Accepted for PEARL'01, http://www.real-time.de/real-time/prog/p2001cfp.html

# Echtzeitanforderungen bei der Integration von IEC 61131-3 Funktionsbausteinen und UML-RT Capsules

Torsten Heverhagen, Rudolf Tracht

Universität Essen, FB12, Automatisierungstechnik, Schützenbahn 70, D-45276 Essen
Torsten.Heverhagen@uni-essen.de, Rudolf.Tracht@uni-essen.de

**Überblick.** Mit Hilfe von Funktionsbausteinadaptern (FBAs) kann die Kommunikation zwischen Funktionsbausteinen der IEC 61131-3 und Capsules der Unified Modeling Language (UML) in einer formalen Sprache spezifiziert werden. Diese Spezifikation wird während der Entwurfsphase eines Systemes erstellt und ist hardware-unabhängig. Nachrichten, die von Capsules an Funktionsbausteine geschickt werden sollen, werden dabei in zeitabhängige Belegungen von Eingangsvariablen der Funktionsbausteine umgesetzt. Belegungen der Ausgangsvariablen von Funktionsbausteinen werden in Nachrichten umgewandelt, die an Capsules gesendet werden können. In diesem Artikel wird gezeigt, dass man aus der Verwendung von FBAs frühzeitig Synchronisationsprobleme zwischen Funktionsbausteinen und Capsules erkennen und Echtzeitanforderungen an die für die Kommunikation zuständige Software und Hardware ableiten kann.

## 1. Motivation und Einleitung

Speicherprogrammierbare Steuerungen (SPSen) haben im Bereich der Automatisierungstechnik eine weite Verbreitung gefunden. Die Einsatzgebiete erstrecken sich von der Produktautomatisierung (z.B. in Werkzeugmaschinen) über die Fertigungs- und Gebäudeautomatisierung bis hin zur Prozessautomatisierung. Wegen ihrer hohen Zuverlässigkeit, guten Echtzeiteigenschaften und einfachen Handhabung werden sie vor allem in der operativen Ebene der Unternehmenshierarchie [1] eingesetzt.

Auf der taktischen Ebene der Unternehmenshierarchie findet man hauptsächlich Standard-Personalcomputer (PCs) oder Workstations, deren Aufgaben in den Bereichen Betriebsdatenerfassung, Produktionsplanung, Produktionssteuerung und Visualisierung liegen. Echtzeitfähigkeit wird von den Computersystemen dieser Ebene nicht gefordert. Eine Kommunikation zwischen taktischer und operativer Ebene darf nicht die Echtzeitfähigkeit der in der operativen Ebene eingesetzten SPSen beeinflussen. Das gilt gleichermaßen für die Kommunikation zwischen dem World Wide Web (WWW) und der operativen Ebene.

Aber auch in der operativen Ebene werden parallel zu SPSen zunehmend auf PC-Technik basierende Automatisierungsgeräte eingesetzt, die ähnlichen Echtzeitanforderungen wie die SPSen unterliegen. Hier kann man deshalb auch eine echtzeitfähige

Kommunikation zwischen PC-basierten Automatisierungsgeräten und SPSen erlauben.

Wir können deshalb allgemein feststellen, dass die Kommunikationsbeziehungen zwischen SPS- und PC-basierten Computersystemen sehr vielfältig sind und auch einer besonderen Beachtung ihrer Echtzeiteigenschaften bedürfen. In diesem Artikel untersuchen wir diese Echtzeiteigenschaften auf der Ebene des Softwareentwurfes näher.

Die Programmierung von SPSen erfolgt hauptsächlich durch Sprachen der IEC 61131-3. In der Entwurfsphase werden in erster Linie die Ablaufsprache und Funktionsbausteine verwendet. Dem Konzept einer Softwarekomponente entspricht dabei der Funktionsbaustein.

PC-basierte Automatisierungsgeräte werden zunehmend mittels objektorientierter Sprachen wie C++ oder Java programmiert. In der Entwurfsphase wird zumeist die UML eingesetzt. Dem Konzept einer Softwarekomponente entspricht hier die Klasse. Im Bereich der echtzeitfähigen objektorientierten Softwareentwicklung werden häufig spezielle Klassen zur Modellierung aktiver Objekte eingesetzt – sogenannte Capsules [5].

In [7] haben wir Funktionsbausteinadapter (FBAs) vorgestellt, die es bereits in der Entwurfsphase eines Systems erlauben, Kommunikationsbeziehungen zwischen Capsules und Funktionsbausteinen zu modellieren. Mit FBAs werden Funktionsbausteine so ummantelt, dass sie aus Sicht der UML wie Capsules angesprochen werden können. Aus der Sicht der IEC 61131-3 kann ein FBA wie ein Funktionsbaustein behandelt werden. Es ist dabei unerheblich, ob das Capsule eine Softwarekomponente modelliert, die sich in der operativen Ebene, in der taktischen Ebene oder im WWW befindet. Der Funktionsbaustein könnte sich in einer "klassischen" SPS oder in einer Soft-SPS auf einem PC oder Mikrocontroller befinden. In [8] wurden zwei Realisierungen eines FBA gegenübergestellt, wobei in beiden Fällen das Capsule auf einem Industrie-PC und der Funktionsbaustein auf einer SPS (SIMATIC-S7) liefen, aber unterschiedliche Kommunikationskanäle verwendet wurden (PROFIBUS und Direktverdrahtung von digitalen Ein-/Ausgängen).

In den Abschnitten 2 und 3 wird zunächst ein FBA in Form eines konkreten Beispieles eingeführt. Dieses Beispiel wird in den darauf folgenden Abschnitten zur genaueren Erläuterung der dort vorgestellten Konzepte verwendet. Im Abschnitt 4 wird die Behandlung eines Synchronisationsproblemes erläutert, das aus der Nebenläufigkeit von Capsule und Funktionsbaustein resultiert. Abschnitt 5 beschreibt, wie man zeitliche Anforderungen an die unter dem FBA liegende Software und Hardware ableiten kann. Mit Abschnitt 6 schließen wir diesen Artikel durch eine Zusammenfassung ab.

## 2. Beispiel für eine Kommunikationsbeziehung

In diesem Abschnitt führen wir beispielhaft einen Funktionsbausteinadapter ein, den wir *MyFBA* nennen. Dazu beschreiben wir in 2.1 den Funktionsbaustein *MyFB*, der später mit einem Capsule *MyCapsule* kommunizieren soll. In 2.2 stellen wir das Capsule *MyCapsule* vor. Abschnitt 2.3 enthält schließlich die Spezifikation des FBA *MyFBA*.

## 2.1. Der Funktionsbaustein MyFB

Abbildung 1 zeigt *MyFB* in Form eines Diagrammes. Die Definition der benutzerdefinierten Datentypen *In_Data* und *Out_Data* sind in Abbildung 2 angegeben.

Die für die Eingangs- und Ausgangsvariablen erlaubten Belegungen sind in Abbildung 3 in Form eines Zeitdiagrammes beschrieben. Den Eingangsvariablen *A*, *B* und *C* ist das Präfix *MyFBA* vorangestellt, um zu zeigen, dass diese Variablen vom FBA gesetzt werden, der später vorgestellt wird.

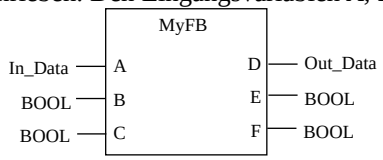


**Abbildung 1. Der Funktionsbaustein**

*MyFB* kann eine Nachricht empfangen und eine Nachricht senden. Der Empfang der *Nachricht B* beginnt zum Zeitpunkt $t_2$ mit einer positiven Flanke in *B*. Dabei übernimmt *MyFB* nacheinander zwei Werte in *A* und bestätigt die Übernahme mit *F*. Bei der zweiten Bestätigung in *F* stellt *MyFB* in *D* Daten für den Kommunikationspartner bereit ($t_6$). Deren Übernahme wird mit *B* bestätigt ($t_7$). Zum Zeitpunkt $t_8$ ist die Übermittlung der *Nachricht B* beendet. Die *Nachricht E* sendet *MyFB*, indem er zunächst Daten in *D* zur Verfügung stellt und anschließend in *E* eine positive Flanke ausgibt ($t_{10}$). Die Übernahme der Daten wird durch ein positive Flanke in *C* bestätigt. Zum Zeitpunkt $t_{12}$ ist die Datenübertragung beendet. Beim Übermitteln

```
TYPE Out_Data
STRUCT
  var1: INT;
  var2: INT;
END_STRUCT
END_TYPE

TYPE In_Data
  INT
END_TYPE
```

**Abbildung 2. Datentypdefinitionen**

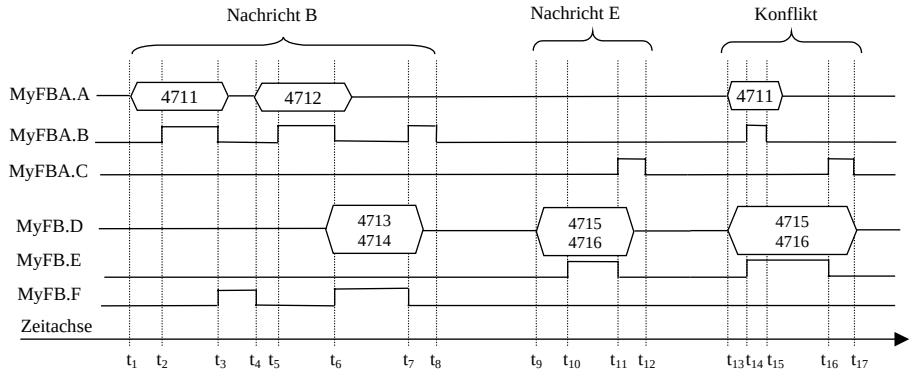


**Abbildung 3. Kommunikation zwischen MyFBA und MyFB**

der *Nachricht B* darf *E* nicht verwendet werden. Das Gleiche gilt für *B* beim Übermitteln der *Nachricht E*. Die *Nachricht E* hat höhere Priorität, so dass im Konfliktfall ($t_{14}$) der Sender der *Nachricht B* zurücknimmt.

Das in Abbildung 3 vorgestellte Verhalten soll im weiteren auch als das FB-Protokoll bezeichnet werden. Das mit dem Capsule verbundene Protokoll wird im nächsten Abschnitt vorgestellt.

## 2.2. Das Capsule MyCapsule

In Abbildung 4 ist ein Klassendiagramm gegeben, das die für die Kommunikation mit *MyCapsule* notwendigen Elemente enthält. Eine ausführliche Erklärung der Stereotypen <<Capsule>> und <<Protocol>> findet man in [5] und [6].

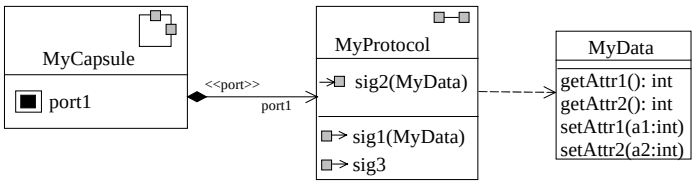


**Abbildung 4. Die Klassen MyCapsule, MyProtocol und MyData**

*MyCapsule* enthält einen Port *port1*, der das Protokoll *MyProtocol* implementiert. Über *port1* kann *MyCapsule* die Nachrichten *sig1* und *sig3* senden und die Nachricht *sig2* empfangen. Mit den Nachrichten *sig1* und *sig2* werden Objekte der Klasse *MyData* versendet bzw. empfangen. Die Klasse *MyData* enthält zwei Attribute vom Typ *int*, auf die über Operationen der Klasse zugegriffen werden kann. Der Kommunikationspartner von *MyCapsule* muss einen Port enthalten, der die konjugierte Rolle von *MyProtocol* implementiert. Das heißt, dass über diesen Port *sig2* gesendet und *sig1* und *sig3* empfangen werden können. Mögliche Kommunikationssequenzen kann man aus dem Statechart in Abbildung 5 ableiten. Der Zustand *free* bedeutet, dass der Kommunikationskanal frei ist. In diesem Zustand können die Nachrichten *sig1* und *sig2* gesendet werden. Nach dem Senden von *sig1* ist der Kommunikationskanal erst wieder frei, wenn *sig2* und *sig3* in dieser Reihenfolge gesendet wurden. Das Senden von *sig2* im Zustand *free* erfordert *sig3*, um wieder in *free* zurückzukehren.

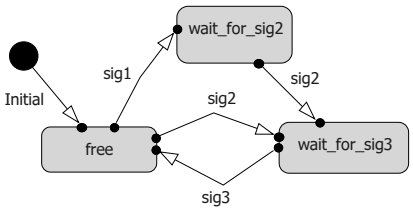


**Abbildung 5. Statechart zu MyProtocol**

Ähnlich wie in Zeitdiagrammen kann man Kommunikationssequenzen in der UML mit Hilfe von Sequenzdiagrammen darstellen. In Abbildung 6 ist als Kommunikationspartner *MyFBA* dargestellt, der im nächsten Abschnitt vorgestellt wird. Mit *sig1* wird *myData1{attr1=4711, attr2=4712}* übertragen. Sig2 enthält *myData2{attr1=4713, attr2=4714}* bzw. *myData3{attr1=4715, attr2=4716}*. Sollten die Kommunikationspartner gleichzeitig versuchen, den Kanal zu belegen, dann entscheiden Prioritäten, die man den Nachrichten zuordnen kann, über den Gewinner. In *MyProtocol* hat *sig2* die höchste, *sig1* die zweithöchste und *sig3* die niedrigste Priorität.

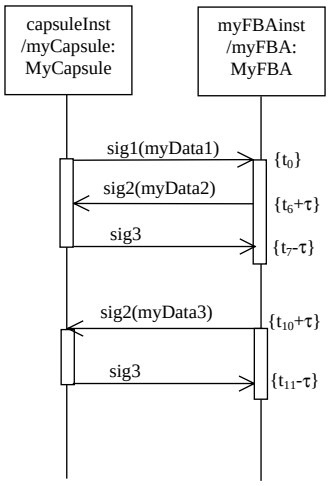


**Abbildung 6. Sequenzdiagramm**

Man erkennt bereits die Parallelen zwischen dem hier vorgestellten Protokoll und dem FB-Protokoll.

Eine formale Abbildung beider Protokolle aufeinander liefern die nächsten Abschnitte.

## 2.3. Der Funktionsbausteinadapter MyFBA

```
FUNCTION_BLOCK_ADAPTER MyFBA
  FB_Variables
    VAR_IN
      D: Out_Data;
      E, F: BOOL;
    END_VAR
    VAR_OUT
      A: In_Data;
      B, C: BOOL;
    END_VAR
  END_FB_Variables

  Capsule_Ports
    ~port1: MyProtocol;
  END_Capsule_Ports

  Signal_Mapping
    No_Signal: (NOT B) & (NOT E);
    ~port1.sig1 raises FBSignal(B);
    FBSignal(E) raises ~port1.sig2;
  END_Signal_Mapping
```

**Abbildung 7. Struktureller Teil des FBA**

Ein FBA besitzt einen strukturellen und einen operationalen Teil. Der strukturelle Teil von MyFBA ist in Abbildung 7 in textueller Form gegeben. Die Schreibweise ist an die aus der IEC 61131-3 bekannten angelehnt. Es müssen die Ein- und Ausgabevariablen zum Funktionsbaustein (FB) und die Ports zu den Capsules deklariert werden. (Ein FBA muss nicht zwangsläufig nur einen FB mit einem Capsule verbinden.) Im FBA können die Typen aus der IEC 61131-3 gleichermaßen wie die UML-Typen verwendet werden. Beginnt ein Portname mit dem Zeichen "~", dann implementiert das Port die konjugierte Rolle des Protokolls. Man kann also *port1* von *MyCapsule* mit *~port1* von *MyFBA* verbinden. Genauso kann man die Ausgangsvariablen von *MyFB* mit den Eingangsvariablen von *MyFBA* und umgekehrt verknüpfen.

Die Abbildung der Nachrichten beider Protokolle aufeinander (*Signal_Mapping*) ist sehr wichtig für die Funktionsweise des FBA. Es muss der Ruhezustand im Funktionsbausteinprotokoll definiert sein (*No_Signal*) und die Startbedingungen für eine Nachrichtenübermittlung aus dem Ruhezustand heraus. Die Prioritäten in beiden Protokollen müssen übereinstimmen. *MyFBA* empfängt *sig1* und wandelt es in *Nachricht B* (*FBSignal(B)*) für *MyFB* um. *MyFBA* empfängt *Nachricht E* (*FBSignal(E)*) und wandelt sie in *sig2* um. Wie das geschieht und welche Aktionen dazu ausgeführt werden müssen beschreibt der operationale Teil des FBA.

Abbildung 8 zeigt den operationalen Teil des FBA. Dieser Teil besteht aus zwei Operationen. *On_UMLSignal(s1: ~port.sig1)* beschreibt die Aktionen zum Übermitteln von *Nachricht B* an *MyFB*. Die Klausel *On_Exception* ist optional und wird beim unvorhergesehenen Verlassen der Operation ausgeführt. *On_FBSignal(E)* wandelt *Nachricht E* in Nachricht *sig2* um. Nach der Klausel *Signals* können Variablen deklariert werden, die als UML-Nachrichten versendet werden können. Zwischen *Begin* und *END* sind Anweisungen wie *delay*, *waitFor*, *sendSync*, *sendAsync* sowie Zuweisungen zu Variablen erlaubt. Variablen vom Typ einer UML-Nachricht haben die

gleichen Eigenschaften (Attribute, Operationen) wie die der assoziierten Datenklasse (siehe auch Abbildung 4). Das Verhalten von Operationen dieser Klassen kann in UML durch OCL-Constraints in Form von Vor- und Nachbedingungen beschrieben werden [3]. Eine genauere Erläuterung der beiden Operationen von *MyFBA* erfolgt in den nächsten Abschnitten.

```
FBA_Operations
 On_UMLSignal (s1: ~port1.sig1)
  Signals
    s2: ~port1.sig2;
    s3: ~port1.sig3;
  Begin
    A := s1.getAttr1();
    B := True;
    waitFor( F, T#50ms );
    B := False;
    waitFor( F = False, T#50ms );
    A := s1.getAttr2();
    B := True;
    waitFor( F, T#50ms );
    B := False;
    s2.setAttr1( D.var1 );
    s2.setAttr2( D.var2 );
    sendSync( s2, s3, T#3s );
    B := True;
    delay( T#2ms );
    B := False;
  END
  On_Exception
    Begin
      B := False;
      A := 0;
    END
 END_On_UMLSignal

 On_FBSignal (E)
  Signals
    s1: ~port1.sig2;
    s2: ~port1.sig3;
  Begin
    s1.setAttr1( D.var1 );
    s1.setAttr2( D.var2 );
    sendSync( s1, s2, T#3s );
    C := True;
    delay( T#2ms );
    C := False;
  END
 END_On_FBSignal
END_FUNCTION_BLOCK_ADAPTER
```

**Abbildung 8. Operationaler Teil des FBA**

## 3. Das Zeitverhalten von MyFBA

Um das Zeitverhalten von MyFBA zu kennzeichnen, sollen hier die Zeitmarken aus Abbildung 3 und Abbildung 6 verwendet werden.

*Sig1* trifft bei $t_0$ am Port des FBA ein. Damit beginnt die Abarbeitung der entsprechenden Operation für *sig1*. Nach dem Setzen von *A* befindet sich der FBA zwischen $t_1$ und $t_2$. Zu $t_2$ wird *B* auf *True* gesetzt. Ab diesem Zeitpunkt wartet der FBA mit der Anweisung *waitFor* auf die Bedingung *F* für maximal 50 ms. Diese Deadline muss aus der Spezifikation von *MyFB* bekannt sein. Zu $t_3$ wird *B* zurückgesetzt und auf das Zurücksetzen von *F* gewartet. Dann wird *A* und *B* erneut gesetzt und ab $t_5$ wieder auf die Bedingung *F* gewartet. Bei $t_6$ wird *B* zurückgesetzt und die Attribute von *s2* werden entsprechend der Werte in *D* gesetzt. Zum Zeitpunkt $t_6+\tau$ wird mit *sendSync* die Nachricht *s2* an *MyCapsule* gesendet und für maximal 3 s auf die Nachricht *s3* von *MyCapsule* gewartet. Diese Deadline muss aus der Spezifikation von *MyCapsule* oder aus dem FB-Protokoll bekannt sein. Abgeschlossen wird die Operation, indem mit Hilfe der Anweisung *delay* für 2 ms *B* auf *True* gesetzt wird. Diese Zeit muss der Spezifikation von *MyFB* entnommen werden.

Bei Eintreffen der *Nachricht E* von *MyFB* wird die Operation *On_FBSignal(E)* ausgeführt. Zunächst werden die Attribute von *s1* entsprechend *D* belegt und zum Zeitpunkt $t_{10}+\tau$ die Nachricht *s1* an *MyCapsule* geschickt. Dann wird für maximal 3 s auf das Eintreffen der Nachricht *s2* gewartet. Abgeschlossen wird die Operation mit dem Setzen von *C* auf *True* für 2 ms. Diese Zeit muss ebenfalls der Spezifikation von *MyFB* entnommen werden.

Bisher haben wir noch nicht den Konfliktfall aus Abbildung 3 berücksichtigt. Der Zustand zum Zeitpunkt $t_{14}$ kann im schlechtesten Fall bis zum Erreichen der ersten *waitFor* Anweisung in *On_UMLSignal(...)* eintreten. Ist das der Fall, muss die Abarbeitung dieser Operation abgebrochen werden. Dabei werden die Anweisungen nach der Klausel *On_Exception* ausgeführt. (Diese Anweisungen werden auch beim Erreichen einer Deadline ausgeführt.)

Der gerade erwähnte Konfliktfall führt zwangsläufig zum Problem der Synchronisation innerhalb eines FBA, das im Abschnitt 4 behandelt wird. Die bisher vorgestellten Konzepte und die FBA-Sprache aus Abbildung 7 und Abbildung 8 sind die Mittel, mit denen man beim Entwurf eines FBA arbeiten muss. Die in den weiteren Abschnitten folgenden Erläuterungen sollen einem tieferen Verständnis von FBAs dienen und zu einer möglichst großen Automatisierung bei der Implementierung von FBAs führen.

## 4. Synchronisation innerhalb von Funktionsbausteinadaptern

Wie wir bereits in Abschnitt 1 ausgeführt haben, können sich Capsules und Funktionsbausteine, die von FBAs verbunden werden, auf unterschiedlichen Computersystemen, die durch ein Netzwerk verbunden sind, befinden (siehe auch [8]). Das bedeutet, dass ein FBA Rechnergrenzen überschreiten kann. Selbst wenn keine Rechnergrenzen überschritten werden, beinhaltet ein FBA im allgemeinen Fall zwei Prozesse, die synchronisiert werden müssen. Wir bezeichnen den ersten als F-Prozess,

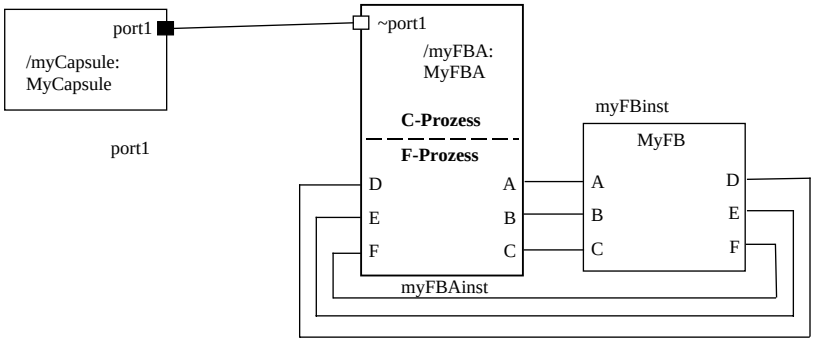


**Abbildung 9. Darstellung von C-Prozess und F-Prozess in MyFBA**

der für die Belegung der FB-Variablen zuständig ist, und den zweiten als C-Prozess, der das Senden und Empfangen von Nachrichten über Ports übernimmt. In Abbildung 9 ist *MyFBA* in graphischer Form in die beiden Prozesse aufgeteilt dargestellt. Das Diagramm in Abbildung 9 zeigt weiterhin, wie *MyFBA* mit *MyFB* in einem Funkti-

onsblockdiagramm nach IEC 61131-3 und mit *MyCapsule* in einem Strukturdiagramm nach [5] verbunden werden kann.

Zur Unterstützung bei der Erläuterung der FBA-internen Arbeitsweise soll das Statechart von *MyFBA* herangezogen werden (Abbildung 10). Um das Statechart etwas komprimierter darstellen zu können, haben wir die Namen der Zustände (*S1* bis *S25*) neben die Zustände in Kursivschrift eingezeichnet. In den Zuständen stehen Freitextinformationen, die einen Zustand näher kennzeichnen sollen. In den vollständig Weiß unterlegten Zuständen ist der F-Prozess aktiv und der C-Prozess wartet auf eine Nachricht vom F-Prozess. In den vollständig Grau unterlegten Zuständen ist es umgekehrt. In den Zuständen *S1* und *S2* sind beide Prozesse nicht synchronisiert. Sie sind deshalb mit einem grau-weißem Muster unterlegt. Die Zustände, die einen grau-weißen Gradienten aufweisen, markieren eine Kommunikation zwischen den beiden Prozessen, bei der gleichzeitig die Aktivität von einem Prozess zum anderen weitergegeben wird.

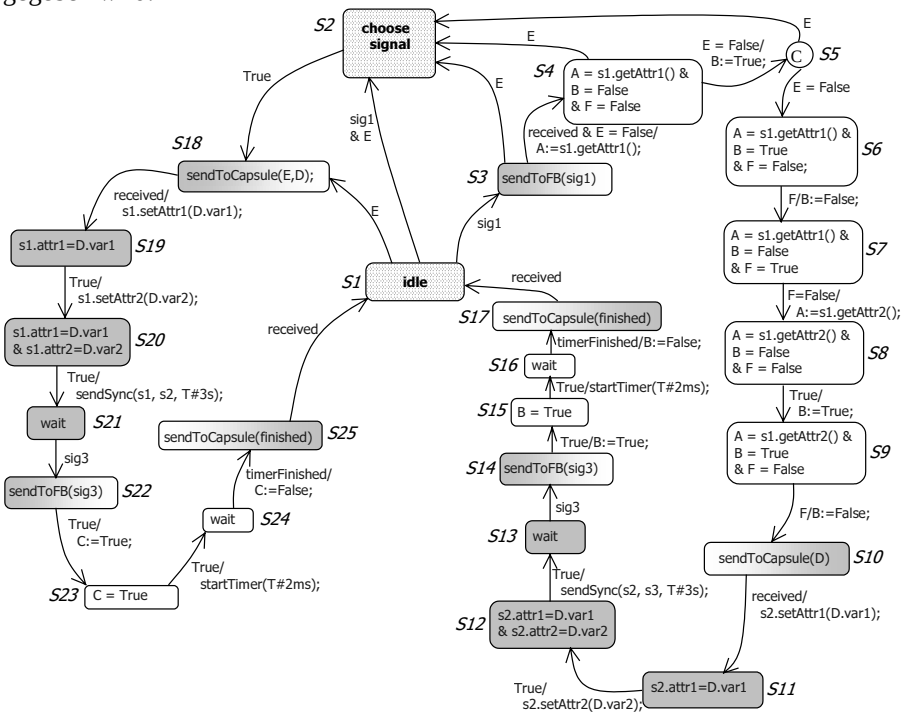


**Abbildung 10. Statechart von MyFBA**

Im Ruhezustand befindet sich *MyFBA* in *S1*. In diesem Zustand wartet der C-Prozess auf Nachrichten von *MyCapsule* oder auf eine Nachricht vom F-Prozess. Der F-Prozess wartet seinerseits auf Nachrichten von *MyFB* oder vom C-Prozess. Sollten beide Prozesse gleichzeitig Nachrichten von *MyFB* bzw. *MyCapsule* empfangen, geht *MyFBA* in Zustand *S2* über. Dieser Zustand ist dadurch gekennzeichnet, dass die beiden Prozesse unterschiedliche Nachrichten bearbeiten möchten. Dieser Konflikt wird bei der prozessinternen Kommunikation erkannt und mit Hilfe der Prioritätsvergabe gelöst, die in *MyProtocol* (Abschnitt 2.2) vorgenommen wurde. In *MyFBA* wird

deshalb in den Zustand *S18* übergegangen. In diesen Zustand wird aus *S1* direkt übergegangen, wenn der F-Prozess die *Nachricht E* erhält, während der C-Prozess weiter auf Nachrichten wartet.

Im Zustand *S18* werden dem C-Prozess vom F-Prozess die notwendigen Informationen (Art der Nachricht und Daten) übermittelt. Der F-Prozess wartet danach auf eine Antwort vom C-Prozess. Die Zustandsfolge *S18* bis *S25* repräsentiert die fehlerfreie Abarbeitung der Operation *On_FBSignal(E)*. Da *Nachricht E* die höchste Priorität besitzt, kann diese Operation nicht durch eine höherpriore Nachricht unterbrochen werden. Nach der Übermittlung der Nachricht *finished* in *S25* warten beide Prozesse wieder auf Nachrichten von außerhalb.

Erkennt der C-Prozess im Zustand *S1* die Nachricht *sig1*, während der F-Prozess weiter auf Nachrichten wartet, wird im konflikt- und fehlerfreien Fall die Zustandsfolge *S3* bis *S17* abgearbeitet. Ab dem Zustand *S6* ist laut Definition (Abschnitt 2.1) die *Nachricht E* nicht mehr erlaubt. Bis *S5* kann *MyFB* jederzeit damit beginnen, *Nachricht E* zu senden. In diesem Fall führt *MyFBA* die Aktionen unter *On_Exception* aus und geht in *S2* über.

Im Statechart wurden aus Gründen der Übersichtlichkeit die Transitionen und ein Zustand weggelassen, die beim Erreichen einer Deadline ausgeführt werden. Neben *S1* und *S2* gäbe es dann noch einen weiteren Timeout-Zustand, in dem die beiden Prozesse nicht synchronisiert sind. Bei Erreichen einer Deadline würden dann aus den Zuständen *S2, S3, S6, S7, S9, S10, S13, S17, S18, S21* und *S25* Transitionen in den Timeout-Zustand übergehen. In diesem Zustand würde so lange verblieben, bis beiden Prozessen der Fehlerfall bekannt ist. Deadlines können aufgrund eines FBA-internen oder FBA-externen Kommunikationsfehlers erreicht werden.

Nachdem wir in diesem Abschnitt das Verhalten eines FBAs auf einer logischen Ebene betrachtet haben, sollen daraus im nächsten Abschnitt Anforderungen an die Implementierungsphase abgeleitet werden. Soweit es möglich ist, möchten wir uns dabei weiterhin an die Modellierungsmittel halten, die uns von der UML zur Verfügung gestellt werden.

## 5. Quality of Service

Unter Quality of Service (QoS) versteht man in der UML die quantitative Modellierung von Ressourcen wie Zeit, Speicher oder Prozessorleistung, die von UML-Elementen benötigt werden. Wir möchten uns in diesem Artikel lediglich auf die Ressource Zeit beschränken.

Um eine maximale Zeit angeben zu können, die für die Übersetzungstätigkeit eines FBA benötigt wird, muss man die Abarbeitung einer FBA-Operation unter den ungünstigsten Bedingungen analysieren, bei denen noch keine Deadline überschritten wird. Allgemein setzt sich die maximale Abarbeitungszeit einer FBA-Operation aus den folgenden Bestandteilen zusammen:

a) Summe der Deadlines aller waitFor-Anweisungen,
b) Summe der Deadlines aller sendSync-Anweisungen,
c) Summe der Zeiten aller delay-Anweisungen,
d) Summe der Ausführungszeiten aller ":="-Anweisungen,
e) Summe der Ausführungszeiten von aufgerufenen Operationen einer Datenklasse,

f) Summe der Deadlines für die in einer Operation notwendige FBA-interne Kommunikation.

Die unter a) bis c) aufgeführten Zeiten lassen sich direkt aus den FBA-Operationen (Abbildung 8) ablesen. Die Ausführungszeit einer Zuweisung unter d) entspricht der Zeit für das Kopieren des Inhaltes eines prozess-internen Speicherbereiches. Diese Zeit ist von der Implementierungsumgebung (Programmiersprache, Plattform) des Prozesses abhängig. Die Ausführungszeit einer Operation einer UML-Klasse unter e) kann ebenfalls erst ermittelt werden, wenn deren Implementierungsumgebung festgelegt wurde. Mit Hilfe des in [2] vorgestellten QoS-Frameworks kann man aber schon in der Entwurfsphase Vorgaben für die Implementierung definieren (z.B. *MyData::setAttr1(a1:int)* ← *{qosDelay ="10 us"}*). Die unter f) gesuchten Zeiten sind allgemein wesentlich größer als die unter d) und e). Sie hängen vom FBA-internen Kommunikationsprotokoll und vom Kommunikationskanal zwischen F- und C-Prozess ab.

Existieren bereits Anforderungen an die maximale Ausführungszeit einer FBA-Operation, hat man in der FBA-internen Kommunikation die meisten Freiheitsgrade, um in der Implementierungsphase diese Anforderungen zu erreichen. Ist andererseits die Implementierungsumgebung schon vollständig festgelegt, kann man aus a) bis f) die Antwortzeiten des FBA ermitteln und mit den Anforderungen vergleichen.

## 6. Zusammenfassung

In diesem Artikel haben wir gezeigt, wie man schon in der Entwurfsphase eines Systems Kommunikationsbeziehungen zwischen Funktionsbausteinen und Capsules mit Hilfe von Funktionsbausteinadaptern modellieren kann. Berücksichtigt man die FBA-interne Kommunikation zwischen F- und C-Prozess und die daraus resultierende Notwendigkeit zur Synchronisation beider Prozesse, kann man verlässliche Aussagen über das Zeitverhalten eines FBA machen. Voraussetzung dafür ist, dass die Implementierungsumgebung des FBA bekannt ist. Ist das nicht der Fall, kann man das Quality of Service – Framework der UML verwenden, um zeitliche Anforderungen an das Ausführungsverhalten eines FBA zu stellen, die in der Implementierungsphase berücksichtigt werden müssen.

## 7. Literatur


[1] M. Polke, B. Will, "Neue Qualität der Führung verfahrenstechnischer Prozesse", Automatisierungstechn. Praxis atp 31 (1989), S. 115-126

[2] Response to the OMG RFP for Schedulability, Performance, and Time, Document Version 1.0, OMG document number ad/2000-08-04

[3] UML draft V1.4, OMG document number ad/2001-02-14

[4] Programmable controllers, Part 3, Programming Languages, IEC 61131-3

[5] B. Selic, J. Rumbaugh, “Using UML for Complex Real-Time Systems”, http://www.objectime.com/otl/technical/umlrt.html

[6] Rational Software Corp., "Rational Rose RealTime Users Guide", 1999

[7] T. Heverhagen, R. Tracht, “Integrating UML-RealTime and IEC 61131-3 with Function Block Adapters”, Proc. of ISORC 2001, May 2-4, 2001, IEEE Computer Society. S. 395-402, http://isorc2001.cs.uni-magdeburg.de/

[8] T. Heverhagen, R. Tracht, “Implementing Function Block Adapters”, Proc. of OMER-2, May 2001, Herrsching a. Ammersee, Report Nr. 2001-03, University of the Federal Armed Forces Munich. S. 11-18
http://inf2-www.informatik.unibw-muenchen.de/GROOM/OMER-2/index.html