Funktionale Modellierung eines Postsystems

Claus Dendorfer
Institut für Informatik
Technische Universität München
Postfach 20 24 20, D-8000 München 2

Oktober 1991

Zusammenfassung

Als Fallstudie wird ein Teil eines realen Betriebssystems formal spezifiziert, nämlich das Postsystem des *Multiprocessor Multitasking Kernel* MMK. Der verwendete Formalismus basiert auf der Beschreibung von stromverarbeitenden Funktionen mit Prädikaten. Da es sich für die Spezifikation als nützlich erweist, den Zustand des Postsystems explizit zu verwalten, wird ein zustandsorientierter funktionaler Spezifikationsstil verwendet.

1. Einleitung

Der an der Technischen Universität München entwickelte verteilte Betriebssystemkern MMK ermöglicht die Kommunikation zwischen parallel ablaufenden Prozessen über Postfächer [BBLT 90, S. 8ff]. Da deren Eigenschaften, insbesondere im Hinblick auf das Echtzeitverhalten, nicht offensichtlich sind, bietet sich eine formale Spezifikation an. Im folgenden wird eine solche Spezifikation gegeben. Besonders berücksichtigt wurden "ergonomische" Gesichtspunkte wie Lesbarkeit und Modularität.

Eine Spezifikation beschreibt das Verhalten eines Systems unabhängig von dessen Implementierung und dient als "Kontrakt" zwischen Auftraggeber und Systementwickler. Es zeigt sich auch, daß das Vorhandensein einer formalen Spezifikation zu besseren Implementierungen führt, da potentielle Fehlerquellen früher erkannt werden [Hall 90]. Überdies kann die Korrektheit von sicherheitskritischen Programmen nur bezüglich einer formalen Anforderungsspezifikation bewiesen werden.

Die vorliegende Arbeit besteht aus drei Teilen. Kapitel 2 enthält die Definitionen aller Operatoren des Spezifikationsformalismus in komprimierter Form. Generell hat dieses Kapitel aber nicht den Zweck, den verwendeten Formalismus umfassend einzuführen; dazu wird zum Beispiel auf [Broy et al. 91] verwiesen. Der Leser sollte dieses Kapitel zunächst nur überfliegen und erst bei Schwierigkeiten mit der Spezifikation gründlicher zu Rate ziehen.

Kapitel 3 stellt die eigentliche Spezifikation dar, wie sie (zum Beispiel in einer industriellen Umgebung) als Ergebnis des ersten Entwicklungsabschnittes betrachtet werden könnte, also als "Vertragsgrundlage" zwischen den Spezifizierern und den Implementierern. Dieses Kapitel soll das Postsystem – neben der vollständigen formalen Definition – auch intuitiv verständlich machen. Gemäß dem Ziel, ein Beispiel für eine "reale" Spezifikation zu liefern, finden sich kaum Bemerkungen zum verwendeten Formalismus (ebensowenig, wie man in einem Pascal-Programmtext Anmerkungen zur Sprache Pascal erwarten würde). Einige Kommentare und Querverweise auf die Kapitel 2 und 4 wurden jedoch in eckigen Klammern eingefügt.

In Kapitel 4 stehen weitere Anmerkungen, alternative Ansätze und Gründe für einige Entwurfsentscheidungen. Dieses Kapitel ist für das Verständnis der hier vorgestellten Methode nicht notwendig; es ist vielmehr für die Bewertung und Einordnung des verwendeten Formalismus von Interesse.

2. Einführung in den Formalismus

2.1 Ströme und Standardfunktionen auf Strömen

Wir bezeichnen eine endliche oder unendliche Sequenz als *Strom.* Für die Menge aller Ströme von Elementen aus einer Menge M schreiben wir M^ω:

M^*	Menge der endlichen Sequenzen über M
M^∞	Menge der unendlichen Sequenzen über M
M^ω	Menge der Ströme über M (Definition: $M^\omega = M^* \cup M^\infty$)

Als Standardnotation für Ströme verwenden wir die Aufzählung ihrer Elemente in spitzen Klammern; insbesondere bezeichnet $\langle\rangle$ den leeren Strom.

Ströme werden gebildet durch Anwendung der *Präfixoperation* &: für jedes $m \in M$ und jeden Strom $s = \langle m_1,m_2,\ldots\rangle \in M^\omega$ ist $m\&s$ der Strom $\langle m,m_1,m_2,\ldots\rangle$.

Die Anwendung einer Funktion f auf ein Argument x schreiben wir $f.x$ oder $f(x)$ oder f_x. Die Funktionsapplikation hat eine größere Bindungskraft als alle Infix-Operatoren und ist linksassoziativ. Der Ausdruck $f.\sigma.x \,\&\, y$ wird also $(f.\sigma).x \,\&\, y$ gelesen und könnte auch $f_\sigma.x \,\&\, y$ geschrieben werden. Die funktionale Komposition schreiben wir mit dem Symbol $\circ$ und

#x ergibt die Länge eines Stromes x (Präfix-Notation)

$\#\_ : M^\omega \to \mathbb{N} \cup \{\infty\}$ $\quad \#\langle\rangle = 0$ $\quad \#(m\&x) = 1 + \#x$

x^y ergibt Strom x konkateniert mit y (Infix-Notation; für unendliche x ist x^y = x)

$\_\hat{}\_ : M^\omega \times M^\omega \to M^\omega$ $\quad \langle\rangle\hat{}x = x$ $\quad (m\&x)\hat{}y = m\&(x\hat{}y)$

ft selektiert das erste Element eines nicht-leeren Stroms (ft.⟨⟩ ist nicht definiert)

$ft : M^\omega \to M$ $\quad$ (n. def.) $\quad ft(m\&x) = m$

rt selektiert den Rest eines Stroms

$rt : M^\omega \to M^\omega$ $\quad rt.\langle\rangle = \langle\rangle$ $\quad rt(m\&x) = x$

S©x löscht aus x die Elemente, die nicht in S sind ("Filter-Operator")

$\_©\_ : \wp(M) \times M^\omega \to M^\omega$ $\quad S©\langle\rangle = \langle\rangle$ $\quad S©(m\&x) =$ **if** $m \in S$ **then** $m \,\&\, S©x$ **else** $S©x$

Tabelle 1: Standardfunktionen auf Strömen

definieren: $(f \circ g).x = g.(f.x)$. Wir verwenden in der Spezifikation eine Reihe von Standardfunktionen auf Strömen, die in Tabelle 1 angegeben sind.

Auf der Menge der Ströme M^ω definieren wir die partielle *Präfix-Ordnung* $\sqsubseteq$ durch:

$$x \sqsubseteq y \;\equiv\; \exists\, z \in M^\omega : x \hat{}\, z = y$$

Mit dieser Ordnung bildet M^ω einen vollständigen Bereich mit kleinstem Element $\langle\rangle$, wobei M^∞ gerade die Elemente enthält, die notwendig sind, um M^* gegenüber der Bildung der kleinsten oberen Schranken zu vervollständigen.

2.2 Spezifikation von stromverarbeitenden Funktionen

Unter einer *stromverarbeitenden Funktion* verstehen wir eine Funktion, die als Argument einen Strom (oder ein Tupel von Strömen) hat, als Ergebnis einen Strom (oder ein Tupel von Strömen) liefert, und die bezüglich der oben definierten Präfixordnung (beziehungsweise ihrer Erweiterung auf Tupel) stetig ist. Eine solche Funktion wird spezifiziert durch die Angabe ihrer Eigenschaften, also durch ein Prädikat. Die Forderung nach Stetigkeit (und damit auch nach Monotonie) ist immer implizit vorhanden; sie braucht nicht extra angegeben zu werden. Die Menge aller stetigen Funktionen mit Typ $M^\omega \to N^\omega$, deren Ausgabe immer mindestens die Länge der Eingabe hat, wird zum Beispiel beschrieben durch:

$$P.f \;\equiv\; \forall\, x \in M^\omega : \#x \le \#f.x$$

Für alle $m \in M$ verhält sich $f \mathbin{«} m$ so wie f nach Einlesen des Elements m:
$f \mathbin{«} m : M^\omega \to N^\omega \qquad (f \mathbin{«} m).x = f(m\&x)$
Für alle $n \in N$ gibt $n \mathbin{«} f$ zuerst n aus und verhält sich dann wie f:
$n \mathbin{«} f : M^\omega \to N^\omega \qquad (n \mathbin{«} f).x = n \,\&\, f.x$
Für alle $y \in N^\omega$ gibt $y \mathbin{«} f$ zuerst den Strom y aus und verhält sich dann wie f:
$y \mathbin{«} f : M^\omega \to N^\omega \qquad (y \mathbin{«} f).x = y \wedge f.x$

Tabelle 2: Hilfsfunktionen für Spezifikationen

Gegeben sei eine stromverarbeitende Funktion $f : M^\omega \to N^\omega$. Tabelle 2 definiert drei Hilfsfunktionen, mit denen sich Spezifikationen oft einfacher aufschreiben lassen (der Operator « wird hier dreifach überladen).

Beispielsweise kann eine Funktion $f : \mathbb{N}^\omega \to \mathbb{N}^\omega$, die jedes Element des Eingabestroms verdoppelt, spezifiziert werden durch: $P.f \equiv \forall\, n \in \mathbb{N} : f \mathbin{«} n = 2*n \mathbin{«} f$. Zusammen mit der impliziten Forderung nach Stetigkeit ist die Menge der Funktionen, die P erfüllen, einelementig, und f damit eindeutig definiert.

Oft ist es bequem, ein Prädikat über stromverarbeitenden Funktionen rekursiv zu definieren. Generell gibt es viele Prädikate, die eine gegebene Gleichung erfüllen. Wir vereinbaren, daß durch eine Gleichung stets das *schwächste* Prädikat definiert wird, das diese Gleichung erfüllt. So kann eine Funktion vom Typ $\mathbb{N}^\omega \to \mathbb{N}^\omega$, die jedes Element des Eingabestroms mindestens verdoppelt, so spezifiziert werden:

$$P.f \equiv \forall\, n \in \mathbb{N} : \exists\, g \in \mathbb{N}^\omega \to \mathbb{N}^\omega,\ m \in \mathbb{N} : f \mathbin{«} n = (2*n + m) \mathbin{«} g \ \wedge\ P.g$$

Zu beachten ist, daß eine Definition der Form $f \mathbin{«} n = (2*n + m) \mathbin{«} f$ zu speziell wäre, denn dann müßte f auf gleiche Eingabeelemente jedesmal den gleichen Ausgabewert liefern.

2.3 Modellierung von Systemen

In einem ersten Überblick läßt sich die funktionale Systembeschreibung in die folgenden zwei Schritte aufteilen. Für eine genauere Darstellung verweisen wir auf [Broy et al. 91].

1. Schritt: Einführung und Klassifizierung von *Aktionen*

   Aktionen sind das Verbindungsstück zwischen dem realen System und der Spezifikation. Unter Aktionen verstehen wir Ereignisse, die zu einem irgendeinem Zeit*punkt* stattfinden und keine Dauer haben. Für ein Ereignis, dessen Dauer modelliert werden soll, führt man zwei Aktionen ein; je eine, um den Beginn des Ereignisses und dessen Ende darzustellen.

Jede Aktion wird entweder als *Eingabe*- oder als *Ausgabeaktion* klassifiziert. Eingabeaktionen sind solche, die das System nicht beeinflussen kann und auf die es reagieren kann; Ausgabeaktionen werden ausschließlich vom System bestimmt.

2. Schritt: Spezifikation von stromverarbeitenden Funktionen

Es ist nun eine stromverarbeitende Funktion zu spezifizieren, die ein (Tupel von) Strömen aus Eingabeaktionen auf ein (Tupel von) Strömen aus Ausgabeaktionen abbildet. Jede Funktion, die diese Spezifikation erfüllt, modelliert ein zulässiges Systemverhalten. Nichtdeterminismus wird auf dieser Ebene durch Unterspezifikation dargestellt. Ebenso werden Anforderungen an die Umgebung (*Rely*-Bedingungen) dadurch modelliert, daß das Systemverhalten nicht eingeschränkt ist, wenn diese Anforderungen nicht erfüllt sind.

3. Spezifikation des MMK-Postsystems

3.1 Informelle Beschreibung

Das Postsystem des MMK ist ein Kommunikationsmechanismus zwischen nebenläufigen Prozessen. Postfächer können eingerichtet und gelöscht werden, und Nachrichten können an existierende Postfächer gesendet beziehungsweise von ihnen empfangen werden. Es gibt keine Zugriffskontrolle, so daß jeder Prozeß auf jedes Postfach lesend und schreibend zugreifen und das Postfach auch löschen kann.

Jedes Postfach verwaltet eine Schlange von Nachrichten, deren maximale Länge bei der Initialisierung angegeben wird. Ist die Nachrichtenschlange bei einer Leseanforderung nicht leer, so wird die erste Nachricht aus der Schlange gelöscht und an den lesenden Prozeß übergeben. Ist die Nachrichtenschlange bei einer Schreibanforderung nicht voll, so wird die zu schreibende Nachricht an das Ende der Schlange gesetzt und eine Rückmeldung an den schreibenden Prozeß übergeben.

Zusätzlich enthält jede Lese- und Schreibanforderung einen Parameter mit der maximalen Wartezeit. Kann eine Anforderung nicht sofort bearbeitet werden, weil die Nachrichtenschlange voll beziehungsweise leer ist, so wird der entsprechende Prozeß in einen Wartezustand versetzt. Der Prozeß wird wieder aktiviert, wenn die Anforderung erfolgreich beantwortet wurde oder die maximale Wartezeit abgelaufen ist. Im letzteren Fall wird ein Zeitfehler als Rückmeldung gegeben. Für Postfächer, deren Kapazität 0 ist, kommt eine direkte (synchrone) Kommunikation zwischen zwei Prozessen zustande, wenn innerhalb der Wartezeit einer Anforderung eine komplementäre Anforderung eingeht.

Die hier informell vorgestellten vier Dienste des MMK-Postsystems sind in [BBLT 90] auf den Seiten 35f, 41, 49 und 53 als C-Systemaufrufe beschrieben.

3.2 Aktionen der formalen Modellierung

[Zur methodischen Verwendung von Aktionen siehe Abschnitt 2.3, Schritt 1.]

Um die vier Dienste des MMK-Postsystems formal zu modellieren, führen wir pro Systemaufruf zwei atomare Aktionen ein, nämlich eine *Befehls-* und eine *Rückmeldeaktion*. Befehlsaktionen sind Eingabeaktionen des Postsystems; sie entsprechen dem Zeitpunkt, zu dem ein Systemaufruf begonnen wird. Rückmeldeaktionen sind Ausgaben; sie kennzeichnen das Ende eines Systemaufrufes. Der aufrufende Prozeß befindet sich zwischen diesen Aktionen in einem Wartezustand.

Die Befehlsaktionen ähneln den C-Systemaufrufen. Allerdings enthält jede Befehlsaktion noch einen Identifikator, der die eindeutige Zuordnung der entsprechenden Rückmeldung sicherstellt, wenn die Umgebung diesen Parameter geeignet besetzt. Die Spezifikation selbst stellt nur sicher, daß der Identifikator der Befehlsaktion in der dazugehörenden Rückmeldung wiederholt wird. Neben diesem Parameter (und der Information, daß der Systemaufruf beendet wurde) enthält jede Rückmeldeaktion auch die Ergebnisse des Aufrufs, die in C durch Prozedurparameter übergeben werden.

Weiterhin enthalten die Mengen der Befehls- und Rückmeldeaktionen jeweils das spezielle Element "tick", das in unserer diskreten Zeitmodellierung das Ende eines Zeitintervalls darstellt. Alle Aktionen zwischen zwei solchen Zeitmarken gelten als gleichzeitig.

[Näher gehen wir auf diese Zeitdarstellung im folgenden Abschnitt ein. Einen Vergleich mit alternativen Modellierungen enthält Abschnitt 4.1.]

Wir definieren nun die Menge Cmd der Befehlsaktionen für das Postsystem und die Menge Reply der Rückmeldeaktionen. Dabei stützen wir uns auf folgende Mengen, die als fest vorgegeben betrachtet werden:

Cid	Bezeichner zur Zuordnung der Befehls-/Rückmeldungspaare
Mbx	Postfachbezeichner
Msg	Nachrichten
Time	Wartezeit, ausgedrückt in Zeitintervallen (Definition: Time = $\mathbb{N} \cup \{\infty\}$)

Die Menge Cmd enthält genau die Elemente der folgenden Bauart (es seien $c \in$ Cid, $s \in \mathbb{N}$, $b \in$ Mbx, $m \in$ Msg, $t \in$ Time):

crembox(c,s)	Einrichtung eines neuen Postfaches mit maximalem Fassungsvermögen von s Nachrichten
sendmsg(c,b,m,t)	Senden einer Nachricht m an Postfach b mit maximaler Wartezeit von t Zeitintervallen
recmsg(c,b,t)	Lesen einer Nachricht aus Postfach b mit maximaler Wartezeit von t Zeitintervallen

delmbox(c,b)	Löschen des Postfachs b
tick	Ende eines Zeitintervalls

Wir definieren außerdem die beiden Mengen Send $\subset$ Cmd und Rec $\subset$ Cmd, wobei Send genau die in Cmd enthaltenen Sendebefehle (sendmsg) umfaßt, und Rec die in Cmd enthaltenen Empfangsbefehle (recmsg).

Die Menge Reply enthält genau die Elemente der folgenden Bauart (seien $c \in$ Cid, $b \in$ Mbx, $m \in$ Msg):

ok(c)	erfolgreiches Senden einer Nachricht, oder Löschen eines Postfaches
ok-rec(c,m)	erfolgreiches Empfangen der Nachricht m
ok-cre(c,b)	erfolgreiches Initialisieren des Postfaches b
memory-error(c)	Fehler beim Initialisieren (zu viel Speicherplatz angefordert)
parameter-error(c)	Zugriff auf ein nicht (oder nicht mehr) existierendes Postfach
timeout(c)	Zeitfehler beim Senden oder Empfangen
tick	Ende eines Zeitintervalls

3.3 Definition des Gesamtsystems

Jedes mögliche Verhalten des Postsystems wird beschrieben durch eine stetige stromverarbeitende Funktion f : Mfun, wobei Mfun = $\text{Cmd}^{\omega} \rightarrow \text{Reply}^{\omega}$ ist. Insgesamt modellieren wir das System durch ein Prädikat MAILSYSTEM, so daß MAILSYSTEM.f genau dann gilt, wenn f einem zulässigen Systemverhalten entspricht.

Eine Besonderheit bei der verwendeten Zeitmodellierung ist, daß in gezeiteten Strömen auch bei Aktionen, die als gleichzeitig interpretiert werden, noch deren Reihenfolge innerhalb des Zeitintervalls als zusätzliche Information vorhanden ist. Dies erlaubt es uns, einen Teil des auftretenden Nichtdeterminismus umzuwandeln in ein deterministisches, von dieser Reihenfolge abhängiges Verhalten.

Um durch die Spezifikation die ganze Bandbreite des möglichen Systemverhaltens abzudecken, muß sichergestellt werden, daß das Postsystem nach außen hin unabhängig von der zufälligen Mikrostruktur der Eingaben ist. Das heißt, daß für zwei Eingabeströme x und y,

die sich nur hinsichtlich der Reihenfolge innerhalb der einzelnen Zeitintervalle unterscheiden, auch $\{ f.x \mid MAILSYSTEM.f \} = \{ f.y \mid MAILSYSTEM.f \}$ gilt. Die folgende Definition von MAILSYSTEM mit expliziten Permutationsfunktionen garantiert dies:

MAILSYSTEM.f $\equiv$

Eine Funktion erfüllt das Prädikat MAILSYSTEM, wenn sie sich aus der funktionalen Komposition von einer Permutationsfunktion, einer Postfachfunktion (im engeren Sinne) und wieder einer Permutationsfunktion ergibt:

$$\exists\, h_1 \in Cmd^{\omega} \to Cmd^{\omega},\ g \in Mfun,\ h_2 \in Reply^{\omega} \to Reply^{\omega} :$$
$$PERM.h_1 \wedge MSYS.g \wedge PERM.h_2 \wedge f = h_1 \circ g \circ h_2$$

Eine Permutationsfunktion im obigen Sinne ist eine Funktion, die die Elemente innerhalb der einzelnen Zeitabschnitte beliebig umordnet:

PERM.f $\equiv \forall\, x \in Cmd^{\omega} \cup Reply^{\omega} :$

Zeitmarken ("tick") werden nicht verzögert:

$$f.(x \hat{}\langle tick\rangle) = f.x \hat{}\langle tick\rangle$$

Permutationen finden nur innerhalb eines durch Zeitmarken ("tick") begrenzten Abschnittes statt:

$$x \hat{}\langle tick\rangle \approx f.(x \hat{}\langle tick\rangle)$$

Verwendetes Hilfsprädikat:

$x \approx y$ gilt genau dann, wenn x eine Permutation von y ist:

$$\_\approx\_ : (M^{\omega} \times M^{\omega}) \to \mathbb{B}$$
$$x \approx y \equiv \forall\, m \in M : \#\{m\}©x = \#\{m\}©y$$

3.4 Definition des Zustandsraumes und des eigentlichen Postsystems

[Siehe Abschnitt 4.2 zur Motivation des zustandsorientierten Spezifikationsstils.]

Wir verwenden einen zustandsorientierten Spezifikationsstil. Die Zustände sind strukturiert; sie enthalten für jeden Postfachbezeichner fünf Attribute, die über die folgenden Selektorfunktionen abgefragt werden:

active: Mbx $\to$ State $\to \mathbb{B}$	Boole'scher Wert, der angibt, ob dem Postfachbezeichner ein initialisiertes (und nicht gelöschtes) Postfach zugeordnet

	ist; für ein nicht-aktives Postfach sind die Werte der anderen Attribute beliebig.
size: Mbx → State → ℕ	Fassungsvermögen des Postfachs (maximale Anzahl der Nachrichten).
mail: Mbx → State → Msg$^\omega$	Schlange der im Postfach gespeicherten Nachrichten.
wst: Mbx → State → Send$^\omega$	Schlange mit den Befehlsaktionen der auf Rückmeldung wartenden sendenden Prozesse.
wrt: Mbx → State → Rec$^\omega$	Schlange mit den Befehlsaktionen der auf Rückmeldung wartenden empfangenden Prozesse.

Außerdem sei eine Operation zur punktweisen Änderung des Zustands verfügbar. Zum Beispiel schreiben wir σ[active.b→false], um den Zustand auszudrücken, der aus σ hervorgeht, wenn die active-Komponente des Postfachs b auf false gesetzt wird. Die Zustandsmenge State und die Selektor- und Änderungsfunktionen definieren wir nicht eindeutig. Vielmehr geben wir nur Bedingungen an, die von dieser Menge beziehungsweise den Funktionen erfüllt werden müssen.

[Siehe Abschnitt 4.3 zur Konsistenz dieser Definition und zur Charakterisierung von erreichbaren und unterscheidbaren Elementen. Siehe Abschnitt 4.4 zu alternativen Möglichkeiten der Definition von Zustandsräumen.]

State

active, size, mail, wst, wrt : *wie oben*

_ [_ . _→_] : State × (Mbx → State → V) × Mbx × V → State

Die Menge State darf nicht leer sein:

$\text{State} \neq \emptyset$

Punktweise Veränderung des Zustands:

$\forall\, \sigma \in \text{State}, a, a' \in \text{Mbx} \to \text{State} \to V, b, b' \in \text{Mbx}, v \in V :$

$a, a' \in \{\text{active}, \text{size}, \text{mail}, \text{wst}, \text{wrt}\} \Rightarrow$

$a'.b'.(\sigma[a.b \to v]) = \textbf{if}\ a' = a \wedge b' = b\ \textbf{then}\ v\ \textbf{else}\ a'.b'.\sigma$

Das eigentliche Postsystem MSYS, also die Funktion, die aus einer Sequentialisierung der Eingaben genau eine Sequentialisierung der Rückmeldungen liefert, wird durch das folgende Prädikat beschrieben:

$\text{MSYS.f} \equiv \exists\, \sigma \in \text{State} : \forall\, b \in \text{Mbx} :$

Eine Funktion erfüllt das Prädikat MSYS, wenn sie das Prädikat MS angewandt auf einen initialen Zustand erfüllt. Der initiale Zustand wird im MMK durch eine Konfigurationsdatei beschrieben. Wir begnügen uns mit der Forderung, daß im initialen Zustand aktive Postfächer weder gespeicherte Nachrichten noch wartende Prozesse haben dürfen:

$$MS.\sigma.f \;\wedge$$

$$active.b.\sigma \;\Rightarrow\; mail.b.\sigma = \langle\rangle \wedge wst.b.\sigma = \langle\rangle \wedge wrt.b.\sigma = \langle\rangle$$

MS ist definiert als das schwächste Prädikat, das die Bedingungen der Abschnitte 3.5 bis 3.9 erfüllt.

3.5 Beschreibung des crembox-Befehls

$$MS.\sigma.f \;\Rightarrow\; \forall\, c \in Cid,\, s \in \mathbb{N} : \exists\, b \in Mbx,\, \sigma' \in State,\, g \in Mfun :$$

Ein Postfach wird eingerichtet und der entsprechende Bezeichner zurückgegeben (erfolgreicher Fall)

$$f \,\text{«}\, crembox(c,s) = ok\text{-}cre(c,b) \,\text{«}\, g \;\wedge$$

$$\neg\, active.b.\sigma \;\wedge\; MS.\sigma'.g \;\wedge$$

$$active.b.\sigma' \wedge size.b.\sigma' = s \wedge mail.b.\sigma' = \langle\rangle \wedge wst.b.\sigma' = \langle\rangle \wedge wrt.b.\sigma' = \langle\rangle$$

oder, nichtdeterministisch

$\vee$

tritt ein Speicherüberlauf auf.

$$f \,\text{«}\, crembox(c,s) = memory\text{-}error(c) \,\text{«}\, g \;\wedge\; MS.\sigma.g$$

3.6 Beschreibung des sendmsg-Befehls

$MS.\sigma.f \;\Rightarrow\; \forall\, c \in Cid, b \in Mbx, m \in Msg, t \in Time : \exists\, r \in Rec^{\omega}, g \in Mfun :$

Zur Schreiberleichterung sei r die Sequenz der bei diesem Postfach wartenden Empfangsbefehle:

$r = wrt.b.\sigma$

Falls das Postfach aktiv ist und Empfangsbefehle warten (kann nur bei leerem Postfach der Fall sein), dann wird die Nachricht direkt übertragen:

$active.b.\sigma \;\wedge\; r \neq \langle\rangle \;\Rightarrow$

$f \mathbin{«} sendmsg(c,b,m,t) = ok(c) \mathbin{«} ok\text{-}rec(getc(ft.r), m) \mathbin{«} g \;\wedge$

$MS_{\sigma[wrt.b \rightarrow rt.r]}.g$

Falls das Postfach aktiv ist, keine Empfangsbefehle warten und Platz frei ist, dann wird die Nachricht im Postfach gespeichert:

$active.b.\sigma \;\wedge\; r = \langle\rangle \;\wedge\; \#mail.b.\sigma < size.b.\sigma \;\Rightarrow$

$f \mathbin{«} sendmsg(c,b,m,t) = ok(c) \mathbin{«} g \;\wedge$

$MS_{\sigma[mail.b \rightarrow mail.b.\sigma^{\frown}\langle m\rangle]}.g$

Falls das Postfach aktiv ist, keine Empfangsbefehle warten und kein Platz frei ist, dann wird der Sendebefehl in die Warteschlange eingereiht:

$active.b.\sigma \;\wedge\; r = \langle\rangle \;\wedge\; \#mail.b.\sigma = size.b.\sigma \;\Rightarrow$

$f \mathbin{«} sendmsg(c,b,m,t) = g \;\wedge$

$MS_{\sigma[wst.b \rightarrow wst.b.\sigma^{\frown}\langle sendmsg(c,b,m,t)\rangle]}.g$

Falls das Postfach nicht aktiv ist, erfolgt eine entsprechende Rückmeldung:

$\neg\, active.b.\sigma \;\Rightarrow\; f \mathbin{«} sendmsg(c,b,m,t) = parameter\text{-}error(c) \mathbin{«} f$

[Die einzelnen Blöcke in der obigen Definition sind durch Konjunktionen verknüpft.]

Verwendete Hilfsfunktion:

getc selektiert den Befehlsbezeichner aus einem Sende- oder Empfangsbefehl:

$getc : Send \cup Rec \rightarrow Cid$

$getc.sendmsg(c,b,m,t) = c$

$getc.recmsg(c,b,t) = c$

3.7 Beschreibung des recmsg-Befehls

$MS.\sigma.f \;\Rightarrow\; \forall\, c \in Cid,\, b \in Mbx,\, t \in Time : \exists\, s \in Send^{\omega},\, m \in Msg^{\omega},\, g \in Mfun :$

Zur Schreiberleichterung sei s die Sequenz der bei Postfach b wartenden Sendebefehle und m die Nachrichtenschlange von b, der die Nachricht des ersten wartenden Sendebefehls angehängt ist:

$s = wst.b.\sigma$

$s \neq \langle\rangle \;\Rightarrow\; m = mail.b.\sigma \,^\frown\, getm(ft.s)$

Falls das Postfach aktiv ist und Sendebefehle warten, dann rückt der erste Sendebefehl in die (temporäre) Nachrichtenschlange m nach, und die erste Nachricht aus m wird übertragen:

$active.b.\sigma \;\wedge\; s \neq \langle\rangle \;\Rightarrow$

$f \mathbin{«} recmsg(c,b,t) = ok\text{-}rec(c, ft.m) \mathbin{«} ok(getc(ft.s)) \mathbin{«} g \;\wedge$

$MS_{\sigma[mail.b \to rt.m][wst.b \to rt.s]}.g$

Falls das Postfach aktiv und nicht-leer ist und keine Sendebefehle warten, dann wird die erste Nachricht direkt an den Empfangsbefehl übertragen:

$active.b.\sigma \;\wedge\; s = \langle\rangle \;\wedge\; \#mail.b.\sigma > 0 \;\Rightarrow$

$f \mathbin{«} recmsg(c,b,t) = ok\text{-}rec(c, ft.(mail.b.\sigma)) \mathbin{«} g \;\wedge$

$MS_{\sigma[mail.b \to rt.(mail.b.\sigma)]}.g$

Falls das Postfach aktiv und leer ist und keine Sendebefehle warten, dann wird der Empfangsbefehl in die Warteschlange eingereiht:

$active.b.\sigma \;\wedge\; s = \langle\rangle \;\wedge\; \#mail.b.\sigma = 0 \;\Rightarrow$

$f \mathbin{«} recmsg(c,b,t) = g \;\wedge$

$MS_{\sigma[wrt.b \to wrt.b.\sigma \,^\frown \langle recmsg(c,b,t)\rangle]}.g$

Falls das Postfach nicht aktiv ist, erfolgt eine entsprechende Rückmeldung:

$\neg\, active.b.\sigma \;\Rightarrow\; f \mathbin{«} recmsg(c,b,t) = parameter\text{-}error(c) \mathbin{«} f$

Verwendete Hilfsfunktionen:

getc selektiert den Befehlsbezeichner aus einem Sende- oder Empfangsbefehl (siehe Abschnitt 3.6)

getm selektiert die Nachricht aus einem Sendebefehl:

$getm : Send \to Msg$

$getm.sendmsg(c,b,m,t) = m$

3.8 Beschreibung des delmbox-Befehls

$MS.\sigma.f \Rightarrow \forall c \in Cid, b \in Mbx : \exists g \in Mfun :$

Falls das Postfach aktiv ist, dann wird es gelöscht und alle wartenden Prozesse erhalten eine Rückmeldung mit Parameterfehler:

$active.b.\sigma \Rightarrow$

$f \mathbin{«} delmbx(c,b) = mkerr.(wst.b.\sigma) \mathbin{«} mkerr.(wrt.b.\sigma) \mathbin{«} ok(c) \mathbin{«} g \wedge$

$MS_{\sigma[active.b \rightarrow false]}.g$

Falls das Postfach nicht aktiv ist, erfolgt ein Parameterfehler:

$\neg\, active.b.\sigma \Rightarrow f \mathbin{«} delmbox(c,b) = parameter\text{-}error(c) \mathbin{«} f$

Verwendete Hilfsfunktionen:

mkerr generiert zu einer Liste von Sende- oder Empfangsbefehlen die entsprechenden Parameterfehler:

$mkerr : Send^{\omega} \cup Rec^{\omega} \rightarrow Reply^{\omega}$

$mkerr \mathbin{«} x = parameter\text{-}error(getc.x) \mathbin{«} mkerr$

getc selektiert den Befehlsbezeichner aus einem Sende- oder Empfangsbefehl (siehe Abschnitt 3.6)

3.9 Beschreibung des tick-Befehls

$MS.\sigma.f \Rightarrow \exists\, w \in \{timeout(c) \mid c \in Cid\}^{\omega}, \sigma' \in State, g \in Mfun :$

w ist eine Liste, die für jeden Befehlsbezeichner genau so viele timeout-Meldungen enthält, wie Befehle mit Zeitparameter 0 und diesem Befehlsbezeichner bei irgendeinem aktiven Postfach warten:

$\forall\, c \in Cid : \#\{timeout(c)\} \copyright w = expired.c.\sigma$

Beim Ablauf eines Zeitintervalls werden die anfallenden timeout-Fehler generiert:

$f \ll tick = w \ll tick \ll g \wedge MS.\sigma'.g$

Alle an aktiven Postfächern wartenden Befehle mit Zeitparameter 0 werden gelöscht; bei den anderen wartenden Befehlen wird der Zeitparameter um 1 vermindert:

$\forall\, a \in Mbx \to State \to V, b \in Mbx : active.b.\sigma \Rightarrow a.b.\sigma' = timestep.a.b.\sigma$

Verwendete Hilfsfunktionen und -mengen:

expired.c.σ ist die Anzahl der wartenden Befehle mit Bezeichner c, deren Wartezeit abgelaufen ist. Offensichtlich gilt $0 \le expired.c.\sigma \le 1$, wenn die Befehlsbezeichner eindeutig sind:

$expired : Cid \to State \to \mathbb{N}$

$expired.c.\sigma = \sum_{b \in Mbx,\ active.b.\sigma} \#EXP.c \copyright (wst.b.\sigma \wedge wrt.b.\sigma)$

EXP.c ist die Menge aller Befehle mit Identifikator c und Zeitparameter 0:

$EXP.c = \{sendmsg(c,b,m,0) \mid b \in Mbx, m \in Msg\} \cup \{recmsg(c,b,0) \mid b \in Mbx\}$

timestep verringert für alle wartenden Befehle eines Postfaches die Zeitkomponente und löscht wartende Befehle mit Zeitkomponente 0:

$timestep : (Mbx \to State \to V) \to Mbx \to State \to V$

$timestep.a.b.\sigma = \textbf{if}\ a \in \{wst, wrt\}\ \textbf{then}\ step(a.b.\sigma)\ \textbf{else}\ a.b.\sigma$

step verringert für eine Befehlswarteschlange jeweils die Zeitkomponente und löscht Befehle mit Zeitkomponente 0:

$step : Send^{\omega} \cup Rec^{\omega} \to Reply^{\omega}$

$step.(sendmsg(c,b,m,t) \,\&\, x) = \textbf{if}\ t > 0\ \textbf{then}\ sendmsg(c,b,m,t-1) \,\&\, step.x\ \textbf{else}\ step.x$

$step.(recmsg(c,b,t) \,\&\, x) = \textbf{if}\ t > 0\ \textbf{then}\ recmsg(c,b,t-1) \,\&\, step.x\ \textbf{else}\ step.x$

4. Bemerkungen zum verwendeten Formalismus

4.1 Zeitmodellierung

Mehrere Zeitdarstellungen sind möglich. Der Formalismus muß gleichzeitige Aktionen erlauben, da im Postsystem zum Beispiel beliebig viele timeout-Fehler gleichzeitig auftreten können. Der in [BD 90] vorgestellte Zeitbegriff ist daher weniger geeignet. Für die vorliegende Arbeit wurden folgende drei Zeitmodellierungen in Erwägung gezogen:

a) Die Menge der gezeiteten Ströme über M ist $(M \times \mathbb{N})^\omega$, wobei die zweite Komponente eines Stromelements einen Zeitstempel darstellt. Diese Zeitstempel wachsen im Strom monoton.

b) Die Menge der gezeiteten Ströme über M ist $(M \cup \{tick\})^\omega$, wobei die Stromelemente zwischen zwei Zeitmarken "tick" als jeweils zu einem Zeitintervall gehörig betrachtet werden. Kein solches Intervall enthält unendlich viele Elemente.

c) Die Menge der gezeiteten Ströme über M ist $(\wp M)^\omega$, wobei jedes Stromelement die Menge derjeniger Aktionen darstellt, die während *eines* Zeitintervalls aufgetreten sind. Keine solche Menge enthält unendlich viele Elemente.

Für einen diskreten Zeitbegriff erscheint die erste Modellierung unnötig kompliziert, ohne einen greifbaren Vorteil zu bieten. Eine Darstellung, bei der jeder Zeitpunkt einer reellen Zahl entspricht, läßt sich allerdings nur über die Zeitstempelmethode bewerkstelligen.

Die dritte Modellierung ist methodisch gesehen die sauberste, da hier die Ströme keine "Mikrostruktur" neben der Zeitinformation tragen. Auf die expliziten Permutationsfunktionen PERM kann dann verzichtet werden. Allerdings ist es umständlich, Funktionen zu spezifizieren, die auf Strömen von Mengen operieren. Die folgende Formel drückt zum Beispiel aus, daß für alle Zustände σ unter der Bedingung $P.\sigma$ und bei Eingabe des Befehls u ein Zustandsübergang nach $\tau.\sigma$ möglich ist, der als Ausgabe die Menge R von Rückmeldungen liefert:

$$\forall\, \sigma \in \text{State},\ f \in \text{Mfun},\ U \in \wp\text{Cmd},\ \mathcal{U} \in (\wp\text{Cmd})^\omega :$$
$$MS.\sigma.f \ \wedge\ P.\sigma \ \wedge\ u \in U$$
$$\Rightarrow$$
$$\exists\, g \in \text{Mfun},\ W \in \wp\text{Reply},\ \mathcal{W} \in (\wp\text{Reply})^\omega :$$
$$f.(U \,\&\, \mathcal{U}) = W \,\&\, \mathcal{W} \ \wedge\ R \subseteq W \ \wedge\ MS.(\tau.\sigma).g \ \wedge\ g.((U \setminus \{u\}) \,\&\, \mathcal{U}) = (W \setminus R) \,\&\, \mathcal{W}$$

Natürlich können Schreiberleichterungen für Formeln dieser Art eingeführt werden, so zum Beispiel die Notation von Zustandsübergängen als Tabelle. Dazu ist allerdings ein erheblicher Aufwand an zusätzlichen Definitionen notwendig, der sich für eine relativ kleine Spezifikation wie die vorliegende nicht lohnt. Auch lassen sich nicht alle Zustandsübergänge nach diesem Schema beschreiben; dies gilt zum Beispiel für die Zeitfortschaltung (also die Reaktion auf die leere Eingabemenge).

Die zweite Zeitmodellierung erlaubt es, mit "normalen" Strömen zu arbeiten. Nachdem die Konzentration des Nichtdeterminismus auf zwei explizite Permutationsfunktionen insbesondere auch für die Beweistechnik Vorteile hat, wurde diese Modellierung gewählt.

4.2 Zustandsorientierte Spezifikation

Eine Spezifikation kann zustandsorientiert oder ablauforientiert aufgebaut sein. In einer ablauforientierten Spezifikation werden Funktionen durch direkte Charakterisierung ihrer Eigenschaften, also der zulässigen Ein- und Ausgabeströme beschrieben (so zum Beispiel auf S. 22 von [BD 90]). Dagegen wird bei der zustandsorientierten Spezifikation der Zustand als Hilfsobjekt eingeführt. Abhängig vom bisherigen Zustand bewirken Eingaben eine sofortige Ausgabe und einen Zustandsübergang, der wiederum die Wirkung zukünftiger Eingaben beeinflußt. Oft verwendete Formalismen für die zustandsorientierte Spezifikation sind zum Beispiel VDM [Jones 86], Z [Spivey 87] oder die Transitions-Axiom-Methode [Lamport 83]. Auch Mischformen sind möglich; Zustände können zum Beispiel als Prädikate über Abläufen codiert werden.

Ein Einwand gegenüber zustandsorientierten Spezifikationen ist, daß die Definition eines Zustandsraumes (oder auch nur die algebraische Spezifikation von Zugriffs- und Modifizierungsfunktionen) schon eine erste Entwurfsentscheidung darstellt, die auf der Ebene der abstrakten Spezifikation eines Systems noch nicht getroffen werden sollte. Dagegen ist einzuwenden, daß der Zustandsbegriff der Spezifikation mit dem der Implementierung nicht identisch sein muß. Zustandsorientierte Spezifikationen sind oft leichter zu schreiben, zu lesen und zu verstehen. Dafür lassen sich folgende Gründe aufführen:

a) Wenn die Bedeutung der Zustandskomponenten intuitiv klar ist, kann man die Wirkung der Eingaben unabhängig voneinander verstehen. Zustände sind also eine "enge Schnittstelle" zwischen den einzelnen Aktionen. Demgegenüber wird bei der ablauforientierten Spezifikation oft eine Reihe von Eingaben mit einer Reihe von Ausgaben in Beziehung gesetzt, was die Modularität der Spezifikation verringert und die Wirkung der *einzelnen* Aktionen weniger klar werden läßt.

b) Wenn der Zustandsraum mit hinreichend mächtigen Mitteln (zum Beispiel algebraischer Spezifikation) definiert wurde, so lassen sich Standarddatentypen verwenden, die schon

einen Teil der Funktionalität eines Systems abdecken. So können zum Beispiel Schlangen von Nachrichten als Zustandskomponenten auftreten, was schon einige Eigenschaften impliziert ("first-in-first-out", zerstörendes Auslesen). In einer ablauforientierten Spezifikation müßten solche Eigenschaften durch Prädikate explizit ausgedrückt werden.

c) Die einzelnen Komponenten eines strukturierten Zustands bieten beim Schreiben der Spezifikation eine Prüfliste, ob alle Wirkungen einer Eingabe berücksichtigt wurden. Demgegenüber hat man bei ablauforientierten Spezifikationen keinen Anhaltspunkt, wann genug (oder alle) Systemeigenschaften beschrieben sind.

Diese Argumentation erhebt keinen Anspruch auf Allgemeingültigkeit. Es sind durchaus Systeme vorstellbar, die sich bequemer ablauforientiert beschreiben lassen. Insbesondere ist es im allgemeinen nicht möglich, Lebendigkeitseigenschaften in einem zustandsorientierten Stil zu spezifizieren. Diese treten im Beispiel der Postsystemspezifikation nicht auf, da die explizite Zeitdarstellung Lebendigkeits- in Sicherheitseigenschaften umwandelt. Insgesamt wurde daher die vorliegende Spezifikation zustandsorientiert erstellt.

4.3 Anmerkungen zur Definition des Zustandsraumes

In Abschnitt 3.4 wurden der Zustandsraum State und die Selektor- und Änderungsfunktionen durch eine Reihe von Bedingungen definiert. Diese Technik ist angelehnt an die mathematische Schreibweise. Im Gegensatz zur Methode der formalen algebraischen (axiomatischen) Spezifikation wird hier beispielsweise State nicht als Sortenbezeichner, sondern gleich als Bezeichner für die entsprechende Trägermenge in einem beliebigen Modell aufgefaßt. Es muß natürlich gezeigt werden, daß die Definition überhaupt konsistent ist, also daß mathematische Strukturen existieren, die die Bedingungen erfüllen. Im vorliegenden Fall macht man sich leicht klar, daß State als die Menge der Terme verstanden werden kann, die durch eine Konstante anystate und die Änderungsfunktion $\_[\_.\_\rightarrow\_]$ gebildet sind. Zusätzlich geben wir nun noch eine weitere, implementierungsnähere Definition des Zustandsraumes, die ebenfalls den aufgestellten Bedingungen genügt:

$State = Mbx \rightarrow (\mathbb{B} \times \mathbb{N} \times Msg^{\omega} \times Send^{\omega} \times Rec^{\omega})$

$active.b.\sigma = select1.(\sigma.b)$ … *analog für size, mail, wst, wrt*

$select1.(p,q,r,s,t) = p$ … *analog für select2 bis select5*

$\sigma[a.b\rightarrow v] = \sigma'$, **where** $\sigma'.b' =$ **if** $a = active \wedge b = b'$ **then** $update1.(\sigma.b).v$
elif … *analog für size, mail, wst, wrt*
else $\sigma.b'$

$update1.(p,q,r,s,t).v = (v,q,r,s,t)$ … *analog für update2 bis update5*

Jeder Zustandsraum, der die in Abschnitt 3.4 gegebenen Bedingungen erfüllt, enthält überflüssige Elemente, also Zustände, die entweder von keinem zulässigen Anfangszustand aus erreichbar sind oder die sich von anderen Zuständen nicht beobachtbar unterscheiden. Eine Reihe von Fehlern in einer Spezifikation läßt sich aufdecken, wenn solche Elemente formal beschrieben werden, denn dann können zwei unabhängige Beschreibungen der erreichbaren beziehungsweise unterscheidbaren Zustände (eine explizite und die durch die Spezifikation implizierte) auf ihre Konsistenz überprüft werden.

Wir betrachten zunächst nicht-erreichbare Zustände. Im Beispiel des Postsystems genügen die erreichbaren Zustände folgenden zusätzlichen Bedingungen:

$\text{Invar}(\sigma) \equiv \forall\, b \in \text{Mbx} :$
Ein aktives Postfach kann nicht mehr Nachrichten enthalten als seine Kapazität angibt: $\text{active.b.}\sigma \Rightarrow \#\text{mail.b.}\sigma \leq \text{size.b.}\sigma$ *An einem aktiven Postfach mit Nachrichten dürfen keine Empfangsbefehle warten:* $\text{active.b.}\sigma \wedge \#\text{mail.b.}\sigma \neq \langle\rangle \Rightarrow \text{wrt.b.}\sigma = \langle\rangle$ *An einem aktiven Postfach mit Platz in der Nachrichtenschlange dürfen keine Sendebefehle warten:* $\text{active.b.}\sigma \wedge \#\text{mail.b.}\sigma < \text{size.b.}\sigma \Rightarrow \text{wst.b.}\sigma = \langle\rangle$

Natürlich ist es nicht sinnvoll, das Prädikat Invar in die Definition des Zustandsraumes einzuarbeiten, da dann entweder die Definition inkonsistent werden würde oder die geforderten Bedingungen - unabhängig von der eigentlichen Spezifikation - garantiert wären. Vielmehr muß man Invar als Invariante betrachten und diese Eigenschaft (also, daß Invar durch alle Startzustände erfüllt und durch alle erlaubten Transitionen erhalten wird) als zusätzliches Theorem beweisen.

Äquivalente Zustände sind im Postsystembeispiel solche, bei denen sich nur die Attributwerte von inaktiven Postfächern unterscheiden:

$\text{Equiv}(\sigma,\sigma') \equiv \forall\, b \in \text{Mbx} :$
Alle Postfächer müssen entweder in beiden Zuständen inaktiv sein, oder sie müssen in allen Attributwerten übereinstimmen: $\text{active.b.}\sigma \vee \text{active.b.}\sigma' \Rightarrow$ $\text{active.b.}\sigma = \text{active.b.}\sigma' \wedge \text{size.b.}\sigma = \text{size.b.}\sigma' \wedge$ $\text{mail.b.}\sigma = \text{mail.b.}\sigma' \wedge \text{wst.b.}\sigma = \text{wst.b.}\sigma' \wedge \text{wrt.b.}\sigma = \text{wrt.b.}\sigma'$

Als Beweisverpflichtung fordern wir, daß sich äquivalente erreichbare Zustände hinsichtlich der durch sie charakterisierten Postsystemfunktionen nicht unterscheiden:

$$\forall\, \sigma,\sigma' \in \text{State} : \text{Invar}(\sigma) \wedge \text{Invar}(\sigma') \wedge \text{Equiv}(\sigma,\sigma') \;\Rightarrow\; MB_\sigma = MB_{\sigma'}$$

Der Beweis dieser Eigenschaft kann mit Induktion über den Berechnungsablauf geführt werden. Man kann zusätzlich auch die andere Implikationsrichtung fordern (also, daß sich nicht-äquivalente Zustände auch beobachtbar unterscheiden); eine Eigenschaft, die man mit "voller Abstraktheit" bezeichnet.

Äquivalente Zustände können auch durch eine restriktivere Invariante "abgefangen" werden, so daß jede bisherige Äquivalenzklasse auf einen Repräsentanten reduziert wird. Im vorliegenden Beispiel kann man etwa fordern, daß nicht-aktive Postfächer die Größe 0 und leere Nachrichten- und Befehlswarteschlangen haben müssen. Die Notwendigkeit, solche Repräsentanten auszuzeichnen (und in der Spezifikation die entsprechende Attributbelegung sicherzustellen), ist aber methodisch abzulehnen.

4.4 Alternative Möglichkeiten der Konstruktion von Zustandsräumen

Die in Abschnitt 3.4 angewandte Methode, den Zustandsraum nicht-eindeutig durch Angabe von Bedingungen zu charakterisieren, läßt sich in zwei Richtungen abwandeln: es kann entweder die Definition "konkretisiert" (also explizit und eindeutig gemacht) werden, oder man kann die Technik der algebraischen Spezifikation einsetzen. Wir befassen uns zunächst mit der zweiten Möglichkeit.

Ein algebraischer Formalismus hat den Vorteil, daß die Spezifikations- und Implementierungsebenen klar voneinander getrennt sind. Dies verursacht aber einen gewissen Mehraufwand, sowohl, weil der verwendete Formalismus zuerst eingeführt werden muß als auch aus technischen Gründen. So ist beispielsweise statt der einfachen Mengeninklusion $\text{Rec} \subset \text{Cmd}$ ein Subsortenmechanismus notwendig. Zur Vereinfachung wird daher in der vorliegenden Arbeit die freiere, mathematisch orientierte Schreibweise bevorzugt.

Die erste oben erwähnte Möglichkeit, nämlich eine explizite (und eindeutige) Konstruktion des Zustandsraumes, sieht zum Beispiel wie in Abschnitt 4.3 oder, einfacher, folgendermaßen aus:

$$\begin{aligned}
\text{Attr} &= \{\text{active, size, mail, wst, wrt}\} \\
\text{Value} &= \mathbb{B} \cup \mathbb{N} \cup \text{Msg}^\omega \cup \text{Send}^\omega \cup \text{Rec}^\omega \\
\text{State} &= \text{Mbx} \to \text{Attr} \to \text{Value}
\end{aligned}$$

Die Selektion von Zustandskomponenten erfolgt dann durch Anwendung des Postfachbezeichners und des Attributnamens auf den Zustand; so bezeichnet σ.b.active den Wert der active-Komponente des Postfaches b. Zur punktweisen Änderung des Zustands dient die folgende Funktion:

$$\_ [\_ . \_ \rightarrow \_] : State \times Mbx \times Attr \times Value \rightarrow State$$

$$(\sigma[b.a \rightarrow v]).b'.a' = \textbf{if } b = b' \wedge a = a' \textbf{ then } v \textbf{ else } \sigma.b'.a'$$

Der Hauptnachteil dieser Definition ist, daß hier keine Zuordnung von Attributnamen und Wertetypen stattfindet; es braucht beispielsweise nicht unbedingt σ.b.active $\in \mathbb{B}$ gelten. Eine solche Zuordnung muß durch die Spezifikation sichergestellt werden. Analog zu der in Abschnitt 4.3 vorgestellten Invariantenmethode gibt es allerdings die Möglichkeit, den Zustandsraum durch ein Prädikat so einzuschränken, daß die erlaubten Werte eines Attributs typkorrekt sind. Als weitere Möglichkeit kann man ein hinreichend mächtiges Typsystem einführen, das über abhängige Typen die Definition von beliebigen Zustandsräumen ermöglicht. Auch äquivalente Zustände lassen sich wie in Abschnitt 4.3 methodisch berücksichtigen.

Die explizite Konstruktion des Zustandsraumes hat vor allem den Vorteil, daß sie dem Leser der Spezifikation eine konkrete Vorstellung vermittelt. Der Zustand der Spezifikation wird mit dem der Implementierung über Repräsentations- und Abstraktionsfunktionen verbunden, so daß die Bandbreite der möglichen Implementierungen nicht eingeschränkt ist. Im vorliegenden Fall wurde jedoch der axiomatischen Beschreibung der Vorzug gegeben, da sich hiermit ein die Attributtypen erhaltender Zustandsraum einfacher definieren läßt.

5. Abschließende Bemerkungen

Auch wenn das MMK-Postsystem nicht besonders umfangreich ist, kann dessen Spezifikation doch nicht mehr als Spielbeispiel aufgefaßt werden. In dem Bemühen, eine möglichst klare Beschreibung dieses Systems ohne notationellen Ballast zu erstellen, wurden manche Entscheidungen getroffen, die in größeren Spezifikationen anders ausfallen mögen. Trotzdem kann das hier vorgestellte Beispiel dem Anwender eine Grundlage für eigene Fallstudien geben. Dabei ist es besonders wichtig, nicht dogmatisch nach irgendwelchen festen Regeln vorzugehen, sondern flexibel diejenigen Mittel einzusetzen, die dem Einzelfall am besten gerecht werden.

Danksagung

Mein besonderer Dank gebührt Manfred Broy und Rainer Weber für vielfältige Hinweise, Anregungen und Diskussionen sowie für Kommentare zu Vorversionen dieses Textes. Die Arbeit [Weber 91] zum gleichen Thema hat meine Vorgehensweise erheblich beeinflußt. Unterstützt wurde ich von der Technischen Universität München im Rahmen des Sonderforschungsbereiches 342: Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen.

Literatur

[BBLT 90] Th. Bemmerl, A. Bode, Th. Ludwig, S. Tritscher: MMK – Multiprocessor Multitasking Kernel. SFB-Bericht 342/26/90 A, Technische Universität München, Dezember 1990.

[BD90] M. Broy, C. Dendorfer: Functional Modelling of Operating System Structures by Timed Higher Order Stream Processing Functions. SFB-Bericht 342/22/90 A, Technische Universität München, November 1990.

[Broy et al. 91] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. Gritzner, R. Weber: The Design of Distributed Systems. Vorversion 1991.

[Hall 90] Anthony Hall: Seven Myths of Formal Methods. IEEE Software, September 1990, S. 11–19.

[Jones 86] C. B. Jones: Systematic Software Development Using VDM. Prentice-Hall 1986.

[Lamport 83] Leslie Lamport: Specifying Concurrent Program Modules. ACM Transactions on Programming Languages and Systems, April 1983, S. 190–222.

[Weber 91] R. Weber: Echtzeit-Anforderungsspezifikation und ihre Anwendung auf die Beschreibung eines Postfachsystems. Vorversion 1991.

[Spivey 87] J. M. Spivey: The Z Notation - A Reference Manual. Prentice-Hall 1987.