Grundlagen von Interaktionsausdrücken

Christian Heinlein, Abt. DBIS

Juni 1997

1. Einleitung

Dieser Bericht stellt eine umfassende Einführung in die Grundlagen von *Interaktionsausdrücken* dar und dient damit gleichermaßen als „Tutorium“ und als „Referenzhandbuch“ zu diesem Formalismus.

Obwohl Interaktionsausdrücke primär mit der Zielsetzung entwickelt wurden, einen geeigneten Formalismus zur Beschreibung (und Implementierung) von *Inter-Workflow-Abhängigkeiten* zur Verfügung zu stellen, ist ihre Anwendbarkeit nicht auf dieses spezielle Gebiet beschränkt. Vielmehr stellen sie einen sehr allgemeinen Formalismus zur Beschreibung (und damit zur Lösung) nahezu beliebiger *Synchronisationsprobleme* dar.

Aus diesem Grund werden Interaktionsausdrücke in diesem Bericht unabhängig von der speziellen Anwendungsdomäne Inter-Workflow-Abhängigkeiten eingeführt und mit Beispielen aus ganz unterschiedlichen Bereichen veranschaulicht.

Der Bericht ist grob wie folgt aufgebaut:

- Abschnitt 2 definiert *Aktionen* als „Grundbausteine“ von Interaktionsausdrücken, grenzt sie gegenüber *Aktivitäten* ab und erläutert kurz das diesem Bericht zugrundegelegte Ausführungsmodell von Interaktionsausdrücken.
- Abschnitt 3 definiert die *Basisoperatoren* von Interaktionsausdrücken: sequentielle und parallele Komposition und Iteration sowie Disjunktion, Konjunktion und Synchronisation.
- Abschnitt 4 behandelt Vorrang- und einfache „Rechenregeln“.
- Abschnitt 5 definiert *Multiplikatoren* als eine naheliegende Verallgemeinerung der in Abschnitt 3 eingeführten binären Operatoren.
- Abschnitt 6 erläutert kurz die beiden wesentlichen *Auswahlprinzipien* von Interaktionsausdrücken (*implizite* und *vorausschauende* Auswahl) und grenzt sie gegenüber typischen „imperativen“ Kontrollkonstrukten ab.
- Abschnitt 7 demonstriert die typische Verwendung von Interaktionsausdrücken anhand zahlreicher Beispiele und versucht auf diese Weise, einen ersten Eindruck ihrer Ausdrucksmächtigkeit zu vermitteln.
- Abschnitt 8 behandelt *Abstraktionsmechanismen* (Makros und benutzerdefinierte Operatoren) als wichtige Hilfsmittel zur Verbesserung der Lesbarkeit und Änderungsfreundlichkeit (Wartbarkeit) von Interaktionsausdrücken.
- Abschnitt 9 diskutiert kurz alternative *Notationen* von Interaktionsausdrücken als weiteres Potential zur Erhöhung ihrer Benutzerfreundlichkeit, insbesondere für „Informatik-Laien“.

- Abschnitt 10 schließlich definiert parametrisierte Ausdrücke und *Quantoren* als eine weitere Verallgemeinerung von Multiplikatoren, die in der Praxis von wesentlicher Bedeutung ist, und greift in diesem Zusammenhang einige Beispiele aus Abschnitt 7 wieder auf.

Zu den folgenden Aspekten von Interaktionsausdrücken, die in diesem Bericht nicht oder nur sehr kurz behandelt werden, sind weiterführende Berichte in Vorbereitung:

- Graphische Repräsentation von Interaktionsausdrücken.
- Formale Semantik und Eigenschaften von Interaktionsausdrücken.
- Implementierung von Interaktionsausdrücken.
- Vergleich von Interaktionsausdrücken mit verwandten Ansätzen.
- Erweiterungen von Interaktionsausdrücken.
- Konditionale und temporale Aspekte bei Interaktionsausdrücken.
- Beschreibung von Inter-Workflow-Abhängigkeiten mit Interaktionsausdrücken.
- Integration von Interaktionsausdrücken mit Workflow-Management-Systemen.

Außerdem sei an dieser Stelle auf den bereits erschienenen Bericht “Interaction Expressions − A Powerful Formalism for Describing Inter-Workflow Dependencies” (UIB 97-04) verwiesen, in dem einige dieser Aspekte bereits teilweise behandelt werden. Insbesondere findet sich dort eine erste Diskussion verwandter Arbeiten.

2. Grundbegriffe

Aktionen stellen die „Grundbausteine“ von Interaktionsausdrücken dar. Wir nehmen an, daß Aktionen von einem hier nicht näher spezifizierten *Benutzer* (bei dem es sich auch um einen „Agenten“ im weitesten Sinne handeln kann) *ausgeführt* werden können, und daß eine solche Ausführung *keine zeitliche Ausdehnung* besitzt. Außerdem sollen zwei Aktionen *niemals gleichzeitig* ausgeführt werden können.

Da man es in der realen Welt jedoch typischerweise mit zeitlich ausgedehnten *Aktivitäten* zu tun hat, deren Ausführung auch zeitlich überlappen kann, werden diese formal als eine sequentielle Komposition (siehe Abschnitt 3.3) zweier Aktionen A_{start} und A_{term} definiert, die das *Starten* bzw. *Beenden* der Aktivität A bezeichnen. Auf diese Weise läßt sich eine nebenläufige Ausführung von Aktivitäten auf eine sequentielle Ausführung von Aktionen zurückführen, die technisch wesentlich einfacher zu handhaben ist.

Im Kontext von Workflow-Management stellt jeder Arbeitsschritt eines Workflows eine Aktivität dar, während das Starten bzw. Beenden eines Schritts eine Aktion darstellt.

Interaktionsausdrücke (im folgenden auch kurz als *Ausdrücke* bezeichnet) legen *zulässige Ausführungsreihenfolgen* von Aktionen fest.

Ein Ausdruck selbst wird „ausgeführt“, indem eine Folge von Aktionen ausgeführt wird, die aus Sicht dieses Ausdrucks zulässig ist. Wenn mehrere solche Folgen existieren (was häufig der Fall ist), nehmen wir (als mentales Modell) an, daß nichtdeterministisch stets die „richtige“ gewählt wird, d. h. diejenige Folge von Aktionen ausgeführt wird, die im „Einklang“ mit den Absichten des oben erwähnten Benutzers steht.

Für eine reale Implementierung bedeutet dies natürlich, daß sie *alle* zulässigen Alternativen verfolgen muß, damit die vom Benutzer beabsichtigte Ausführungsreihenfolge auf jeden Fall akzeptiert wird, sofern sie zulässig ist.

Im folgenden werden Interaktionsausdrücke anhand dieses „intuitiven Ausführungsmodells“ definiert. Durch Aussagen der Form „Ein Ausdruck der Gestalt ... wird ausgeführt, indem ... ausgeführt wird“ wird im Prinzip die Menge der Ausführungsreihenfolgen festgelegt, die aus Sicht dieses Ausdrucks zulässig sind. Eine theoretisch fundiertere Definition mit Hilfe von formalen Sprachen (die aber bei weitem nicht so gut verständlich ist) folgt in einem separaten Bericht.

3. Elementare Ausdrücke

Im folgenden bezeichnen x und y beliebige Ausdrücke, während $a, b, c, \ldots$ einzelne Aktionen darstellen.

3.1 Atomare Ausdrücke

Ein *atomarer Ausdruck* der Gestalt a wird ausgeführt, indem die Aktion a einmal ausgeführt wird.

3.2 Selektion

Eine *Selektion* oder *Disjunktion* der Gestalt $x \mid y$ wird ausgeführt, indem von den beiden Ausdrücken x und y *genau einer* ausgeführt wird.

Der Ausdruck $a \mid b$ läßt also die Ausführung von a oder von b zu.

3.3 Sequentielle Komposition und Iteration

Eine *sequentielle Komposition* der Gestalt $x - y$ wird ausgeführt, indem zuerst der Ausdruck x und *anschließend* der Ausdruck y ausgeführt wird.

Eine *sequentielle Iteration* der Gestalt $x*$ wird ausgeführt, indem der Ausdruck x beliebig oft (auch keinmal) *nacheinander* ausgeführt wird. Formal ist dies äquivalent zu dem Ausdruck

$$\lambda \mid x \mid x - x \mid x - x - x \mid \ldots,$$

wobei λ für eine leere Folge von Aktionen steht.

Der Ausdruck $a - b$ erlaubt also die Ausführungsreihenfolge ab, während $(a - b)*$ die Reihenfolgen $\lambda, ab, abab, \ldots$ zuläßt.

3.4 Parallele Komposition und Iteration

Analog zur sequentiellen Komposition bzw. Iteration können Ausdrücke auch *parallel* verknüpft bzw. wiederholt werden.

Eine *parallele Komposition* der Gestalt $x + y$ wird ausgeführt, indem die beiden Ausdrücke x und y *parallel* ausgeführt werden. Da zwei Aktionen jedoch niemals gleichzeitig ausgeführt werden können, bedeutet parallele Ausführung hier *überlappende* oder *verschränkte* Ausführung der beiden Ausdrücke.

Eine *parallele Iteration* der Gestalt $x\#$ wird ausgeführt, indem beliebig viele (auch keine) Ausprägungen des Ausdrucks x parallel ausgeführt werden. Analog zur sequentiellen Iteration gilt daher:

$$x\# = \lambda \mid x \mid x + x \mid x + x + x \mid \ldots$$

Der Ausdruck $a + b$ erlaubt also die Ausführungsreihenfolgen ab und ba, während $(a - c) + (b - c)$ eine beliebige „Vermischung“ der Reihenfolgen ac und bc, d. h. die Ausführungsreihenfolgen $acbc$, $abcc$, $bcac$ und $bacc$ zuläßt.

Der Ausdruck $(a - b)\#$ erlaubt eine beliebige Vermischung von beliebig vielen Reihenfolgen ab, d. h. unter anderem λ, ab, $abab$, $aabb$ usw.

3.5 Konjunktion

Eine *Konjunktion* der Gestalt $x \,\&\, y$ wird ausgeführt, indem die beiden Ausdrücke x und y *simultan* ausgeführt werden, d. h. es sind nur Ausführungsreihenfolgen zulässig, die sowohl von x als auch von y zugelassen werden.

Der Ausdruck $(a \mid b) \,\&\, (b \mid c)$ ist also z. B. äquivalent zum Ausdruck b. Man beachte hierbei einen wesentlichen Unterschied zur parallelen Komposition: Während der Ausdruck $(a \mid b) + (b \mid c)$ eine beliebige „Vermischung“ von a oder b mit b oder c (also letztlich die Ausführungsreihenfolgen ab, ba, ac, ca, bb, bc, cb) erlaubt, läßt der Ausdruck $(a \mid b) \,\&\, (a \mid c)$ nur die Reihenfolge b zu.

„Automatentheoretisch“ gesprochen, wird ein „Eingabesymbol“ (d. h. hier eine Aktion) bei einer parallelen Komposition von *einem* der beiden Teilausdrücke, bei einer Konjunktion jedoch von *beiden* Teilausdrücken *gleichzeitig* verarbeitet.

3.6 Synchronisation

Eine *Synchronisation* der Gestalt $x \mathbin{@} y$ stellt eine in der Praxis häufig benötigte Mischform der parallelen Komposition $x + y$ und der Konjunktion $x \,\&\, y$ dar. Sie wird ausgeführt, indem *gemeinsame* Aktionen der Ausdrücke x und y wie bei der Konjunktion *simultan* ausgeführt werden, während Aktionen, die nur in *einem* der beiden Ausdrücke x oder y auftreten, wie bei der parallelen Komposition *unabhängig* voneinander ausgeführt werden.

Unter der Annahme, daß $x_1, x_2, \ldots$ (bzw. $y_1, y_2, \ldots$) die Aktionen sind, die nur im Ausdruck x (y), nicht jedoch im Ausdruck y (x) auftreten, läßt sich die Synchronisation von x und y formal wie folgt darstellen:

$$x \mathbin{@} y = (x + (y_1 \mid y_2 \mid \ldots)^*) \,\&\, (y + (x_1 \mid x_2 \mid \ldots)^*).$$

Durch die parallele Komposition von x mit $(y_1 \mid y_2 \mid \ldots)^*$ wird erreicht, daß der linke Teilausdruck der Konjunktion parallel zur Ausführung von x beliebige Ausführungen von $y_1, y_2, \ldots$ erlaubt, und umgekehrt.

Als Beispiel betrachte man den Ausdruck

$$(a - c) \mathbin{@} (b - c) = ((a - c) + b^*) \,\&\, ((b - c) + a^*),$$

dessen erster (bzw. zweiter) Teil Ausführungsreihenfolgen der Gestalt $B_1 a B_2 c B_3$ (bzw. $A_1 b A_2 c A_3$) zuläßt, wobei B_1 bis B_3 bzw. A_1 bis A_3 jeweils für eine beliebige Folge von b's bzw. a's steht. Der Gesamtausdruck erlaubt somit die beiden Reihenfolgen abc und bac.

Im Gegensatz hierzu würde die parallele Komposition $(a - c) + (b - c)$, wie oben bereits erwähnt, die Reihenfolgen $acbc$, $abcc$, $bcac$, $bacc$ zulassen, während die Konjunktion $(a - c) \,\&\, (b - c)$ *unerfüllbar* wäre, weil es keine Ausführungsreihenfolge gibt, die sowohl von $a - c$ als auch von $b - c$ zugelassen wird.

Dieses Beispiel verdeutlicht sehr gut die typische Verwendung der Synchronisation: Unabhängig voneinander entwickelte Teilausdrücke sollen im Prinzip zwar konjunktiv verknüpft werden, ein Teilausdruck soll aber Aktionen, über die er „keine Aussage macht“, nicht blockieren, sondern jederzeit zulassen.

Dementsprechend ist die Synchronisation *x* @ *y* in dem Spezialfall, daß die *Alphabete* (d. h. Aktionsmengen) der Ausdrücke *x* und *y* *gleich* sind, äquivalent zur Konjunktion *x* & *y*, während sie in dem anderen Spezialfall, daß die Alphabete von *x* und *y* *disjunkt* sind, äquivalent zur parallelen Komposition *x* + *y* ist.

3.7 Anmerkung

Die Wahl der Operatorsymbole (z. B. – für sequentielle und + für parallele Komposition) mag auf den ersten Blick willkürlich erscheinen. Zum Teil werden dieselben Symbole in verwandten Ansätzen sogar mit ganz anderer Bedeutung verwendet. Daher soll hier eine kurze Begründung für die getroffene Wahl gegeben werden, die vielleicht auch als Merkhilfe dienen kann.

– als Zeichen für sequentielle Komposition wurde graphischen Darstellungen nachempfunden, in denen aufeinanderfolgende Aktionen i. d. R. durch Striche oder Pfeile verbunden werden. Verwandte Ansätze verwenden oft ein Semikolon oder gar keinen Operator (d. h. lediglich Aneinanderreihung von Ausdrücken) für die sequentielle Komposition.

* als Zeichen für sequentielle Iteration ist allgemein üblich. Außerdem steht es mit dem Symbol für sequentielle *Komposition* (–) insofern in Beziehung, als es optisch aus mehreren Strichen zusammengesetzt ist.

+ als Zeichen für parallele Iteration ist eher ungewöhnlich. In verwandten Ansätzen wird häufig | oder || verwendet, während + in manchen theoretischen Arbeiten eher für Disjunktion verwendet wird. Durch die Wahl von + soll jedoch zum einen die Dualität zur *sequentiellen* Komposition (–) ausgedrückt werden, zum anderen wird + umgangssprachlich oft als „und“ gesprochen, und damit wird im Gegensatz zur Disjunktion („oder“) ausgedrückt, daß *beide* Teilausdrücke ausgeführt werden sollen.

Für die parallele Iteration gibt es kein gebräuchliches Zeichen, weil sie in fast keinem der verwandten Ansätze vorkommt. Die Wahl des Zeichens fiel auf #, um die Analogie zur *sequentiellen* Iteration anzudeuten: Während * aus mehreren – zusammengesetzt ist, ist # aus mehreren + zusammengesetzt. (Dies ist auch ein weiterer Grund, die parallele Komposition durch + auszudrücken).

| als Zeichen für Disjunktion wurde der in Unix gebräuchlichen Notation für reguläre Ausdrücke sowie der Programmiersprache C entlehnt.

Letzteres gilt auch für & als Zeichen für Konjunktion. In verwandten Ansätzen, in denen Konjunktion auftritt, wird i. d. R. dasselbe Zeichen verwendet.

@ als Zeichen für Synchronisation ist relativ willkürlich gewählt. Wenn man will, kann man das stilisierte a jedoch als Abkürzung für ein logisches “and” auffassen, denn schließlich stellt die Synchronisation ja eine Variante der Konjunktion dar.

4. Vorrang- und einfache Rechenregeln

4.1 Iterationen

Die Iterationen * und # haben − wie für unäre Operatoren üblich − Vorrang vor allen anderen Operatoren. Da beide postfix angewandt werden, erübrigt sich die Frage nach ihrem Vorrang untereinander.

Eine Folge von *-Operatoren ist äquivalent zu einem einzigen * (d. h. * ist *idempotent*), und eine Folge von *- und #-Operatoren ist äquivalent zu einem einzigen # (d. h. # ist ebenfalls idempotent, * und # sind *vertauschbar*, und # *dominiert* *).

4.2 Kompositionen

Die sequentielle Komposition − bindet stärker als ihr paralleles Pendant +, weil man vermutlich häufiger parallel auszuführende Sequenzen als sequentiell auszuführende „Parallelitäten“ formuliert.

4.3 Boolesche Operatoren

Wie üblich bindet die Konjunktion stärker als die Disjunktion, beide jedoch − wie für Boolesche Operatoren üblich − schwächer als die bisher genannten Basisoperatoren.

Beide Operatoren & und | sind idempotent, d. h. es gilt: $x \,\&\, x = x \mid x = x$.

4.4 Synchronisation

Da die Synchronisation in der Praxis meist auf der „äußersten Ebene“ zur Verknüpfung unabhängig entwickelter Teilausdrücke verwendet wird, ist ihr Vorrang niedriger als der aller anderen Operatoren.

Auch die Synchronisation @ ist idempotent.

4.5 Assoziativität und Kommuntativität

Alle binären Operatoren sind *assoziativ* und − mit Ausnahme der sequentiellen Komposition − auch *kommutativ*. Somit erübrigt sich die Frage nach impliziter Klammerung bei gleichem Vorrang.

4.6 Klammerung

Klammern können nach Belieben zur expliziten Vorrangregelung oder zur Verbesserung der Lesbarkeit eingesetzt werden. Um bei mehrfach verschachtelten Klammerausdrücken die Übersichtlichkeit zu erhöhen, können neben den üblichen runden Klammern auch eckige und geschweifte Klammern zur Gruppierung verwendet werden.

4.7 Anmerkung

Die genannten Rechenregeln erscheinen intuitiv einleuchtend, können aber natürlich nur mit Hilfe einer formalen Definition der Operatoren wirklich bewiesen werden.

Über Sinn und Unsinn der gewählten Vorrangregeln kann man natürlich ebenso diskutieren wie über die Wahl der Operatorsymbole. Wir haben oben wiederum versucht, die getroffene Wahl zu begründen und damit zumindest einprägsam zu machen.

Es zeigt sich jedoch in der Praxis, daß der hohe Vorrang der unären Operatoren nicht unbedingt ihrer typischen Verwendung entspricht. Meist werden sie nicht auf einzelne Aktionen, sondern auf komplexe Ausdrücke angewandt, die dann geklammert werden müssen. Da andererseits nicht klar ist, an welcher Stelle der Vorrangskala sie besser „aufgehoben“ wären, belassen wir es bei der oben genannten und allgemein üblichen Regelung.

5. Multiplikatoren

Multiplikatoren stellen eine naheliegende Verallgemeinerung der binären Operatoren –, +, &, | und @ dar, ähnlich wie das aus der Mathematik bekannte Summen- oder Produktzeichen (Σ bzw. Π) eine Verallgemeinerung der arithmetischen Operationen Addition bzw. Multiplikation darstellt.

Um jedoch nicht für jeden binären Operator ein zusätzliches Multiplikatorsymbol einführen zu müssen, verwenden wir eine ebenfalls aus der Mathematik stammende Konvention, nach der ein beliebiger binärer Operator (wie z. B. die Mengenoperatoren ∪ und ∩) auch als entsprechender „Summenoperator“ verwendet werden kann, wenn er etwas größer geschrieben und mit den gewünschten Bereichsgrenzen versehen wird (z. B. $\bigcup_{i=1}^{n} M_i$).

5.1 Einfache Multiplikatoren

Ein beliebiger binärer Operator ~ kann wie folgt als *einfacher Multiplikator* verwendet werden:

$\overset{n}{\sim} x = x \sim \ldots \sim x$ (n-mal x, $n > 0$);

$\overset{n}{\sim} x = \lambda$ für $n \leq 0$.

n heißt hierbei *Faktor*.

Aufgrund der Idempotenz der Operatoren &, | und @ gilt natürlich

$\overset{n}{\&} x = \overset{n}{|} x = \overset{n}{@} x = x$ (für $n > 0$),

so daß diese Notation nur für die sequentielle und parallele Komposition wirklich nützlich ist.

5.2 Indizierte Multiplikatoren

Weiterhin kann jeder binäre Operator ~ als *indizierter Multiplikator* verwendet werden:

$$\underset{k=m}{\overset{n}{\sim}} x_k = x_m \sim \ldots \sim x_n \; (m, n \in \mathbb{Z}).$$

Die Variable k heißt hierbei *Index*, m und n heißen *untere* bzw. *obere Bereichsgrenze*. m und n können beliebige ganze Zahlen sein, insbesondere kann $m = n$ oder auch $m > n$ gelten.

Der Index k kann innerhalb des Ausdrucks x_k entweder als Faktor oder als Bereichsgrenze geschachtelter Multiplikatoren verwendet werden. Beispielsweise besagt der Ausdruck $\overset{k}{—} x$, daß der Ausdruck x genau k-mal wiederholt werden muß, während der Ausdruck $\underset{k=m}{\overset{n}{|}} \overset{k}{—} x$ m bis n Wiederholungen von x erlaubt.

5.3 Vorrangregeln

Syntaktisch gesehen sind Multiplikatoren unäre Präfix-Operatoren. Ihr Vorrang ist höher als der von binären Operatoren und niedriger als der der unären Postfix-Operatoren. Da alle Multiplikatoren präfix angewandt werden, erübrigt sich die Frage nach dem Vorrang untereinander.

Die Anmerkungen zu unären Operatoren aus Abschnitt 4.7 treffen in gleicher Weise auf Multiplikatoren zu. Auch sie werden i. d. R. auf komplexe Ausdrücke angewandt, die dann explizit geklammert werden müssen.

5.4 Anmerkung

Streng genommen sind Multiplikatoren nur „syntactic sugar“; sie können stets gemäß ihrer Definition durch binäre Operatoren ersetzt werden. Dennoch gibt es einige wichtige Gründe für ihre Verwendung:

- Die resultierenden Ausdrücke sind z. T. *erheblich* kompakter, übersichtlicher und *änderungsfreundlicher* als die äquivalenten „ausmultiplizierten“ Ausdrücke.
- Auch bei der internen Repräsentation eines Ausdrucks in einem Algorithmus kann ein Multiplikator-Ausdruck u. U. kompakter dargestellt und *effizienter verarbeitet* werden als sein ausmultipliziertes Äquivalent.
- („Ungebundene“) Faktoren und Bereichsgrenzen von Multiplikatoren können als Variablen interpretiert werden, die erst zur Laufzeit mit konkreten numerischen Werten belegt werden.

Darüber hinaus stellen Multiplikatoren quasi eine Vorstufe von *Quantoren* dar (siehe Abschnitt 10), die sich formal nicht mehr direkt auf die ursprünglichen binären Operatoren zurückführen lassen.

6. Auswahlprinzipien

6.1 Implizite Auswahl

Die Operatoren für sequentielle Komposition, Selektion und sequentielle Iteration erinnern an die aus Programmier- oder auch Workflow-Beschreibungssprachen bekannten Konstrukte *Sequenz*, *Verzweigung* und *Wiederholung*. Ein wesentlicher Unterschied besteht jedoch darin, daß es in Interaktionsausdrücken per se keine *Bedingungen* gibt. Bei einer Selektion $a \mid b$ beispielsweise entscheidet nicht der Wahrheitsgehalt eines Booleschen Ausdrucks darüber, welcher Zweig gewählt wird, sondern einfach die Tatsache, welche Aktion zuerst ausgeführt wird. (Da zwei Aktionen niemals gleichzeitig ausgeführt werden können, gibt es immer eine „zuerst ausgeführte“ Aktion!) Dasselbe gilt für die Termination einer Iteration wie z. B. $a*$. Wenn a erneut ausgeführt wird, fährt die Iteration fort, wenn eine zulässige Folgeaktion ausgeführt wird (z. B. b beim Ausdruck $a* - b$), wird sie beendet. Dieses Prinzip wird im folgenden als *implizite Auswahl* bezeichnet.

6.2 Vorausschauende Auswahl

Derartige Auswahl-Entscheidungen können jedoch nicht immer „sofort“ getroffen werden. Beispielsweise erlauben bei dem Ausdruck $a - b \mid a - c$ beide Zweige der Selektion die Ausführung der Aktion a. Wenn sie also ausgeführt wird, ist zunächst nicht klar, welcher Zweig der Selektion damit „gewählt“ wird. Erst durch eine nachfolgende Ausführung von b oder c kann diese Entscheidung quasi „rückwirkend“ getroffen werden.

Ähnlich verhält es sich bei einem Ausdruck wie $a* - a - b$. Bei Ausführung der Aktion a kann noch nicht entschieden werden, ob dies als Teil der Iteration $a*$ oder als Ausführung der nachfolgenden Aktion a interpretiert werden soll. Diese Entscheidung kann immer erst „einen Schritt später“ getroffen werden: Wenn ein weiteres a ausgeführt wird, gehörte das vorige zur Iteration; wenn jedoch b ausgeführt wird, wurde zuvor das der Iteration folgende a ausgeführt.

Wie bereits in Abschnitt 2 erwähnt wurde, kann man für ein mentales Modell − in Analogie zu nichtdeterministischen Automaten − postulieren, daß in solchen Fällen auf „magische“ Weise stets die „richtige“ Entscheidung getroffen wird. Dieses Prinzip wird im folgenden als *vorausschauende Auswahl* bezeichnet.

Es muß jedoch noch einmal betont werden, daß es in Wirklichkeit keine „Magie“ gibt, sondern daß eine Entscheidung, die momentan noch nicht getroffen werden kann, aufgeschoben wird, d. h. daß *jede* momentan denkbare Alternative solange weiterverfolgt wird, bis sie sich definitiv als falsch erweist (weil eine Aktion ausgeführt wird, die aus Sicht dieser Alternative unzulässig ist). Das Prinzip der vorausschauenden Auswahl soll lediglich das mentale Ausführungsmodell von Interaktionsausdrücken vereinfachen, indem statt der Menge aller möglichen Alternativen immer nur eine „richtige“ (d. h. bis zum Schluß mögliche) Alternative betrachtet wird.

Wie wir im weiteren sehen werden, erlaubt die Kombination dieser beiden Auswahlprinzipien Formulierungen, deren Kompaktheit und Eleganz mit Formalismen, die auf expliziten Bedingungen beruhen, kaum erreicht werden kann.

7. Beispiele

Um die im letzten Abschnitt aufgestellte Behauptung ein wenig zu untermauern, soll an dieser Stelle die Ausdrucksmächtigkeit von Interaktionsausdrücken mit Hilfe einiger Beispiele verdeutlicht werden. Anhand von bekannten und z. T. auch weniger bekannten (aber interessanten!) Synchronisationsproblemen soll demonstriert werden, daß viele derartige Probleme mit Hilfe von Interaktionsausdrücken auf eine sehr kompakte und elegante Weise gelöst werden können.

7.1 Aktivitäten

Wie eingangs bereits erwähnt wurde, kann eine Aktivität A als sequentielle Komposition $A_{\text{start}} - A_{\text{term}}$ dargestellt werden, wobei die Aktionen A_{start} und A_{term} das Starten bzw. Beenden der Aktivität A bezeichnen.

Da in praktischen Anwendungen fast immer Aktivitäten (und nicht Aktionen) auftreten, kann der Name einer Aktivität A in Ausdrücken stets als Abkürzung für diese Sequenz verwendet werden.

7.2 Bekannte Synchronisationsprobleme

Erzeuger/Verbraucher

Gegeben sei eine Aktivität *produce*, die ein Objekt erzeugt, das anschließend von einer Aktivität *consume* verbraucht wird. Sofern es nur einen einelementigen Zwischenpuffer für Objekte gibt, müssen die beiden Aktivitäten abwechselnd ausgeführt werden, und es muß mit *produce* begonnen werden:

$$(produce - consume)*.$$

Falls ein unbegrenzter Zwischenpuffer zur Verfügung steht, kann die Iteration jedoch auch parallel ablaufen:

$$(produce - consume)\#.$$

Dieser Ausdruck stellt sicher, daß die Anzahl der begonnenen *consume*-Aktivitäten die Anzahl der beendeten *produce*-Aktivitäten niemals übersteigt.

Wenn der Zwischenpuffer maximal n Elemente aufnehmen kann, dürfen maximal n *produce-consume*-Sequenzen parallel ablaufen:

$$\overset{n}{+}(produce - consume)*.$$

Man beachte, daß aufgrund der Vorrangregeln in jedem Zweig der parallelen Komposition eine eigene Iteration ausgeführt wird. Der Ausdruck

$$\left(\overset{n}{+}(produce - consume)\right)*,$$

bei dem die Iteration nach außen gezogen wurde, besagt etwas anderes! Er erlaubt zunächst die parallele (oder auch sequentielle) Ausführung von n *produce-consume*-Sequenzen. Die (n+1)-te derartige Sequenz kann jedoch erst begonnen werden, wenn die ersten n Sequenzen alle beendet sind!

Semaphore

Ein binäres Semaphor mit Initialwert 1 und den Operationen P (Erwerben, d. h. Dekrementieren) und V (Freigeben, d. h. Inkrementieren) kann wie folgt beschrieben werden:

$$(P - V)*.$$

Ein Semaphor mit Initialwert 0, das beliebige ganzzahlige Werte ≥ 0 annehmen kann, kann wie folgt beschrieben werden:

$$(V - P)\#.$$

Analog zum Erzeuger/Verbraucher-Problem wird durch diesen Ausdruck sichergestellt, daß jedem begonnenen P ein „zugehöriges“ beendetes V vorausgeht.

Wechselseitiger Ausschluß

Eine Menge von Aktivitäten (oder komplexen Ausdrücken) $A, B, \ldots$, die auf eine gemeinsame Ressource zugreifen und daher nicht gleichzeitig ausgeführt werden dürfen, kann wie folgt synchronisiert werden:

$$(A \mid B \mid \ldots)*.$$

Bei jeder Iteration kann *entweder A oder B oder* ... begonnen werden, und die nächste Iteration kann erst beginnen, wenn die vorige beendet ist.

Leser/Schreiber

Das Leser/Schreiber-Problem kann als Verallgemeinerung des Problems des wechselseitigen Ausschlusses betrachtet werden: Zwei Aktivitäten *read* und *write* greifen auf dieselbe Ressource zu, allerdings ist es zulässig, das mehrere *read*-Aktivitäten gleichzeitig aktiv sind. Eine *write*-Aktivität schließt jedoch alle anderen Aktivitäten aus:

$$(read\,\# \mid write)*.$$

Diese erste Lösung des Problems „bevorzugt“ Leser: Solange noch mindestens ein Leser aktiv ist, können weitere Leser gestartet werden, selbst wenn bereits ein Schreiber „wartet“. Der Schreiber kann u. U. „verhungern“.

Um dies zu vermeiden, muß der Schreiber zunächst die Möglichkeit erhalten, seine Schreibabsicht „anzumelden“, was z. B. durch Ausführen einer Aktion *wantwrite* geschehen kann. Diese Aktion muß jederzeit ausführbar sein und dafür sorgen, daß anschließend nur noch *write* und keine neuen Ausprägungen von *read* mehr gestartet werden können:

$$(read\,\# \mid write)* \;@\; (wantwrite - write \mid read_{\text{start}})*.$$

Man beachte hier die typische Verwendung des Synchronisationsoperators @! Beachte außerdem, daß die *Aktion* $read_{\text{start}}$ im zweiten Teil der Synchronisation nicht durch die *Aktivität read* ersetzt werden darf, weil sonst *wantwrite* während einer Ausführung von *read* nicht ausgeführt werden könnte und der beabsichtigte Effekt damit wieder zunichte gemacht wäre.

Diese zweite Lösung des Leser/Schreiber-Problems behandelt Leser und Schreiber insofern „gleichberechtigt“, als während einer Ausführung von *write* kein weiterer Schreiber seine Schreibabsicht anmelden kann. Erst nach Beendigung von *write* kann *wantwrite* erneut ausgeführt werden.

Will man schließlich Schreiber „bevorzugen“, d. h. bereits während der Ausführung von *write* zulassen, daß sich ein weiterer Schreiber mit *wantwrite* anmeldet, so kann man das wie folgt ausdrücken:

$$(read\,\# \mid write)* \;@\; ((wantwrite - write)\,\# \mid read_{\text{start}})*.$$

Beachte, daß die generelle „Leser/Schreiber-Bedingung“ (Schreiber brauchen exklusiven Zugriff)

nach wie vor durch den ersten Teil der Synchronisation ausgedrückt wird, während der zweite Teil die „Priorität“ zwischen Lesern und Schreibern regelt. In dieser dritten Formulierung drückt er aus, daß *wantwrite* jederzeit zulässig ist und daß jedem ausgeführten *wantwrite* ein „zugehöriges“ *write* folgen muß, bevor das nächste $read_{\text{start}}$ zulässig ist. Man beachte außerdem eine gewisse Dualität zwischen den beiden Teilen der Synchronisation.

7.3 Spezielle Kontrollkonstrukte

Im Kontext von Workflow-Management (und möglicherweise auch für andere Anwendungen) wünscht man sich häufig mächtigere Kontrollkonstrukte als die drei aus prozeduralen Programmiersprachen bekannten Basiskonstrukte Sequenz, Verzweigung und Wiederholung.

Im folgenden werden für einige dieser Konstrukte Formulierungen vorgeschlagen, die insbesondere die Ausdrucksmächtigkeit von Multiplikatoren verdeutlichen.

Beliebige Reihenfolge

Ein typisches derartiges Konstrukt ist eine Menge von Aktivitäten oder Ausdrücken $A, B, \ldots$, die sequentiell in beliebiger Reihenfolge ausgeführt werden dürfen. Da jeder Ausdruck jedoch nur einmal ausgeführt werden darf, scheitern Formulierungen, die nur auf den drei „prozeduralen Basiskonstrukten“ basieren.

Mit Hilfe von paralleler Komposition kann man zumindest formulieren, daß jeder Ausdruck *genau einmal* ausgeführt werden darf:

$$A + B + \ldots$$

Kombiniert man dies mit dem bereits erwähnten Ausdruck für wechselseitigen Ausschluß, so erhält man sofort das gewünschte Resultat:

$$(A + B + \ldots) \,\&\, (A \mid B \mid \ldots)^*.$$

(Statt & könnte hier auch @ verwendet werden, da die Alphabete der beiden Teilausdrücke gleich sind.)

Ist die Anzahl n der Ausdrücke $A, B, \ldots$ bekannt, so kann man die „unbestimmte“ Iteration $(A \mid B \mid \ldots)^*$ auch durch die explizitere Formulierung $\overset{n}{—}(A \mid B \mid \ldots)$ ersetzen:

$$(A + B + \ldots) \,\&\, \overset{n}{—}(A \mid B \mid \ldots).$$

m-aus-n-Verzweigungen

Ersetzt man in diesem Ausdruck den Teilausdruck $(A + B + \ldots)$ durch den Teilausdruck $(A? + B? + \ldots)$ (wobei $x?$ eine Abkürzung für $x \mid \lambda$ sei, vgl. Abschnitt 8.2), der besagt, daß jeder Ausdruck $A, B, \ldots$ *höchstens* einmal ausgeführt werden darf, und ersetzt man außerdem den Faktor n durch einen Faktor $m \le n$, so erhält man eine (sequentielle) m-aus-n-Verzweigung, die besagt, daß von den n Ausdrücken $A, B, \ldots$ genau m ausgeführt werden sollen:

$$(A? + B? + \ldots) \,\&\, \overset{m}{—}(A \mid B \mid \ldots).$$

Um noch allgemeiner auszudrücken, daß m_1 bis m_2 Ausdrücke ausgeführt werden sollen, kann man den Multiplikator $\overset{m}{—}$ durch die Konstruktion $\bigsqcup_{m=m_1}^{m_2} \overset{m}{—}$ ersetzen:

$$(A? + B? + \ldots) \,\&\, \bigsqcup_{m=m_1}^{m_2} \overset{m}{—}(A \mid B \mid \ldots).$$

Um schließlich auszudrücken, daß die m bzw. m_1 bis m_2 Ausdrücke auch parallel ausgeführt werden dürfen, kann man die sequentielle Komposition $\overset{m}{-}$ jeweils durch ihr paralleles Pendant $\overset{m}{+}$ ersetzen.

7.4 Einfache Workflow-Beschreibungen

Interaktionsausdrücke können prinzipiell dazu verwendet werden, den Kontrollfluß eines Workflows zu beschreiben, sofern dieser nicht auf expliziten Bedingungen beruht. (Eine Erweiterung von Interaktionsausdrücken um explizite Bedingungen ist jedoch in Vorbereitung.) Zwei einfache Beispiele sollen dies verdeutlichen.

Ein speisender Philosoph

Die von Dijkstra vorgestellten „speisenden Philosophen“ verbringen ihr Leben bekanntlich mit Nachdenken (Aktivität *think*) und Essen (*eat*). Um essen zu können, müssen sie den Speisesaal betreten und sich an den Tisch setzen (*enter*) sowie die Gabeln zu ihrer linken und rechten in die Hand nehmen (*getleft* bzw. *getright*). Nach dem Essen legen sie die Gabeln wieder zurück auf den Tisch (*putleft* bzw. *putright*) und verlassen den Raum (*leave*), Der „Lebenszyklus“ eines solchen Philosophen kann daher wie folgt beschrieben werden:

$$(think - enter - (getleft + getright) - eat - (putleft + putright) - leave)^*.$$

Medizinische Untersuchungen

Die Untersuchung eines Patienten in einem Krankenhaus − einschließlich aller hierfür erforderlichen Vor- und Nachbereitungen − stellt ein etwas realistischeres Beispiel eines Workflows dar: Nachdem eine bestimmte Untersuchung (z. B. eine Sonographie oder eine Röntgenaufnahme) von einem Stationsarzt angeordnet wurde (Aktivität *order*), wird mit der leistungserbringenden Stelle (z. B. Innere Medizin oder Radiologie) ein Termin vereinbart (*appoint*) und der Patient für die Untersuchung vorbereitet (*prepare*) und über mögliche Risiken und Nebenwirkungen aufgeklärt (*inform*). Die Reihenfolge dieser beiden Schritte ist zwar beliebig, da beide jedoch den Patienten „benötigen“, können sie nicht gleichzeitig ausgeführt werden. Danach kann der Patient für die Untersuchung abgerufen (*call*) und tatsächlich untersucht werden (*examine*). Anschließend wird vom untersuchenden Arzt ein Befund verfaßt (*report*), der dann vom Stationsarzt gelesen wird (*read*):

$$order - (appoint + (prepare - inform \mid inform - prepare)) - call - examine - report - read.$$

8. Abstraktion

Die Beispiele des vorigen Abschnitts haben deutlich gemacht, daß Interaktionsausdrücke − insbesondere für den geübten Benutzer − ein mächtiges Werkzeug zur Lösung verschiedenster Synchronisationsprobleme darstellen, daß die resultierenden Formulierungen aber − insbesondere für ungeübte Benutzer − nicht immer auf Anhieb verständlich sind.

Hinzu kommt, daß es sich nicht immer vermeiden läßt, daß ein bestimmter Teilausdruck mehrfach in einem Ausdruck auftritt, was je nach Komplexität des Teilausdrucks schnell zu unübersichtlichen und schwer pflegbaren Ausdrücken führen kann.

Aus diesen Gründen ist es sinnvoll, die folgenden Abstraktionsmechanismen einzuführen.

8.1 Makros

Mit Hilfe eines einfachen Textersetzungsmechanismus, wie er von Makroprozessoren bekannt ist, kann man die Lösung eines bestimmten (Teil-)Problems zentral an *einer* Stelle formulieren und anschließend an *vielen* Stellen wiederverwenden. Außerdem ist es möglich, daß Makros für komplizierte Teilausdrücke (wie z. B. für eine m-aus-n-Verzweigung) von einem „Experten“ bereitgestellt und dann von einem „Laien“ ohne Kenntnis ihrer internen Struktur benutzt werden können.

Einfache Makros

Beispiele für einfache Makrodefinitionen sind

$$opt(x) = x \mid \lambda$$

oder

$$excl(x, y) = (x \mid y)*.$$

Auf der linken Seite des Gleichheitszeichens steht jeweils ein Bezeichner (der *Name* des Makros), gefolgt von einer optionalen Liste *formaler Parameter*. Auf der rechten Seite steht ein beliebiger Ausdruck, der *Ersatzausdruck* des Makros.

Beim *Aufruf* eines Makros in einem Ausdruck werden die formalen Parameter im Ersatzausdruck des Makros durch die aktuellen Aufrufparameter ersetzt, bevor der resultierende Ausdruck als Ersatz für den Makroaufruf verwendet wird. Damit durch den Textersatz keine unerwarteten Effekte aufgrund von Vorrangregeln auftreten, werden sowohl die aktuellen Makroparameter als auch der resultierende Ersatzausdruck implizit geklammert. Beispielsweise wird der Ausdruck $a - opt(b \,\&\, c)$ (erwartungsgemäß) zu $a - ((b \,\&\, c) \mid \lambda)$ expandiert, und nicht etwa zu $a - b \,\&\, c \mid \lambda$, was aufgrund der Vorrangregeln äquivalent zu $((a - b) \,\&\, c) \mid \lambda$ wäre.

Der Ersatzausdruck eines Makros kann weitere Makroaufrufe enthalten, sofern auf diese Weise keine rekursiven oder zyklischen Definitionen entstehen. Makroaufrufe können beliebig verschachtelt sein, d. h. die aktuellen Parameter eines Makros können Aufrufe desselben oder anderer Makros enthalten.

Makros mit variabler Anzahl von Parametern

Um auch für Probleme wie die „beliebige Reihenfolge“ elegante Makro-Lösungen formulieren zu können, muß es möglich sein, Makros mit einer variablen Anzahl von Parametern zu definieren:

$$arbseq(x*) = \left(+\, x\right) \,\&\, \left(\mid x\right)*.$$

Hier kann der „iterierte“ formale Parameter $x*$ beim Aufruf des Makros durch beliebig viele aktuelle Parameter $A, B, \ldots$ ersetzt werden. Ein *formaler Multiplikator* $\sim y(x)$ im Ersatzausdruck eines solchen Makros wird dann beim Aufruf durch den Ausdruck $y(A) \sim y(B) \sim \ldots$ ersetzt.

8.2 Benutzerdefinierte Operatoren

Unäre und binäre Operatoren

Gelegentlich ist es vorteilhaft, wenn man für häufig verwendete „Abkürzungen“ nicht immer einen Makroaufruf formulieren muß, sondern eine äquivalente Operatorschreibweise verwenden kann. Zum Beispiel könnte man anstelle oder zusätzlich zu den Makros *opt* und *excl* auch zwei Operatoren ? (postfix) und <> (infix) definieren:

$x\,? = x \mid \lambda;$

$x <> y = (x \mid y)*.$

Auf der linken Seite des Gleichheitszeichens steht jetzt ein Ausdruck der Gestalt $\sim x$ (wenn ein unärer Präfix-Operator ~ definiert werden soll), $x \sim$ (wenn ein unärer Postfix-Operator ~ definiert werden soll) oder $x \sim y$ (wenn ein binärer Infix-Operator ~ definiert werden soll). Hierbei sind x und y beliebige Bezeichner, die *formale Operanden* genannt werden und den formalen Makroparametern von oben entsprechen, während ~ ein beliebiges Operatorsymbol darstellt.

Auf der rechten Seite des Gleichheitszeichens steht wie bei Makrodefinitionen ein beliebiger Ausdruck, der als Ersatzausdruck des Operators bezeichnet wird.

Der Ersetzungsmechanismus funktioniert ähnlich wie bei Makros. Wenn bei der syntaktischen Analyse eines Ausdrucks ein Teilausdruck der Gestalt $\sim x$, $x \sim$ oder $x \sim y$ identifiziert wird, wird dieser Teilausdruck durch den Ersatzausdruck des Operators ~ ersetzt, nachdem die formalen durch die „aktuellen“ Operanden ersetzt wurden. Ebenso wie bei Makroersetzungen werden auch hier die aktuellen Operanden sowie der gesamte Ersatzausdruck implizit geklammert.

Per Default haben alle benutzerdefinierten Postfix-/Präfix-Operatoren denselben Vorrang wie die vordefinierten Postfix-/Präfix-Operatoren (d. h. wie die Iteratoren/Quantoren). Außerdem haben alle benutzerdefinierten Infix-Operatoren denselben Vorrang, der höher ist als der aller vordefinierten binären Operatoren. Bei gleichem Vorrang wird implizit von links nach rechts geklammert. Mit Hilfe geeigneter Deklarationen können diese Regeln jedoch überschrieben werden.

Benutzerdefinierte binäre Operatoren können genauso wie die vordefinierten binären Operatoren als Multiplikatoren verwendet werden.

n-äre Operatoren

In Analogie zu Makros mit variabler Anzahl von Parametern können auch Operatoren mit einer variablen Anzahl von Operanden definiert werden. Um beispielsweise die „beliebige Reihenfolge“ mit Hilfe eines Operators : formulieren zu können, kann dieser wie folgt definiert werden:

$$: x = \left(+ x\right) \& \left(\mid x\right)*.$$

Anschließend wird dann ein Ausdruck der Gestalt $A : B : \ldots$ durch den Ausdruck $(A + B + \ldots) \,\&\, (A \mid B \mid \ldots)*$ ersetzt.

Syntaktisch gesehen, ist ein solcher n-ärer Operator sehr ähnlich zu einem binären. Der einzige Unterschied besteht darin, daß ein Ausdruck wie $A : B : \ldots$ *nicht* implizit zu $((A : B) : \ldots)$ geklammert, sondern als Ganzes betrachtet wird.

Aufgrund der Ähnlichkeit gelten jedoch dieselben Vorrangregeln wie für binäre Operatoren, und auch die Multiplikatorschreibweise ist zulässig.

8.3 Anmerkung

Obwohl die *Definition* von Makros und Operatoren mit beliebig vielen Parametern/Operanden nicht ganz trivial ist, sollte man sich dennoch vor Augen halten, daß ihre *Verwendung* in Ausdrücken wesentlich zu deren *Vereinfachung* und zur *Verbesserung ihrer Lesbarkeit* − insbesondere für ungeübte Benutzer − beiträgt. Man beachte in diesem Zusammenhang auch, daß der „Autor“ und der „Anwender“ eines Makros/Operators i. d. R. verschiedene Personen („Experte“ bzw. „Laie“) sind, und daß letztere nicht mit Begriffen wie formale Multiplikatoren, Vorrangregeln und implizite Klammerung belastet werden müssen.

8.4 Äquivalenz von Makros und Operatoren

Ein unärer, binärer oder n-ärer Operator kann prinzipiell auch als Makro mit einem bzw. zwei bzw. beliebig vielen Parametern aufgefaßt und entsprechend verwendet werden, z. B.:

?(*a*);

<> (*a*, *b*);

: (*A*, *B*, *C*, *D*).

Umgekehrt kann ein Makro mit einem, zwei oder beliebig vielen Parametern auch als unärer bzw. binärer bzw. n-ärer Operator aufgefaßt und verwendet werden:

a opt oder *opt a*;

a excl b;

A arbseq B arbseq C arbseq D.

In einem separaten Bericht wird gezeigt werden, daß Makros und Operatoren bei Verwendung einer graphischen Notation von Interaktionsausdrücken sogar rein äußerlich nicht mehr voneinander zu unterscheiden sind.

9. Alternative Notationen

Eine weitere Möglichkeit zur Verbesserung der Lesbarkeit und Verständlichkeit von Interaktionsausdrücken stellt die Verwendung einer benutzerfreundlicheren Notation dar.

9.1 Graphische Notation

Mit Hilfe von graphischen Darstellungen können insbesondere „Verzweigungen“ (Selektion, parallele Komposition, Konjunktion und Synchronisation) wesentlich übersichtlicher dargestellt werden als in der bisher verwendeten formalen Notation, weil die einzelnen „Zweige“ übereinander, statt nacheinander angeordnet werden können.

Außerdem kann man durch die konsequente Verwendung von *Symbolpaaren* (z. B. Anfang und Ende einer Verzweigung oder einer Iteration) auf zusätzliche Klammern verzichten und dadurch die Übersichtlichkeit ebenfalls deutlich erhöhen.

Ein konkreter Vorschlag für eine graphische Notation, der auch fortgeschrittene Konzepte wie Makros, benutzerdefinierte Operatoren und Quantoren (vgl. Abschnitt 10) berücksichtigt, folgt in einem separaten Bericht.

9.2 Umgangssprachliche Formulierungen

Ein anderer denkbarer Ansatz, um die Formulierung und Verständlichkeit von Interaktionsausdrücken zu verbessern, besteht darin, die Menge der zulässigen Ausführungsreihenfolgen *umgangssprachlich* zu beschreiben und mit Hilfe eines Übersetzers in äquivalente Interaktionsausdrücke zu verwandeln. Beispielsweise ist die Formulierung

„Zuerst *a*, dann beliebig oft *b*, und dann *c*“

offensichtlich äquivalent zu dem Ausdruck $a - b * - c$. Bei verschachtelten Ausdrücken kann man möglicherweise durch geeignetes *Einrücken* die gewünschte Gruppierung der Teilausdrücke zum Ausdruck bringen. Es bleibt jedoch anhand von realistischen Beispielen zu klären, ob die resultierenden Formulierungen letztlich wirklich besser verständlich sind als eine gut gestaltete graphische Darstellung.

9.3 Lineare Notation

Der Vollständigkeit halber sei an dieser Stelle eine weitere Notation von Interaktionsausdrücken erwähnt, die benötigt wird, um Ausdrücke durch Programme zu verarbeiten, in Dateien zu speichern etc.

In dieser streng *linearen* Notation werden die Multiplikatoren $\overset{n}{\sim}$ und $\underset{k=m}{\overset{n}{\sim}}$ sowie Quantoren $\underset{p}{\sim}$ (vgl. Abschnitt 10) durch die Formulierungen `~{n}`, `~[k=m]{n}` bzw. `~[p]` ersetzt (wobei `~` bzw. `~` wieder für einen beliebigen binären Operator steht). Damit ein formaler Multiplikator $\sim$ von dem zugehörigen binären Operator ~ unterschieden werden kann, wird er als `~[]` notiert.

10. Parameter und Quantoren

Obwohl Interaktionsausdrücke in der bis jetzt vorgestellten Form bereits ein sehr mächtiges Werkzeug zur Lösung von Synchronisationsproblemen darstellen, fehlt ihnen doch noch ein wesentliches Konzept, um für wirklich realistische Anwendungen einsetzbar zu sein. Als einfaches Beispiel betrachte man die in Abschnitt 7.2 vorgestellte „Leser/Schreiber-Bedingung“:

$$(\mathit{read}\,\#\mid \mathit{write})*.$$

Diese Formulierung geht implizit davon aus, daß es nur ein *einziges* Objekt gibt, auf dem die Aktivitäten *read* und *write* synchronisiert werden müssen. In der Realität hat man es aber natürlich mit einer großen, oft sogar unbekannten Anzahl von Objekten zu tun, auf denen der Zugriff jeweils *individuell* synchronisiert werden muß.

Prinzipiell kann man versuchen, dieses Problem auf einer anderen „Ebene“ zu lösen. Bei Pfadausdrücken zum Beispiel sind sowohl Aktivitäten als auch die Synchronisationsbedingungen zwischen ihnen als Teil eines Abstrakten Datentyps (ADT) definiert und werden implizit auf jedes Objekt dieses Typs separat angewandt. Auf diese Weise kann man zwar Probleme wie das Leser/Schreiber-Problem ohne weiteres lösen, es können jedoch keine Abhängigkeiten zwischen Aktivitäten verschiedener ADTs definiert werden.

Um das Problem allgemein zu lösen, werden daher im folgenden *parametrisierte Ausdrücke* und *Quantoren* eingeführt.

10.1 Parallele Komposition

Um beim Leser/Schreiber-Problem unterscheiden zu können, auf welches Objekt eine Aktivität zugreifen will, ist es naheliegend, die Aktivitäten *read* und *write* mit einem Parameter p zu versehen, der das betroffene Objekt (im Sinne eines Objekt-Identifikators) bezeichnet.

Für ein bestimmtes Objekt p_1 kann dann die Leser/Schreiber-Bedingung wie folgt spezifiziert werden:

$$(\mathit{read}(p_1)\,\#\mid \mathit{write}(p_1))*.$$

Mit Hilfe von paralleler Komposition läßt sich weiterhin für eine fest vorgegebene Menge von Objekten $p_1, \ldots, p_n$ festlegen, daß die Bedingung für jedes dieser Objekte einzeln einzuhalten ist:

$$(read(p_1)\# \mid write(p_1))^* + \ldots + (read(p_n)\# \mid write(p_n))^*.$$

Durch eine naheliegende Verallgemeinerung der Multiplikator-Schreibweise (bisher durfte ein Multiplikator-Index nur als Faktor oder Bereichsgrenze weiterer Multiplikatoren verwendet werden!) kann dies natürlich als

$$\mathop{+}_{i=1}^{n} (read(p_i)\# \mid write(p_i))\#$$

abgekürzt werden.

Unter der Annahme, daß die Menge *aller möglichen* Objekte $p_1, p_2, \ldots$ bekannt (und höchstens abzählbar) ist, könnte man (zumindest formal) einen „Grenzübergang“ für $n \to \infty$ vollziehen und somit durch die Formulierung

$$\mathop{+}_{i=1}^{\infty} (read(p_i)\# \mid write(p_i))\#$$

zum Ausdruck bringen, daß die Leser-Schreiber-Bedingung für jedes beliebige Objekt p_i separat einzuhalten ist.

Da die Menge $\{ p_1, p_2, \ldots \}$ in der Praxis jedoch meist nicht bekannt ist, verwenden wir stattdessen einfach die folgende „offene“ Multiplikator-Schreibweise:

$$\mathop{+}_{p} (read(p)\# \mid write(p))\#.$$

Indem der Wertebereich des Parameters p offengelassen wird, wird angedeutet, daß er i. d. R. weder genau bekannt noch von besonderer Bedeutung ist; p soll einfach alle potentiell in Frage kommenden Werte „durchlaufen“.

Aufgrund dieser Ähnlichkeit zu dem aus der Prädikatenlogik bekannten Allquantor $\forall$ wird ein solcher offener Multiplikator auch als *Quantor* bezeichnet.

Man beachte, daß sich ein (Quantor-)*Parameter* prinzipiell von einem (Multiplikator-)*Index* unterscheidet: Ein Index durchläuft in einer festgelegten Reihenfolge eine Teilmenge der ganzen Zahlen und kann als Faktor oder Bereichsgrenze weiterer Multiplikatoren verwendet werden. Ein Parameter hingegen „durchläuft“ in einer nicht festgelegten Reihenfolge eine nicht näher spezifizierte Menge von „Objekten“ und kann daher nur als Parameter von Aktionen (oder Aktivitäten) verwendet werden.

10.2 Selektion

Die Vorgehensweise bei der „Definition“ des Quantors $\mathop{+}_{p}$ läßt sich auch auf den Operator | übertragen:

$$\Big|_{p} x(p) = \Big|_{i=1}^{\infty} x(p_i).$$

In der Sprechweise von Abschnitt 3 wird ein Ausdruck der Gestalt $\Big|_{p} x(p)$ also ausgeführt, indem der Ausdruck $x(p)$ für *eine* bestimmte Belegung p_i des Parameters p ausgeführt wird. Dies entspricht in gewisser Weise der Bedeutung des prädikatenlogischen Existenzquantors $\exists$.

10.3 Alphabet eines Quantor-Ausdrucks

Der Begriff des *Alphabets* eines Ausdrucks x wurde im Kontext der Synchronisation (vgl. Abschnitt 3.6) informell als die Menge aller Aktionen a eingeführt, die im Ausdruck x vorkommen. So besitzt der Ausdruck $x = a - b + a - c$ beispielsweise das Alphabet $\alpha(x) = \{ a, b, c \}$.

Diese Definition des Alphabets läßt sich prinzipiell auch auf parametrisierte Ausdrücke übertragen. Beispielsweise besitzt der parametrisierte Ausdruck $x(p, q) = a(p) - b(q) + a(p) - c(p, q)$ rein formal das Alphabet $\{ a(p), b(q), c(p, q) \}$.

Um das Alphabet eines Quantor-Ausdrucks $\underset{p}{\sim} x(p)$ sinnvoll zu definieren, verwenden wir zum einen die Quantor-Definition

$$\underset{p}{\sim} x(p) = \overset{\infty}{\underset{i=1}{\sim}} x(p_i)$$

und zum anderen die offensichtliche Regel, daß sich das Alphabet eines zusammengesetzten Ausdrucks $x \sim y$ als Vereinigung der Alphabete von x und y ergibt, und erhalten somit:

$$\alpha\left(\underset{p}{\sim} x(p)\right) = \overset{\infty}{\underset{i=1}{\bigcup}} \alpha(x(p_i)) = \underset{p}{\bigcup} \alpha(x(p)).$$

$\underset{p}{\bigcup}$ bezeichne hierbei die Vereinigung über alle möglichen Werte von p.

Anschaulich bedeutet das, daß die Parameter einer Aktion quasi als Teil des Aktionsnamens betrachtet werden, so daß z. B. $a(p)$ und $a(q)$ für $p \neq q$ formal verschiedene Aktionen sind.

10.4 Synchronisation

Obwohl in praktischen Anwendungen (vgl. Abschnitt 10.7) meist nur die Quantoren $\underset{p}{+}$ und $\underset{p}{\mid}$ benötigt werden, läßt sich nun − nach der Erweiterung der Alphabet-Definition − auch der Operator @ zu einem Quantor verallgemeinern:

$$\underset{p}{@}\, x(p) = \overset{\infty}{\underset{i=1}{@}}\, x(p_i).$$

Für einen so definierten Ausdruck $\underset{p}{@}\, x(p)$ gilt hierbei (vgl. auch Abschnitt 3.6):

Wenn alle Aktionen des Teilausdrucks $x(p)$ wirklich vom Parameter p abhängen, dann sind die Alphabete der Teilausdrücke $x(p_1), x(p_2), \ldots$ disjunkt, und die Synchronisation $\underset{p}{@}\, x(p)$ daher äquivalent zur parallelen Komposition $\underset{p}{+}\, x(p)$.

Andernfalls, d. h. wenn der Teilausdruck $x(p)$ Aktionen enthält, die nicht vom Parameter p abhängen, dann sind diese Aktionen allen Zweigen der Synchronisation gemeinsam, und folglich müssen sich alle Zweige zur Ausführung dieser Aktionen synchronisieren. Dies kann allerdings sehr schnell zu mehr oder weniger sinnlosen Ausdrücken wie z. B. dem folgenden führen:

$$\underset{p}{@}\, (a(p) - b - c(p)),$$

der faktisch äquivalent zu dem Ausdruck $\underset{p}{@}\, a(p)$ ist, weil kein Zweig der Synchronisation über die Ausführung von $a(p)$ hinaus kommt. (Da b von allen Zweigen gemeinsam ausgeführt werden muß, kann es erst ausgeführt werden, nachdem *alle* Zweige $a(p)$ ausgeführt haben. Da es jedoch potientiell unendlich viele Zweige gibt, tritt dieser Zustand nie ein.)

Anders verhält es sich jedoch mit dem Ausdruck

$$\underset{p}{@}\, (a(p)^* - b - c(p)^*),$$

bei dem die Ausführung der Aktion $a(p)$ in jedem Zweig optional ist und daher jeder Zweig sofort (oder auch nach einigen Ausführungen von $a(p)$) zur Ausführung der gemeinsamen Aktion b bereit ist. Der Ausdruck erlaubt also Ausführungsreihenfolgen der Gestalt AbC, wobei A bzw. C für eine beliebige Folge von $a(p)$'s bzw. $c(p)$'s steht.

10.5 Konjunktion

Rein formal läßt sich auch die Konjunktion zu einem Quantor verallgemeinern:

$$\mathop{\&}_{p} x(p) = \mathop{\&}_{i=1}^{\infty} x(p_i).$$

Der praktische Nutzen einer solchen Definition ist jedoch noch weitaus geringer als der des Synchronisationsquantors: Da eine Konjunktion nur Aktionen zuläßt, die allen Zweigen gemeinsam sind, sind Aktionen, die wirklich vom Parameter p abhängen, nie zulässig!

10.6 Sequentielle Komposition

Die sequentielle Komposition nimmt unter den binären Operatoren insofern eine Sonderstellung ein, als sie *nicht kommutativ* ist. Aus diesem Grund läßt sie sich nicht ohne weiteres zu einem Quantor verallgemeinern, weil die Bedeutung der Definition

$$\mathop{—}_{p} x(p) = \mathop{—}_{i=1}^{\infty} x(p_i) = x(p_1) - x(p_2) - \ldots$$

von der Anordnung der p_i abhinge. Da die Menge der p_i in der Praxis jedoch nicht bekannt ist, kann man natürlich auch keine Annahmen über ihre Anordnung machen.

Alternativ könnte man daher den Ausdruck $\mathop{—}_{p} x(p)$ als Verallgemeinerung der „beliebigen Reihenfolge“ (vgl. Abschnitt 7) auf unendlich viele Komponenten auffassen, d. h. definieren, daß die Ausdrücke $x(p_1)$, $x(p_2)$, … sequentiell *in beliebiger Reihenfolge* ausgeführt werden dürfen.

Dies läßt sich jedoch auch mit den bereits definierten Quantoren wie folgt ausdrücken:

$$\left(\mathop{+}_{p} x(p)\right) \& \left(\mathop{|}_{p} x(p)\right)^{*},$$

d. h. die Schreibweise $\mathop{—}_{p} x(p)$ wäre „redundant“.

Außerdem wären dann die Definitionen für $x - y$ bzw. $\mathop{—}_{k=m}^{n} x_k$ auf der einen Seite und $\mathop{—}_{p} x(p)$ auf der anderen Seite nicht mehr konsistent: Beim ersten und zweiten Ausdruck ist die Ausführungsreihenfolge der Teilausdrücke der sequentiellen Komposition vorgeschrieben, während sie beim dritten Ausdruck beliebig wäre. Um diese Inkonsistenz zu beseitigen, müßte man konsequenterweise die Definition der sequentiellen Komposition durch die der „beliebigen Reihenfolge“ ersetzen (was den „Vorteil“ hätte, daß dann alle binären Operatoren kommutativ wären). Allerdings gäbe es dann keine Möglichkeit mehr, Reihenfolgebeziehungen mit Interaktionsausdrücken zu beschreiben!

Aus diesen Gründen verzichten wir auf eine Quantor-Definition der sequentiellen Komposition und nehmen die dadurch entstehende „Lücke“ im Modell der Interaktionsausdrücke in Kauf.

10.7 Beispiele

Ein speisender Philosoph

Mit Hilfe von parametrisierten Ausdrücken und Quantoren kann der Lebenszyklus eines speisenden Philosophen nun wie folgt neu beschrieben werden.

Zunächst erhält jede der Aktivitäten *think*, *enter*, *eat* und *leave* einen Parameter p zur Bezeichnung eines Philosophen. Weiterhin werden die Aktivitäten *getleft* und *getright* sowie *putleft* und *putright* durch die neuen Aktivitäten $get(p, f)$ und $put(p, f)$ (Philosoph p nimmt Gabel f bzw. legt sie zurück) ersetzt. Schließlich erhält die Aktivität *eat* neben dem Parameter p zwei weitere Parameter l und r, die die beiden Gabeln bezeichnen, mit denen der Philosoph p gerade ißt.

Die ersten beiden Schritte in einer Iteration des Zyklus sind nun $think(p)$ und $enter(p)$:

$$think(p) - enter(p).$$

Anschließend muß ausgedrückt werden, daß der Philosoph zwei Gabeln l und r vom Tisch nimmt (daß es die beiden zu seiner linken bzw. rechten sind, ist zunächst unerheblich), d. h. den Ausdruck $get(p, l) + get(p, r)$ für eine beliebige Belegung der Parameter l und r ausführt:

$$\Big|_{l,r} [get(p, l) + get(p, r)] = \Big|_{l} \Big|_{r} [get(p, l) + get(p, r)].$$

Anschließend ißt der Philosoph mit *diesen beiden* Gabeln, bevor er sie wieder auf den Tisch zurücklegt, d. h. die Aktivitäten $eat(p, l, r)$ sowie $put(p, l)$ und $put(p, r)$ müssen ebenfalls im „Scope“ des Quantors $\Big|_{l,r}$ ausgeführt werden:

$$\Big|_{l,r} \{ [get(p, l) + get(p, r)] - eat(p, l, r) - [put(p, l) + put(p, r)] \}.$$

Schließlich verläßt der Philosoph den Raum wieder und der Zyklus beginnt von vorne:

$$\Big(think(p) - enter(p) - \Big|_{l,r} \{ [get(p, l) + get(p, r)] - eat(p, l, r) - [put(p, l) + put(p, r)] \} - leave(p) \Big)^*.$$

Man beachte, daß die Auswahl eines Zweiges der Selektion $\Big|_{l,r}$, d. h. die „Bindung“ der Variablen l und r an bestimmte konkrete Werte, für jede Ausführung dieses Quantor-Ausdrucks erneut vorgenommen wird. Das bedeutet praktisch, daß der Philosoph p in jedem Zyklus mit anderen Gabeln essen kann.

Wenn man dies vermeiden möchte, kann man die Selektion wie folgt nach außen ziehen:

$$\Big|_{l,r} \{ think(p) - enter(p) - [get(p, l) + get(p, r)] - eat(p, l, r) - [put(p, l) + put(p, r)] - leave(p) \}^*$$

Jetzt wird die Selektion im „ganzen Leben“ des Philosophen nur einmal ausgeführt, was zur Folge hat, daß die einmal gewählten Werte für l und r anschließend nicht mehr verändert werden. Man beachte in diesem Zusammenhang jedoch, daß die tatsächliche Wahl der Werte nach dem Prinzip der vorausschauenden Auswahl zunächst aufgeschoben werden muß, weil die Ausführung der Aktivitäten *think* und *enter* noch keine Wahl für l und r erlaubt. Erst bei Ausführung der Aktivitäten *get* kann die Wahl dann rückwirkend getroffen werden. Für eine Implementierung von Interaktionsausdrücken bedeutet dies, daß sie zunächst − zumindest „virtuell“ − eine potientiell unendlich große Zahl von Alternativen verfolgen muß! Eine Möglichkeit, dies „reell“, d. h. mit einer endlichen Kapazität von Ressourcen, und auch möglichst effizient zu realisieren, wird in einem separaten Bericht beschrieben.

Eine Menge von speisenden Philosophen

Unabhängig davon, wie der Lebenszyklus eines einzelnen Philosophen nun im Detail formuliert wird, kann man ihn über alle Philosophen p quantifizieren, um auszudrücken, daß jeder Philosoph nach demselben Schema lebt:

$$\mathop{+}_{p} \{\, think(p) - enter(p) - \ldots - leave(p) \,\}^*.$$

Allerdings sind nun gewisse „globale Integritätsbedingungen“ zu beachten, damit das Zusammenleben der Philosophen harmonisch abläuft: Jede Gabel f kann zu jedem Zeitpunkt von höchstens einem Philosophen p benutzt werden, d. h. nachdem irgendein Philosoph p die Gabel f genommen hat, muß dieser sie erst wieder zurücklegen, bevor sie von einem anderen (oder auch demselben) Philosophen erneut genommen werden kann:

$$\mathop{+}_{f}\left(\mathop{|}_{p}[get(p,\ f) - put(p,\ f)]\right)^*.$$

Man achte auch hier wieder sorgfältig auf den korrekten Scope der Quantoren!

Durch Verknüpfung dieses „gabelorientierten“ Ausdrucks mit dem obigen „philosophenorientierten“ Ausdruck kann man die Einhaltung der gewünschten Integritätsbedingung im Leben der Philosophen erzwingen:

$$\mathop{+}_{p} \{\, think(p) - enter(p) - \ldots - leave(p) \,\}^* \mathbin{@} \mathop{+}_{f}\left(\mathop{|}_{p}[get(p,\ f) - put(p,\ f)]\right)^*.$$

Man beachte hier wiederum die Wirkungsweise des Synchronisationsoperators @: Da der gabelorientierte Teilausdruck keine Aussagen über die Aktivitäten *think*, *enter* usw. macht, erlaubt er deren Ausführung zu jedem Zeitpunkt.

Um, wie in der ursprünglichen „Aufgabenstellung“ von Dijkstra vorgesehen, a priori genau festzulegen, welcher Philosoph mit welchen Gabeln essen darf, könnte man versuchen, wie folgt vorzugehen:

$$get(p_1,\ f_1)^* + get(p_1,\ f_2)^* + get(p_2,\ f_2)^* + get(p_2,\ f_3)^* + \ldots.$$

Hierbei seien $p_1, p_2, \ldots$ sowie $f_1, f_2, \ldots$ konkrete Philosophen- bzw. Gabel-„Identifier“. Dieser Ausdruck erlaubt also dem Philosophen p_1 explizit, die Gabeln f_1 und f_2 zu nehmen, usw. Verknüpft man diesen Ausdruck jedoch ebenfalls mittels Synchronisation mit den obigen beiden Ausdrücken, so erweist er sich als wirkungslos: Da er beispielsweise keine Aussage über die Aktivität $get(p_1,\ f_3)$ macht, erlaubt er deren Ausführung zu jedem Zeitpunkt, d. h. er ist prinzipiell nicht in der Lage, die Ausführung bestimmter Aktivitäten explizit zu *verbieten*. (Eine konjunktive Verknüpfung scheidet natürlich ebenfalls aus, weil der resultierende Gesamtausdruck dann unerfülbar wäre, vgl. Abschnitt 3.5.)

Obwohl das Problem auf den ersten Blick unlösbar erscheint − Interaktionsausdrücke legen ja prinzipiell *zulässige* Ausführungsreihenfolgen fest −, kann man den gewünschten Effekt doch mit einem kleinen Trick erzielen. Damit ein Ausdruck x (der mit einem anderen Ausdruck y mittels Synchronisation verknüpft wird) eine bestimmte Aktion a verbieten kann, muß er diese Aktion auf jeden Fall enthalten, allerdings in einer „Position“, in der sie nie zulässig ist. Im konkreten Fall könnte man den obigen Ausdruck wie folgt erweitern:

$$get(p_1,\ f_1)^* + get(p_1,\ f_2)^* + get(p_2,\ f_2)^* + get(p_2,\ f_3)^* + \ldots \mathbin{\&} \mathop{+}_{p,\ f} get(p,\ f)^*,$$

um sicherzustellen, daß er alle möglichen $get(p,\ f)$-Aktivitäten enthält. Aufgrund der konjunktiven Verknüpfung läßt dieser Ausdruck jedoch nur die Ausführung der explizit genannten Aktivitäten $get(p_1,\ f_1)$, $get(p_1,\ f_2)$ usw. zu und verbietet damit indirekt die Ausführung der anderen!

Nach der Lösung dieses Problems verbleibt eine letzte Schwierigkeit. Wenn alle Philosophen gleichzeitig am Tisch sitzen und jeder erfolgreich *eine* Gabel genommen hat, kann keiner mehr eine zweite

Gabel nehmen, da es nur soviele Gabeln wie Philosophen gibt. Da aber auch keiner der Philosophen „in der Lage“ ist, seine Gabel wieder zurückzulegen, bevor er mit Hilfe einer zweiten Gabel gegessen hat, ist ein *Deadlock* entstanden. Eine einfache Abhilfe für dieses Problem besteht in der Einführung eines „Türstehers“, der darauf achtet, daß sich zu jedem Zeitpunkt höchstens $n-1$ Philosophen im Raum befinden, wenn n die Gesamtzahl der Philosophen bezeichnet. Da aufgrund der begrenzten Gabelzahl ohnehin nicht alle n gleichzeitig essen können, stellt dies keine wesentliche Einschränkung für die „Praxis“ dar. Der folgende Ausdruck leistet dies:

$$\overset{n-1}{+}\left(\underset{p}{|}\,[enter(p) - leave(p)]\right)^*.$$

Zusammenfassend kann man das Problem der speisenden Philosophen (für $n = 5$) unter Zuhilfenahme einiger Makros wie folgt lösen:

$$phil(p) = \Big(think(p) - enter(p) - \underset{l,r}{|}\,\{\,[get(p, l) + get(p, r)] - eat(p, l, r) - [put(p, l) + put(p, r)]\,\} - leave(p)\Big)^*;$$

$$guard(n) = \overset{n-1}{+}\left(\underset{p}{|}\,[enter(p) - leave(p)]\right)^*;$$

$$fork(f) = \left(\underset{p}{|}\,[get(p, f) - put(p, f)]\right)^*;$$

$$assign(p, l, r) = get(p, l)^* + get(p, r)^*;$$

$$assign5 = assign(p1, f1, f2) + assign(p2, f2, f3) + assign(p3, f3, f4) + assign(p4, f4, f5) + assign(p5, f5, f1) \,\&\, \underset{p,\,f}{+}\, get(p, f)^*;$$

$$\underset{p}{+}\, phil(p) \;@\; guard(5) \;@\; \underset{f}{+}\, fork(f) \;@\; assign5.$$

Alternativ könnte man die Funktionalität von $phil(p)$ und $assign(p, l, r)$ auch wie folgt kombinieren:

$$phil(p, l, r) = (think(p) - enter(p) - [get(p, l) + get(p, r)] - eat(p, l, r) - [put(p, l) + put(p, r)] - leave(p))^*;$$

$$phil5 = phil(p1, f1, f2) + phil(p2, f2, f3) + phil(p3, f3, f4) + phil(p4, f4, f5) + phil(p5, f5, f1);$$

und den Gesamtausdruck dann wie folgt formulieren:

$$phil5 \;@\; guard(5) \;@\; \underset{f}{+}\, fork(f).$$

10.8 Zusammenfassung

Parametrisierte Ausdrücke und Quantoren stellen eine wichtige Verallgemeinerung elementarer Ausdrücke und Operatoren dar. Prinzipiell lassen sich alle kommutativen binären Operatoren zu Quantoren verallgemeinern, für praktische Anwendungen sind jedoch meist nur die parallele Komposition und die Selektion relevant, die den prädikatenlogischen Quantoren $\forall$ bzw. $\exists$ entsprechen.