

Verifikation von Funktionsbausteinadaptern durch Modelchecking

Verification of Function Block Adapters through Modelchecking

Torsten Heverhagen

Herrn Prof. Dr.-Ing. Rudolf Tracht zum 60. Geburtstag gewidmet

Die Unified Modeling Language (UML) gewinnt in der Automatisierungstechnik immer mehr an Bedeutung. In der Version 2.0 enthält die UML so genannte „Ports“, die unter anderem eine Integration der UML in ein stark heterogenes automatisierungstechnisches Umfeld erleichtern. Speziell zur Integration mit Sprachen für speicherprogrammierbare Steuerungen wurden Ports der UML als Schnittstelle zu Funktionsbausteinen der IEC 61131-3 eingesetzt. Die Umsetzung der unterschiedlichen Protokollarten erfolgt hierbei durch Funktionsbausteinadapter (FBAs). In diesem Artikel wird das logische, plattformunabhängige Verhalten von FBAs mit Hilfe kommunizierender endlicher Automaten beschrieben. Durch symbolisches Modelchecking wird dieses Automatenmodell anschließend auf häufige Spezifikationsfehler überprüft.

The Unified Modeling Language (UML) is more and more important to the field of automation and control. Within version 2.0 the UML contains so called "Ports". Among other things, Ports ease the integration of the UML into heterogeneous environments, which are typical for automation and control. Especially for the integration with languages for programmable controllers Ports are used as interfaces to function blocks of the IEC 61131-3. The transformation of different protocol types is handled by function block adapters. The logical, platform-independent behavior of a function block adapter is described by a system of communicating finite automata. These automata are investigated by symbolic model checking for some typical specification errors.

Schlagwörter: UML, IEC 61131-3, Endlicher Automat, Verifikation, Temporale Logik, Funktionsbaustein

Keywords: Unified Modeling Language, programmable controllers, finite automaton, verification, temporal logic, function block

1 Einleitung

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

In heutigen Produktionsunternehmen findet man oberhalb der Steuerungsebene hauptsächlich Standard-Personal-

computer (PCs) oder Workstations, deren Aufgaben in den Bereichen Betriebsdatenerfassung, Produktionsplanung, Produktionssteuerung und Visualisierung liegen. Auf PC-Technik basierende Automatisierungsgeräte werden aber auch in der Steuerungsebene parallel zu SPSen zunehmend eingesetzt. 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.

In [1] wurden Funktionsbausteinadapter (FBAs) vorgestellt, die es bereits in der Entwurfsphase eines Systems erlauben, Kommunikationsbeziehungen zwischen Klassen der UML und Funktionsbausteinen der IEC 61131-3 zu modellieren. Mit FBAs werden Funktionsbausteine so ummantelt, dass sie aus Sicht der UML wie Klassen angesprochen werden können. Aus der Sicht der IEC 61131-3 können FBAs wie Funktionsbausteine behandelt werden.

In diesem Artikel soll das logische, plattformunabhängige Verhalten von FBAs auf typische Spezifikationsfehler untersucht werden. Es gibt zurzeit noch keine Werkzeuge, die eine gemeinsame Verifikation von Funktionsbausteinen und UML-Klassen erlauben. Der in diesem Beitrag gewählte Ansatz zur Verifikation von FBAs 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 UML-Protokoll und FB-Protokoll heißen. Sie werden in den Abschnitten 2.1 und 2.2 vorgestellt.

Als notwendige Ergänzung zur getrennten Verifikation der UML-Klassen und der Funktionsbausteine wird ein FBA dahingehend verifiziert, ob er die Übersetzung zwischen UML-Protokoll und FB-Protokoll fehlerfrei durchführt. Diese Verifikation erfolgt durch Modelchecking mit Hilfe des Werkzeuges SMV [2]. Dazu müssen ein FBA und dessen Protokolle in ein System kommunizierender endlicher Automaten überführt werden, um ein für den Modelchecker geeignetes mathematisches Modell zu erhalten [3].

Dieser Beitrag gliedert sich weiterhin wie folgt: Zunächst wird im Abschnitt 2 ein konkretes Beispiel für einen FBA und dessen Schnittstellenprotokolle eingeführt und erläutert. Im Abschnitt 3 wird als notwendige Vorarbeit zur Verifikation ein System kommunizierender endlicher Automaten entwickelt, das den Beispiel-FBA und dessen Umgebung modelliert. Das entwickelte Automatenmodell wird in Abschnitt 4 als Modell zur Verifikation des FBA verwendet. Eine Zusammenfassung schließt diesen Beitrag in Abschnitt 5 ab.

2 Beispiel „Bohrung“

Als Anwendungsbeispiel für einen Funktionsbausteinadapter soll eine einfache Bohreinheit dienen. Dieses Beispiel ist dem PLCopen Standard „Function blocks for motion control“ [4] entnommen. Eine Bohrung soll wie in Bild 1 dargestellt durch drei Bewegungsabläufe realisiert werden.

Das 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 Bohrergeschwindigkeit 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.

Bild 2 zeigt ein Zeitdiagramm für Weg und Geschwindigkeit der translatorischen Bewegung der Bohrmaschine. Jeder Abschnitt dieser Bewegung soll mit einem Funktionsbaustein gesteuert werden. Bild 3 zeigt vier Funktionsbausteininstanzen, deren Namen den Abschnitten in Bild 2 entsprechen.

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 der Bohrung vorgegeben werden. Ist der Bewegungsablauf vollständig abgeschlossen, wird das durch eine positive Flanke in *rev.Done* signalisiert. Im Fehlerfall wird eine positive Flanke in

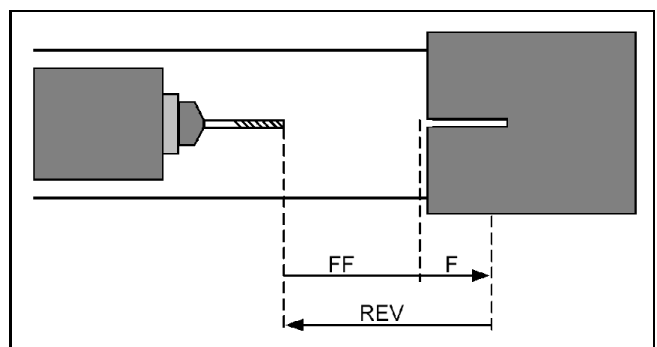


Bild 1: Prinzipschema Bohrung (Quelle: [4]).

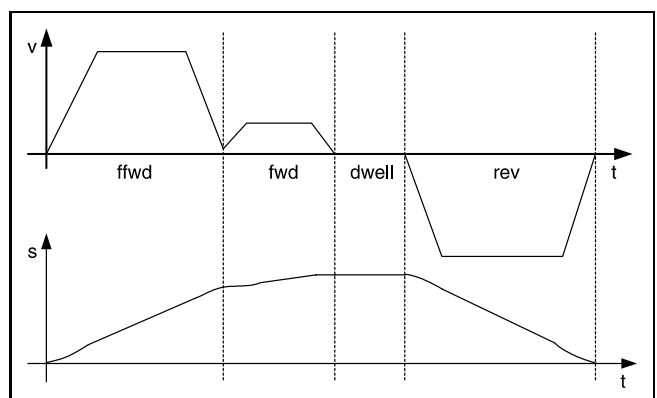


Bild 2: Geschwindigkeit und Weg (Quelle: [4]).

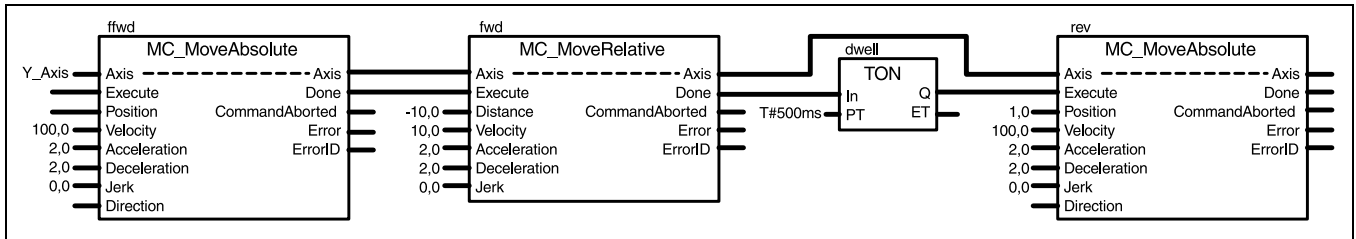


Bild 3: FBS-Diagramm (Quelle: [4]).

Error ausgegeben. Ein Fehlercode wird in *ErrorID* bereitgestellt. Wird aufgrund einer vorzeitigen negativen Flanke in *ffwd.Execute* der Bohrvorgang abgebrochen, gibt ein gerade aktiver FB eine positive Flanke in *CommandAborted* aus.

Da für die weiteren Betrachtungen nur die offenen Eingangs- und Ausgangsvariablen von Bedeutung sind, soll die Bewegungssteuerung aus Bild 3 im nächsten Abschnitt zu einem übergeordneten Funktionsbaustein zusammengefasst werden.

2.1 Das Protokoll der Bewegungssteuerung

Bild 4 zeigt den Funktionsbaustein *MC_FB*, der die Funktionsbausteininstanzen aus Bild 3 enthält und zusammenfasst. Die Eingangsvariable *Execute* ist direkt mit *ffwd.Execute* aus Bild 3 verbunden. Die Ausgangsvariable *Ended* kennzeichnet die Beendigung des Bohrvorganges, was aufgrund eines Abbruchs, eines Fehlers oder der vollständigen Ausführung des Bohrvorganges geschehen kann. Bild 5 zeigt die Beschaltung von *Ended*. Die übrigen Ausgangsvariablen sind direkt mit Ausgängen aus Bild 3 verbunden (z. B. *Done* mit *rev.Done*, *ffwd_CA* mit *ffwd.CommandAborted* usw.).

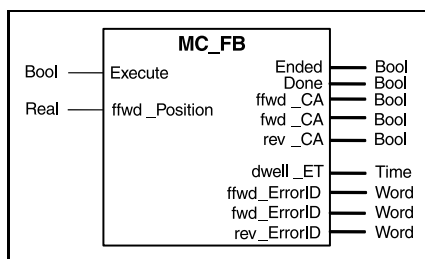


Bild 4: Zusammengefasster Funktionsbaustein *MC_FB*.

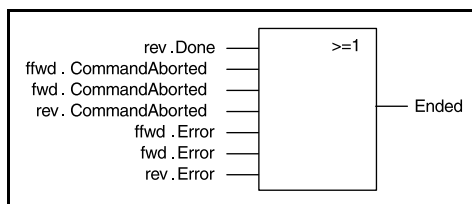


Bild 5: Schaltung für das *Ended*-Signal.

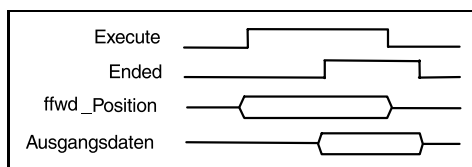


Bild 6: Zeitdiagramm für das Protokoll des Funktionsbausteines.

Das Zeitdiagramm in Bild 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*.

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 Flanke 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.

2.2 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 objektorientiert mit Hilfe der UML/C++ entwickelt wurde. Im Bild 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.

Die Operation *Ended* von *IDrillEnd* hat einen Parameter vom Typ der Klasse *DrillEndData* (Bild 8). Damit der Zustand der Bewegungssteuerung ausgewertet wer-

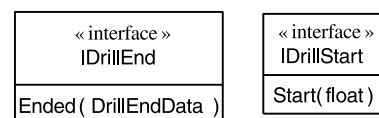


Bild 7: Interface-Klassen.

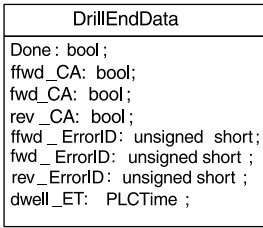


Bild 8: UML-Klasse *DrillEndData*.

den kann, enthält *DrillEndData* ein Attribut 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 *PLCTime* in C++ definiert.

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

Die UML wurde in Version 2.0 [5] um so genannte Ports erweitert, die bereits aus [6] und [7] bekannt sind. In Bild 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 (Bild 9 rechts). Das damit definierte Verhalten soll zur Unterscheidung vom FB-Protokoll im Folgenden als UML-Protokoll bezeichnet werden.

Das Protokoll-Statechart aus Bild 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 sehr sinnvoll eingesetzt werden. Das soll in Abschnitt 3 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.

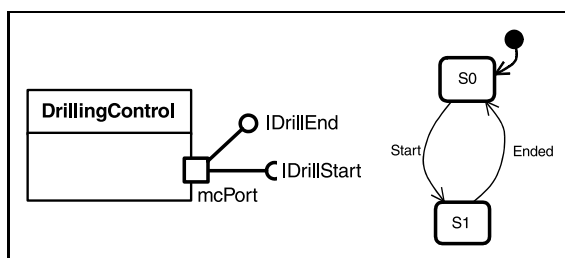


Bild 9: UML-Klasse *DrillingControl* mit Port *mcPort* und Protokoll-Statechart für das *mcPort*.

2.3 Der Funktionsbausteinadapter MC_FBA

Damit der Funktionsbaustein *MC_FB* und die Klasse *DrillingControl* miteinander kommunizieren können, benötigt man einen Protokolladapter. Der in Bild 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 [1].

Ein Funktionsbausteinadapter besitzt gleichermaßen Schnittstellen zu einem Port einer UML-Klasse wie auch zu einem Funktionsbaustein. Der *MC_FBA* aus Bild 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*.

In Bild 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 Bild 11 ist eine Kombination eines Strukturdiagramms aus UML Version 2.0 [5] und der Funktionsbausteinsprache aus IEC 61131-3, wie in [1] erstmals vorgeschlagen wurde.

Das dynamische Verhalten von FBAs wird durch deren Operationen beschrieben. Der *MC_FBA* besitzt nur eine Operation, die in Bild 12 angegeben ist. Die verwendete Spezifikationsprache 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 ([8; 9]).

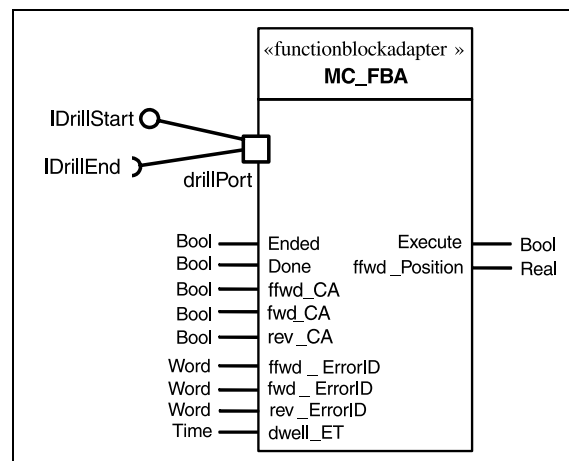


Bild 10: Klasse für *MC_FBA*.

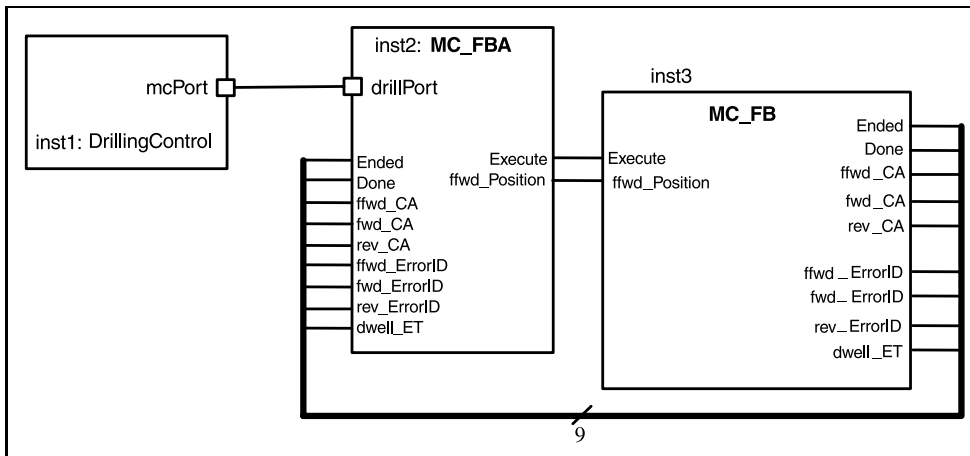


Bild 11: Strukturdiagramm kombiniert mit der FB-Sprache.

```

1) On_UMLSignal( s1: drillPort.Start)
2)   s2: drillPort.Ended;
3) Begin
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)  sendAsync(s2);
18) End_On_UMLSignal

```

Bild 12: FBA-Operation in FBA-Sprache.

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 Operation stehenden Semantik erfolgt in Abschnitt 3.3 anhand eines endlichen Automaten.

3 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 Au-

tomaten 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 [10]. 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 Moore-Automaten ([10]) modelliert: $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$. Dabei ist Δ ein endliches Ausgabealphabet und λ die Ausgabefunktion von Q auf Δ .

3.1 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 (Bild 9). 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 Bild 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 Bild 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 Bild 9 sind, gegeben. Zusätzlich benötigt man noch ein Eingabezeichen 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.

Tabelle 1: Bedeutung der Eingabezeichen.

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.

Tabelle 2: Übergangsfunktion.

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}

Ended senden wollen) zu definieren. Für das *drillPort* ist $\Sigma_{port} = \{e_0, e_1, e_2, e_3\}$.

Die Bedeutungen der Eingabezeichen sind in Tabelle 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.

In Tabelle 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.

3.2 Beschreibung des FB-Protokolls als endlichen Automaten M_{fbp}

Die Aufgabe eines endlichen Automaten für FB-Protokolle ist vergleichbar mit der eines 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 Bild 6 lässt sich der (ebenfalls ausgabefreie) 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 3 zeigt die Zuordnung der Zustände zu den Variablenbelegungen.

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 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 3: Bedeutung der Zustände.

Zustand	Belegung von Execute und Ended
q ₀	$\neg Execute \wedge \neg Ended$
q ₁	$Execute \wedge \neg Ended$
q ₂	$Execute \wedge Ended$
q ₃	$\neg Execute \wedge Ended$
q _{Err}	beliebig

Tabelle 4: Bedeutung der Eingabezeichen.

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
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. e₈) 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.

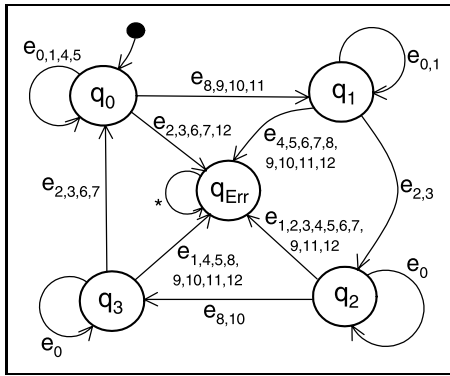


Bild 13: Transitionsdiagramm für M_{fbp} ($e_{x,y}, \dots \Leftrightarrow e_x \vee e_y \vee \dots$).

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. Bild 13 zeigt M_{fbp} als Zustandsdiagramm.

3.3 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 5 und Tabelle 6 erläutert.

Tabelle 5: Bedeutung der Eingabesymbole.

Eingabesymbol	Eingabe vom Funktionsbaustein		Eingabe vom DrillingControl
	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 6: Bedeutung der Ausgabesymbole.

Ausgabesymbol	Ausgabe an den Funktionsbaustein		Ausgabe an DrillingControl
	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 7 beschrieben werden.

Tabelle 7: Bedeutung der Zustände.

Zustand	Interpretation
q_0	Die FBA-Operation wird noch nicht ausgeführt. Der FBA wartet.
q_1	Der Wert in <i>ffwd_Position</i> wird geändert, bevor <i>Execute</i> auf <i>true</i> gesetzt wird. Zeile 4) in Bild 12.
q_2	<i>Execute</i> wird auf <i>true</i> gesetzt. Zeile 5) in Bild 12.
q_3	<i>Execute</i> und <i>ffwd_Position</i> werden gleichzeitig gesetzt.
q_4	Es wird auf eine positive Flanke in <i>Ended</i> gewartet. Zeile 6)
q_5	Die Ausgangsdaten des <i>MC_FB</i> werden eingelesen. Da das Einlesen nur FBA-intern geschieht, werden die Zeilen 7) bis 14) in Bild 12 zu einem Zustand zusammengefasst.
q_6	Es wird eine negative Flanke in <i>Execute</i> ausgegeben.
q_7	Es wird auf eine negative Flanke in <i>Ended</i> gewartet.
q_8	Das <i>Ended</i> -Signal wird an <i>DrillingControl</i> geschickt.
q_{Err}	M_{fba} erhielt eine unerwartete Eingabe.

Im Bild 14 ist aus Gründen der besseren Übersichtlichkeit der Zustand q_{Err} nicht enthalten. Man kann deshalb nicht die vollständige Übergangsfunktion δ_{fba} und Ausgabefunktion λ_{fba} daraus ablesen. (Im Zustand q_{Err} hat M_{fba} die Ausgabe a_0 . Die vollständige Übergangsfunktion enthält zusätzlich zu den Transitionen aus Bild 14 noch Übergänge zu q_{Err} , die mit den Eingabesymbolen beschriftet sind, die in den Zuständen q_0 bis q_8 nicht erlaubt sind.)

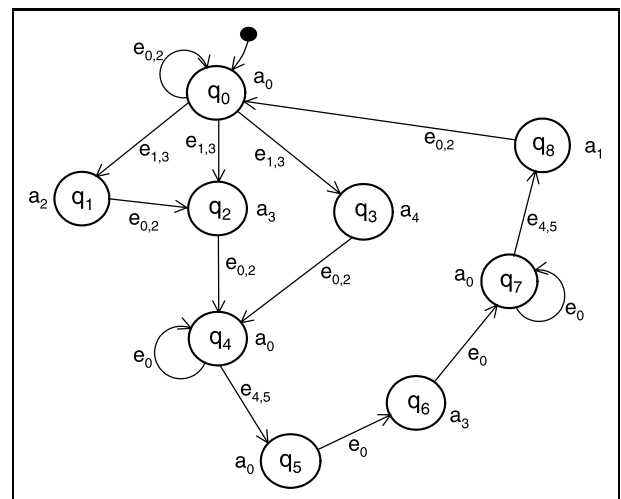


Bild 14: Transitionsdiagramm zum FBA-Automat (ohne q_{Err}).

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 Bild 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 Bild 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 aufeinander folgenden SPS-Zyklen geschrieben, dann durchläuft M_{fba} den Pfad $q_0 \rightarrow q_1 \rightarrow q_2$. Durch die hier vorgestellte Modellierung können somit auch unterschiedliche Implementierungsmöglichkeiten in der späteren Verifikation berücksichtigt werden.

3.4 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. Aus Platzgründen können M_{fb} und M_{drill} in diesem Beitrag nicht vollständig beschrieben werden. Für die Aussagen in Abschnitt 4.2 sind aber folgende Informationen notwendig:

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

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

3.5 Kommunikation zwischen den Automaten

Damit die Automaten miteinander kommunizieren können, wird in diesem Abschnitt für jeden Automaten 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 FBAs 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 *MC_FB* (M_{fb}) sind nur von den Ausgaben von *MC_FBA* (M_{fba}) abhängig:

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

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 1, Bild 14 und Abschnitt 3.4 entnehmen kann.

Bild 15 veranschaulicht grafisch die Kommunikationsstruktur der Automaten.

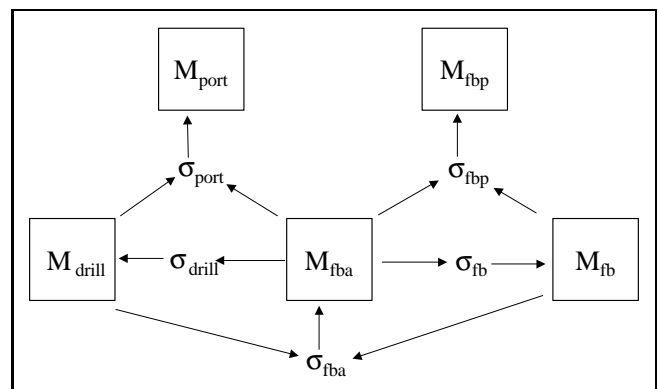


Bild 15: Kommunikationsbeziehungen zwischen den Automaten.

4 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 [2].

Mit Hilfe der SMV Eingabesprache können Kripke-Strukturen und temporallogische Aussagen in CTL (Computation Tree Logic) spezifiziert werden [3]. Dazu wird zunächst in Abschnitt 4.1 gezeigt, wie die Automaten aus Abschnitt 3 in eine Kripke-Struktur überführt werden. In Abschnitt 4.2 werden für verschiedene Eigenschaften CTL-Formeln entwickelt, die verifiziert werden sollen. Die Abschnitte 4.3 und 4.4 erläutern beispielhaft den Umgang mit SMV.

4.1 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 [3].

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 3 folgendermaßen beschreiben:

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

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

$$I = \{s_0\}$$

$$R = \{(s, s') \mid$$

$$(s'[q_{\text{drill}}] = \delta_{\text{drill}}(s[q_{\text{drill}}], \sigma_{\text{drill}}(\lambda_{\text{fba}}(s[q_{\text{fba}}])))$$

$$\wedge (s'[q_{\text{port}}] =$$

$$\delta_{\text{port}}(s[q_{\text{port}}], \sigma_{\text{port}}(\lambda_{\text{drill}}(s[q_{\text{drill}}]), \lambda_{\text{fba}}(s[q_{\text{fba}}])))$$

$$\wedge (s'[q_{\text{fba}}] =$$

$$\delta_{\text{fba}}(s[q_{\text{fba}}], \sigma_{\text{fba}}(\lambda_{\text{drill}}(s[q_{\text{drill}}]), \lambda_{\text{fb}}(s[q_{\text{fb}}])))$$

$$\wedge (s'[q_{\text{fbp}}] =$$

$$\delta_{\text{fbp}}(s[q_{\text{fbp}}], \sigma_{\text{fbp}}(\lambda_{\text{fba}}(s[q_{\text{fba}}]), \lambda_{\text{fb}}(s[q_{\text{fb}}])))$$

$$\wedge (s'[q_{\text{fb}}] = \delta_{\text{fb}}(s[q_{\text{fb}}], \sigma_{\text{fb}}(\lambda_{\text{fba}}(s[q_{\text{fba}}])))\}$$

Die Markierungsfunktion kann mit

$$AP = \{(s[q_{\text{drill}}] = x_1) \mid x_1 \in Q_{\text{drill}}\}$$

$$\cup \{(s[q_{\text{port}}] = x_2) \mid x_2 \in Q_{\text{port}}\}$$

$$\cup \{(s[q_{\text{fba}}] = x_3) \mid x_3 \in Q_{\text{fba}}\}$$

$$\cup \{(s[q_{\text{fbp}}] = x_4) \mid x_4 \in Q_{\text{fbp}}\}$$

$$\cup \{(s[q_{\text{fb}}] = x_5) \mid x_5 \in Q_{\text{fb}}\}$$

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

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

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

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

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 3 beschrieben.

4.2 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 („es existiert ein Pfad“), den Allquantor A („für alle Pfade“) 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:

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

oder

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

Weitere Beispiele für zu verifizierende Eigenschaften sind:

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

$$AG (s[q_{\text{drill}}] = q_1 \Rightarrow$$

$$AX (s[q_{\text{fba}}] = q_1 \vee s[q_{\text{fba}}] = q_2 \vee s[q_{\text{fba}}] = q_3))$$

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

$$AG (s[q_{\text{drill}}] = q_1 \Rightarrow$$

$$AF (s[q_{\text{port}}] = q_0 \wedge s[q_{\text{fba}}] = q_0 \wedge s[q_{\text{fbp}}] = q_0))$$

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

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

4.3 Die SMV Eingabesprache

Um die Kripke-Struktur und die CTL-Spezifikationen dem SMV Tool zur Verifikation zur Verfügung stellen zu kön-

nen, müssen sie in der SMV-Sprache formuliert werden. Bild 16 zeigt den Automat M_{port} als Modul in der SMV-Sprache formuliert.

```

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;
  };
}

```

Bild 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 3 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, dem Gesamtsystem, werden für jedes Automaten-Modul Variablen mit den Namen *drill*, *port*, *fba*, *fbp* und *fb* angelegt und durch die Kommunikationsfunktionen zusammenschaltet. 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 4.2 wird in der SMV-Syntax wie folgt ausgedrückt:

```

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

```

4.4 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 solcher Gegenbeispiele kann die Spezifikation aus Abschnitt 2 und daraus resultierend das Automatenmodell aus Abschnitt 3 so lange verbessert werden, bis alle CTL-Formeln erfüllt sind.

Die in den Abschnitten 2 und 3 vorgestellten Modelle sind bereits das Ergebnis einer Überarbeitung durch das Modelchecking, sodass alle CTL-Formeln als verifiziert gelten.

5 Zusammenfassung und Ausblick

In diesem Beitrag 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 Funktionsbausteinen und Ports zuständig ist.

Ein FBA spezifiziert die Protokollumsetzung zwischen Funktionsbausteinen und Ports 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 Beitrag vorgestellten Automatenmodelle verfeinert und auf gleiche Weise wie in Abschnitt 4 vorgestellt verifiziert werden.

In [8] wurden zwei Realisierungen eines FBA gegenübergestellt, wobei in beiden Fällen die UML-Klasse auf einem Industrie-PC und der Funktionsbaustein auf einer SPS (SIMATIC-S7) liefen, aber unterschiedliche Kommunikationskanäle verwendet wurden (PROFIBUS und Direktverdrahtung von digitalen Ein-/Ausgängen). Weitere plattformabhängige Aspekte von FBAs wurden in [11] und [9] diskutiert.

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.

Eine vollständige Definition der Syntax und Semantik von FBAs in der Form eines UML-Profiles wurde in [12] angegeben.

Danksagung

Für die hilfreichen Hinweise und Diskussionen zu diesem Beitrag möchte ich mich bei Prof. Dr.-Ing. R. Tracht, Dr. Martin Große-Rhode und Prof. Dr. B. Müller-Clostermann bedanken.

Literatur

- [1] *T. Heverhagen, R. Tracht*: Integrating UML-RealTime and IEC 61131-3 with Function Block Adapters. Proc. of ISORC 2001, IEEE Computer Society, S. 395–402.
- [2] Symbolic Model Verifier.
URL: <http://www-2.cs.cmu.edu/~modelcheck/smv.html>
[Zugriff am: 2.10.2002].
- [3] *E.M. Clarke, O. Grumberg, D. Peled*: Model Checking. MIT Press, 2000.
- [4] *PLCopen Technical Committee 2 TF*: Function Blocks for Motion Control. 2001, S. 53–54
URL: <http://www.plcopen.org/forms/motioncontrol.htm>
[Zugriff am: 2.10.2002].
- [5] *OMG Dokument ad/2002-09-02*: Unified Modeling Language: Superstructure. Version 2 Beta R1 (draft),
URL: <ftp://ftp.omg.org/pub/docs/ad/02-09-02.pdf>
[Zugriff am: 2.10.2002].
- [6] *B. Selic, J. Rumbaugh*: Using UML for Complex Real-Time Systems. URL: <http://www.rational.com/products/rosert/whitepapers.jsp> [Zugriff am: 2.10.2002].
- [7] *B. Selic, G. Gullekson, P.T. Ward*: Real-Time Object-Oriented Modeling. John Wiley & Sons, 1994.
- [8] *T. Heverhagen, R. Tracht*: Implementing Function Block Adapters. OMER – Object-Oriented Modeling of Embedded Real-Time Systems (Andy Schürr (Hrsg.)), GI-Edition – Lecture Notes in Informatics (LNI), P-5, Bonner Köllen Verlag, Bonn 2002, S. 122–134.
- [9] *T. Heverhagen, J. Shi, M. von Groll, R. Tracht*: Kommunikation zwischen Funktionsbausteinen und UML Capsules in einer industriellen Softwareumgebung. VDI-Tagung Software Engineering in der industriellen Praxis, VDI-Berichte 1666, VDI-Verlag, Düsseldorf 2002, S. 121–132.
- [10] *J.E. Hopcroft, J.D. Ullman*: Einführung in die Automaten- und Komplexitätstheorie, Formale Sprachen und Komplexitätstheorie. Addison-Wesley, 1994.
- [11] *T. Heverhagen, R. Tracht*: Echtzeitanforderungen bei der Integration von IEC 61131-3 Funktionsbausteinen und UML-RT Capsules. PEARL 2001, Echtzeitkommunikation und Ethernet/Internet (P. Holleczeck, B. Vogel-Heuser (Hrsg.)), Informatik aktuell, Springer-Verlag 2001, S. 87–96.
- [12] *T. Heverhagen*: Integration von Sprachen für speicherprogrammierbare Steuerungen in die Unified Modeling Language durch Funktionsbausteinadapter. Dissertation Universität Essen, Essen, 2003.

Manuskripteingang: 14. Oktober 2002.



Dipl.-Ing. Torsten Heverhagen ist wissenschaftlicher Mitarbeiter am Lehrstuhl für Automatisierungstechnik (Prof. Dr.-Ing. R. Tracht) des Fachbereiches 12 an der Universität Essen. Sein Hauptarbeitsgebiet ist der Softwareentwurf im Bereich der Automatisierungstechnik.

Adresse: Lehrstuhl für Automatisierungstechnik, FB 12, Universität Essen, D-45117 Essen, Tel.: +49 (201) 183-4320, Fax: +49 (201) 183-2089, E-Mail: Torsten.Heverhagen@uni-essen.de