Zeitschrift: Technische Mitteilungen / Schweizerische Post-, Telefon- und Telegrafenbetriebe = Bulletin technique / Entreprise des postes, téléphones et télégraphes suisses = Bollettino tecnico / Azienda delle poste, dei telefoni e dei telegrafi svizzeri

Herausgeber: Schweizerische Post-, Telefon- und Telegrafenbetriebe

Band: 56 (1978)

Heft: 1

Artikel: Zur formalen Beschreibung von Echtzeitsystemen

Autor: Vogel, Eberhard W.

DOI: https://doi.org/10.5169/seals-875191

Nutzungsbedingungen

Die ETH-Bibliothek ist die Anbieterin der digitalisierten Zeitschriften auf E-Periodica. Sie besitzt keine Urheberrechte an den Zeitschriften und ist nicht verantwortlich für deren Inhalte. Die Rechte liegen in der Regel bei den Herausgebern beziehungsweise den externen Rechteinhabern. Das Veröffentlichen von Bildern in Print- und Online-Publikationen sowie auf Social Media-Kanälen oder Webseiten ist nur mit vorheriger Genehmigung der Rechteinhaber erlaubt. Mehr erfahren

Conditions d'utilisation

L'ETH Library est le fournisseur des revues numérisées. Elle ne détient aucun droit d'auteur sur les revues et n'est pas responsable de leur contenu. En règle générale, les droits sont détenus par les éditeurs ou les détenteurs de droits externes. La reproduction d'images dans des publications imprimées ou en ligne ainsi que sur des canaux de médias sociaux ou des sites web n'est autorisée qu'avec l'accord préalable des détenteurs des droits. En savoir plus

Terms of use

The ETH Library is the provider of the digitised journals. It does not own any copyrights to the journals and is not responsible for their content. The rights usually lie with the publishers or the external rights holders. Publishing images in print and online publications, as well as on social media channels or websites, is only permitted with the prior consent of the rights holders. Find out more

Download PDF: 30.04.2026

ETH-Bibliothek Zürich, E-Periodica, https://www.e-periodica.ch

Zur formalen Beschreibung von Echtzeitsystemen

Eberhard W. VOGEL, Bern

621.395.345:681.3

Zusammenfassung. *In der Abteilung Forschung und Entwicklung der PTT-Betriebe wurde eine formale Sprache zur Beschreibung beziehungsweise Spezifikation von Echtzeitsystemen, wie prozessorgesteuerten Telefonzentralen, Datenendgeräten usw., entwickelt. Damit eröffnet sich die Möglichkeit, in Zukunft über eine derartige formale Beschreibung direkt den elektronischen Rechner heranzuziehen als Hilfe für Entwurf und Analyse, Simulation und Dokumentation. Die Sprache, die sich noch im Stadium der Erprobung befindet, wird in ihren Grundzügen beschrieben.*

Description formelle de systèmes en temps réel

Résumé. *Un langage formel servant à décrire et à spécifier les systèmes en temps réel tels que les centraux téléphoniques à commande par processeur, les équipements terminaux de données, etc., a été développé à la Division des recherches et du développement de l'Entreprise des PTT. Il sera désormais possible de recourir directement à l'ordinateur, par l'intermédiaire d'une telle description formelle, dans le domaine des projets et de l'analyse ainsi que pour des tâches de simulation et de documentation. L'auteur décrit les caractéristiques de ce langage, qui en est encore au stade expérimental.*

Descrizione formale dei sistemi in tempo reale

Riassunto. *Nella Divisione ricerche e sviluppo dell'Azienda delle PTT è stato creato un linguaggio formale per la descrizione, rispettivamente la specificazione di sistemi in tempo reale, come centrali telefoniche comandate mediante processori, impianti terminali, ecc. Sarà così possibile in futuro far capo direttamente ai calcolatori elettronici per i progetti, le analisi, le simulazioni e la documentazione. Il linguaggio, che al momento attuale è ancora in fase sperimentale, viene descritto nelle sue caratteristiche.*

1 Einleitung

Wenn im folgenden von *Echtzeitsystemen* die Rede ist, so ist damit beispielsweise eine rechnergesteuerte Telefonzentrale gemeint. Hier soll nicht versucht werden, den Begriff *System* exakt zu definieren; man kann sich darunter eine (meist komplexe) Vorrichtung zur Durchführung bestimmter Prozesse vorstellen, etwa eine elektronische Anlage mit einer Hardware- und einer Software-Komponente. Von einem Echtzeitsystem erwartet man, dass es zu vorbestimmten Zeiten bestimmte Aktionen durchführt, Signale sendet, Operationen startet usw. Es liegt in der Natur der Sache, dass das System dazu auch laufend Informationen von der Umwelt aufnehmen und registrieren muss. Schon die ersten automatischen Telefonzentralen kann man als Echtzeitsysteme ansehen, wenn es dabei auch hauptsächlich darauf ankam, alle Funktionen möglichst rasch auszuführen. Unsere heutigen Rechner arbeiten dagegen so schnell, dass man von ihnen eher einmal verlangen muss, mit einer bestimmten Aktion zu warten, um in der Zwischenzeit anderes zu erledigen. Weitgefasst ist der Begriff sinnvoll auch auf sehr einfache Vorrichtungen anwendbar, wie z. B. ein Flip-Flop, wenn die Zeit für die Funktion eine wichtige Rolle spielt. Eine Einschränkung muss allerdings gemacht werden: Es werden hier nur *diskrete Ereignisse* betrachtet. Ein Ereignis ist entweder eingetreten oder nicht; Zwischenzustände, Anstiegszeiten usw. werden ignoriert.

Grosse Systeme können leicht so komplex sein, dass sogar die Hersteller nicht genau wissen, wie sie in allen denkbaren Fällen reagieren, die im ordnungsgemässen Betrieb auftreten können. Noch weniger bekannt ist meist, wie sich das System bei einer Störung, beim Ausfall einer Komponente usw. verhält. Man versucht zwar, die Auswirkungen von Fehlern einzudämmen, indem man beispielsweise das System in getrennte Moduln mit genau definiertem Kommunikationsprotokoll oder durch die Bereitstellung von Neustart- und Rekonfigurationsprozeduren usw. einteilt. Bei Telefonzentralen bereinigen sich zudem geringere Störungen teilweise von selbst, indem der Teilnehmer bei seltsamen Reaktionen der Zentrale einfach den Hörer auflegt. Dennoch darf behauptet werden, dass man Theorie und Entwurf komplexer Echtzeitsysteme heute noch nicht in dem Masse beherrscht, wie es wünschenswert wäre.

Es liegt nahe, den Computer selbst wieder als Hilfe für Entwicklung und Analyse der Systeme heranzuziehen, aber dazu müssen die Systeme so beschrieben werden können, dass die Beschreibung zum Beispiel auf Lochkarten übertragen und vom Rechner «verstanden» (das heisst bezüglich Syntax und Semantik analysiert) werden kann; dies soll hier unter einer *formalen Beschreibung* verstanden werden. Es ist klar, dass eine solche Beschreibung nur dann von Nutzen ist, wenn sie leidlich vollständig ist, wenn sie also beispielsweise als Basis für eine Simulation der Systeme dienen kann.

Schaltungen können zwar schon lange gut dokumentiert werden und ebenso natürlich Rechnerprogramme, aber beides nur auf ganz bestimmter, niedriger Abstraktionsebene. Das Zusammenwirken dieser beiden Komponenten lässt sich erst dann formal beschreiben, wenn es für beide *eine* gemeinsame Sprache gibt. Dies ist auch für den Entwurf wichtig, weil man in vielen Fällen über die Art der Verwirklichung erst in einem fortgeschrittenen Stadium entscheiden will. Bereits die Darstellung einer elektronischen Schaltung durch Gates und Flip-Flops ist eine weitgehende Abstraktion, nämlich der wirksamen physikalischen Vorgänge. Konzepte wie *Schieberegister, Buffer, Speicher, Prozessoren,* ferner *Warteschlangen* usw., die durch das Zusammenwirken vieler elektronischer Grundelemente verwirklicht werden müssen, gehören einer noch höheren Abstraktionsebene an. Gerade die höheren Niveaus sind nötig, um die grossen Zusammenhänge verstehen zu können. Es ist daher eine wichtige Forderung für eine Methode der formalen Beschreibung, dass sie nicht nur für Einzelheiten auf den unteren Ebenen brauchbar ist, sondern auch für die höheren Abstraktionsebenen.

Es gibt bereits eine ganze Reihe von Darstellungsmethoden für die verschiedenen Aspekte bei Echtzeitsystemen, die aber alle entweder nicht genügend umfassend oder mehr oder weniger mit bestimmten Abstraktions-

ebenen und bestimmten Anwendungsbereichen verknüpft sind. Zu nennen sind hauptsächlich

- Schaltbilder mit Symbolen für Gates und Flip-Flops
- Programmiersprachen und die Programmdarstellung durch Flussdiagramme
- Zustandsdiagramme des endlichen Automaten [1]
- Diagramme zur Darstellung von Vermittlungsabläufen [2], [3], [4], [5]
- *Petri*-Netze und ähnliche Modelle für parallele Berechnungen (siehe zum Beispiel [6])
- Hardware-Beschreibungssprachen allgemeiner Art, wie CDL [7], [8] oder zur Darstellung von Blockstrukturen, Befehlsstrukturen, Schnittstellen usw. [9], [10]

Daneben gibt es eine Vielzahl von ganz besonders auf individuelle Bedürfnisse zugeschnittenen Methoden, wie sie notgedrungen und ohne grosse Ambitionen bei den Herstellern von HW-SW-Systemen eingeführt wurden. Für allgemeine Darstellungen von Systemen (einschliesslich Hardware) wurden auch schon Programmiersprachen herangezogen (zum Beispiel APL [11]), und die Anwendung von Simulationssprachen wäre denkbar. Da aber diese Sprachen im Grunde für andere Zwecke entwickelt wurden und aus technischen Gründen auf die sequentielle Arbeitsweise der Rechner abgestimmt sind, erhält man damit im allgemeinen nur umschreibende, in den Einzelheiten nicht getreue Darstellungen, die zwar die Schnittstelle zur Umwelt richtig wiedergeben, aber notwendigerweise Einzelheiten enthalten, die man eigentlich nicht darstellen will oder die mit der Realisierung nicht übereinstimmen.

In der Abteilung Forschung und Entwicklung PTT wurde der Versuch unternommen, eine Sprache speziell zur Beschreibung von Echtzeitsystemen zu entwickeln. Diese Sprache, SYM (für *Sy*stem *M*odell), soll im folgenden in ihren Grundzügen beschrieben werden; es kann sich hier freilich nur um eine Einführung handeln.

Es ist ein wesentliches Kennzeichen, dass SYM auf einem mathematischen Modell für eine gewisse Klasse von Echtzeitsystemen basiert, indem mit den Ausdrucksmitteln von SYM jedes System innerhalb des durch das Modell gegebenen Rahmens spezifiziert werden kann. Im Prinzip würde dazu natürlich die mathematische Symbolik ausreichen, sobald einmal das Modell vorliegt, aber SYM ist dieser besonderen Aufgabe besser angepasst, und damit verständlicher. Durch das Modell wird nicht nur die freie Wahl der Abstraktionsebene, sondern auch eine gewisse Vollständigkeit und die Widerspruchsfreiheit der Sprache sichergestellt. Insbesondere ist es möglich, durch das Einführen geeigneter Abkürzungen die Ausdruckskraft der Sprache zu erhöhen, ohne befürchten zu müssen, dass durch die Kombination solcher neuer Ausdrucksformen Unklarheiten und Widersprüche entstehen.

Das zugrundeliegende Modell basiert auf der Vorstellung von diskreten Ereignissen und einer diskreten Zeit (das heisst alle Zeitangaben sind als Vielfaches einer gegebenen Zeiteinheit zu machen); es ist in [12] skizziert und soll hier nicht weiter betrachtet werden. Eine vollständige Definition von Modell und Sprache (mit gewissen Abweichungen gegenüber der hier gegebenen Einführung) liegt in [13] vor. Diese Darstellung ist einigermassen umfangreich und zum Teil ziemlich abstrakt. Die Sprache ist noch im Zustand der Erprobung, und die Erfahrungen bei der Anwendung werden voraussichtlich Änderungen nahelegen.

Es muss betont werden, dass das bei der Entwicklung der Sprache verfolgte Ziel vorerst die formale Beschreibung der Systeme war, besonders, um komplexe Systeme in Pflichtenheften usw. in eindeutiger Form spezifizieren zu können. Da die Sprache auf einem allgemeinen Modell basiert, wird bei einer solchen Beschreibung normalerweise offengelassen, wie weit das spezifizierte System später durch Verwendung von Schaltelementen (Hardware) oder alternativ durch Programmanweisungen (Software) verwirklicht werden soll. Zudem zeigen bereits die vorliegenden Erfahrungen bei der Anwendung von SYM auf die funktionelle Spezifikation des nationalen Autotelefonsystems NATEL, dass der Formalismus der Sprache zu einer präzisen Darstellung zwingt, in der alle Lücken offensichtlich werden.

Weitere Entwicklungsziele, die den Nutzen der Sprache erheblich steigern würden, sind

- automatische Überprüfung der Syntax einer Spezifikation und Plausibilitätskontrollen bezüglich der Semantik,
- Implementierung für tiefergehende Analysen und Simulation der Systeme auf der Basis einer Beschreibung in SYM und
- Programme zur automatischen Erzeugung von Zustandsdiagrammen und ähnlichem, um die Funktionen des spezifizierten Systems zu verdeutlichen.

Es sei besonders auf die Bedeutung dieser geplanten Implementation für eine zuverlässige und lückenlose Dokumentation komplexer Systeme hingewiesen: Bei Änderungen müssen lediglich einige Lochkarten der Beschreibung ausgewechselt werden, alles übrige lässt sich automatisch erledigen. Ein Fernziel besteht schliesslich in der automatisierten Generierung von Software und Hardware.

2 Grundelemente der Beschreibungssprache SYM

21 Speicher und Prozessoren

Wenn man allgemein für Echtzeitsysteme ein mathematisches Modell aufstellen will, so muss man sich zunächst über die Grundelemente klarwerden, aus denen alles aufgebaut werden soll. In [10] kommen die Autoren auf insgesamt sieben verschiedene Typen, die sie wie folgt benennen: *Processors, Memories, Links, Switches, Controls, Data Operators, Transducers.* Grundsätzlich sind jedoch zwei Grundtypen ausreichend, deren Funktion am besten durch die Bezeichnungen *Speicher* und *Prozessor* gekennzeichnet wird. Beim endlichen Automaten[1] beispielsweise verkörpert der Speicher die Gedächtniseigenschaft und der Prozessor die Transformationseigenschaft. Bei der Anwendung von SYM muss alles in die Sprache der Speicher und Prozessoren übersetzt werden. Beispiele für Speicher sind: Elektronische Speicher, Register, Schalter (speichern den Wert *ein* beziehungsweise *aus*), Leitungen (speichern ein Potential) in der Hardware; man kann auch von den Leitungen abstrahieren und einen Puls oder ein Signal als eine

[1] Als Echtzeitsystem betrachtet; es wird angenommen, dass jeder Schritt eine gewisse Zeit erfordert

Wertfolge in einem besonderen Speicher ansehen, den man dann zweckmässig mit dem Namen des Pulses oder Signals bezeichnet. In der Software: Variablen, Indikatoren.

Prozessoren verkörpern die aktiven Elemente. In der Hardware: Prozessoren, Addierwerke usw.; in der Software: Subroutinen, Prozeduren.

Die Art der Darstellung hängt natürlich von der gewählten Abstraktionsebene ab. Bei den Speichern kann man je nach ihrer Bedeutung für das System zunächst rein informal verschiedene Typen unterscheiden; neben dem die Gedächtnisfunktion betonenden Haupttyp *Register* beispielsweise

- *Nachrichten,* die bei den Prozessoren Reaktionen auslösen;
- *Indikatoren* als Repräsentanten von Teilzuständen, die die Art der Reaktionen der Prozessoren bestimmen;
- *Schalter* zur Verkörperung von komplexen Aktionen (wie «sende Summton»), die in der Beschreibung nicht näher erklärt werden sollen; wenn der Schalter auf «ein» steht, bedeutet dies, dass die Aktion läuft usw.

22 Darstellung der Grundelemente in SYM

Die Speicher werden in SYM durch *Parameter* verkörpert. Ein Parameter kann zu verschiedenen Zeiten verschiedene Werte annehmen, besonders hat er stets einen «gegenwärtigen Wert». Ein Parameter ist also einmal durch seinen besonderen Namen gekennzeichnet und dann durch die Menge der Werte, die er annehmen kann. Alle Parameter müssen deklariert werden. In SYM werden alle Werte als ganze Zahlen angesehen, auch wenn sie in der Deklaration andere Bezeichnungen erhalten haben; die Zuordnung hängt dann von der Reihenfolge in der Deklaration ab, indem dem ersten Wert die Zahl 0 zugeordnet wird, dem zweiten die Zahl 1 usw. Jeder Parameter kann überdies einen speziellen Wert, bezeichnet mit ∂, annehmen, der Unbestimmtheit oder Nichtexistenz symbolisiert. Dies ist Teil des Modellmechanismus, der ein effektives Simulieren von Systemen mit unendlich vielen Parametern und Regeln gestattet: Ein Parameter mit dem Wert ∂ gilt sozusagen als nur latent vorhanden. Nur *endlich* viele Parameter dürfen gleichzeitig einen Wert ungleich ∂ haben.

Die Namen der Parameter und ihrer Werte können frei gewählt werden, mit Vorteil natürlich so, dass ihre Bedeutung zum Ausdruck kommt. Die Namen können mit Hilfe des Zeichens «–» gegliedert werden.

Beispiele:

Parametername	Wertmenge
a	{∂, 0, 1, 2, ...}
leitung	{∂, low, high} oder {∂, 0, 1}
teilnehmer-frei	{∂, falsch, wahr} oder {∂, 0, 1}
zustand	{∂, ruhe, warten, in-betrieb} oder {∂, 0, 1, 2}

Die Prozessoren verkörpern die aktiven Elemente des Systems. Unter bestimmten Bedingungen, wie Wertgleichheit zweier Parameter oder Erscheinen eines Pulses, treten sie in Aktion und ordnen einzelnen Parametern neue Werte zu. Dies ist ein wesentlicher Punkt von Modell und Sprache, dass ein Übergang zu bestimmten Werten zu bestimmten Zeiten vorgemerkt wird, also gewissermassen latente Wertänderungen, die in vorbestimmten Zeitabständen (zum Beispiel sofort) automatisch erfolgen, falls sie nicht vorher durch andere Reaktionen aufgehoben worden sind. Ein typisches Beispiel ist das Senden eines Pulses mit einer bestimmten Verzögerung.

Prozessoren werden in SYM durch eine oder mehrere *Regeln* dargestellt. Eine Regel besteht gewöhnlich aus einem *Bedingungsteil* und einem *Reaktionsteil,* getrennt durch einen Pfeil →, wobei der Bedingungsteil auch fehlen kann. *Beispiele* für *Bedingungen* sind:

zustand = ruhe
zähler ≠ 0
anzahl ≥ (bedarf + 1)
(zeichenempfang ∧ ¬ alarm) = wahr

In den ersten beiden Beispielen steht links ein Parameter und rechts ein Wert. Im dritten Beispiel steht auf der rechten Seite eine Funktion, die einen Parameter als Argument enthält; in Bedingungen und Reaktionen verkörpern die Parameter stets ihren gegenwärtigen Wert. Im letzten Beispiel haben wir die Form

$$\varphi(a, b) = n,$$

wo φ eine Funktion[2] bedeutet, a und b Parameter und n einen Wert. Für die häufig auftretende Bedingung

$$\varphi\ (a, b, ...) = 1$$

gibt es in SYM die Abkürzung

$$\varphi\ (a, b, ...),$$

so dass wir die letzte Bedingung aus unseren Beispielen auch einfach als

(zeichenempfang ∧ ¬ alarm)

schreiben können, falls *falsch* mit 0 identifiziert wird und *wahr* mit 1.

Beispiele für *Reaktionen:*

indikator ← 1
zähler ← (zähler + 1)
signal ← 1/5:0
a ← 7:b
zeitüberwachung ← 0/d:1/1:0

Im ersten Beispiel wird der Parameter *indikator* unmittelbar auf 1 gesetzt, und im zweiten Beispiel *zähler* um 1 erhöht. Im dritten Beispiel erhält *signal* den Wert 1 und mit einer Verzögerung von 5 Zeiteinheiten den Wert 0. Im nächsten Beispiel übernimmt a mit einer Verzögerung von sieben Zeiteinheiten den Wert von b. Im letzten Beispiel schliesslich wird ein Puls der Länge 1 erzeugt, und zwar mit einer Verzögerung, die dem gegenwärtigen Wert des Parameters d entspricht. Es gilt die Regel, dass alle ursprünglich angesetzten Wertänderungen eines Parameters durch eine Reaktion aufgehoben werden, soweit sich die Werte widersprechen. Das bedeutet, dass im vierten Beispiel a alle Änderungen von b mitmacht, wenn auch verzögert, während durch die letzte Reaktion zum Beispiel ein ursprünglich für eine

[2] In diesem Falle eine Boole'sche Funktion; die logischen Operationen *und, und/oder* und *nicht* werden hier durch ∧, ∨ und ¬ bezeichnet

spätere Zeit vorgesehener Puls aufgehoben wird. Terme der Art

Verzögerung: Neuer Wert

können – getrennt durch den Schrägstrich – beliebig aneinandergehängt werden.
Beispiele für *Regeln:*

signal, zustand = ruhe → summton ←1,
zeitüberwachung ← 0/d:1, zustand ← wahl;

→ a ←¬(p ∧ q);

x = 1, a ≥ 0 → b ← (b + 1), c ← b, x ← 0

Die erste Regel bedeutet (frei übersetzt): «Wenn im Zustand *Ruhe* das Signal erscheint, so schalte den Summton ein, stelle die Zeitüberwachung auf die Zeit d ein und gehe in den Zustand *Wahl* über». Diese Regel wird nur einmal angewendet, da wiederholte Anwendung nichts ändern würde. Als zweites Beispiel haben wir eine *bedingunslose* Regel; der Wert des Parameters a wird permanent durch die Boole'sche Funktion ¬ (p ∧ q) bestimmt. Auch diese Regel wird nach jeder Änderung von p oder q höchstens einmal angewendet. Durch die dritte Regel wird im Falle x = 1 und a ≥ 0 der Wert des Parameters b um 1 erhöht, und c erhält den ursprünglichen Wert von b, wie er vor Anwendung der Regel war; bei dieser Regel wird nur durch die Reaktion x ← 0 die wiederholte Anwendung verhindert.

Ein System wird durch eine Liste von Parametern zusammen mit einer Liste von Regeln beschrieben. Die genaue Interpretation einer solchen Beschreibung ist durch einen – im Modell *Systemfunktion* genannten – *Simulationsalgorithmus* gegeben:

- Falls in der gegenwärtigen Situation eine oder mehrere Regeln anwendbar sind, wird eine davon zufällig ausgewählt und angewendet.

Eine *Situation* wird dabei durch die gegenwärtigen und vorgemerkten (latenten) Werte aller Parameter charakterisiert, und eine Regel gilt als *anwendbar,* falls zum gegenwärtigen Zeitpunkt ihre Bedingungen erfüllt sind *und* ihre Anwendung (das heisst das Ausführen der Reaktionen) die Situation ändert.

- Falls keine Regel anwendbar ist, geht man zum nächsten Zeitpunkt über, wo mindestens einer der Parameter seinen Wert ändert; dieser Zeitpunkt wird dann neue «Gegenwart». Falls es einen solchen Punkt nicht gibt, muss der Algorithmus abgebrochen werden.

Je nach der Ausgangssituation ergeben sich im allgemeinen ganz verschiedene (sinnvolle oder sinnlose) Simulationsabläufe, so dass zur Definition eines Systems auch die Spezifikation der Ausgangssituation gehört.

Wenn mehrere Regeln anwendbar sind, kann nur eine nach der anderen angewendet werden, aber gleichwohl zum «gleichen Zeitpunkt» im Sinne der Simulation. Die Reihenfolge der Anwendung bestimmt eine Feinstruktur der Ereignisfolge, die vom Zufall abhängig ist. In vielen Anwendungen gibt es zeitliche Abstände, die um Grössenordnungen kleiner sind als die hauptsächlich betrachteten. Diese Abstände sind häufig gar nicht bekannt, oder man will sie nicht betrachten. Wenn man ihnen den Wert *Null* zuordnet, ergibt das Modell dennoch ein realistisches Bild.

Die Deklaration der Parameter und die Spezifikation der Ausgangssituation können hier nicht behandelt werden. In den folgenden Beispielen werden daher nur die Regeln wiedergegeben; die Parameter und ihre Werte lassen sich meist indirekt daraus entnehmen.
Beispiel 1. Ein *Nor-Gate* mit verzögerter Reaktion lässt sich durch eine bedingungslose Regel der Art

→ a ← 6: ¬ (p ∨ q)

beschreiben, wobei die Parameter p und q die ankommenden Leitungen darstellen und a die abgehende. Die hier angenommene Verzögerung beträgt sechs Zeiteinheiten.
Beispiel 2. Erzeugung von *Takt-Pulsen:*

takt = 0 → takt ← a:1
takt = 1 → takt ← d:0

Der Wert des Parameters d bestimmt die Pulslänge, ihr Abstand berechnet sich durch die Summe von a und d.
Beispiel 3. Endlicher Automat:
Durch die Regel

zustand = z1, eingabe = a
→ zustand ←1:z2, ausgabe ←1:x

lässt sich ausdrücken, dass der Automat bei der Eingabe a im Zustand z1 in den Zustand z2 übergeht und das Zeichen x ausgibt. Es wurde dabei eine Verzögerung der Reaktionen von einer Zeiteinheit eingeführt, um die Anzahl der Schritte, die der Automat ausführt, mit der Zeit identifizieren zu können. Der ganze Automat lässt sich allein mit derartigen Regeln beschreiben, für die ein *Makro* (siehe 6) einzuführen sich lohnen würde. Damit könnte man beispielsweise die obige Regel symbolisieren durch

<zl a z2 x>

usw.

Der endliche Automat stellt ein Beispiel für ein *nicht autonomes* System dar. Ein derartiges System ist im Prinzip als ein Teilsystem anzusehen, das für eine Simulation durch ein weiteres Teilsystem ergänzt werden muss, das die Umwelt repräsentiert und das in unserem Falle den Wert des Parameters *eingabe* periodisch ändern würde. In diesem Beispiel liesse sich letzteres allerdings auch durch eine geeignete Spezifikation der Ausgangssituation erreichen. Es wäre ferner möglich, mehrere solche (numerierte) Automaten durch Regeln der Art

→ eingabe-2 ← ausgabe-1

usw. zu verkoppeln.
Beispiel 4. Petri-Netze (siehe zum Beispiel [6]):
Es sei angenommen, dass jede *Stelle* beliebig viele Marken *(Token)* speichern kann. Das Netz in *Figur 1* wird durch folgende Regeln vollständig beschrieben (wobei jede eine *Transition* verkörpert):

ω, a ≥ 1 → a ← (a − 1), b ← (b + 1), c ← (c + 1), ω ← 0/1:1
ω, c ≥ 1, d ≥ 1 → c ← (c − 1), d ← (d − 1), a ← (a + 1), ω ← 0/1:1
ω, a ≥ 1, b ≥ 1 → b ← (b − 1), d ← (d + 1), ω ← 0/1:1

Auch hier würde sich bei grösseren Netzen die Einführung eines Makros lohnen, das wegen der variablen Anzahl der beteiligten Parameter rekursiv definiert werden

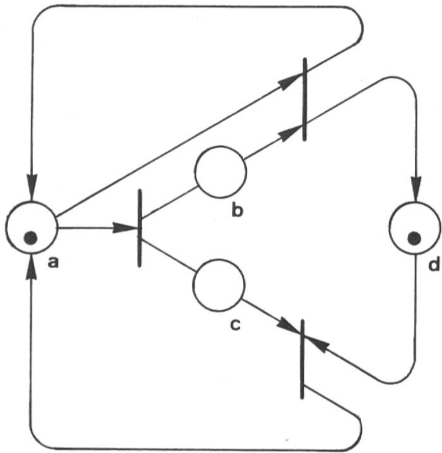


Fig. 1
Beispiel für ein Petri-Netz. Die «Stellen» werden in der SYM-Beschreibung durch die Parameter a, b, c und d dargestellt, die «Transitionen» durch Regeln

müsste. Die Stellen werden durch Parameter wiedergegeben, deren Wert die Anzahl der gespeicherten Marken angibt. Eine Transition kann nur stattfinden («feuern»), wenn alle vorangehenden Stellen mindestens eine Marke enthalten. ω ist ein Hilfsparameter, der dafür sorgt, dass zu jedem Zeitpunkt nur ein Übergang erfolgt. Auf dieser letzteren Forderung würde man in praktischen Anwendungen natürlich nicht bestehen, sondern statt dessen für jede Transition eine realistische Verzögerung (oder auch die Verzögerung 0) spezifizieren, womit der Hilfsparameter überflüssig würde. Dieses Beispiel soll Ähnlichkeiten und Unterschiede zwischen einer Darstellung durch SYM und einer Darstellung durch Petri-Netze deutlich machen.

3 Weiterer Ausbau der Sprache

Da SYM auf einem mathematischen Modell basiert, ist es möglich, Abkürzungen für komplexere Strukturen genau zu definieren und auf diese Weise die Ausdruckskraft der Sprache zu erhöhen, ohne dass die Gefahr besteht, dass die Kombination verschiedener Ausdrucksmittel undefiniert ist oder zu Widersprüchen führt.

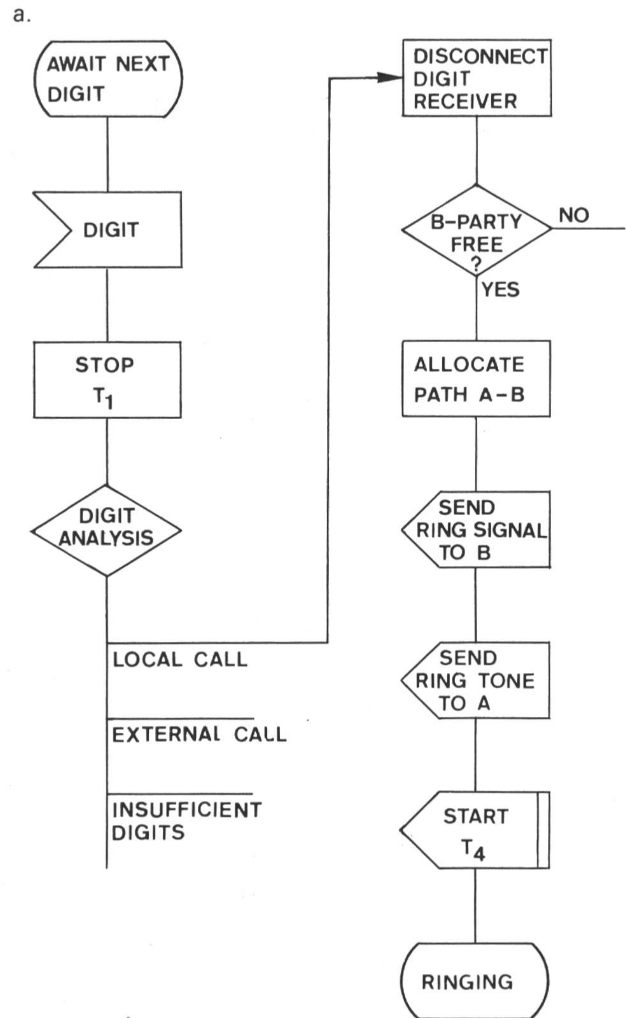
Zunächst seien zwei einfache Abkürzungen betrachtet: Durch Nachrichten oder Signale ausgelöste Aktionen sollen normalerweise nur einmal durchgeführt werden; daher muss in die entsprechenden Regeln eine geeignete Selbstblockierung eingebaut werden. Die einfachste Möglichkeit besteht darin, die auslösende Nachricht zu vernichten, wie in der folgenden Abkürzung:

$$\mathbf{m}\ \text{digit}::Bd \rightarrow Re := \text{digit} \neq \partial, Bd \rightarrow Re,\ \text{digit} \leftarrow \partial$$

Dabei bedeutet *Bd* eine Folge von Bedingungen und *Re* eine Folge von Reaktionen. Diese Lösung ist nur möglich, wenn sich die übermittelte Information ausschliesslich an einen einzigen Empfänger richtet. Für einen Taktpuls, der mehrere unabhängige Aktionen auslösen soll, kann folgende Abkürzung gebraucht werden:

$$\begin{aligned} &\mathbf{v}\ \text{takt} :: Bd \rightarrow Re := \\ &\text{takt}, \omega = 0,\ Bd \rightarrow Re,\ \omega \leftarrow 1; \\ &\text{takt} = 0 \quad \rightarrow \quad \omega \leftarrow 0 \end{aligned}$$

In der ersten Regel wird ein Hilfsparameter ω benutzt, um die Regel nach einmaliger Anwendung zu sperren. Durch die zweite wird diese Sperre wieder aufgehoben, sobald der Parameter «takt» zum Wert 0 zurückkehrt. Der Hilfsparameter ω ist individuell verschieden für jede Anwendung dieser Abkürzung zu denken. Die erste Abkürzung wird in *Beispiel 5* benützt, das die Übersetzung eines SDL-Diagramms [5] in SYM illustriert. *Figur 2a* zeigt (leicht modifiziert) einen Ausschnitt des Vermittlungsvorganges für lokale Gespräche aus dem Anhang zu den Empfehlungen Z.101—104 und *Figur 2b* die Übersetzung[3]. Es muss dazu bemerkt werden, dass SDL eine



b.

```
m digit: :
state = avait_digit          → timer ← reset
                               state ← digit_analysis ;

state = digit_analysis,
local_call,
b_party = free               → digit_receiver ← off,
                               b_party ← occupied,
                               < allocate path a b >,
                               ring_signal_b ← on,
                               ring_tone_a ← on,
                               timer ← t4: set,
                               state ← ringing ;
```

Fig. 2
Teil eines Vermittlungsvorganges
a) Beschrieben in SDL
b) Beschrieben in SYM

[3] Um den Vergleich zu erleichtern, wurden die englischen Ausdrücke beibehalten

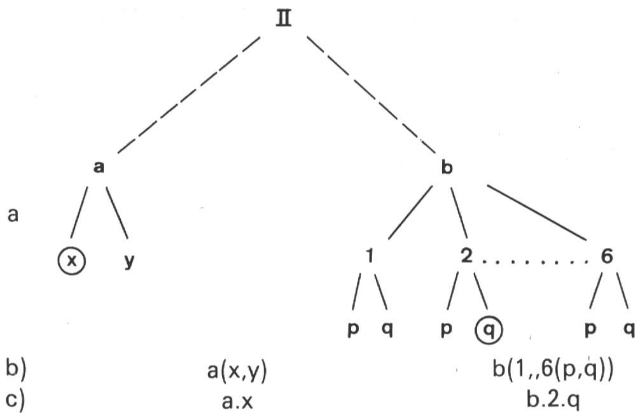


Fig. 3
Beispiel für Parameterstrukturen

a) Baumstruktur; a und b sind Parameternamen, 1...6 sowie x, y, p und q Bezeichnungen für die Komponenten
b) Darstellung in SYM
c) Bezeichnung der eingekreisten Komponenten

nur halbformale Methode ist, weil der Inhalt der Kästchen bisher nicht formalisiert wurde, und dass schon aus diesem Grunde die Interpretation der Diagramme nicht genau festgelegt ist. Ein gegebenes SDL-Diagramm lässt sich daher auf verschiedene Arten übersetzen. In unserem Falle wurde angenommen, dass die Reihenfolge der Operationen im allgemeinen nicht spezifiziert werden soll.

Die Entscheidung *digit-analysis* wurde durch einen Zwischenzustand symbolisiert (der verlassen wird, sobald der Indikator *local-call* gesetzt ist), die Entscheidung *b-party-free* durch einen Indikator. *ring-tone-a* kann man sich als einen Schalter vorstellen, der ein- oder ausgeschaltet wird, usw. <**allocate path** a b> ist ein zunächst nicht definiertes Makro (siehe nachstehend), das für eine einfachste Simulation nicht benötigt wird. Im übrigen müssten die Regeln zum Simulieren noch ergänzt werden, beispielsweise damit der Indikator *local-call* wirklich gesetzt wird usw.

Andere Übersetzungen sind möglich, insbesondere unter weiterem Gebrauch von Makros und unter Verwendung der unten beschriebenen Moduln, im besonderen der Sequenzen zur Darstellung einer bestimmten Reihenfolge der Aktionen.

4 Parameterstrukturen

Grundsätzlich kann alles durch eine genügend grosse Zahl von Parametern ausgedrückt werden. Für eine kompakte und klare Beschreibung ist es jedoch wünschenswert, die Menge der Parameter beispielsweise in ein- und mehrdimensionale Felder *(Arrays)* usw. gliedern zu können. Als Basis für diese Gliederung dient in SYM die Baumstruktur der Graphentheorie. Alle Knoten erhalten Namen; die Endknoten repräsentieren die Parameter im bisher gebrauchten Sinne des Wortes. Im folgenden wird jedoch jede Teilstruktur als ein *Parameter im weiteren Sinne* angesehen. *Figur 3* zeigt a) zwei solche Parameterstrukturen, b) ihre Bezeichnung in SYM und c) die Bezeichnung für die eingekreisten Komponenten. Man kann noch einen weiteren Schritt tun und alle Parameter als Teile eines einzigen *Generalparameters* Π auffassen, wie in Figur 3 angedeutet. Ein Parameter darf unendlich viele Komponenten haben, die durch 1,. := 1,2,3,... (analog «0,.» usw.) symbolisiert werden.

In Zusammenhang mit strukturierten Parametern sind viele Abkürzungen möglich. Die Parameter der Figur 3 vorausgesetzt, symbolisiert beispielsweise

$$a = b.2$$

die Gesamtheit der Bedingungen

$$a.x = b.2.p,\ a.y = b.2.q$$

Analog b.5 ← a usw. Parameter (im weiteren Sinne) können durch das Symbol & verkettet werden. So stellt a&b die Struktur

$$a\&b(a(x,y),b(1,,6(p,q)))$$

dar. Was bei u = v&w verglichen wird, ist allein die Folge der Endknotenwerte (das heisst der gegenwärtigen Werte der Parameter im engeren Sinne), nicht die Struktur. Analog für r ← s&t.

Wichtig ist besonders, dass Parameternamen zur Bezeichnung von Komponenten verwendet werden können. So hat zum Beispiel

$$u.(v)$$

die Bedeutung «u.3», falls 3 der gegenwärtige Wert des Parameters v ist. v dient gewissermassen als Pointer für die «Tabelle» u. Mehrfache Verweise sind möglich:

$$x \leftarrow s.(t.(u)).(v)$$

usw., siehe *Figur 4*.

Im folgenden Beispiel stellt die *Komponentenvariable* **i** eine beliebige (aber jedenfalls deklarierte) Komponente dar:

$$a.\mathbf{i} = 0 \rightarrow x \leftarrow \mathbf{i}$$

Definiert ist dies durch die Gesamtheit der Regeln

$$a.1 = 0 \rightarrow x \leftarrow 1;$$
$$a.2 = 0 \rightarrow x \leftarrow 2;$$
— — — — —

(mit allen *sinnvollen* Ersetzungen von **i**; hier wurde ein Parameter a(1,.) als deklariert angenommen).

5 Regelstrukturen

Um Prioritäten, gegenseitiges Blockieren von Regeln usw. darstellen zu können, wurden in SYM gewisse Re-

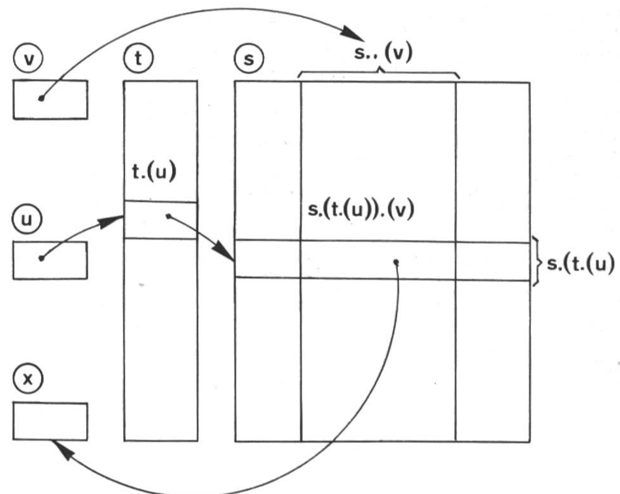


Fig. 4
Verweisketten und ihre Darstellung in SYM; s, t, u, v und x sind Parameternamen

gelstrukturen eingeführt, die allgemein *Moduln* genannt werden. Eine Regel ist die einfachste Form eines Moduls. Wenn wir Bedingungsfolgen mit

Bd beziehungsweise Bd_1, Bd_2 usw.

und Moduln mit

M beziehungsweise M_1, M_2 usw.

bezeichnen, lassen sich in SYM unter anderem folgende Moduln unterscheiden:

$$[M_1;\ M_2;\ ...]$$

ist eine einfache Gruppierung, eine Zusammenfassung endlich oder unendlich vieler Moduln zu einem einzigen Modul. Mit

$$[(1)\ M_1;\ (2)\ M_2;\ ...]$$

wird eine Gruppe mit *Prioritäten* bezeichnet: M_1 weist die höchste Priorität auf, dann folgt M_2 usw. Falls die M_i Regeln sind, bedeutet dies, dass M_2 nur dann angewendet werden darf, wenn M_1 nicht anwendbar ist, usw. In

$$[(a_1)\ M_1;\ (a_2)\ M_2;\ ...]$$

bestimmt der gegenwärtige Wert des Parameters a_i die Priorität von M_i (variable Prioritäten). Ein *Block*

$$Bd \rightarrow [M_1;\ M_2;\ ...;\ \underbrace{Bd_i \rightarrow \mathbf{exit};}_{M_i}\ ...;\ \underbrace{Bd_j \rightarrow \mathbf{exit};}_{M_j}\ ...]$$

verkörpert eine komplexere Operation, die durch die *Eintrittsbedingung Bd* ausgelöst wird. Die inneren Moduln M_1, M_2 usw. sind nur zugänglich, wenn sie durch Anwendung einer speziellen Eintrittsregel mit den Bedingungen Bd freigegeben wurden (gemäss der genauen Definition des Blockes); gleichzeitig blockiert sich die Regel dadurch selbst. Wenn dann eine der Regel M_i oder M_j zur Anwendung kommt, wird der Block wieder verlassen, das heisst durch die symbolische Reaktion **exit** werden die inneren Moduln gesperrt, während die Eintrittsregel erneut freigegeben wird. — Ihre eigentliche Bedeutung haben die Blöcke nur in Verbindung mit *starken Prioritäten.* Ein Beispiel dafür ist der Modul:

$$[1 \# B_1;\ 2 \# [B_{21};\ B_{22}];\ 3 \# B_3;\ 4 \# B_4]$$

Hier bedeuten die B_k und B_{mn} Blöcke. Für starke Prioritäten gilt: Sobald ein Block betreten wird, werden alle Moduln gleicher oder niedrigerer Priorität gesperrt[4]. Das bedeutet praktisch, dass man sich niemals gleichzeitig in zwei verschiedenen Blöcken gleicher Priorität befinden kann und dass mit dem Betreten eines Blockes alle Operationen der Blöcke niederer Priorität unterbrochen werden (Unterbruchsystem). In unserem Beispiel kann der Block B_{22} den Block B_{21} sperren und die Blöcke B_3 und B_4 unterbrechen, während B_{22} von B_{21} gesperrt und von B_1 unterbrochen werden kann.

Als ein solcher Block lässt sich wiederum eine *Sequenz* definieren (wie hier nicht ausgeführt); Schreibweise:

$$Bd \Rightarrow [M_1; M_2;\ ...;\ M_n]$$

[4] Einzelne Moduln können durch Voranstellen des Symbols * davon ausgenommen werden, was zum Beispiel bei den letzten beiden Regeln in der Definition von 'v takt ::' angebracht wäre.

Tabelle I. Moduln in SYM

Typ	Form
Regel	$Bd \rightarrow Re$
Gruppe ohne Prioritäten	$[M_1;\ M_2;\ ...]$
Gruppe mit schwachen Prioritäten (fest)	$[(1)\ M_1;\ (2)\ M_2;\ ...]$
Gruppe mit schwachen Prioritäten (variabel)	$[(a_1)\ M_1;\ (a_2)\ M_2;\ ...]$
Gruppe mit starken Prioritäten	$[1 \# M_1;\ 2 \# M_2;\ ...]$
Block	$Bd \rightarrow [M_1;\ M_2;\ ...;\ \underbrace{\widetilde{Bd} \leftarrow \mathbf{exit};}_{M_i}\ ...]$
Sequenz	$Bd \Rightarrow [M_1;\ M_2;\ ...;\ M_n]$

Eine Sequenz symbolisiert eine bestimmte Reihenfolge der Operationen. Falls bei erfüllten Bedingungen Bd der Block betreten wird, wird zuerst versucht, M_1 anzuwenden, wenn dies nicht möglich ist, M_2 usw. Jeder Modul kann höchstens einmal angewendet werden. Wenn beispielsweise die M_i bedingungslose Regeln verkörpern, werden die entsprechenden Reaktionen in der gegebenen Reihenfolge je einmal ausgeführt. Falls eines der M_i ein Block ist, wird die Sequenz erst nach Verlassen dieses Blockes weiterverfolgt. Es bestehen — hier nicht beschriebene — Möglichkeiten zum Bilden von Schleifen oder zum vorzeitigen Verlassen der Sequenz.

In *Tabelle I* wurden die hier eingeführten Moduln nochmals zusammengestellt.

SYM lässt sich als eine Art Programmiersprache betrachten, indem man die Moduln mit den Anweisungen (Statements) der Programmiersprachen gleichsetzt. Aber während bei den gebräuchlichen Programmiersprachen der sequentielle Ablauf das Normale ist und Parallelität nur ausnahmsweise (zum Beispiel durch *Fork* und *Join*) eingeführt werden kann, ist bei SYM unbeschränkte Parallelität der Normalfall, und sequentielle Aktionen müssen mit Hilfe der «Sequenzen» dargestellt werden.

6 Makros

Makros sind vom Benützer definierte Abkürzungen für Folgen von Bedingungen, Reaktionen oder Moduln, die in SYM speziell drei besonderen Zwecken dienen:

a) Als Abkürzung für häufig auftretende Formen, die sich nur durch die eingesetzten Parameter oder Werte unterscheiden; siehe die Beispiele 3 und 4.
b) Zur Erweiterung der Sprache: SYM wurde absichtlich sehr allgemein gehalten. In besonderen Anwendungsbereichen werden spezielle Konzepte und Konstruktionen gebraucht, für die es in SYM keine besondere Symbolik gibt; Beispiele: RS-Flip-Flop, Warteschlange, Case-Statement, Pushdown-Speicher. SYM gestattet es, eine Realisierung eines solchen Konzeptes durch Makros explizit als *Konzept* zu deklarieren, wobei dann nur die Art der Reaktion als definiert gilt, nicht die spezielle Realisierung.
c) Für den sogenannten *Topdown-Entwurf:* Zur Beschreibung eines komplexeren Systems kann man zunächst eine Struktur angeben, die im wesentlichen nur aus geeignet benannten Makros als Repräsentanten von Teilsystemen besteht; diese Teilsysteme werden später definiert, im allgemeinen wieder mit Hilfe von Makros, usw.

Für die Definition der Makros sei hier nur ein Beispiel angeführt:

```
% <verzögerung $N> ::=
            → [¢ ← $N:1; ¢=1→ exit,¢ ← 0] ;;
```

Links von '::=' ist die Form des Makros spezifiziert; dabei ist $N eine *Makrovariable,* für die bei der Anwendung eine natürliche Zahl ≥1 einzusetzen ist (oder ein Parameter mit einer entsprechenden Wertemenge):

```
<verzögerung150> .
```

Makrovariable gibt es auch für Parameter, Moduln usw. Das hervorgehobene Wort **verzögerung** dient der Identifizierung des Makros (bei einer Implementierung der Sprache können derartige Worte je nach den verfügbaren Möglichkeiten unterstrichen oder auf andere Art gekennzeichnet werden). Makros sind in einer Beschreibung stets durch die Klammern '<' und '>' gekennzeichnet. — Rechts steht die Bedeutung des Makros, das, was bei einer Realisierung bzw. Kompilierung für das Makro einzusetzen ist. In unserem Falle ist dies ein bedingungsloser Block, der, vom Zeitpunkt des Eintretens an gerechnet, nach $N Zeiteinheiten wieder verlassen wird. ¢ repräsentiert einen Hilfsparameter, der in der Abkürzung nicht in Erscheinung treten soll. Als Element einer Sequenz verzögert dieses Makro den Ablauf und kann somit die in der Realität auftretende Bearbeitungszeit symbolisieren.

7 Weitere Beispiele für die Anwendung der Sprache

Beispiel 6. Definition eines *Parallel-Addierers:*

Das folgende Makro ist eine Abkürzung für einen Modul, der zwei Binärzahlen a und b addiert, wobei das Resultat den Namen r erhält. a, b und r werden durch Parameter mit den Komponenten 0...3 dargestellt; jede Komponente kann die Werte 0 und 1 annehmen, und die Folge dieser Werte wird als Binärzahl interpretiert. Für die Addition wird noch ein mit c bezeichnetes Übertragsregister benötigt.

```
%<r ← add a b>
::=
    [(1) c ←(((a(1,,23) △ b(1,,23)) ▽
              (a(1,,23) △ c(1,,23))) ▽
              (b(1,,23) △ c(1,,23)))&0;
     (2) r ←(a+b+c) mod 2            ]
```

Der Modul auf der rechten Seite enthält zwei bedingungslose Regeln (bei denen der Pfeil → fortgelassen werden darf) mit den Prioritäten 1 und 2. Die erste hat die Form

$$c \leftarrow \varphi(a,b,c)\ \&\ 0,$$

wo φ eine *komponentenweise* zu verstehende Boole'sche Funktion ist. Diese Regel wird zuerst iterativ so lange angewendet, bis sich c nicht mehr ändert. Die angehängte (verkettete) Null bewirkt, dass auf jeden Fall die Komponente c.23 (die der am wenigsten signifikanten Stelle entspricht) den Wert 0 erhält. Mit zweiter Priorität werden dann a, b und c komponentenweise modulo 2 addiert, was der exklusiven Oder-Verknüpfung entspricht.

Beispiel 7. Beschreibung eines einfachen Aufgabenverteilers *(Scheduler)* für einen Prozessor, beispielsweise für eine rechnergesteuerte Telefonzentrale:

Es ist dabei angenommen, dass alle Ereignisse (auch die externen, wie das Abheben des Hörers durch einen Teilnehmer) in eine Warteschlange eingereiht werden, die *Eingabebereich* heisst. Beschreibung auf der obersten Abstraktionsebene:

```
[1 # v takt ::
       <bearbeite eingabebereich> ;
 2 # <aktionsklasse 1>          ;
 3 # <aktionsklasse 2>          ;
 4 # <aktionsklasse 3>          ]
```

Wir haben hier ein Unterbruchsystem mit vier Stufen, und in den untersten drei Stufen drei Klassen von Aktionen mit verschiedener Priorität. Sobald der Taktpuls kommt, werden alle diese Aktionen zunächst unterbrochen, bis die Operation «bearbeite Eingabebereich» ausgeführt ist:

```
% <bearbeite eingabebereich>
::=
    → [1 # <eingabebereich nicht leer>
           ⇒ [<prozess&ereignis
                ← ws eingabebereich> ;
               <bearbeite eintragung>];
       2 # exit ]
;;
```

<q **nicht leer**> symbolisiert die Bedingung, dass die Warteschlange q mindestens ein Element enthält, und <a ← **ws** q> und <**ws** q ← b> (siehe nachstehend) symbolisieren das Übertragen des ersten Elementes der Warteschlange nach a beziehungsweise das Einreihen eines weiteren Elementes b an ihrem Ende; diese Makros sind hier nicht definiert. Solange der Eingabebereich nicht leer ist, wird die jeweils erste Zeile als Prozessnummer und Ereignis gespeichert und bearbeitet; erst wenn die Warteschlange leer ist, wird der Block verlassen, worauf die unterbrochenen Aktionen wieder aufgenommen werden.

```
% <bearbeite eintragung>
::=
    ⇒ [aktion & priorität
        ← ze-tabelle.(z-liste.(prozess)).(ereignis);
       <ws buffer.(priorität) ← aktion & prozess> ;
       <verzögerung d>]
;;
```

Aus einer Zustand/Ereignis-Tabelle *(ze-tabelle)* werden Aktionsnummer und Priorität entnommen. Die «Z-Liste» enthält für jeden Prozess den gegenwärtigen Zustand. Dann wird die Aktionsnummer zusammen mit der Nummer des betroffenen Prozesses in die der Priorität j entsprechende Warteschlange *buffer.j* eingereiht. <**verzögerung** d> symbolisiert die bei diesen Operationen verbrauchte Zeit. Jede Aktionsklasse enthält eine Liste aller Aktionen und die Prozedur zum Aufrufen der nächsten Aktion:

```
% <aktionsklasse $N>
::=
    [1 # <liste $N> ;
     2 # <buffer. $N nicht leer >
         ⇒ [<aktn.$N & proz. $N ← ws buffer. $N> ;
            start. $N. (aktn. $N) ← 1      ]]
;;
```

Die einzelnen Aktionen in der Liste haben zwar höhere Priorität, können aber nur durch ein Signal *start.j.k* von der Aufrufprozedur her mit niedriger Priorität gestartet werden. Wenn eine Aktion abgeschlossen und die entsprechende Warteschlange *buffer.j* nicht leer ist, werden die Nummern der nächsten Aktion und des betroffenen Prozesses nach *aktn.j* beziehungsweise *proz.j* übertragen; anschliessend wird das Startsignal gegeben. Die Form der Aktionslisten ist schliesslich

```
% <liste k >
::=
    [m start.k.1 :: → < tue dies mit proz.k> ;
     m start. k.2 :: → < tue das mit proz.k> ;
     — — — — —                              ]
;;
```

wobei k = 1, 2 oder 3.

8 Schlussbemerkungen

Diese Darstellung der Beschreibungssprache SYM musste notwendigerweise unvollständig bleiben; für weitere Einzelheiten sei auf [13] verwiesen. Wie dort gezeigt wird, lassen sich stochastische Grössen ohne Schwierigkeiten in die Sprache einführen, was zum Beispiel für die Simulation einer Telefonzentrale wichtig ist.

Es war eines der Hauptanliegen, eine Sprache zu entwerfen, in der alles Notwendige möglichst direkt ausgedrückt werden kann. Es galt dabei einen Pfad zwischen der Weitschweifigkeit von ALGOL und der undurchsichtigen Kompaktheit von APL zu finden. Nach der gesammelten Erfahrung lässt sich SYM auf den verschiedensten Abstraktionsebenen anwenden. Das Problem ist eher, dass die Freiheit in der Wahl der Ausdrucksmittel ziemlich gross ist. In vielen Fällen müssen für ein gegebenes Anwendungsgebiet die Abstraktionsebenen, das heisst gewisse allgemeine Konzepte erst erarbeitet werden. Die Erfahrung zeigt ferner, dass in einem Punkte eine gewisse Präzision in der Beschreibung der Systeme verlangt werden muss, nämlich hinsichtlich der zeitlichen Relationen. Die hier auftretenden Probleme werden jedoch nicht durch die Sprache künstlich erzeugt, sie liegen in der Natur der Sache. Ihre Nichtbeachtung kann bei einer Verwirklichung der Systeme leicht zu Fehlverhalten führen.

Bibliographie

[1] *Minsky M.* Computation: Finite and Infinite Machines. Prentice-Hall International, 1972.

[2] *Kawashima H., Futami K.* and *Kano S.* Functional Specification of Call Processing by State Transition Diagram. IEEE Trans. Communication Technology 19 (October 1971) No 5, p. 581...587.

[3] *Hemdal G.* The Function Flow Chart, a Specification and Design Tool for SPC-Exchanges. Software Engineering for Telecommunication Switching Systems, IEE Conference Publication (1973) No 97, p. 262...270.

[4] *UKPO.* Glossary of Symbols and Conventions for Progression Chart Purposes. AGSD-Document CA4/11, 1972.

[5] *The International Telegraph and Telephone Consultative Commitee (CCITT).* Sixth Plenary Assembly. Recommendation Z. 101...104, Functional Specification and Description Language (SDL) (erscheint voraussichtlich 1977).

[6] *Baer J. L.* A Survey of Some Theoretical Aspects of Multiprocessing. Computing Surveys 5 (March 1973) No 1, p. 31...80.

[7] *Chu Y.* Computer Organization and Micro-Programming. Prentice-Hall 1972.

[8] *Chu Y.* (ed.). Hardware Description Languages. Spezialnummer von: Computer (IEEE) 7 (December 1974) No 12.

[9] 1975 International Symposium on Computer Hardware Description Languages and Their Applications. IEEE Konferenzbericht, 1975.

[10] *Bell C. G.* and *Newell A.* Computer Structures: Readings and Examples. McGraw-Hill, 1971.

[11] *Falkoff A. D., Iverson K. E.* and *Sussenguth E. H.* A Formal Description of System/360. IBM Systems Journal 3 (1964), p. 198...261.

[12] *Vogel E. W.* A Model Approach to the Description of Hardware Systems. 1975 International Symposium on Computer Hardware Description Languages and their Applications. IEEE Konferenzbericht (1975), p. 32...37.

[13] *Vogel E. W.* SYM 76: A Description Language for Real-Time Systems. First draft, 1976 (nicht veröffentlicht).