Integration von Sprachen für speicherprogrammierbare Steuerungen in die Unified Modeling Language durch Funktionsbausteinadapter

Vom Fachbereich 12 - Maschinenwesen

an der Universität Duisburg-Essen (Standort Essen)

zur Erlangung des akademischen Grades

Doktor-Ingenieur

genehmigte Abhandlung.

Vorgelegt von Dipl.-Ing. Torsten Heverhagen

Datum der mündlichen Prüfung: 2. Juli 2003

Vorsitzender: Prof. Dr.-Ing. V. Hans

Gutachter: Prof. Dr.-Ing. R. Tracht

Prof. Dr. rer. nat. M. Goedicke

Danksagung

Die vorliegende Dissertation entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Automatisierungstechnik des Fachbereiches 12 der Universität Essen. Mein besonderer Dank gilt hierbei meinem Doktorvater und Leiter des Lehrstuhls, Herrn Prof. Dr.-Ing. R. Tracht, für ein sehr interessantes und motivierendes Arbeitsumfeld und für die stetige Unterstützung in Form von weiterführenden Hinweisen und Diskussionen, die mir bei der Bearbeitung der mit dieser Arbeit verbundenen Themen sehr halfen.

Die Idee der Integration von Beschreibungssprachen der Automatisierungstechnik und der Softwaretechnik wurde in einem gemeinsamen Projektantrag von Herrn Prof. Dr. M. Goedicke und Herrn Prof. Dr.-Ing. R. Tracht formuliert und resultierte in einem von der DFG geförderten Projekt „INTAS“. Für die Möglichkeit der Mitarbeit in diesem Projekt möchte ich mich hiermit bedanken. Wesentliche Forschungsschwerpunkte, Lösungskonzepte und Anregungen sind aus dieser Projektmitarbeit in die vorliegende Arbeit eingeflossen. Herrn Prof. Dr. M. Goedicke möchte ich weiterhin für die Bereitschaft danken, das Koreferat für meine Dissertation zu übernehmen.

Einen Großteil meiner Kenntnisse im Bereich der objektorientierten Softwareentwicklung konnte ich bereits während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Datenbanksysteme und Wissensrepräsentation des Fachbereiches 6 der Universität Essen erwerben. Dem Leiter des Lehrstuhls, Herrn Prof. Dr. R. Unland, möchte ich für die kontinuierliche Unterstützung und das Interesse an meiner Arbeit danken. Den Zugang zum Gebiet der objektorientierten Softwareentwicklung verdanke ich dem früheren Betreuer meiner Diplomarbeit Prof. Dr.-Ing. B. Franczyk, der mir die Mitarbeit in verschiedenen industriellen Projekten ermöglichte.

Für die enge und für mich vorteilhafte Zusammenarbeit am Lehrstuhl von Prof. Unland und später im DFG-Projekt INTAS möchte ich mich bei meiner langjährigen Kollegin Frau Dr. Bettina Enders bedanken. Unsere zahlreichen Diskussionen zeigten in mehreren gemeinsamen Veröffentlichungen ihre positiven Resultate.

Während meiner Arbeit am Lehrstuhl für Automatisierungstechnik konnte ich mir immer der tatkräftigen Unterstützung von Herrn Helmut Steinhäuser sicher sein, der mir bei vielen Problemen beim Aufbau der fertigungstechnischen Fallstudie half. Große Unterstützung erhielt ich auch durch Herrn Jingyi Shi, der freundlicherweise einige meiner Aufgaben am Lehrstuhl übernahm, damit ich mehr Zeit zum Schreiben meiner Dissertation hatte.

Beim Aufbau der fertigungstechnischen Fallstudie konnte ich auf die Arbeit vieler Studenten zurückgreifen, ohne die der Aufbau der Fallstudie nicht möglich gewesen wäre. Für die teilweise langjährige Mitarbeit möchte ich mich bei Dirk Zander, Christian Bombosch, Thorsten Grams, Mike Jäger und Max von Groll bedanken.

Für die Bereitstellung des in der Fertigungsanlage eingesetzten Echtzeitbetriebssystems EUROSPlus möchte ich der EUROS Embedded Systems GmbH danken. Die ELSO GmbH Sondershausen stellte uns großzügig Schaltwerkeinsätze und Grundplatten zur Verfügung, die von der Fertigungsanlage bearbeitet werden können. Die Firma Rational Software stellte dankenswerterweise das Tool Rational Rose RealTime zur Verfügung. Für das Interesse an meiner Arbeit möchte ich speziell John Hogg danken, der durch die Bereitstellung einiger unserer Konferenzbeiträge auf den WWW-Seiten der Firma Rational unsere Öffentlichkeitswirksamkeit verbesserte (http://www.rational.com/products/rosert/whitepapers.jsp).

Sehr hilfreich für das Verständnis verschiedener fachlicher Probleme waren für mich unter anderem Gespräche mit Prof. Dr. Bruno Müller-Clostermann, Prof. Dr. Andy Schürr und Dr. Martin Große-Rhode, die sich freundlicherweise Zeit für mich nahmen.

Im außerberuflichen Bereich möchte ich mich besonders bei meiner Freundin, Heike Mandel, bedanken, die viel Geduld und Verständnis bewiesen hat, wenn ich meistens kurz vor Abgabetermin einer Veröffentlichung weniger Zeit für sie hatte. Bei meinen Eltern und Freunden möchte ich mich dafür bedanken, dass sie nicht den Glauben an mich verloren haben, obwohl seit Ende meines Studiums viele Jahre vergingen, bis ich mein Ziel, eine Dissertation zu schreiben, erreicht hatte.

Zu großem Dank bin ich meinem langjährigen Freund und früheren Kommilitonen Robert Hirschfeld verpflichtet, der sich die Mühe gemacht hat, meine Arbeit kritisch durchzulesen, und durch viele Verbesserungsvorschläge wesentlich zur Steigerung der Qualität der vorliegenden Arbeit beigetragen hat.

Abschließend möchte ich mich bei allen Kollegen, Studenten und Freunden bedanken, die ich bisher noch nicht erwähnt habe, die aber ebenfalls zum Gelingen meiner Arbeit beigetragen haben: Mathilda Hirschfeld, Matthias Riebisch, Witold Wendrowski, Krzysztof Czarnecki, Heike Herden, Anne Carpentier, Thomas Eymann, Christophe Rey-Herme, Kay Wilhelm, Sabine Orthey, meine ehemaligen Kollegen in der Firma BERATA, meine Ansprechpartner in der GNS mbH, Herr Bauriedel und Herr Stepan, und viele andere, die ich aus Platzgründen an dieser Stelle nicht mehr erwähnen konnte.

Inhaltsverzeichnis

1 Einleitung

Speicherprogrammierbare Steuerungen (SPS) haben im Bereich der Automatisierungstechnik eine weite Verbreitung gefunden. Die Einsatzgebiete erstrecken sich von der Produktautomatisierung über die Fertigungs- und Gebäudeautomatisierung bis hin zur Prozessautomatisierung. Wegen ihrer hohen Zuverlässigkeit, guten Echtzeiteigenschaften und einfachen Handhabung werden sie vor allem in operativen Ebenen (hauptsächlich der Steuerungsebene) der Unternehmenshierarchie eingesetzt.

In heutigen Produktionsunternehmen findet man oberhalb der Steuerungsebene in erster Linie Standard-Personalcomputer (PCs) oder Workstations, deren Aufgaben in den Bereichen Betriebsdatenerfassung, Produktionsplanung, Produktionssteuerung und Visualisierung liegen. Auf PC-Technik basierende Automatisierungsgeräte werden aber auch zunehmend in der Steuerungsebene parallel zu speicherprogrammierbaren Steuerungen eingesetzt. Da alle Automatisierungsgeräte einer Produktionsanlage einer gemeinsamen Aufgabe dienen, besteht normalerweise die Notwendigkeit einer Vernetzung innerhalb der Steuerungsebene, zu höheren Ebenen und zur Feldebene. Die hohen Anforderungen an Zuverlässigkeit und Sicherheit in der Steuerungsebene müssen trotz dieser heterogenen Struktur erfüllt werden.

Man kann deshalb allgemein feststellen, dass Kommunikationsbeziehungen zwischen SPS- und PC-basierten Computersystemen sehr vielfältig sein können und einer besonderen Untersuchung bedürfen. Je früher im Entwicklungsprozess mit der Analyse dieser Schnittstellen begonnen werden kann, desto geringer ist der Aufwand zur Behebung von eventuellen Spezifikationsfehlern. Erschwert wird diese Analyse dadurch, dass für die Softwareentwicklung beider Computersysteme häufig unterschiedliche Sprachen eingesetzt werden. Um eine möglichst frühzeitige Analyse unterstützen zu können, müssen insbesondere die in der Entwurfsphase verwendeten Sprachen betrachtet werden.

Die Programmierung von speicherprogrammierbaren Steuerungen erfolgt hauptsächlich durch Sprachen der [IEC 61131]. In der Entwurfsphase werden in erster Linie die Ablaufsprache und Funktionsbausteine verwendet. Dem Konzept einer Programmorganisationseinheit 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 zu einem Funktionsbaustein vergleichbaren Programmorganisationseinheit entspricht hier die Klasse. Im Bereich der echtzeitfähigen objektorientierten Softwareentwicklung werden häufig spezielle Klassen, so genannte Capsules, zur Modellierung aktiver Objekte eingesetzt [SelRum 1999a].

Funktionsbausteine können ohne zusätzliche Modellierungselemente nicht direkt mit Capsules verbunden werden. Das liegt in der Art und Weise begründet, wie externe Schnittstellen von Funktionsbausteinen und von Capsules zur Verfügung gestellt werden. In [HevTra 2001a] wurden deshalb Funktionsbausteinadapter (FBAs) vorgestellt, die es bereits in der Entwurfsphase eines Systems erlauben, Kommunikationsbeziehungen zwischen Capsules und Funktionsbausteinen zu modellieren. Mit Funktionsbausteinadaptern werden Funktionsbausteine so ummantelt, dass sie aus Sicht der UML wie Capsules angesprochen werden können. Aus der Sicht der IEC 61131-3 kann ein Funktionsbausteinadapter 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 Internet befindet. Der Funktionsbaustein könnte sich in einer klassischen speicherprogrammierbaren Steuerung oder in einer Soft-SPS auf einem PC oder Mikrocontroller befinden. In [HevTra 2001b] wurden zwei Realisierungen eines Funktionsbausteinadapter gegenübergestellt, wobei in beiden Fällen das Capsule auf einem Industrie-PC und der Funktionsbaustein auf einer speicherprogrammierbaren Steuerung (SIMATIC-S7) liefen, aber unterschiedliche Kommunikationskanäle verwendet wurden (PROFIBUS und Direktverdrahtung von digitalen Ein-/Ausgängen). Weitere plattformabhängige Aspekte von Funktionsbausteinadaptern wurden in [HevTra 2001c] und [HeShGrTr 2002] diskutiert.

Ein Nachteil bei der Verwendung von Capsules ist, dass sie nicht als Standard-Modellierungselement in der UML 1.x enthalten sind. Sie müssen deshalb als Stereotyp definiert werden. Im September 2002 wurde in [UML Superstructure] die UML in einer vorläufigen Version 2.0 um Konzepte erweitert, die es erlauben, jede UML-Klasse wie ein Capsule zu behandeln. Wird diese Erweiterung in der endgültigen Version 2.0 der UML bestätigt, wovon allgemein ausgegangen wird [Bjo 2001], dann wird der Begriff des Capsules überflüssig. Ein Funktionsbausteinadapter ist dann auf jede UML-Klasse anwendbar, um diese mit einem Funktionsbaustein zu verbinden. Der Begriff Capsule wird deshalb zur Definition von Funktionsbausteinadaptern nicht mehr benötigt. Das soll in dieser Arbeit berücksichtigt werden. Da die existierenden Werkzeuge [Rat 2001] aber noch nicht an die UML 2.0 angepasst sind, konnte in Abschnitt 4.2 noch nicht die UML in der Version 2.0 verwendet werden.

Im Weiteren wird zunächst im Kapitel 2 ein Überblick über die UML und die Sprachen für speicherprogrammierbare Steuerungen gegeben und auf aktuelle Entwicklungstrends hingewiesen. Außerdem wird ein Framework vorgestellt, dass den Softwareentwicklungsprozess für ein heterogenes Umfeld unterstützt, in dem mehrere Entwicklungsteams parallel arbeiten. Die Einbettung der UML und der SPS-Sprachen in dieses Framework soll die bereits angesprochene Problemstellung und Notwendigkeit der Integration dieser beiden Sprachen verdeutlichen.

Im Kapitel 3 werden als zentraler Beitrag dieser Arbeit das Konzept der Funktionsbausteinadapter und ein neues UML-Profil vorgestellt, das auf Basis der UML in der Version 2.0 eine Definition für Syntax und Semantik von Funktionsbausteinadaptern liefert. Funktionsbausteinadapter können überall dort eingesetzt werden, wo die UML mit funktionsbausteinorientierten Sprachen wie den Sprachen für speicherprogrammierbaren Steuerungen kombiniert werden soll. Die Problemstellung wird nochmals konkretisiert, bevor ausführlich der Lösungsansatz der Funktionsbausteinadapter vorgestellt und mit anderen Lösungsansätzen verglichen wird.

Beispiele für die Anwendung von Funktionsbausteinadaptern werden in Kapitel 4 vorgestellt. Das erste Anwendungsbeispiel soll die Möglichkeiten der Verifikation von Funktionsbausteinadaptern aufzeigen, die schon in einem frühen Stadium des Softwareentwurfes gegeben sind. Das zweite Anwendungsbeispiel zeigt, wie ein Funktionsbausteinadapter in einer realen Anlage implementiert werden kann.

Einen Ausblick auf Fortführungen und eine Zusammenfassung dieser Arbeit liefert das abschließende Kapitel dieser Arbeit.

2 Stand der Technik

In diesem Kapitel sollen zunächst die beiden in der Einleitung bereits erwähnten Sprachen beschrieben werden, die im Bereich der Automatisierungstechnik häufig zur Programmierung und zum Entwurf von Software eingesetzt werden. Dabei sind die Sprachen nach IEC 61131-3 die klassischen Programmiersprachen der Automatisierungsgeräte (Abschnitt 2.1), während die UML erst in den letzten Jahren ihren Einzug in dieses Anwendungsfeld begonnen hat (Abschnitt 2.2).

2.1 Speicherprogrammierbare Steuerungen (IEC 61131)

Unter Federführung der International Electrotechnical Commission (IEC) wurde der Standard IEC 61131 "Programmable Controllers" entwickelt [IEC 61131]. Dieser Standard umfasst die Erfahrungen, die in 20 Jahren auf dem Gebiet der speicherprogrammierbaren Steuerungen gesammelt worden sind. Die Programmiersprachen für speicherprogrammierbare Steuerungen sind in IEC 61131-3 (Teil 3 von [IEC 61131]) normiert worden.

Standardisiert wurden verschiedene grafische und textorientierte SPS-Sprachen, denen ein gemeinsames Hard- und Softwaremodell zugrunde liegt. Die Literatur zu speicherprogrammierbaren Steuerungen ist sehr zahlreich (z.B. [NeGrLuSi 1998]). An dieser Stelle sollen nur die für Funktionsbausteinadapter wichtigen Konzepte besprochen werden.

Üblicherweise werden SPS-Programme zyklisch ausgeführt, was im Abschnitt 2.1.1 erläutert wird. Im Abschnitt 2.1.2 werden die Datentypen der IEC 61131-3 aufgelistet und kurz beschrieben, damit sie im Abschnitt 2.1.3 auf die Parameter von Funktionsbausteinen angewendet werden können.

2.1.1 Zyklische Ausführung von SPS-Programmen

Speicherprogrammierbare Steuerungen haben in erster Linie die Aufgabe, gemessene physikalische Größen eines zu steuernden Prozesses zu verarbeiten und neue Stellgrößen, die den zu steuernden Prozess beeinflussen, zu berechnen und auszugeben. Es muss dabei sichergestellt werden, dass alle Messwerte, auf die während der Programmabarbeitung zugegriffen wird, zum gleichen Zeitpunkt gemessen wurden. Deshalb existiert in jeder SPS ein spezieller Speicherbereich, in dem die Messgrößen hinterlegt werden, nachdem sie gleichzeitig gemessen wurden. Dieser Speicherbereich wird Prozesseingangsabbild genannt. Für die Ausgangsgrößen (Stellgrößen) der SPS gibt es ein Prozessausgangsabbild. In einer SPS wird immer zuerst das Prozesseingangsabbild erstellt, dann werden die SPS-Programme ausgeführt, die ihre Berechnungsergebnisse zunächst in das Prozessausgangsabbild schreiben, bevor alle

Werte des Prozessausgangsabbildes von der SPS gleichzeitig in den realen Prozess ausgegeben werden (Abbildung 2.1).

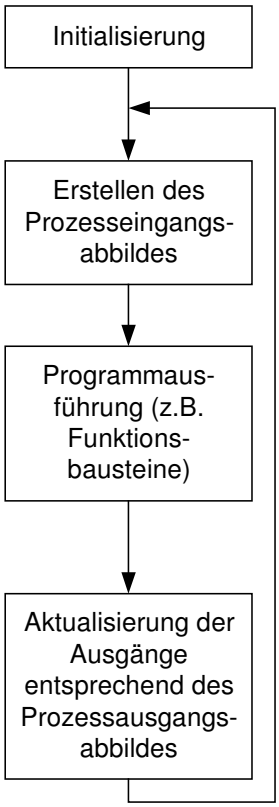


Abbildung 2.1 SPS-Zyklus (vereinfacht)

Nach der Aktualisierung der Prozessausgänge werden wieder die Prozessgrößen gemessen und der Zyklus beginnt erneut. Dieser Zyklus ist typisch für eine SPS und wird deshalb SPS-Zyklus genannt. Ein Vorteil der zyklischen Programmbearbeitung ist, dass sich eine SPS sehr ähnlich im Vergleich zu diskret verdrahteten Verknüpfungssteuerungen verhält. Die Eingänge werden aus der Sicht des Anwenders praktisch parallel bearbeitet. Lediglich die Bearbeitungszeit für den SPS-Zyklus bewirkt, dass die Ausgangsgrößen zeitverzögert ausgegeben werden. Um auch bei der Programmierung den Softwareentwickler an diskrete Verknüpfungssteuerungen zu erinnern, wurden grafische Sprachen wie die Funktionsbausteinsprache (Abschnitt 2.1.3) entwickelt, die den Entwurfssprachen für diskrete Steuerungen nahe kommen.

2.1.2 Datentypen der IEC 61131-3

Die Datentypen, die in der IEC 61131-3 standardisiert wurden, unterteilen sich in drei Gruppen: Die elementaren Datentypen sind für alle SPS-Programmierwerkzeuge verbindlich vorgeschrieben. Darauf aufbauend gibt es Vorschriften zur Bildung von abgeleiteten (benutzerdefinierten) Datentypen. Für die Anwendung bei der Definition von überladenen Funktionen wurden generische Datentypen definiert.

2.1.2.1 Elementare Datentypen

Zu den elementaren Datentypen zählen Typen für die Darstellung von Zahlen, Zeichenketten, Uhrzeit, Datum und Bitgruppen. Der Wertebereich der elementaren Datentypen der IEC 61131-3 unterscheidet sich leicht von denen anderer Programmiersprachen. Deshalb wird in Anhang C zu jedem elementaren Datentyp dessen Wertebereich aufgeführt.

2.1.2.2 Abgeleitete Datentypen

Anwender- oder herstellerdefinierte Datentypen können durch die Schlüsselwörter TYPE ... END_TYPE deklariert werden. Dabei können verschiedene Ziele verfolgt werden.

Eine direkte Ableitung bedeutet, dass der Datentyp einfach nur umbenannt wird:

```
TYPE TemperaturTyp: SINT; END_TYPE
```

Soll der Wertebereich eines elementaren Datentyps eingeschränkt werden, dann kann das durch eine Bereichsangabe nach dem Datentyp geschehen:

```
TYPE TemperaturTyp: SINT (-40..70); END_TYPE
```

Ein Aufzählungstyp legt fest, dass Variablen dieses Typs nur Werte annehmen können, die in der zugehörigen Liste der Bezeichner festgelegt sind:

```
TYPE WochentagTyp: (Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag,
                    Sonntag);
END_TYPE
```

Ein Strukturtyp legt fest, dass Variablen dieses Typs Unterelemente haben können, auf die über Namen zugegriffen werden kann und die definierten Typen angehören:

```
TYPE Pos_info
 STRUCT
  Pos_Status: BOOL;
  Part_on: Part_info;
 END_STRUCT
END_TYPE

TYPE Part_info
 STRUCT
  Part_Type: INT;
  Part_Sender: INT;
  Part_Receiver: INT;
 END_STRUCT
END_TYPE
```

Ein Feldtyp legt fest, dass Variablen dieses Typs eine Aneinanderreihung gleichartiger Elemente sind, wobei auf die Komponenten über einen Index zugegriffen werden kann:

```
TYPE TRS_pos: array[1..7] of Pos_info; END_TYPE
```

2.1.2.3 Generische (allgemeine) Datentypen

Allgemeine Datentypen können auf überladene Eingänge oder Ausgänge von Funktionen und Funktionsbausteinen angewendet werden. Die allgemeinen Typen werden durch die Vorsilbe ANY gekennzeichnet (Abbildung 2.2).

```
ANY
  ANY_DERIVED
  ANY_ELEMENTARY
    ANY_MAGNITUDE
      ANY_NUM
        ANY_REAL
          LREAL
          REAL
        ANY_INT
          LINT, DINT, INT, SINT
          ULINT, UDINT, UINT, USINT
      TIME
    ANY_BIT
      LWORD, DWORD, WORD, BYTE, BOOL
    ANY_STRING
      STRING
      WSTRING
    ANY_DATE
      DATE_AND_TIME
      DATE, TIME_OF_DAY
```

Abbildung 2.2 Generische Datentypen der IEC 61131-3 (Quelle: [IECDatenTypen 1993])

Der allgemeine Datentyp für abgeleitete Typen ist meistens ANY_DERIVED. Es gilt dabei aber folgende Ausnahme bei der Anwendung von allgemeinen auf abgeleitete Datentypen:

Bei einer direkten Ableitung oder einer Wertebereichseinschränkung hat der abgeleitete Typ den gleichen allgemeinen Typ wie der elementare Typ, von dem er abgeleitet wurde. Für den Temperaturtyp

```
TYPE TemperaturTyp: SINT (-40..70); END_TYPE
```

ist deshalb ANY_INT der allgemeine Typ und nicht ANY_DERIVED.

2.1.3 Funktionsbausteine nach IEC 61131-3

Dem Konzept einer Softwarekomponente mit Schnittstellendefinition und davon gekapselter Verhaltensbeschreibung und Implementierung kommen die Funktionsbausteine am nächsten. Sie dienen in der IEC 61131-3 als Programmorganisationseinheit. Das bedeutet, dass der Programmteil eines Funktionsbausteins in einer beliebigen Sprache der IEC 61131-3 (Ablaufsprache, Strukturierter Text, Anweisungsliste, Kontaktplan, Funktionsbausteinsprache) geschrieben werden kann.

Der prinzipielle Aufbau eines Funktionsbausteins ist in Abbildung 2.3 dargestellt.

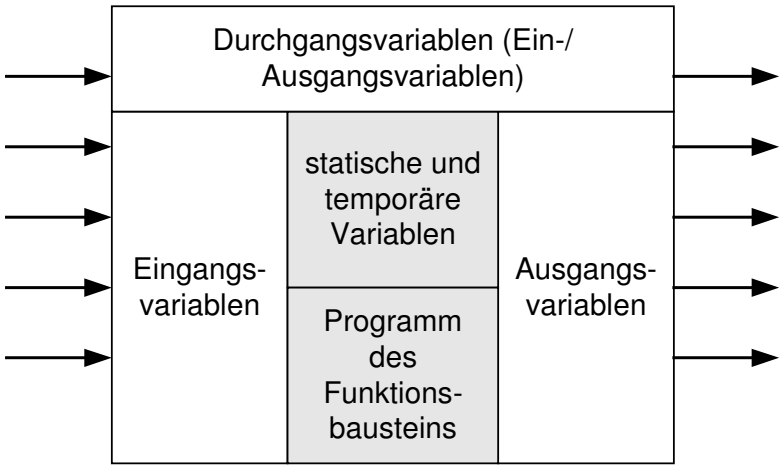


Abbildung 2.3 Prinzipieller Aufbau eines Funktionsbausteines

Die externe Schnittstelle eines Funktionsbausteines wird durch die Eingangs-, Ausgangs- und Durchgangsvariablen beschrieben. Intern besitzt ein Funktionsbaustein statische und temporäre Variablen und das eigentliche Programm, das das Verhalten der Funktionsbausteines bestimmt.

Eingangsvariablen können vom internen Programm nur gelesen, aber nicht geschrieben werden. Schreibender Zugriff auf Eingangsvariablen ist nur von außerhalb erlaubt. Ausgangsvariable können vom internen Programm sowohl gelesen als auch geschrieben werden. Von außerhalb des Funktionsbausteines dürfen Ausgangsvariablen nur gelesen werden. Durchgangsvariablen dürfen innerhalb und außerhalb des Funktionsbausteines geschrieben und gelesen werden. Sie verhalten sich ähnlich wie globale Variablen, auf die nur von den Funktionsbausteinen aus zugegriffen werden kann, bei denen sie deklariert sind. Auf statische und temporäre Variablen kann nur vom internen Programm aus zugegriffen werden.

Das Programm eines Funktionsbausteines wird, wenn nicht anders definiert, zyklisch ausgeführt. Das heißt, innerhalb eines SPS-Zyklus muss das Programm vollständig abgearbeitet werden. Da nach der Initialisierung der SPS abgesehen von den temporären alle Variablen des Funktionsbausteines ihren Zustand über einen Zyklus hinaus beibehalten, muss die mehrmalige Ausführung eines Funktionsbausteines mit gleichen Eingangsbelegungen nicht immer die gleichen Ausgangsgrößen liefern.

Zur Darstellung von Funktionsbausteinen kann die Funktionsbausteinsprache verwendet werden. Die Deklaration der Schnittstelle eines Funktionsbausteines kann wie in Abbildung 2.4 gezeigt erfolgen.

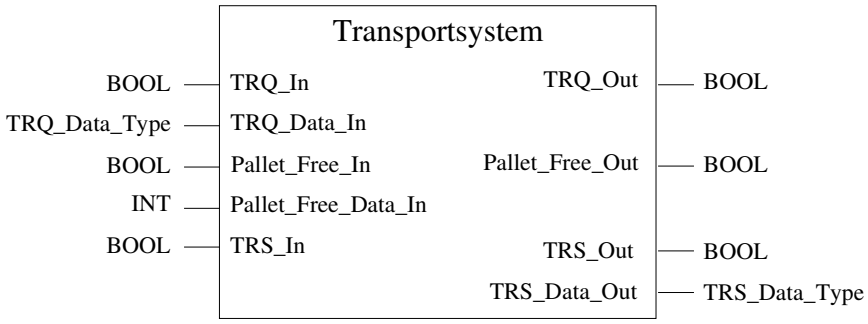


Abbildung 2.4 Beispiel für Funktionsbaustein-Deklaration (siehe auch Seite 91)

Die Eingangsvariablen werden links und die Ausgangsvariablen rechts eingetragen. Die Namen der Variablen findet man innerhalb des Funktionsbausteines und die Datentypen außerhalb. Das Verhalten eines Funktionsbausteines wird häufig mit Hilfe von Zeitdiagrammen beschrieben. Das Zeitdiagramm aus Abbildung 2.5 zeigt einen Ausschnitt der Verhaltensbeschreibung des Funktionsbausteines *Transportsystem*.

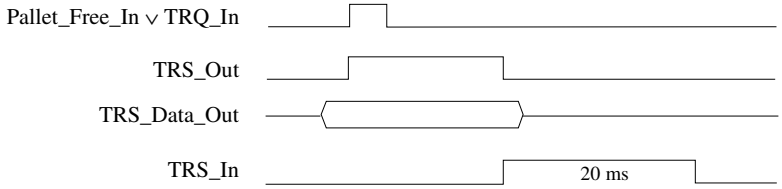


Abbildung 2.5 Zeitdiagramm zur Erläuterung des Verhaltens (siehe auch Seite 91)

Funktionsbausteine müssen instanziiert werden, bevor sie aufgerufen werden können. Ausgänge von Instanzen können mit Eingängen anderer Instanzen verbunden werden. Eingänge können auch mit konstanten Werten belegt werden. Die Namen der Instanzen werden oberhalb des Funktionsbausteinsymbols geschrieben (Abbildung 2.6).

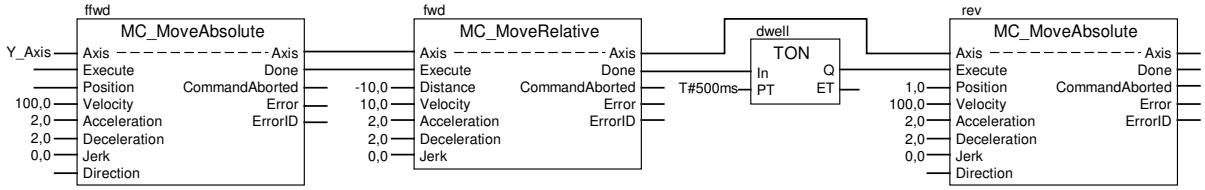


Abbildung 2.6 Beispiel für ein Diagramm in der Funktionsbausteinsprache (siehe auch Seite 71)

Der große Unterschied zwischen Funktionsbausteinen und Softwarekomponenten, wie sie aus [SelRum 1999a] und Architekturbeschreibungssprachen [GoeCraDob 1994] bekannt sind, ist, dass bei Funktionsbausteinen die Schnittstellen durch Variablen und deren zeitlichen Belegungen gegeben sind, und nicht durch Operationen mit einer Anzahl von Parametern.

2.2 Unified Modeling Language

Die Unified Modeling Language (UML) ist eine überwiegend grafische Sprache, die es erlaubt, die Artefakte eines softwareintensiven Systems zu visualisieren, zu spezi-

fizieren, zu konstruieren und zu dokumentieren [BooRumJac 1999]. Eine erste Version der UML wurde 1997 von der Object Management Group (OMG) als Standard verabschiedet. Die aktuelle Version 1.4 der UML wird voraussichtlich im Jahr 2003 durch Version 2.0 abgelöst. Dieser Arbeit soll eine in [UML Superstructure] veröffentlichte Arbeitsversion der Superstructure der UML 2.0 zugrunde gelegt werden. Einige Bestandteile der UML wie zum Beispiel die Object Constraint Language (OCL) sind zurzeit noch nicht veröffentlicht worden und müssen deshalb der Version 1.4 der UML entnommen werden.

In den folgenden Abschnitten werden einige Konzepte der UML vorgestellt, die zum Verständnis dieser Arbeit wichtig sind. Besonders herauszuheben ist dabei das Port-Konzept, das in der UML 2.0 erstmalig enthalten ist. Im Gegensatz zu vielen anderen Sprachstandards bietet die UML Erweiterungsmöglichkeiten an, die es UML-Entwicklern oder UML-Werkzeugherstellern erlauben, eigene Ergänzungen zu den Standardelementen der UML hinzuzufügen. Für das Verständnis dieser Erweiterungsmöglichkeiten ist das Verständnis der UML-Spracharchitektur notwendig.

2.2.1 Die Spracharchitektur der UML

Die zentralen Elemente der Objektorientierung sind Klasse und Objekt. Objekte werden aus Klassen erzeugt. Die Klasse beschreibt dabei die Struktur und das Verhalten der aus ihr erzeugten Objekte. Eine Klasse befindet sich aus der Sicht eines Objektes (einer Instanz) auf einer Metaebene. Klassen können ihrerseits auch wieder als Objekte dargestellt werden, die von Klassen auf einer höheren Metaebene instanziiert wurden.

Die allgemeinen Eigenschaften von Klassen und Objekten werden von der Object Management Group (OMG) standardisiert. Als Ergebnis dieser Standardisierung sind u. a. zwei unterschiedliche Arten von Sprachen entstanden, mit denen man Objekte beschreiben kann: Die IDL (Interface Definition Language) [CORBA 1998] und die UML. Die IDL dient zur Beschreibung von Schnittstellen, die Objekte in einer CORBA-Umgebung zur Verfügung stellen.

Die Definitionen der Sprachen UML und IDL basieren auf dem gleichen Metamodell, dem MOF-Modell [MOF OMG]. MOF steht für Meta Object Facility. Der Zusammenhang zwischen MOF, IDL und UML ist in Abbildung 2.7 dargestellt. Im Weiteren soll auf die IDL und CORBA nicht näher eingegangen werden, da an dieser Stelle nur darauf hingewiesen werden soll, dass das MOF-Modell nicht ausschließlich für die UML definiert wurde.

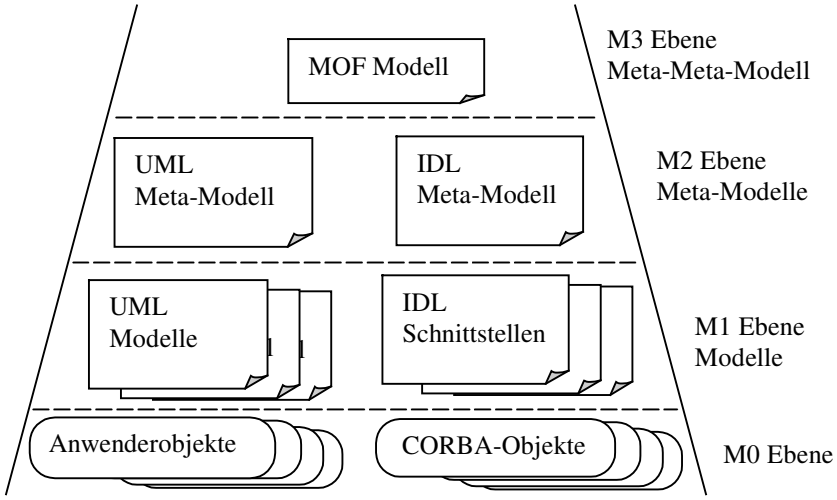


Abbildung 2.7 Spracharchitektur der UML

Auf der M0 Ebene befinden sich die Objekte, die von den Anwendern der UML bzw. der IDL (mit Hilfe von Klassen, Schnittstellen, Statecharts, Strukturdiagrammen usw.) modelliert wurden. Die M0 Ebene entsteht, wenn ablauffähige Programme auf einem Computer ausgeführt werden. Die im Speicher eines Computers oder verteilt in einem Computernetzwerk vorhandenen Objekte (oder auch Instanzen) sind dann die, die zur M0 Ebene gehören und nach Vorschrift der Modelle aus der M1 Ebene gebildet wurden.

Auf der M1 Ebene befinden sich die Modelle, die von den Anwendern der UML bzw. der IDL definiert wurden. In Bezug auf die UML befinden sich in dieser Ebene insbesondere Objekte wie Klassen mit Attributen, Operationen, Assoziationen und Zustandsmaschinen. Neben diesen „eigentlichen“ Objekten der M1 Ebene benötigt man auch Diagramme wie Klassendiagramme, Strukturdiagramme, Statecharts usw., die Zusammenhänge zwischen diesen Objekten visualisieren und verdeutlichen oder das dynamische Verhalten von Objekten beschreiben. Die Diagramme sind ebenfalls Objekte der M1 Ebene, deren Klassen in der M2 Ebene definiert werden müssen.

Auf der M2 Ebene befinden sich Metaklassen, die von den Standardisierungsgruppen der OMG definiert wurden oder werden. Eine sehr häufig benutzte Metaklasse heißt *Class*. Alle Klassen auf der M1 Ebene sind Instanzen von Class. In [UML Superstructure] wird die Definition von Class um ein wesentliches Konzept erweitert – dem Port-Konzept, das im Abschnitt 2.2.2 beschrieben wird.

Die M2-Klassen sind ihrerseits auch wieder Objekte der Klassen auf der M3 Ebene. Diese Meta-Ebenen Architektur hat sich bereits in objektorientierten Programmiersprachen wie Smalltalk [GolRob 1989] bewährt.

2.2.2 Das Port-Konzept der UML in der Version 2.0

Unter einem Port versteht man in der UML eine Anschlussstelle, über die Nachrichten versendet und empfangen werden können. Welche Nachrichten das sind, wird über die angebotenen (*provided*) und geforderten (*required*) Schnittstellen (*Interfaces*) eines Ports definiert. Die Metaklasse *EncapsulatedClassifier* beinhaltet eine Assoziation namens *port*, durch die ihr Ports zugeordnet werden können. Abbildung 2.8 zeigt, dass EncapsulatedClassifier eine Oberklasse von *Class* ist. Dadurch können jeder Klasse auf der M1 Ebene Ports zugeordnet werden.

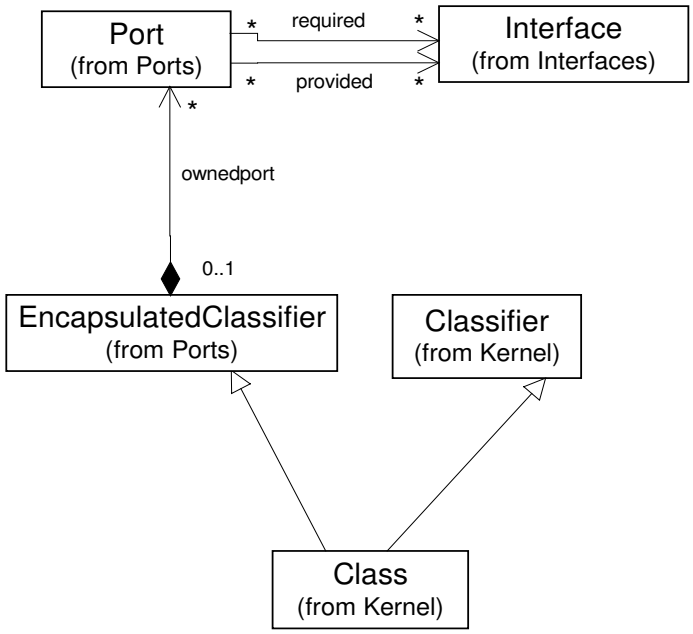


Abbildung 2.8 Ausschnitt aus dem Metamodell (M2) der UML

Die Darstellung von Ports in Klassendiagrammen auf der M1 Ebene war bisher nur durch das UML-Werkzeug [Rat 2001] definiert. In [UMLPorts] wird eine offizielle Notation vorgeschlagen, die sich leicht von der bisher bekannten unterscheidet. Insbesondere wurde das Konzept der Protokolle durch required und provided Interfaces ersetzt. Abbildung 2.9 zeigt ein Beispiel für eine Klasse mit Port in der neuen Notation.

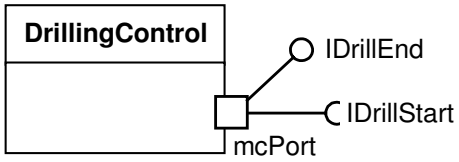


Abbildung 2.9 Beispiel für eine Klasse mit einem Port

Die Klasse *DrillingControl* enthält das Port *mcPort*, das *IDrillEnd* als Interface anbietet und *IDrillStart* benötigt. In Strukturdiagrammen, die eine Abwandlung von Kollaborationsdiagrammen sind, können Ports miteinander verbunden werden (Abbildung 2.10).

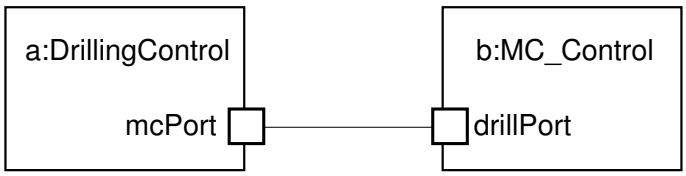


Abbildung 2.10 Beispiel eines Strukturdiagramms

Der in [SelRum 1999a] geprägte Begriff des konjugierten Ports, der in Strukturdiagrammen schwarz dargestellt wird, ist in [UMLPorts] nicht enthalten. In der UML 2 kann man zwei Ports miteinander verbinden, wenn ihre geforderten und benötigten Schnittstellen zueinander passen.

Ein weiterer wichtiger Unterschied zwischen [SelRum 1999a] und dem neuen Port-Konzept der UML 2 ist, dass in [SelRum 1999a] Klassen ausschließlich (!) über Ports miteinander kommunizieren dürfen. Dadurch ist zu jeder Klasse nicht nur die Exportschnittstelle, sondern auch die Importschnittstelle bekannt. Das entspricht teilweise dem Konzept einer wiederverwendbaren Architekturkomponente aus [GoeCraDob 1994]. Auch in UML-Werkzeugen wie [Rat 2001] sind ausschließlich Ports zur Schnittstellenbeschreibung erlaubt.

Im Weiteren soll aus den eben genannten Gründen davon ausgegangen werden, dass jede aktive Klasse, also eine Klasse, deren Verhalten durch ein Statechart beschrieben wird, ausschließlich über Ports mit anderen Klassen kommunizieren darf. Diese Annahme ermöglicht die Ausdehnung des Konzeptes der Funktionsbausteinadapter auf die *Specification and Description Language (SDL)* [SDL 1993], was im Weiteren aber nicht näher besprochen wird. Einen Vergleich zwischen der UML und der SDL liefert [SelRum 1999b].

2.2.3 Die Object Constraint Language

Die Object Constraint Language (OCL) ist eine Sprache zur Formulierung von Einschränkungen für UML-Artefakte. Der OCL liegt eine formale Grammatik zugrunde, die in [OCL Grammatik] definiert wird. Die OCL kann und wird in verschiedenen Metaebenen mit unterschiedlichen Zielen verwendet werden.

Auf der M1 Ebene können mit Hilfe der OCL Einschränkungen für Anwenderobjekte festgelegt werden. Das können zum Beispiel Wertebereichseinschränkungen von Attributen von Anwenderobjekten sein. Einschränkungen dieser Art werden als Invarianten bezeichnet. Für Operationen können auch Vor- und Nachbedingungen spezifiziert werden. Eine wichtige Eigenschaft der OCL ist, dass der Zustand von Objekten nicht durch OCL-Befehle verändert werden darf. Man kann in der OCL auch nicht beschreiben, was passiert, wenn eine Bedingung verletzt wird. Durch diese Eigenschaften wird gewährleistet, dass die OCL frei von Nebeneffekten ist.

Auf der M2 Ebene können Einschränkungen für Sprachelemente der UML definiert werden. So wurde zum Beispiel für die Metaklasse Interface festgelegt, dass alle Eigenschaften öffentlich sein müssen. Mit Hilfe der OCL kann das folgendermaßen ausgedrückt werden:

```
context Interface inv: self.feature->forAll( f | f.visibility = #public )
```

Ein UML-Werkzeug, das Interfaces unterstützt, muss sicherstellen, dass alle Einschränkungen eingehalten werden. Da die OCL eine formale Sprache ist, können die mit ihrer Hilfe formulierten Einschränkungen direkt von UML-Werkzeugen verarbeitet werden.

2.2.4 Erweiterungsmöglichkeiten der UML

Mit Hilfe von Constraints (Einschränkungen, die mit Hilfe der OCL definiert wurden) kann nicht nur die Standardisierungsgruppe der UML auf der M2 Ebene arbeiten, sondern auch Hersteller oder Anwender von UML-Werkzeugen. Dadurch können werkzeug- oder projektspezifische Einschränkungen formuliert werden. Andererseits ist es manchmal zweckmäßig, UML-Elementen zusätzliche Informationen (zum Beispiel über Echtzeiteigenschaften) hinzuzufügen. Dafür gibt es die Erweiterungsmöglichkeit der TaggedValues (tagged value). Mit Hilfe von TaggedValues kann man existierenden Elementen auf der Ebene M1 oder M2 neue Attribute oder Referenzen anfügen, ohne die Definition dieser Elemente zu verändern. TaggedValues können genauso wie normale Attribute oder Referenzen typisiert werden und erhalten ebenfalls einen Namen.

Wenn in einem Projekt oder in einem Werkzeug ein UML-Element mit und ohne Erweiterungen eingesetzt werden soll, dann können die Constraints und TaggedValues auch auf einen Stereotyp und nicht direkt auf das Standard-UML-Element angewendet werden. Ein Stereotyp basiert immer auf einem UML-Element und wird durch einen eigenen Namen gekennzeichnet. Im Gegensatz zu normalen UML-Elementen darf ein Stereotyp ausschließlich Constraints und TaggedValues enthalten. In einem UML-Werkzeug kann ein Stereotyp dann wie ein normales UML-Element verwendet werden, das um die TaggedValues erweitert und um die Constraints eingeschränkt wurde.

Wenn eine Menge von Constraints, TaggedValues und Stereotypen einem gemeinsamen Zweck dienen oder aus anderen Gründen zusammengefasst werden sollen, können sie einem UML-Profil (profile) zugeordnet werden. Ein Profil ist ein spezielles UML-Paket (package), das auf der M2 Ebene der UML zur Organisation der UML-Erweiterungen genutzt werden kann [UMLProfile].

Beispiele für Stereotypen, Constraints und TaggedValues, die in einem Profil zusammengefasst werden, werden im nächsten Abschnitt „Spezifikation von Echtzeiteigenschaften in der UML“ beschrieben.

2.2.5 Spezifikation von Echtzeiteigenschaften in der UML

Die Spezifikation von Echtzeiteigenschaften wurde in den ersten Versionen der UML nicht durch spezielle Konzepte unterstützt. Wenn echtzeitspezifische Anforderungen formuliert werden sollten, geschah das durch textuelle Annotationen. Um diesen Schwachpunkt zu beseitigen, wurde die Version 1.4 der UML um ein UML-Profil ergänzt, das Elemente zur Spezifikation von Performanz, Ausführbarkeit und Zeit von Standard-UML-Elementen enthält [UML Sched, Perf, Time]. Dieses Profil soll diesem Abschnitt zugrunde gelegt werden.

Da die UML ein Softwaresystem auf verschiedenen Abstraktionsstufen beschreiben kann, müssen auch Echtzeiteigenschaften auf diesen verschiedenen Abstraktionsstufen spezifiziert werden können. Deshalb unterscheidet man zwischen den Eigenschaften, die von einer Implementierungsplattform angeboten werden, und den Eigenschaften, die von einer höheren Abstraktionsebene gefordert werden.

Auch innerhalb einer Abstraktionsebene kann man zwischen angebotenen und geforderten Echtzeiteigenschaften differenzieren. In Abbildung 2.11 ist ein Kollaborationsdiagramm zu sehen, in dem zwei Klienten auf einen gemeinsamen Server zugreifen.

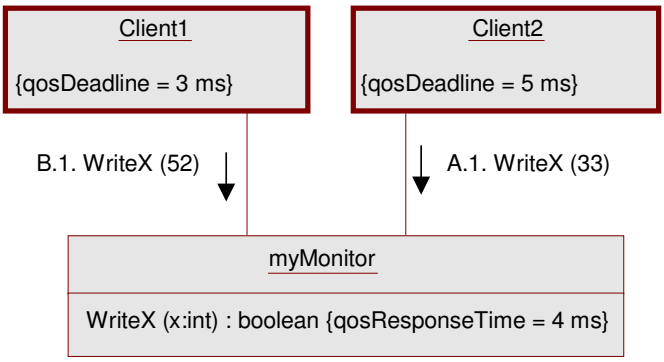


Abbildung 2.11 Kollaborationsdiagramm mit Echtzeiteigenschaften

Client1 wurde das TaggedValue *qosDeadline* mit dem Wert *3 ms* hinzugefügt. Demgegenüber hat die Operation *WriteX* von *myMonitor* eine Antwortzeit von *4 ms*. Das ist aus dem TaggedValue *qosResponseTime* abzulesen, um das die Operation ergänzt wurde.

Die beiden TaggedValues qosDeadline und qosResponseTime wurden in [UML Sched, Perf, Time] zur Verfügung gestellt und können allen dafür geeigneten UML-Elementen auf der M1 Ebene hinzugefügt werden. Weiterhin wurden neue Stereoty-

pen zur Darstellung von Zeit und anderer Ressourcen in diesem Profil definiert. Durch die einheitliche und formale Definition solcher TaggedValues und Stereotypen sollen UML-Werkzeuge in die Lage versetzt werden, UML-Modelle auf Widersprüche oder Konsistenz hinsichtlich ihrer Echtzeitanforderungen zu überprüfen.

Da die meisten Softwaresysteme in der Automatisierungstechnik, insbesondere auch die Funktionsbausteinadapter, Echtzeitanforderungen unterliegen, soll an dieser Stelle darauf hingewiesen werden, dass sich das UML-Profil für Ausführbarkeit, Performanz und Zeit direkt auf Funktionsbausteinadapter anwenden lässt [HevTra 2001c]. In der Version 2.0 der UML ist das UML-Profil aus [UML Sched, Perf, Time] voraussichtlich als Standard-Paket enthalten.

2.3 Das ViewPoint-Framework

In die Entwicklung komplexer automatisierungstechnischer Systeme sind normalerweise mehrere Menschen oder sogar mehrere Teams involviert, die unterschiedliche Sichtweisen auf das System haben. Solche Sichtweisen können zum Beispiel daraus resultieren, dass nur Teilbereiche des Gesamtsystems betrachtet werden, oder daraus, dass das Gesamtsystem auf unterschiedlichen Abstraktionsebenen dargestellt wird. Weitere Sichtweisen ergeben sich aus ökonomischen und arbeitsorganisatorischen Gesichtspunkten, worauf an dieser Stelle aber nicht näher eingegangen werden soll.

Natürlicherweise gibt es zwischen den gebildeten Sichtweisen auch Überlappungen und Abhängigkeiten, woraus sich die Notwendigkeit einer Koordinierung der mit diesen verschiedenen Sichtweisen verbundenen Arbeiten ableiten lässt. Erschwert wird diese Koordination zusätzlich dadurch, dass in den verschiedenen Sichtweisen oft nicht die gleichen Beschreibungssprachen zum Einsatz kommen.

Ein Framework, das zur Integration verschiedener Sichtweisen während der Systementwicklung dient, wurde in [GoFiKrNuFi 1992] vorgestellt. Eine Sichtweise wird dort *ViewPoint* genannt.

Das ViewPoint-Framework eignet sich sehr gut zur Erläuterung der mit dieser Arbeit angesprochenen Problematik. Die Softwareentwickler für die SPS- und die PC-basierten Computersysteme sind häufig auch in voneinander getrennte Arbeitsgruppen aufgeteilt, in denen zudem noch unterschiedliche Sprachen (IEC 61131-3 bzw. UML) verwendet werden. In den nächsten beiden Abschnitten soll deshalb das ViewPoint-Framework etwas genauer vorgestellt werden. Im Abschnitt 3.9 erfolgt dann eine Einbettung des Konzeptes der Funktionsbausteinadapter in das ViewPoint-Framework.

2.3.1 ViewPoints

Ein ViewPoint ist der Basisbaustein des ViewPoint-Frameworks. Ein ViewPoint repräsentiert die Sicht, Sichtweise oder Perspektive, die ein am Softwareentwicklungsprozess beteiligter Akteur hat. Im Rahmen dieser Arbeit sollen in erster Linie Akteure wie UML- oder SPS-Entwickler betrachtet werden. Weitere Beispiele für Akteure wären die Anwender des zu entwickelnden Softwaresystems oder Projektleiter des Entwicklungsteams.

Der Inhalt eines ViewPoints wird in fünf Bestandteile, die *Slots* genannt werden, aufgegliedert:

- *Representation Style*: enthält eine Beschreibung der Sprache oder der Notation, mit deren Hilfe der Ausschnitt des Gesamtsystems beschrieben wird, der durch den ViewPoint betrachtet wird.
- *Domain*: enthält eine natürlichsprachliche Beschreibung des Ausschnittes der realen Welt, der durch den ViewPoint betrachtet wird.
- *Specification*: enthält eine formale Beschreibung (Spezifikation) des Ausschnittes der realen Welt in der Sprache oder in der Notation, die im *Representation Style* definiert wurde.
- *Work Plan*: enthält eine Vorgehensweise oder einen Entwicklungsprozess, nach der oder nach dem eine Spezifikation aufgestellt werden kann.
- *Work Record*: enthält eine Historie der Aktivitäten, die zum aktuellen Zustand der Spezifikation geführt haben, und den aktuellen Entwicklungsstand der Spezifikation.

In Abbildung 2.12 sind zwei ViewPoints beispielhaft dargestellt. Beide beziehen sich auf das Anwendungsbeispiel *Motion Control*, das in Abschnitt 4.1 vorgestellt wird. Das erste ViewPoint bezieht sich auf die Sichtweise eines SPS-Entwicklers, der die Steuerung des Bewegungsablaufs der translatorischen Achse einer Bohrmaschine entwickeln soll. Die Spezifikation wurde in der Funktionsbausteinsprache erstellt. Die erlaubte Belegung der Eingänge und der Ausgänge des im ersten ViewPoint dargestellten Funktionsbausteins kann grafisch durch Zeitdiagramme dargestellt werden. Zeitdiagramme sind eine weitere Sichtweise auf das Problem der Bewegungssteuerung. Deshalb ist in Abbildung 2.12 ein weiteres ViewPoint für ein Zeitdiagramm eingeführt worden. Eine genaue inhaltliche Erläuterung der Beispielspezifikationen in den ViewPoints findet man in Abschnitt 4.1.

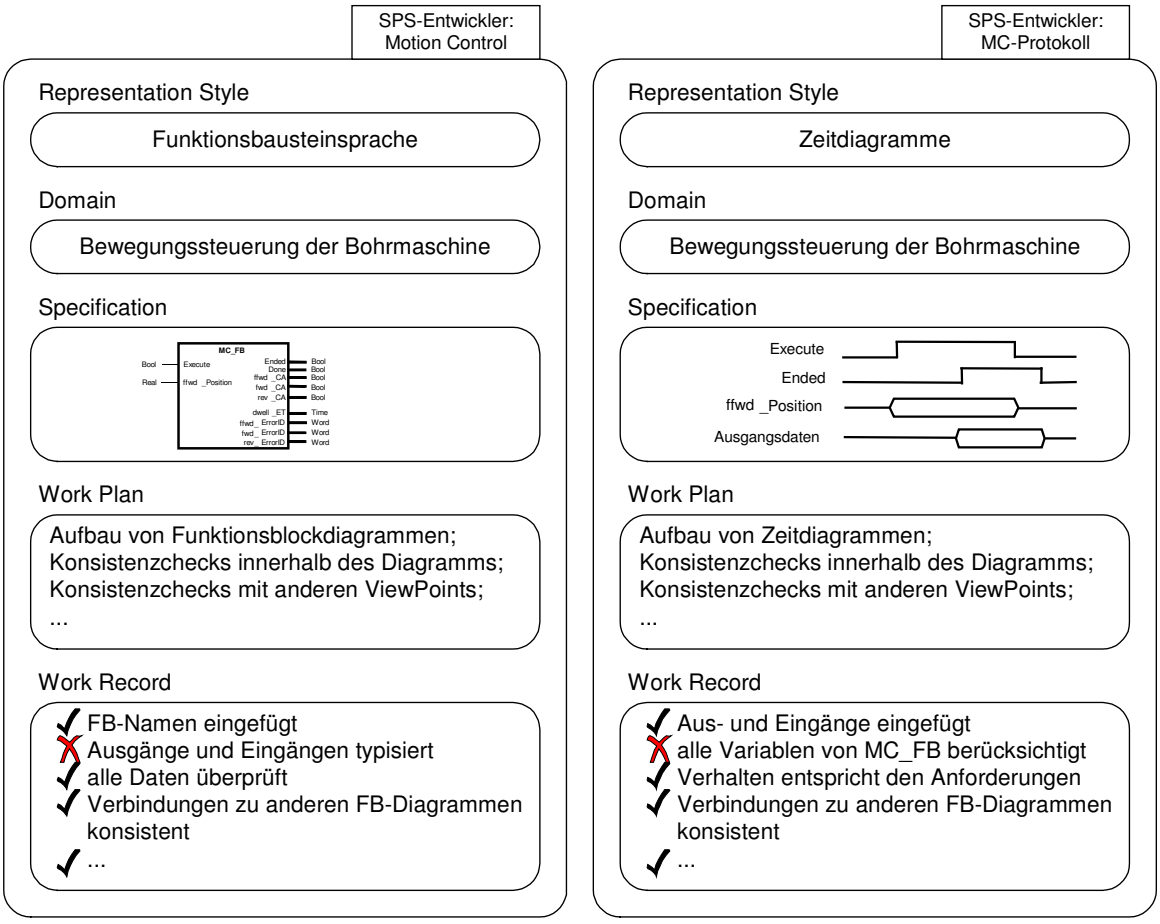


Abbildung 2.12 Zwei Beispiele für ViewPoints von SPS-Entwicklern

Im Work Plan der ViewPoints wird unter anderem beschrieben, wie die Konsistenz mit anderen ViewPoints sichergestellt werden kann. So sollte zum Beispiel jeder Eingang und jeder Ausgang des Funktionsbausteins im Zeitdiagramm enthalten sein. Im Work Record ist aufgeführt, ob solche Konsistenzchecks durchgeführt wurden, und welche Ergebnisse sie erbrachten.

Der Funktionsbaustein aus Abbildung 2.12 ist in eine übergeordnete Steuerung der gesamten Bohrzelle eingebettet. Das Entwicklungsteam der Gesamtsteuerung muss verschiedene Teilsysteme der Bohrzelle koordinieren. Als Beschreibungssprache wird dort die UML verwendet, weil die Gesamtsteuerung ereignisorientiert arbeiten soll und ein PC-basiertes Computersystem eingesetzt wird.

In Abbildung 2.13 sind zwei ViewPoints dargestellt, die sich auf die Gesamtsteuerung der Bohrzelle beziehen, in der sich die Bohrmaschine befindet. Die erste Spezifikation wurde in der Form eines UML-Klassendiagramms erstellt. Die darin enthaltene Klasse enthält ein Port, dessen Protokoll in Form eines Protokollstatecharts dargestellt werden kann. Für das zum Port gehörende Protokollstatechart wurde ein weiteres ViewPoint in Abbildung 2.13 eingeführt. Eine mögliche Konsistenzbedingung zwischen diesen beiden ViewPoints wäre, dass alle im Protokollstatechart enthaltenen Nachrichten auch im Port definiert wurden.

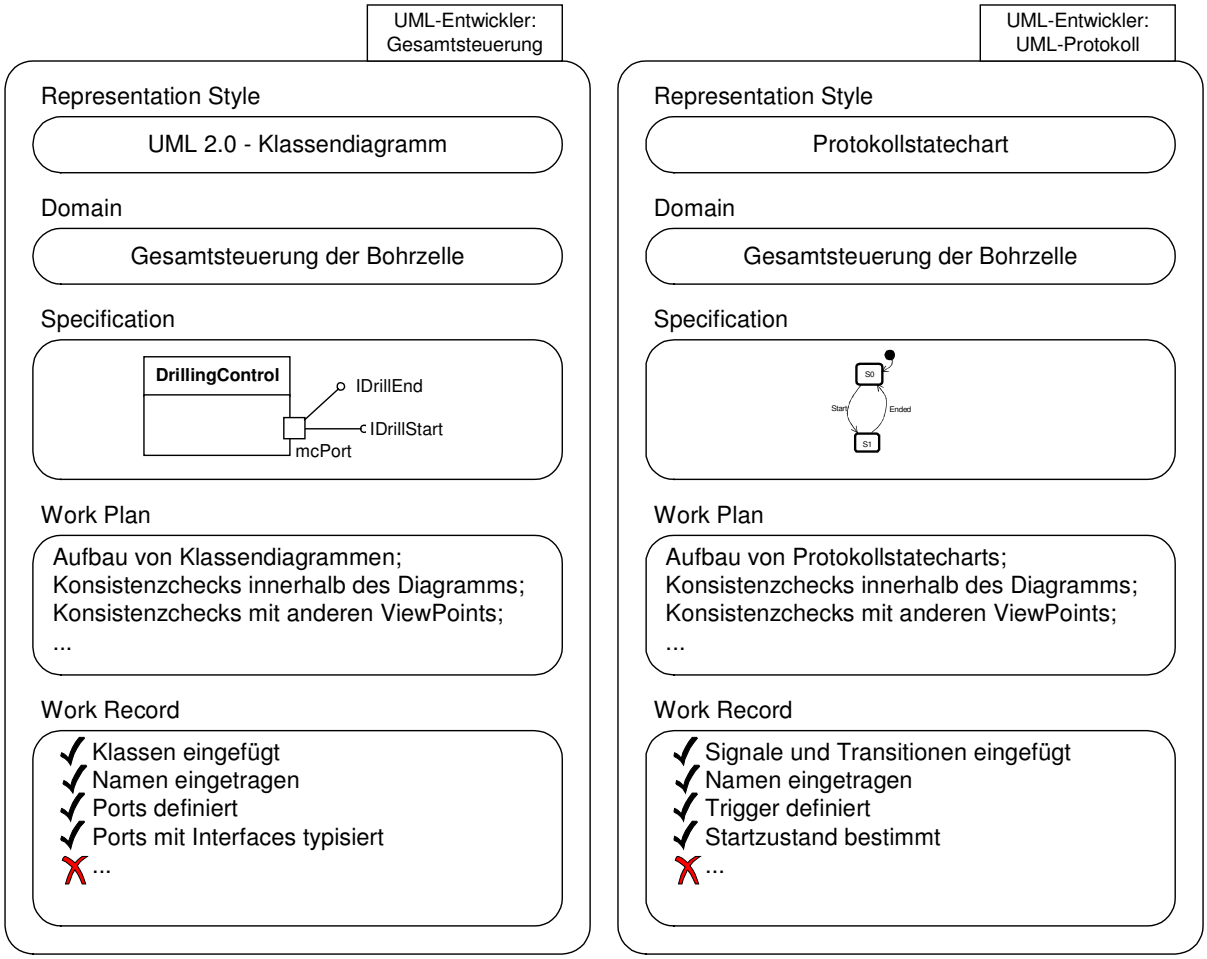


Abbildung 2.13 Zwei Beispiele für ViewPoints von UML-Entwicklern

Kompliziertere Konsistenzchecks zwischen ViewPoints ergeben sich, wenn die Spezifikationen unterschiedlichen Sprachfamilien angehören, wie das bei den SPS-Sprachen und der UML der Fall ist. In den Beispielen aus Abbildung 2.12 und Abbildung 2.13 gilt es sicherzustellen, dass die Daten, die zu bestimmten Zeitpunkten in den Ausgängen der Funktionsbausteine der Bewegungssteuerung bereitgestellt werden, mit Hilfe von Nachrichten über ein Port der Gesamtsteuerung gesendet werden. Umgekehrt müssen die Nachrichten, die über ein Port der Gesamtsteuerung an die Bewegungssteuerung versendet werden, in Belegungen der offenen Eingänge der Funktionsbausteine der Bewegungssteuerung umgewandelt werden. Da die eben genannten Konsistenzbedingungen am besten in einer formalen Spezifikation festgehalten werden sollten, wurden die in der Einleitung bereits erwähnten Funktionsbausteinadapter entwickelt, deren Konzept im Kapitel 3 eingeführt wird.

2.3.2 ViewPoint-Templates

Wenn in mehreren ViewPoints die gleiche Spezifikationssprache verwendet wird, dann wiederholen sich die Inhalte im Representation Style und im Work Plan der ViewPoints. In solchen Fällen ist es hilfreich, so genannte ViewPoint-Templates einzurichten, die schon teilweise oder vollständig ausgefüllte Slots enthalten. Diese

Templates können dann bei der Erzeugung neuer ViewPoints als Vorlage dienen und das mehrfache Eintragen gleicher Inhalte ersparen. Eine erster Ansatz zur Bereitstellung von geeigneten ViewPoint-Templates für die UML und die SPS-Sprachen wurde bereits in [EndHev 2002] vorgeschlagen.

3 Funktionsbausteinadapter

In diesem Kapitel soll das Konzept der Funktionsbausteinadapter vorgestellt werden, das es erlaubt, Funktionsbausteine in die UML zu integrieren. Das Ziel der Integration von Funktionsbausteinen ist hierbei, eine Schnittstelle zwischen objektorientierten Sprachen wie der UML und funktionsbausteinorientierten Sprachen wie der IEC 61131-3 zu schaffen.

Funktionsbausteine werden häufig in Sprachen wie Matlab/Simulink [Simulink 2002] zur Modellierung kontinuierlicher Systeme oder in Sprachen mit zyklischem Ausführungsverhalten wie in den Programmiersprachen für speicherprogrammierbare Steuerungen [IEC 61131] verwendet.

Im Abschnitt 3.1 soll zunächst die Problemstellung, die durch Funktionsbausteinadapter angesprochen wird, konkretisiert. Anschließend wird der für diese Problemstellung vorgeschlagene Lösungsansatz (Abschnitt 3.2) allgemein erläutert. Einen Vergleich zu existierenden Lösungsansätzen wird in Abschnitt 3.3 vorgenommen. Im Abschnitt 3.4 wird auf noch weitere Sprachen verwiesen, denen Funktionsbausteine oder Funktionsblöcke als Softwarekomponentenkonzept zugrunde liegen. Funktionsbausteinadapter lassen sich prinzipiell auf solche Sprachen anwenden.

Obwohl das Konzept der Funktionsbausteinadapter prinzipiell auf alle Arten von Funktionsbausteinen oder Funktionsblöcken anwendbar ist, soll es zunächst auf die IEC 61131-3 angewendet werden. Spezifisch für die IEC 61131-3 sind hierbei nur deren Datentypen, die deshalb in Abschnitt 3.5 erläutert werden. Im Abschnitt 3.6 wird schließlich das UML-Profil *FunctionBlockAdapters* vorgestellt. Hinweise zur Verifikation und zur Implementierung von Funktionsbausteinadaptern werden in den Abschnitten 3.7 und 3.8 gegeben. Eine Einordnung des Konzeptes der Funktionsbausteinadapter in das ViewPoint-Framework findet sich in Abschnitt 3.9.

3.1 Problemstellung

Funktionsbausteine können ohne zusätzliche Modellierungselemente nicht direkt mit Ports von UML-Klassen verbunden werden. Das liegt in der Art und Weise begründet, wie externe Schnittstellen von Funktionsbausteinen und von UML-Klassen zur Verfügung gestellt werden. Wenn Funktionsbausteine miteinander kommunizieren, geschieht das über eine zeitlich kontinuierliche Belegung von Schnittstellenvariablen mit Werten. Ereignisse können als sprungförmige Wertänderungen in solchen Variablen signalisiert werden. Aber auch die Zeitdauer, mit der eine Ausgangsvariable keine Wertänderung aufweist, kann eine Information für einen Funktionsbaustein darstellen, dessen Eingangsvariable mit dieser Ausgangsvariable verbunden ist. Die zeitlich veränderliche Belegung von Schnittstellenvariablen kann verschieden interpretiert werden. Deshalb ist es notwendig, die Bedeutung dieser Belegungen in zu-

sätzlichen Sprachen als ein Protokoll zu dokumentieren. Solche Sprachen können grafischer Art wie Zeitdiagramme oder Zustandsdiagramme oder textueller Art wie die natürliche Sprache sein. Im Weiteren soll ein auf diese Weise beschriebenes Protokoll *FB-Protokoll* heißen. Beispiele für FB-Protokolle finden sich in den Abschnitten 4.1.1.1 und 4.2.1.1. In Abbildung 2.12 wurde ein FB-Protokoll bereits in Form zweier ViewPoints dargestellt.

Demgegenüber kommunizieren UML-Objekte durch den Austausch von Nachrichten. Eine Nachricht beinhaltet eine Menge von Parametern und wird durch einen Namen und einen Typ gekennzeichnet. Wie bereits in Abschnitt 2.2.2 erläutert wurde, soll hier davon ausgegangen werden, dass Nachrichten ausschließlich über Ports versendet und empfangen werden. Eine vorgeschriebene Reihenfolge, die beim Nachrichtenaustausch eingehalten werden muss, kann in Form eines Protokollstatecharts beschrieben werden. Protokolle dieser Art sollen im Weiteren auch *UML-Protokoll* genannt werden, die beispielhaft in den Abschnitten 4.1.1.2 und 4.2.1.2 zu finden sind. In Abbildung 2.13 wurde ein UML-Protokoll bereits in Form zweier ViewPoints dargestellt.

Damit FB-Protokolle auf UML-Protokolle abgebildet werden können, müssen Ereignisse in den zeitlichen Belegungen von Schnittstellenvariablen von Funktionsbausteinen in einen (eventuell datenbehafteten) Nachrichtenaustausch über UML-Ports umgesetzt werden. Das soll im Folgenden durch einige vereinfachte Beispiele verdeutlicht werden.

Im ersten Beispiel soll eine Ausgangsvariable eines Funktionsbausteins beobachtet werden. In Abbildung 3.1 rechts ist ein Funktionsbaustein mit einer Ausgangsvariablen *DO* vom Typ *DataType1* dargestellt. Würden zwei Funktionsbausteine miteinander kommunizieren, dann könnte man diese Variable einfach mit einer geeigneten Eingangsvariablen des anderen Funktionsbausteins verbinden. Sollen aber stattdessen Nachrichten, die den Wert der Variablen als Parameter beinhalten, über ein Port versendet werden, dann müssen diese Nachrichten explizit generiert werden. Außerdem müssen die Zeitpunkte für die Generierung dieser Nachrichten festgelegt werden.

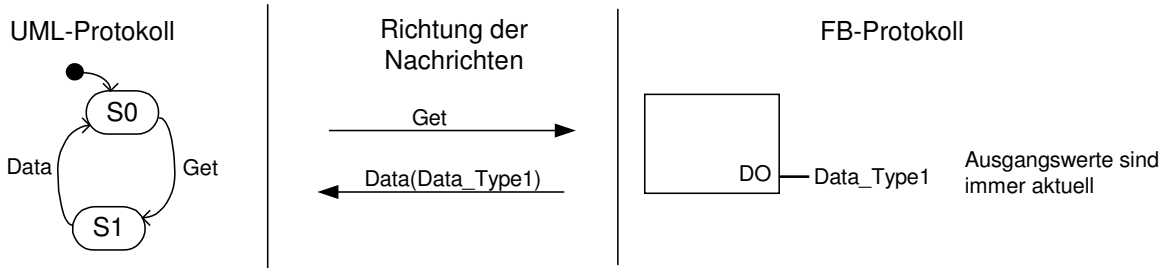


Abbildung 3.1 Beobachten einer Variablen

In Abbildung 3.1 links ist ein UML-Protokoll schematisch vereinfacht dargestellt, nach dem das Eintreffen einer Nachricht *Get* das Versenden der Nachricht *Data*, die den Datenwert von *DO* enthält, auslöst. Die Nachricht *Get* könnte zum Beispiel durch einen periodischen Zeitgeber ausgelöst werden oder durch eine explizite Benutzereingabe in einer Benutzeroberfläche. Eine andere Möglichkeit, bei der die Nachricht *Get* nicht benötigt wird, ist die Generierung der Nachricht *Data* bei jeder Veränderung von *DO*. Beide Möglichkeiten sollten von Funktionsbausteinadaptern unterstützt werden.

Wenn ein Funktionsbaustein explizit die Zeitpunkte bestimmen möchte, zu denen der Wert der Ausgangsvariable in einer Nachricht versendet werden soll, kann das durch eine zusätzliche Ausgangsvariable geschehen (Abbildung 3.2). Bei jeder positiven oder negativen Flanke von *Q* in Abbildung 3.2 wird eine Nachricht *Signal* mit dem aktuellen Wert von *DO* versendet.

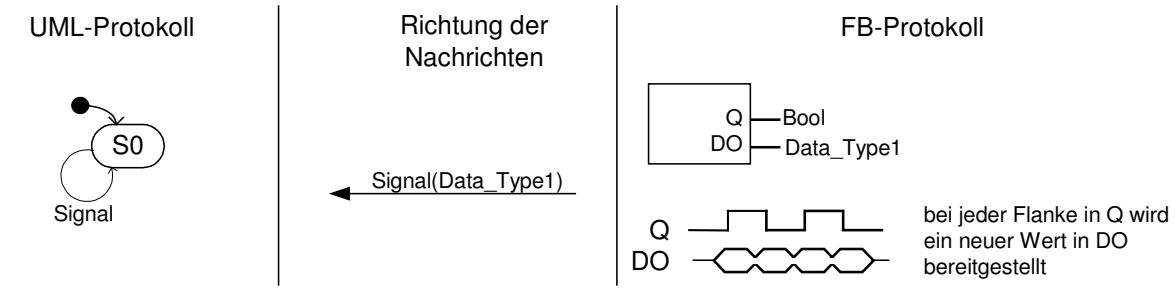


Abbildung 3.2 Toggle-Variable

Selbstverständlich könnte das Beispiel aus Abbildung 3.2 auch so formuliert werden, dass nur bei einer positiven oder nur bei einer negativen Flanke in *Q* eine Nachricht gesendet wird.

Ein ähnliches Beispiel ist in Abbildung 3.3 dargestellt. Dort soll beim Überschreiten eines Schwellwertes eine Nachricht *Oberhalb* und beim Unterschreiten eine Nachricht *Unterhalb* versendet werden. Das Versenden von *Oberhalb* ist also mit einer positiven Flanke im Zeitdiagramm in Abbildung 3.3 rechts verknüpft. *Unterhalb* ist entsprechend mit einer negativen Flanke von $DO \leq 234$ verbunden.

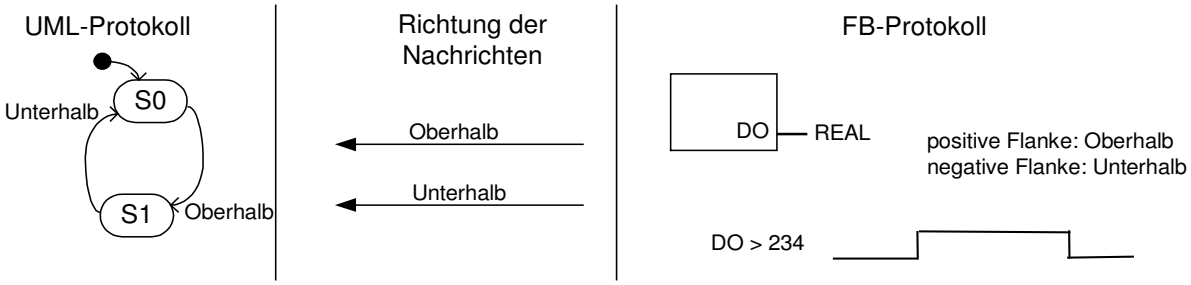


Abbildung 3.3 Beobachtung von Schwellwerten

In Abbildung 3.4 ist ein FB-Protokoll dargestellt, bei dem ein Funktionsbaustein mit einer positiven Flanke in *Q* Daten in *DO* so lange zur Verfügung stellt, bis nach einer positiven Flanke in *In* neue Daten in *DI* vorhanden sind. Erst nachdem *Q* und *In* wieder zurückgesetzt werden, wird in *Q* eine erneute positive Flanke erlaubt. Dieses FB-Protokoll soll auf ein UML-Protokoll abgebildet werden, bei dem zuerst eine Nachricht *Request* mit einem Parameter vom Typ *Data_Type1* empfangen werden muss, bevor mit der Nachricht *Answer*, die einen Parameter vom Typ *Data_Type2* enthält, geantwortet wird.

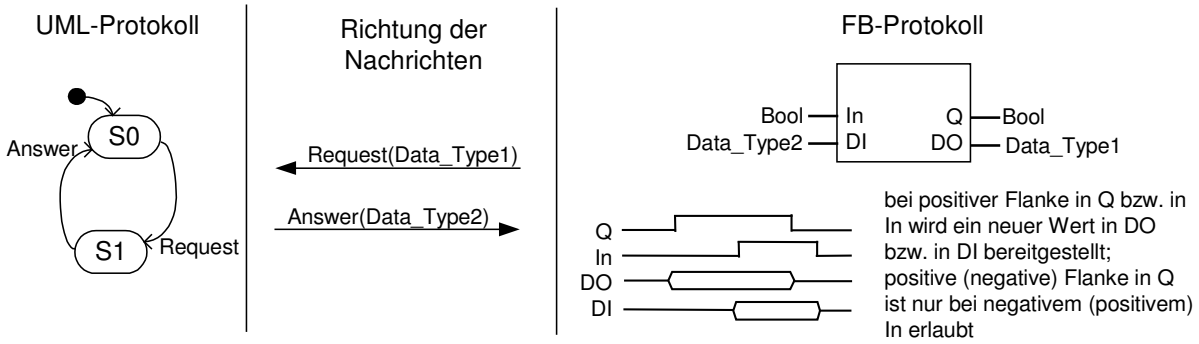


Abbildung 3.4 Anfrage-Antwort Protokoll

Zu einem ähnlichen Protokoll gelangt man, wenn die Anfrage zuerst an den Funktionsbaustein gerichtet wird und der Funktionsbaustein auf die Anfrage antwortet. Dazu müssen nur die Richtungen der Nachrichten und die Reihenfolgen der Flanken in *Q* und *In* vertauscht werden. Ein solches Protokoll wird im Anwendungsbeispiel *Motion Control* im Abschnitt 4.1 eingesetzt.

3.2 Problemlösung

Um ein FB-Protokoll auf ein UML-Protokoll abzubilden, soll eine spezielle Art von Protokolladaptern eingesetzt werden, die für unterschiedlichste Anforderungen konfigurierbar sind. Diese Protokolladapter sollen *Funktionsbausteinadapter* genannt werden. Eine wichtige Anforderung für Funktionsbausteinadapter ist, dass die Protokollumsetzung auf einer plattformunabhängigen Ebene beschrieben werden soll. Außerdem sollte es für SPS-Entwickler nicht notwendig sein, sich in die gesamte UML einarbeiten zu müssen, um das Verhalten eines Funktionsbausteinadapters zu verstehen.

3.2.1 Schnittstellen von Funktionsbausteinadaptern

Funktionsbausteinadapter benötigen zum einen Schnittstellen in Form von Ports und zum anderen Schnittstellenvariablen wie sie auch von Funktionsbausteinen verwendet werden. In Abbildung 3.5 ist ein Funktionsbausteinadapter *FBA* als UML-Klasse dargestellt. Neu in dieser Darstellung sind Eingangsvariablen und Ausgangsvariab-

len, die der Darstellungsweise bei Funktionsbausteinen nachgebildet wurden. Schnittstellenvariablen von Funktionsbausteinen können nicht einfach als Attribute von UML-Klassen betrachtet werden. Dafür gibt es zwei Gründe. Der erste Grund ist der Zugriffsschutz. Öffentliche Attribute von UML-Klassen können sowohl von innerhalb der UML-Klasse wie auch von außerhalb gelesen und verändert werden. Deklariert man sie als *readonly*, dann dürfen sie sowohl von außerhalb als auch von innerhalb einer Klasse nicht verändert werden. Eingangsvariablen von Funktionsbausteinen dürfen demgegenüber von außerhalb verändert, aber von innerhalb eines Funktionsbausteins nur gelesen werden. Ausgangsvariablen von Funktionsbausteinen dürfen von außerhalb nur gelesen und von innerhalb gelesen und verändert werden.

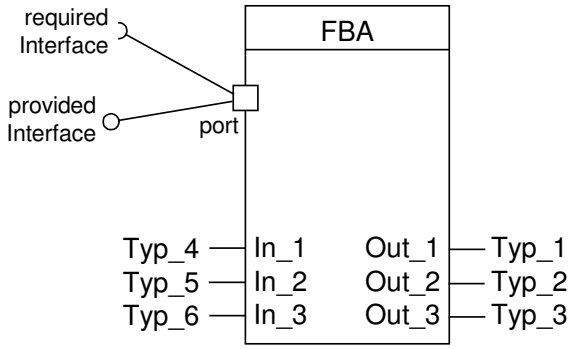


Abbildung 3.5 Darstellung eines Funktionsbausteinadapters im Klassendiagramm

In der Objektorientierung würden sich die aus dem ersten Grund resultierenden Anforderungen dadurch nachbilden lassen, dass man die Attribute selbst als nichtöffentlich deklariert, aber öffentliche Zugriffsoperationen mit Eingabe- und Rückgabeparametern bereitstellt. Der zweite Grund für die Nichteignung von UML-Attributen als Schnittstellenvariablen erklärt, warum Zugriffsoperationen allein auch nicht ausreichend sind. Ein großer Vorteil der Verwendung von Funktionsbausteinen ist, dass man erst bei Instanziierung festlegen muss, mit welchen Schnittstellenvariablen anderer Funktionsbausteine die eigenen Schnittstellenvariablen verbunden werden sollen (oder ob sie mit konstanten Werten belegt werden sollen). Diese Festlegung erfolgt zumeist in Form der Funktionsbausteinsprache (siehe auch Abbildung 2.6 bzw. Abschnitt 2.1.3). Um eine ähnlich komfortable grafische Darstellungsweise wie die in der Funktionsbausteinsprache zu erhalten, werden Schnittstellenvariablen in Abschnitt 3.6 als Stereotyp von Ports eingeführt.

3.2.2 Verhaltensbeschreibung von Funktionsbausteinadaptern

Neben den verschiedenen Schnittstellen zu UML-Klassen und zu Funktionsbausteinen benötigt ein Funktionsbausteinadapter auch eine Beschreibung seines dynamischen Verhaltens. Dieses Verhalten bestimmt die Abbildung der UML-Protokolle auf die FB-Protokolle, die mit den Schnittstellen verbunden sind. Dabei muss das Eintref-

fen von Nachrichten über ein Port in Änderungsereignisse von Schnittstellenvariablen umgesetzt werden. Umgekehrt resultiert aus Änderungsereignissen in Eingangsvariablen das Senden von Nachrichten über Ports. Die Anforderungen an die Protokollumsetzung können sehr vielfältig und unter Umständen auch sehr komplex sein. Einige Beispiele wurden bereits im Abschnitt 3.1 erläutert. In realen Systemen kommen oft Kombinationen dieser Beispiele oder völlig neue Anforderungen vor.

Das dynamische Verhalten eines Funktionsbausteinadapters kann entweder durch die FBA-Sprache oder wie in der UML üblich durch Statecharts beschrieben werden. Die FBA-Sprache ist eine textuelle Sprache, die speziell für Funktionsbausteinadapter entwickelt wurde und deren Komplexität weitaus geringer als die von Statecharts ist. Sie wurde so konzipiert, dass die häufigsten Anforderungen an die Protokollumsetzung in einfacher und übersichtlicher Form umgesetzt werden können. Als wesentliche Bestandteile enthält die Sprache

- Befehle zum Senden von Nachrichten über Ports,
- Befehle zum Warten auf Nachrichten in Ports oder auf Änderungsereignisse in Schnittstellenvariablen,
- Befehle zum Zeitverzögern und
- Ausdrücke für Zuweisungen und zum Zugriff auf Schnittstellenvariablen und Parameter von Nachrichten.

Ein Vorteil bei der Verwendung der FBA-Sprache ist, dass SPS-Entwickler sich nicht in die komplexere Syntax von Statecharts einarbeiten müssen. Weiterhin besteht weniger die Gefahr, dass einem Funktionsbausteinadapter Verhalten zugewiesen wird, das über die eigentliche Aufgabe eines Protokolladapters hinaus geht.

Falls die Anforderungen an die Protokollumsetzung so komplex sind, dass die Mittel der FBA-Sprache nicht ausreichend sind, muss die Protokollumsetzung vollständig durch Statecharts beschrieben werden. In den meisten Fällen resultieren solche komplexen Anforderungen aber aus der Tatsache, dass über die Protokollumsetzung hinausgehende Anforderungen formuliert wurden. So fehlen in der FBA-Sprache zum Beispiel Befehle für eine Verarbeitung von Daten, die eventuell in Nachrichten enthalten sind. Eine solche Datenverarbeitung ist besser in Operationen von UML-Klassen aufgehoben, denen die Daten einer Nachricht als Parameter übergeben werden können.

Zur weiteren Vereinfachung wurde in Funktionsbausteinadaptern ein Konzept zur Modularisierung der Verhaltensbeschreibung eingeführt, durch das das Gesamtverhalten in einzelne *Übersetzungen* aufgeteilt werden kann. In einer Übersetzung kann zum Beispiel das Verhalten zusammengefasst werden, das für die Umsetzung einer Nachricht in Änderungsereignisse von Schnittstellenvariablen nötig ist. Durch diese

Modularisierung lässt sich wiederum die Protokollumsetzung in weniger komplexe Teilprobleme untergliedern, die sich für eine Beschreibung mit Hilfe der FBA-Sprache eignen.

Ein Funktionsbausteinadapter ist mit Standardverhalten ausgestattet, das nicht explizit von einem Entwickler angegeben werden muss. Dazu gehört die Speicherung der Werte von Schnittstellenvariablen in speziellen Attributen und das Scheduling der Übersetzungen eines Funktionsbausteinadapters. Eine vollständige Übersicht über Syntax und Semantik von Funktionsbausteinadaptern liefert Abschnitt 3.6.

3.2.3 Konfigurationsmöglichkeiten von Funktionsbausteinadaptern

Zu den Konfigurationsmöglichkeiten von Funktionsbausteinadaptern zählen auf Klassenebene folgende Eigenschaften:

- Anzahl der Ports, deren Kardinalität und Protokolle
- Anzahl der Schnittstellenvariablen und deren Datentypen
- Anzahl der Übersetzungen und deren Konfiguration

Auf die Konfiguration der Übersetzungen wird am Ende dieses Abschnittes eingegangen. In Abbildung 3.5 wurde bereits ein Beispiel für einen Funktionsbausteinadapter angegeben. Er enthält ein Port und sechs Schnittstellenvariablen. Die Entscheidung darüber, mit welchen anderen Funktionsbausteininstanzen und UML-Instanzen der Funktionsbausteinadapter verbunden wird, muss erst auf Instanzebene festgelegt werden. Im einfachsten Fall ist es wie in Abbildung 3.6 der Fall, dass alle Schnittstellenvariablen mit einer Funktionsbausteininstanz verbunden sind. Da das Port in diesem Beispiel die Kardinalität Eins hatte, konnte es nur mit einer anderen Instanz verbunden werden.

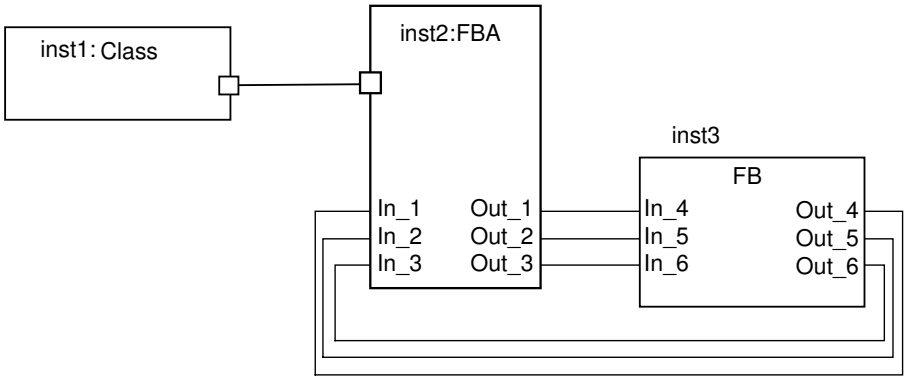


Abbildung 3.6 Strukturdiagramm für einen Funktionsbausteinadapter

Wenn alle Schnittstellenvariablen einem FB-Protokoll angehören, kann das Verhalten durch nur eine Übersetzung beschrieben werden. Es ist aber auch möglich, das Verhalten auf mehrere Übersetzungen aufzuteilen.

In Abbildung 3.7 ist ein Funktionsbausteinadapter mit drei Ports dargestellt. *Port1* und *Port3* unterstützen das gleiche Protokoll. Dieses Protokoll soll als das UML-Protokoll, das in Abbildung 3.1 dargestellt wird, angenommen werden. Die Schnittstellenvariablen sollen diesmal nicht alle dem gleichen FB-Protokoll zugeordnet werden. Die Eingangsvariable *In_1* soll immer einen aktuellen Eingangswert beinhalten, was dem FB-Protokoll aus Abbildung 3.1 entspricht. Aus den Anforderungen der mit diesem Funktionsbausteinadapter verbundenen Anwendung geht hervor, dass *Port1* und auch *Port3* an dieser Eingangsvariablen interessiert sind. Die restlichen Schnittstellenvariablen sollen einem anderen FB-Protokoll angehören.

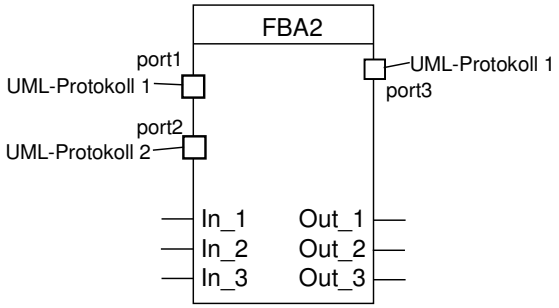


Abbildung 3.7 Komplexeres Beispiel für einen Funktionsbausteinadapter

Eine mögliche Zusammenschaltung mit anderen Instanzen zeigt das Strukturdiagramm aus Abbildung 3.8. Die UML-Instanzen *Inst1* und *Inst5* sind mit *Port1* und *Port3* verbunden und sollen darüber die aktuellen Werte der Ausgangsvariablen von *Inst4* erhalten, die mit *In_1* verbunden ist.

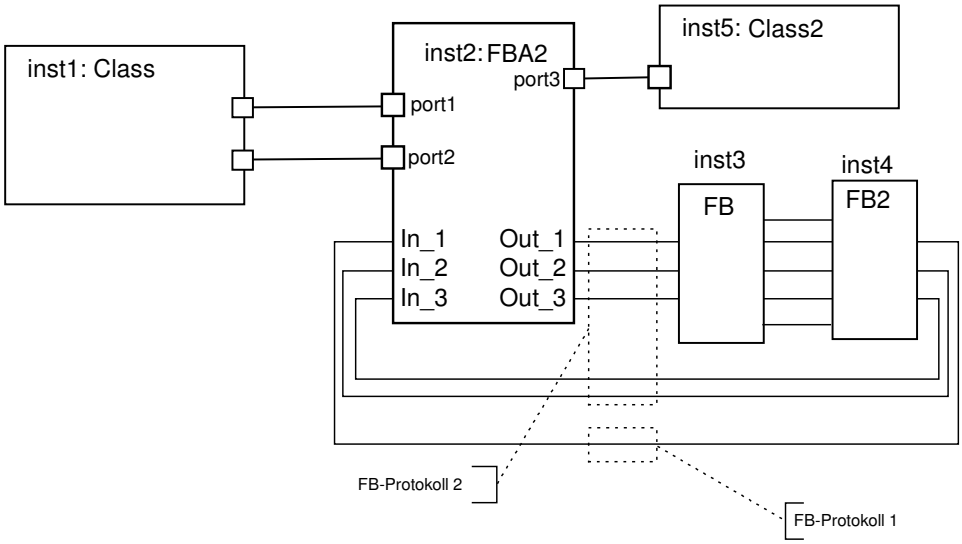


Abbildung 3.8 Strukturdiagramm für einen komplexeren Funktionsbausteinadapter

Um die beiden Ports *Port1* und *Port3* zu bedienen, ist es sinnvoll, eine Übersetzung pro Port anzulegen. Die Übersetzung für *Port1* hat als auslösendes Ereignis die Nachricht *Get*. Im Verlauf der Übersetzung muss eine Nachricht *Data* erzeugt und mit dem Datenwert aus *In_1* ausgefüllt werden.

```
trigger: (s1: port1.Get)
signals: (s1: port1.Get, s2: port1.Data)
translationBody: {
  s2 := In_1;
  send(s2);
}
```

Unter *Signals* werden alle UML-Nachrichten aufgeführt, die verarbeitet werden müssen. Im *TranslationBody* findet sich schließlich die Beschreibung der Übersetzung in Syntax der FBA-Sprache. Trigger, Signals und TranslationBody sind TaggedValues von Übersetzungen, die im Abschnitt 3.6.7 erläutert werden.

Die zweite Übersetzung unterscheidet sich von der ersten nur dadurch, dass die Nachrichten von *Port3* verarbeitet werden.

```
trigger: (s1: port3.Get)
signals: (s1: port3.Get, s2: port3.Data)
translationBody: {
  s2 := In_1;
  send(s2);
}
```

Neben den eben vorgestellten Übersetzungen fehlt noch mindestens eine Übersetzung für das *Port2*, die an dieser Stelle aber nicht erläutert werden soll. Da die beiden Übersetzungen für *Port1* und *Port3* den Zustand der Schnittstellenvariablen nicht verändern und auch keine Nachrichten über *Port2* verschicken, können sie nebenläufig zum restlichen Verhalten des Funktionsbausteinadapters ablaufen. Dem kann durch ein weiteres TaggedValue *IsOrthogonal* Rechnung getragen werden (siehe Abschnitt 3.6.7).

Wenn eine Übersetzung durch ein Änderungsereignis eines Booleschen Ausdrucks ausgelöst werden soll, dann muss dieser Ausdruck als Trigger angegeben werden. Unter der Annahme, dass eine Eingangsvariable *DI* eines Funktionsbausteinadapters dem FB-Protokoll aus Abbildung 3.3 entspricht, können folgende Übersetzungen formuliert werden:

```
trigger: (DI > 234)
signals: (s1: einPort.Oberhalb)
translationBody: {
  send(s1);
}

trigger: (DI <= 234)
signals: (s1: einPort.Unterhalb)
translationBody: {
  send(s1);
}
```

Angenommen, ein Funktionsbausteinadapter hat zwei Eingangsvariablen *In* und *DI*, die dem FB-Protokoll von *Q* und *DO* aus Abbildung 3.2 entsprechen, dann kann eine dazugehörige Übersetzung folgendermaßen aussehen:

```
trigger: (In.ValueChanged)
signals: (s1: einPort.Signal)
translationBody: {
  s1 := DI;
  send(s1);
}
```

Die Schreibweise mit dem Suffix *.ValueChanged* kennzeichnet eine beliebige Wertänderung in der jeweiligen Eingangsvariable (siehe auch Schlussbemerkung von Abschnitt 3.6.7.3). Diese Schreibweise kann auf jeden Datentyp angewendet werden. Soll zum Beispiel das FB-Protokoll aus Abbildung 3.1 auf das UML-Protokoll aus Abbildung 3.2 umgesetzt werden, kann das wie folgt formuliert werden, wenn die Eingangsvariable des Funktionsbausteinadapters *DI* heißt:

```
trigger: (DI.ValueChanged)
signals: (s1: einPort.Signal)
translationBody: {
  s1 := DI;
  send(s1);
}
```

Bei dieser Übersetzung können niemals zwei gleiche Werte hintereinander übermittelt werden. Ob das gewünscht ist oder nicht, muss aus den Anforderungen für die Protokollumsetzung ermittelt werden.

Um die Liste von Konfigurationsmöglichkeiten zu vervollständigen, sollen an dieser Stelle die Konfigurationsmöglichkeiten von Übersetzungen angegeben werden:

- Angabe eines auslösenden Ereignisses
- Menge von Nachrichten, die von der Übersetzung verarbeitet und gesendet werden
- Angabe, ob die Übersetzung nebenläufig zu anderen Übersetzungen stattfindet
- Verhaltensbeschreibung in Form der FBA-Sprache
- Verhaltensbeschreibung zur Ausnahmebehandlung bei Zeitschrankenüberschreitungen

Eine detaillierte Beschreibung dieser Konfigurationsmöglichkeiten ist in Abschnitt 3.6.7 zu finden.

3.3 Vergleich zu anderen Lösungsansätzen

In diesem Abschnitt werden Ansätze zur Integration der UML mit SPS-Sprachen vorgestellt, die parallel zu dem in dieser Arbeit gewählten Ansatz entstanden sind oder bereits vorher existierten.

Da Funktionsbausteine bzw. Funktionsblöcke häufig in Beschreibungssprachen für kontinuierliche Systeme zu finden sind und die UML eine rein ereignisdiskrete Se-

mantik aufweist, soll in Abschnitt 3.3.2 eine Beziehung zu hybriden Systemen hergestellt werden.

3.3.1 Existierende Ansätze zur Integration der UML mit der IEC 61131-3

Bisher sind noch wenig Ansätze zur Integration der UML mit Sprachen für speicherprogrammierbare Steuerungen bekannt. Ein in [BauEng 2002] veröffentlichter Ansatz beruht auf der Integration von Statecharts der UML mit der Ablaufsprache der IEC 61131-3. Da Statecharts nicht nur in der UML zur Verhaltensbeschreibung von Objekten verwendet werden, ist dieser Ansatz nicht auf die UML beschränkt. Ein Statechart wird als übergeordnete Steuerung eingesetzt, durch die bestimmt wird, welches Ablaufsprachediagramm in welchem Zustand gestartet wird. Auf diese Weise entsteht eine neue visuelle Sprache zur Verhaltensmodellierung, deren Komplexität höher ist, als die der Ablaufsprache oder als die von Statecharts. Das kann als Nachteil dieses Ansatzes betrachtet werden.

Im Gegensatz dazu müssen durch das Konzept der Funktionsbausteinadapter nicht die Verhaltensbeschreibungen von Funktionsbausteinen und UML-Klassen vermischt werden. Die Komplexität der Sprache zur Verhaltensbeschreibung von Funktionsbausteinadaptern ist wesentlich geringer als die von Funktionsbausteinen (z.B. Ablaufsprache) oder UML-Klassen (Statecharts).

3.3.2 Hybride Systeme

Der Begriff der hybriden Systeme wird in verschiedenen Forschungszweigen verwendet, wenn Modelle, die auf verschiedenen Ansätzen beruhen, miteinander kombiniert werden. Im Bereich der Automatisierungstechnik versteht man unter einem hybriden System zumeist die Kombination aus mathematischen Modellen, denen eine kontinuierliche Zeit zugrunde liegt (z. B. Differenzialgleichungssysteme), mit diskreten mathematischen Modellen (z. B. der Automatentheorie). In den meisten Ansätzen für hybride Systeme werden visuelle, anwenderfreundliche Sprachen kombiniert. In [Schn 1999] werden kontinuierliche Modelle hauptsächlich in Petrinetze integriert. Andere Ansätze vereinigen Statecharts mit kontinuierlichen Modellen [FilBor 2001], [BicRadSch 2001].

Betrachtet man Funktionsbausteine als Modell mit kontinuierlicher Zeitbasis, dann zielen Funktionsbausteinadapter auf eine ähnliche Problematik wie hybride Systeme ab. Die Lösungsansätze lassen sich aber deutlich voneinander unterscheiden: Während bei hybriden Systemen normalerweise diskrete Verhaltensmodelle wie Petrinetze oder Statecharts um kontinuierliche Verhaltensmodelle erweitert werden, basieren Funktionsbausteinadapter auf einem komponentenbasierten Ansatz. Unabhängig davon, welches konkrete diskrete Verhaltensmodell einer rein diskreten Komponente

zugrunde liegt, soll diese mit einer rein kontinuierlichen Komponente, der ein beliebiges kontinuierliches Verhaltensmodell innewohnt, verbunden werden.

Der Funktionsbausteinadapter selbst basiert auf einer rein ereignis-diskreten Semantik. Diese Vereinfachung beruht auf der Tatsache, dass sich die Schnittstellen zwischen kontinuierlichem und diskretem System in den für diese Arbeit betrachteten Fällen immer auf relativ einfache Muster wie eine Schwellwertüberschreitung, eine Zeitschrankenüberschreitung, eine Wertänderung oder einer Kombination daraus zurückführen ließen [HeShGrTr 2002], [Hev 2003], [HevTra 2001a], [HevTra 2001e], [HevTra 2001b], [HevTra 2001c]. Im Funktionsbausteinadapter werden solche Ereignisse genutzt, um aus den aktuellen Zuständen der Schnittstellenvariablen von Funktionsbausteinen Nachrichten zu generieren, die über UML-Ports gesendet werden können. Auf diese Weise werden das kontinuierliche System und das diskrete System vollständig voneinander entkoppelt. Das diskrete System benötigt kein Wissen über die Bedeutung der Zustände der Variablen des kontinuierlichen Systems und das kontinuierliche System muss kein Port-Konzept enthalten.

Ein Ansatz hybrider Systeme, bei dem diskrete Steuerungen, die durch Statecharts oder die Ablaufsprache beschrieben werden, über ein Modul-Konzept mit kontinuierlichen Modellen verbunden werden, findet sich in [OtReEnMo 2000]. Die Schnittstellen der dort verwendeten Module werden wie bei Funktionsblöcken aus Eingangs- und Ausgangsgrößen gebildet. Ein Port-Konzept ist nicht enthalten. Funktionsbausteinadapter wären dort eine sinnvolle Ergänzung, wenn mit Port-basierten Modellen kommuniziert werden soll.

3.4 Andere funktionsbausteinorientierte Sprachen

Neben der IEC 61131-3 gibt es weitere Sprachen, denen Funktionsbausteine bzw. Funktionsblöcke als Konzept für Softwarekomponenten zugrunde liegen. Allgemeinere Definitionen des Funktionsbaustein-Konzeptes wurden in [ISP 1993] und [IEC 61499] festgelegt. Im deutschsprachigen Raum unterscheidet man nach [NeGrLuSi 1998] zwischen den Begriffen Funktionsbaustein und Funktionsblock. Während der Begriff Funktionsbaustein meistens im Zusammenhang mit der IEC 61131-3 verwendet wird, bezeichnet der Begriff Funktionsblock Softwarekomponenten in allgemeinerer Form.

In der [IEC 61499] wurden die Funktionsbausteine der IEC 61131-3 aufgegriffen und um neue Konzepte erweitert. So kann man bei Funktionsbausteinen der IEC 61499 zwischen Schnittstellenvariablen unterscheiden, die Ereignisse signalisieren und solchen, die Daten übertragen. Diese Trennung wäre auch im Zusammenhang mit Funktionsbausteinadaptern wünschenswert, da sie hier von der Interpretation des Softwareentwicklers abhängig ist. Da die ereignisübermittelnden Schnittstellenvariablen

der IEC 61499 von außerhalb eine Funktionsblockes aber auch als einfache Boolesche Variablen betrachtet werden können, lassen sich Funktionsbausteinadapter direkt auf die IEC 61499 anwenden.

[Simulink 2002] enthält eine funktionsblockorientierte Sprache zur mathematischen Modellierung von kontinuierlichen Systemen. Die darin enthaltenen Funktionsblöcke entsprechen Übertragungsgliedern mit Eingangsgrößen, Ausgangsgrößen und internen Zustandsgrößen. Das Verhalten wird durch Differenzialgleichungssysteme beschrieben. Die Eingangs- und Ausgangsgrößen lassen sich ähnlich wie die Schnittstellenvariablen von Funktionsbausteinen der IEC 61131-3 behandeln, sodass man prinzipiell Simulinkblöcke über Funktionsbausteinadapter mit UML-Modellen verknüpfen kann. Lediglich die Datentypen und deren Wertebereiche müssten angepasst werden.

3.5 Integration von Datentypen der IEC 61131-3 in die UML

Damit die Sprachen der IEC 61131-3 in die UML integriert werden können, müssen die in der IEC 61131-3 definierten Datentypen in der UML modelliert werden. Für diesen Zweck können die in der UML enthaltenen Metaklassen, die in Abbildung 3.9 dargestellt sind, verwendet werden.

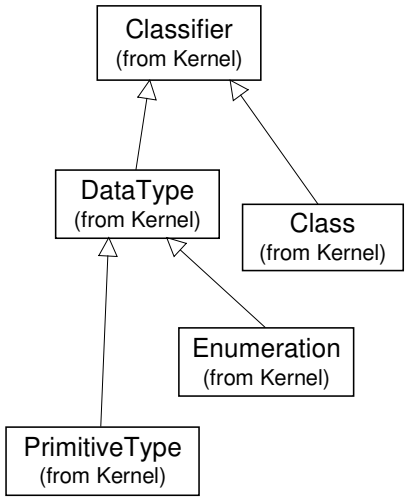


Abbildung 3.9 Stereotypen für SPS-Datentypen (Metamodell, M2 Ebene)

Als allgemeine Metaklasse für Datentypen wurde in der UML die Klasse *DataType* definiert [UMLDataType]. UML-Datentypen können ebenso wie UML-Klassen Attribute und Operationen deklarieren. Im Unterschied zu UML-Klassen haben die Instanzen von UML-Datentypen aber keine Identität. Für einfache, außerhalb der UML definierte Datentypen wie die elementaren Datentypen der IEC 61131-3 ist die Metaklasse *PrimitiveType* ausreichend. Werte von Datentypen des Typs PrimitiveType werden als atomar betrachtet. Benutzerdefinierte zusammengesetzte Datentypen der

IEC 61131-3 sollten als *DataType* nachgebildet werden. Benutzerdefinierte Aufzählungstypen der IEC 61131-3 können als *Enumeration* modelliert werden.

Durch die generischen Datentypen der IEC 61131-3 werden die Datentypen in eine Hierarchie aufgegliedert, die in Abschnitt 2.1.2.3 bereits erläutert wurde. Diese Hierarchie lässt sich sehr gut durch Vererbungsbeziehungen nachbilden (Abbildung 3.10).

PLC_ANY ist der allgemeinste Datentyp. Er entspricht dem generischen Typ *ANY* der IEC 61131-3. Alle Datentypen der IEC 61131-3 gehören automatisch diesem Typ an [IECDatenTypen 1993]. Das gilt auch für benutzerdefinierte Typen. In der UML sollten benutzerdefinierte Typen deshalb als Untertyp von *PLC_ANY_DERIVED* deklariert werden.

Im Weiteren sollen alle elementaren und generischen Datentypen der IEC 61131-3 den Präfix „PLC_“ erhalten, damit sie nicht mit Datentypen anderer Programmiersprachen verwechselt werden können. Die Namen von benutzerdefinierten Typen werden nicht mit diesem Präfix versehen.

3.5.1 Elementare Datentypen der IEC 61131-3

In diesem Abschnitt sollen die elementaren Datentypen der IEC 61131-3 durch Vererbungsbeziehungen den entsprechenden generischen Datentypen zugeordnet und als *PrimitiveType* klassifiziert werden. *PLC_TIME* ist direkt unter *PLC_ANY_MAGNITUDE* angesiedelt, wie aus Abbildung 3.10 zu erkennen ist.

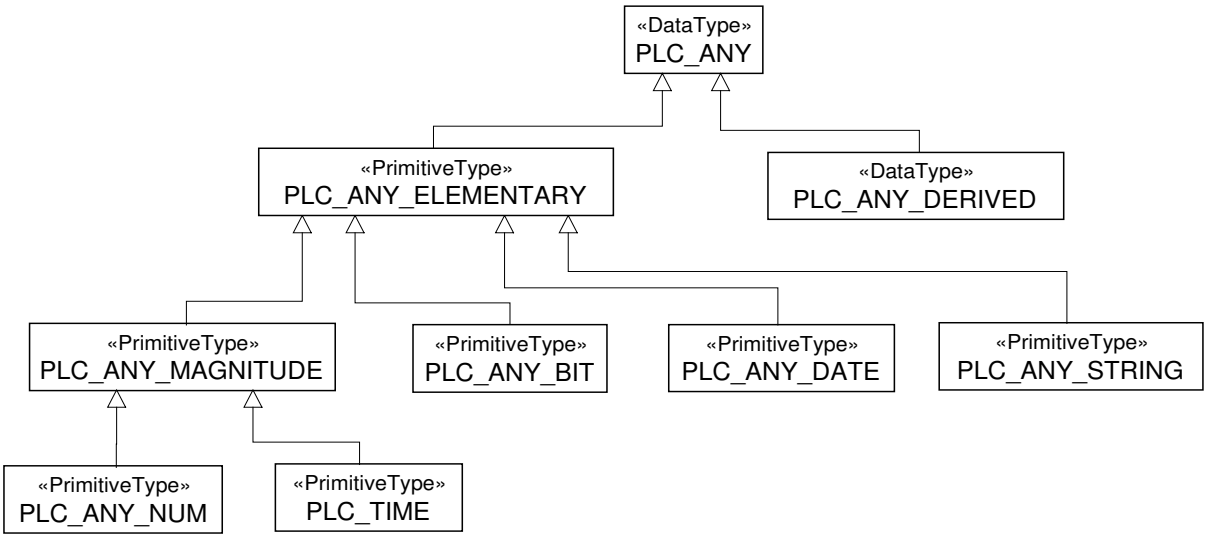


Abbildung 3.10 Generische Datentypen der IEC 61131-3 (Modell, M1 Ebene)

PLC_ANY_STRING besitzt zwei Unterklassen (Abbildung 3.11)

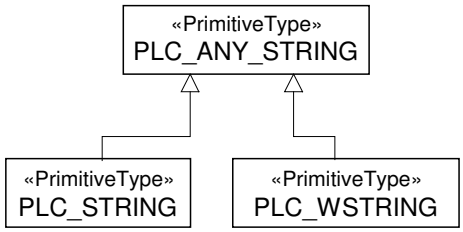


Abbildung 3.11 Elementare Datentypen der IEC 61131-3

Datentypen für ganze Zahlen werden *ANY_INT* zugeordnet (Abbildung 3.12).

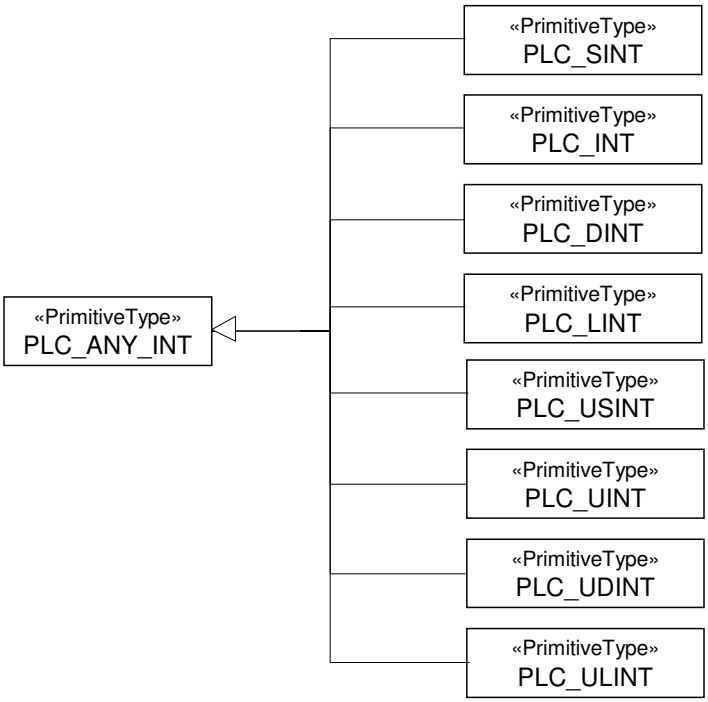


Abbildung 3.12 Typen für ganzzahlige Wertebereiche

Fließkommatypen sind in Abbildung 3.13 dargestellt.

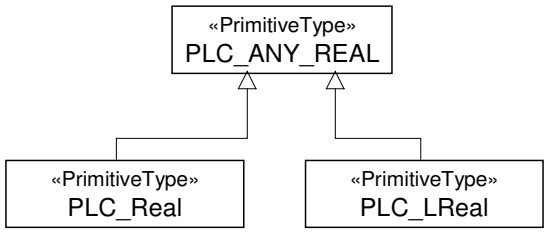


Abbildung 3.13 Typen für reelle Wertebereiche

Datentypen für bitadressierbare Variablen stellt Abbildung 3.14 dar.

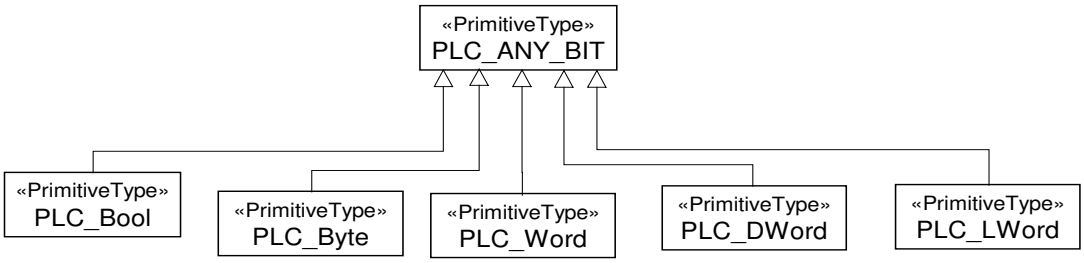


Abbildung 3.14 Datentypen für Bitvariablen

Werte von Variablen für Datum und Uhrzeit werden in der IEC 61131-3 durch strukturierte Zeichenketten beschrieben. Da diese Strukturierung aber nicht aus Attributen, sondern nur aus formatiertem Text erreicht wird, sollen die entsprechenden Datentypen ebenfalls als *PrimitiveType* definiert werden (Abbildung 3.15).

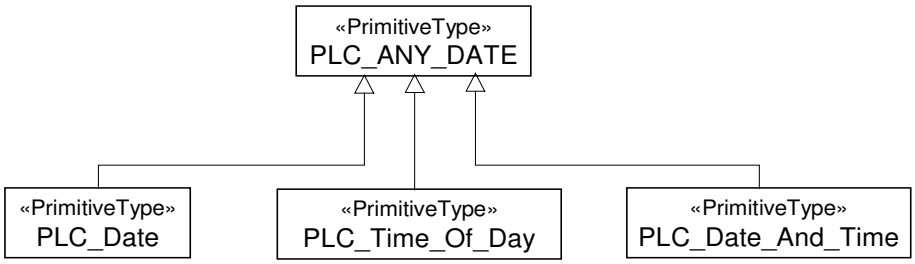


Abbildung 3.15 Datentypen für Datum und Tageszeit

Für die Syntax von Werten für elementare Datentypen gelten die Regeln aus der IEC 61131-3. Sollen Wertebereiche von Datentypen eingeschränkt werden, kann das mit Hilfe der OCL formuliert werden, z. B.:

```
context PLC_SINT inv: (-128 <= self) & (self < 128)
```

3.5.2 Abgeleitete Datentypen der IEC 61131-3

Es ist in der IEC 61131-3 möglich, neue Datentypen von elementaren Datentypen abzuleiten, um weitere Wertebereichseinschränkungen vorzunehmen. In der UML kann dies ebenfalls durch OCL-Bedingungen nachgebildet werden (siehe oben).

Soll zum Beispiel ein Temperatur-Datentyp gebildet werden, könnte dieser in der IEC 61131-3 so aussehen:

```
TYPE TemperaturTyp: SINT (-40..70); END_TYPE
```

TemperaturTyp muss dann in der UML als Untertyp von *PLC_SINT* definiert werden (Abbildung 3.16 rechts). Die Einschränkung des Wertebereiches erfolgt analog zum Beispiel im vorherigen Abschnitt:

```
context TemperaturTyp inv: (-40 <= self) & (self <= 70)
```

Ein Aufzählungstyp z. B.:

```
TYPE WochentagTyp: (Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag,
                    Sonntag);
END_TYPE
```

wird in der UML als *Enumeration* und als Untertyp von *PLC_ANY_DERIVED* definiert (Abbildung 3.16 links).

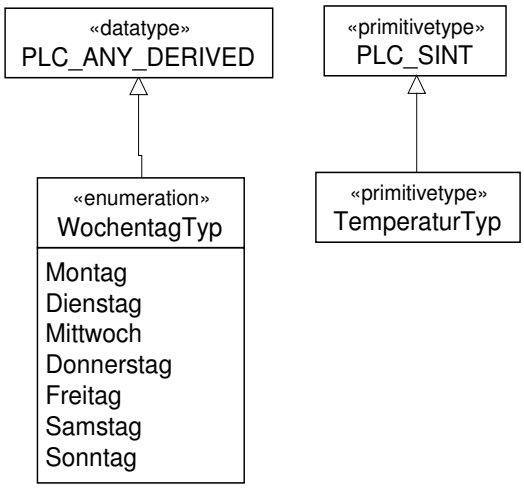


Abbildung 3.16 Einfache abgeleitete (benutzerdefinierte) Datentypen

Benutzerdefinierte und zusammengesetzte Datentypen wie z. B.:

```
TYPE Pos_info
 STRUCT
  Pos_Status: BOOL;
  Part_on: Part_info;
 END_STRUCT
END_TYPE

TYPE Part_info
 STRUCT
  Part_Type: INT;
  Part_Sender: INT;
  Part_Receiver: INT;
 END_STRUCT
END_TYPE
```

können in der UML als *DataType* mit Attributen nachgebildet werden (Abbildung 3.17).

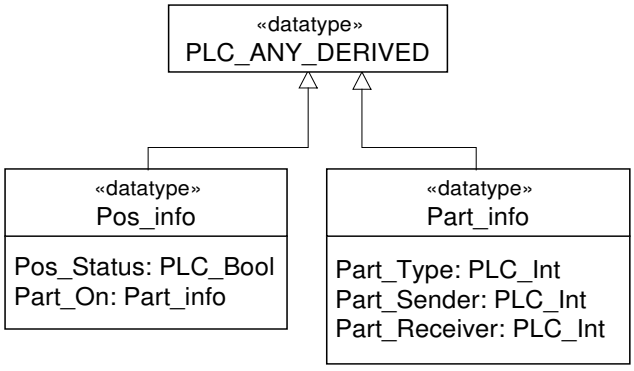


Abbildung 3.17 Beispiele für zusammengesetzte Typen der IEC 61131-3

Felder werden in der UML als Aggregation mit einer Kardinalität entsprechend der Größe des Feldes beschrieben. Ein Feld wie z. B.:

```
TYPE TRS_pos: array[1..7] of Pos_info; END_TYPE
```

sieht in einem UML-Klassendiagramm wie in Abbildung 3.18 dargestellt aus.

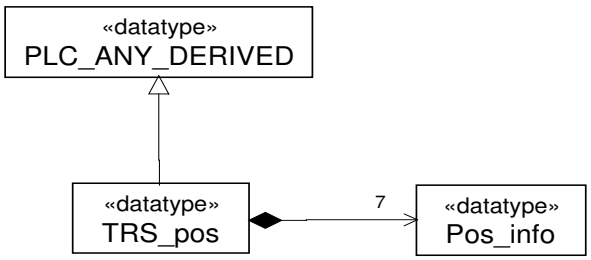


Abbildung 3.18 Beispiel für einen Feldtyp der IEC 61131-3

Wenn im weiteren Verlauf dieser Arbeit Datentypen der IEC 61131-3 in UML-Diagrammen verwendet werden, soll immer davon ausgegangen werden, dass sie auf die soeben dargelegte Weise in der UML nachgebildet wurden.

3.6 Das Profil „FunctionBlockAdapters“

In diesem Abschnitt wird als zentraler Bestandteil dieser Arbeit das UML-Profil für Funktionsbausteinadapter vorgestellt. Es soll „FunctionBlockAdapters“ genannt werden.

Ein UML-Profil ist ein spezielles UML-Paket, das *Stereotypen*, *Constraints* und *Tagged Values* enthält [UMLProfile]. Ein erster Ansatz zur Definition eines Stereotyps *FunctionBlockAdapter* wurde bereits in [HevTra 2001e] für die UML in der Version 1.4 beschrieben. Der dort vorgestellte Ansatz soll hier vervollständigt und auf die UML in der Version 2.0 angewendet werden. Da das Konzept der *Ports* erst in der UML 2.0 eingeführt wurde, musste der Ansatz aus [HevTra 2001e] vollständig überarbeitet werden.

Abbildung 3.19 zeigt ein Paket-Diagramm mit dem Profil *FunctionBlockAdapters* und den UML-Paketen, von denen das Profil abhängig ist. Es werden Elemente aus den Paketen für Ports [UMLPorts], für Interfaces [UMLInterfaces], für Klassen [UMLClasses], für ProtocolStateMachines [UMLProtocolStateMachines] und für CommonBehaviors [UMLCommonBehaviors] benötigt.

Das Profil *FunctionBlockAdapters* definiert im Wesentlichen zwei Konzepte - das des Funktionsbausteines (*FunctionBlock*) und das des Funktionsbausteinadapters (*FunctionBlockAdapter*), die beide die Metaklasse *Class* stereotypisieren. Zur Unterstützung der beiden eben genannten Stereotypen mussten noch einige weitere Metaklassen der UML stereotypisiert werden. Tabelle 3.1 listet alle im Profil enthaltenen

Stereotypen mit Namen, der stereotypisierten Metaklasse und einer Kurzbeschreibung auf. Grafisch werden die Stereotypen in Abbildung 3.20 gezeigt.

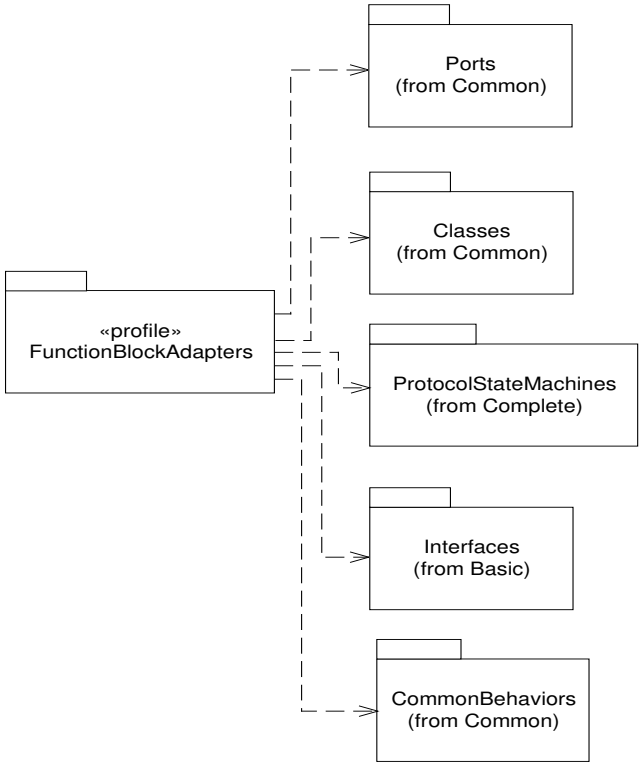


Abbildung 3.19 Das UML-Profil „FunctionBlockAdapters“

Tabelle 3.1 Stereotypen des Profils FunctionBlockAdapters

Stereotyp-Name	Angewendet auf	Beschreibung
FbInterface	Interface	Ist ein spezielles Interface, das zur Beschreibung des Verhaltens von Eingangs- und Ausgangsvariablen von Funktionsbausteinen benötigt wird.
FbPort	Port	Mit Hilfe von FBAPorts werden Eingangs- und Ausgangsvariablen von Funktionsbausteinen als Ports nachgebildet.
FbInPort	Port	Repräsentiert Ports für Eingangsvariablen von Funktionsbausteinen
FbOutPort	Port	Repräsentiert Ports für Ausgangsvariablen von Funktionsbausteinen
FunctionBlock	Class	Beschreibt das Konzept Funktionsbaustein
FunctionBlock-Adapter	Class	Beschreibt das Konzept des Funktionsbausteinadapters
FbaTranslation	Class	Dient zur Beschreibung des Verhaltens von Funktionsbausteinadaptern

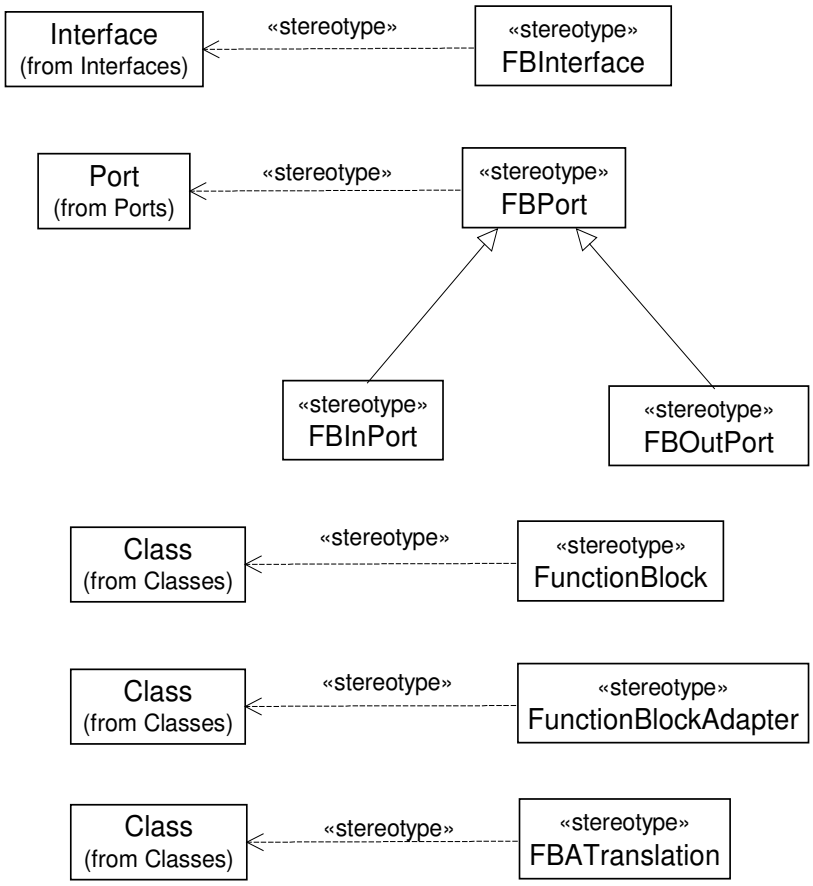


Abbildung 3.20 Stereotypen zur Definition von FBAs (Metamodell)

In den folgenden Unterabschnitten werden die Stereotypen ausführlich beschrieben.

3.6.1 FBInterface

Interfaces werden in der UML verwendet, um die öffentlichen Schnittstellen (Operationen/Attribute) von Klassen oder Ports zu beschreiben. Schnittstellen von Funktionsbausteinen werden demgegenüber durch Eingangs- und Ausgangsvariablen beschrieben. Für UML-Klassen interessante Ereignisse sind hierbei Wertänderungen in diesen Variablen. Solche Ereignisse sollen als ein UML-Signal namens *ValueChanged* übermittelt werden. Als Parameter enthält das Signal den neuen Wert der Variable. Definiert werden solche Signale als Operationen in FBInterfaces. Abbildung 3.21 zeigt beispielhaft zwei FBInterfaces, eine für einen benutzerdefinierten Datentyp *Pos_info* und eine für den Datentyp *BOOL* der IEC 61131-3.

Allen FBInterfaces ist gemeinsam, dass sie als Einziges eine Operation *ValueChanged* deklarieren. Unterschiedlich sind nur die Datentypen des Parameters dieser Operation. Eine *ValueChanged*-Operation beinhaltet immer genau einen Parameter. Die Datentypen können beliebiger Art sein und sind nicht auf die der IEC 61131-3 beschränkt. Dadurch lassen sich Funktionsbausteinadapter auch auf Funktionsblöcke anderer Sprachen (z. B. Matlab/Simulink) anwenden.

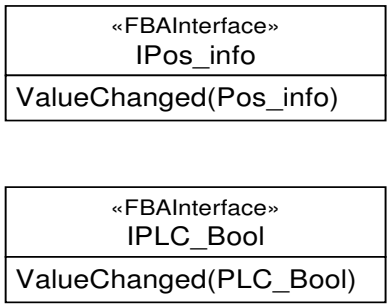


Abbildung 3.21 Beispiele für FBAInterfaces

Der Stereotyp *FBInterface* unterscheidet sich von der Metaklasse *Interface* durch die soeben beschriebenen Einschränkungen, die formal in der OCL folgendermaßen ausgedrückt werden können:

```
context FBInterface inv:
      (self.feature->size = 1) &
      (self.feature->forAll( f |
            (f.name = 'ValueChanged') &
            (f.classifier = Operation) &
            (f.parameter->size = 1))
      )
```

Als Namenskonvention für *FBInterfaces* soll hier angenommen werden, dass der Name immer mit einem großen „I“ beginnt, an das sich der Name des Datentyps des Parameters der *ValueChanged*-Operation anschließt (vgl. Abbildung 3.21).

Mit Hilfe von *FBInterfaces* ist es möglich, Eingangs- und Ausgangsvariablen von Funktionsbausteinen durch Ports zu modellieren.

3.6.2 FBInPort

FBInPort soll Eingangsvariablen von Funktionsbausteinen nachbilden. Eingangsvariablen haben den Zweck, das interne Verhalten eines Funktionsbausteins unabhängig davon zu machen, mit welchen externen Funktionsbausteinen eine Funktionsbausteininstanz über ihre Eingangspins verknüpft ist. Deshalb dürfen Eingangsvariablen vom Funktionsbaustein, dem sie angehören, nur gelesen, aber nicht verändert werden. Von externen Funktionsbausteinen dürfen Eingangsvariablen aber beliebig verändert werden. Ein solches Konzept ist mit Attributen von Klassen in der Objektorientierung nicht direkt nachvollziehbar, da diese nur entweder öffentlich (public) zugreifbar sind (also auch von innerhalb der Klasse, der sie angehören) oder von der Außenwelt gekapselt werden. Das am besten geeignete Konzept zur Nachbildung von Eingangsvariablen ist deshalb ein Port, das externe Signale an das interne Verhalten einer Klasse weiterleitet. Ein solches Port muss eine Schnittstelle vom Typ *FBInterface* anbieten. Ein Beispiel für eine Eingangsvariable namens *Execute* vom Typ *Bool* zeigt Abbildung 3.22 rechts. In der Abbildung links ist die Nachbildung dieser Eingangsvariable mit Hilfe eines *FBInPorts* zu sehen.

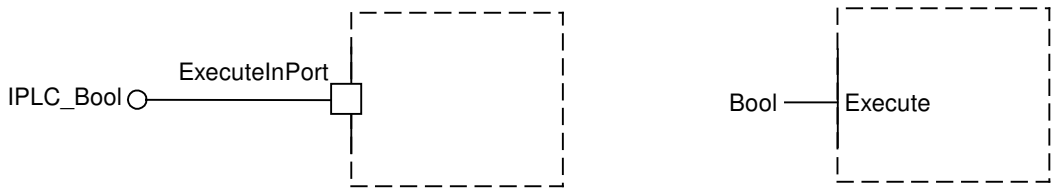


Abbildung 3.22 Vergleich eines FBInPort mit einer Eingangsvariable

Der Stereotyp *FBInPort* zeichnet sich im Gegensatz zu normalen Ports der UML dadurch aus, das er nur genau ein Interface vom Typ *FBInterface* als *provided* zur Verfügung stellt. Die Menge der *required Interfaces* ist leer:

```
context FBInPort inv:
      (self.required->size = 0) &
      (self.provided->size = 1) &
      (self.provided->forAll( i | (i.classifier = FBInterface)))
```

3.6.3 FBOutPort

FBOutPorts sind das Gegenstück zu *FBInPorts*. Sie modellieren Ausgangsvariablen von Funktionsbausteinen. Der Stereotyp *FBOutPort* besitzt genau ein *required FBInterface* und keine *provided Interfaces*:

```
context FBOutPort inv:
      (self.provided->size = 0) &
      (self.required->size = 1) &
      (self.required->forAll( i | (i.classifier = FBInterface)))
```

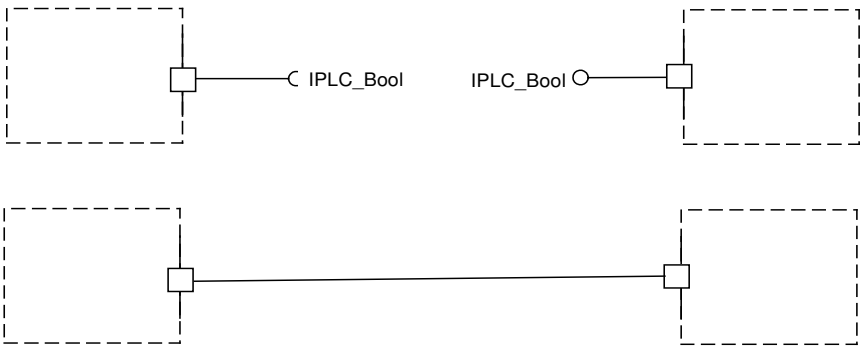


Abbildung 3.23 oben: Ports mit Schnittstellen-Typ; unten: verbundene Ports im Strukturdiagramm

Genau wie bei normalen UML-Ports können auch FBPorts im Strukturdiagramm miteinander verbunden werden, wenn ihre Schnittstellen übereinstimmen (Abbildung 3.23).

3.6.4 FBPort

Der Stereotyp *FBPort* ist die Oberklasse von *FBInPort* und *FBOutPort.* Beiden Unterklassen gemeinsam ist, dass sie die Metaklasse *Port* stereotypisieren (Abbildung 3.20). Weiterhin sollen die *ValueChanged*-Operationen als Signale behandelt werden:

```
context FBPort inv: self.isSignal = true
```

Für die Darstellung von FBPorts in Klassen- und Strukturdiagrammen kann eine spezielle Notation eingesetzt werden, die der von Eingangs- und Ausgangspins bei Funktionsbausteinen entnommen wurde. In Abbildung 3.24 sind nochmals die gleichen FBPorts aus Abbildung 3.23 in der speziellen Notation dargestellt.

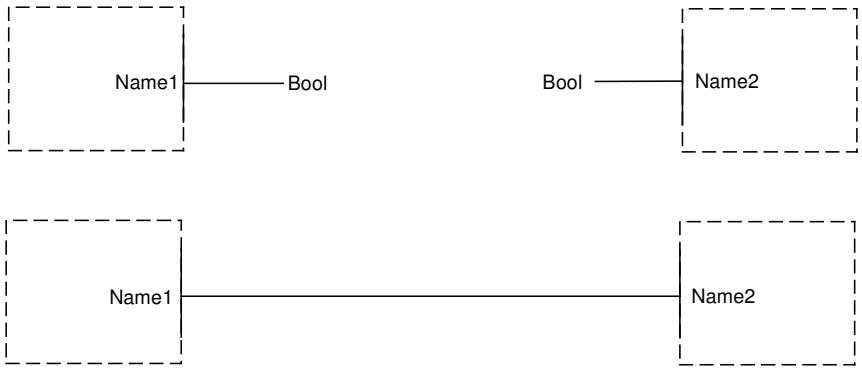


Abbildung 3.24 Pin-Notation im Klassendiagramm (oben) und im Strukturdiagramm (unten)

3.6.5 FunctionBlock

Der Stereotyp *FunctionBlock* dient zur Darstellung von Funktionsbausteinen mit ihren Eingangs- und Ausgangsvariablen in Klassen- und Strukturdiagrammen. Das interne Verhalten von Funktionsbausteinen soll nicht näher betrachtet werden. Es kann durch Verhaltensmodelle der UML oder anderer Sprachen (z.B. IEC 61131-3) beschrieben werden. *FunctionBlock* ist ein Stereotyp der Metaklasse *Class*. Es gilt dabei die Einschränkung, dass ein *FunctionBlock* nur öffentliche *FBPorts* enthalten darf:

```
context FunctionBlock inv:
      self.feature->forAll( f |
            (f.classifier = FBPort) &
            (f.visibility = #public)
      )
```

3.6.5.1 Notation

Für die Darstellung in Klassendiagrammen gibt es zwei Möglichkeiten. In Standard-UML Werkzeugen ohne Erweiterung für Funktionsbausteine müssen die Stereotypen durch Schlüsselwörter in eckigen Klammern *«keyword»* gekennzeichnet werden (Abbildung 3.25 oben). In UML-Werkzeugen mit Erweiterung für Funktionsbausteine können spezielle Notationen zur Verfügung gestellt werden, aus denen die Zuordnung zu den entsprechenden Stereotypen ersichtlich ist (Abbildung 3.25 unten).

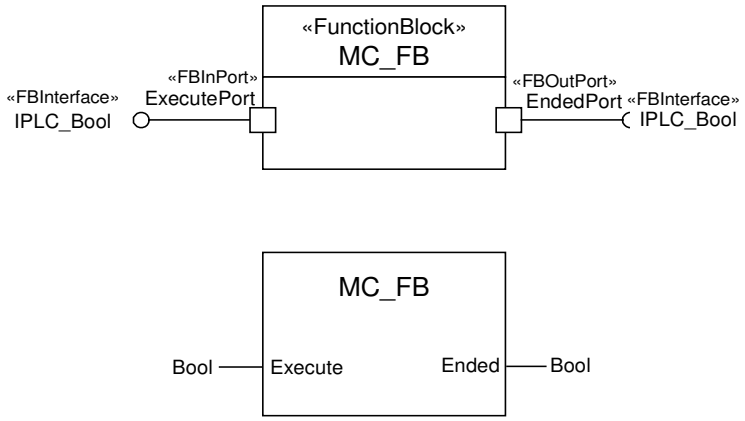


Abbildung 3.25 Beispiele für Funktionsbausteine

In Abbildung 3.25 unten wurde z. B. eine Darstellung gewählt, die der aus IEC 61131-3 entspricht. Genauso könnte man *MC_FB* aber auch als Funktionsblock nach Matlab/Simulink [Simulink 2002] darstellen.

3.6.6 FunctionBlockAdapter

Der Stereotyp *FunctionBlockAdapter* ist die zentrale Komponente des Profils *FunctionBlockAdapters*. Ein Funktionsbausteinadapter enthält sowohl normale Ports als auch *FBPorts*.

```
context FunctionBlockAdapter inv:
      self.feature->exists( classifier = FBPort ) &
      self.feature->exists( classifier = Port & classifier != FBPort )
```

Im Gegensatz zum Stereotyp *FunctionBlock* wird das Verhalten eines Funktionsbausteinadapters vollständig beschrieben. Das kann einerseits wie in der UML üblich durch Statecharts geschehen. Alternativ dazu kann eine spezielle Sprache, die FBA-Sprache, verwendet werden. Die FBA-Sprache ist eine textuelle Sprache, deren Komplexität erheblich geringer ist als die von Statecharts. Sie wird ausführlich im Abschnitt 3.6.7.1 behandelt. Damit man einem Funktionsbausteinadapter Beschreibungen in der FBA-Sprache zuordnen kann, enthält der Stereotyp *FunctionBlockAdapter* eine zusätzliche Referenz (als *tagged value*) *translations* auf den Stereotyp *FBATranslation* (Abbildung 3.26).

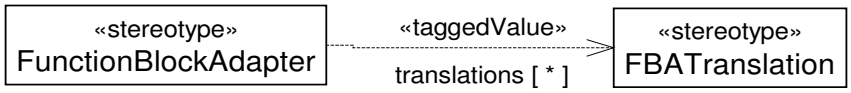


Abbildung 3.26 TaggedValue "translations"

Die Aufgabe eines Funktionsbausteinadapters ist es, Ereignisse von normalen Ports in Ereignisse von *FBPorts* und umgekehrt zu übersetzen. Wurde ein Ereignis vollständig in Ereignisse anderer Ports umgesetzt, so ist eine Übersetzung (*FBATranslation*) abgeschlossen. Ein Funktionsbausteinadapter kann mehrere Übersetzungen beinhal-

ten. Die Menge aller enthaltenen Übersetzungen ist in *translations* gegeben. Die Beziehung zwischen FBATranslations und Statecharts werden ausführlich in Abschnitt 3.6.7 diskutiert.

3.6.6.1 Notation

Funktionsbausteinadapter können ähnlich wie Funktionsbausteine auf unterschiedliche Weise in UML-Diagrammen dargestellt werden. In Standard-UML Werkzeugen ohne Erweiterung für Funktionsbausteine müssen die Stereotypen durch Schlüsselwörter in eckigen Klammern *«keyword»* gekennzeichnet werden (Abbildung 3.27 oben). In UML-Werkzeugen mit Erweiterung für Funktionsbausteine können spezielle Notationen zur Verfügung gestellt werden, aus denen die Zuordnung zu den entsprechenden Stereotypen ersichtlich ist (Abbildung 3.27 unten).

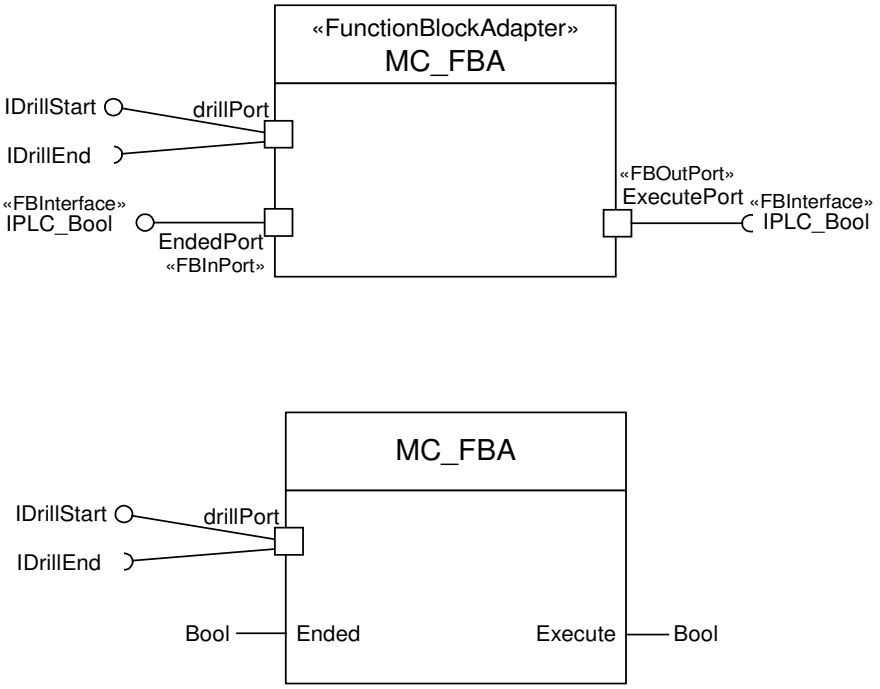


Abbildung 3.27 Beispiele für FBAs in unterschiedlicher Schreibweise

Die in Abbildung 3.27 unten gewählte Darstellungsweise eines Funktionsbausteinadapters unterscheidet sich von der eines einfachen Funktionsbausteines dadurch, dass auch normale UML-Ports erlaubt sind und der Klassenname wie bei UML-Klassen in einem eigenen Bereich geschrieben wird. Die in einem Funktionsbausteinadapter enthaltenen Übersetzungen können in einem speziellen Listenbereich eingetragen werden (Abbildung 3.28).

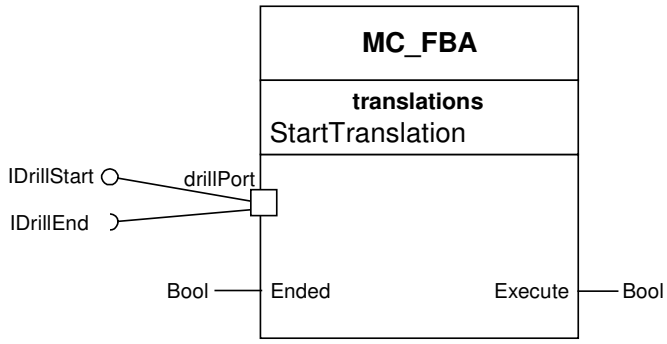


Abbildung 3.28 Darstellung mit translations-Liste

3.6.6.2 Schnittstellenvariablen

Für jedes *FBPort* eines Funktionsbausteinadapters gibt es ein Attribut vom Typ des Parameters des *ValueChanged*-Signals des entsprechenden *FBPorts*. Diese Attribute weisen immer den aktuellen Wert auf, der zuletzt über das dazugehörige *FBPort* gesendet oder empfangen wurde. Sie dienen als Ersatz für Eingangs- und Ausgangsvariablen und sollen die Verhaltensbeschreibung des Funktionsbausteinadapters vereinfachen. Im Weiteren sollen diese speziellen Attribute Schnittstellenvariablen vom Funktionsbausteinadapter genannt werden.

In Abbildung 3.27 erkennt man in der oberen Schreibweise, dass dort die Namen der *FBPorts* verwendet werden (z.B. *EndedPort*), während in der unteren Schreibweise die Namen der Schnittstellenvariablen zu sehen sind.

3.6.7 FBATranslation

Der Stereotyp *FBATranslation* dient zur Beschreibung von Übersetzungen innerhalb eines Funktionsbausteinadapters. Eine Übersetzung ist eine Abbildung von Ereignissen normaler UML-Ports auf *FBPorts* und umgekehrt. Für die Abbildung ist die zeitliche Reihenfolge der Ereignisse von Bedeutung. Durch eine *FBATranslation* ist eine Übersetzung vollständig definiert. Eine *FBATranslation* muss immer einem Funktionsbausteinadapter zugeordnet sein. Der Typ des Funktionsbausteinadapters ist im TaggedValue *fba* gegeben (Abbildung 3.29).

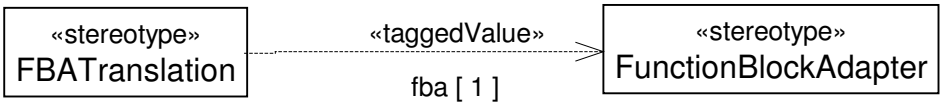


Abbildung 3.29 Tagged value "fba"

Eine Instanz einer *FBATranslation*-Klasse muss eine Referenz (einen Zeiger) auf eine Instanz eines Funktionsbausteinadapters vom in *fba* angegebenen Typ besitzen. Diese referenzierte Instanz muss ihrerseits die referenzierende *FBATranslation* beinhalten (Abbildung 3.30).

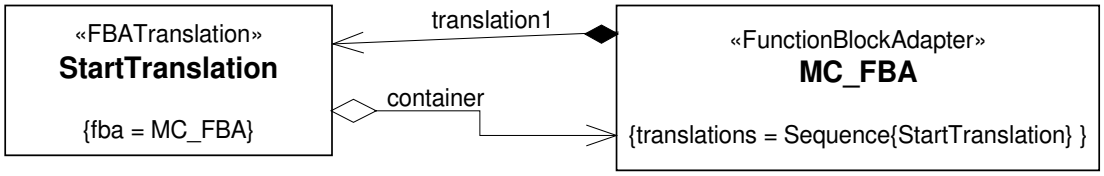


Abbildung 3.30 MC_FBA ist Container für StartTranslation

Eine Instanz der Klasse *StartTranslation* aus Abbildung 3.30 referenziert über *container* die Instanz von *MC_FBA*, in der sie in *translation1* enthalten ist.

FBATranslation benötigt weitere TaggedValues, die in Tabelle 3.2 aufgeführt sind.

Der TaggedValue *isOrthogonal* ist von Bedeutung, wenn es mehrere Übersetzungen in einem Funktionsbausteinadapter gibt. Wenn *isOrthogonal* gesetzt ist, dann wird die Übersetzung sofort ausgeführt, wenn das in *trigger* gegebene Ereignis auftritt, auch wenn parallel dazu schon andere Übersetzungen laufen.

Tabelle 3.2 TaggedValues für FBATranslation

Name	Typ	Kardinalität	Beschreibung
fba	FunctionBlock-Adapter	1	Enthält die Funktionsbausteinadapter-Klasse, der die FBATranslation zugeordnet wird.
isOrthogonal	Boolean	1	Zeigt an, ob die FBATranslation nebenläufig zu anderen Übersetzungen ausgeführt werden kann.
trigger	Event	1	Kennzeichnet das Ereignis, das die Übersetzung startet. Das kann entweder ein ChangeEvent oder ein SignalEvent sein.
signals	Signal	0..*	Eine Menge von Signalen, die von der Übersetzung verarbeitet oder erzeugt werden.
translation-Body	String	1	Beinhaltet eine Beschreibung des Verhaltens der FBATranslation in der Syntax der FBA-Sprache.
exception-Body	String	0..1	Beinhaltet eine Beschreibung des Verhaltens der FBATranslation in der Syntax der FBA-Sprache beim Auftreten eines Fehlers in der Übersetzung.

Name	Typ	Kardinalität	Beschreibung
fbProtocol	ProtocolStatechart	0..1	Gibt ähnlich wie ein ProtokollStatechart eines Ports die Reihenfolge vor, in der Ereignisse im Kommunikationsprotokoll zwischen *fba* und mit ihm verbundenem Funktionsbaustein während der Übersetzung auftreten dürfen.

Das in *trigger* gegebene Ereignis ist entweder ein SignalEvent, das sich auf einen Port von *fba* bezieht, oder ein ChangeEvent, das sich auf Eingangsvariablen von *fba* bezieht. Als Syntax für den ChangeExpression eines ChangeEvents soll die Syntax von FBA_Event_Expression aus Abschnitt 3.6.7.1 definiert werden.

Für jedes Signal aus *signals* muss eine *FBATranslation* mindestens eine Referenz auf ein SignalEvent besitzen. Diese Referenzen werden innerhalb einer Übersetzung genutzt, um auf Attribute dieser Ereignisse zuzugreifen.

Alle *FBATranslations*, die im TaggedValue *translations* von *fba* enthalten sind, und bei denen *isOrthogonal* nicht gesetzt ist, werden nicht parallel zueinander ausgeführt. Tritt ein Ereignis, das eine nicht-parallele *FBATranslation* starten kann, auf, während eine andere nicht-parallele *FBATranslation* bereits läuft, muss die laufende Übersetzung zunächst abgearbeitet werden. Danach kann erst die nächste nicht-parallele Übersetzung starten. Voraussetzung dafür ist aber, dass das Ereignis nicht bereits in der vorherigen Übersetzung verarbeitet wurde. Der wichtigste Grund dafür, mehrere Übersetzungen als nicht-parallel zu definieren, ist der, dass innerhalb dieser Übersetzungen auf dieselben Variablen oder Ports zugegriffen werden kann. Dabei können die Variablen oder Ports im Kontext unterschiedlicher Übersetzungen unterschiedliche Bedeutungen erlangen.

Wenn mehrere Ereignisse, die nicht-parallele Übersetzungen starten können, gleichzeitig auftreten, dann entscheidet die Reihenfolge der Übersetzungen in *translations* von *fba* darüber, welche Übersetzung gestartet wird. Die übrigen Startereignisse, deren Übersetzungen nicht gestartet wurden, werden so behandelt, als wären sie während der Abarbeitung der gestarteten Übersetzung eingetroffen.

Die TaggedValues *translationBody* und *exceptionBody* enthalten eine Zeichenkette, die der Syntax der FBA-Sprache entsprechen muss. Diese Syntax wird im Abschnitt 3.6.7.1 erläutert. In *translationBody* wird das Verhalten beschrieben, das zur Übersetzung notwendig ist. Treten bei der Ausführung dieses Verhaltens Fehler auf, dann wird die Ausführung abgebrochen und das in *exceptionBody* enthaltene Verhalten

gestartet. Fehler sind Zeitschrankenüberschreitungen, die beim Warten auf Ereignisse auftreten können.

Das in *fbProtocol* gegebene Protokollstatechart beschreibt das Kommunikationsprotokoll zwischen *fba* und dem (den) mit dem *fba* verbundenen Funktionsbaustein(en). Als Trigger für die Transitionen des Statecharts sind deshalb nur ChangeEvents erlaubt. Diese ChangeEvents dürfen sich nur auf die Schnittstellenvariablen von *fba* beziehen. Wie später gezeigt wird, können die in *fbProtocol* und den Ports gegebenen Protokollstatecharts zur Verifikation des Funktionsbausteinadapters eingesetzt werden.

Die für die TaggedValues gemachten Einschränkungen werden in Tabelle 3.3 formal mit der OCL beschrieben.

Tabelle 3.3 Constraints für FBATranslation

Beschreibung	Object Constraint Language

Der in *fba* gegebene Funktionsbausteinadapter muss die FBATranslation in *translations* enthalten.	`context FBATranslation inv:`
	`  fba.translations->includes(self)`
	
	`context FunctionBlockAdapter inv:`
	`  translations->forAll( t | t.fba=self)`

trigger ist entweder ein SignalEvent oder ein ChangeEvent	`context FBATranslation inv:`
	`  trigger.classifier=ChangeEvent |`
	`  trigger.classifier=SignalEvent`

Jedes Signal aus signals muss eine Reception[1] in einem der Ports von fba haben.	`context FBATranslation inv:`
	`  signals->forAll( s |`
	`  fba.ownedPort->exists( p |`
	`  p.required->exists( i |`
	`  i.ownedReception->includes(s.reception)`
	`  )`
	`  |`
	`  p.provided->exists( i |`
	`  i.ownedReception->includes(s.reception)`
	`  )`
	`  )`
	`  )`

Wenn *trigger* ein SignalEvent ist, muss das zugehörige Signal auch in signals enthalten sein.	`context FBATranslation inv:`
	`  if trigger.classifier=SignalEvent then`
	`  signals->includes(trigger.signal)`

[1] Eine Reception gehört zu einem Signal und kennzeichnet ein Verhalten (z.B. eine Operation), das beim Empfänger des Signals ausgelöst werden soll (siehe auch [UMLCommonBehaviors]).

Beschreibung	Object Constraint Language

Für jedes Signal aus *signals* muss eine *FBATranslation* mindestens eine Referenz auf ein SignalEvent besitzen.	`context FBATranslation inv:`
	`  signals->forAll( s |`
	`  self.reference->exists( a |`
	`  if a.classifier=SignalEvent`
	`  then a.signal=s`
	`  else false`
	`  )`
	`  )`

Als Trigger für die Transitionen von *fbProtocol* sind nur ChangeEvents erlaubt.	`context FBATranslation inv:`
	`  fbProtocol.region->forAll( r |`
	`  r.transitions->forAll( t |`
	`  t.trigger.classifier = ChangeEvent`
	`  )`
	`  )`

3.6.7.1 Syntax der FBA-Sprache

In diesem Abschnitt soll die Syntax der FBA-Sprache, die in *translationBody* und *exceptionBody* verwendet wird, mit Hilfe einer kontextfreien Grammatik $G_{FBA} = (T_{FBA}, N_{FBA}, P_{FBA}, S_{FBA})$ definiert werden. T_{FBA} ist eine Menge terminaler Symbole, N_{FBA} eine Menge nichtterminaler Symbole, P_{FBA} eine Menge von Produktionsregeln und S_{FBA} ein Startsymbol aus N_{FBA}. Da es sich um eine kontextfreie Grammatik handelt, kann sie auch in der EBNF (erweiterte Backus-Naur-Form) dargestellt werden [Hed 2000].

Das Startsymbol heißt *FBA_statement_list* und besteht aus einer durch Semikolons getrennten Sequenz von Befehlen:

```
FBA_statement_list ::= FBA_statement ';' {FBA_statement ';'}
```

Ein Befehl (*FBA_statement*) kann entweder das Warten auf ein Ereignis auslösen (*FBA_waitFor_statement*), die weitere Ausführung der Übersetzung um eine vorgegebene Zeit verzögern (*FBA_delay_statement*), ein Signal über einen Port versenden (*FBA_send_statement*) oder einer Variablen einen Wert zuweisen (*FBA_assignment*):

```
FBA_statement ::= FBA_waitFor_statement | FBA_delay_statement |
                  FBA_send_statement | FBA_assignment
```

Ein Wartebefehl erhält ein oder zwei Parameter. Der erste kennzeichnet das Ereignis, auf das gewartet werden soll und der zweite gibt optional eine Zeitschranke an, die beim Warten nicht überschritten werden darf.

```
FBA_waitFor_statement ::= 'waitFor' '(' FBA_event_expression
                              [',' FB_time_literal]
                          ')'
```

Beispiele für Wartebefehle sind:

```
waitFor(Ended);
waitFor(s3, T#5s);
...
```

Eine Zeitverzögerung enthält nur einen Parameter, der die Länge der Zeitverzögerung bestimmt:

```
FBA_delay_statement ::= 'delay' '(' FB_time_literal ')'
```

Ein Beispiel für eine Zeitverzögerung ist:

```
delay(T#5ms);
...
```

Das Senden eines Signals über einen Port wird durch einen Befehl ausgelöst, der nur einen Parameter benötigt. Dieser Parameter enthält den Namen der Referenz auf ein SignalEvent der FBATranslation (siehe Tabelle 3.3).

```
FBA_send_statement ::= 'send' '(' FBA_postfixExpression ')'
```

Ein Sendebefehl kann zum Beispiel so aussehen:

```
send(s3);
...
```

Eine Zuweisung kann in der FBA-Sprache unterschiedliche Formen annehmen. Ausgangsvariablen sollen Werte aus Signalen zugewiesen werden, die über Ports eingetroffen sind. Signalen, die über Ports versendet werden sollen, müssen vorher Werte aus Eingangsvariablen zugewiesen werden. Alternativ können Ausgangsvariablen und Signalen auch mit konstanten Werten belegt werden. Zuweisungen zu Attributen von Signalen müssen nicht zwangsläufig durch den Zuweisungoperator := erfolgen, sondern können auch durch Aufruf entsprechender Operationen der Datenklassen ausgeführt werden.

```
FBA_assignment ::=
      (FBA_postfixExpression ':='
            (FBA_postfixExpression | FB_constant))
      | FBA_postfixExpression
```

Um auf Eigenschaften von UML-Klassen wie Funktionsbausteinadaptern in einer formalen Sprache zugreifen zu können, gibt es in der Grammatik der OCL [OCL Grammatik] das Symbol postfixExpression. Das Symbol FBA_postfixExpression stellt eine vereinfachte, auf die FBA-Sprache angepasste Syntax dar.

```
FBA_postfixExpression ::= UML_propertyCall {
                              ('.' | '->') UML_propertyCall }
```

Das Symbol *UML_propertyCall* entspricht dem Symbol *propertyCall* aus [OCL Grammatik].

```
UML_propertyCall ::= propertyCall
```

Das Symbol *FB_constant* entspricht dem Symbol *constant* aus der IEC 61131-3 [IECGrammatik 1993]. Soll das Profil *FunctionBlockAdapters* auf Funktionsbausteine anderer Sprachen als der SPS-Sprachen angewendet werden, kann an dieser Stelle auf die entsprechende Syntax verwiesen werden.

```
FB_constant ::= constant
```

Das Symbol *FBA_event_expression* vom Wartebefehl ist entweder eine Referenz auf ein SignalEvent, die der Syntax des *FBA_postfixExpression* entspricht, oder ein *FB_expression*.

```
FBA_event_expression ::= FBA_postfixExpression | FB_expression
```

Das Symbol *FB_expression* entspricht der Syntax des Ausdrucks expression, der in der IEC 61131-3 [IECGrammatik 1993] definiert ist. Die Syntax von Zeitausdrücken wird an gleicher Stelle beschrieben.

```
FB_expression ::= expression
```

```
FB_time_literal ::= time_literal
```

Diese Syntax kann ebenso wie *FB_constant* an andere funktionsbausteinorientierte Sprachen angepasst werden.

Für die Ausnahmebehandlung in *exceptionBody* soll zusätzlich die Einschränkung gelten, dass keine *waitFor*- und *delay*-Anweisungen erlaubt sind. Für *exceptionBody* gilt deshalb die Grammatik mit dem Startsymbol *FBA_exception_list*.

```
FBA_exception_list ::= FBA_exception_action ';' {FBA_exception_action ';'}
```

```
FBA_exception_action ::= FBA_send_statement | FBA_assignment
```

Eine *FBA_exception_action* kann entweder eine *Send*-Anweisung oder eine Zuweisung sein.

3.6.7.2 Notation

Ähnlich wie bei Operationen von UML-Klassen sollen nicht alle Eigenschaften von Übersetzungen in Klassendiagrammen sichtbar sein. In UML-Werkzeugen mit Erweiterung für Funktionsbausteinadapter kann ein spezielles Dialogfenster für die Bearbeitung von Übersetzungen angeboten werden. Abbildung 3.31 zeigt beispielhaft ein Dialogfenster für die StartTranslation des Funktionsbausteinadapters MC_FBA aus Abbildung 3.28.

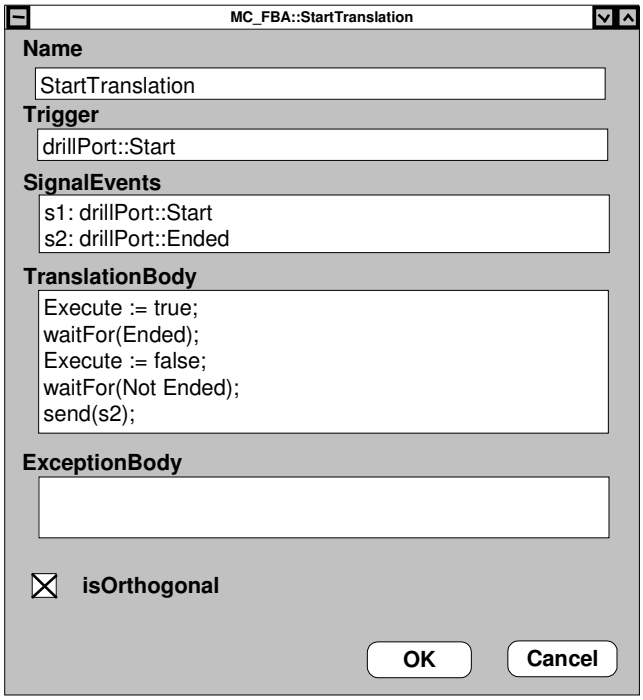


Abbildung 3.31 Dialogfenster zum Bearbeiten einer FBATranslation

In UML-Werkzeugen ohne Erweiterung für Funktionsbausteinadapter können diese Informationen über Textnotizen in Klassendiagrammen den Übersetzungen zugeordnet werden.

3.6.7.3 Semantik der FBA-Sprache

Die Semantik der FBA-Sprache soll vollständig auf die Semantik von Statecharts abgebildet werden. Deshalb werden in diesem Abschnitt Regeln aufgestellt, die es erlauben, aus den Elementen der FBATranslations eines Funktionsbausteinadapters dessen Statechart zu generieren.

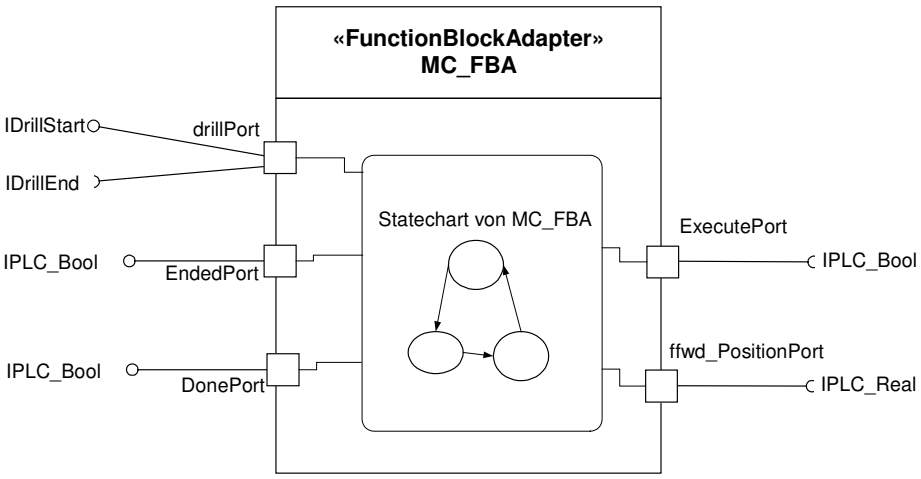


Abbildung 3.32 Statechart von MC_FBA, das mit allen Ports verbunden ist

Das generierte Statechart muss in der Lage sein, alle Übersetzungen des Funktionsbausteinadapters durchzuführen. Alle Ereignisse, die in den Ports des Funktionsbausteinadapters auftreten können, werden von diesem Statechart verarbeitet (Abbildung 3.32).

Im Weiteren soll das Statechart von MC_FBA als Beispiel zur Verdeutlichung der hier aufgestellten Regeln dienen. Zunächst beinhaltet jeder Funktionsbausteinadapter ein Standardverhalten, das dafür sorgt, dass bei Eintreffen von Signalen in FBInPorts die entsprechenden Eingangsvariablen aktualisiert werden (Abbildung 3.33).

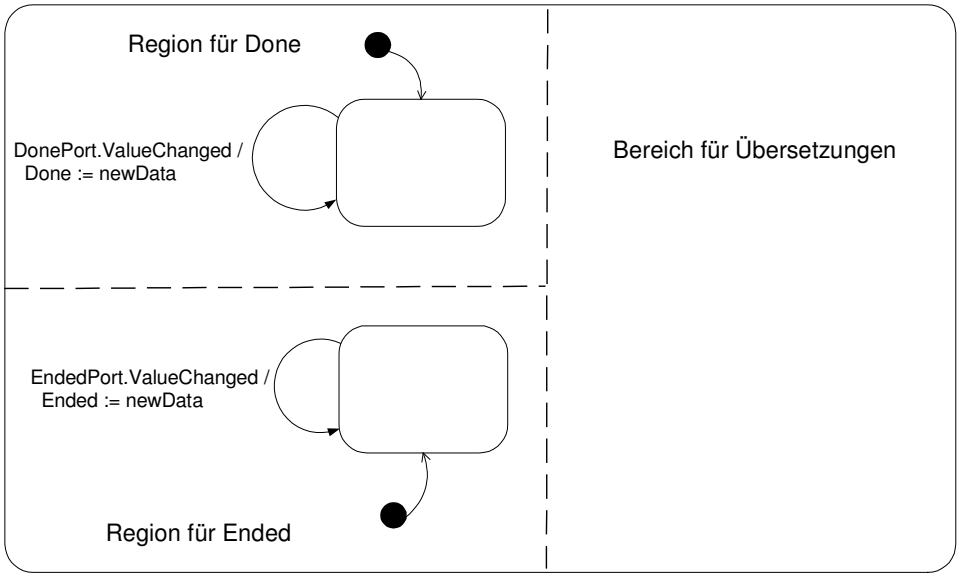


Abbildung 3.33 Verarbeitung der Signale von FBInPorts

Da sich die Variablen von Funktionsbausteinen auch gleichzeitig ändern können, ist im Statechart für jede Eingangsvariable eine Region vorgesehen. Dadurch werden die ValueChanged-Signale der FBInPorts nebenläufig zum restlichen Verhalten des Funktionsbausteinadapters bearbeitet. Im Bereich für Übersetzungen aus Abbildung 3.33 sind die Teile des Statecharts enthalten, die für die in den FBATranslations beschriebenen Übersetzungen zuständig sind.

Für jede Übersetzung, bei der *isOrthogonal* gesetzt ist, wird eine eigene Region in das Statechart eingefügt. Solche Regionen haben alle einen ähnlichen Aufbau: Ausgehend vom Startpunkt wird in einen Wartezustand *S0* übergegangen, der nur durch eine Transition *t0* verlassen werden kann. Der Auslöser dieser Transition ist durch das TaggedValue *trigger* der FBATranslation gegeben (Abbildung 3.34).

Wenn im *translationBody* keine Wartebefehle (*waitFor*) oder Zeitverzögerungen (*delay*) eingebaut sind, dann endet die Transition *t0* in *S0*. Ansonsten endet die Transition im nachfolgenden Wartezustand.

Jeder Wartebefehl und jede Zeitverzögerung wird zu einem Wartezustand, der durch nur eine Transition verlassen werden kann. Bei einer *waitFor*-Anweisung wird diese Transition durch das Ereignis ausgelöst, das durch den ersten Parameter der Anweisung beschrieben wird. Bei einer *delay*-Anweisung wird diese Transition durch ein timeout-Signal ausgelöst, das von einem Zeitgeber nach Ablauf der im Parameter der *delay*-Anweisung angegebenen Zeitspanne gesendet wird. Dieser Zeitgeber wird beim Eintritt in den *delay*-Wartezustand in der Entry-Aktion gestartet. Wird bei einer *waitFor*-Anweisung der zweite Parameter angegeben, dann muss auch in einem *waitFor*-Wartezustand ein Zeitgeber in der Entry-Aktion gestartet werden. Wird der *waitFor*-Zustand verlassen, bevor die Zeitschranke überschritten wurde, muss der Zeitgeber in der Exit-Aktion wieder gestoppt werden.

Die Transition, die einen Wartezustand verlässt, endet entsprechend der Reihenfolge im nächsten Wartezustand, der durch eine *delay*- oder *waitFor*-Anweisung im *translationBody* gegeben ist. Nach dem letzten Wartezustand endet die Transition im Zustand *S0*.

Die Zuweisungs- und Send-Befehle im translationBody werden zu Aktionen, die beim Ausführen einer Transition abgearbeitet werden. Ein Send-Befehl wird zu einer SendSignalAction [UMLSendSignalAction]. Durch den Parameter ist ein SignalEvent gegeben, das den Signaltyp und die Argumente für die SendSignalAction enthält. Die Zuweisung eines Wertes einer Variable zu einem Attribut eines SignalEvents wird zu einer ReadAttributeAction [UMLReadAttributeAction] zum Lesen des Wertes der Variable rechts von := und einer AddAttributeValueAction [UMLAddAttributeValueAction] zum Schreiben der Variable links von :=. Eine Zuweisung zu einer Ausgangsvariable wird auf eine ReadAttributeValueAction, eine AddAttributeValueAction und auf eine SendSignalAction abgebildet. Die AddAttributeAction weist der Ausgangsvariable des Funktionsbausteinadapters den neuen Wert zu, während die SendSignalAction ein ValueChanged-Signal über das entsprechende FBOutPort sendet. Wird einer Variablen ein konstanter Wert zugewiesen, fällt die ReadAttributeAction in beiden eben genannten Fällen weg.

Erfolgt eine Zuweisung zu einem Attribut eines SignalEvents über den Aufruf einer Operation der Signal-Klasse, dann ist die Semantik durch die Aktionen definiert, die der Operation zugeordnet wurden.

Eine Zeitschrankenüberschreitung in einem *waitFor*-Wartezustand bedeutet, dass im übergeordnetem Statechart eine Transition *t_err* ausgelöst wird, die zum Ausgangszustand *S0* führt. Diese Transition beinhaltet die durch *exceptionBody* beschriebenen Aktionen. Damit eine solche Transition nicht mehrmals von jedem Wartezustand aus nach *S0* eingeführt werden muss, sollen alle Zustände außer *S0* einer Übersetzung zu

einem übergeordneten, zusammengesetzten Zustand *S1* zusammengefasst werden. *t_err* startet im Zustand *S1* und endet in *S0*.

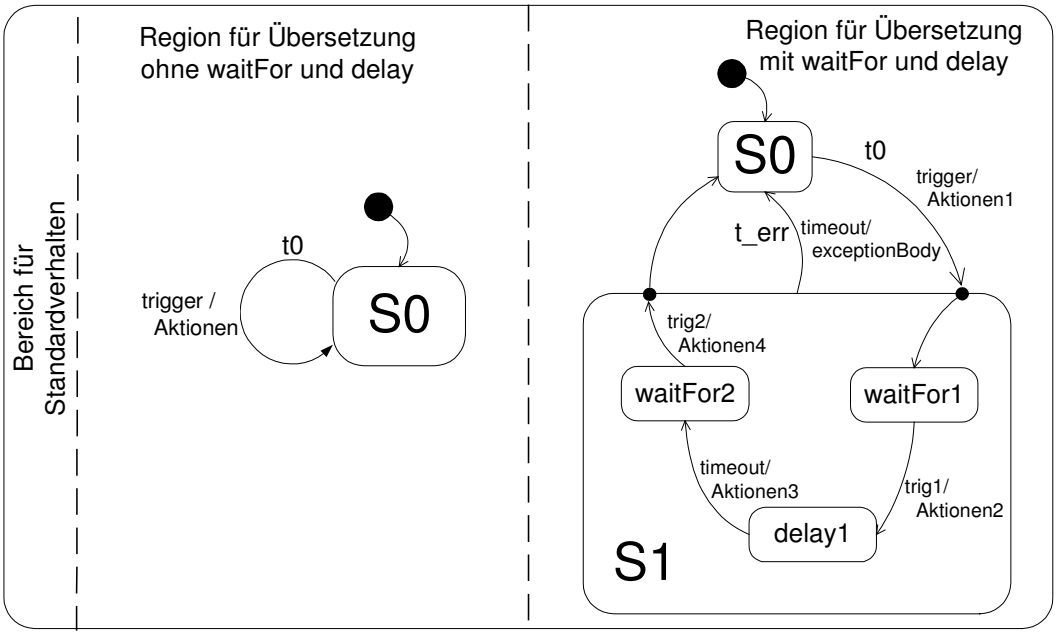


Abbildung 3.34 Statechart mit zwei orthogonalen Übersetzungen

Alle Übersetzungen eines Funktionsbausteinadapters, die nicht orthogonal sind (isOrthogonal=false), werden in einer Region des Statecharts zusammengefasst, weil sie nicht nebenläufig ausgeführt werden können(Abbildung 3.35).

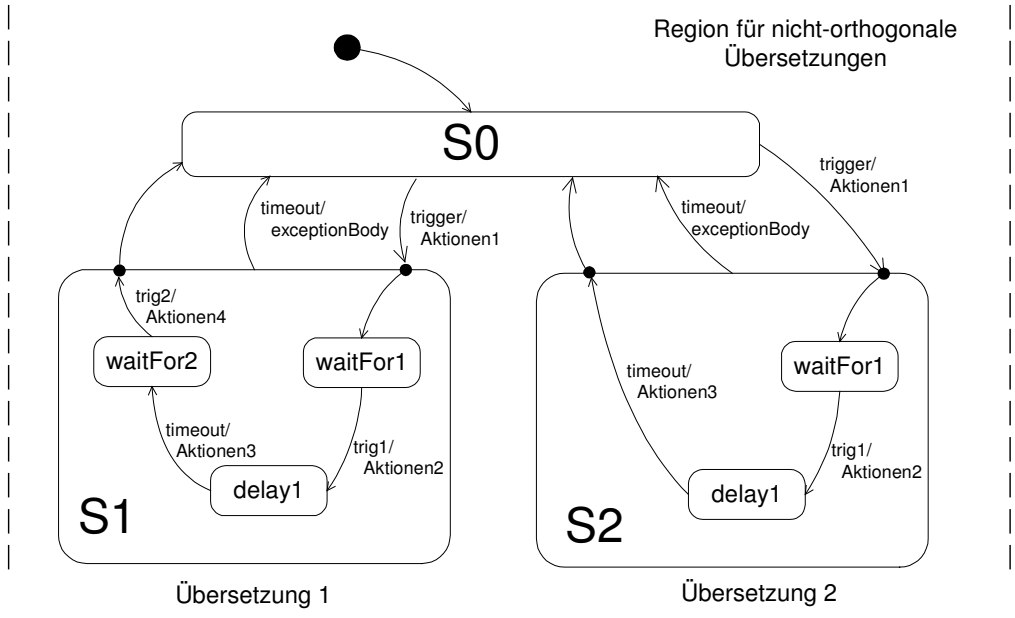


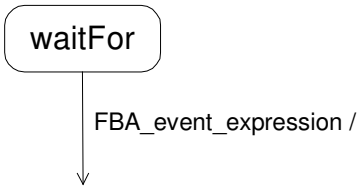
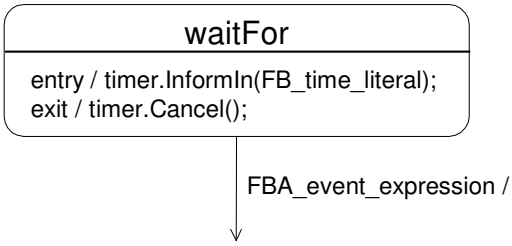
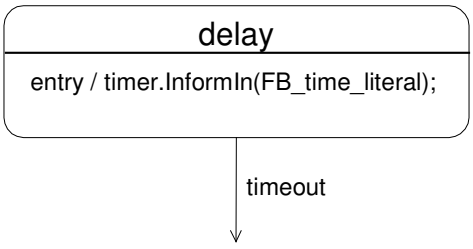
Abbildung 3.35 Region für nicht-orthogonale Übersetzungen

In dieser Region gibt es einen gemeinsamen Ausgangszustand *S0*, der von mehreren Transitionen (eine für jede nicht-orthogonale Übersetzung) verlassen wird. Für jede Übersetzung gibt es einen übergeordneten Zustand, der genauso wie bei orthogonalen

Übersetzungen aufgebaut ist. Die beiden Transitionen, die den übergeordneten Zustand einer Übersetzung verlassen, enden in dem gemeinsamen Zustand *S0*.

Abschließend für dieses Kapitel sollen in Tabelle 3.4 allgemeine Regeln angegeben werden, die die Syntax der FBA-Sprache auf UML-Elemente abbilden. Die in Tabelle 3.4 aufgelisteten Regeln wiederholen die Aussagen dieses Abschnitts, verwenden dabei aber Elemente der Grammatik der FBA-Sprache. Mit Hilfe dieser Regeln kann eine Graph-Grammatik entwickelt werden, die eine Generierung von Teilen eines FBA-Statecharts formal beschreibt [EndHev 2002]. Die in Tabelle 3.4 gezeigten Regeln könnten zur Generierung von Unter-Statecharts wie S1 oder S2 aus Abbildung 3.34 oder Abbildung 3.35 verwendet werden.

Tabelle 3.4 Semantik der FBA-Sprache

Syntax	Semantik
`'waitFor' '(' FBA_event_expression ')'`	
`'waitFor' '(' FBA_event_expression ',' FB_time_literal ')'`	
`'delay' '(' FB_time_literal ')'`	
`'send' '(' FBA_postfixExpression ')'`	«SendSignalAction»
`FBA_postfixExpression ':=' FB_constant`	«WriteAttributeValueAction» nur bei Ausgangsvariable zusätzlich: «SendSignalAction»
`FBA_postfixExpression`	Semantik der aufgerufenen Operation
`FBA_postfixExpression ':=' FBA_postfixExpression`	«ReadAttributeAction» «WriteAttributeValueAction» nur bei Ausgangsvariable zusätzlich: «SendSignalAction»

Wie bereits in den Kommentaren zu Abbildung 3.1 und zu Abbildung 3.2 angesprochen wurde, soll es auch möglich sein, eine beliebige Wertänderung einer Eingangsvariable als ChangeEvent verwenden zu können. Deshalb soll ein Ausdruck, der mit dem Namen einer Eingangsvariable beginnt und mit dem Suffix *.ValueChanged* endet (z.B.: *In.ValueChanged*), das Eintreffen eines ValueChanged-Signals über den FBInPort der Eingangsvariable kennzeichnen (siehe auch Abbildung 3.33).

3.7 Ansatz zur Verifikation von Funktionsbausteinadaptern

Das Ziel der Verifikation von Funktionsbausteinadaptern ist es, das Verhalten auf typische Spezifikationsfehler zu untersuchen. Typische Spezifikationsfehler sind zum Beispiel:

- Nichtspezifizierter Empfang: Es tritt ein Ereignis auf, das vom Funktionsbausteinadapter im aktuellen Zustand nicht erwartet wird, oder der Funktionsbausteinadapter erzeugt ein Ereignis, das in einem beteiligten Protokoll nicht erlaubt ist.
- Deadlock: Ein Funktionsbausteinadapter kommt in einen Zustand, der nicht mehr verlassen werden kann.
- Dynamischer Deadlock: Ein Funktionsbausteinadapter kommt in eine Art Endlosschleife, bei der mit der Umgebung entsprechend der Protokolle kommuniziert wird, aber der Ausgangszustand oder Endzustand nie erreicht wird.

Als Ergebnis der Verifikation sollte der Beweis stehen, dass die untersuchten Spezifikationsfehler noch enthalten sind oder nicht. Ein anzustrebendes Verfahren für die Verifikation ist die Modellprüfung (Modelchecking), weil es ein mathematisches Beweisverfahren ist, dessen Ergebnis für den gesamten Zustandsraum gilt [ClaGruPel 2000]. Beim Modelchecking muss die zu überprüfende Spezifikation in ein mathematisches Modell überführt werden. Das mathematische Modell ist meistens ein System kommunizierender endlicher Automaten oder einfach nur ein einzelner endlicher Automat. Nachdem das mathematische Modell erstellt wurde, müssen die zu untersuchenden Eigenschaften (Spezifikationsfehler) mit Hilfe der Temporalen Logik formuliert werden. Das Prinzip des Modelchecking besteht darin, nachzuweisen, ob das mathematische Modell die in Temporaler Logik spezifizierten Eigenschaften erfüllt oder nicht. Dazu muss der gesamte Zustandsraum des Automatenmodells untersucht werden. Zur Unterstützung dieses Beweisverfahrens ist wegen der Größe des Zustandsraums ein Werkzeug (Modelchecker) notwendig. Im Rahmen der Arbeit mit Funktionsbausteinadaptern wurde der Modelchecker SMV [SMV 2001] eingesetzt. Die darin verwendete Form Temporaler Logik ist die *Computation Tree Logic* (CTL). Die Umsetzung des Verhaltens von Funktionsbausteinadaptern in ein mathematisches Modell, das mit Hilfe der SMV-Sprache formuliert wird, erfolgt zurzeit noch manu-

ell. In diesem Abschnitt sollen allgemeine Hinweise zur Verifikation von Funktionsbausteinadaptern durch Modelchecking gegeben werden, die aber in konkreten Anwendungsbeispielen noch um spezifische Eigenschaften ergänzt werden müssen.

Bei der Überführung des Verhaltens von Funktionsbausteinadaptern in ein mathematisches Modell gibt es verschiedene Vorgehensweisen. Zunächst sollen die Gemeinsamkeiten dieser Vorgehensweisen erläutert werden, die darin bestehen, einen Funktionsbausteinadapter in endliche Automaten zu überführen. Ein Funktionsbausteinadapter kann in mindestens einen endlichen Automaten mit Ausgabe der Form

$$M_{fba} = (Q_{fba}, \Sigma_{fba}, \Delta_{fba}, \delta_{fba}, \lambda_{fba}, q_0)$$

überführt werden, wobei Q_{fba} eine endliche Menge von Zuständen, Σ_{fba} ein endliches Eingabealphabet, δ_{fba} die Übergangsfunktion, die $Q_{fba} \times \Sigma_{fba}$ auf Q_{fba} abbildet, $q_0 \in Q_{fba}$ der Ausgangszustand, Δ_{fba} ein endliches Ausgabealphabet und λ_{fba} die Ausgabefunktion von Q_{fba} auf Δ_{fba} ist [HopUll 1994]. Die Entwicklung dieser Automaten kann ähnlich wie bei der Entwicklung von Statecharts aus den Übersetzungen erfolgen (Abschnitt 3.6.7.3 und Tabelle 3.4). Dabei entspricht jede Region des FBA-Statecharts einem Automaten. Es entsteht dadurch für jede Übersetzung, bei der IsOrthogonal gesetzt ist, ein eigenständiger Automat und für alle Übersetzungen zusammen, bei denen IsOrthogonal nicht gesetzt ist, ein weiterer. Jeder dieser Automaten kann einzeln betrachtet werden, weshalb im Folgenden immer nur ein Automat stellvertretend für einen Funktionsbausteinadapter steht.

Das Eingabealphabet lässt sich aus den Nachrichten, die vom Funktionsbausteinadapter über Ports empfangen werden und aus Änderungsereignissen der Eingangsvariablen ableiten. Das Ausgabealphabet kann aus den Nachrichten, die über Ports gesendet werden und aus Änderungsereignissen in den Ausgangsvariablen abgeleitet werden. Die Zustandsmenge entspricht bei der einfachsten Umsetzung der Menge von Befehlen im TranslationBody des Funktionsbausteinadapters, ergänzt um einen Startzustand und ergänzt um einen Fehlerzustand, in den übergegangen werden soll, wenn der Funktionsbausteinadapter eine Eingabe erhalten hat, die nicht erwartet wurde. Unter Umständen können Lesebefehle zu einem gemeinsamen Zustand zusammengefasst werden. Die Zustandsübergangsfunktion ergibt sich aus der Reihenfolge der Befehle im TranslationBody einer Übersetzung bzw. aus den Transitionen des Statecharts der betrachteten Region des Funktionsbausteinadapters. Zusätzlich müssen von jedem Zustand, in dem nicht alle Eingabesymbole erwartet werden, Transitionen in den Fehlerzustand eingeführt werden, die bei unerwarteten Eingaben feuern. Die Ausgabefunktion entsteht dadurch, dass man jedem Zustand, der für einen Send-Befehl steht, und jedem Zustand, der ein Änderungsereignis in einer Ausgangsvariable erzeugt, das entsprechende Ausgabesymbol zuordnet.

Nachdem der Funktionsbausteinadapter in einen oder mehrere Automaten übersetzt wurde, gibt es zwei Wege für das weitere Vorgehen. Im ersten Weg wird der zu untersuchende Zustandsraum nur durch den Automat des Funktionsbausteinadapters bestimmt. Alle zu untersuchenden Eigenschaften und das Verhalten der Systemumgebung müssen mit Hilfe von CTL-Formeln beschrieben werden.

Im zweiten Weg müssen für die beteiligten Kommunikationspartner und für die Protokolle weitere Automaten entwickelt werden, sodass sich der Zustandsraum aus einem Produkt der kommunizierenden Automaten ergibt. Die Eigenschaften der Systemumgebung sind dann in diesem Automatensystem mit enthalten. Im zweiten Weg ist es hilfreich, dem Eingabe- und Ausgabealphabet der Automaten noch jeweils ein Symbol zuzufügen, das für keine Eingabe bzw. keine Ausgabe steht. Dadurch wird es leichter, das Produkt zu bilden. Die Kommunikation zwischen den Automaten wird durch Kommunikationsfunktionen bestimmt. In Abbildung 3.36 ist ein System kommunizierender Automaten dargestellt, wobei M_{port} für das UML-Protokoll, M_{fbp} für das FB-Protokoll, M_{class} für den Kommunikationspartner auf UML-Seite und M_{fb} für den Funktionsbaustein stehen. Die Kommunikationsfunktionen sind mit σ gekennzeichnet. Sie bilden Ausgaben der Automaten auf Eingaben anderer Automaten ab. Durch die Pfeilrichtungen in Abbildung 3.36 wird bestimmt, welche Ausgaben welcher Automaten auf die Eingaben welcher Automaten abgebildet werden. Ein Beispiel für ein solches Automatensystem wird in Abschnitt 4.1.2 beschrieben.

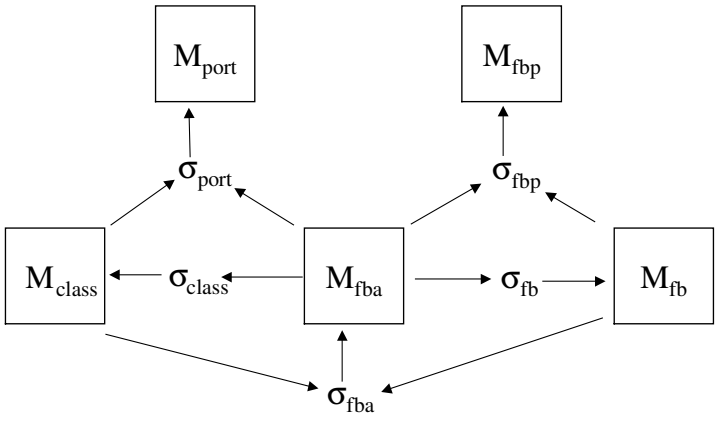


Abbildung 3.36 Kommunikation zwischen den Automaten

Der Unterschied zwischen der ersten und der zweiten Vorgehensweise ist, dass bei der ersten die Komplexität der CTL-Formeln höher ist, als bei der zweiten. Umgekehrt ist bei der zweiten Vorgehensweise das Automatenmodell komplexer als bei der ersten Vorgehensweise. Da für Ungeübte der Umgang mit CTL-Formeln und deren Formulierung schwieriger sind, als das Verständnis von Automaten, wurde im Abschnitt 4.1.2 die zweite Vorgehensweise gewählt. Dort finden sich weitere Hinweise zu dieser Vorgehensweise und auch die zur Untersuchung der Spezifikationsfehler notwendigen CTL-Formeln.

Funktionsbausteinadapter können auf mehreren Abstraktionsebenen verifiziert werden. Auf der höchsten wird eine Übersetzung nur einem Automat zugeordnet, wie das in Abschnitt 4.1.3 und Abbildung 3.36 geschehen ist. Berücksichtigt man, dass die Funktionsbausteininstanz und die UML-Instanz im Allgemeinen in unterschiedlichen Prozessen innerhalb eines Computersystems oder sogar auf getrennten Computersystemen arbeiten, muss noch ein zusätzliches funktionsbausteininternes Kommunikationsprotokoll betrachtet werden. Die Vorgehensweise für die Verifikation durch Modelchecking ist aber prinzipiell die gleiche, bis auf den Unterschied, dass jede Übersetzung durch drei Automaten modelliert wird, wobei einer das interne Protokoll repräsentiert und die beiden anderen die auf zwei Prozesse aufgeteilte Übersetzung.

3.8 Implementierung von Funktionsbausteinadaptern

Nachdem ein Funktionsbausteinadapter auf einer plattformunabhängigen Ebene mit Hilfe der FBA-Sprache vollständig beschrieben wurde, kann zur plattformabhängigen Implementierung übergegangen werden. Eine Besonderheit bei Funktionsbausteinadaptern ist es, dass sie normalerweise auf mindestens zwei verschiedenen Plattformen implementiert werden müssen. Eine Plattform basiert auf PC- oder Mikrocontroller-Technik und die andere auf SPS-Technik. Auch wenn keine Hardware-SPS eingesetzt wird, so läuft der Funktionsbaustein zumindest in einem eigenständigen Prozess (einer Soft-SPS) innerhalb der PC-Plattform. Für einen Funktionsbausteinadapter bedeutet das, dass er immer Prozessgrenzen oder sogar Computersystemgrenzen überschreiten muss. Intern besteht ein Funktionsbausteinadapter aus zwei Teilen, wobei ein Teil auf der PC- oder Mikrocontroller-Plattform und der andere auf der SPS-Plattform arbeitet. Der erste funktionsbausteinadapterinterne Teil soll C-Prozess und der andere F-Prozess genannt werden.

In Abbildung 3.37 wurden diese Zusammenhänge grafisch dargestellt. Die zwei grau unterlegten Rechtecke stehen für zwei unterschiedliche Computersysteme, die über ein Netzwerk miteinander verbunden sind. Für die Kommunikation über das Netzwerk gibt es plattformabhängige Kommunikationsdienste, die funktionsbausteinadapterintern verwendet werden. Die zu einem Funktionsbausteinadapter gehörenden Teile sind in Abbildung 3.37 gestrichelt eingerahmt.

Der C-Prozess hat die Aufgaben

- Kommunikation mit der UML-Instanz über Ports entsprechend des UML-Protokolls,
- Kommunikation mit dem F-Prozess über das Netzwerk entsprechend des funktionsbausteinadapterinternen Protokolls und

- Konvertierung der Daten aus dem UML-Protokoll in das funktionsbausteinadapterinterne Protokoll und umgekehrt.

Der F-Prozess hat die Aufgaben

- Kommunikation mit der Funktionsbausteininstanz über die Schnittstellenvariablen entsprechend des vereinbarten FB-Protokolls,
- Kommunikation mit dem C-Prozess über das Netzwerk entsprechend des funktionsbausteinadapterinternen Protokolls und
- Konvertierung der Daten aus dem FB-Protokoll in das funktionsbausteinadapterinterne Protokoll und umgekehrt.

Die Implementierung des F-Prozesses erfolgt in einer Sprache der IEC 61131-3 und die Implementierung des C-Prozesses in einer objektorientierten Sprache wie C++ oder Java. In Abschnitt 4.2 wird ein Anwendungsbeispiel vorgestellt, bei dem der C-Prozess mit Hilfe von Statecharts/C++ und der F-Prozess mittels der Ablaufsprache/Strukturiertem Text realisiert wurden. Für das Netzwerk wurde ein Feldbus vom Typ PROFIBUS-DP eingesetzt. Weitere Hinweise zur Implementierung von Funktionsbausteinadaptern finden sich in Abschnitt 4.2.

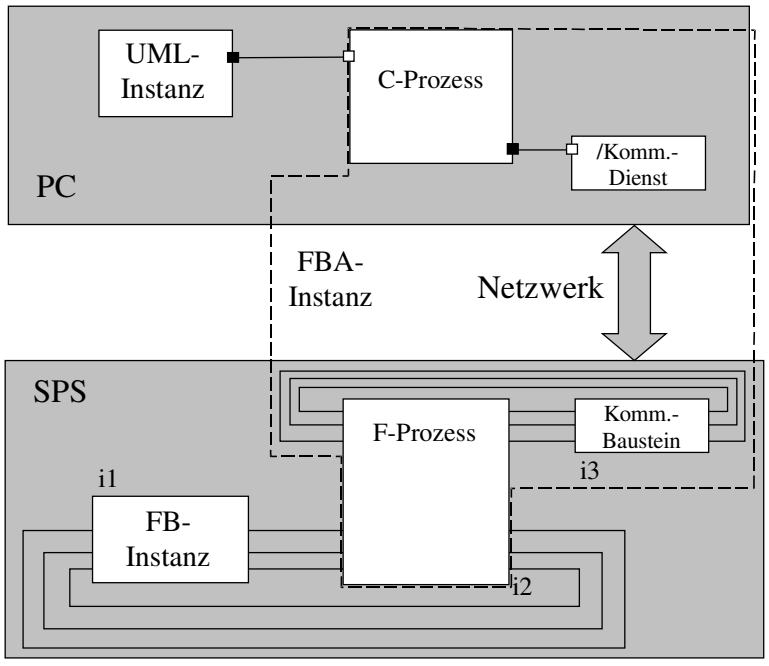


Abbildung 3.37 Schema für die Implementierung

Im nächsten Abschnitt soll das Konzept der Funktionsbausteinadapter in das in Abschnitt 2.3 vorgestellte ViewPoint-Framework eingebettet werden.

3.9 Einbettung in das ViewPoint-Framework

Funktionsbausteinadapter stellen im Vergleich zu reinen UML-Klassen und auch zu Funktionsbausteinen eine eigenständige Sichtweise dar. Sie werden deshalb durch eigenständige ViewPoints repräsentiert. Ein erster Ansatz für eine Konfiguration von ViewPoint-Templates wurde bereits in [EndHev 2002] vorgestellt. Während dort ein formaler Ansatz gewählt wurde, soll an dieser Stelle eine nichtformale Erläuterung erfolgen, die keinen Anspruch auf Vollständigkeit hat, aber der Einordnung in einen größeren Zusammenhang eines Softwareentwicklungsprozesses dienen soll.

Das Konzept der Funktionsbausteinadapter beinhaltet insgesamt drei Darstellungsweisen, die eigenständige ViewPoints verdienen. Zwei der Sichtweisen beziehen sich auf statische Zusammenhänge, die in Klassendiagrammen und Strukturdiagrammen der UML bzw. in der Funktionsbausteinsprache dargestellt werden können. Die Sichtweise auf einen Funktionsbausteinadapter, die sich bei dessen Darstellung in einem Klassendiagramm ergibt, wird durch ein ViewPoint-Template VPT_{FBA} (Abbildung 3.38 rechts unten) beschrieben.

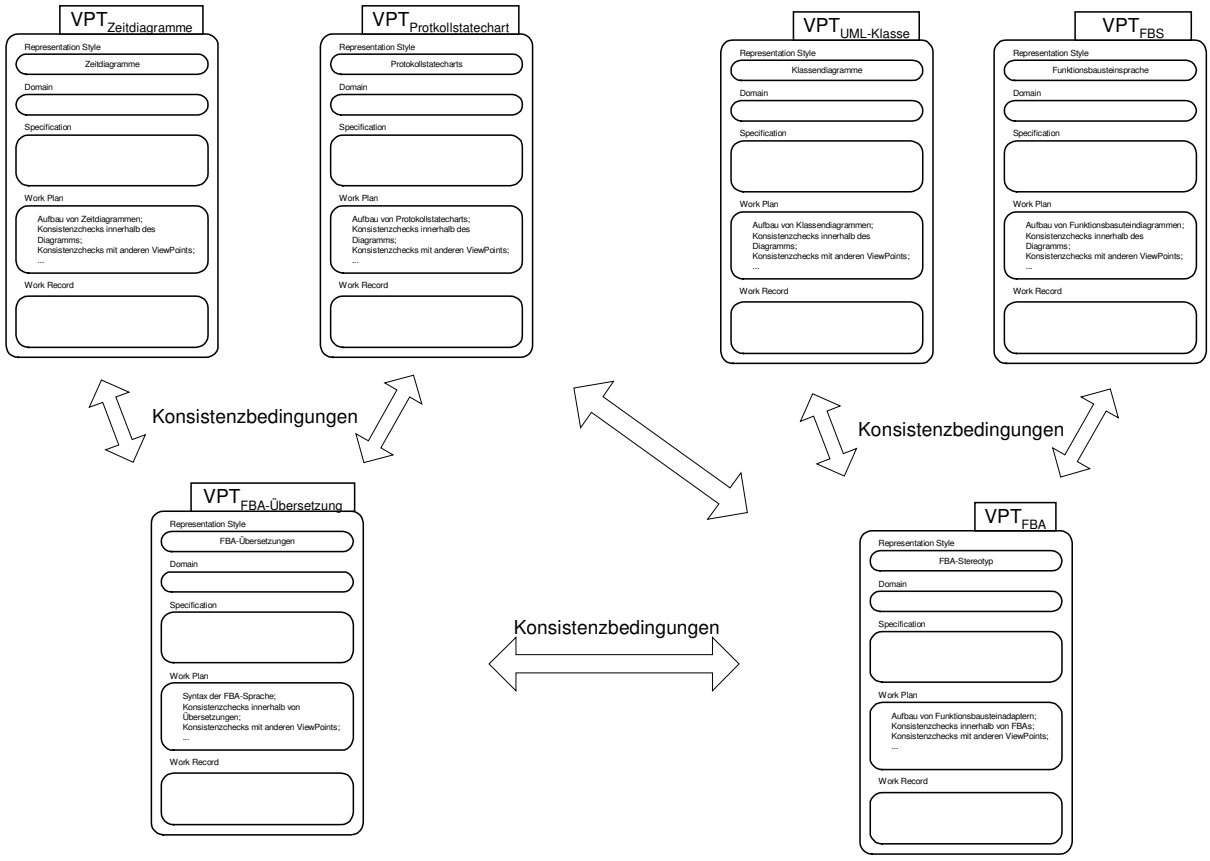


Abbildung 3.38 ViewPoint-Templates $VPT_{FBA\text{-}Übersetzung}$ und VPT_{FBA}

Die Beschreibung des Representation Style, der in VPT_{FBA} eingesetzt werden darf, liefern die Abschnitte 3.6.6.1 und 3.6.5.1. In der Specification ist alles erlaubt, was auch in normalen Klassendiagrammen erlaubt ist. Die Sichtweise von VPT_{FBA} unterscheidet sich von der normaler Klassendiagramme dadurch, dass sie um Eingangs-

und Ausgangspins ergänzt wurde. Der Work Plan enthält die Hinweise und Vorschriften, die bereits in den Abschnitten 3.6.1 bis 3.6.6 gegeben wurden. Konsistenz muss mit Klassendiagrammen (ViewPoint-Template $VPT_{UML\text{-}Klassen}$), in denen die Klassen dargestellt sind, die mit Funktionsbausteinen verbunden werden müssen, und mit Funktionsbausteindiagrammen (ViewPoint-Template VPT_{FBS}) sichergestellt werden, in denen Funktionsbausteine enthalten sind, die mit UML-Klassen verbunden werden sollen.

Zu den wichtigsten Konsistenzregeln, die im Zusammenhang mit anderen ViewPoints beachtet werden müssen, zählen:

- Es müssen Eingangs- und Ausgangspins definiert sein, die mit dem Funktionsbaustein, der an UML-Ports gekoppelt werden soll, verbunden werden können (siehe auch Abschnitt 2.1.3).
- Es müssen Ports mit geeigneten Protokollen enthalten sein, die mit den Ports der zu verbindenden UML-Klassen kompatibel sind (siehe auch Abschnitt 2.2.2).

Im Zusammenhang mit ViewPoints für Übersetzungen (ViewPoint-Template $VPT_{FBA\text{-}Übersetzung}$) gibt es ebenfalls Konsistenzbedingungen, die weiter unten erläutert werden.

Das ViewPoint-Template $VPT_{\text{FBA-Übersetzung}}$ in Abbildung 3.38 unten links ist die Vorlage von ViewPoints für Übersetzungen. Die in Representation Style und Work Plan gehörenden Inhalte wurden bereits in Abschnitt 3.6.7 dargestellt. Die wichtigsten Konsistenzregeln im Zusammenhang mit anderen ViewPoints sind:

- Es können nur Schnittstellenvariablen, die im Funktionsbausteinadapter deklariert sind, verwendet werden.
- Es können nur Nachrichten empfangen oder versendet werden, die in den Ports des Funktionsbausteinadapters erlaubt sind.
- Nachrichten dürfen nur in der Reihenfolge versendet oder empfangen werden, wie es in den UML-Protokollen vorgeschrieben wird.
- Die Belegungen von Schnittstellenvariablen dürfen nur in der im FB-Protokoll vorgeschriebenen Weise erfolgen.

Die sicherste, aber auch die aufwendigste Methode, um die Einhaltung der Protokolle innerhalb einer Übersetzung zu gewährleisten, ist die Verifikation durch Modelchecking (Abschnitt 3.7). Weiterhin sinnvoll sind Simulation und Testen des Verhaltens der Übersetzungen.

Um die Darstellung von Instanzen oder Rollen von Funktionsbausteinadaptern in Strukturdiagrammen unterstützen zu können, wird noch ein weiteres ViewPoint-Template $VPT_{\text{FBA-Instanz}}$ in Abbildung 3.39 eingeführt.

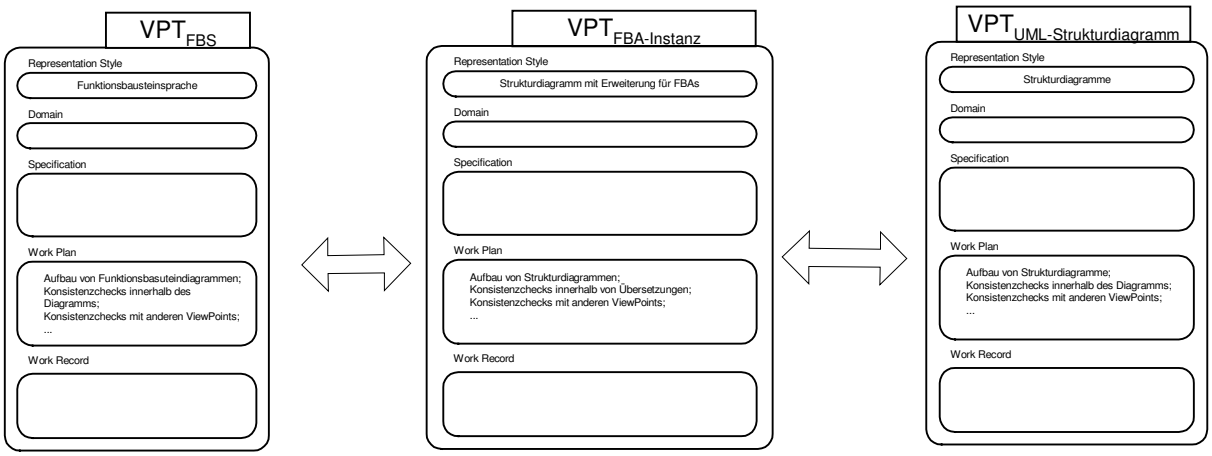


Abbildung 3.39 ViewPoint-Template für FBA-Strukturdiagramme

Ein Beispiel für eine FBA-Instanz ist in Abbildung 3.6 zu sehen. Der Unterschied zu normalen Instanzen der UML besteht nur darin, dass die FBPorts wie Eingangs- oder Ausgangspins von Funktionsbausteinen dargestellt und miteinander verbunden werden (siehe auch Abschnitt 0 und Abbildung 3.24). Insgesamt gelten für die Darstellung von FBA-Instanzen die Regeln der UML-Strukturdiagramme. Wenn ein und dieselbe FBA-Instanz in mehreren Diagrammen vorkommt, z.B. als Instanz eines Funktionsbausteins in einem Funktionsbausteindiagramm, dann muss zwischen diesen Diagrammen (Sichtweisen) sichergestellt werden, dass die Verbindungen zwischen Ports bzw. Pins immer identisch sind.

Das ViewPoint-Template VPT_{FBS} steht für die Funktionsbausteinsprache der IEC 61131-3. Im nächsten Abschnitt werden weitere funktionsblockorientierte Sprachen vorgestellt, die ebenfalls für die Integration in die UML über Funktionsbausteinadapter geeignet sind.

4 Anwendungsbeispiele

Zur besseren Verdeutlichung der Wirkungsweise und Einsatzmöglichkeiten von Funktionsbausteinadaptern werden in diesem Kapitel zwei Anwendungsbeispiele vorgestellt. Das erste Beispiel wurde bereits in [Hev 2003] beschrieben und soll in Abschnitt 4.1 weiter ergänzt und aktualisiert werden. Es soll die Möglichkeit der Verifikation aufzeigen, die bereits in einem frühen Stadium des Softwareentwurfes gegeben ist. Das zweite Beispiel erläutert, wie ein Funktionsbausteinadapter in einem realen System implementiert werden kann. Dieses Beispiel wurde ebenfalls bereits in verschiedenen Veröffentlichungen und Berichten verwendet ([HeShGrTr 2002], [HevTra 2001a], [HevTra 2001d], [EnGoHeTr 2001]). Da in der hinter diesem Beispiel stehenden Fertigungsanlage das Konzept des Funktionsbausteinadapters als erstes und am vollständigsten angewendet wurde, soll es im Abschnitt 4.2 mit aufgeführt werden.

4.1 Anwendungsbeispiel „Motion Control“

In diesem Anwendungsbeispiel soll das logische, plattformunabhängige Verhalten eines Funktionsbausteinadapters auf typische Spezifikationsfehler untersucht werden. Es gibt zurzeit noch keine Werkzeuge, die eine gemeinsame Verifikation von Funktionsbausteinen und UML-Klassen erlauben. Der in diesem Anwendungsbeispiel gewählte Ansatz zur Verifikation von Funktionsbausteinadapters nutzt deshalb nur die Schnittstellenbeschreibungen von Funktionsbausteinen und UML-Klassen. Als Vorarbeit können Funktionsbausteine und UML-Klassen getrennt voneinander durch geeignete Werkzeuge verifiziert werden, um sicherzustellen, dass sie ihre jeweiligen Schnittstellenprotokolle einhalten. Die jeweiligen Protokolle sollen in diesem Beitrag entsprechend FB-Protokoll und UML-Protokoll heißen. Sie werden in den Abschnitten 4.1.1.1 und 4.1.1.2 vorgestellt.

Als notwendige Ergänzung zur getrennten Verifikation der UML-Klassen und der Funktionsbausteine wird ein Funktionsbausteinadapter dahingehend verifiziert, ob er die Übersetzung zwischen UML-Protokoll und FB-Protokoll fehlerfrei durchführt. Diese Verifikation erfolgt durch Modelchecking mit Hilfe des Werkzeuges SMV [SMV 2001]. Dazu müssen ein Funktionsbausteinadapter und dessen Protokolle in ein System kommunizierender endlicher Automaten überführt werden, um ein für den Modelchecker geeignetes mathematisches Modell zu erhalten [ClaGruPel 2000].

Dieses Anwendungsbeispiel gliedert sich weiterhin wie folgt: Zunächst wird im Abschnitt 4.1.1 ein konkretes Beispiel für einen Funktionsbausteinadapter und dessen Schnittstellenprotokolle eingeführt und erläutert. Im Abschnitt 4.1.2 wird als notwendige Vorarbeit zur Verifikation ein System kommunizierender endlicher Automaten

entwickelt, das den Beispiel-Funktionsbausteinadapter und dessen Umgebung modelliert. Das entwickelte Automatensystem wird in Abschnitt 4.1.3 als Modell zur Verifikation des Funktionsbausteinadapter verwendet. Eine Zusammenfassung schließt dieses Anwendungsbeispiel in Abschnitt 4.1.4 ab.

4.1.1 Beispiel „Bohrung“

Als Anwendungsbeispiel für einen Funktionsbausteinadapter soll eine einfache Bohreinheit dienen. Dieses Beispiel ist dem PLCopen Standard „Function blocks for motion control“ [PLCOpenDrill] entnommen. Eine Bohrung soll wie in Abbildung 4.1 dargestellt durchgeführt werden.

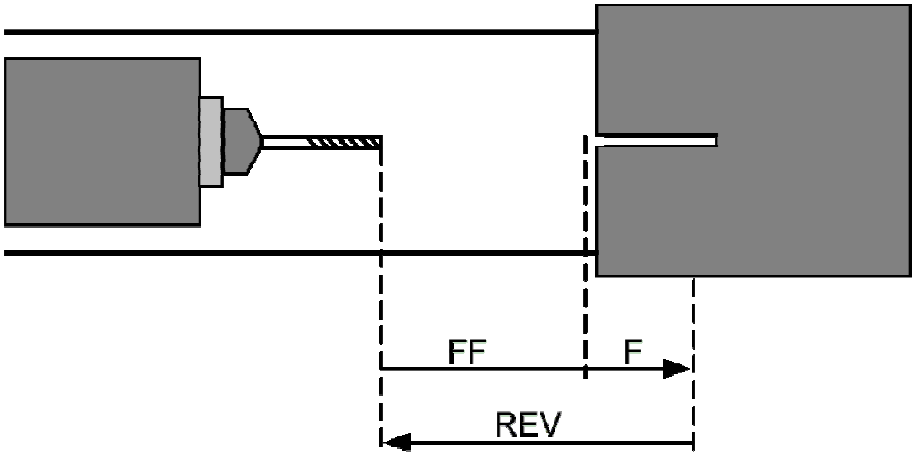


Abbildung 4.1 Prinzipschema Bohrung (Quelle: [PLCOpenDrill])

Sie verläuft in vier Zeitabschnitten:

ffwd: Die Bohrmaschine wird mit hoher Geschwindigkeit an das Werkstück herangefahren. Dabei wird gleichzeitig die Drehbewegung der Spindel gestartet.

fwd: Die Bohrung wird mit langsamer Bohrgeschwindigkeit durchgeführt.

dwell: Die Bohrmaschine verbleibt für 500 ms in der Bohrung, um alle Späne aus der Bohrung zu entfernen.

rev: Die Bohrmaschine wird anschließend auf die Ausgangsposition zurückgefahren. Dabei wird die Drehbewegung der Spindel gestoppt.

Abbildung 4.2 zeigt ein Zeitdiagramm für Weg und Geschwindigkeit der translatorischen Bewegung der Bohrmaschine. Jeder Abschnitt dieser Bewegung soll mit einem Funktionsbaustein gesteuert werden. Abbildung 4.3 zeigt vier Funktionsbausteininstanzen, deren Namen den Abschnitten in Abbildung 4.2 entsprechen.

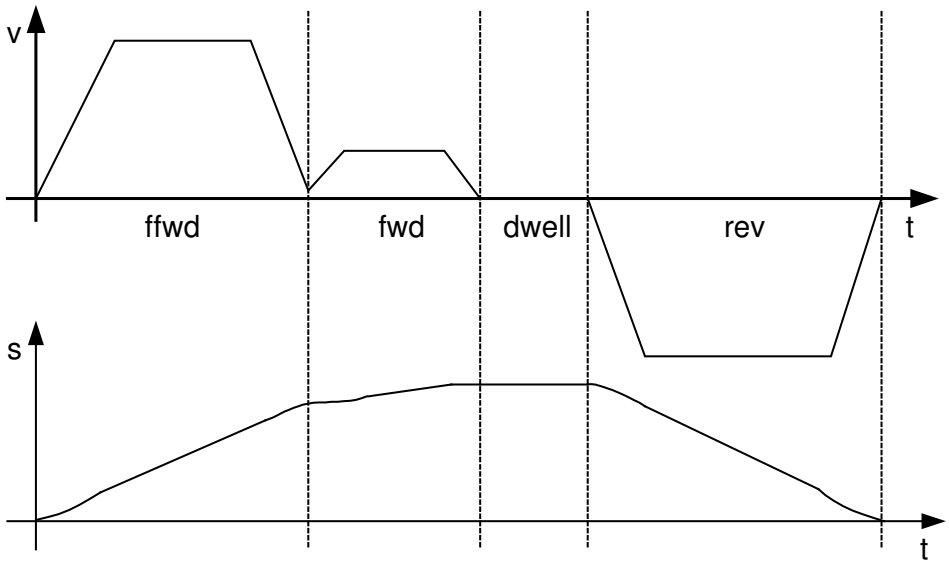


Abbildung 4.2 Geschwindigkeit und Weg (Quelle: [PLCOpenDrill])

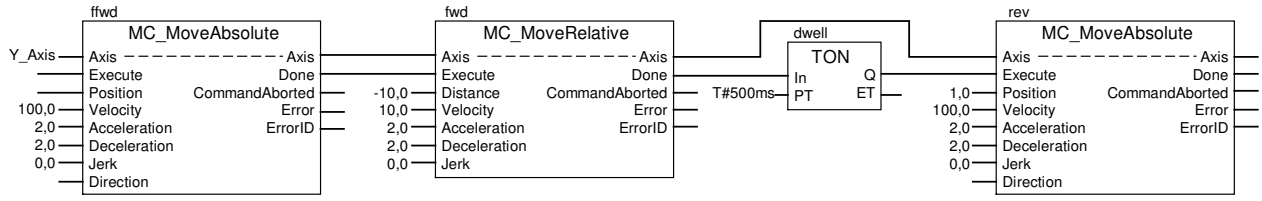


Abbildung 4.3 FB-Diagramm (Quelle: [PLCOpenDrill])

ffwd und *rev* sind Instanzen von *MC_MoveAbsolute*, denen eine absolute Zielposition für die Bewegung vorgegeben wird. *fwd* ist vom Typ *MC_MoveRelative*, bei dem ein Abstand relativ zur aktuellen Position angegeben werden muss. *dwell* ist einfach eine Einschaltverzögerung (*TON*), wie sie in IEC 61131-3 definiert ist. Die meisten Eingänge der Funktionsbausteininstanzen sind mit konstanten Werten belegt. Die *Direction* Eingänge müssen nicht mit Werten versehen werden.

Das Starten des Bewegungsablaufes erfolgt durch eine positive Flanke in *ffwd.Execute*. In *ffwd.Position* muss abhängig vom Werkstück die Ausgangsposition vorgegeben werden. Ist der Bewegungsablauf vollständig abgeschlossen, wird das durch eine positive Flanke in *rev.Done* signalisiert. Im Fehlerfall wird eine positive Flanke in *Error* ausgegeben. Ein Fehlercode wird in *ErrorID* bereitgestellt. Wird aufgrund einer vorzeitigen negativen Flanke in *ffwd.Execute* der Bohrvorgang abgebrochen, gibt ein gerade aktiver Funktionsbaustein eine positive Flanke in *CommandAborted* aus.

Da für die weiteren Betrachtungen nur die offenen Eingangs- und Ausgangsvariablen von Bedeutung sind, soll die Bewegungssteuerung aus Abbildung 4.3 im nächsten Abschnitt zu einem übergeordneten Funktionsbaustein zusammengefasst werden.

4.1.1.1 Das Protokoll der Bewegungssteuerung

Abbildung 4.4 zeigt den Funktionsbaustein *MC_FB*, der die Funktionsbausteininstanzen aus Abbildung 4.3 enthält und zusammenfasst. Die Eingangsvariable *Execute* ist direkt mit *ffwd.Execute* aus Abbildung 4.3 verbunden. Die Ausgangsvariable *Ended* kennzeichnet die Beendigung des Bohrvorganges, was aufgrund eines Abbruchs, eines Fehlers oder der vollständigen Ausführung des Bohrvorganges geschehen kann. Abbildung 4.5 zeigt die Beschaltung von *Ended*. Die übrigen Ausgangsvariablen sind direkt mit Ausgängen aus Abbildung 4.3 verbunden (z.B. *Done* mit *rev.Done*, *ffwd_CA* mit *ffwd.CommandAborted*, usw.).

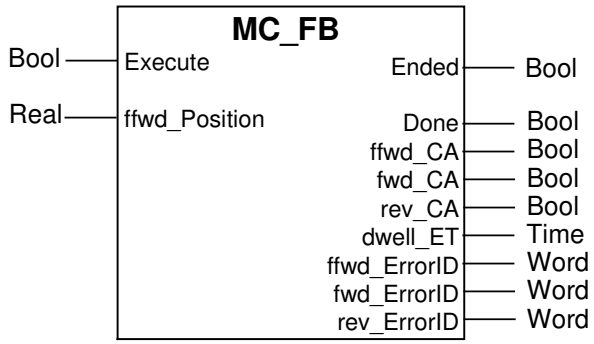


Abbildung 4.4 Zusammengefasster Funktionsbaustein MC_FB

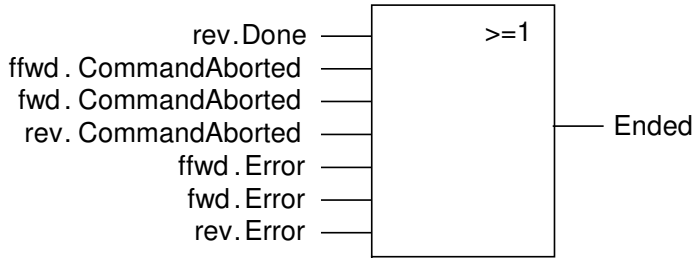


Abbildung 4.5 Schaltung für das Ended-Signal

Das Zeitdiagramm in Abbildung 4.6 zeigt die geforderte zeitliche Belegung der Eingangs- und Ausgangsvariablen von *MC_FB*. Die Ausgangsvariablen, abgesehen von *Ended*, sind unter „Ausgangsdaten“ zusammengefasst. Wenn *Ended* = *True* gilt, darf sich kein Wert in den Ausgangsdaten ändern. Wenn *Ended* = *False* sollen die Ausgangsdaten nicht genutzt werden, und können sich deshalb beliebig verhalten. Gleiches gilt für den Zusammenhang zwischen *Execute* und *ffwd_Position*.

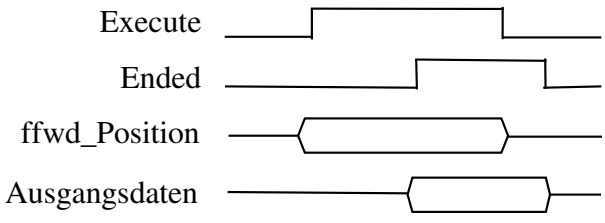


Abbildung 4.6 Zeitdiagramm für das Protokoll des Funktionsbausteines

Eine positive Flanke in *Execute* startet den Bohrvorgang. Eine positive Flanke in *Ended* signalisiert das Ende des Bohrvorganges. Eine negative Flanke in *Execute* teilt dem *MC_FB* mit, dass die Ausgangsdaten übernommen wurden. Eine negative Flan-

ke in *Ended* bedeutet, dass die Ausgangsdaten nicht mehr gültig sind. Eine positive Flanke in *Execute* ist nur erlaubt, wenn *Ended* = *False* ist.

Dieses Protokoll soll im Weiteren auch FB-Protokoll genannt werden. Über dieses Protokoll muss der Bewegungssteuerung der Startbefehl von der Gesamtsteuerung der Arbeitszelle mitgeteilt werden. Die Gesamtsteuerung der Arbeitszelle soll im weiteren Bohrsteuerung genannt werden. Die Bohrsteuerung hat unter anderem die Aufgabe, die von einem Vision System gelieferte genaue Position des Werkstückes in die Positionsvorgabe für *ffwd_Position* umzurechnen.

4.1.1.2 Das UML-Protokoll der Bohrsteuerung

Es soll davon ausgegangen werden, dass der Befehl zum Starten des Bohrvorganges zusammen mit der Ausgangsposition von einer Bohrsteuerung gesendet wird, die objektorientert mit Hilfe der UML/C++ entwickelt wurde. Im Abbildung 4.7 sind zwei UML-Interfaces *IDrillEnd* und *IDrillStart* dargestellt. *IDrillStart* muss von der Bewegungssteuerung implementiert werden, damit sie gestartet werden kann. *IDrillEnd* muss von der Bohrsteuerung implementiert werden, damit sie über das Ende des Bohrvorganges benachrichtigt werden kann.

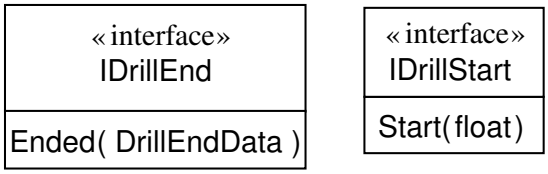


Abbildung 4.7 Interface-Klassen

Die Operation *Ended* von *IDrillEnd* hat einen Parameter vom Typ der Klasse *DrillEndData* (Abbildung 4.8). Damit der Zustand der Bewegungssteuerung ausgewertet werden kann, enthält *DrillEndData* Attribute für jede Ausgangsvariable von *MC_FB*. Als Datentypen wurden geeignete C++ Typen gewählt. Da es für den Typ *Time* der IEC 61131-3 keine direkte Entsprechung in C++ gibt, wurde der Typ *PLC_Time* in C++ definiert.

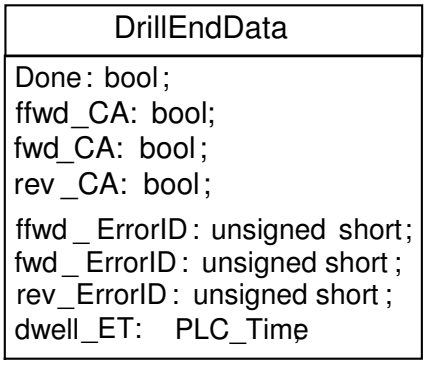


Abbildung 4.8 UML-Klasse DrillEndData

Die Operation *Start* von *IDrillStart* enthält im Parameter vom Typ *float* die Ausgangsposition der Bohrung.

Die UML wurde in Version 2.0 [UML Superstructure] um so genannte Ports erweitert, die bereits aus [SelGulWar 1994] und [SelRum 1999a] bekannt sind. In Abbildung 4.9 links wird die UML-Klasse *DrillingControl* mit dem Port *mcPort* definiert. Ports werden mit *required* (erforderlichen) und *provided* (angebotenen) Schnittstellen typisiert. Die Operationen der angebotenen Schnittstellen werden im Port zu einer Liste eingehender Signale, während die Operationen der erforderlichen Schnittstellen zu einer Liste ausgehender Signale zusammengefasst werden. Für *mcPort* ist damit *Ended* ein eingehendes Signal und *Start* ein ausgehendes Signal. Einem Port kann auch ein Protokoll-Statechart zugeordnet werden, mit dem eine Reihenfolge für die Signale des Ports festgelegt werden kann (Abbildung 4.9 rechts). Das damit definierte Verhalten soll zur Unterscheidung vom FB-Protokoll im Folgenden als UML-Protokoll bezeichnet werden.

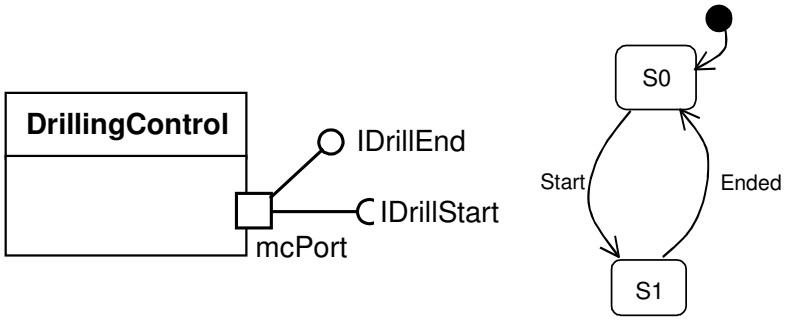


Abbildung 4.9 UML-Klasse DrillingControl mit Port mcPort und Protokoll-Statechart für das mcPort

Das Protokoll-Statechart aus Abbildung 4.9 rechts zeigt, dass die beiden Signale des Protokolls nur abwechselnd, beginnend mit *Start*, gesendet werden dürfen. Dieses Statechart hat nur beschreibenden Charakter. Die Transitionen können deshalb keine Aktionen enthalten. Für die Verifikation der zu entwickelnden Anwendung kann ein Protokoll-Statechart aber sehr sinnvoll eingesetzt werden. Das soll in Abschnitt 4.1.2 gezeigt werden.

Im nächsten Abschnitt wird das UML-Protokoll durch einen Adapter auf das FB-Protokoll abgebildet. Dieser Adapter wird dabei zuerst als Funktionsbausteinadapter dargestellt, um anschließend zusammen mit den Protokollen vereinfacht in endliche Automaten überführt zu werden.

4.1.1.3 Der Funktionsbausteinadapter MC_FBA

Damit der Funktionsbaustein *MC_FB* und die Klasse *DrillingControl* miteinander kommunizieren können, benötigt man einen Protokolladapter. Der in Abbildung 4.10 dargestellte Stereotyp «functionblockadapter» einer UML-Klasse dient als ein solcher

Protokolladapter, der speziell für die Umsetzung von FB-Protokollen auf UML-Protokolle entwickelt wurde (siehe auch Kapitel 3).

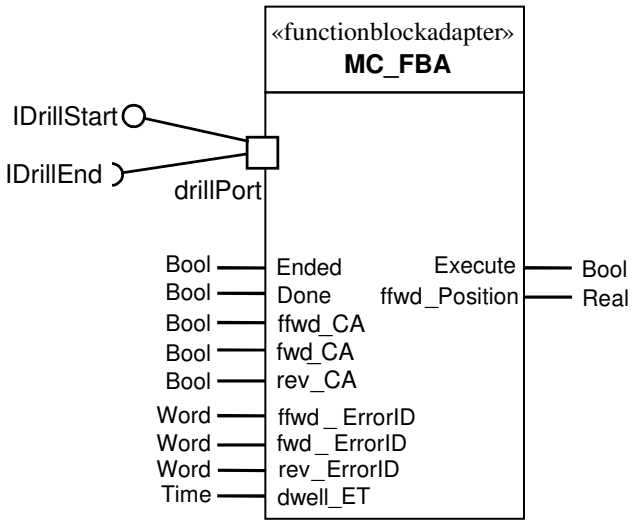


Abbildung 4.10 Klasse für MC_FBA

Ein Funktionsbausteinadapter besitzt gleichermaßen Schnittstellen zu einem Port einer UML-Klasse wie auch zu einem Funktionsbaustein. Der *MC_FBA* aus Abbildung 4.10 kann über das *drillPort* mit dem *mcPort* von *DrillingControl* verbunden werden. Das wird dadurch gewährleistet, dass *drillPort* die Schnittstellen anbietet, die von *mcPort* gefordert werden und umgekehrt. Außerdem muss *drillPort* das gleiche Protokoll-Statechart wie *mcPort* enthalten.

Damit der FBA in einem Funktionsblockdiagramm mit dem *MC_FB* zusammengeschaltet werden kann, müssen in ihm geeignete Schnittstellenvariablen deklariert werden. Der Einfachheit halber erhalten diese Variablen den gleichen Namen wie die Variablen von *MC_FB*.

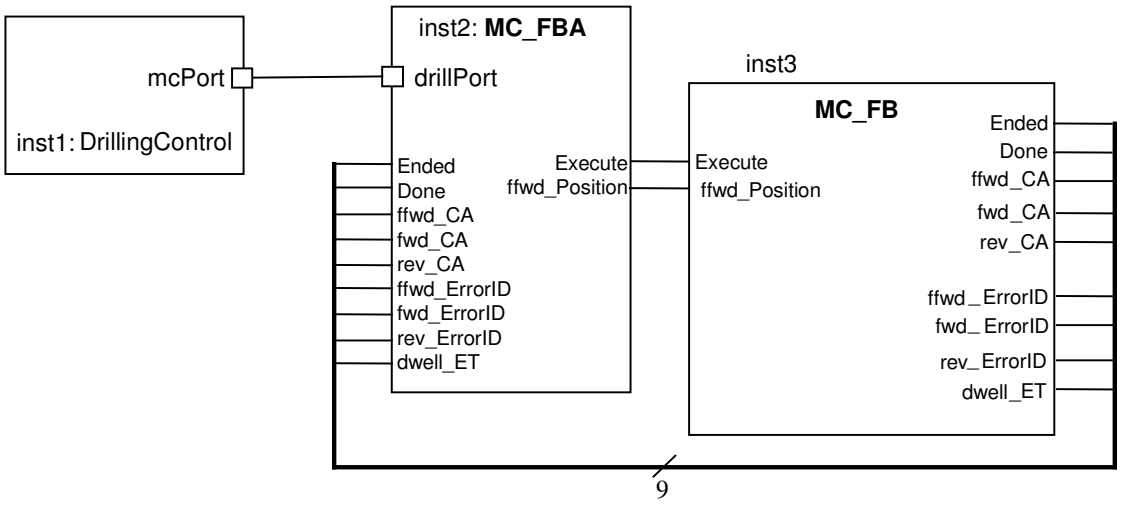


Abbildung 4.11 Strukturdiagramm kombiniert mit der FB-Sprache

In Abbildung 4.11 ist eine Instanz *inst2* von *MC_FBA* dargestellt, die mit einer Instanz *inst1* von *DrillingControl* und einer Instanz *inst3* von *MC_FB* verbunden ist.

Die Diagrammart aus Abbildung 4.11 ist eine Kombination eines Strukturdiagramms aus UML Version 2.0 [UML Superstructure] und der Funktionsbausteinsprache aus IEC 61131-3, wie in [HevTra 2001a] erstmals vorgeschlagen wurde.

Das dynamische Verhalten von FBAs wird durch deren FBATranslations (Übersetzungen) beschrieben. Der *MC_FBA* besitzt nur eine Übersetzung, die in Abbildung 4.12 angegeben ist.

```
(1)  trigger: (s1: drillPort.Start)
(2)  signals: (s1: drillPort.Start, s2: drillPort.Ended)
(3)  translationBody: {
(4)    ffwd_Position := s1;
(5)    Execute := true;
(6)    waitFor(Ended);
(7)    s2.Done := Done;
(8)    s2.ffwd_CA := ffwd_CA;
(9)    s2.fwd_CA := fwd_CA;
(10)   s2.rev_CA := rev_CA;
(11)   s2.ffwd_ErrorID := ffwd_ErrorID;
(12)   s2.fwd_ErrorID := fwd_ErrorID;
(13)   s2.rev_ErrorID := rev_ErrorID;
(14)   s2.dwell_ET := dwell_ET;
(15)   Execute := false;
(16)   waitFor(Not Ended);
(17)   send(s2);
(18) }
```

Abbildung 4.12 FBATanslation in FBA-Sprache

Die verwendete Spezifikationssprache ist eine Mischung aus Strukturiertem Text der IEC 61131-3 und UML-Syntax. Diese Sprache wird zur Spezifikation der logischen Abläufe verwendet, um eine eindeutige Beschreibung der Protokollumsetzung und der Abbildung der Datentypen zu erhalten. Die Implementierung geschieht durch eine Kombination von objektorientierten Sprachen wie C++ und Sprachen aus der IEC 61131-3 ([HevTra 2001b], [HeShGrTr 2002]).

Wenn das *Start*-Signal vom *MC_FBA* empfangen wird, beginnt die Abarbeitung der Operation. Zunächst wird die im *Start*-Signal enthaltene Position in die Variable *ffwd_Position* geschrieben. Dann wird eine positive Flanke in *Execute* ausgegeben. Anschließend wird auf eine positive Flanke in *Ended* gewartet. Ist diese eingetroffen, werden die Ausgangsdaten vom *MC_FB* in das *Ended*-Signal geschrieben. Danach wird mit einer negativen Flanke in *Execute* die Übernahme der Daten bestätigt. Das *Ended*-Signal des UML-Protokolls wird erst an *DrillingControl* geschickt, nachdem in *Ended* des FB-Protokolls eine negative Flanke erkannt wurde.

Eine weitere Erläuterung der hinter dieser FBATranslation stehenden Semantik erfolgt in Abschnitt 4.1.2.3 anhand eines endlichen Automaten.

4.1.2 Modellierung durch endliche Automaten

Für die spätere Verifikation des *MC_FBA* ist es erforderlich, das Verhalten von *DrillingControl*, UML-Protokoll, *MC_FBA*, FB-Protokoll und *MC_FB* durch endliche Automaten zu beschreiben. Es kommen dabei zwei Typen endlicher Automaten zum Einsatz.

Ein endlicher Automat ohne Ausgabe ist ein Viertupel

$$M = (Q, \Sigma, \delta, q_0),$$

wobei Q eine endliche Menge von Zuständen, Σ ein endliches Eingabealphabet, δ die Übergangsfunktion, die $Q \times \Sigma$ auf Q abbildet und $q_0 \in Q$ der Ausgangszustand ist [HopUll 1994]. Für die Protokolle reichen Automaten dieser Art aus, da sie nur der Kontrolle dienen, ob von den kommunizierenden Automaten die Protokolle eingehalten werden.

MC_FBA, *DrillingControl* und *MC_FB* werden als nichtdeterministische Moore-Automaten ([HopUll 1994]) modelliert:

$$M = (Q, \Sigma, \Delta, \delta, \lambda, q0).$$

Dabei ist Δ ein endliches Ausgabealphabet und λ die Ausgabefunktion von Q auf Δ.

4.1.2.1 Beschreibung des UML-Protokolls durch einen endlichen Automat M_{port}

Der endliche Automat für das UML-Protokoll ist ausgabelos, da das *drillPort*, das dieses Protokoll implementiert, nur zur Beobachtung des Protokolls eingesetzt werden soll:

$$M_{port} = (Q_{port}, \Sigma_{port}, \delta_{port}, q_0).$$

Die Zustandsmenge Q_{port} erhält man aus der Menge der Zustände des Protokoll-Statecharts (Abbildung 4.9 rechts). Gibt es in diesem Statechart Zustände, in denen nicht alle Signale erlaubt sind, muss man Q_{port} noch einen weiteren Zustand hinzufügen. Dieser Zustand wird eingenommen, wenn von einem Kommunikationspartner das Protokoll verletzt wurde. Der Automat für das Statechart aus Abbildung 4.9 enthält die Zustände

$$Q_{port} = \{q_0, q_1, q_{Err}\},$$

wobei q_{Err} den zusätzlichen Fehlerzustand bezeichnet.

Der Ausgangszustand q_0 ist durch den Startpunkt in Abbildung 4.9 markiert. In diesem Beitrag heißt der Ausgangszustand immer q_0.

Das Eingabealphabet Σ_{port} ist durch die Menge der Signale, die Trigger der Transitionen in Abbildung 4.9 sind, gegeben. Zusätzlich benötigt man noch ein Eingabezei-

chen für den Fall, wenn das Port kein Signal erhält. Weiterhin ist es auch sinnvoll, ein Eingabezeichen für verbotene Eingaben (z.B. wenn *DrillingControl* und *MC_FBA* gleichzeitig *Start* bzw. *Ended* senden wollen) zu definieren. Für das *drillPort* ist

$\Sigma_{port} = \{ e_0, e_1, e_2, e_3 \}$.

Tabelle 4.1: Bedeutung der Eingabezeichen von Σport

Eingabezeichen	Bedeutung für das Port
e_0	Das Port empfängt kein Signal.
e_1	Das Port empfängt das Start-Signal.
e_2	Das Port empfängt das Ended-Signal vom FBA.
e_3	Das Port empfängt gleichzeitig Ended und Start.

Die Bedeutungen der Eingabezeichen sind in Tabelle 4.1 beschrieben.

Die Übergangsfunktion δ_{port} ist durch die Transitionen des Statecharts gegeben. Existiert eine Kombination aus Zustand und Eingabezeichen im Statechart nicht, dann muss im Automat ein Übergang zum Fehlerzustand eingeführt werden.

Tabelle 4.2: Übergangsfunktion δ_{port}

Aktueller Zustand	Eingabe	Nächster Zustand
q_0	e_0	q_0
q_0	e_1	q_1
q_1	e_0	q_1
q_1	e_2	q_0
sonst		q_{Err}

In Tabelle 4.2 ist die Übergangsfunktion des Automaten für *drillPort* angegeben.

Die in diesem Abschnitt vorgenommene Beschreibung eines Ports durch einen Automaten ohne Ausgabe berücksichtigt nicht die Daten, die eventuell zu einem UML-Signal gehören. Es soll davon ausgegangen werden, dass eventuelle Daten immer fehlerfrei übertragen werden.

Im nächsten Abschnitt soll eine ähnlich vereinfachte Darstellung von Protokollen für Funktionsbausteine erreicht werden.

4.1.2.2 Beschreibung des FB-Protokolls als endlicher Automat M_{fbp}

Die Aufgabe eines endlichen Automaten für FB-Protokolle ist vergleichbar mit der von UML-Protokolls. Es müssen Ereignisse in den Ausgangsvariablen der Funktionsbausteine dahingehend kontrolliert werden, ob das Kommunikationsprotokoll zwischen den Bausteinen eingehalten wird.

Für das Zeitdiagramm aus Abbildung 4.6 lässt sich der (ebenfalls ausgabelose) Automat

$$M_{fbp} = (Q_{fbp}, \Sigma_{fbp}, \delta_{fbp}, q_0)$$

folgendermaßen ableiten:

Die Zustandsmenge Q_{fbp} ist die Menge der vier Zustände, die sich aus den möglichen Belegungen der Variablen *Execute* und *Ended* ergeben, ergänzt um einen Fehlerzustand:

$$Q_{fbp} = \{q_0, q_1, q_2, q_3, q_{Err}\}.$$

Tabelle 4.3 zeigt die Zuordnung der Zustände zu den Variablenbelegungen.

Tabelle 4.3: Bedeutung der Zustände aus Q_{fbp}

Zustand	Belegung von Execute und Ended
q_0	¬Execute ∧ ¬Ended
q_1	Execute ∧ ¬Ended
q_2	Execute ∧ Ended
q_3	¬Execute ∧ Ended
q_{Err}	beliebig

Die Eingabesymbole von M_{fbp} entsprechen Änderungsereignissen der Ausgangsvariablen von *MC_FBA* und *MC_FB*. Da sich die Belegungen verschiedener Variablen gleichzeitig verändern können, muss für jede mögliche Kombination ein Zeichen festgelegt werden. In Tabelle 4.4 sind die Eingabezeichen aus Σ_{fbp} den Änderungsereignissen zugeordnet. Ein "–" kennzeichnet keine Änderung des Wertes der jeweiligen Variable, während ein "x" für eine Wertänderung steht.

Tabelle 4.4: Bedeutung der Eingabezeichen aus Σ_{fbp}

Eingabezeichen	Eingabe vom FBA		Eingabe vom FB	
	Execute	ffwd_Position	Ended	Ausgangsdaten
e_0	-	-	-	-
e_1	-	-	-	x
e_2	-	-	x	-
e_3	-	-	x	x
e_4	-	x	-	-
e_5	-	x	-	x
e_6	-	x	x	-
e_7	-	x	x	x

Eingabezeichen	Eingabe vom FBA		Eingabe vom FB	
	Execute	**ffwd_Position**	**Ended**	**Ausgangsdaten**
e_8	x	-	-	-
e_9	x	-	-	x
e_{10}	x	x	-	-
e_{11}	x	x	-	x
e_{12}	sonst (laut FB-Protokoll verbotene Belegungen)			

Durch diese Art der Kodierung wird eine positive und eine negative Flanke z.B. in *Execute* durch gleiche Zeichen (z.B. e_8) repräsentiert. Gleiches gilt für *Ended*. Da aber die Belegung dieser beiden Variablen durch die Zustände (abgesehen von q_{Err}) bestimmt ist, kann trotzdem eine positive von einer negativen Flanke unterschieden werden. Die Inhalte der anderen Variablen werden vom FB-Protokoll nicht betrachtet. Wichtig sind nur die Zeitpunkte von Änderungen in den Inhalten.

Die Übergangsfunktion δ_{fbp} ist durch die zeitliche Reihenfolge der Zustände im Zeitdiagramm gegeben. Existiert eine Kombination aus Zustand und Eingabezeichen nicht, dann muss im Automat ein Übergang zum Fehlerzustand eingeführt werden. Abbildung 4.13 zeigt M_{fbp} als Zustandsdiagramm.

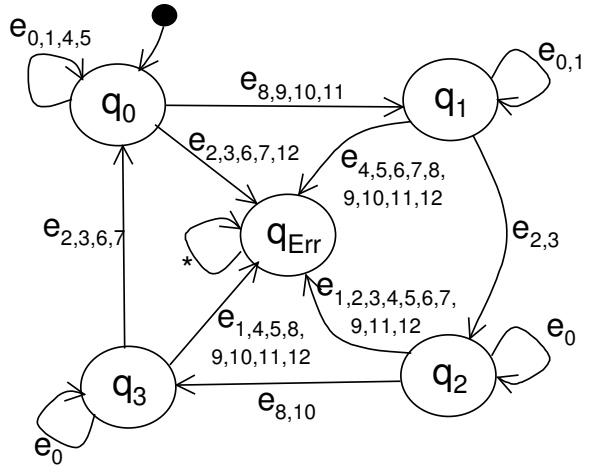


Abbildung 4.13 Transitionsdiagramm für M_{fbp} ($e_{x,y,...} \Leftrightarrow e_x \vee e_y \vee ...$)

4.1.2.3 Beschreibung des Funktionsbausteinadapters als endlichen Automat M_{fba}

Da der *MC_FBA* mit anderen Automaten kommunizieren soll, wird er als Moore-Automat

$$M_{fba} = (Q_{fba}, \Sigma_{fba}, \Delta_{fba}, \delta_{fba}, \lambda_{fba}, q_0)$$

modelliert. Ausgabe- und Eingabealphabet sind in Tabelle 4.5 und Tabelle 4.6 erläutert.

Tabelle 4.5: Bedeutung der Eingabesymbole aus Σ_{fba}

Eingabesymbol	Eingabe vom Funktionsbaustein		Eingabe vom Drilling-Control
	Ended	Ausgangsdaten	
e_0	-	-	-
e_1	-	-	Start
e_2	-	x	-
e_3	-	x	Start
e_4	x	-	-
e_5	x	x	-
e_6	sonst		

Tabelle 4.6: Bedeutung der Ausgabesymbole aus Δ_{fba}

Ausgabesymbol	Ausgabe an den Funktionsbaustein		Ausgabe an Drilling-Control
	Execute	ffwd_Position	
a_0	-	-	-
a_1	-	-	Ended
a_2	-	x	-
a_3	x	-	-
a_4	x	x	-

Die Zeichen "-" und "x" haben die gleiche Bedeutung wie in den vorherigen Abschnitten.

Q_{fba} beinhaltet neun Zustände, die in Tabelle 4.7 beschrieben werden.

Tabelle 4.7: Bedeutung der Zustände aus Q_{fba}

Zustand	Interpretation
q_0	Die FBA-Translation wird noch nicht ausgeführt. Der FBA wartet.
q_1	Der Wert in *ffwd_Position* wird geändert, bevor *Execute* auf *true* gesetzt wird. Zeile (4) in Abbildung 4.12.
q_2	*Execute* wird auf *true* gesetzt. Zeile (5) in Abbildung 4.12.
q_3	*Execute* und *ffwd_Position* werden gleichzeitig gesetzt.
q_4	Es wird auf eine positive Flanke in *Ended* gewartet. Zeile (6)
q_5	Die Ausgangsdaten des *MC_FB* werden eingelesen. Da das Einlesen nur FBA-intern geschieht, werden die Zeilen (7) bis (14) zu einem Zustand zusammengefasst.
q_6	Es wird eine negative Flanke in *Execute* ausgegeben.
q_7	Es wird auf eine negative Flanke in *Ended* gewartet.
q_8	Das Ended-Signal wird an *DrillingControl* geschickt.

Zustand	Interpretation
q_{Err}	M_{fba} erhielt eine unerwartete Eingabe.

Im Abbildung 4.14 ist aus Gründen der besseren Übersichtlichkeit der Zustand q_{Err} nicht enthalten. Man kann deshalb nicht die vollständige Übergangsrelation δ_{fba} und Ausgabefunktion λ_{fba} daraus ablesen. (Im Zustand q_{Err} hat M_{fba} die Ausgabe a_0. Die vollständige Übergangsrelation enthält zusätzlich zu den Transitionen aus Abbildung 4.14 noch Übergänge zu q_{Err}, die mit den Eingabesymbolen beschriftet sind, die in den Zuständen q_0 bis q_8 nicht erlaubt sind.)

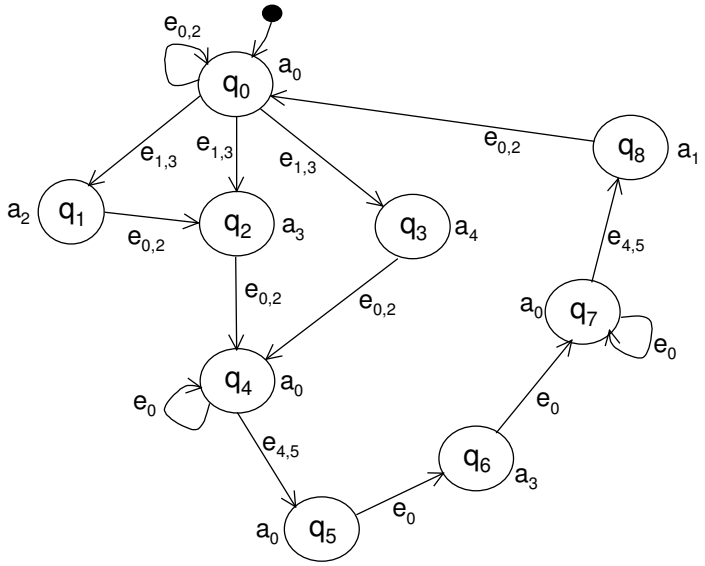


Abbildung 4.14 Transitionsdiagramm zum Funktionsbausteinadapter-Automat (ohne q_{Err})

Die nichtdeterministischen Übergänge in q_0 modellieren zwei unterschiedliche Gegebenheiten. Zum einen wird zwar immer die im *Start*-Signal enthaltene Position in *ffwd_Position* geschrieben (Zeile (4) in Abbildung 4.12), aber wenn die neue Position gleich der vorherigen Position ist, dann gibt es kein Änderungsereignis in *ffwd_Position*. In diesem Fall wird direkt von q_0 nach q_2 gewechselt und nur a_3 ausgegeben. Zum anderen gibt es verschiedene Implementierungsmöglichkeiten für die Zeilen (4) und (5) aus Abbildung 4.12. In einer SPS können diese beiden Befehle innerhalb eines SPS-Zyklus abgearbeitet werden, sodass die Wertänderungen in *Execute* und *ffwd_Position* (sofern die Position geändert wurde) gleichzeitig stattfindet. Der Automat gibt in diesem Fall a_4 aus (Zustand q_3). Werden die beiden Variablen aber in aufeinanderfolgenden SPS-Zyklen geschrieben, dann durchläuft M_{fba} den Pfad q_0 q_1 q_2. Durch die hier vorgestellte Modellierung können somit auch unterschiedliche Implementierungsmöglichkeiten in der späteren Verifikation berücksichtigt werden.

4.1.2.4 Modellierung der Systemumgebung

Als Systemumgebung für den FBA zählen *DrillingControl* und der Funktionsbaustein. Das einzige, vom realen Verhalten dieser beiden Softwarekomponenten Bekannte, ist aber, dass sie sich an die vereinbarten Protokolle halten müssen. Die Automaten

$$M_{drill} = (Q_{drill}, \Sigma_{drill}, \Delta_{drill}, \delta_{drill}, \lambda_{drill}, q_0)$$

und

$$M_{fb} = (Q_{fb}, \Sigma_{fb}, \Delta_{fb}, \delta_{fb}, \lambda_{fb}, q_0)$$

wurden deshalb als nichtdeterministische Moore-Automaten modelliert, die alle erlaubten Eingaben zum FBA ausnutzen. Sie werden im Anhang A vollständig spezifiziert. Für die Aussagen in Abschnitt 4.1.3.2 sind folgende Informationen ausreichend:

M_{drill} sendet im Zustand q_1 das *Start*-Signal.

M_{fb} verändert in den Zuständen q_0 und q_4 keine Ausgangsdaten.

4.1.2.5 Kommunikation zwischen den Automaten

Damit die Automaten miteinander kommunizieren können, wird in diesem Abschnitt für jeden Automat eine Funktion definiert, die das aktuelle Eingabesymbol abhängig von den Ausgaben anderer Automaten bestimmt. Diese Funktionen sollen σ-Funktionen genannt werden.

Die Eingaben von *DrillingControl* (M_{drill}) sind nur von den Ausgaben des Funktionsbausteinadapters abhängig:

$$\sigma_{drill}: \Delta_{fba} \rightarrow \Sigma_{drill}$$

Die Eingaben des Ports (M_{port}) sind von den Ausgaben von *DrillingControl* (M_{drill}) und *MC_FBA* (M_{fba}) abhängig:

$$\sigma_{port}: \Delta_{drill} \times \Delta_{fba} \rightarrow \Sigma_{port}$$

Die Eingaben des FBAs (M_{fba}) sind von den Ausgaben von *DrillingControl* (M_{drill}) und *MC_FB* (M_{fb}) abhängig:

$$\sigma_{fba}: \Delta_{drill} \times \Delta_{fb} \rightarrow \Sigma_{fba}$$

Die Eingaben des FB-Protokolls (M_{fbp}) sind von den Ausgaben von *MC_FBA* (M_{fba}) und *MC_FB* (M_{fb}) abhängig:

$$\sigma_{fbp}: \Delta_{fba} \times \Delta_{fb} \rightarrow \Sigma_{fbp}$$

Die Eingaben des *DrillingFB* (M_{fb}) sind nur von den Ausgaben von *MC_FBA* (M_{fba}) abhängig:

σ_{fb}: $\Delta_{fba} \rightarrow \Sigma_{fb}$

Durch die Definition der σ-Funktionen ist es möglich, aus den aktuellen Zuständen der Automaten M_{drill}, M_{fba} und M_{fb} die aktuellen Eingaben aller Automaten zu bestimmen. Ist z.B. M_{drill} im Zustand q_1 und M_{fba} im Zustand q_0, dann ist das aktuelle Eingabesymbol für M_{port}:

$\sigma_{port}(\lambda_{drill}(q_1), \lambda_{fba}(q_0)) = e_1$

wie man aus Tabelle 4.1, Abbildung 4.14 und Abschnitt 4.1.2.2 entnehmen kann. Weiterhin sind die Funktionen im Anhang A vollständig in Tabellenform angegeben.

Abbildung 4.15 veranschaulicht grafisch die Kommunikationsstruktur der Automaten.

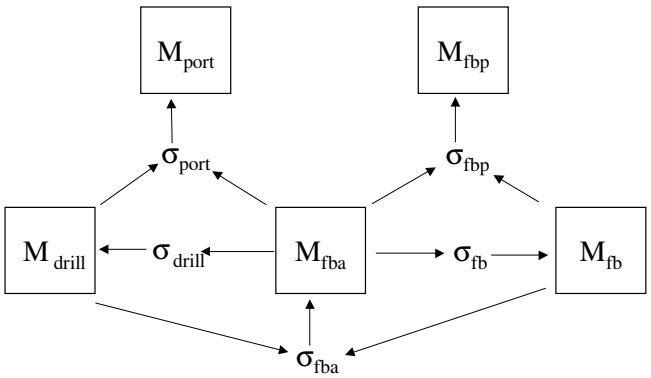


Abbildung 4.15 Kommunikationsbeziehungen zwischen den Automaten

4.1.3 Verifikation durch Modelchecking

Die Funktionsweise des *MC_FBA* soll in diesem Abschnitt mit Hilfe eines Modelcheckers verifiziert werden. Ein Modelchecker, der häufig zur Verifikation von Protokollen, digitalen Schaltungen und ähnlichen Problemen eingesetzt wird, ist der SMV-Modelchecker [SMV 2001].

Mit Hilfe der SMV Eingabesprache können Kripke-Strukturen und temporallogische Aussagen in CTL (Computation Tree Logic) spezifiziert werden [McM 1993]. Dazu wird zunächst in Abschnitt 4.1.3.1 gezeigt, wie die Automaten aus Abschnitt 4.1.2 in eine Kripke-Struktur überführt werden. In Abschnitt 4.1.3.2 werden für verschiedene Eigenschaften CTL-Formeln entwickelt, die verifiziert werden sollen. Die Abschnitte 4.1.3.3 und 4.1.3.4 erläutern beispielhaft den Umgang mit SMV.

4.1.3.1 Überführung der Automaten in eine Kripke-Struktur

Eine Kripke-Struktur ist ein Tupel K = (S, I, R, L), wobei S eine endliche Menge von Zuständen, $I \subseteq S$ eine Menge von Startzuständen, $R \subseteq S \times S$ eine totale Übergangsrelation und $L: S \rightarrow 2^{AP}$ eine Markierungsfunktion (AP als Menge atomarer Aussagen) ist [ClaGruPel 2000].

Da bei allen Automaten die Eingaben Funktionen von Ausgaben anderer Automaten, und diese Ausgaben wiederum Funktionen der aktuellen Zustände der Automaten sind, werden die Zustände in S eindeutig durch die Zustände der Automaten festgelegt. Man kann die Kripke-Struktur für die kommunizierenden Automaten aus Abschnitt 4.1.2 folgendermaßen beschreiben:

$$S \subseteq Q_{drill} \times Q_{port} \times Q_{fba} \times Q_{fbp} \times Q_{fb}$$

Sei $s \in S$, mit den Tupelkomponenten $s[q_{drill}] \in Q_{drill}$, $s[q_{port}] \in Q_{port}$, $s[q_{fba}] \in Q_{fba}$, $s[q_{fbp}] \in Q_{fbp}$ und $s[q_{fb}] \in Q_{fb}$. Weiterhin sei $s_0 \in S$ mit

$$s_0 = (q_0, q_0, q_0, q_0, q_0)$$

der einzige Startzustand der Kripke-Struktur.

Dann ergeben sich I und R zu:

$$I = \{s_0\}$$

$$\begin{aligned} R = \{\ & (s, s') \mid \\ & (s'[q_{drill}] = \delta_{drill}(s[q_{drill}], \sigma_{drill}(\lambda_{fba}(s[q_{fba}])))) \\ \wedge\ & (s'[q_{port}] = \delta_{port}(s[q_{port}], \sigma_{port}(\lambda_{drill}(s[q_{drill}]), \lambda_{fba}(s[q_{fba}])))) \\ \wedge\ & (s'[q_{fba}] = \delta_{fba}(s[q_{fba}], \sigma_{fba}(\lambda_{drill}(s[q_{drill}]), \lambda_{fb}(s[q_{fb}])))) \\ \wedge\ & (s'[q_{fbp}] = \delta_{fbp}(s[q_{fbp}], \sigma_{fbp}(\lambda_{fba}(s[q_{fba}]), \lambda_{fb}(s[q_{fb}])))) \\ \wedge\ & (s'[q_{fb}] = \delta_{fb}(s[q_{fb}], \sigma_{fb}(\lambda_{fba}(s[q_{fba}])))) \ \} \end{aligned}$$

Die Markierungsfunktion kann mit

$$\begin{aligned} AP = \{\ & (s[q_{drill}] = x_1) \mid x_1 \in Q_{drill}\} \\ \cup\ & \{\ (s[q_{port}] = x_2) \mid x_2 \in Q_{port}\} \\ \cup\ & \{\ (s[q_{fba}] = x_3) \mid x_3 \in Q_{fba}\} \\ \cup\ & \{\ (s[q_{fbp}] = x_4) \mid x_4 \in Q_{fbp}\} \\ \cup\ & \{\ (s[q_{fb}] = x_5) \mid x_5 \in Q_{fb}\} \end{aligned}$$

als Menge der wahren Aussagen von AP im Zustand s definiert werden:

$$L(s) = \{\ p \in AP \mid p = True\ \}$$

Auf diese Weise wird jedem Zustand s der Kripke-Struktur eine Menge von fünf Aussagen über die Zustände der Automaten zugeordnet.

Ein Pfad in der Kripke-Struktur K ist eine unendliche Folge von Zuständen

$$s_0 s_1 s_2 \ldots, \text{ sodass } (s_i, s_{i+1}) \in R \text{ für alle } i \geq 0.$$

Für jeden Zustand $s \in S$ gibt es einen unendlichen Berechnungsbaum mit der Wurzel s, und den Kanten $(s', s'') \in R$. Der Berechnungsbaum mit der Wurzel s_0 enthält alle

Pfade, die K durchlaufen kann. Damit sind alle möglichen Abarbeitungsreihenfolgen der kommunizierenden Automaten aus Abschnitt 4.1.2 beschrieben.

4.1.3.2 Spezifikation der zu verifizierenden Eigenschaften

Aussagen über einen Berechnungsbaum werden mit Hilfe temporaler Logik gemacht. Speziell kommt beim SMV-Modelchecker die Computation Tree Logic (CTL) zum Einsatz. Sie enthält den Existenzquantor E, den Allquantor A und die zeitlichen Operatoren X, F, G.

X bedeutet "im nächsten Schritt", F bedeutet "irgendwann" und G bedeutet "immer, für alle Zeiten".

Zusammen mit den Aussagen aus der Menge AP und beliebigen Booleschen Operatoren können mit Hilfe der CTL-Quantoren und CTL-Operatoren die zu verifizierenden Eigenschaften spezifiziert werden.

Soll z.B. überprüft werden, ob irgendwann einer der Automaten eine Eingabe erhält, die im aktuellen Zustand des Automaten nicht erwartet wird, dann kann das auf unterschiedliche Weise formuliert werden:

$$\neg \mathrm{EF}\ (s[q_{port}] = q_{Err} \vee s[q_{fba}] = q_{Err} \vee s[q_{fbp}] = q_{Err})$$

oder

$$\mathrm{AG}\ \neg(s[q_{port}] = q_{Err} \vee s[q_{fba}] = q_{Err} \vee s[q_{fbp}] = q_{Err})$$

Weitere Beispiele für zu verifizierende Eigenschaften sind:

Nachdem das Capsule das *Start*-Signal gesendet hat, ist der Nachfolgezustand des Funktionsbausteinadapter immer q_1, q_2 oder q_3:

$$\mathrm{AG}\ (s[q_{cap}] = q_1 \Rightarrow \mathrm{AX}\ (s[q_{fba}] = q_1 \vee s[q_{fba}] = q_2 \vee s[q_{fba}] = q_3))$$

Nachdem das Capsule das *Start*-Signal gesendet hat, kehren der Funktionsbausteinadapter und die Protokolle immer wieder in den Ausgangszustand zurück (Es gibt keine dynamischen Deadlocks):

$$\mathrm{AG}\ (s[q_{cap}] = q_1 \Rightarrow \mathrm{AF}\ (s[q_{port}] = q_0 \vee s[q_{fba}] = q_0 \vee s[q_{fbp}] = q_0))$$

Während der Funktionsbausteinadapter die Ausgangsdaten des FB liest, dürfen diese sich nicht verändern:

$$\mathrm{AG}\ (s[q_{fba}] = q_5 \Rightarrow (s[q_{fb}] = q_0 \vee s[q_{fb}] = q_4))$$

4.1.3.3 Die SMV Eingabesprache

Um die Kripke-Struktur und die CTL-Spezifikationen dem SMV Tool zur Verifikation zur Verfügung stellen zu können, müssen sie in der SMV-Sprache formuliert werden. Abbildung 4.16 zeigt den Automat M_{port} als Modul in der SMV-Sprache formuliert. Die vollständige Spezifikation ist im Anhang B aufgeführt.

```
module UMLPort()
{
  inp: {e0, e1, e2, e3};

  state: {q0, q1, qErr};

  init(state) := q0;

  next(state) := switch(state, inp) {
    (q0, e0): q0;
    (q0, e1): q1;
    (q1, e0): q1;
    (q1, e2): q0;
     default: qErr;
  };
}
```

Abbildung 4.16 Der Automat M_{Port} in der SMV-Sprache

Obwohl man in der SMV-Sprache eine Kripke-Struktur beschreiben muss, ist es durch eine geeignete Formulierung möglich, dass sich in den Modulen die Struktur der kommunizierenden Automaten aus Abschnitt 4.1.2 widerspiegelt.

Alle Module, die einen Automaten modellieren, haben den gleichen Aufbau. Eingaben werden symbolisch durch die Variable *inp* und der Zustand durch *state* repräsentiert.

In einem weiteren Modul werden für jedes Automaten-Modul Variablen mit den Namen *drill*, *port*, *fba*, *fbp* und *fb* angelegt und durch die Kommunikationsfunktionen zusammengeschaltet. Im *main*-Modul wird eine Variable namens *s* deklariert, die das Gesamtsystem beinhaltet.

Das *main*-Modul enthält die CTL-Formeln, in denen über die Variable *s* auf die Zustände der Automaten zugegriffen werden kann. Die erste Formel aus Abschnitt 4.1.3.2 wird in der SMV-Syntax wie folgt ausgedrückt:

```
!EF(s.fba.state=qErr | s.fbp.state=qErr | s.port.state=qErr)
```

4.1.3.4 Verifikation durch das Tool SMV

Die Aufgabe des SMV-Modelcheckers ist der Nachweis, dass die Kripke-Struktur die aufgestellten CTL-Formeln erfüllt oder nicht erfüllt. Als Verifikationsergebnis liefert der Modelchecker *True* oder *False* zu jeder CTL-Formel. Wird eine Formel nicht erfüllt (*False*), generiert der Modelchecker ein Gegenbeispiel (ein Pfad, für den die CTL-Formel nicht gilt). Mit Hilfe dieses Gegenbeispieles kann die Spezifikation aus

Abschnitt 4.1.1 und daraus resultierend das Automatenmodell aus Abschnitt 4.1.2 solange verbessert werden, bis alle CTL-Formeln erfüllt sind.

Die in den Abschnitten 4.1.1 und 4.1.2 vorgestellten Modelle sind bereits das Ergebnis einer Überarbeitung durch das Modelchecking, sodass alle CTL-Formeln als verifiziert gelten.

4.1.4 Zusammenfassung zum Anwendungsbeispiel „Motion Control“

In diesem Anwendungsbeispiel wurde gezeigt, wie ausgehend von Spezifikationen, die in der UML und der IEC 61131-3 verfasst wurden, ein verifizierbares Automatenmodell entwickelt werden kann. Dabei wurde ein spezielles Modellierungselement, der Funktionsbausteinadapter, betrachtet, der für die Spezifikation der Kommunikation zwischen einem Funktionsbaustein und einem Port einer UML-Klasse zuständig ist.

Ein Funktionsbausteinadapter spezifiziert die Protokollumsetzung zwischen Funktionsbaustein und Port auf einer logischen, plattformunabhängigen Ebene. Auf dieser Ebene wurden verschiedene typische Protokollfehler wie nicht-spezifizierter Empfang oder dynamischer Deadlock untersucht. Werden durch das Modelchecking dabei die CTL-Formeln verifiziert (Ergebnis: *true*), entspricht das einem mathematischen Beweis, dass die untersuchten Fehler nicht im Modell auftreten können.

Es ist dabei unerheblich, ob die UML-Klasse eine Softwarekomponente modelliert, die sich in der Steuerungsebene, in einer der höheren Ebenen der Unternehmenshierarchie oder im Internet befindet. Der Funktionsbaustein könnte sich in einer „klassischen“ SPS oder in einer Soft-SPS auf einem PC oder Mikrocontroller befinden.

Nachdem die plattformunabhängige Ebene verifiziert ist, kann dann im nächsten Schritt zur Verifikation der plattformabhängigen Spezifikationen übergegangen werden. Dazu müssen die in diesem Kapitel vorgestellten Automatenmodelle verfeinert und auf gleiche Weise wie in Abschnitt 4.1.3 vorgestellt verifiziert werden.

4.2 Anwendungsbeispiel Fertigungsanlage

In diesem Anwendungsbeispiel soll über Erfahrungen beim Einsatz von Funktionsbausteinadaptern in einer konkreten fertigungstechnischen Anlage berichtet werden. In dieser Anlage wird ein Transportsystem mit Hilfe einer SPS (S7-315-2DP) gesteuert. Dieses Transportsystem muss mit einer Fertigungszelle kommunizieren, die in den Sprachen UML/C++ entwickelt und implementiert wurde. Die Kommunikation findet über einen Feldbus (PROFIBUS-DP) statt, der mit Hilfe eines RS232-Gateways mit der Fertigungszelle verbunden ist. Es soll hierbei gezeigt werden, wie ausgehend von den Anforderungen der Entwurf mit Hilfe von Funktionsbausteinadaptern durchgeführt und anschließend mit Hilfe kommerziell verfügbarer Software-Werkzeuge (SIMATIC Step 7™ [SIEMENS Step 7] und Rational Rose RealTime™ [Rat 2001]) implementiert wird.

4.2.1 Anwendungsbeispiel: Fertigungsanlage

Das Anlagenschema der Fertigungsanlage ist in Abbildung 4.17 dargestellt.

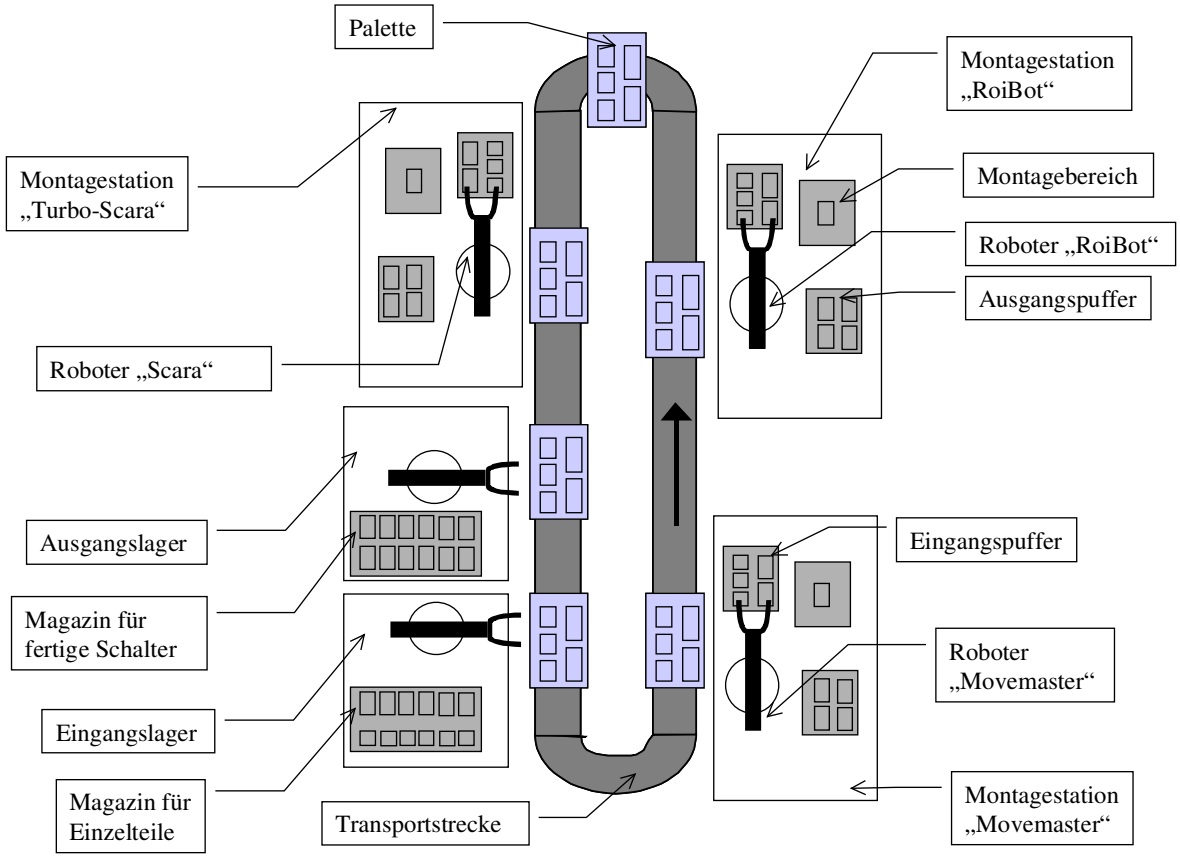


Abbildung 4.17 Schema der Fertigungsanlage

Drei Montagestationen werden über ein Transportsystem vom Eingangslager mit Einzelteilen beliefert. Die fertig montierten Produkte werden über das Transportsys-

tem zum Ausgangslager gebracht. Dabei arbeiten die einzelnen Zellen (Montagestationen, Eingangs-, Ausgangslager und Transportsystem) weitestgehend autonom. Benötigt zum Beispiel eine Montagestation Einzelteile zur Montage eines Produktes, fordert sie diese beim Transportsystem an. Das Transportsystem ist dafür verantwortlich, eine geeignete Palette zum Eingangslager zu bewegen und diesem die Positionen innerhalb der Palette mitzuteilen, auf die Einzelteile gestellt werden sollen. Nachdem das Eingangslager seine Arbeit beendet hat, wird die Palette zur entsprechenden Montagestation gefahren. Der Montagestation wird vom Transportsystem die Ankunft einer Palette mitgeteilt. Für den Informationsaustausch zwischen Transportsystem und den Roboterzellen (Lager oder Montagestation) werden drei unterschiedliche Typen von Nachrichten verwendet:

Die Transportanforderung (Transport Request, TRQ) wird von einer Montagestation an das Transportsystem gesendet. Die Nachricht TRQ beinhaltet Informationen über die Art und Anzahl der benötigten oder abzutransportierenden Einzelteile oder Produkte.

Die Antwort auf eine Transportanforderung (Transport Response, TRS) wird vom Transportsystem an eine Roboterzelle gesendet, wenn Einzelteile oder Produkte vor einer Roboterzelle angekommen sind. Die Nachricht TRS beinhaltet Informationen über die Art der Teile und deren Positionen auf der Palette.

Die Palettenfreigabe (Pallet Free, PF) wird von einer Roboterzelle an das Transportsystem gesendet, wenn alle Teile von der Palette entnommen bzw. auf die Palette gestellt wurden und sich der Roboterarm aus dem Kollisionsbereich der Palette entfernt hat. Als Information wird dieser Nachricht nur die Nummer der Roboterzelle mitgegeben, damit das Transportsystem die Nachricht einer Roboterzelle zuordnen kann.

Mit diesen drei Nachrichten wurde die Kommunikation zwischen Transportsystem und Roboterzellen zunächst nur informell beschrieben. Im weiteren Verlauf dieses Beitrages soll eine formale Beschreibung vorgestellt werden. Dazu stellen Abschnitt 4.2.1.1 die Softwareschnittstelle des Transportsystems in Form eines Funktionsbausteines dar. Abschnitt 4.2.1.2 erläutert dann die Softwareschnittstelle einer Roboterzelle, die in Form einer UML Capsule-Klasse vorliegt.

4.2.1.1 Software-Schnittstelle des Transportsystems

Die Steuerung des Transportsystems ist in Form eines Funktionsbausteines gegeben, der in Abbildung 4.18 zu sehen ist. Der Funktionsbaustein besitzt drei Gruppen von Eingangs- und Ausgangsvariablen, deren Namen jeweils mit dem Prefix *TRQ*, *Pallet_Free* oder *TRS* beginnen. Die Variablen *TRQ_In*, *TRQ_Out* und *TRQ_Data_In* werden vom Transportsystem zur Entgegennahme der Transportanforderung verwendet. Die Variablen *Pallet_Free_In*, *Pallet_Free_Out* und *Pallet_Free_Data_In* wer-

den für die Nachricht Palettenfreigabe benutzt. Mit den Variablen *TRS_In*, *TRS_Out* und *TRS_Data_Out* kann das Transportsystem eine Antwort auf eine Transportanforderung versenden. *TRQ_In* und *Pallet_Free_In* zeigen dem Transportsystem den Beginn einer Nachrichtenübermittlung von einer Roboterzelle an. Mit *TRS_Out* startet das Transportsystem eine Nachrichtenübermittlung an eine Roboterzelle.

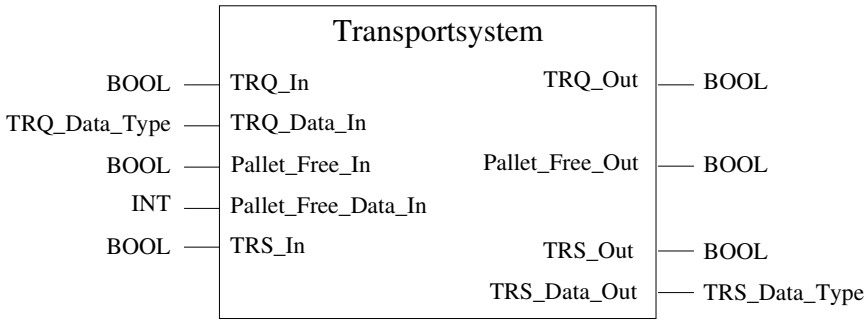


Abbildung 4.18 Funktionsbaustein Transportsystem

Um das Beispiel aus den Abschnitten 4.2.2 und 4.2.3 zu unterstützen, soll hier die Antwort auf eine Transportanforderung (TRS – Transportrequest-ReSponse) genauer erläutert werden. Dazu dient das Zeitdiagramm aus Abbildung 4.19. Die Nachricht TRS besitzt die höchste Priorität. Da der Kommunikationskanal zwischen Transportsystem und Roboterzelle nicht die gleichzeitige Übermittlung mehrerer Nachrichten erlaubt, muss die Übertragung von Nachrichten mit niedrigerer Priorität abgebrochen werden, falls diese zur gleichen Zeit gesendet wurden.

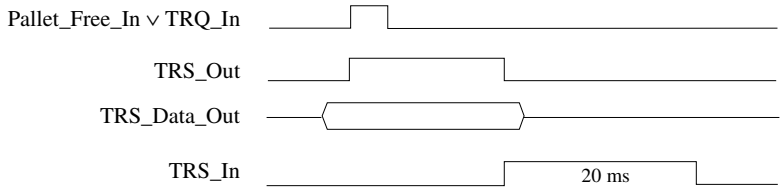


Abbildung 4.19 Zeitdiagramm zur Nachricht TRS

Die Daten, die mit TRS gesendet werden sollen, werden vom Transportsystem in *TRS_Data_Out* zur Verfügung gestellt. Mit einer positiven Flanke in *TRS_Out* wird dem Empfänger signalisiert, dass die Daten zur Verfügung stehen. Der Empfänger bestätigt dann in *TRS_In* die Übernahme der Daten durch eine positive Flanke. *TRS_In* soll frühestens nach 20ms wieder zurückgesetzt werden.

TRS_Data_Out ist vom Typ *TRS_data_type*. Das ist ein benutzerdefinierter Datentyp nach IEC 61131-3 (Abbildung 4.20). Er beinhaltet die ID der Transportanforderung, zu der die Antwort gehört (*TRQ_ID*), die ID der Roboterzelle, an die die TRS gesendet werden soll (*TRS_Robot*) und die Informationen über die Beladung der Palette (*TRS_pos*).

```
TYPE  TRS_data_type
 STRUCT
  TRQ_ID: INT := 0;
  TRS_Robot: INT := 0;
  TRS_pos: array[1..7] of Pos_info;
 END_STRUCT
END_TYPE

TYPE Pos_info
 STRUCT
  Pos_Status: BOOL := FALSE;
  Part_on: Part_info;
 END_STRUCT
END_TYPE

TYPE Part_info
 STRUCT
  Part_Type: INT := 0;
  Part_Sender: INT := 0;
  Part_Receiver: INT := 0;
 END_STRUCT
END_TYPE
```

Abbildung 4.20 Definition der Datentypen zu TRS

Jede Palette besitzt 7 Positionen, auf denen Teile platziert werden können. *TRS_pos* ist ein Feld, das für jede Position einer Palette eine Datenstruktur vom Typ *Pos_info* enthält. Das Transportsystem kann eine Roboterzelle auffordern, ein Teil auf eine Position einer Palette zu stellen, oder eines zu entnehmen. Soll ein Teil auf die Position gestellt werden, ist *Pos_Status* gleich *FALSE*. In *Part_on* befinden sich genauere Informationen über die Art des Teiles, das auf der Position zur Verfügung steht bzw. auf diese gestellt werden soll. In *Part_Type* wird die Typnummer des Teiles gespeichert. *Part_Sender* und *Part_Receiver* werden zurzeit nur intern des Transportsystems verarbeitet und deshalb nicht an eine Roboterzelle weitergeleitet (siehe Abschnitt 4.2.2).

Im nächsten Abschnitt wird der Kommunikationspartner dieses Funktionsbausteines und dessen Schnittstelle vorgestellt.

4.2.1.2 Software-Schnittstelle einer Roboterzelle

Die Steuerung einer Roboterzelle ist in Form eines UML-Capsules gegeben. Ein Capsule ist einer UML-Klasse ähnlich. Man verwendet es zur Modellierung der Struktur und des Verhaltens von Objekten. Im Unterschied zu normalen Klassen dürfen Capsules aber keine öffentlichen (public) Attribute oder Operationen besitzen. Eine Kommunikation zwischen Capsules ist nur über so genannte Ports möglich. Um Nachrichten über solche Ports versenden und empfangen zu können, müssen diese Nachrichten in Protokollen definiert werden. Protokolle sind ebenfalls eine spezielle Art von Klassen.

Abbildung 4.21 zeigt ein UML-Klassendiagramm, das das Protokoll *TransportProtocol* und zwei Capsules *Robot* und *Transportsystem* enthält. *Robot* besitzt ein Port *transportPort*, das dem *TransportProtocol* zugeordnet ist. Deshalb kann es die Nachrichten *TR_response*, *TR_ackn*, *PF_ackn* empfangen und die Nachrichten *Transport_request* bzw. *Pallet_free* senden. Das Transportsystem soll die Nachrichten *Transport_request* und *Pallet_free* empfangen und die restlichen Nachrichten senden können. Deshalb muss es einen Port (*~transportPort*) besitzen, das die umgekehrte (*conjugated*) Rolle des *TransportProtocol* implementiert. Damit können die beiden Capsules miteinander kommunizieren. Eine ausführlichere Beschreibung der Konzepte Capsule, Port und Protocol findet sich in [SelRum 1999a].

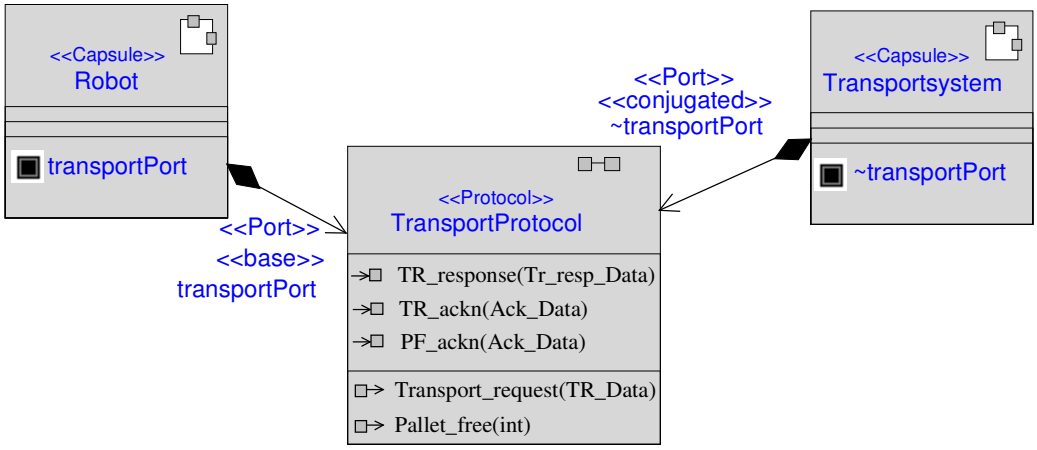


Abbildung 4.21 Klassendiagramm zum Transportprotokoll

Das Transportsystem liegt natürlich wie in Abschnitt 4.2.1.1 vorgestellt als Funktionsbaustein vor. Es wurde in Abbildung 4.21 als Capsule eingeführt, um das Konzept des *conjugated* Port zu erläutern. Abschnitt 4.2.2 ersetzt das Capsule *Transportsystem* mit einem Funktionsbausteinadapter. Vorher sollen aber noch die Daten, die mit den Nachrichten versendet werden, näher erläutert werden.

Jede Nachricht eines Protokolls kann optional Daten beinhalten. Die Antwort auf eine Transport-anforderung beinhaltet zum Bei-spiel Informationen über Teile, die auf eine Palette gestellt oder von einer Palette entnommen werden sollen und deren Positionen (siehe vorheriger Abschnitt). Im *TransportProtocol* heißt diese Nachricht *TR_response*. Mit dieser Nachricht kann man eine Instanz der Klasse *Tr_resp_Data* verschicken. Diese Klasse ist in Abbildung 4.22 in Form eines UML-Klassendiagramms abgebildet.

Ebenso wie der strukturierte Datentyp *TRS_data_type* aus Abschnitt 4.2.1.1 beinhaltet *Tr_resp_Data* die ID des Roboters (*id*) und die ID der Transportanforderung (*TR_id*). Für jede Position auf der Palette besitzt die Klasse ein Attribut vom Typ *Position*. *Position* ist eine Klasse, die in Abbildung 4.22 rechts dargestellt ist. Sie enthält zwei Attribute *loaded* und *type*. *loaded* ist *false*, wenn die Position leer ist und *true*, wenn ein Teil auf der Position der Palette steht. Ist *type* gleich Null, dann soll die Ro-

boterzelle diese Position nicht bedienen. Ist in *type* die ID eines Teiletyps gespeichert, soll ein Teil dieses Typs auf diese Position gestellt oder von dieser Position entnommen werden (entsprechend *loaded*). Da die beiden Attribute der Klasse *Position* nicht öffentlich sind, kann man auf sie nur über Operationen wie *setType*, *getType*, *setLoaded* und *getLoad* zugreifen. Obwohl alle Attribute der Klasse *Tr_resp_Data* öffentlich sind, wurden in dieser Klasse ebenfalls Zugriffsoperationen zur Verfügung gestellt. Mit der Operation *setIDs* kann man zum Beispiel die beiden IDs für die Roboterzelle und die Transportanforderung setzen (siehe Abbildung 4.25).

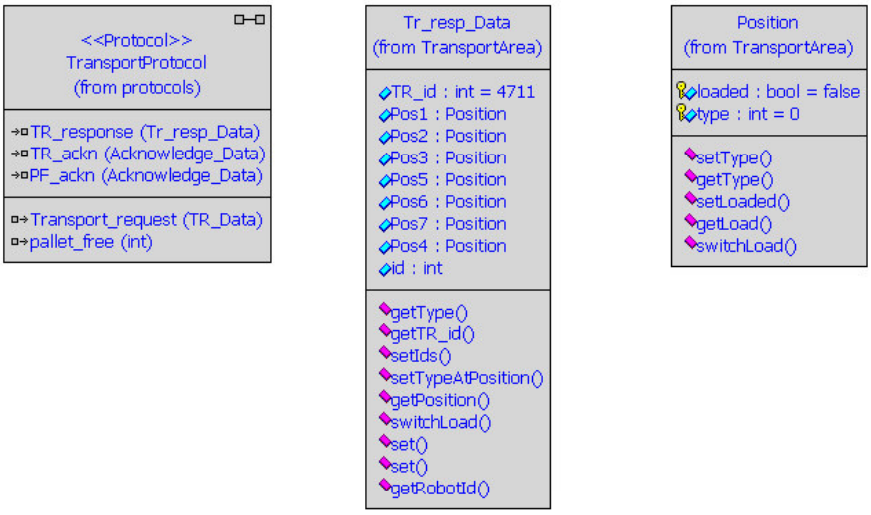


Abbildung 4.22 Klassendiagramm zu Datenklassen

Eine Roboterzelle erwartet eine Antwort auf eine Transportanforderung als UML-Nachricht der Form, wie wir sie in diesem Abschnitt beschrieben haben. Da die Steuerung des Transportsystems aber als Funktionsbaustein vorliegt, muss die FB-Nachricht *TRS* geeignet in die UML-Nachricht *Tr_response* umgesetzt werden. Das Gleiche gilt auch für die anderen Nachrichten, die im Transportprotokoll enthalten sind. Die formale Spezifikation dieser Umsetzung wird im folgenden Abschnitt vorgestellt. Abschnitt 4.2.3 beschreibt die Implementierung dieser Kommunikationsbeziehungen.

4.2.2 Modellierung der Kommunikationsbeziehungen

Um ein Capsule mit einem Funktionsbaustein verbinden zu können, wurde ein Funktionsbausteinadapter eingeführt (Abbildung 4.23). Die Attribute des Adapters sind die Eingangs- und Ausgangsvariablen des Funktionsbausteines, wobei die Schreibweise der bei IEC 61131-3 üblichen entspricht. Ausgangsvariable des Funktionsbausteins sind Eingangsvariable des Funktionsbausteinadapters und umgekehrt.

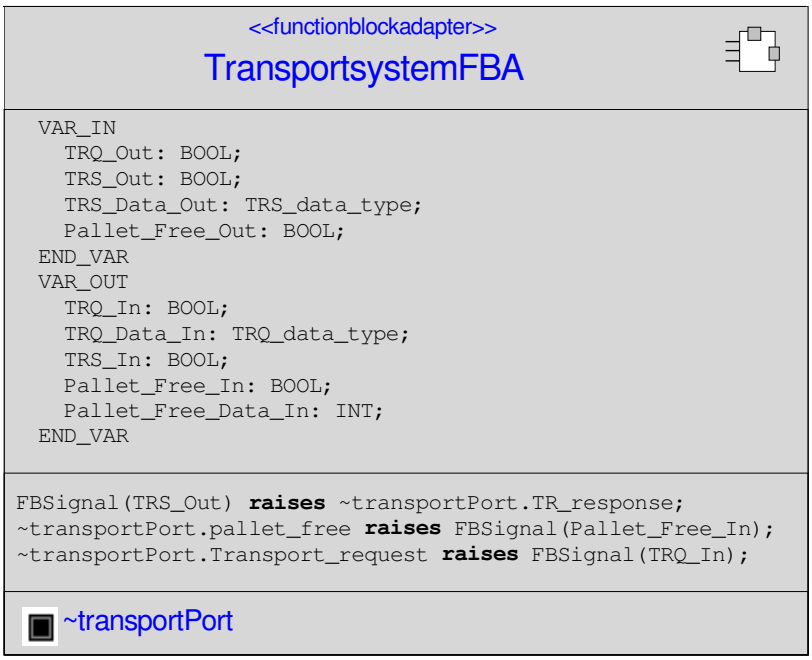


Abbildung 4.23 TransportsystemFBA

Das Port *~transportPort* des Funktionsbausteinadapter implementiert die konjugierte Rolle des Protokolls *TransportProtocol*. Damit kann das Port mit dem des Roboter-Capsules verbunden werden. Genauso können die Eingangs- und Ausgangsvariablen des Funktionsbausteinadapter mit denen des Funktionsbausteins verbunden werden (Abbildung 4.24).

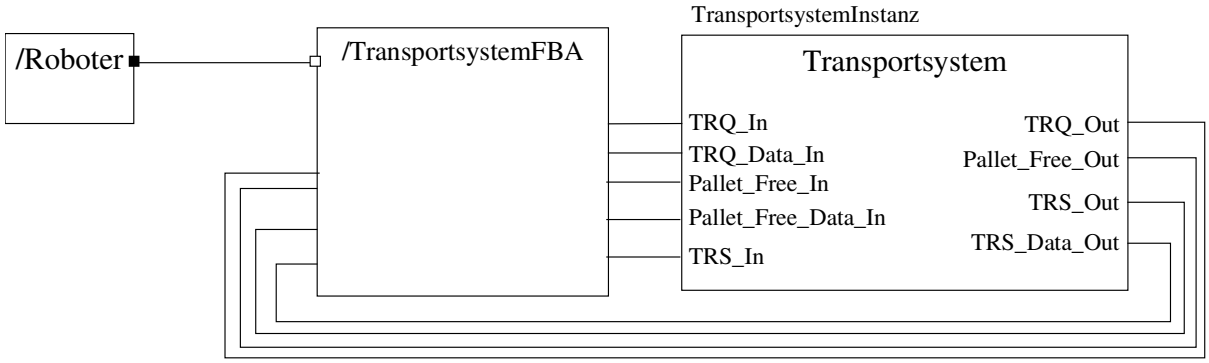


Abbildung 4.24 Strukturdiagramm mit Funktionsbaustein und Funktionsbausteinadapter

Abbildung 4.24 zeigt eine Kombination aus UML-RT Strukturdiagramm und dem Funktionsblockdiagramm nach IEC 61131-3. Damit ist die statische Struktur des Funktionsbausteinadapter vollständig festgelegt. Das dynamische Verhalten des Funktionsbausteinadapters wird durch dessen Operationen bestimmt. In Abbildung 4.23 erkennt man im zweiten Listenbereich, in dem bei normalen UML-Klassen die Operationen stehen, drei Zeilen mit dem Schlüsselwort *raises*. Links von *raises* steht ein UML- bzw. FB- Signal, das in das FB- bzw. UML- Signal rechts von *raises* übersetzt wird. Mit dem Schlüsselwort *FBSignal* links von *raises* werden Boolesche Ausdrücke definiert, die der Funktionsbausteinadapter zur Erkennung des Startes einer Nachrichtenübertragung vom Funktionsbaustein nutzt. Rechts von *raises* steht

FBSignal, um festzulegen, welche Boolesche Bedingung den Start einer Nachrichtenübertragung an den Funktionsbaustein signalisiert. Die UML-Signale sind in der Schreibweise *Portname.Signalname* notiert.

Jeder *raises*-Anweisung wird eine FBA-Operation zugeordnet. In einer FBA-Operation wird die zeitliche Belegung der Eingangs- und Ausgangsvariablen des Funktionsbausteinadapter beschrieben.

```
On_FBSignal(TRS_Out)
Signals
  s1: ~transportPort.TR_response;
Begin
  s1.setIds( TRS_Data_Out.TRS_Robot, TRS_Data_Out.TRQ_ID);
  s1.Pos1.setType( TRS_Data_Out.Position[1].Part_on.Part_type);
  s1.Pos1.setLoaded( TRS_Data_Out.Position[1].Pos_Status);
  s1.Pos2.setType( TRS_Data_Out.Position[2].Part_on.Part_type);
  s1.Pos2.setLoaded( TRS_Data_Out.Position[2].Pos_Status);
  s1.Pos3.setType( TRS_Data_Out.Position[3].Part_on.Part_type);
  s1.Pos3.setLoaded( TRS_Data_Out.Position[3].Pos_Status);
  s1.Pos4.setType( TRS_Data_Out.Position[4].Part_on.Part_type);
  s1.Pos4.setLoaded( TRS_Data_Out.Position[4].Pos_Status);
  s1.Pos5.setType( TRS_Data_Out.Position[5].Part_on.Part_type);
  s1.Pos5.setLoaded( TRS_Data_Out.Position[5].Pos_Status);
  s1.Pos6.setType( TRS_Data_Out.Position[6].Part_on.Part_type);
  s1.Pos6.setLoaded( TRS_Data_Out.Position[6].Pos_Status);
  s1.Pos7.setType( TRS_Data_Out.Position[7].Part_on.Part_type);
  s1.Pos7.setLoaded( TRS_Data_Out.Position[7].Pos_Status);
  sendAsync(s1);
  TRS_In := true;
  delay(T#20ms);
  TRS_In := false;
End_On_FBSignal
```

Abbildung 4.25 Operation *On_FBSignal(TRS_Out)*

Abbildung 4.25 zeigt beispielhaft, wie die Umsetzung der Antwort auf eine Transportanforderung vom FB-Signal in ein UML-Signal durch eine FBA-Operation beschrieben wird. Als erstes definiert die Operation eine Instanz *s1* des Signals *TR_response*. Dann werden die Daten von *s1* entsprechend der Daten von *TRS_Data_Out* gesetzt. Dazu kann *s1* wie eine Instanz der Klasse *TR_resp_Data* betrachtet werden. Der Zugriff auf Elemente der strukturierten Variable *TRS_Data_Out* erfolgt wie in der nach IEC 61131-3 üblichen Schreibweise. Es dürfen dabei nur Variablen elementaren Datentyps einander zugewiesen werden, weil für diese Datentypen eine Abbildung zwischen UML und IEC 61131-3 innerhalb des FBA-Frameworks existiert. Nachdem alle Daten umgesetzt wurden, kann das Signal mit *sendAsync* gesendet werden. Zum Schluss wird dem Funktionsbaustein mit einer positiven Flanke in *TRS_In* die erfolgreiche Nachrichtenübermittlung bestätigt.

Der Funktionsbausteinadapter enthält noch zwei weitere Operationen für die UML-Signale *Transport_request* und *pallet_free*. Diese Operationen besitzen einen ähnlichen Aufbau mit dem Unterschied, dass *TRQ_Data_In* und *Pallet_Free_Data_In* Werte aus den UML-Signalen zugewiesen werden müssen.

Damit ist das Verhalten und die Struktur des Funktionsbausteinadapter auf logischer, hardwareunabhängiger Ebene vollständig beschrieben. Diese Beschreibung kann nun als formale Spezifikation für die weitere Implementierung der Schnittstelle zwischen Capsule und Funktionsbaustein verwendet werden. Im nächsten Kapitel soll ein Eindruck vermittelt werden, wie die Implementierung durch den Einsatz kommerziell verfügbarer Software-Werkzeuge durchgeführt werden kann.

4.2.3 Implementierung der Kommunikationsbeziehungen

Der interne Aufbau des TransportsystemFBA muss der Verteilung des Funktionsbausteinadapter über zwei Computersysteme Rechnung tragen. Dieser Sachverhalt ist in Abbildung 4.26 dargestellt.

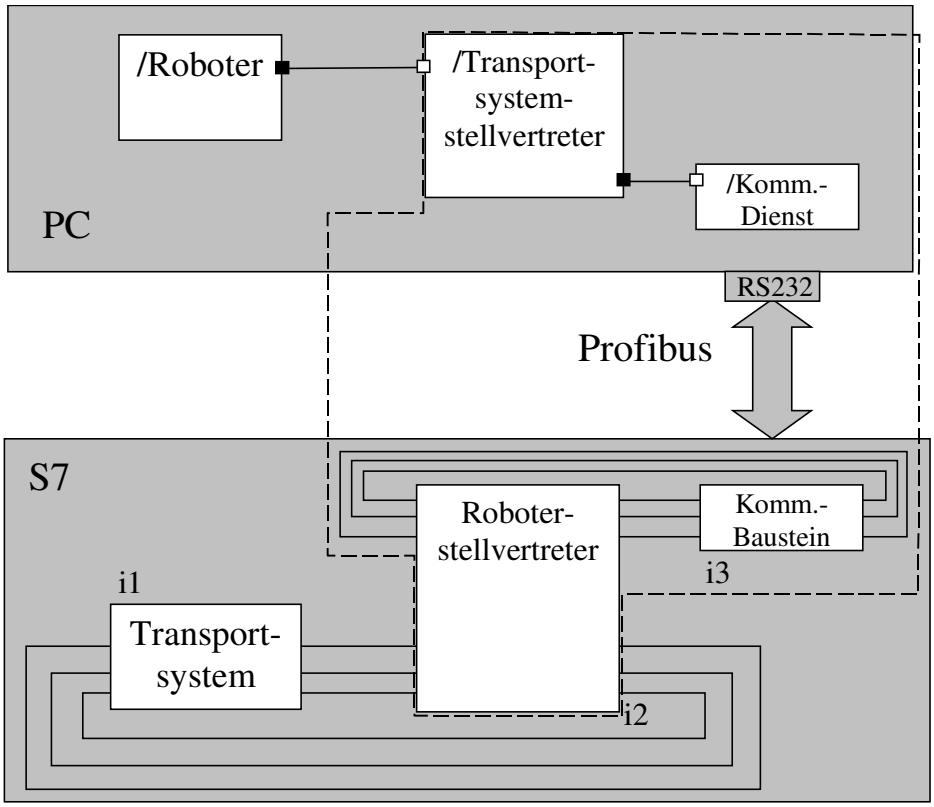


Abbildung 4.26 Struktur für die Implementierung des Funktionsbausteinadapter

Wie in der Einleitung bereits erwähnt wurde, ist die Steuerung des Transportsystems auf einer SPS als Funktionsbaustein implementiert. Die Steuerung der Roboterzelle hingegen befindet sich auf einem PC und liegt als Capsule-Klasse vor. Die Instanz des Funktionsbausteines Transportsystem ist deshalb in Abbildung 4.26 in der SPS (S7) dargestellt (Instanzname *i1*) während die Instanz der Capsule-Klasse *Roboter* auf dem PC ausgeführt wird. PC und SPS kommunizieren über ein Feldbussystem (PROFIBUS-DP) miteinander. Die Bestandteile des in Abschnitt 4.2.2 eingeführten Funktionsbausteinadapters sind in Abbildung 4.26 gestrichelt umrandet dargestellt. Es gibt zum einen ein Capsule, das innerhalb des Funktionsbausteinadapter auf der Seite des PC als Stellvertreter für das Transportsystem fungiert, und zum anderen ei-

nen Funktionsbaustein, der auf der Seite der SPS als Stellvertreter für die Robotersteuerung arbeitet. Diese beiden „Stellvertreterbausteine“ verwenden plattformabhängige Kommunikationsdienste/-bausteine, um untereinander Nachrichten auszutauschen.

Der PC ist über eine serielle Schnittstelle mit einem RS232-Gateway des Profibus-DP verbunden. Zum Ansprechen dieser Schnittstelle werden die Windows-API Funktionen zur Dateibearbeitung benötigt. Um einen Polling-Betrieb erlauben zu können, muss die serielle Schnittstelle im asynchronen Modus betrieben werden. Das bedeutet, dass das Lesen und Schreiben über die Schnittstelle in einem Hintergrundprozess verläuft, während die Anwendung (in Abbildung 4.26 ist das der *Transportsystemstellvertreter*) über die Windows-API Funktion *GetOverlappedResult* den Status der Lese- bzw. Schreiboperationen abfragen kann.

Die SPS (S7 315-2DP) ist sehr eng mit dem Profibus-DP gekoppelt. Die Eingangs- und Ausgangsdatenbereiche der Profibus-Slaves (z.B. des RS232-Gateways) werden automatisch auf den Eingangs- und Ausgangsdatenbereich der SPS abgebildet. Über Statusbits wird der SPS mitgeteilt, ob das RS232-Gateway gerade Daten sendet oder empfängt.

Für den Transportsystem- bzw. Roboterstellvertreter möchten wir im Folgenden C-Prozess (C für Capsule) und F-Prozess (F für Funktionsbaustein) schreiben. Das soll verdeutlichen, dass innerhalb des Funktionsbausteinadapter zwei nebenläufige Prozesse synchronisiert werden müssen. Dazu ist weiterhin ein Funktionsbausteinadapter-internes Kommunikationsprotokoll notwendig, das zwischen F-Prozess und C-Prozess über den Profibus betrieben werden muss.

Der C-Prozess hat die Aufgaben

- Kommunikation mit dem Roboter-Capsule über den *~transportPort* entsprechend des *TransportProtocol*,
- Kommunikation mit dem F-Prozess über den Profibus entsprechend des Funktionsbausteinadapter-internen Protokolls und
- Konvertierung der Daten aus dem *TransportProtocol* in das Funktionsbausteinadapter-interne Protokoll und umgekehrt.

Der F-Prozess hat die Aufgaben

- Kommunikation mit dem Funktionsbaustein Transportsystem über die Schnittstellenvariablen entsprechend des vereinbarten FB-Protokolls,
- Kommunikation mit dem C-Prozess über den Profibus entsprechend dem Funktionsbausteinadapter-internen Protokolls und

- Konvertierung der Daten aus dem FB-Protokoll in das Funktionsbausteinadapter-interne Protokoll und umgekehrt.

In der Funktionsbausteinadapter-internen Kommunikation muss das Eintreffen einer externen Nachricht dem jeweils anderen Prozess mitgeteilt und zugehörige Daten übertragen werden. Weiterhin müssen Konfliktfälle behandelt werden, wenn zum Beispiel die Roboterzelle und das Transportsystem gleichzeitig eine Nachricht übermitteln möchten. Für solche Fälle müssen die Nachrichten mit Prioritäten versehen werden (siehe Abschnitt 4.2.1.1). In der Konfliktbehandlung wird dann die Übermittlung der Nachricht mit niedrigerer Priorität abgebrochen. Das wird anschließend dem Sender der niedrigprioren Nachricht mitgeteilt. Die Datenkonvertierung in das bzw. aus dem Funktionsbausteinadapter-internen Protokoll ist im Wesentlichen eine Serialisierung bzw. Deserialisierung der Datenstrukturen des Funktionsbausteines und des Capsules. Dabei muss man zum Beispiel darauf achten, dass eine Variable vom Typ *INT* nach IEC 61131-3 nur zwei Byte lang ist und damit einer C++-Variable vom Typ *short int* entspricht. Außerdem ist die Reihenfolge von höher- und niederwertigem Byte in beiden Programmiersprachen unterschiedlich.

Das Verhalten des C-Prozesses wird in Rational Rose RealTime mit Hilfe eines Statecharts beschrieben. Dieses Statechart ist in Abbildung 4.27 dargestellt.

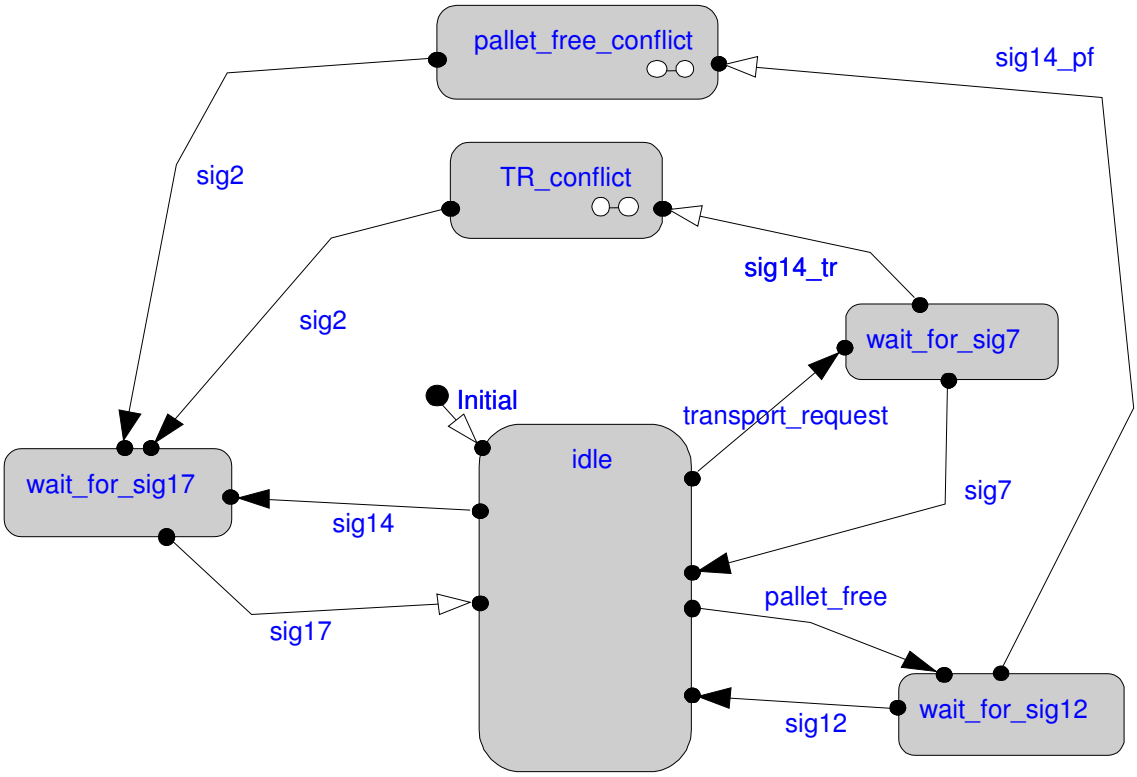


Abbildung 4.27 Statechart des C-Prozesses

Das Verhalten des F-Prozesses wird in SIMATIC Step 7 Graph in der Ablaufsprache dargestellt. Abbildung 4.28 zeigt aus Gründen der besseren Übersichtlichkeit nur einen Ausschnitt aus dem gesamten Diagramm für den F-Prozess. In diesem Beitrag

wird nur den Teil der beiden Diagramme betrachten, der für die Übermittlung der FB-Nachricht *TRS_Out* aus Abbildung 4.25 im konfliktfreien Fall notwendig ist.

Im Funktionsbausteinadapter-internen Protokoll werden die Nachrichten durch Nummern gekennzeichnet. Zum Beispiel entspricht der FB-Nachricht *TRS_Out* die Nachricht *sig14* für den C-Prozess (Abbildung 4.27) bzw. *sig14_Out* für den F-Prozess (Abbildung 4.28). Die Nachrichten mit den Nummer 15 und 17 werden zur Synchronisation zwischen beiden Prozessen des Funktionsbausteinadapters bei der Übermittlung von *TRS_Out* verwendet. Der F-Prozess verwendet weiterhin die Eingangsvariable *OK_In*, um eine Bestätigung für die erfolgreiche Übermittlung einer Nachricht durch das RS232-Gateway zu bekommen.

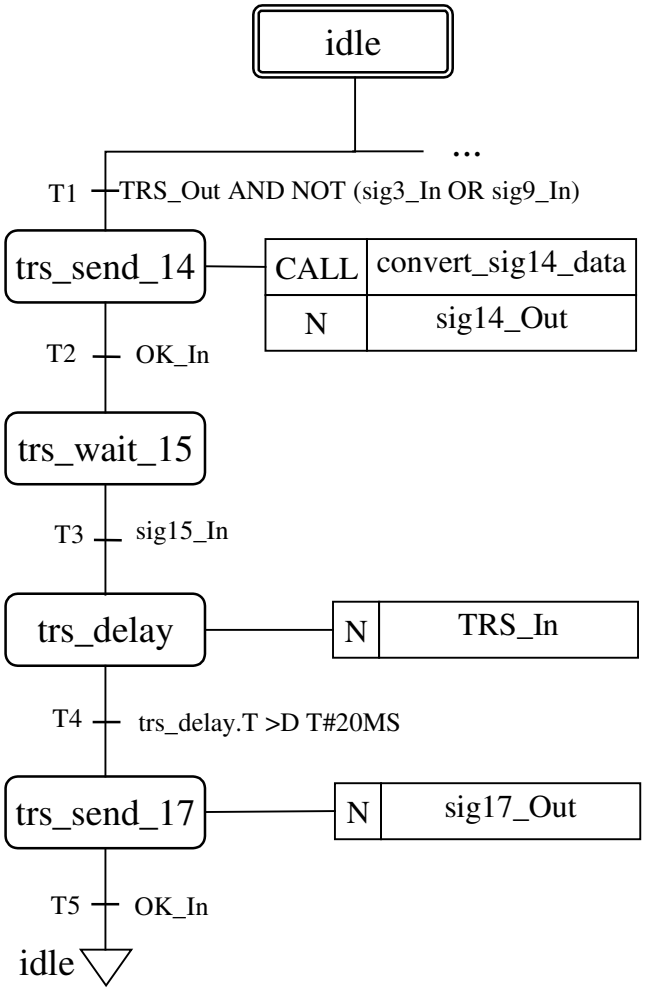


Abbildung 4.28 Ablaufsprache-Diagramm für F-Prozess

Werden keine Nachrichten übermittelt, befindet sich der C-Prozess im Zustand *idle* (Abbildung 4.27) und der F-Prozess im Schritt *idle* (Abbildung 4.28). Sendet also das Transportsystem *TRS_Out*, schaltet im konfliktfreien Fall die Transition *T1*. Im Schritt *trs_send_14* serialisiert der F-Prozess die aus *TRS_Data_Out* übernommenen Daten (mit der Funktion *convert_sig14_data*) und schreibt sie in den Ausgangsdatenbereich des RS232-Gateways. Dieses wird durch das Setzen von *sig14_Out* dazu auf-

gefordert, die Daten zu senden. Nach erfolgtem Senden schaltet *T2* und der F-Prozess wartet auf die Nachricht 15 vom C-Prozess. Der C-Prozess führt nach Erhalt der Nachricht 14 die mit der Transition *sig14* verbundene Aktion aus. In dieser Aktion werden die empfangenen Daten zunächst deserialisiert und einer Instanz der Klasse *Trans_resp_Data* zugewiesen. Damit ist die Spezifikation aus Abbildung 4.25 bis Zeile 19 erfüllt. In der Aktion wird weiterhin die Nachricht *TR_response* an das Roboter-Capsule gesendet und zum Schluss die Nachricht 15 an den F-Prozess. Damit wird dem F-Prozess signalisiert, dass Zeile 20 aus Abbildung 4.25 erfüllt wurde und nun *TRS_In* des Transportsystems für 20ms auf *TRUE* gesetzt werden muss (Zeilen 21 bis 23 aus Abbildung 4.25). Am Ende sendet der F-Prozess die Nachricht 17 an den C-Prozess. Danach gehen beide Prozesse wieder in *idle* über und sind zur Übermittlung einer neuen Nachricht bereit.

Die Umsetzung der Nachrichten Transport_request und Pallet_free geschieht auf ähnliche Art und Weise mit dem Unterschied, dass die Übertragungsrichtung vom Capsule zum Funktionsbaustein verläuft.

4.2.4 Zusammenfassung zum Anwendungsbeispiel „Fertigungsanlage“

In diesem Anwendungsbeispiel wurde gezeigt, wie man schon in der Entwurfsphase eines Systems Kommunikationsbeziehungen zwischen Funktionsbausteinen und Capsules mit Hilfe von Funktionsbausteinadaptern modellieren kann. Für die Implementierung eines Funktionsbausteinadapter muss man zunächst ein Funktionsbausteinadapter-internes Kommunikationsprotokoll entwickeln, das auch die notwendige Funktionsbausteinadapter-interne Synchronisation zwischen F- und C-Prozess berücksichtigt.

Es wurde das Verhalten des C-Prozesses durch ein Statechart und das Verhalten des F-Prozesses durch die Ablaufsprache beschrieben. Es zeigte sich, dass sich beide Darstellungsformen in ihrer jeweiligen Software-Umgebung (nach IEC 61131-3 oder nach UML) für diese Aufgabe eignen.

Als nachteilig erwies sich, dass in der Version des genutzten Ablaufsprache-Tools [SIEMENS Graph 5] keine benutzerdefinierten Datentypen für Schnittstellenvariablen verwendet werden konnten. Es musste deshalb auf eine Kombination aus globaler Variable und einer Funktion (siehe *convert_sig14_data* aus Abschnitt 4.2.3) zurückgegriffen werden. In dem eingesetztem UML-Tool [Rat 2001] mussten plattformabhängige Windows-API Funktionen benutzt werden, um über die serielle Schnittstelle kommunizieren zu können. Hier wäre ein vom Hersteller mitgeliefertes plattformunabhängiges Protokoll auf Port-Ebene wünschenswert, sodass bei der Code-Generierung für andere Plattformen (z.B. für Echtzeit-Betriebssysteme) an dieser Stelle keine Anpassung notwendig ist.

5 Zusammenfassung und Ausblick

Mit dieser Arbeit wurde das Konzept der Funktionsbausteinadapter vorgestellt, wodurch das Problem der Integration der IEC 61131-3 in die UML gelöst wurde. Funktionsbausteinadapter sind spezielle Protokolladapter, die in der Lage sind, FB-Protokolle auf UML-Protokolle abzubilden. Die Problematik, die zur Notwendigkeit einer solchen Abbildung führt, wurde anhand des ViewPoint-Frameworks erläutert. Weiterhin wurden die aktuellen Entwicklungen im Standardisierungsprozess der UML berücksichtigt, die voraussichtlich nicht vor Veröffentlichung der hier vorliegenden Arbeit in einer Version 2.0 der UML münden. Kleinere Ergänzungen im neuen Port-Konzept der UML [UMLPorts] sind zwar noch zu erwarten, aber dadurch dürfte das Konzept der Funktionsbausteinadapter nicht beeinflusst werden, da es sich bereits mit den Vorgängerversionen [SelRum 1999a] und [SelGulWar 1994] des aktuellen Port-Konzepts bewährt hat.

Das Verhalten von Funktionsbausteinadaptern kann durch die FBA-Sprache beschrieben werden, die speziell für diesen Zweck entwickelt wurde. Die Syntax der FBA-Sprache und auch der Funktionsbausteinadapter wurde in einer Kombination aus formaler Grammatik und einem UML-Profil definiert, das *FunctionBlockAdapters* genannt wurde. Die Semantik der FBA-Sprache wurde vollständig auf die Semantik von UML-Statecharts abgebildet. Es ist deshalb auch möglich, direkt Statecharts als Verhaltensbeschreibung von Funktionsbausteinadaptern zu verwenden. Dadurch gehen natürlich die Vorteile der FBA-Sprache verloren.

Als wichtigsten Bestandteil enthält das UML-Profil *FunctionBlockAdapters* den Stereotyp *FunctionBlockAdapter*. Im Unterschied zu herkömmlichen UML-Klassen können Funktionsbausteinadapter Eingangs- und Ausgangsvariablen enthalten, über die sie mit Funktionsbausteinen verbunden werden können. Die FBA-Sprache wird innerhalb von Funktionsbausteinadaptern in Übersetzungen aufgeteilt. Für diese Übersetzungen enthält das UML-Profil ebenfalls einen Stereotyp, der *FBATranslation* genannt wurde. FBA-Translations enthalten alle Informationen, die im Zusammenhang mit der FBA-Sprache von Bedeutung sind. Durch die geringere Komplexität der FBA-Translations im Vergleich zu Statecharts wird die Verständlichkeit von Funktionsbausteinadaptern für SPS-Entwickler verbessert.

Funktionsbausteinadapter eignen sich sehr gut für eine Verifikation durch Modelchecking. Ein erster Ansatz, der auf einfachen endlichen Automaten beruht und das SMV-Tool als Modelchecker verwendet, wurde in dieser Arbeit vorgestellt. Im Anwendungsbeispiel *Motion Control* wurde die Eignung dieses Ansatzes für reale praktische Probleme gezeigt.

Neben allgemeinen Hinweisen zur Implementierung von Funktionsbausteinadaptern wurde im Anwendungsbeispiel *Fertigungsanlage* ein komplexes Problem bearbeitet, bei dem besonders plattformabhängige Implementierungsaspekte betrachtet wurden.

Aus den Anwendungsbeispielen und den allgemeinen Erkenntnissen, die in dieser Arbeit vorgestellt wurden, könnte eine zweckmäßige Vorgehensweise für den Einsatz von Funktionsbausteinadaptern folgendermaßen zusammengefasst werden:

Nachdem für das gesamte Softwaresystem eine Anforderungsanalyse und erste Überlegungen über die Architektur ausgearbeitet wurden, kann zunächst die Entwicklung des UML-Teilsystems und des SPS-Teilsystems unabhängig voneinander durchgeführt werden. Wenn der Entwurf von UML-Klassen und von Funktionsbausteinen so weit fortgeschritten ist, dass die Übergänge zwischen beiden Teilsystemen entworfen werden müssen, ist der Einsatz von Funktionsbausteinadaptern angebracht.

Zunächst sollten die Schnittstellenvariablen der Funktionsbausteine und deren erlaubte Belegungen durch Zeitdiagramme und textuelle Beschreibungen festgelegt werden. Dadurch ist das FB-Protokoll definiert. Anschließend oder parallel dazu werden die dazu passenden UML-Schnittstellen entworfen. Dazu kann das Port-Konzept der UML herangezogen werden, wodurch das UML-Protokoll beschrieben wird. Um die unterschiedlich definierten Protokolle aufeinander abzubilden, werden ein oder mehrere Funktionsbausteinadapter entworfen, die die geforderten Protokolle unterstützen.

Nachdem die FBA-Translations ausgearbeitet wurden, können die Funktionsbausteinadapter verifiziert und validiert werden. Sind die dabei gewonnenen Erkenntnisse erfolgreich in einer Überarbeitung der Funktionsbausteinadapter eingegangen, kann zu implementierungsnäheren Phasen übergegangen werden. Vor der eigentlichen Programmierung ist es sinnvoll, ein funktionsbausteinadapterinternes Protokoll zu entwickeln, das bei der Kommunikation zwischen dem funktionsbausteinseitigen Teil und dem UML-seitigen Teil der Funktionsbausteinadapter verwendet wird. Dieses Protokoll kann ebenfalls validiert und verifiziert werden. In der darauffolgenden Programmierung kommen Sprachen der IEC 61131-3 und objektorientierte Sprachen zum Einsatz.

5.1 Entwurfsmuster für Funktionsbausteinadapter

Um den Einsatz von Funktionsbausteinadaptern zu vereinfachen, wäre es hilfreich, eine Menge von Entwurfsmustern zu entwickeln. Zum Beispiel lässt sich der im Anwendungsbeispiel *Motion Control* beschriebene Funktionsbausteinadapter auf eine Vielzahl von Regelkreisen anwenden, wenn man ihn etwas abstrakter darstellt. Dadurch können viele Erkenntnisse beim Entwurf und bei der Verifikation wiederverwendet werden.

In Abschnitt 3.1 wurden Anforderungen für Funktionsbausteinadapter in verallgemeinerter Form dargestellt. Diese und die Anforderungen und Lösungsvorschläge aus den Anwendungsbeispielen könnten als Ausgangspunkt für einen Entwurfsmusterkatalog für Funktionsbausteinadapter dienen.

In Anwendungen, bei denen keine Entwurfsmuster eingesetzt werden können, ist der Aufwand für Validierung und Verifikation relativ hoch. Deshalb wäre eine umfassende Werkzeugunterstützung zum Entwurf, zum Testen und zur Verifikation von Funktionsbausteinadaptern wünschenswert.

5.2 Verifikation von Funktionsbausteinadaptern

Im Anwendungsbeispiel *Motion Control* wurden einfache endliche Automaten als Verifikationsmodell eingesetzt. Da im Bereich der Automatisierungstechnik meistens Echtzeitanforderungen berücksichtigt werden müssen, wäre es bei der Verifikation vielversprechend, zeitbehaftete Automaten als Modell zur Verifikation zu verwenden [NeGrLuSi 1998].

In vielen Fällen können Zeitschrankenüberschreitungen auch als Nichtdeterminismus in einfachen endlichen Automaten modelliert werden. Dadurch können eventuelle Ausnahmebehandlungen von Funktionsbausteinadaptern ebenfalls verifiziert werden.

Da die Entwicklung mathematischer Automatenmodelle und Kripke-Strukturen aus Funktionsbausteinadaptern ein arbeitsintensiver Prozess ist, sind Werkzeuge zur Generierung solcher Modelle wünschenswert. Eine notwendige Vorarbeit zur Entwicklung solcher Werkzeuge ist eine formale Beschreibung für diesen Generierungsprozess, in dem ausgehend von einer Spezifikation in der Form von UML-Artefakten und der FBA-Sprache mathematische Modelle erzeugt werden müssen. Eine Möglichkeit wäre, über den Umweg eines Statecharts zu einem Modell endlicher Automaten zu gelangen. In Abschnitt 3.6.7.3 wird die Beziehung zwischen FBA-Translations und Statecharts besprochen. Die dort gewonnenen Erkenntnisse können in ein Werkzeug zur Generierung von Statecharts aus FBA-Spezifikationen einfließen. Für die Generierung endlicher Automaten aus Statecharts existieren bereits Werkzeuge [DaMöYi 2001].

5.3 Verallgemeinerung von Funktionsbausteinadaptern

Schließlich ist die Ausdehnung und Anwendung von Funktionsbausteinadaptern auf andere funktionsbausteinorientierte Sprachen, wie es bereits mehrfach in dieser Arbeit angesprochen wurde, eine weitere Möglichkeit der Fortsetzung von Arbeiten auf dem Gebiet der Funktionsbausteinadapter. Neben anderen funktionsbausteinorientierten Sprachen gibt es aber auch Sprachen wie SDL [SDL 1993], die ein zur UML ähnliches Portkonzept aufweisen. Obwohl SDL keine zur UML vergleichbaren Er-

weiterungsmechanismen enthält, kann die Idee der Funktionsbausteinadapter dort ebenfalls verwendet werden.

Die gleichzeitige Verwendung unterschiedlicher Sprachen zur Softwareentwicklung ist in der automatisierungstechnischen Praxis häufig eine Notwendigkeit. Funktionsbausteinadapter erleichtern hierbei die Integration von zwei der wichtigsten Standards zur Softwareentwicklung in der Automatisierungstechnik. Durch eine Verallgemeinerung des Konzeptes der Funktionsbausteinadapter können beliebige port-basierte Sprachen mit beliebigen funktionsblock-basierten Sprachen integriert werden. Das ist besonders interessant für das Gebiet der hybriden Systeme, in dem ereignis-diskrete Modelle mit kontinuierlichen Modellen vereinigt werden müssen.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[BauEng 2002] N. Bauer, S. Engell
A Comparison of Sequential Function Charts and Statecharts and an Approach towards Integration
Workshop INT'02
http://tfs.cs.tu-berlin.de/~mgr/int02/papers/bauer.ps.gz

[BicRadSch 2001] L. Bichler, A. Radermacher, A. Schürr
Combining Data Flow Equations with UML/Realtime
Proc. of ISORC 2001
S. 403-410
IEEE Computer Society 2001

[Bjo 2001] M. Bjoerkander
UML and Real-time Systems
OMER, GI-Edition - Lecture Notes in Informatics (LNI), P-5
Andy Schürr (Hrsg.)
Bonner Köllen Verlag 2001
S. 22-35

[BooRumJac 1999] G. Booch, J. Rumbaugh, I. Jacobson
The Unified Modeling Users Guide
Addison-Wesley 1999

[ClaGruPel 2000] E. M. Clarke, O. Grumberg, D. Peled
Model Checking
MIT Press 2000

[CORBA 1998] *The Common Object Request Broker -- Architecture and Specification*,
2.2 edition, 1998
ftp://ftp.omg.org/pub/docs/formal/98-07-01.pdf

[DaMöYi 2001] Alexandre David, M. Oliver Möller, Wang Yi
Formal Verification of UML Statecharts with Real-time Extensions
http://www.verify-it.de/papers/nwpt01.pdf

[EndHev 2002] B. E. Enders, T. Heverhagen,
Consistency During the Design of Function Block Adapters Using Integration by the Viewpoint Framework,
Proc. of the IDPT 2002 Conference - The Sixth World Conference on Integrated Design & Process Technology, Pasadena, California, June 23-28, 2002

[EnGoHeTr 2001] B. Enders, M. Goedicke, T. Heverhagen, R. Tracht
Arbeitsberichte zum DFG-Projekt "INTAS"
DFG-Geschäftszeichen GO 774/1-1
Universität Essen, FB 12, Automatisierungstechnik 2001

[FilBor 2001] A. Filippov, A. Borshchev
Daimler-Chrysler Modeling Contest: Car Seat Model
OMER, GI-Edition - Lecture Notes in Informatics (LNI), P-5
Andy Schürr (Hrsg.)
Bonner Köllen Verlag 2001
S. 46-50

[GaHeJoVl 1995] E. Gamma, R. Helm, R. Johnson, J. Vlissides
Design Patterns
Addison Wesley 1995

[GoeCraDob 1994] M. Goedicke, J. Cramer, E. Doberkat
Formal Methods
in: *Software Reusability*, W. Schäfer, R. Prieto-diaz, M. Matsumoto (Hrsg.)
S. 79 ff.
Ellis Horwood Workshop Series 1994

[GoFiKrNuFi 1992] M. Goedicke, A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein;
Viewpoints: A Framework for Integrating Multiple Perspectives in System Development
International Journal of Software Engineering and Knowledge Engineering 2(1)
S. 31-57.

[GolRob 1989] A. Goldberg, D. Robson
Smalltalk 80 – The Language
Addison-Wesley 1989

[HarPol 1998] D. Harel, M. Politi
Modeling Reactive Systems with Statecharts
McGraw-Hill
New York 1998

[Hed 2000] U. Hedtstück
Einführung in die Theoretische Informatik
Oldenbourg 2000
S. 43 ff.

[HeShGrTr 2002] T. Heverhagen, J. Shi, M. von Groll, R. Tracht
Kommunikation zw. FBs und UML-Capsules in einer industr. Softwareumgebung
VDI-Tagung Software Engineering in der industrieellen Praxis, VDI-Berichte 1666
S. 121-132
VDI-Verlag, Düsseldorf 2002

[Hev 2003] T. Heverhagen
Verifikation von Funktionsbausteinadaptern durch Modelchecking
Zeitschrift at-Automatisierungstechnik 4/2003
Oldenbourg 2003

[HevTra 2001a] T. Heverhagen, R. Tracht
Integrating UML-RealTime and IEC 61131-3 with Function Block Adapters
Proc. of ISORC 2001
S. 395-402
IEEE Computer Society 2001

[HevTra 2001b] T. Heverhagen, R. Tracht
Implementing Function Block Adapters
OMER, GI-Edition - Lecture Notes in Informatics (LNI), P-5
Andy Schürr (Hrsg.)
Bonner Köllen Verlag 2001
S. 122-134

[HevTra 2001c] T. Heverhagen, R. Tracht
Echtzeitanforderungen b. d. Integr. v. IEC 61131-3 FBs u. UML-RT Capsules
PEARL 2001, Echtzeitkommunikation und Ethernet/Internet
P.Holleczek, B.Vogel-Heuser
S. 87-96
Informatik aktuell, Springer-Verlag 2001

[HevTra 2001d] T. Heverhagen, R. Tracht
Negotiation Scenarios between autonomous Robot Cells: A Case Study
Proc. of Tunisian-German Conference Smart Systems and Devices
S. 499-504
Hammamet, March 2001

[HevTra 2001e] T. Heverhagen, R. Tracht
Using Stereotypes of the UML in Mechatronic Systems
Proc. of the 1. International Conference on Information Technology in Mechatronics, ITM'01, October 1-3, 2001, Istanbul, UNESCO Chair on Mechatronics, Bogazici University, Istanbul, Turkey,
S. 333-338

[HopUll 1994] J. E. Hopcroft, J. D. Ullman
Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie
Addison-Wesley 1994

[IEC 559] IEC 559
Binary floating-point arithmetic for microprocessor systems

[IEC 61131] International Electrotechnical Commission
IEC 61131 - Programmable Controllers Part 1 - 5
1992 Genf

[IEC 61499] International Electrotechnical Commission
Technical Committee No. 65:
Industrial-Process Measurement and Control
Working Group 6: *Function Blocks*

[IECDatenTypen 1993] International Electrotechnical Commission
Technical Committee No. 65:
Industrial-Process Measurement and Control
Sub-Committee 65B: Devices
Working Group 7: Programmable Controllers
Voting Draft – IEC 61131-3, 2nd Ed.
S. 31 ff.

[IECGrammatik 1993] International Electrotechnical Commission
Technical Committee No. 65:
Industrial-Process Measurement and Control
Sub-Committee 65B: Devices
Working Group 7: Programmable Controllers
Voting Draft – IEC 61131-3, 2nd Ed.
S. 152 ff.

[ISP 1993] Interoperable Systems Project: Fieldbus Specification, Version 3.0, 1993

[McM 1993] K. L. MCMillian
Symbolic Model Checking: An Approach to the state explosion problem
Kluwer Academic Publishers 1993

[MOF OMG] Meta Object Facility Specification
Version 1.3, 1999
OMG
ftp://ftp.omg.org/pub/docs/ad/99-06-05.pdf

[NeGrLuSi 1998] Neumann, Grötsch, Lubkoll, Simon
SPS-Standard: IEC 1131
Programmierung in verteilten Automatisierungssystemen
Oldenbourg 1998

[OCL Grammatik] OMG *Unified Modeling Language Specification (draft) Version 1.4 draft,*
Kapitel 6.9 Grammar
OMG February 2001
Seite "6-96" ff.
ftp://ftp.omg.org/pub/docs/ad/01-02-14.zip

[OtReEnMo 2000] M. Otter, M. Remelhe, S. Engell, P. Mostermann
Hybrid Models of Physical Systems and Discrete Controllers
Zeitschrift at-Automatisierungstechnik 9/2000
Oldenbourg 2003
S. 426-437

[PLCOpenDrill] PLCopen Technical Committee 2 TF
Function Blocks for Motion Control
S. 53-54
Drilling Example
http://www.plcopen.org/forms/motioncontrol.htm

[Rat 2001] *Rational Rose RealTime Users Guide*
Rational Software Corp. 2001

[Schn 1999] E. Schnieder
Methoden der Automatisierung
Vieweg 1999
S. 115 ff.

[SDL 1993] ITU-T: Recommendation Z.120 – CCITT Specification and Description Language (SDL)
Genf 1993

[SelGulWar 1994] B. Selic, G. Gullekson, P. T. Ward
Real-Time Object-Oriented Modeling
John Wiley & Sons 1994

[SelRum 1999a] B. Selic, J. Rumbaugh
Using UML for Complex Real-Time Systems
Rational Software 1999
www.rational.com/products/rosert/whitepapers.jsp

[SelRum 1999b] B. Selic, J. Rumbaugh
Mapping SDL Semantics to UML
Rational Software 1999
www.rational.com/products/rosert/whitepapers.jsp

[SIEMENS Graph 5] SIMATIC Step 7 Graph 5 Users Guide
SIEMENS AG

[SIEMENS Step 7] SIMATIC Step 7 Users Guide
SIEMENS AG

[Simulink 2002] Simulink
Dynamic System Simulation for MATLAB
The Mathworks, Inc. 2000

[SMV 2001] *Symbolic Model Verifier*
Homepage
http://www-2.cs.cmu.edu/~modelcheck/smv.html

[UML Sched, Perf, Time] *Response to the OMG RFP for Schedulability, Performance, and Time*,
Document Version 1.0, OMG document number ad/2000-08-04
ftp://ftp.omg.org/pub/docs/ad/00-08-04.pdf

[UML Superstructure] OMG Dokument ad/2002-09-02
Unified Modeling Language: Superstructure. Version 2 Beta R1 (draft)
OMG 2002
ftp://ftp.omg.org/pub/docs/ad/02-09-02.pdf

[UMLAddAttributeValueAction] In [UML Superstructure]
S. 191 ff.
Chapter 11. Common::Actions

[UMLClasses] In [UML Superstructure]
S. 297 ff.
Chapter 13. Common::Classes

[UMLCommonBehaviors] In [UML Superstructure]
S. 311 ff.
Chapter 15. Common::CommonBehaviors

[UMLDataType] In [UML Superstructure]
S. 62 ff.
The DataTypes diagram

[UMLInterfaces] In [UML Superstructure]
S. 121 ff.
Chapter 7. Basic::Interfaces

[UMLPorts] In [UML Superstructure]
S. 347 ff.
Chapter 18. Common::Ports

[UMLProfile] In [UML Superstructure]
S. 85 ff.
Chapter 3. Foundation::Profiles

[UMLProtocolStateMachines] In [UML Superstructure]
S. 495 ff.
Chapter 27. Complete::ProtocolStateMachines

[UMLReadAttributeAction] In [UML Superstructure]
S. 220 ff.
Chapter 11. Common::Actions

[UMLSendSignalAction] In [UML Superstructure]
S. 102 ff.
Chapter 11. Basic::Actions

Anhang A: Ergänzungen zum Abschnitt 4.1.2

In diesem Anhang soll die Beschreibung des Modells kommunizierender Automaten aus Abschnitt 4.1 vervollständigt werden. Der Automat M_{drill} wird mit folgenden Tabellen beschrieben werden:

Tabelle 12: Bedeutung der Eingabesymbole von Σ_{drill}

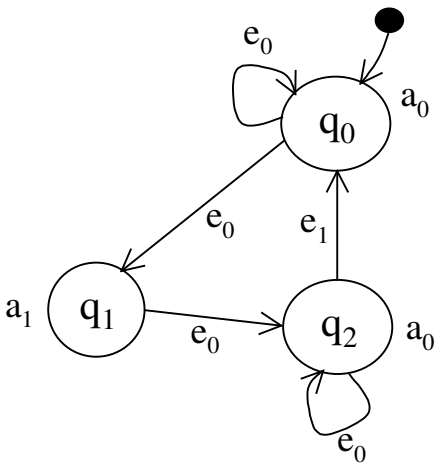
Eingabesymbol	Bedeutung für DrillingControl
e_0	**DrillingControl empfängt kein Signal.**
e_1	**DrillingControl empfängt das Ended-Signal vom Funktionsbausteinadapter.**

Tabelle 13: Bedeutung der Ausgabesymbole von Δ_{drill}

Ausgabesymbol	Bedeutung für das DrillingControl
a_0	**DrillingControl sendet kein Signal.**
a_1	**DrillingControl sendet das Start-Signal an den Funktionsbausteinadapter.**

Tabelle 14: Übergangsfunktion δ_{drill}

Aktueller Zustand	Eingabe	Nächster Zustand
q_0	e_0	$\{q_0, q_1\}$
q_1	e_0	q_2
q_2	e_0	q_2
q_2	e_1	q_0
sonst		q_{Err}



Transitionsdiagramm zu Tabelle 14

Tabelle 15: Ausgabefunktion λ_{drill}

Aktueller Zustand	Ausgabe
q_0	a_0
q_1	a_1
q_2	a_0
q_{Err}	a_0

Der Automat M_{fb} kann folgendermaßen durch Tabellen beschrieben werden:

Tabelle 16: Bedeutung der Eingabesymbole von Σ_{fb}

Eingabesymbol	Eingabe vom FBA	
	Änderung in Execute	Änderung in ffwd_Position
e_0	-	-
e_1	-	x
e_2	x	-
e_3	x	x

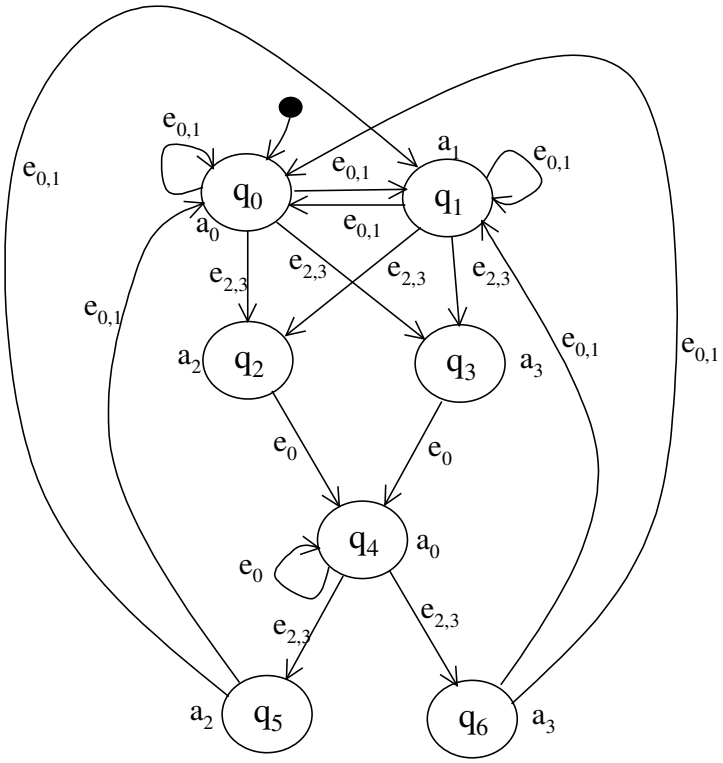
Tabelle 17: Bedeutung der Ausgabesymbole von Δ_{fb}

Ausgabesymbol	Änderung in Ended	Änderung in den Ausgangsvariablen außer Ended
a_0	-	-
a_1	-	x
a_2	x	-
a_3	x	x

Tabelle 18: Übergangsfunktion δ_{fb}

Aktueller Zustand	Eingabe	Nächster Zustand
q_0	e_0	q_0, q_1
q_0	e_1	q_0, q_1
q_0	e_2	q_2, q_3
q_0	e_3	q_2, q_3
q_1	e_0	q_0, q_1
q_1	e_1	q_0, q_1
q_1	e_2	q_2, q_3
q_1	e_3	q_2, q_3
q_2	e_0	q_4
q_3	e_0	q_4
q_4	e_0	q_4

Aktueller Zustand	Eingabe	Nächster Zustand
q_4	e_2	q_5, q_6
q_4	e_3	q_5, q_6
q_5	e_0	q_0, q_1
q_5	e_1	q_0, q_1
q_6	e_0	q_0, q_1
q_6	e_1	q_0, q_1



Transitionsdiagramm zu Tabelle 18

Tabelle 19: Ausgabefunktion λ_{fb}

Aktueller Zustand	Ausgabe
q_0	a_0
q_1	a_1
q_2	a_2
q_3	a_3
q_4	a_0
q5	a2
q_6	a_3

Damit die Automaten miteinander kommunizieren können, wird im Folgenden für jeden Automat eine Funktion definiert, die das aktuelle Eingabesymbol abhängig von den Ausgaben anderer Automaten bestimmt.

$\sigma_{cap}: \Delta_{fba} \rightarrow \Sigma_{cap}$

Δ_{fba}	Σ_{cap}
a_0	e_0
a_1	e_1
a_2	e_0
a_3	e_0
a_4	e_0

$\sigma_{port}: \Delta_{cap} \times \Delta_{fba} \rightarrow \Sigma_{port}$

Δ_{cap}	Δ_{fba}	Σ_{port}
a_0	a_0	e_0
a_0	a_1	e_2
a_0	a_2	e_0
a_0	a_3	e_0
a_1	a_0	e_1
a_1	a_1	e_3
a_1	a_2	e_0
a_1	a_3	e_0

$\sigma_{fba}: \Delta_{cap} \times \Delta_{fb} \rightarrow \Sigma_{fba}$

Δ_{port}	Δ_{fbp}	Σ_{fba}
a_0	a_0	e_0
a_0	a_1	e_2
a_0	a_2	e_4
a_0	a_3	e_5
a_1	a_0	e_1
a_1	a_1	e_3
a_1	a_2	e_6
a_1	a_3	e_6

$\sigma_{fbp}: \Delta_{fba} \times \Delta_{fb} \rightarrow \Sigma_{fbp}$

Δ_{fba}	Δ_{fb}	Σ_{fbp}
a_0	a_0	e_0
a_0	a_1	e_1
a_0	a_2	e_2
a_0	a_3	e_3
a_1	a_0	e_0
a_1	a_1	e_1
a_1	a_2	e_2
a_1	a_3	e_3
a_2	a_0	e_4
a_2	a_1	e_5
a_2	a_2	e_6
a_2	a_3	e_7
a_3	a_0	e_8
a_3	a_1	e_9
a_3	a_2	e_{12}
a_3	a_3	e_{12}
a_4	a_0	e_{10}
a_4	a_1	e_{11}
a_4	a_2	e_{12}
a_4	a_3	e_{12}

$\sigma_{fb}: \Delta_{fba} \rightarrow \Sigma_{fb}$

Δ_{fba}	Σ_{fb}
a_0	e_0
a_1	e_0
a_2	e_1
a_3	e_2
a_4	e_3

Anhang B: SMV-Modell zu Abschnitt 4.1.3.3

Dieser Anhang listet die Spezifikation der kommunizierenden Automaten aus dem Anwendungsbeispiel *Motion Control* auf. Sie dient dem SMV-Modelchecker als Eingabe und ist in der SMV-Sprache geschrieben. *DrillingControl* heißt in diesem Modell „Capsule“.

```
module main

VAR

  s: system;

SPEC

  !EF(s.fba.state=qErr | s.fbp.state=qErr | s.port.state=qErr) &
  AG (s.cap.outp=a1 -> AX (s.fba.state=q1 | s.fba.state=q2 |
      s.fba.state=q3)) &

  AG (s.cap.outp=a1 -> AF (s.fba.state=q0 & s.fbp.state=q0 &
      s.port.state=q0)) &

  AG ((s.fbp.state=q1 | s.fbp.state=q2) -> !s.fba.outp=a2) &
  AG (s.fba.state=q5 -> s.fbp.state=q2)


module Capsule()
{
  inp: {e0, e1};
  outp: {a0, a1};

  state: {q0, q1, q2, qErr};

  init(state) := q0;
  next(state) := switch(state, inp) {
    (q0, e0): {q0,q1};
    (q1, e0): q2;
    (q2, e0): q2;
    (q2, e1): q0;
     default: qErr;
  };

  outp := switch(state) {
    q1:      a1;
    default: a0;
  };
}

module UMLPort()
{
  inp: {e0, e1, e2, e3};

  state: {q0, q1, qErr};

  init(state) := q0;
```

```
  next(state) := switch(state, inp) {
    (q0, e0): q0;
    (q0, e1): q1;
    (q1, e0): q1;
    (q1, e2): q0;
     default: qErr;
  };
}

module FBAdapter()
{
  inp: {e0, e1, e2, e3, e4, e5, e6};
  outp: {a0, a1, a2, a3, a4};

  state: {q0, q1, q2, q3, q4, q5, q6, q7, q8, qErr};

  init(state) := q0;

  next(state) := switch(state, inp) {
    (q0, e0): q0;
    (q0, e2): q0;
    (q0, e1): {q1, q2, q3};
    (q0, e3): {q1, q2, q3};
    (q1, e0): q2;
    (q1, e2): q2;
    (q2, e0): q4;
    (q2, e2): q4;
    (q3, e0): q4;
    (q3, e2): q4;
    (q4, e0): q4;
    (q4, e4): q5;
    (q4, e5): q5;
    (q5, e0): q6;
    (q6, e0): q7;
    (q7, e0): q7;
    (q7, e4): q8;
    (q7, e5): q8;
    (q8, e0): q0;
    (q8, e2): q0;
     default: qErr;
  };

  outp := switch(state) {
    q0: a0;
    q1: a2;
    q2: a3;
    q3: a4;
    q4: a0;
    q5: a0;
    q6: a3;
    q7: a0;
    q8: a1;
  };
}

module FBProtocol()
{
  inp: {e0, e1, e2, e3, e4, e5, e6, e7, e8, e9, e10, e11, e12};

  state: {q0, q1, q2, q3, qErr};

  init(state) := q0;
```

```
  next(state) := switch(state, inp) {
    (q0, e0):  q0;
    (q0, e1):  q0;
    (q0, e4):  q0;
    (q0, e5):  q0;
    (q0, e8):  q1;
    (q0, e9):  q1;
    (q0, e10): q1;
    (q0, e11): q1;
    (q1, e0):  q1;
    (q1, e1):  q1;
    (q1, e2):  q2;
    (q1, e3):  q2;
    (q2, e0):  q2;
    (q2, e8):  q3;
    (q2, e10): q3;
    (q3, e0):  q3;
    (q3, e2):  q0;
    (q3, e3):  q0;
    (q3, e6):  q0;
    (q3, e7):  q0;
     default:  qErr;
  };
}

module FBlock()
{
  inp: {e0, e1, e2, e3};
  outp: {a0, a1, a2, a3};

  state: {q0, q1, q2, q3, q4, q5, q6, qErr};

  init(state) := q0;

  next(state) := switch(state, inp) {
    (q0, e0): {q0, q1};
    (q0, e1): {q0, q1};
    (q0, e2): {q2, q3};
    (q0, e3): {q2, q3};
    (q1, e0): {q0, q1};
    (q1, e1): {q0, q1};
    (q1, e2): {q2, q3};
    (q1, e3): {q2, q3};
    (q2, e0): q4;
    (q3, e0): q4;
    (q4, e0): q4;
    (q4, e2): {q5, q6};
    (q4, e3): {q5, q6};
    (q5, e0): {q0, q1};
    (q5, e1): {q0, q1};
    (q6, e0): {q0, q1};
    (q6, e1): {q0, q1};
     default: qErr;
  };

  outp := switch(state) {
    q0: a0;
    q1: a1;
    q2: a2;
    q3: a3;
    q4: a0;
    q5: a2;
    q6: a3;
```

```
  };
}



module system()
{
  cap: Capsule;
  port: UMLPort;
  fba: FBAdapter;
  fbp: FBProtocol;
  fb: FBlock;

  cap.inp := switch(fba.outp) {
    a0: e0;
    a1: e1;
    a2: e0;
    a3: e0;
    a4: e0;
  };

  port.inp := switch(cap.outp, fba.outp) {
    (a0, a0): e0;
    (a0, a1): e2;
    (a0, a2): e0;
    (a0, a3): e0;
    (a0, a4): e0;
    (a1, a0): e1;
    (a1, a1): e3;
    (a1, a2): e0;
    (a1, a3): e0;
    (a1, a4): e0;
  };

  fba.inp := switch(cap.outp, fb.outp) {
    (a0, a0):  e0;
    (a0, a1):  e2;
    (a0, a2):  e4;
    (a0, a3):  e5;
    (a1, a0):  e1;
    (a1, a1):  e3;
    (a1, a2):  e6;
    (a1, a3):  e6;
  };

  fbp.inp := switch(fba.outp, fb.outp) {
    (a0, a0): e0;
    (a0, a1): e1;
    (a0, a2): e2;
    (a0, a3): e3;
    (a1, a0): e0;
    (a1, a1): e1;
    (a1, a2): e2;
    (a1, a3): e3;
    (a2, a0): e4;
    (a2, a1): e5;
    (a2, a2): e6;
    (a2, a3): e7;
    (a3, a0): e8;
    (a3, a1): e9;
    (a3, a2): e12;
    (a3, a3): e12;
    (a4, a0): e10;
```

```
    (a4, a1): e11;
    (a4, a2): e12;
    (a4, a3): e12;
  };

  fb.inp := switch(fba.outp) {
    a0: e0;
    a1: e0;
    a2: e1;
    a3: e2;
    a4: e3;
  };
}
```

Anhang C: Elementare Datentypen der IEC 61131-3

Der Wertebereich der elementaren Datentypen der IEC 61131-3 unterscheidet sich leicht von denen anderer Programmiersprachen. Deshalb wird in der folgenden Tabelle zu jedem elementaren Datentyp dessen Wertebereich aufgeführt. Die Wertebereiche werden, soweit es möglich ist, als Mengen dargestellt. Z ist dabei die Menge der ganzen Zahlen und N die Menge der natürlichen Zahlen. Mit einem Stern * wird die Menge aller Wörter über den Zeichen einer Menge gekennzeichnet.

Tabelle 20: Wertebereiche der elementaren Datentypen

Typname	Beschreibung	Wertebereich
BOOL	Boolean	{ FALSE, TRUE } oder {0, 1}
SINT	Short Integer	$\{ s \in Z \mid -2^7 \leq s < 2^7 \}$
INT	Integer	$\{ s \in Z \mid -2^{15} \leq s < 2^{15} \}$
DINT	Double Integer	$\{ s \in Z \mid -2^{31} \leq s < 2^{31} \}$
LINT	Long Integer	$\{ s \in Z \mid -2^{63} \leq s < 2^{63} \}$
USINT	Unsigned Short Integer	$\{ u \in N \mid 0 \leq u < 2^8 \}$
UINT	Unsigned Integer	$\{ u \in N \mid 0 \leq u < 2^{16} \}$
UDINT	Unsigned Double Integer	$\{ u \in N \mid 0 \leq u < 2^{32} \}$
ULINT	Unsigned Long Integer	$\{ u \in N \mid 0 \leq u < 2^{64} \}$
REAL	Real numbers	definiert in [IEC 559]
LREAL	Long reals	definiert in [IEC 559]
TIME	Zeitspanne	nicht genau definiert; eine Zeitspanne kann positiv und negativ sein; mögliche Zeiteinheiten sind Tage, Stunden, Minuten, Sekunden und Millisekunden Es soll als Wertebereich angenommen werden: $\{$ (signum, days, hours, minutes, seconds, milliseconds) $\mid$ signum $\in \{-, +\} \wedge$ days$\in N \wedge$ hours$\in N \wedge$ minutes$\in N \wedge$ seconds$\in N \wedge$ milliseconds$\in N \}$

Typname	**Beschreibung**	**Wertebereich**
DATE	Datum (ausschließlich ohne Zeit)	{ (day, month, year) | day∈ N ∧ month∈ N ∧ year∈ N ∧ day≤31 ∧ month≤12 }
TIME_OF_DAY oder TOD	Tageszeit (ohne Datum)	{ (hour, minute, second, millisecond) | hour∈ N ∧ minute∈ N ∧ second∈ N ∧ millisecond∈ N ∧ hour≤23 ∧ minute≤59 ∧ second≤59 ∧ millisecond≤99}
DATE_AND_TIME oder DT	Datum und Tageszeit	$\text{Wertebereich}_{\text{DATE}} \times \text{Wertebereich}_{\text{TIME\_OF\_DAY}}$
STRING	Zeichenkette	(ASCII\{'$', ' " ', " ' "} ∪ {'$$', '$L' , '$N' , '$P', '$R' , '$T' , '$l' , '$n' , '$p' , '$r' , '$t' })*
WSTRING	Zeichenkette	(Unicode\{'$', ' " ', " ' "} ∪ {'$$', '$L', '$N', '$P', '$R', '$T', '$l', '$n', '$p', '$r', '$t' })*