einen solchen Fehler zu vermeiden.

DO-178B

Der DO-178B "Software considerations in airborne systems and equiptment certification" ist ein international anerkannter Standard zur Zertifizierung sicherheitskritischer Software im Bereich des Flugzeugbaus. Der U.S. Amerikanische Standard wird in Europa unter dem Namen ED-12B verwendet.

Der DO-178B umfasst, anders als der IEC 61508, nur den Software-Lebenszyklus. Es sind zwar Schnittstellen zum System-Lebenszyklus definiert, dieser wird aber nicht speziell betrachtet.

Umfasst den Software-Lebenszyklus

Der DO-178B definiert Richtlinien, die den Einklang zwischen den Anforderungen der Luftfahrt und der Software sicherstellen sollen. Diese Richtlinien beschreiben Ziele für Software-Lebenszyklusprozesse, Aktivitäten um diese zu erreichen und Nachweise um die Einhaltung der Ziele zu überprüfen.

- Beschreiben der Ziele (objectives)
- Beschreiben der (Prozess-)Aktivitäten die zur Erfüllung der Ziele führen
- Beschreiben der Nachweise, die die Einhaltung der Ziele bestätigen

Der Software-Lebenszyklus wird im Standard als die Gesamtheit aller Prozesse sowie deren Abfolge und gegenseitige Beeinflussung verstanden. Er wird in folgende 6 Prozesse untergliedert:

- Planung
- Entwicklung
- Verifikation
- Konfigurationsmanagement
- Qualitätssicherung
- Schnittstelle zur Zertifizierungsbehörde

Wichtig ist, dass der Standard kein Prozessmodell im besonderen Vorschlägt, sondern nur einen Rahmen festlegt. Man hat die Freiheit ein Prozessmodell zu verwenden, das die Rahmenkriterien erfüllt. Das V-Modell wäre ein solches Prozessmodell, das den Rahmenbedinungen des DO-178B

gerecht wird. Oftmals ist es aber so, dass Firmen selbst Prozessmodelle definieren oder bereits vorhandene modifizieren um ihre über Jahre etablierten und durch Erfahrung optimierten Prozesse verwenden zu können. Er legt aber dennoch ein gewisses Vorgehen fest, so muss das Software-Kritikalitätslevel (siehe Tabelle 2) festgelegt werden. Aus diesem leiten sich dann spezielle Anforderungen ab. So muss eine Software, die eine höhere Kritikalität hat, ausführlicher und umfangreicher getestet werden, als eine weniger kritische.

Design Assurance Levels (DAL)	Failure Condition
Level A	**Catastrophic**
Level B	**Hazardous/Severe-Major**
Level C	**Major**
Level D	**Minor**
Level E	**No Effect**

Tabelle 2: DO-178B Kritikalitätsstufen
(gekürzt aus [Inc92] S.7)

Planungsprozess Im Planungsprozess soll der Softwareentwicklungsprozess definiert werden. Die Abhängigkeit und Abfolge von Prozessen soll festgelegt werden. Die Umgebung des Software-Lebenszyklus soll beschrieben werden (welche Methoden und Tools werden eingesetzt?) und es soll festgelegt werden, was für Entwicklungsstandards zur Anwendung kommen. Dabei muss stehts darauf geachtet werden, dass die Planung den Anforderungen des DO-178B genügt, denn dies ist für die erfolgreiche Zertifizierung essentiell! Im Detail sollen die folgenden Pläne ausgearbeitet werden:

- Plan of Software Aspects of Certification
- Software Development Plan
- Software Verification Plan
- Software Configuration Management Plan
- Software Quality Assurance Plan

Neben diesen Dokumenten fordert der Standard noch weitere Dokumente, deren Umfang je nach Kritikalitätsstufe der Software variiert. Der Standard stellt also formale Anforderungen an den Gesamten Projektablauf.

Umfang der Dokumente ist abhänig von der Kritikatlität

Entwicklungsprozess Es wird kein Enwicklungsprozess vorgeschrieben, sondern nur Ziele von Teilprozessen beschrieben. In welcher Reihenfolge diese ablaufen oder ob sie teilweise parallel ausgeführt werden, kann selbst definiert werden. Die Ergebnisse der Teilprozesse sollen stehts nachvollziehbar sein, d.h. die Anforderungen an die höheren Ebenen müssen sich in den Teilprozessen wiederfinden. Es besteht die Möglichkeit der Revision eines vorangegangenen Entwicklungsschrittes, falls Inkonsistenzen oder Fehler auftreten.

Verifikation Durch die Verifikation sollen die Ergebnisse der Softwareentwicklung, auf Fehlerfreiheit und Richtigkeit, überprüft werden. Der Verifikationsprozess soll Fehler aufspüren und diese an den Softwareentwicklungsprozess zurück melden, damit diese dann behoben werden können. Dies geschieht durch Reviews und Analysen. Zu den Zielen der Verifikation gehört es zu prüfen, ob die High-Level-Anforderungen in Architektur und Low-Level Requirements umgesetzt wurden, ob die Low-Level Anforderungen in Sourcecode umgesetzt wurden, ob der Ausführbare Code die Softwareanforderungen erfüllt und ob die im Standard für den jeweiligen DAL Level vorgesehen Mittel korrekt und vollständig eingesetzt wurden.

Konfigurationsmanagement Die Aufgabe des Konfigurationsmanagements ist von organisatorischer Natur. Es legt Verantwortlichkeiten fest, sorgt für Nachvollziehbarkeit und betreibt Auswirkungskontrolle. Somit sind konkrete Ziele des Konfigurationsmanagementprozesses, beispielsweise die Identifikation der Konfigurationseiniten (Configuration Items), die Sicherstellung, dass Schritte und insbesondere Änderungen durch Revisionen nachvollziehbar sind, sowie die Verfügbarkeit der Daten, zu garantieren.

Ergebnisse des Standards Der Standard stellt Anforderungen an die Eindeutigkeit, Vollständigkeit, Konsistenz, Klarheit, Verifizierbarkeit, Nachvollziehbarkeit und Änderbarkeit des Software-Lebenszyklus. Das Ergebnis setzt sich aus Plänen, Standards, Anforderungen, Design, Code, Verifikationsergebnissen, Berichten und Qualitätssicherungsdokumenten zusammen. Das Zentrale Dokument ist hierbei das "Plan of Software Aspects of Certification".

Zusammenfassung Der DO-178B ist im Vergleich zu anderen Standards "gut lesbar", weil er Sachverhalte konkretisiert. Nach dem Abschluss der Planungsphase sind die zum Einsatz kommenden Tools, Methoden sowie der Zeitplan festgelegt. Für die Verifikation und die Tests sind modellbasierte Ansätze vorgeschlagen. Die Anforderungen an Software, der höchsten Kritikalitätsstufe (DAL-A), sind im Vergleich zu denen anderen Sicherheitskritischen Standards sehr hoch.

Vergleich DO-178B und IEC 61508 Der IEC 61508 ist eine europäische Norm und findet kaum außerhalb Europas Anwendung. Die DO-178B hat bezüglich internationaler Zulassungsrichtlinen (von allen Luftfahrtbehörden anerkannt) einen Referenzsstatus. Vieles aus DO-178B findet sich auch in anderen Standards zur sicherheitskritischen Softwarenentwicklung wieder.
Der DO-178B schreibt zu erstellende Dokumente konkret vor.
Im IEC61508 wird zwischen funktionalen Anforderungen und Integrationsanforderungen unterschieden. Beim DO-178B wird nach Low-Level und High-Level Anforderungen unterschieden.

Viele Ansätze des DO-178B finden sich im IEC61508 wieder

2.3 andere Branchenspezifische Standards

In Abschnitt 2.1 wurde bereits darauf verwiesen, dass es für die unterschiedlichen Branchen jeweils spezielle Zertifizierungsstandards gibt.

Luftfahrt DO-178B / ED-12B

Die DO-178B wurde bereits im Abschnitt 2.2 behandelt.

Automobil OSEK/VDX, AUTOSAR

Das OSEK/VDX Gremium hat diverse Standards für Software im Automobil geschaffen. So z.B. die Spezifikation OSEK-OS, welche ein Echtzeitbetriebssystem zum Einsatz in Embedded-Systemen beschreibt. Die wesentlichen Teile der OSEK Spezifikationen sind in ISO 17356 zusammengefasst.

AUTOSAR ist ein internationaler Verbund mit dem Ziel, einen offenen Standard für Software-Architekturen in Kraftfahrzeugen zu etablieren.

Verkehrstechnik (DIN) EN 50128

Die EN 50128 ist eine europäische Norm für sicherheitsrelevante Software des Schienenverkehrs.

Medizintechnik FDA 510(k)

U.S.-Amerikanischer Standard zur Zertifizierung medizinischer Geräte.

Atomkraft (DIN) IEC 880

Standard für Software für Rechner im Sicherheitssystem von Kernkraftwerken. Es werden Anforderungen für jede Stufe der Software-Entwicklung

vom Entwurf über Entwicklung, Qualitätsprüfung, Betrieb und die Dokumentation festgelegt.

Militär MIL-STD-498

U.S.-Amerikanischer Standard zur Zertifizierung militärischer Geräte.

3 Vorgehen in der Luftfahrt allgemein

Das Vorgehen in der Luftfahrt orientiert sich an den vom Kunden vorgegebenen Anforderungen. In der Regel wird ein Avionik-Projekt nach DO178B zertifiziert, dies ist ebenso eine Vorgabe des Kunden. Infolgedessen werden die Vorgaben und Richtlinien des DO178B eingehalten. Das Vorgehen orientiert sich somit an dem Standard. Oftmals legt der Kunde sein gewünschtes Prozessmodell vor, das dann zur Entwicklung verwendet werden muss. Es spielt auch eine Rolle, ob es sich um eine Neu- oder Weiterentwicklung eines Produktes handelt. Aus diesen Gründen gibt es kein generell angewandtes Vorgehen.

Rapid Prototyping Um ein Beispiel für ein mögliches Vorgehen bei einer Neuentwicklung zu geben, möchte ich das "Rapid Prototyping" [AbRS94] anführen. Diese Methodik zielt auf das schnelle erzeugen von funktionalen Modellen (Prototypen) in einem frühen Stadium des Projekts ab. Da diese Methode informal ist, lassen sich Prototypen schnell erzeugen, ändern oder verwerfen. Es muss dabei nicht auf die Struktur, Wartbarkeit und Wiederverendbarkeit der Software geachtet werden. Ein Ansatz ist, durch einen solchen Prototypen, die Kundenanforderungen zu realisieren und zu testen. Bereits beim Prototyping können Probleme identifiziert werden, und die Machbarkeit der Erfüllung der Anforderungen überprüft werden. Dies führt dazu, dass für Probleme Lösungen erarbeitet und getestet werden können, ohne dass ein formaler Weg eingehalten werden muss.

Schnelle Verfügbarkeit eines Prototypen-Modells

Ein Vorteil zeigt sich in der hohen Flexibilität und den geringen Kosten dieser Methode. Zudem kann dem Kunden bereits in einem frühen Stadium des Projektes ein funktionierendes Modell gezeigt werden, und durch Kunden-Feedbacks können Anforderungen weiter spezifiziert und konkretisiert werden. Somit können funktionale Prototypen erstellt werden, die schon vor beginn der eigentlichen Entwicklung, die Komplexität und die Struktur des Systems bzw. der Software erkennen lassen.

Frühe Erkennung von Problemen

Nachdem Abschluss des Prototypings wird die formale Entwicklung eingeleitet. Die erstellten Modelle werden analysiert und helfen bei der Konkretisierung der Anforderungen. Sie dienen aber nicht als Grundlage zur Codeerzeugung, sondern werden nach der Analyse weggeworfen. Da Problemstellungen bereits durch die Modelle bekannt sind, kann die Planung

optimiert werden und Zeit zum Erarbeiten von Lösungsstrategien eingeplant werden. Zudem kann die Planung direkt auf das Ziel gerichtet werden, da die "Richtung", in welche die Entwicklung geht, durch die Prototypen klar wurde. Ein Vorteil hierbei ist, dass die Entwicklung jetzt weitestgehend Straight-Through erfolgen kann, denn es wird zwar noch nicht identifizierte Probleme geben, jedoch wird diese Zahl geringer sein als ohne Prototyping. Somit wird kaum noch Zeit für formale Änderungen am Projekt benötigt. Ein realitätsnahes Beispiel wäre die Verwendung eines durch Dritte gefertigten Hardwaretreibers, der bei der Erstellung der Prototypen bereits vorlag. Der Treiber ist so implementiert, dass bestimmte, für das Projekt wichtige Funktionalitäten, nicht ermöglicht werden. Dies wäre beim Prototyping aufgefallen und man hätte ein Vorgehen definieren können. Wird der Treiber selbst geändert, neugeschrieben, oder vom Hersteller um die benötigte Funktionalität erweitert? Beim Prototyping kann jede der Lösungen verwendet werden, ohne Änderungen an der Projektplanung machen zu müssen. Wird aber dieses Problem bei einer formalen Entwicklung festgestellt, so muss es eine Änderung der Planung geben. Es muss begründet werden warum der Treiber nicht verwendet werden kann, wie die genaue Lösung aussieht und auf was für Komponenten der Software diese Änderung im Projekt Auswirkungen haben kann. Diese vermeintlich kleine Änderung, die durch den Austausch des Treibers vorgenommen wird, zieht einen enormen Rattenschwanz an Formalien nach sich. Aus wirtschaftlicher Sicht ist dies ein nicht unbedeutender Kostenfaktor.

Gezielte Ausrichtung der Entwicklung auf die Lösung der Probleme

Das genannte Vorgehen ist in Abbildung 1 dargestellt. Es soll deutlich werden, dass an Stelle der formalen Entwicklung zunächst das Rapid Prototyping steht, dessen Ergebnisse dann zur formalen Planung des Projekts beiträgt.

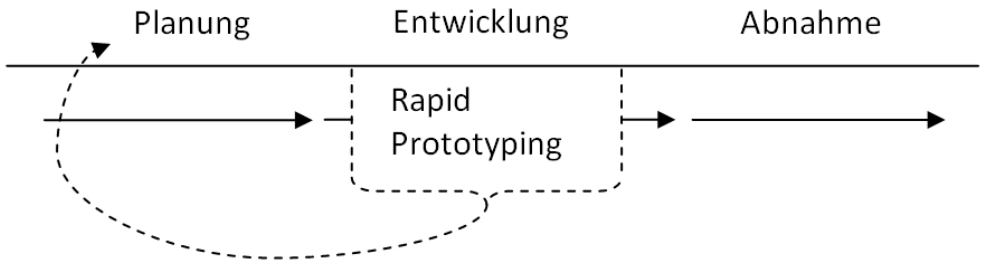

Abbildung 1: Rapid Prototyping im Projektfluss

Bezugnehmend auf den DO-178B Standard kann diese Methodik verwendet werden. Diese macht keine Vorgaben zur Verwendung bestimmter Methoden. Selbstverständlich müssen bestimmte Formalien eingehalten werden um die Konformität zu wahren, jedoch wird Rapid Prototyping vor der formalen Planung durchgeführt. Da DO-178B auch keine Vorgehensmodelle vorschreibt, ist eine Modellbasierter Ansatz in keinem Fall verbindlich.

Rapidprototyping ist mit DO-178B verwendbar

Jedoch sind diese weitverbreitet und haben sich in der Praxis bewährt. Neben dem Rapid Prototyping, das auf die Erstellung funktionaler Modelle abzielt, gibt es noch weitere Modellierungsmethoden. Bekannte Beispiele zur abstrakten Modellierung sind "UML" oder "Methode-B".

Unabhängig von einer verwendeten Modellierung, gliedert sich ein Entwicklungsprozess oftmals formal in "Analyse", "Entwurf", "Implementierung" und "Test" auf (vgl. [FK04]).

3.1 Entwicklungsmethoden

3.1.1 Analyse

Die Analyse stellt die Spezifikation funktionaler Softwareanforderungen in den Vordergrund. Hierzu gibt es formale Spezifikationssprachen wie LOTOS [BB87], SCR [BH99], VDM [Jon90], Z [Spi92]. Diese unterstützen die Modellierung durch mathematische Methoden. Es gibt allerdings keine verbreitete Methode zur Erstellung einer Implementierung aus einem solchen Modell. Fraglich wäre ohnehin der Nutzen, denn der Kunde fordert ein Produkt und kein Modell. Somit rät die mehrheitliche Meinung von einer Verwendung eines Modells als Implementierungsgrundlage ab.

3.1.2 Entwurf

Oftmals werden auch für den Entwurf formale Methoden benutzt. So ist durch UML ein objektorientierter Entwurf möglich. Ein anderer Ansatz wäre das bereits genannte (Rapid) Prototyping oder aber auch eine Modellierung durch die Spezifikationssprache Z.

3.1.3 Implementierung

Die reine Implementierung wird i.d.R. manuell und ohne Verwendung spezieller Methoden durchgeführt. Allerdings ist oftmals bei der Codeerstellung ein Coding Standard einzuhalten - dieser kann als eine Methode interpretiert werden, denn er legt einen formalen Rahmen für die Entwicklung fest. So kann ein Coding Standard dafür sorgen, dass sich Softwareanforderungen in den einzelnen Funktionen wiederfinden lassen und dies auch Dokumentiert wird. Hierdurch ist eine Verknüpfung der Verifikation mit der Implementierung möglich.

3.1.4 Test

Tests werden in Verbindung mit Testwerkzeugen durchgeführt. Somit ist eine Automatisierung von Tests möglich. Es gibt auch Ansätze zur automatischen Generierung von Test-Cases, allerdings basieren diese dann auf einem zuvor erstellten Modell.
Generell kann bei der Art der Tests zwischen Whitebox-Tests (Code wird

untersucht) und Blackbox-Tests (Code ist unbekannt) unterschieden werden. Eine mögliche Testvarianten ist z.B. die in der DO-178B beschriebene "Code Coverage Analysis". Die hiervon aufwendige Form stellt der "modified condition / decision coverage" (MC/DC) Test dar (wird gefordert bei DAL-A Zertifizierung, [Inc92] Tabelle A-7). In DO-178B wird zwischen den Teststufen "Low-Level Test", "Software Integration Test" und "Hardware/-Software Integration Test" unterschieden. Diese Begriff sind ein Äquivalent zu den in der (nicht Avionik-) Literatur gebräuchlichen Begriffen: "Unit Testing", "Integeration Testing" und "System Integration Testing".

4 Beispiel: Coding Standard

Ziele eines Coding Stadards

Ein Coding Standard beschreibt Regeln zur Quellcode-Erstellung, welche der Programmierer einzuhalten hat. Der Umfang eines Coding Standards hängt von den Zielen ab, die durch ihn erreicht werden sollen. Aus diesem Grund gibt es auch keinen einheitlichen und für jedes Projekt verbindlichen Coding Standard. Dies ist schon allein deshalb nicht möglich, weil ein Coding Standard für eine Programmiersprache erstellt wird und nicht auf eine andere angewandt werden kann.

Mögliche Ziele eines Coding Standards können sein:

- Erhöhen der Lesbarkeit...
- Verbessern der Robustheit...
- Vereinfachung der Wiederverwendbarkeit...
- Senken des Abstraktionsgrads...
- Erzeugen von Modularität...
- Erhöhung der Wartbarkeit...
- Umsetzung von Programmierparadigmen[2]
- Vermeidung von Redundanzen...

... des/im Quellcode(s).

Realisierung der Ziele

Diese Ziele werden auf unterschiedlichen Wegen erreicht. Hierzu beschreibt ein Coding Standard ein Regelwerk. Die Einhaltung dieser Regeln wird vom Programmierer verlangt und kann durch bestimmte Softwarewerkzeuge überprüft werden. Ein Compilier, der zur Übersetzung des Codes verwendet wird, stellt in der Regel hierzu keine Mechanismen bereit.

[2]Programmierparadigmen sind z.B. Imperative-, Deklarative-, Objektorientierte-, Komponentenorientierte-, ...- und Generische-Programmierung.

Mögliche Ansätze für Regeln zur Realisierung der Ziele:

- Verringerung des Sprachumfangs
  (Welche Funktionalitäten dürfen / dürfen nicht benutzt werden)
- Formatierung des Quelltextes
  (Einrückung, Absätze/Umbrüche, Leerzeichen, ...)
- Namenskonventionen
  (homogener Aufbau von Funktionsnamen, Variablennamen, ...)
- Anwendung bestimmter Entwurfsmuster
  (Design-Patterns)
- Typisierung
  (Wahl des Typs eines Symbols)
- Einheitliche Kommentierung
  (Um beispielsweise eine automatische Dokumentierung[3] zu ermöglichen)

Programmierstil

Solche Regeln sind Programmierern auch als "Programmierstil" bekannt. Da jeder Programmierer einen eigenen Programmierstil entwickelt (vergleichbar mit dem Schriftbild eines Aufsatzes), fällt es dem Einzelnen oft schwer sich einem solchen Regelwerk zu unterwerfen - umfangreiche Coding Standards stoßen nicht selten zunächst auf Ablehnung beim Programmierer. Jedoch sind die Vorteile eines Coding Standards klar erkennbar; durch gute Strukturierung und Homogenität erhöht sich die Nachvollziehbarkeit. So kann sich ein Dritter schneller in einem, nicht von ihm geschriebenen, Quellcode zurechtfinden (Wartbarkeit). Dies erzeugt eine Unabhängigkeit vom einzelnen Programmierer und führt zwangsweise weg von der Denkweise (überzogen dargestellt): "Ich schreibe den Code so, dass nur ich ihn verstehe, um das Projekt von mir abhängig zu machen.".
Zudem kann beispielsweise durch die Verwendung genormter Kommentare und unter Zuhilfenahme diverser Softwarewerkzeuge die Dokumentation zu einem großen Teil automatisiert werden.

Projektgröße

Aus meiner Erfahrung macht die Erstellung gewisser Konventionen schon bei einem Projekt an dem mehr als eine Person arbeitet Sinn. Allein das Einhalten von Kommentar- und Namenskonventionen erleichtert die gemeinsame Programmierung ungemein.

4.1 Kriterien für die Auswahl eines Standards

Wie im vorangestellten Abschnitt erwähnt, ist die Auswahl eines Coding Standards von mehreren Kriterien abhängig. Das wichtigste Kriterium ist

die Programmiersprache in welcher Code geschrieben wird. Aber auch Kundenanforderungen, Forderungen durch Standards/Normen oder der Wunsch nach gut strukturiertem Code sind Konstanten, die bei der Auswahl eines Coding Standards berücksichtigt werden müssen.

Abhänig von der Programmiersprache

Folgende Kriterien gehören zu den wichtigsten:

- Programmiersprache
- Anforderungen durch Kunden
- Forderung durch Standards und Normen (DO-178B, IEC 61508-7)
- Anforderung durch Projektpartner

Wird, wie z.B. in DO-178B, kein expliziter Coding Standard vorgeschrieben, so muss eine Auswahl getroffen werden.
Das Eingeben des Ausdrucks "Coding Standard" bei Google Suche liefert mehrere Hundert Treffer. Dies zeigt die enorme Anzahl an bereits vorhandenen Coding Standards für die Unterschiedlichsten Programmiersprachen (C, C++, Java, PHP, .net extentions, usw.). Dies kommt von der Tatsache, dass es eben keinen allgemeingültigen, sondern Projektspezifische, Coding Standards gibt.

Welcher Coding Standard ist nun aber für das Projekt der richtige? Muss zwangsläufig ein eigener erstellt werden? Was genau muss ein solcher Standard enthalten um zum Projekt zu passen?
Die Antworten auf diese Fragen liefert der Projektrahmen.
Die verwendete Programmiersprache schränkt die Auswahl bereits ein. Ebenso setzt die Art der Applikation einen Rahmen. Ein Coding Standard für ein Windows Programm, das Werte aus einer statischen Tabelle grafisch aufbereitet, ist ein anderer, als der für eine sicherheitskritische Steuerung. Sicherheitskritikalität ist also auch bei Coding Standards ein zentrales Thema.
Angenommen das zu entwickelnde System soll in C geschrieben werden und wird als sicherheitskritisch nach IEC 61508 eingestuft, so muss ein Coding Standard verwendet werden. Dies ist der Norm zu entnehmen und wird durch Tabelle 3 verdeutlicht.

C ist ohne Coding Standard nicht für sicherheitskritische Systeme geeignet

Ein möglicher Coding Standard für die Konstellation C, safety-critical, IEC61508-konform, wäre MISRA-C. Dieser dient häufig in Projekten als Grundlage zur Erstellung eines eigenen Standards. Auch in Avionikprojekten fließen oftmals teile des MISRA Coding Standards ein. Gernerell sind Avionikprojekte von Natur aus sehr mächtig in ihrer Gesamtkomplexität, deshalb werden für solche Projekte in der Regel immer eigene Coding Standards erstellt. Diese beruhen, neben den Kundenanforderungen und Anforderungen durch Normen, auf langer Erfahrung.

[3]Zur automatisierten Dokumentierung gibt es Werkzeuge, wie z.B. Doxygen oder Javadoc, welche eine einheitliche Struktur der Kommentare voraussetzten.

Programmiersprache	SIL1	SIL2	SIL3	SIL 4
C	+	o	-	-
C mit Teilmenge und Codierrichtlinie und Verwendung von statischen Analysewerkzeugen	++	++	++	++

"++": Das Verfahren oder die Maßnahme wird für diesen Sicherheits-Integritätslevel besonders empfohlen. [...]
"+": Das Verfahren oder die Maßnahme wird für diesen Sicherheits-Integritätslevel empfohlen. [...]
"o": Das Verfahren oder die Maßnahme hat keine Empfehlung für oder gegen die Verwendung.
"–": Das Verfahren oder die Maßnahme wird für diesen Sicherheits-Integritätslevel ausdrücklich nicht empfohlen.

Tabelle 3: IEC61508 Empfehlungen für Programmiersprachen (gekürzt aus IEC 61508-7 [CEN01], S. 81, Tabelle C.1 - Empfehlungen für bestimmte Programmiersprachen)

4.2 Mögliche Standards abhängig von der Programmiersprache

Das letzte Kapitel hat gezeigt, dass die Programmiersprache das wichtigste Kriterium zur Auswahl eines Coding Standards ist. Unter der Annahme, das ein Coding Standard für sicherheitskritische Embedded-Projekte gesucht wird, werden hier Coding Standards in Abhängigkeit zur Programmiersprache genannt.

4.2.1 C

C ist die in der Embedded-Programmierung am häufigsten eingesetzt Programmiersprache. Dies ergibt sich aus der sehr einfachen Möglichkeiten, direkt auf die Hardware zuzugreifen. Zudem lässt sich mit C eine gute Kontrolle über die Ressourcen (Speicher, CPU) erreichen. C hat aber auch seine Tücken - ein unerfahrener Programmierer macht schnell Fehler, die nicht offensichtlich sind aber zu Nebeneffekten führen können die dann schwer diagnostizierbar sind (Laufzeitüberprüfung). Ausserdem ist die Sprache C, wie sie in ISO 9899:1990 definiert ist, oft nicht eindeutig. So ist manches Verhalten implementationsabhänig (Compiler) oder auch undefiniert. Aus diesem Grund macht es Sinn eine Richtlinie für die Erstellung von C-Code zu verwenden.

C ist im Embeddedbereich State-of-the-art

Der populärste C Coding Standard ist MISRA-C in der aktuellen Fassung MISRA-C:2004 [MIS04].
Dieser Coding Standard wurde von der Motor Industry Software Reliablility Asscociation (MISRA) erarbeitet. Und enthält 141 Regeln von denen 121 als verbindlich und 20 als empfohlen eingeordnet sind. MISRA-C beschreibt eine "sichere" Teilmenge der in ISO 9899:1990 definierten Pro-

MISRA-C

grammiersprache C. Zudem ist der Coding Standard in Anlehnung an die IEC 61508 erstellt. Die ursprüngliche Fassung war speziell für die Entwicklung im Automotivebereich ausgerichtet, mit der Fassung von 2004 wurde der Standard aber verallgemeinert.

Der Coding Standard beschreibt Regeln zu den folgenden Kategorien ([MIS04] S. 1):

- Umgebung
- Spracherweiterungen
- Dokumentation
- Character-Sets
- Identifiers (Symbole)
- Typen
- Konstanten
- Deklaration und Definition
- Initialisierung
- Arithmetische Type-Konvertierung
- Pointer Type-Konvertierung
- Expressions
- Controll Statement Expressions
- Control Statements
- Switch Anweisungen
- Funktionen
- Pointer und Arrays
- Strukturen und Unions
- Präprozessoranweisungen
- Standard-Librarys
- Laufzeitfehler

Neben den eigentlichen Regeln, wird im Dokument zum Standard auf Problemstellungen eingegangen, die sich in der Embedded-Entwicklung ergeben und die zu "unsicherem" Code führen können; und es werden häufige Fehler, die bei der Programmierung unter C gemacht werden, durch die Regeln ausgeschlossen. Zudem wird die Absicht, die hinter einer Regel steckt, stets erklärt und häufig anhand von Beispielen verdeutlicht.

Beispiele für Regeln, entnommen aus dem MISRA-C Standard [MIS04].

- "Rule 9.1 (required): All automatic variables shall have been assigned a value before being used.

  The intent of this rule is that all variables shall have been written to before they are read. This does not necessarily require initialisation at declaration. [...]" [MIS04] S. 33

- "Rule 11.1 (required): Conversions shall not be performed between pointer to a function and andy type other than an integral type.

  Conversion of a function pointer to a different type of pointer results in undefined behaviour. This means, for example, that a pointer to a function cannot be converted to a pointer to a different type of function. " [MIS04] S. 47

- "Rule 14.7 (required): A fucntion shall have a single point of exit at the end of the function.

  This is required by IEC 61508, under good programming style." [MIS04] S. 61

- "Rule 14.4 (required): The goto statement shall not be used." [MIS04] S. 61

Klare Struktur

Es fällt schnell auf, dass die Regeln durch ihre do / do not Struktur sehr klar formuliert sind. Dies macht es für den Programmierer einfach sie anzuwenden. Allerdings ist es in der Praxis nicht immer Zweckmäßig alle 141 Regeln strikt zu befolgen. Denn oftmals lässt sich eine Verletzung unter bestimmten Gegebenheiten nicht vermeiden. Hier hat sich die sog. Deviation Procedure durchgesetzt, in welcher der Programmierer Regeln, die er teilweise oder gar nicht beachtet, dokumentiert (Warum war die Verletzung notwendig?). Es gibt noch andere Formen der Deviation Procedure, z.B. die Project Deviation, bei der zu beginn des Projektes der anzuwendende Regelsatz, sowie die zu ignorierenden Regeln festgelegt werden. In der Praxis werden beide Varianten, teilweise auch in Kombination, angewandt.

Anpassung des Standards

An dieser Stelle möchte ich eine Literaturempfehlung aussprechen. Das Werk "Safer C: Developing Software for High-Integrity and Safety-Critical Systems" von Les Hatton [Hat95] verdeutlicht die Problematiken, die bei der Verwendung von C in sicherheitskritischen Systemen auftreten und kann in gewisser Weise als Leitfaden bei der Entwicklung solcher Systeme dienen.

Leitfaden zur Entwicklung

4.2.2 C++

Da C++ in sicherheitskritischer Embedded-Software immer häufiger eingesetzt wird, gibt es für diese Programmiersprache auch populäre Coding Standards.

C++ wird immer häufiger eingesetzt

Der Joint Strike Fighter Air Vehicle C++ Coding Standard (JSF C++) [Cor05] der Teilweise auf vielen MISRA-C Regeln aufbaut, und aus der Avionik stammt, beschreibt erstmals die Verwendung von C++ in sicherheitskritischer Embedded Software, in dem er eine Teilmenge von C++ beschreibt. Hierzu definiert er 221 Regeln. Ein signifikanter Unterschied wird durch Regeln zur Codemetrik sichtbar, welche in MISRA-C nicht vorhanden sind. Beispielsweise legt der Standard die maximale Länge einer Funktion fest; "Rule 1: Any one function (or method) will contain no more than 200 logical source lines of code (L-SLOCs)." ([Cor05] S. 12)

JSF C++

Nach dem Erscheinen des JSF C++, reagierte auch MISRA mit ihrem MISRA-C++ [MIS08] Coding Standard auf die Situation, dass C++ vermehrt in der Embedded Software eingesetzt wird.

MISRA-C++

Der Einsatz von C++ in Sicherheitskritischen Systemen, bringt gewisse Probleme mit sich, auf diese reagieren die Coding Standards. Einige Beispiele werden im Folgenden aufgeführt:

- Präprozessor Anweisungen sind bis auf "include-guards" nicht erlaubt
- Starke Restriktionen für Kontrollstrukturen (if,else,switch,while,for,do)
- Restriktionen bei der Verwendung von goto und setjmp/longjmp
- Einschränkung von impliziten und expliziten Typecasts
- Instanzvariablen in Klassen sind generell "private"
- Restriktion der Mehrfachvererbung
- Verwendung von dynamischer Speicher-allocation stark eingeschränkt (new, delete)

Um den Nutzen und die Absicht die hinter diesen Konventionen steckt verstehen zu können, wird exemplarisch das Beispiel "Einschränkung von

impliziten und expliziten Typecasts" aufgegriffen. Hinter dieser Einschränkung verbirgt sich die Anforderung nach der Portabilität von Quellcode, denn C++ definiert keine fixen Speichergrößen für die Standard-Datentypen. C++ (auch so C) führt arithmetische Operationen immer als int oder long aus.

Das Codelisting 1 zeigt ein Beispiel, bei dem zwei Werte vom Datentyp short multipliziert werden. Das Ergebnis wird in einer integer Variable gespeichert. Das Ergebnis dieser arithmetischen Operation ist auf einem 32Bit System definiert und wäre wie erwartet 100000, hingegen ist auf einem 16Bit Rechner das Ergebnis nicht definiert, denn die int32 Variable würde vor der Operation in eine int (16Bit) Variable gewandelt werden, das Ergebnis würde zu einem Überlauf führen und das falsche Ergebnis würde zurück in die int32 Variable geschrieben werden; "result" enthält einen falschen Ergebniswert. Somit ist dieser Quellcode nicht auf ein 16bit System portierbar.

Ein realitätsnahes Beispiel

```
1  short a = 10000;
2  short b = 10;
3  int32 result = a * b;
```

Listing 1: Arithmetische Operation C++

Ein weiteres Beispiel für Probleme mit Datentypen ist der C++ Datentyp "char". Denn er kann entweder signed oder unsigned sein. Deshalb darf er in beiden Coding Standards ohne Präfix nur zur Speicherung von Zeichen verwendet werden.

Vergleich JSF C++ sowie MISRA-C++ beruhen auf vielen MISRA-C Regeln. Beide Standards erlauben Makros nur in sehr restriktiver Form. Ebenso fordern beide Standards defensive Programmiertechniken. Unterschiede werden z.B. in den JSF C++ Regeln zum Codestil und Codemetriken sichtbar, diese enthält MISRA-C++ nicht. MISRA-C++ erlaubt Exceptions, nicht aber so JSF C++. Null-Pointer werden nach MISRA-C++ durch das NULL-Macro gehandhabt, JSF C++ empfiehlt hier die Verwendung von 0.

4.2.3 Java

Java ist kaum gebräuchlich in safety-critical Systemen

Für Java gibt es keinen repräsentativen Coding Standard für sicherheitskritischen, echtzeitfähigen Code. Dies lässt sich durch die Tatsache erklären, dass Java noch keinen wirklichen Einzug in den sicherheitskritischen Embedded-Bereich gehalten hat. Zwar gibt es gut durchdachte Ansätze zum Einsatz von Java im Embedded-Bereich, so z.B. Real-Time Java und

die Jazelle[4] Technologie, allerdings finden diese in der Praxis noch kaum Anwendung. Langfristig gesehen wird Java auch im Embedded-Bereich eine immer größere Rolle spielen. Somit darf man davon ausgehen, dass in den nächsten Jahren ein repräsentativer Coding Standard für Java für den Embedded-Bereich erscheinen wird.

(Um Verwechslungen vorzubeugen: Java wird in der Embedded Software eingesetzt (Handy, PDA, ...) allerdings nicht im Kontext der harten Echtzeit und Sicherheitskritikalität - hier sind C/C++ die momentan noch bessere Alternative.)

4.3 MISRA-C Codebeispiel

In diesem Kapitel wird der MISRA-C Coding Standard anhand von einem Quellcodebeispiel verdeutlicht. Hierzu wurde als Grundlage die "MISRA-C Exemplar Suite" verwendet, diese enthält zu den meisten MISRA Regeln sehr ausführliche Codebeispiele. Um einen Überblick und ein Gefühl für MISRA-C zu bekommen, wurde der nachfolgende Code (Listings 2, 3, 4 und 5) aus Beispielen der Exemplar Suite erstellt und erweitert.
Der Code enthält Kommentare (grün gefärbt), die auf die jeweiligen Regeln hinweisen und eine Aussage über die Konformität (compliant / not compliant) der jeweiligen Code-Stelle machen.

Listing 2: collection.c

```
1  #include "mc2_types.h"
2  #include "mc2_header.h"
3  #include "stdlib.h"
4  /*
5   *  20.9  The  input/output library <stdio.h> shall not be used in production code
6   */
7  #include <stdio.h>              /* Not Compliant to include stdio.h */
8  /*
9   *  20.12  The time handling functions of <time.h> shall not be used
10  */
11 #include <time.h>               /* Not Compliant to include time.h */
12
13
14
15  /*
16  * 2.3 The character sequence
17  *     "/*" shall not be used within a comment.   Not Compliant
18  */
19 #define DISABLE_IRQS asm { "CLI" }
20 #define ENABLE_IRQS  asm { "SEI" }
21
22 static void mc2_0201_fn1 ( void );
23 static void mc2_0201_fn2 ( void );      // This comment is Not Compliant: Rule 2.2
24 static void mc2_0201_fn3 ( void );
25 static void mc2_0401_fn1 ( void );
26 static void mc2_0601_fn1 ( void );
27 static void mc2_0701_fn1 ( void );
28 static void mc2_0802_fn1 ( const int32_t qualifier_param );
29 static void mc2_0807_fn1 ( void );
30 static void mc2_0901_fn1 ( void );
31 static void mc2_1306_fn1 ( void );
32 static void mc2_2004_fn1 ( void );
33
34 /*
35  * 9.3 In an enumerator list, the "=" construct shall not be used to
36  *     explicitly initialise members other than the first, unless all
37  *     items are explicitly initialised.
38  */
```

[4]Jazelle ist eine von ARM entwickelte Technologie, die darauf abzielt eine Java VM direkt in den Prozessor zu integrieren (Stichwort: Java Prozessor). Es gibt bereits Prozessoren die diese Technik teilweise unterstützen, z.B. ARM7EJ-S.

```
39 typedef enum
40 {
41    mc2_0903_green_1,
42    mc2_0903_yellow_1 = 5       /* Not Compliant */
43 } mc2_0903_colour_1;
44
45 typedef enum
46 {
47    mc2_0903_green_2  = 5,
48    mc2_0903_yellow_2 = 5       /* Compliant */
49 } mc2_0903_colour_2;
50
51 /*
52  * 6.4 Bit fields shall only be defined to be of type unsigned int or signed
53  *     int.
54  */
55 struct bitfield
56 {
57    unsigned bf_unsigned:4;
58    signed bf_signed:2;
59    unsigned int bf_unsigned_i:6;
60    signed int bf_signed_i:5;
61    int bf_int:5;                          /* Not Compliant */
62    char bf_char:6;                        /* Not Compliant */
63    enum bf_enum { bf_enumerated }:2;      /* Not Compliant */
64    short bf_short: 5;                     /* Not Compliant */
65    unsigned char bf_unsigned_char: 3;     /* Not Compliant */
66 };
67
68 /*
69  * 8.7 Objects shall be defined at block scope if they are only accessed from
70  *     within a single function.
71  */
72 static int32_t should_be_local;       /* Not Compliant */
73
74 static void mc2_0807_fn1 ( void )
75 {
76    static int32_t local_ssd = 0;      /* Compliant */
77 }
78
79
80 /*
81  * 2.1 Assembly language shall be encapsulated and isolated.
82  * All use of assembler in this file is not compliant with Rule 1.1
83  */
84 /* Conforming example - all assembler is encapsulated in MACROS               */
85 static void mc2_0201_fn1 ( void )
86 {
87    DISABLE_IRQS;                          /* Disable all interrupts.          */
88    use_int16 ( 0 );                       /* Do something.                    */
89    ENABLE_IRQS;                           /* Enable all interrupts.           */
90 }
91
92 /* Non-conforming example - assembler and 'C' are mixed in a 'C' function.    */
93 static void mc2_0201_fn2 ( void )
94 {
95    asm { "CLI" };                         /* Not Compliant                    */
96    use_int16 ( 1 );                       /* Do something.                    */
97    asm { "SEI" };                         /* Not Compliant                    */
98 }
99
100 /* Conforming example - all assembler is encapsulated in a 'C' function.      */
101 static void mc2_0201_fn3 ( void )
102 {
103    asm { "CLI" };                         /* Disable all interrupts.          */
104    asm { "NOP" };                         /* Do nothing.                      */
105    asm { "SEI" };                         /* Re-enable all interrupts.        */
106 }
107
108 static void mc2_0401_fn1 ( void )
109 {
110    /* Not Compliant - all hexadecimal escape sequences are banned */
111    /*  4.1 Only those escape sequences that are defined in the ISO C standard shall  be used.
112         /* The following list is not exhaustive */
113    use_char ( '\x0' );     /* Not Compliant */
114    use_char ( '\x01' );    /* Not Compliant */
115    use_char ( '\x12' );    /* Not Compliant */
116    use_char ( '\xAB' );    /* Not Compliant */
117
118    /* 4.2 Trigraphs shall not be used. */
119    /* Trigraphs - Non Compliant */
120    use_char_ptr ( "??(" );      /* Not Compliant */
121    use_char_ptr ( "??)" );      /* Not Compliant */
122    use_char_ptr ( "??<" );      /* Not Compliant */
123 }
124
125 /*
126  * 6.1 The plain char type shall be used only for the storage and use of
127  *     character values.
128   * 6.2 Signed and unsigned char type shall be used only for the storage and use
129  *     of numeric values.
130  */
131 static void mc2_0601_fn1 ( void )
```

```
132 {
133    char_t character_value = 'a';        /* Compliant */
134    char_t plain_character_data = 'b';
135    int8_t numeric_value = 5;
136    int8_t signed_numeric_value = 0;
137    uint8_t unsigned_numeric_value = 0U; /* U to comply with Rule 10.6 */
138
139    use_char ( character_value );
140     /* Compliant - but implementation dependent */
141    character_value = ( char_t ) numeric_value;
142
143    while ( character_value == numeric_value )     /* Not Compliant - types */
144    {
145       while ( character_value == '2' )
146       {
147          character_value = numeric_value;         /* Not Compliant */
148           }
149         }
150
151    while ( signed_numeric_value == '1' )      /* Not Compliant */
152    {
153      unsigned_numeric_value = plain_character_data;
154    }
155 }
156
157 /*
158  * 7.1 Octal constants (other than zero) and octal escape sequences shall not be
159  *     used.
160  */
161 static void mc2_0701_fn1 ( void )
162 {
163    int8_t octal_const, n_octal_const;
164    char_t * n_octal_point = "abc\017";           /* Not Compliant */
165
166    n_octal_const = 017;                          /* Not Compliant */
167
168 }
169
170 /*
171  * 8.2 Whenever an object or function is declared or defined, it's type shall be
172  *     explicitly stated.
173  */
174 /* 8.3 For each function parameter the type given in the declaration and
175  *     definition shall be identical, and the return types shall also be
176  *     identical.
177  */
178 static void mc2_0802_fn1 ( int32_t qualifier_param ) /* Not Compliant */
179 {
180    const implicit_const_int = 1;  /* Not Compliant - implicit type */
181    const int16_t explicit_const_int = 2;
182 }
183
184 /*
185  * 9.1 All automatic variables shall have been assigned a value before
186  *     being used.
187  */
188 static void mc2_0901_fn1 ( void )
189 {
190         int32_t variable1 = 1;
191         int32_t variable2 = 1;
192         int32_t variable3;
193         int32_t variable4;
194
195         variable1 = variable2;          /* Compliant                      */
196         variable3 = variable4;          /* Not Compliant               */
197 }
198
199 /*
200  * 13.6 Numeric variables being used within a for loop for iteration
201  *      counting shall not be modified in the body of the loop.
202  *
203  */
204 static void mc2_1306_fn1 ( void )
205 {
206    uint16_t  mc2_1306_var1;
207    uint16_t  mc2_1306_var2;
208
209    mc2_1306_var1 = 5U;
210
211    for ( mc2_1306_var2 = 1U;
212          ( mc2_1306_var2 < 22U ) && ( mc2_1306_var1 != 8U );
213          mc2_1306_var2++ )
214    {
215       mc2_1306_var2 = mc2_1306_var2 + mc2_1306_var1;    /* Not compliant */
216       mc2_1306_var1 = 6U ( );
217    }
218 }
219
220 /*
221  *  20.4  Dynamic heap memory shall not be used.
222  */
223 static void mc2_2004_fn1 ( void )
224 {
```

```
225   char_t  *  mc2_2004_p ;
226
227   mc2_2004_p = ( char_t * ) malloc ( 11U );  /* Not Compliant: use of malloc  */
228   (void) realloc ( mc2_2004_p, 20U );        /* Not Compliant: use of realloc */
229   free ( mc2_2004_p );                      /* Not Compliant: use of free    */
230 }
231
232 void mc2 ( void )
233 {
234    const int32_t qualifier_param = 10;
235    /* Not compliant: 2.4 Sections of code should not be "commented out". */
236    /*
237        mc2_0201_fn1 ( );
238        */
239    mc2_0201_fn2 ( );
240    mc2_0201_fn3 ( );
241    mc2_0401_fn1 ( );
242    mc2_0601_fn1 ( );
243    mc2_0701_fn1 ( );
244    mc2_0802_fn1 ( qualifier_param );
245    mc2_0807_fn1 ( );
246    mc2_0901_fn1 ( );
247    mc2_1306_fn1 ( );
248    mc2_2004_fn1 ( );
249 }
250
251
252 /* end of mc2_0201.c */
```

Listing 3: mc2_main.c

```
1 #include "mc2_types.h"
2 #include "mc2_header.h"
3
4 int32_t main ( void );
5
6 int32_t main ( void )
7 {
8    mc2 ( );
9
10    return 0;
11 }
```

Listing 4: mc2_types.h

```
1 #ifndef MC2_TYPES_H
2 #define MC2_TYPES_H
3
4 typedef signed   int            bool_t; /* Used for Boolean by enforcement examples */
5 typedef          char           char_t;
6 typedef signed   char           int8_t;
7 typedef signed   short          int16_t;
8 typedef signed   int            int32_t;
9 typedef signed   long           int64_t;
10 typedef unsigned char           uint8_t;
11 typedef unsigned short          uint16_t;
12 typedef unsigned int            uint32_t;
13 typedef unsigned long           uint64_t;
14 typedef          float          float32_t;
15 typedef          double         float64_t;
16 typedef          long double    float128_t;
17
18 #endif
```

Listing 5: mc2_header.h

```
1 #ifdef MC2_HEADER_H
2
3 #error File mc2_header.h included 2nd time
4
5 #else
6 #define MC2_HEADER_H
7
8 /* Functions returning a variable */
9
10 extern bool_t get_bool ( void );
11 extern char_t get_char ( void );
12 extern int8_t get_int8 ( void );
13 extern int16_t get_int16 ( void );
14 extern int32_t get_int32 ( void );
15 extern int64_t get_int64 ( void );
16 extern uint8_t get_uint8 ( void );
17 extern uint16_t get_uint16 ( void );
18 extern uint32_t get_uint32 ( void );
19 extern uint64_t get_uint64 ( void );
20 extern float32_t get_float32 ( void );
21 extern float64_t get_float64 ( void );
22 extern float128_t get_float128 ( void );
```

```
23
24 extern bool_t *get_bool_ptr ( void );
25 extern char_t *get_char_ptr ( void );
26 extern int8_t *get_int8_ptr ( void );
27 extern int16_t *get_int16_ptr ( void );
28 extern int32_t *get_int32_ptr ( void );
29 extern int64_t *get_int64_ptr ( void );
30 extern uint8_t *get_uint8_ptr ( void );
31 extern uint16_t *get_uint16_ptr ( void );
32 extern uint32_t *get_uint32_ptr ( void );
33 extern uint64_t *get_uint64_ptr ( void );
34 extern float32_t *get_float32_ptr ( void );
35 extern float64_t *get_float64_ptr ( void );
36 extern float128_t *get_float128_ptr ( void );
37
38 /* Functions that use a variable */
39
40 extern void use_bool ( bool_t use_bool_param );
41 extern void use_char ( char_t use_char_param );
42 extern void use_int8 ( int8_t use_int8_param );
43 extern void use_int16 ( int16_t use_int16_param );
44 extern void use_int32 ( int32_t use_int32_param );
45 extern void use_int64 ( int64_t use_int64_param );
46 extern void use_uint8 ( uint8_t use_uint8_param );
47 extern void use_uint16 ( uint16_t use_uint16_param );
48 extern void use_uint32 ( uint32_t use_uint32_param );
49 extern void use_uint64 ( uint64_t use_uint64_param );
50 extern void use_float32 ( float32_t use_float32_param );
51 extern void use_float64 ( float64_t use_float64_param );
52 extern void use_float128 ( float128_t use_float128_param );
53
54 extern void use_bool_ptr ( bool_t *use_bool_ptr_param );
55 extern void use_char_ptr ( char_t *use_char_ptr_param );
56 extern void use_int8_ptr ( int8_t *use_int8_ptr_param );
57 extern void use_int16_ptr ( int16_t *use_int16_ptr_param );
58 extern void use_int32_ptr ( int32_t *use_int32_ptr_param );
59 extern void use_int64_ptr ( int64_t *use_int64_ptr_param );
60 extern void use_uint8_ptr ( uint8_t *use_uint8_ptr_param );
61 extern void use_uint16_ptr ( uint16_t *use_uint16_ptr_param );
62 extern void use_uint32_ptr ( uint32_t *use_uint32_ptr_param );
63 extern void use_uint64_ptr ( uint64_t *use_uint64_ptr_param );
64 extern void use_float32_ptr ( float32_t *use_float32_ptr_param );
65 extern void use_float64_ptr ( float64_t *use_float64_ptr_param );
66 extern void use_float128_ptr ( float128_t *use_float128_ptr_param );
67
68
69 extern void mc2 ( void );
70
71 #endif
```

4.4 Prüfung der Einhaltung

Sicherheitskritische Standards und Normen fordern eine Überprüfung und Analyse des Codes durch geeignete Programme. Zum einen soll die Einhaltung eines vorgegebenen Coding Standards überprüft werden, zum anderen soll die Komplexität des Codes (Nesting usw.) bestimmt werden. Zur Durchführung dieser Aufgabe gibt es Software-Werkzeuge, die den Code (semi-)automatisch statisch überprüfen können.

OpenSource Programme zur "statischen Code-Analyse":

- BLAST
- Fragma-C
- Splint

Kommerzielle Programme zur "statischen Code-Analyse":

- Greenhills Software Double Check
- LDRA Testbed

- PC-Lint
- QA-C
- Red Lizard's Goanna

PC-Lint und QA-C zählen zu den bekanntesten Code-Analyse-Tools. Die Testergebnisse dieser Werkzeuge können zur Zertifizierung nach den Standards DO-178B sowie IEC 61508 beitragen.

PC-Lint

Exemplarisch wird PC-Lint vorgestellt. PC-Lint ist allgemein gesprochen ein "Rule-Checker". Anhand von einem vorgegebenen Regelwerk wird Quelltext auf Konformität überprüft. PC-Lint bringt MISRA-C bereits als Regelwerk mit (zu dem Enthält es Prüfungen zu Scott Mayers Effective C++ und Dan Saks), eine Anpassung der Regeln ist jederzeit manuell möglich. Es kann also auch ein neuer, selbst erstellter Standard eingepflegt werden. Dies ist in Anbetracht des häufigen Einsatzes von Deviation Procedures (wie in Abschnitt 4.2.1) sinnvoll.

Integration in IDE

PC-Lint ist ein Werkzeug, das sich in die bekannteren Entwicklungsumgebungen integrieren lässt. Somit ist eine Ausführung bequem per Mausklick möglich. Alternativ lässt sich PC-Lint komplett über die Befehlszeile bedienen, dies führt zu einer hohen Flexibilität in der Anwendung.
Die Prüfung lässt sich für ein ganzes Projekt oder aber für ein Modul durchführen. PC-Lint erzeugt nach einem Prüflauf eine Ausgabe-Datei, in der die gefundenen nicht konformen Codestellen vermerkt sind. Wurde bei der Konfiguration zusätzlich noch das Regelwerk von Mayers und/oder Saks gewählt, dann bekommt man außerdem hinweise auf potentielle Fehlerquellen im analysierten Code. Das Ganze kann man sich ähnlich wie die Fehlermeldungen eines Compilers vorstellen. Und in der Tat lässt sich, z.B. in Visual Studio[5], die PC-Lint Ausgabe elegant in das Konsolen-Fenster umleiten.

Es bedarf etwas Einarbeitungszeit um den Umgang mit diesem Werkzeug zu beherrschen. Unterstützend wirken hierbei die Ausführliche Dokumentation sowie Beispiele.
Wie bereits erwähnt wurde, ist PC-Lint ein kommerzielles Tool, das aber im Vergleich zu ähnlichen Produkten in der unteren Preisklasse[6] schwebt. Leider stellt der Hersteller "Gimpel Software" keine Testversion bereit, jedoch bietet die Herstellerwebseite[7] eine Live-Demo, in der man auch selbstgeschriebenen Code Prüfen lassen kann.
Aus meiner Sicht, ist PC-Lint den in der Auflistung (s.o.) genannten open-Source Prüfwerkzeugen überlegen. Dies äußert sich durch seinen großen Umfang, die ausgereifte Art eigene Regeln ein zu pflegen, seiner hohen Akzeptanz in der Industrie und nicht zuletzt der Anerkennung durch die Zertifizierungsstellen.

[5] IDE von Microsoft.
[6] Preisliste auf Herstellerwebseite: http://www.gimpel.com
[7] vgl. Fn. 6.

5 Zusammenfassung

Diese Arbeit hat gezeigt, dass bei der Entwicklung von sicherheitskritischen Systemen bestimmte Richtlinien eingehalten werden müssen. Diese Richtlinien werden zum einen von den zuständigen Zertifzierungsbehörden vorgeschrieben, können aber auch zum anderen selbst erweitert und angepasst werden.
Es hat sich herausgestellt das die Anwendung von Richtlinien auch bei unkritischen Systemen Sinn macht, denn dadurch wird dem Projekt ein strukturierter Rahmen vorgegeben.

Es wurde deutlich, dass diese Standards in der Regel zwar keine konkreten Methoden zur Software-Entwicklung vorschreiben, jedoch in der Praxis oftmals ein objektorientiertes und modellorientiertes Vorgehen gewählt wird. Die letztendliche Implementierung durch Code geschieht jedoch manuell, und höchstens in Anlehnung an ein Modell. Auf keinen Fall wird aber ein Modell 1:1 umgesetzt.[8] Es hat sich weiter gezeigt, dass die Beschreibung solcher Modelle oftmals durch formale, mathematische Sprachen erfolgt.

Um die Auswirkungen, die Zertifizierungsrichtlinien auf den letztendlich erstellten Code haben, zu verdeutlichen, wird der Begriff Coding Standard eingeführt. Und es wird vermittelt, dass ein solcher durch die meisten Zertifzierungsrichtlinien vorgeschrieben ist. Es zeigte sich, dass die Auswahl eines Coding Standards stark von der verwendeten Programmiersprache und den Anforderungen durch Zertifzierungsrichtlinien abhängt.
Um ein konkretes Beispiel aus der Industrie anzuführen, wurde der MISRA-C Coding Standard ausführlich betrachtet und durch Beispiele dessen Anwendung erläutert. Um die Einhaltung eines Coding Standards zu Prüfen, wurden Testwerkzeuge vorgestellt, die für diese Aufgabe geeignet sind.

Fazit Diese Arbeit hat deutlich gemacht, wie komplex und anspruchsvoll die Entwicklung von sicherheitskritischen Systemen ist. Und es wird klar, dass Dokumentation (im weitesten Sinne) für einen sehr hohen Aufwand sorgt, der aber unumgänglich ist (Avionik Projektaufwand: 10% Code-Erstellung; 90% Tests, Dokumentation, Verifizierung & Validierung, Management, Zertifizierung).

[8] Ein Modell ist ein Abbild der Wirklichkeit, nicht jedoch die Wirklichkeit selbst. Dies wird oftmals vergessen oder falsch verstanden.

Literatur

[AbRS94] Dr. Ramon Acosta, Carla burns, William Rzepka, and James Sidoran. *Applying Rapid Prototyping Techniques in the Requirements Engineering Environment.* IEEE, 1994.

[BB87] Bolognesi and Brinksma. *Introduction to the ISO specification language LOTOS.* Computer Networks and ISDN Systems, 1987. http://jucmnav.softwareengineering.ca/ftp/pub/Lotos/Intro/BB-LotosTutorial.pdf.

[BH99] Bharadway and Heitmeyer. *Hardware / Software co-design and co-validation using SCR method.* IEEE, 1999.

[CEN01] CENELEC. *Europäische Norm 61508: Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer elektronischer Systeme (Deutsche Fassung).* VDE VERLAG GMBH, Berlin, 2001.

[Cor05] Lockheed Martin Corporation. *Joint Strike Fighter Air Vehicle C++ Coding Standards for the system development and demonstration program.* MISRA Limited, 2005.

[FK04] Peter Forbrig and Immo Kerner. *Softwareentwicklung.* Fachbuchverlag Leipzig, München, 1. auflage edition, 2004. ISBN 3-446-22578-1.

[Hat95] Les Hatton. *Safer C: Developing Software for High-Integrity and Safety-Critical Systems (The Mcgraw-Hill International Series in Software Engineering).* McGraw-Hill Companies, 1995. ISBN 9780077076405.

[Inc92] RTCA Inc. *Software Considerations in Airborne Systems and Equipment Certification.* RTCA, Washington D.C. / USA, 1992.

[Jon90] Cliff Jones. *Systematic Software Development using VDM.* Prentice hall, New York, 2. auflage edition, 1990. ISBN 0-1388-0733-7.

[MIS04] MISRA. *MISRA-C: 2004, Guidelines for the use of the C language in critical systems.* MISRA Limited, Warwickshire UK, 2004. ISBN 0-9524-1562-3.

[MIS08] MISRA. *MISRA-C++: 2008, Guidelines for the use of the C++ language in critical systems.* MISRA Limited, Warwickshire UK, 2008.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice hall, New York, 2. auflage edition, 1992. ISBN 0-1397-8529-9.