Veröffentlicht auf dem Kongress ECT 2002, 04.-06.06.2002, Augsburg

Integration automotiver Standards in die Seriencode-Generierung

Dipl.-Ing. Thomas Thomsen
dSPACE GmbH, Technologiepark 25, 33100 Paderborn
Tel. +49 5251 1638-772, Fax +49 5251 66529, email: tthomsen@dspace.de

Abstract – The introductory chapter of this article describes the current standards that can be applied to production-quality code generators. This is further detailed for three standards – OSEK/VDX, MISRA C and ISO/IEC 15504 (SPiCE) and explained by using dSPACE's TargetLink as an example. In addition, the last chapter contains a comparison between the emerging software quality standard ISO/IEC 15504 (SPiCE) and 'CMM for Software'.

Zusammenfassung – Der folgende Artikel beschreibt in seiner Einleitung heute verfügbare Standards und Normen, welche für die Entwicklung und den Einsatz von Seriencode-Generatoren anwendbar sind. Anhand dreier Normen – OSEK/VDX, MISRA C und ISO/IEC 15504 (SPiCE) – wird das am Beispiel des Seriencode-Generators TargetLink von dSPACE detailliert beschrieben. Dabei wird auf die Anwendbarkeit und die Implementierung der Standards eingegangen. Das letzte Kapitel enthält zusätzlich einen Vergleich zwischen dem kommenden Software-Qualitätsstandard ISO/IEC 15504 (SPiCE) und 'CMM for Software'.

1. Anwendbare Standards für die Code-Generierung

Software-Entwickler für Steuergeräte müssen heute für Entwicklungsprojekte eine Vielzahl technischer Normen und Qualitätsstandards berücksichtigen. Mit dem zunehmenden Einsatz von Tools in der Serienentwicklung wird dieses Thema auch bedeutend für Code-Generatoren. Der Artikel stellt die wichtigsten anwendbaren Normen und Standards zusammen und erklärt deren Implementierung am Beispiel des Code-Generators TargetLink von dSPACE [1].

Die bedeutendste technische Norm für einen Code-Generator ist die Norm seiner Ausgabesprache. In der Regel ist dies C, welche durch die ISO/IEC 9899 [6] international genormt wurde und identisch mit der ANSI X3.159 ist. Hierzu gibt es ergänzende Standards und Publikationen, die dem Zweck der Qualitätssicherung dienen. Jedem Software-Entwickler für Steuergeräte sollte das Buch „Safer C – Developing Software for High-Integrity and Safety-Critical Systems" von Les Hatton [3] bekannt sein. Das Buch sowie eine Vielzahl anderer Quellen bildeten die Grundlage für den britischen MISRA-C-Standard [14], welcher sich zunehmend in der Industrie durchzusetzen scheint. Die Auswirkungen von MISRA C auf die Code-Generierung werden im Kapitel „3. MISRA C" detaillierter erörtert.

Ebenfalls zu der Klasse der Sprachenstandards muss man die proprietären Tool-Sprachen zählen. Diese kommen in der Regel nicht von Verbänden oder internationalen Gremien, sondern haben sich im Laufe der Zeit als De-facto-Industriestandards durchgesetzt. Im Bereich blockdiagrammbasierter Simulationssprachen dürfte dieser De-facto–Status den Simulationstools von The MathWorks (MATLAB®/Simulink®/Stateflow®) zugeschrieben werden. Sie haben im Laufe der Jahre den Sprung von Anwendungen in der Forschung und Lehre hinein in die Industrie geschafft.

Aber auch hier gibt es firmenunabhängige Bestrebungen, einen Blocksatz für Simulationstools zu standardisieren. Vornehmlich ist hier der MSR MEGMA Blocksatz [15] zu nennen, welcher von Vertretern der Automobilindustrie spezifiziert wird. Für die Hersteller von Code-Generatoren hat ein Standardblocksatz den entscheidenden Vorteil, dass die abzubildende Semantik genau spezifiziert ist und sich nicht mehr willkürlich mit der Freigabe neuer Produktversionen des Simulationstools ändert. Ein Code-Generator wäre somit auch einfacher in eine neue Simulationsumgebung zu portieren. Zur Zeit ist noch nicht absehbar, ob sich ein standardisierter Blocksatz, wie beispielsweise durch MSR MEGMA vorgeschlagen, gegen die etablierten, proprietären Blocksätze einzelner Tool-Hersteller durchsetzen kann.

Im Bereich Echtzeit-Betriebssysteme ist die Entwicklung völlig anders verlaufen. Hier hat sich kein proprietäres RTOS im Automobilsektor durchsetzten können. Stattdessen hat sich eine herstellerübergreifende Initiative gebildet, die in den OSEK/VDX-Standard [9, 10, 11, 12] mündete. Das Angebot an kommerziell verfügbaren OSEK-Betriebssystemen ist reichhaltig und der Standard gilt als etabliert. Code-Generatoren werden diesen Standard integrieren müssen, um das Zusammenspiel zwischen Applikationssoftware und dem unterlegten Betriebssystem optimal für den Anwender zu gestalten. Ein von dSPACE entwickeltes Konzept für TargetLink wird im Kapitel „2. Der OSEK/VDX-Standard" an einigen Beispielen kurz vorgestellt.
Eine weitere Pflichtübung eines Code-Generators ist die enge Verknüpfung mit Applikationssystemen. Vom Anwender spezifizierte Variablen oder Parameter sollen in diesen Systemen zur Verfügung stehen. Dieses wird dem Applikationssystem über eine Beschreibungsdatei nach dem ASAM MCD 2MC Standard [13] bekannt gegeben. Alle wichtigen Applikationssysteme unterstützen heute diesen Standard und TargetLink kann die Datei parallel zum generierten Code erzeugen.
Letztendlich muss ein Code-Generator selber bestimmten Qualitätsansprüchen genügen. Auch hierzu lassen sich anwendbare (und nichtanwendbare) Normen finden. Die häufigsten, im Zusammenhang mit der Software-Entwicklung genannten Normen und Standards sind ISO/IEC 15504 (SPiCE), CMM, Bootstrap und IEC 61508. Der Artikel enthält eine kurze Vorstellung der ISO/IEC 15504 [7] und einen Vergleich zu 'CMM for Software' [4, 5] in den wesentlichen Punkten. dSPACE hat sich für die ISO/IEC 15504 entschieden. Kapitel „4. ISO/IEC 15504 und andere Software-Qualitätsstandards" zeigt die Grundlagen für diese Entscheidung auf.

2. Der OSEK/VDX-Standard

Der OSEK-Standard [9, 10, 11, 12] beschreibt eine offene Software-Architektur für den Betrieb von verteilten Regelungssystemen im Kraftfahrzeug. Damit soll die Wiederverwendbarkeit und Portierbarkeit von Steuergeräte-Softwarekomponenten erleichtert werden.
OSEK/VDX wurde als deutsch-französische Initiative 1993 ins Leben gerufen. Seitdem sind die Arbeiten für ereignisgesteuerte Systeme, wie sie in den Teilen OS, COM und NM beschrieben werden, weit fortgeschritten und sollen sich nun in der Praxis bewähren. Bedeutende Automobilhersteller, wie beispielsweise DaimlerChrysler, haben diesen Standard für ihre eigene Steuergeräte-Entwicklung akzeptiert und fordern ihn von ihren Zulieferern. Die Teile OS, COM, NM sowie OIL werden zur Zeit für die Überführung zum ISO-Standard vorbereitet. Die Arbeitsgruppen des OSEK/VDX-Konsortiums konzentrieren sich nun auf die Spezifikation von zeitgesteuerten und fehlertoleranten Systemen (OSEKtime OS und FTCOM).
Für einen Code-Generator sind vier Ebenen der Integration relevant, die im Einzelnen vorgestellt werden:

- Tool-Integration
- Task-Spezifikation
- Integration von Serviceroutinen
- Betriebssystemoptimierte Code-Generierung

Tool-Integration

Ein Code-Generator ist in der Regel mit einem Simulationswerkzeug eng verbunden. TargetLink von dSPACE ist nahtlos in die Tool-Umgebung Simulink und Stateflow von The MathWorks integriert. Während der Code-Generierung werden Modell-Dateien (*.mdl) über eine MATLAB API in den Code-Generator eingelesen und entsprechend den Vorgaben des Anwenders Code erzeugt. Die Grundlagen der Seriencode-Generierung wurden im SAE-Paper „Automatic Generation of Production Quality Code for ECUs" [1] detailliert beschrieben.
Mit der Integration eines Betriebssystems muss ein weiteres Werkzeug in den Spezifikations- und Generierungsprozess eingebunden werden. Der Anwender kann auf Modellebene Tasks und seine Attribute bestimmen, welche sowohl Auswirkungen auf den generierten Code als auch auf die Konfiguration des Betriebssystemkerns haben. Zu diesem Zweck existiert die Beschreibungssprache OIL (OIL: OSEK Implementation Language). Mittels einer OIL-Datei können Betriebssystemobjekte wie Tasks, Events, Messages und Ressourcen definiert werden. Traditionell werden OIL-Dateien per ASCII-Editor beschrieben oder vom OS-Hersteller wird ein OIL-Editor bereitgestellt, welcher per Dialogführung die Erstellung einer OIL-Datei ermöglicht.
Mit einem Code-Generator und Blockdiagrammen kommt nun eine weitere Datenquelle für die OIL-Datei hinzu. In Konfigurationsblöcken können auf Blockdiagrammebene Einstellungen zu Tasks ergänzt oder verän-

dert werden. Gleiches gilt für Inport- und Outport-Blöcke, in denen sich Messages definieren lassen. Alarme und Counter haben wiederum ihre eigenen Blöcke und deren Attribute wandern ebenfalls in die OIL-Datei. Beim Öffnen eines Blockdiagramms muss demnach der aktuelle Stand der OIL-Datei eingelesen werden und OIL-Objekte bzw. deren At-tribute den einzelnen Blöcken zugeordnet werden. Sie stehen dann dem Anwender im Blockdiagramm zur Verfügung und können verändert werden. Spätestens beim Schließen eines Blockdiagramms müssen sie wieder in die OIL-Datei zurückgeschrieben werden. Das Rückschreiben beinhaltet Konsistenzprüfungen, um offensichtlich falsche oder unstimmige Eingaben abzufangen.
Mit den aktualisierten OIL-Dateien kann sich der Anwender sofort einen OS-Kern vom System Generator bauen lassen, der genau zu den in den Blockdiagrammen spezifizierten Tasks passt. Er muss sich nicht mehr um die Konsistenz zwischen Blockdiagrammen, generiertem C-Code und dem Betriebssystem kümmern. Tippfehler, Syntaxfehler oder Fehlspezifikationen in der OIL-Datei gehören der Vergangenheit an. Anwender müssen nicht mehr die Beschreibungssprache OIL im Detail beherrschen.

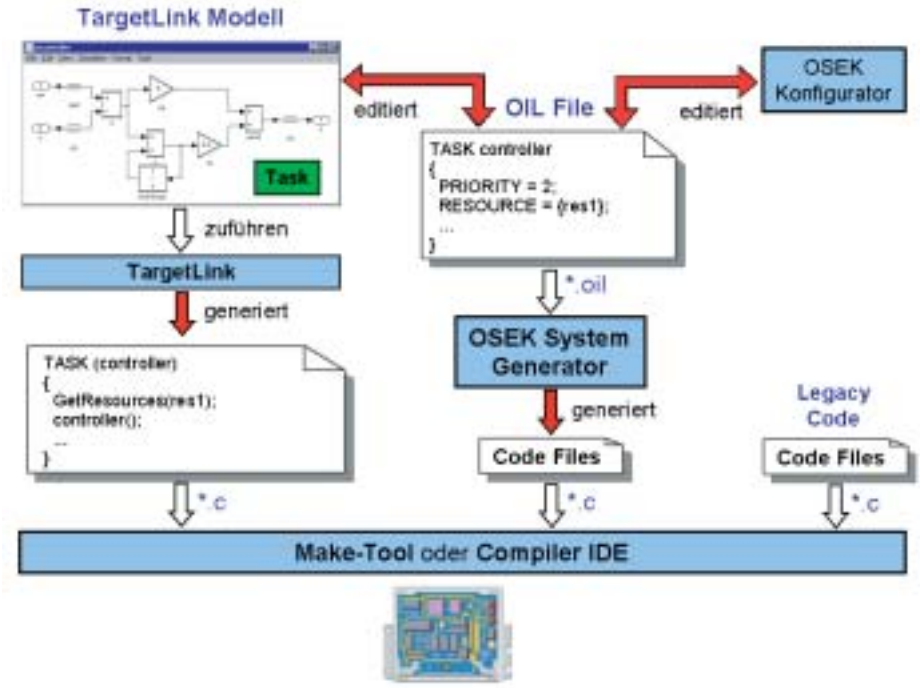


Bild 1: *Integration von blockdiagrammbasiertem Simulationstool, Code-Generator, OSEK System Generator und Compiler in eine durchgängige Toolkette*

	Datei-Format	Datei-Erweiterungen
Blockdiagrammspezifikation --> Cod-Ggenerator	Model *)	*.mdl
Blockdiagrammspezifikation --> OSEK System Generator	OIL	*.oil
Code-Generator --> Make Tool/Compiler Suite	C	*.c, *.h

*) Eingelesen mittels MATLAB API

Tabelle 1: *Austauschformate in der Toolkette MATLAB/Simulink/TargetLink*

Task-Spezifikation

Tasks fassen Software-Einheiten mit gemeinsamen Echtzeitanforderungen wie Timing, Priorität und Unterbrechbarkeit zusammen. In einem Simulink-Modell kann mittels eines Taskblocks ein Subsystem als Task spezifiziert werden. Der Taskblock ist Bestandteil des TargetLink Blocksatzes. In einer Dialogbox kann der Anwender alle Attribute einer Task einstellen. Dazu gehören die Möglichkeit, eine existierende Task auszuwählen oder neue Tasks anzulegen, sowie Priorität, Anzahl der Aktivierungen, Unterbrechbarkeit, benutzte Ressourcen oder zugehörige Events zu spezifizieren. OIL lässt neben Standardattributen auch herstellerspezifische Attribute zu, die im Taskblock ebenfalls einstellbar sind. Ein Beispiel ist die Definition der Stackgröße, die kein OIL-Standardattribut ist. Die in der Task-Dialogbox eingestellten Eigenschaften werden dann später in den zugehörigen Task-Beschreibungsblock in der OIL-Datei übersetzt.

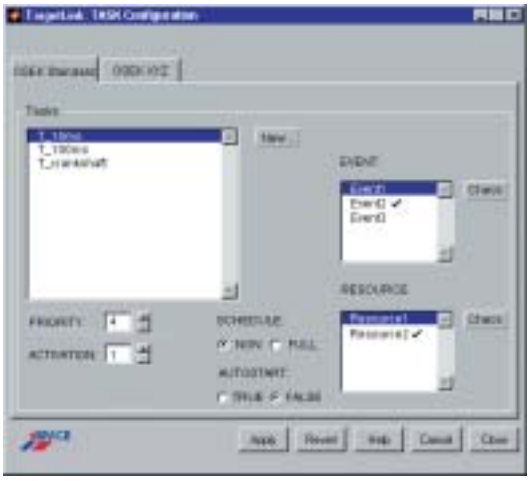


Bild 2: *Task Dialog; mit Eingabemöglichkeiten aller Task-Attribute*

Integration von Serviceroutinen

OSEK OS definiert einheitliche Serviceroutinen für das Scheduling und Mes-saging. So existieren Routinen, die eine Task-Unterbrechung verhindern, andere Tasks aktivieren oder Messages senden. Ein Code-Generator muss erkennen, wann und wo er diese Routinen einsetzen muss. Hierzu werden im Folgenden zwei Beispiele gegeben.

Falls in einem Simulink-Blockdiagramm ein Subsystem über einen Triggereingang aufgerufen wird, kann TargetLink eine Interrupt-Serviceroutine (ISR) mit einem ActivateTask-Befehl generieren, der die Task für das getriggerte Subsystem aufruft. Der Anwender muss dann nur noch die ISR in ein Interrupt-Adressregister eintragen, um den Code auf seinem Prozessor zu integrieren.

Einige OSEK-Konzepte lassen sich nicht mit generischen Simulink-Blöcken beschreiben. TargetLink bietet hier eine erweiterte Bibliothek mit OSEK-Spezialblöcken. Beispielsweise könnte es sein, dass ein getriggertes Subsystem von einer OSEK-Alarmquelle aufgerufen werden soll. TargetLink bietet hierfür einen Counterblock und einen Alarmblock. Der Counterblock wird mit einer Eventquelle verbunden (z.B. mit dem System-'Tick' oder einem Kurbelwellen-Interrupt) und zählt diese Ereignisse. Der Alarmblock vergleicht dann den momentanen Zählerstand des Counterblocks mit einem Schwellwert und ruft nach dessen Erreichen die zugehörige Task auf. Der Code-Generator muss für diesen Fall in der Regel eine Initialisierungstask für den Alarm erzeugen, sowie die richtigen Attribute in die OIL-Datei für Alarm und Counter setzen. Auch in diesem Fall spezifiziert der Anwender nur auf Blockdiagrammebene – TargetLink generiert den zugehörigen Code automatisch.

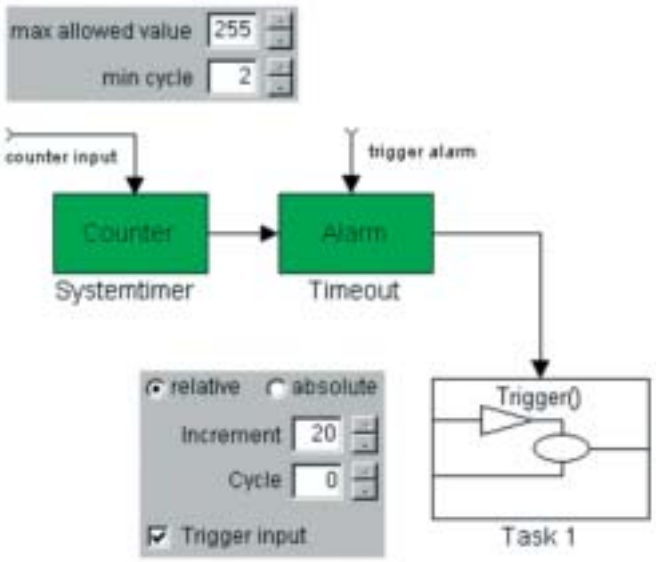


Bild 3: *Alarm- und Counterblöcke zum Einrichten eines OSEK-Alarms*

Betriebssystemoptimierte Code-Generierung

Code-Generatoren mit dem Anspruch, serientauglichen Code zu generieren, nutzen alle verfügbaren Informationen aus Blockdiagrammen und der OIL-Datei, um keinen überflüssigen Code zu generieren. Welche Optimierungspotenziale möglich sind, zeigt ein Beispiel aus der Codierung der Intertask-Kommunikation.

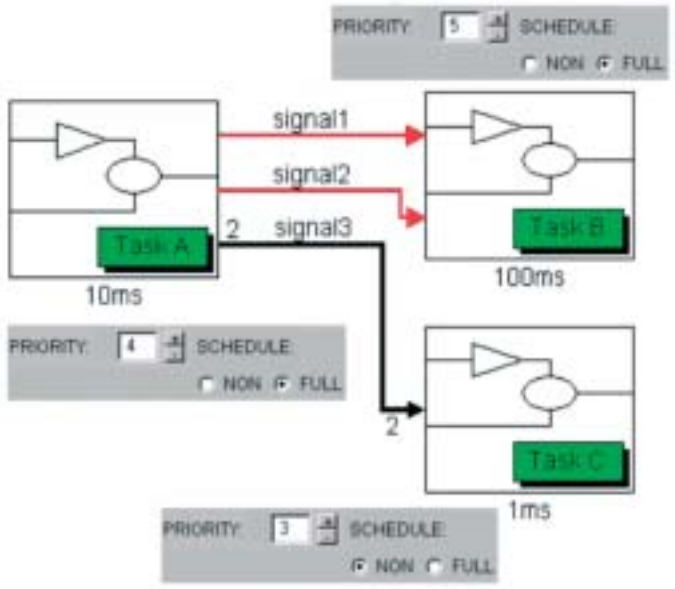


Bild 4: *Beispiele für die Intertask-Kommunikation mit und ohne der Notwendigkeit der Datenkonsistenzsicherung*

In Bild 4 sind drei Tasks dargestellt. Task B hat die höchste Priorität und kann Task A unterbrechen. Task A wiederum berechnet zwei Signale, die in Task B verarbeitet werden. Ohne einen Schutzmechanismus könnte nun Task B genau zu dem Zeitpunkt Task A unterbrechen, in dem bereits ein neuer Wert für Signal 1 berechnet wurde, aber der Wert für Signal 2 noch veraltet oder undefiniert ist. Dann würde Task B mit einem Satz ungültiger Daten rechnen, was leicht zu einem Fehlverhalten der gesamten Regelung führen kann. TargetLink erkennt diesen Zustand im Blockdiagramm und generiert automatisch sicheren Code, in dem keine Daten-

inkonsistenz vorkommen kann. Dieses geschieht beispielsweise in Task A durch Umkopieren von Signal 1 und Signal 2 in lokale Variablen und kurzzeitiges Sperren aller Interrupts. Eine andere Methode wäre, mittels der OSEK-Kommandos 'GetResource' und 'ReleaseResource' in Task A den kritischen Code-Abschnitt vor Unterbrechung zu schützen. Das spart den Code für das Umkopieren, hat aber den Nachteil einer längeren Sperrzeit für höher priorisierte Tasks oder ISRs.
Etwas anders sind die Verhältnisse zwischen Task A und Task C. Auch hier verwendet Task C zwei Signale von Task A in Form eines 2-elementigen Vektors. Task C hat aber eine niedrigere Priorität als Task A und kann diese nicht unterbrechen. Demnach würde TargetLink hier keinen Code für den Datenkonsistenzschutz generieren.
Der Anwender kann die Wahl der Schutzmechanismen für die Datenkonsistenz TargetLink überlassen oder sie mittels einer Dialogbox selber bestimmen.
Eine umfangreiche Beschreibung des Integrationskonzepts für OSEK/VDX kann dem Aufsatz „Connecting Simulink to OSEK: Automatic Code Generation for Real-Time Operating Systems with TargetLink" [2] entnommen werden.

3. MISRA C

Seit 1998 gibt es von der britischen „Motor Industry Software Reliability Associa-tion" (MISRA) einen allgemein zugänglichen Standard für die Verwendung von C in Steuergeräteprojekten bis SIL3. Er wird unter dem Titel „Guidelines for the use of the C language in vehicle based software" geführt, ist aber allgemein unter dem Schlagwort „MISRA C" [14] bekannt. Firmen aus der Automobilindustrie übernehmen den Standard zunehmend als Ersatz oder Ergänzung ihrer internen Standards. Er wird in Kundenprojekten von den Auftraggebern vorgeschrieben, so dass Zulieferer zur Übernahme des Standards praktisch gezwungen werden. Automatische Code-Inspektionstools existieren reichhaltig auf dem Markt und können mit einem gewissen Maß an Zuverlässigkeit Regelverletzungen in einer C-Datei erkennen. Eines der bekannteren Tools mit einem MISRA-C-Modul ist QA C von QA-Systems.
Die Autoren des MISRA-C-Standards haben das verfügbare Wissen aus Literatur, Firmenstandards und der Praxis zusammengetragen. Ziel des Standards ist es, Regeln zur Vermeidung von üblichen Software-Fehlern aufzustellen. Die Fehler stammen aus den folgenden Kategorien:

- Übliche Programmierfehler
- Nichtbeachtung von Lücken in der Sprachendefinition
- Missverständnis der Programmiersprache
- Missverständnis des Compilers
- Compilerfehler und Laufzeitfehler

Die ersten vier der fünf Fehlerklassen beziehen sich auf menschliches Versagen. Dieses wirft in einigen Fällen Probleme auf, wenn von automatisch generiertem Code ebenfalls Regeltreue verlangt wird. Code-Generatoren machen mit einem überzeugenden Grad an Zuverlässigkeit bestimmte Fehler nicht, die für Menschen typisch sind. Der Nutzen bestimmter MISRA-C-Regeln ist dann nicht mehr gegeben. Übrig bleibt dann bestenfalls eine wirkungslose Regel. Andernfalls haben die Regeln einen negativen Effekt auf die Code-Effizienz oder lassen sich durch eine Maschine nicht erfüllen. Im folgenden Abschnitt werden hierzu einige Beispiele erörtert.

Bekannte Probleme mit MISRA C

Wenn einige MISRA-C-Regeln Probleme bei der Code-Generierung bereiten, dann resultiert das hauptsächlich in Verlust an Code-Effizienz, ohne dass sich der erklärte Nutzen der Regel einstellt. Verlust der Code-Effizienz bedeutet einen Mehrbedarf an ROM, RAM, Stack oder Laufzeit.
Bereits Regel 1 des MISRA-C-Standards ist für die Code-Generierung problematisch. Sie ist eine Pflichtregel und verbietet die Verwendung von compilerspezifischen Spracherweiterungen. Es wird keine Erklärung für die Regel gegeben, sondern nur auf einen anderen Standard verwiesen, der die Verwendung einer „standardisierten, strukturierten Sprache" vorschreibt.
Hintergrund der Regel ist die Portabilität des C-Codes. Dieses Problem ist bei Code-Generatoren nicht mehr relevant. Der Input eines Code-Generators sind Blockdiagramme, die in der Regel keine Hardware-Abhängigkeiten kennen. Der generierte C-Code ist nur noch ein Zwischenformat, welches jederzeit neu generiert

werden kann. Seriencode-Generatoren – wie TargetLink – besitzen spezifische Target-Optimierungsmodule, die hocheffizienten Code unter Ausnutzung von compilerspezifischen Spracherweiterungen liefern können. Wenn sich der Prozessor oder Compiler für ein Entwicklungsprojekt ändert, muss lediglich das Target-Optimierungsmodul des Code-Generators ausgetauscht werden. Falls es das nicht gibt, kann immer noch portabler ISO-C-Code erzeugt werden, welcher dann wieder der MISRA-C Regel 1 entspricht. Diese Flexibilität der Code-Generatoren und die Hardware-Unabhängigkeit von Blockdiagrammen machen die Regel 1 überflüssig.

MISRA-C Regel 1 eingehalten: Die Sättigung erfolgt per Plausibilitätscheck im ISO-C-Code.			
Code:			
Laufzeit:	1,1 µs	Prozessor:	Infineon TriCore, 40 MHz
Code-Größe:	48 Bytes	Compiler:	Tasking v. 1.3r1

```
e = REF - POS;
if ((REF >= 0) && (POS < 0) && (e < 0)) {
  e = 32767;
} else {
  if ((REF < 0) && (POS >= 0) && (e >= 0)) {
    e = - 32768;
  }
}
```

MISRA-C Regel 1 nicht eingehalten: Der Castingoperator „`_sat`“ wird verwendet, um Arithmetikoperationen per Hardware zu sättigen.			
Code:			
Laufzeit:	0,3 µs	Prozessor:	Infineon TriCore, 40 MHz
Code-Größe:	12 Bytes	Compiler:	Tasking v. 1.3r1

```
e = (_sat Int16)((_sat Int16)REF-(_sat Int16)POS);
```

Beispiel 1: *Sättigung beim Infineon TriCore. Mit einer compilerspezifischen Spracherweiterung lassen sich 72% Laufzeit und 75% Code-Größe sparen.*

Regel 3 ist verwandt mit der Regel 1. Letztere verbietet Assembler Code, welcher eine compilerspezifische Spracherweiterung ist. Falls dennoch Assembler verwendet werden muss, gibt Regel 3 Einschränkungen vor. Assembler Code und C-Code sollen nicht vermischt werden und Assembler Code soll mit einem C-Funktionsaufruf gekapselt werden. Eine Begründung wird nicht gegeben.
Damit wird die Verwendung von Makros ausgeschlossen. Funktionsaufrufe verbrauchen zusätzliche Rechenzeit, was bei Makros nicht der Fall ist. Durch die Regel 3 wird der Laufzeitvorteil, den man durch die Verwendung von Assembler erzielen kann, wieder zu Nichte gemacht. Dieses gilt insbesondere für häufig aufgerufene Funktionen, wie es beispielsweise Arithmetikoperationen mit doppelter Wortbreite sein können.

MISRA-C Regel 1und 3 eingehalten:			
	Eine 64-Bit Multiplikation wird mittels einer ISO-C-Funktion berechnet.		
Laufzeit:	32,6 µs	Prozessor:	Motorola M68332, 16 MHz
Code-Größe:	32 + 142 (library) Bytes	Compiler:	Microtec v. 4.5r

MISRA-C Regel 1und 3 nicht eingehalten:			
	Eine 64-Bit Multiplikation wird mittels einem Assemblermakro berechnet.		
Laufzeit:	5,5 µs	Prozessor:	Motorola M68332, 16 MHz
Code-Größe:	26 Bytes	Compiler:	Microtec v. 4.5r

Beispiel 2: *Inline-Assemblermakros beim Motorola M68332. Durch die Verwendung der Assemblersprache in einem Makro lassen sich 83% Laufzeit und 85% Code-Größe sparen.*

Regel 28 verbietet die Verwendung der Speicherklasse 'register'. Die Begründung dazu lautet, dass dieses nur eine Empfehlung an den Compiler sei und gute Compiler die richtige Registerbelegung selber finden 'sollten'. Letzteres ist leider nicht immer der Fall. Während der Entwicklung von TargetLink-Optimierungsmodulen sind verschiedene Fälle gefunden worden, wo die Verwendung von 'register' das Anlegen von Variablen im RAM verhindert hat. Das spart Ressourcen in allen drei Disziplinen: Eine RAM-Variable wird weniger erzeugt, es müssen keine Adressen für die Variable im Opcode gespeichert werden und es gibt auch keinen RAM- Zugriff während der Laufzeit.
Nebenbei sei noch bemerkt, dass man mit der Deklaration einer Registervariable den späteren Zugriff mittels Pointer verhindern kann. Geschieht dieses im Code, würde es eine Fehlermeldung während der Kompilierung geben. Das wäre in einigen Fällen eine berechtigte Sicherheitsmaßnahme, welche die Regel 28 nicht erlaubt.

MISRA-C Regel 28 eingehalten:			
	Die Arithmetikoperation wird mittels einem Assemblermakro auf dem Stack ausgeführt.		
Code:			

```
#define AC__I16SUBI16I16_SAT(s1, s2, csatval, r)                    \
{                                                                   \
  Int16 tmp_s1 = s1;                                                \
  Int16 tmp_s2 = s2;                                                \
  r = asm(Int16, " move.w `tmp_s1`,D0",                             \
                 " sub.w  `tmp_s2`,D0",                             \
                 " bvc *+16",                                       \
                 " tst.w D0",                                       \
                 " blt *+8",                                        \
                 " move.w #-"#csatval"-1,D0",                       \
                 " bra *+6",                                        \
                 " move.w #-"#csatval",D0",                         \
}
```

Laufzeit:	3,1 µs	Prozessor:	Motorola M68332, 16 MHz
Code-Größe:	38 Bytes	Compiler:	Microtec v. 4.5r

MISRA-C Regel 28 nicht eingehalten:

Die Arithmetikoperation wird mittels einem Assemblermakro in Prozessorregistern ausgeführt.

Code:

```
#define AC__I16SUBI16I16_SAT(s1, s2, csatval, r)                    \
{                                                                   \
  register Int16 tmp_s1 = s1;                                       \
  register Int16 tmp_s2 = s2;                                       \
  r = asm(Int16, " move.w `tmp_s1`,D0 ",                            \
                 " sub.w `tmp_s2`,D0",                              \
                 " bvc *+16",                                       \
                 " tst.w D0",                                       \
                 " blt *+8",                                        \
                 " move.w #-"#csatval"-1,D0",                       \
                 " bra *+6",                                        \
                 " move.w #-"#csatval",D0",                         \
}
```

Laufzeit:	2,1 µs	Prozessor:	Motorola M68332, 16 MHz
Code-Größe:	30 Bytes	Compiler:	Microtec v. 4.5r

Beispiel 3: *Sättigung beim Motorola M68332. Durch das 'register'-Attribut bei der Deklaration von Zwischenvariablen lassen sich 32% Laufzeit und 21% Code-Größe sparen.*

Regel 37 verbietet die Anwendung von Bit-Operationen auf 'signed integer'-Datentypen. Der Grund ist offensichtlich. Nur zu leicht können solche Operatoren das Vorzeichenbit unerwünscht verändern. Der Fehler wird leicht gemacht und ist schwer zu finden. Um Regel 37 einzuhalten, muss der Software-Entwickler statt eines Bitshift-Right nun eine Division vornehmen. Damit kann ein unerwartet gesetztes Vorzeichenbit nicht mehr in den numerischen Teil der Variable gelangen und somit das Ergebnis verfälschen. Andererseits dauern Divisionen auf üblichen Prozessoren erheblich länger als Bitshifts.
Ein guter Code-Generator führt für sämtliche Signale eine Wertebereichsanalyse durch. Wenn der oben beschriebene Effekt nicht auftreten kann, wird die Bit-Operation verwendet. Auch Software-Entwickler führen Wertebereichsanalysen durch. Anders als ein Code-Generator machen Menschen regelmäßig Fehler dabei und berücksichtigen zukünftige Code-Änderungen nicht. Ein Code-Generator generiert bei jeder Änderung im Blockdiagramm den Code neu und wiederholt die Wertebereichsanalyse. Fehler sind somit per Design ausgeschlossen. Eine Befolgung der Regel 37 hat jetzt nur noch die Folge, dass sich die Laufzeit des generierten Codes erhöht.
Regel 101 verbietet Pointerarithmetik. Die Problematik ist sicher jedem SoftwareExperten (wohl auch aus persönlicher Erfahrung) bekannt. Trotzdem ist die Verwendung von Pointerarithmetik in Suchalgorithmen und Interpolationsroutinen für Steuergerätekennfelder absolut üblich. Bei einigen DSP-Routinen können Laufzeitrestriktionen ohne Pointerarithmetik nicht mehr eingehalten werden.
Anstelle Indexvariablen zu inkrementieren und dann Array-Felder damit zu adressieren, können, mittels Inkrement- oder Dekrementbefehlen auf Pointer angewendet, die gleichen Elemente eines Arrays deutlich effizienter adressiert werden. Manche Prozessoren bieten für diese Operationen spezielle Assembler-Befehle an, die vom Compiler nur dann verwendet werden, wenn der C-Code spezielle C-Codemuster mit Pointerarithmetik enthält.
Von einem Code-Generator kann erwartet werden, dass er keine Fehler in der Pointerarithmetik macht. Wegen der Bedeutung von Kennfeldern in Steuergeräten gehört die Regel 101 mit zu den folgenschwersten Regeln für die Code-Effizienz, wenn sie von einem Code-Generator befolgt werden müsste.

MISRA-C Regel 101 eingehalten: Index-Suchfunktion ohne Pointerarithmetik.			
Laufzeit:	26,1 µs	Prozessor:	Motorola HC12, 8 MHz
Code-Größe:	105 Bytes	Compiler:	Cosmic v. 4.2u

MISRA-C Regel 101 nicht eingehalten: Index-Suchfunktion ohne Pointerarithmetik.			
Laufzeit:	22,9 µs	Prozessor:	Motorola HC12, 8 MHz
Code-Größe:	94 Bytes	Compiler:	Cosmic v. 4.2u

Beispiel 4: *Verwendung von Pointerarithmetik für die Index-Suchfunktion eines Kennfeldes beim Motorola HC12. Mit einem Pointer für die Adressierung des Stützstellenvektors lassen sich 12% Laufzeit und 10% Code-Größe sparen.*

Regel 109 verbietet die Mehrfachverwendung von Variablen. Dieses ist jedoch bei temporären, lokalen Variablen sinnvoll, um den Stackverbrauch zu minimieren. Mittels einer Lebensdaueranalyse stellt der Code-Generator sicher, dass sie in einem Code-Abschnitt nicht mehr gebraucht wird, bevor ihr eine neue Bedeutung zugewiesen wird. Code-Generatoren machen hier keine Fehler. Auch Änderungen im Blockdiagramm stellen keine späteren Fallen dar, weil die Lebensdaueranalyse bei jedem Lauf des Code-Generators wiederholt wird. Regel 50 verbietet den exakten Vergleich von Fließkommavariablen auf Gleichheit oder Ungleichheit. Regel 70 verbietet die Verwendung von rekursiven Funktionsaufrufen. Beide Regeln lassen sich auf Blockdiagrammebene transferieren. So kann Regel 50 immer dann erfüllt werden, wenn der Anwender im Blockdiagramm den 'Relational Operator' Block nicht mit Gleichheit oder Ungleichheit für Fließkommasignale verwendet. Regel 70 wird immer dann eingehalten, wenn der Anwender auf 'Undirected Event Broadcasts' in einem Statechart verzichtet.
Weil nicht alle Anwender von TargetLink nach MISRA C entwickeln müssen, hat dSPACE die Code-Generierung für diese Fälle zugelassen. Anwender, die MISRA-C-Code abzuliefern haben, müssen die beschriebenen Einschränkungen auf Blockdiagrammebene berücksichtigen.
Weitere Regeln des MISRA-C-Standards lassen sich einfach durch die richtige Wahl von Optionen in TargetLink erfüllen. Beispielsweise ist die Begrenzung der Zeichenlänge für Symbole auf 31 signifikante Stellen (Regel 11) im TargetLink- Maindialog einstellbar, ebenso wie die Generierung von reinem ISO-C-Code (Regel 1). Falls #define/#undefine-Direktiven nicht in einem Codeblock auftauchen sollen (Regel 91), kann in der TargetLink-Konfiguration die Variablenklasse LOCAL_MACRO entfernt werden.
Die Kombination der Regeln 43 und 44 ist für generische ISO-C-Code-Generierung nicht erfüllbar. Regel 43 schreibt vor, dass es keine impliziten Typenkonvertierungen im Code geben darf und Regel 44 verbietet überflüssige, explizite Typenkonvertierungen. Beide Regeln sind nur zu erfüllen, wenn die Bitbreite von 'char', 'int' und 'long int' bzw. 'float' und 'double' bekannt sind, was wiederum die Kenntnis des verwendeten Compilers und Prozessors voraussetzt. Im generischen ISO-C-Modus von TargetLink ist dieses nicht bekannt und der Code-Generator muss portablen Code für alle möglichen Bitbreiten generieren. Somit werden tendenziell zu viele Casting-Operatoren im Code erzeugt, die bei spezifischen Compiler/Prozessor-Kombinationen nicht nötig wären.

Eine Statistik

Die oben angeführten Probleme mit dem MISRA-C-Standard und der Code-Generierung sind zahlenmäßig eher gering im Vergleich zu der Gesamtzahl der Regeln. Der Standard beinhaltet insgesamt 127 Regeln. TargetLink von dSPACE erfüllt davon 92 Regeln. Weitere 7 Regeln werden immer befolgt, wenn TargetLink richtig konfiguriert wird. 3 Regeln können eingehalten werden, wenn sich der Anwender auf Blockdiagrammebene einschränkt. 5 Regeln werden von TargetLink größtenteils eingehalten. Lediglich 20 Regeln werden komplett nicht eingehalten. Davon sind 12 geforderte ('required') Regeln und 8 sind empfohlene ('advisory') Regeln. dSPACE kann einige von ihnen in zukünftigen Versionen implementieren. Die Mehrzahl hat aber die oben beschriebenen Nachteile, welche eine Implementierung nicht ratsam erscheinen lässt.

Gesamtzahl der MISRA-C-Regeln: 127
davon:

... voll erfüllt	92
... erfüllt bei richtiger Toolkonfiguration	7
... erfüllt bei richtigem Modellierungsstil	3
... teilweise erfüllt	5
... nicht erfüllt	20

Tabelle 2: *Erfüllungsmatrix von MISRA-C-Regeln in TargetLink*

MISRA erlaubt ausdrücklich Abweichungen vom Standard, wenn diese technisch begründet und dokumentiert sind. Für TargetLink gibt dSPACE ein solches Dokument heraus.
Eine zufriedenstellende Lösung für die Problematik wäre eine Unterscheidung der Regeln nach Gültigkeit für menschliche Programmierer und Code-Generatoren. Das Thema Code-Generierung wurde bereits auf zwei Konferenzen der MISRA besprochen. Allerdings sind derzeit keine Aktivitäten bezüglich einer Änderung des Standards erkennbar.

4. ISO/IEC 15504 und andere Software-Qualitätsstandards

Seitdem es Software-Entwicklung gibt, gibt es das Problem der Qualitätssicherung der Software. Über die Jahre haben sich eine Vielzahl von Software-Qualitätsstandards entwickelt, welche an sich wieder ein Problem darstellen. Nicht jede Firma kann sich nach allen Standards auditieren und beurteilen lassen. Die Ergebnisse der Audits sind untereinander nicht vergleichbar.
Zu Beginn der 90er Jahre hat sich eine Arbeitsgruppe der ISO/IEC gebildet, um auf der Grundlage existierender nationaler oder firmeninterner Qualitätsstandards einen einheitlichen Rahmenstandard zu definieren. Wesentlichen Einfluss auf die Entwicklung des neuen Standards ISO/IEC 15504 hatten CMM, Bootstrap und die ISO 9000 Serie. Die erste Version des Standards ist 1995 erschienen und wurde im Rahmen des europäischen SPiCE-Projektes erprobt. Bis Ende 2001 wurden weltweit ca. 2500 Assessments nach der Norm durchgeführt. Erfahrungen aus den Assessments fließen derzeit in eine Korrektur der Norm ein, die in 2002/2003 zum internationalen Standard erhoben werden soll.
Schon bevor der Standard international verabschiedet ist, fordern die deutschen Automobilhersteller, dass die Steuergeräte-Entwicklung sowie die Entwicklung bedeutender Software-Werkzeuge nach diesem Standard erfolgen soll. Für dSPACE hat sich die Frage gestellt, ob sie dieser Forderung einfach folgen soll, oder alternative Wege zur Software-Qualitätssicherung besser wären. Nach einem kurzen Einblick in die ISO/IEC 15504 folgt ein Vergleich dieser Norm gegen 'CMM for Software', dem heute am häufigsten angewendeten Standard zur Bewertung von Software-Entwicklungsprozessen.

Kurzeinführung ISO/IEC 15504 (SPiCE)

ISO/IEC 15504, im Folgenden SPiCE genannt, hat das Ziel, einen Rahmen für die Bewertung von Software-Entwicklungsprozessen zu geben und soll folgende Zwecke erfüllen:

- Reifegradbestimmung
- Prozessverbesserung

Die Reifegradbestimmung soll einem Auftraggeber eine zuverlässige Basis für die Auswahl von Zulieferern bieten. Das zu bewertende Prozessprofil soll auf bestimmte Geschäftsziele abstimmbar sein. Ergebnisse von Assessments sollen untereinander vergleichbar sein. Aus ihnen sollen sich mögliche Risiken einzelner Zulieferer ableiten können.
Für die Durchführung von Prozessverbesserungen soll ein SPiCE-Assessment die Grundlage bieten, indem es eine vollständige Beschreibung des gegenwärtigen Zustands liefert. Aus ihr sollen sich dann Stärken, Schwächen und Risiken einzelner Prozesse erkennen lassen. Aus den Erkenntnissen können Prioritäten und Maßnahmen zur Prozessverbesserung abgeleitet werden.
Diese Ziele werden von SPiCE durch die Vorgabe eines Prozessreferenzmodells und eines Assessment-Prozesses erreicht. Das Prozessreferenzmodell beschreibt einzelne Arbeitspraktiken, die als unverzichtbar für einen

guten Software-Entwicklungsprozess angesehen werden. Das Modell schreibt nicht einzelne Techniken oder Methoden für einen Prozessschritt vor, sondern legt nur fest, dass bestimmte Prozessschritte existieren müssen und gibt einige Anforderungen an diese vor. Die assessierte Firma oder Organisationseinheit definiert für sich ein kompatibles Prozessmodell, welches den Anforderungen des Prozessreferenzmodells entspricht. Das kompatible Modell ist dann die Grundlage des Assessments. Dessen Ergebnis ist eine Bewertungsmatrix, auch 'Prozessprofil' genannt.
Die Bewertung nach SPiCE erfolgt in zwei Dimensionen: in Prozessen und in Fähigkeiten. Jeder Prozess wird einzeln nach seinen Fähigkeiten bewertet. Ein generisches Prozessreferenzmodell, welches mit SPiCE kompatibel ist, kann der ISO/IEC 12207 [8] entnommen werden. In ihr sind Prozesse in die Prozessgruppen Engineering, Acquirer, Management, Organization und Support aufgeteilt.

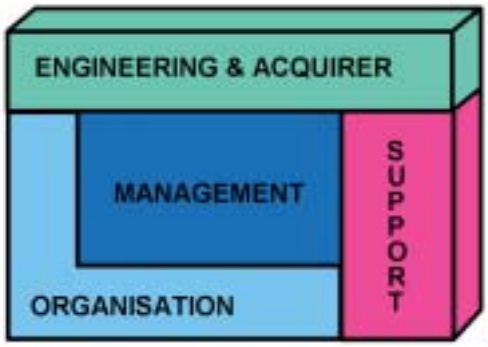


Bild 5: *Prozessmodell nach SPiCE*

Die Engineering Prozessgruppe wiederum besteht aus den Prozessen:

ENG.1	Development
ENG.1.1	System requirements analysis and design
ENG.1.2	Software requirements analysis
ENG.1.3	Software design
ENG.1.4	Software construction
ENG.1.5	Software integration
ENG.1.6	Software testing
ENG.1.7	System integration and testing
ENG.2	System and software maintenance

Bild 6: *Fähigkeitenebenen (Capability Levels) nach SPiCE*

Orthogonal zu der Prozessdimension steht die Dimension der Fähigkeiten (Capability). Nach SPiCE gibt es 6 Ebenen:

0 - Incomplete
1 - Performed
2 - Managed
3 - Established
4 - Predictable
5 - Optimizing

Jede der Ebenen von 1 bis 5 beschreibt einen Satz von Prozessattributen, die zusammengenommen einen bedeutenden Schritt in Richtung eines reiferen Entwicklungsprozesses bedeuten. Ein Assessor 'misst' die Fähigkeiten eines Prozesses mittels sogenannter Indikatoren. Diese können nachvollziehbare Basistätigkeiten sein (z.B. der Gebrauch eines CM-Systems), Eingabedokumente in den Prozess (z.B. Anforderungsspezifikationen) und Ausgabedokumente (z.B. Testreports). Entsprechend den Erkenntnissen, welche der Assessor aus den Indikatoren ziehen kann, vergibt er für jeden Prozess pro Prozessattribut einen Erfüllungsgrad (N: Not, P: Partially, L: Largely, F: Fully). Die ausgefüllte Bewertungsmatrix stellt dann das Prozessprofil dar. Aus ihr kann ein Gesamtreifegrad für die bewertete Organisationseinheit abgeleitet werden.

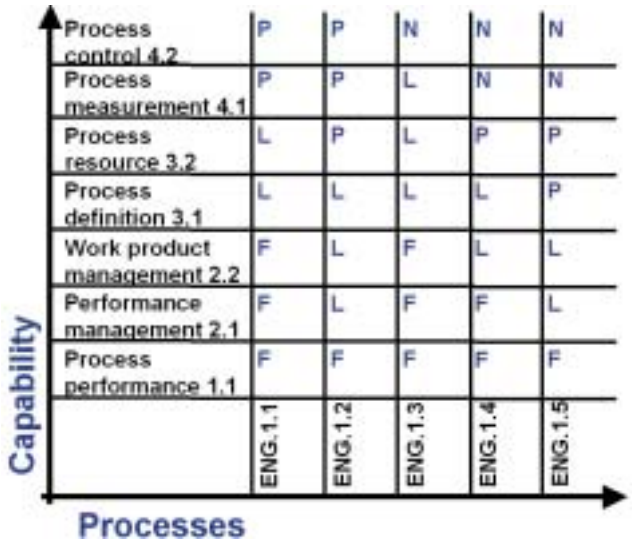


Bild 7: *Einfaches Beispiel für ein Prozessprofil*

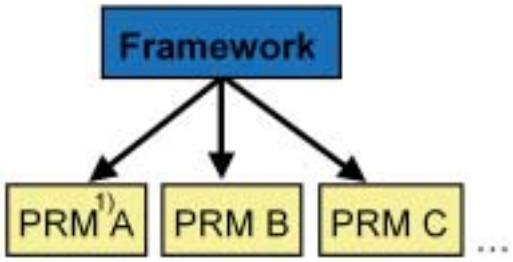
Die beschriebene Assessment-Methode nach SPiCE sieht auf den ersten Blick CMM sehr ähnlich. Insbesondere das Konzept der 'Capability Levels' scheint aus CMM entnommen zu sein. Bei näherer Betrachtung gibt es einige Unterschiede, die bei der Auswahl eines Qualitätsstandards durchaus relevant sein können.

Vergleich ISO/IEC 15504 (SPiCE) mit 'CMM for Software'

Beide Normen weisen in den Details Unterschiede auf. Hier sollen die vier auffälligsten Unterscheidungsmerkmale verglichen werden.

Struktur

Die Struktur der ISO/IEC 15504 (SPiCE) wurde für die endgültige Fassung, welche in 2002/2003 verabschiedet werden soll, nochmals deutlich verändert. Die Prozessdimension ist in die ISO/IEC 12207 verlegt worden. ISO/IEC 15504 (SPiCE) definiert stattdessen den Begriff des Prozessreferenzmodell (PRM) und beschreibt grundlegende Anforderungen daran. Mit der neuen Konstruktion wurde ermöglicht, dass verschiedene Prozessreferenzmodelle für die Bedürfnisse unterschiedlicher Technologiebereiche entwickelt und standardisiert werden können.



1) Process Reference Model

Bild 8: *Struktur von SPiCE*

Neben dem bereits existierenden Software PRM in ISO/IEC 12207 sind weitere geplant bzw. in der Entwicklungsphase:

- System PRM (ISO/IEC 15288)
- Component-Based Development RPM
- Automotive RPM
- Service Management RPM
- ISO 9001 RPM

'CMM for Software' kennt das Konzept der Prozessreferenzmodelle nicht. Es beschreibt genau ein Prozessmodell, nach welchem das Assessment durchgeführt wird. Dieses Modell ist nicht austauschbar oder veränderbar und muss auf alle Technologiedomänen gleichermaßen angewendet werden. Stattdessen gibt es Varianten des CMM für verschiedene Geschäftsdomänen:

- SW-CMM - Capability Maturity Model for Software
- P-CMM - People Capability Maturity Model
- SA-CMM - Software Acquisition Capability Maturity Model
- SE-CMM - Systems Engineering Capability Maturity Model
- IPD-CMM - Integrated Product Development Capability Maturity Model

Derzeit wird im Rahmen des CMMI-Projektes (Capability Maturity Model Integration) versucht, die einzelnen CMMs zusammenzufassen. Das SEI (Software Engineering Institute, Pittsburgh,USA) hat angekündigt, dass sie ihre CMM-Produkte SPiCE-kompatibel machen wird.

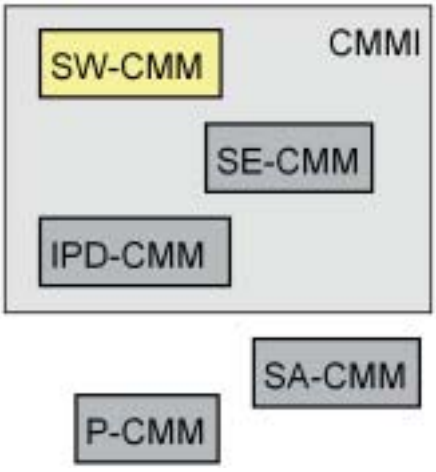


Bild 9: *Struktur von CMM*

Architektur

SPiCE beschreibt ein zweidimensionales, kontinuierliches Referenzmodel. Die zwei Dimensionen sind zum einen die Prozesse und zum anderen die Prozessfähigkeiten. Das Modell wird als kontinuierlich bezeichnet, weil bei jedem Assessment (unabhängig vom angestrebten Capability Level) jeder Prozess betrachtet und bewertet wird.
'CMM for Software' beschreibt ein eindimensionales, gestuftes Referenzmodel. Es gibt nur die Dimension der Prozessfähigkeiten (in CMM *'Maturity Levels'* genannt). Vor dem Assessment wird der angestrebte Maturity Level festgelegt. Mit diesem fest verbunden sind bestimmte Prozesse (in CMM *'key process areas'* genannt), welche während des Assessments betrachtet werden. Die Anzahl der Prozesse steigt demnach mit steigendem Level. Das bedeutet im Umkehrschluss, dass Prozesse oberhalb des angestrebten Levels nicht betrachtet werden. Beispielsweise wird der Software-Test erst ab CMM Level 3 bewertet. Ein CMM Assessment bis zu Level 2 sagt demnach nichts über die Art und Weise aus, wie die betrachtete Organisation ihre Software testet. Sie könnte Software-Produkte ungetestet freigeben und trotzdem CMM Level 2 bestehen. Dieses wäre nach SPiCE nicht möglich. Für Branchen mit sicherheitskritischen Systemen, wie beispielsweise dem Automobilbau oder der Luft- und Raumfahrt, wäre ein CMM Level 2 demnach wenig hilfreich.

Umfang

Das Prozessmodel von SPiCE ist in seiner Form von 1998 in 5 Prozesskategorien mit insgesamt 40 Prozessen aufgeteilt. Eine Untersuchung des SEI aus dem gleichen Jahr hat ergeben, dass 27,5% der SPiCE-Prozesse auf CMM-Prozesse vollständig abgebildet werden können, 50% der SPiCE-Prozesse werden von CMM teilweise abgedeckt, 2,5% sind in CMM nicht vorhanden und 20% der Prozesse in SPiCE wurden für 'CMM for Software' als *'out of scope'* klassifiziert. Das SPiCE-Prozessmodel (zukünftig zu finden in ISO/IEC 12207) ist demnach deutlich umfangreicher.
SPiCE verlangt keine Bewertung aller Prozesse. Einzelne Prozesse können gestrichen werden, wenn sie für die gewünschten Aussagen oder für das verfolgte Geschäftsziel nicht relevant sind. Diese Flexibilität bedeutet, dass SPiCE-Ergebnisprofile nur bei gleicher Wahl der Prozesse vergleichbar sind. 'CMM for Software' kennt keine Einschränkbarkeit des Assessment-Umfangs. Für jeden Maturity Level müssen alle in ihm enthaltenen Prozesse betrachtet werden. Dieses macht CMM-Profile untereinander vergleichbar.

Auditsystem

ISO/IEC 15504 (SPiCE) ist ein internationaler Standard. Assessoren können von verschiedenen Schulungsorganisationen ausgebildet werden und sich beim ESI (European Software Institute, Bilbao, Spanien) registrieren lassen. Es existiert ein offener Markt für Trainings-, Beratungs- und Assessment-Dienstleistungen.
CMM mit seinen Varianten ist ein registriertes Warenzeichen des SEI und wird als Produkt vertrieben. Der Markt für Training-, Beratungs- und Assessment-Dienstleistungen wird vom SEI stark reglementiert. In Europa ist nur das ESI autorisiert, den Einführungskurs für CMM zu geben. Die Durchführung von CMM-Assessments kann vom ESI lediglich koordiniert werden. Ansonsten werden alle Leistungen vom SEI aus den USA erbracht. Dadurch soll ein hoher Qualitätsstandard der Dienstleistungen garantiert werden. Allerdings führt das aber auch zu erhöhten Kosten im Vergleich zu SPiCE.

Zusammenfassung

Neben 'CMM for Software' hat dSPACE auch andere Qualitätsnormen untersucht, u.a. die IEC 61508. Das Ergebnis fiel eindeutig für ISO/IEC 15504 (SPiCE) aus. Die Norm ist auf die Entwicklung eines Code-Generators sehr gut anwendbar. Sie wird zukünftig als Grundlage für den Entwicklungsprozess von TargetLink eingesetzt und von einer unabhängigen Stelle kontrolliert. Im Zusammenspiel mit umfangreichen Produkttests hat sich dSPACE damit einen State-of-the-Art Entwicklungsprozess gegeben, in dem Qualität inhärent eingebaut ist.

Literatur

[1] Hanselmann, H./Kiffmeier, U./Köster, L./Meyer, M.:
"Automatic Generation of Production Quality Code for ECUs"
SAE Technical Paper 99P-12, 1999

[2] Köster, L./Thomsen, T./Stracke, R.:
"Connecting Simulink to OSEK: Automatic Code Generation for Real-Time Operating Systems with TargetLink"
SAE Technical Paper 01PC-117, 2001

[3] Hatton, L.:
"Safer C : Developing Software for High-Integrity and Safety-Critical Systems"
McGraw-Hill, 1995

[4] Paulk, M./Curtis, B./Chrissis, M./Weber, C.:
"Capability Maturity Model for Software (Version 1.1)"
SEI Technical Report, CMU/SEI-93-TR-024, ESC-TR-93-177, 1993

[5] Paulk, M./Weber, C./Garcia, S./Beth, M./Bush, C.M.:
"Key Practices of the Capability Maturity Model SM , Version 1.1"
SEI Technical Report, CMU/SEI-93-TR-025, ESC-TR-93-178, 1993

Aktuelle Normen (Stand Februar 2002)

[6] ISO/IEC 9899:1999/Cor 1:2001, Programming Languages - C

[7] ISO/IEC TR 15504:1998, Information technology – Software Process Assessment

[8] ISO/IEC 12207:1995, Information technology – Software Life Cycle Processes

[9] OSEK/VDX Operating System, Version 2.2, September 2001

[10] OSEK/VDX System Generation, Version 2.3, September 2001

[11] OSEK/VDX Communication, Version 2.2.2, Dezember 2000

[12] (zukünftig) ISO 17356, Interface for Embedded Automotive Application

[13] ASAM MCD 2MC, Version 1.4, März 2000

[14] MISRA Guidelines for the Use of the C Language in Vehicle Based Software, April 1998

[15] MEGMA Standardization of Library Blocks for Graphical Model Exchange, Juli 1999

Links

ISO:	www.iso.ch
OSEK/VDX:	www.osek-vdx.org
ASAM e.V.:	www.asam.de
MISRA:	www.misra.org.uk
ESI, SPiCE:	www.esi.es/Projects/SPICE.html
ISO/IEC JTC1/SC7:	www.sqi.gu.edu.au/sc7/wg10
SEI, CMM:	www.sei.cmu.edu/cmm

Kontakt

Thomas Thomsen
dSPACE GmbH
33100 Paderborn
tthomsen@dspace.de

Produktmanager TargetLink
Technologiepark 25
Deutschland
http://www.dspace.de