Spezifikation der dynamischen Semantik von Basic-Lotos

Diplomarbeit

Thomas Kühne

Fachgebiet Praktische Informatik
Prof. Dr. W. Henhapl

Betreuer: Christoph Thies

29. September 1992

Institut für Systemarchitektur
Fachbereich Informatik
Technische Hochschule Darmstadt

Werden und Sein
über
Nehmen und Haben

oder

Der Weg ist das Ziel
Das Ziel ist die Weisheit

TK

Zusammenfassung

Thema der vorliegenden Diplomarbeit ist die Spezifikation der dynamischen Semantik von Basic-Lotos. Dieser Teil einer Sprachbeschreibung ermöglicht die automatische Generierung eines Interpretierers oder Simulators. Der von Centaur, ein System zur Generierung sprachspezifischer Programmierumgebungen, aus der erarbeiteten Spezifikation generierte Simulator dient als Prototyp für den im Rahmen von Gipe II entstehenden Lotos Simulator. Zunächst erfolgen Einführungen in die Konzepte von Lotos, dessen Simulation und Centaur. Anschließend werden der Entwurf, die Kommentierung der Spezifikation und Vorschläge zur Verbesserung bzw. Erweiterung der Spezifikation vorgestellt. Es folgt eine Bewertung von Centaur bezüglich der verwendeten Formalismen und Konzepte. Abschließend wird die verwendete Spezifikationsmethodik von Centaur mit der Idee des Tsk Formalismus verglichen. Handlungsbedarf und -aufwand für das direkte Spezifizieren eines Simulators mit Centaur einerseits und die Abbildung auf einen Formalismus zur Beschreibung paralleler Prozesse mit vorhandener Simulationsumgebung andererseits werden abgeschätzt.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einleitung

Eines der Hauptinteressen der Informatik ist die automatische Generierung von sprachspezifischen Programmierumgebungen aus formalen Sprachbeschreibungen. Während die automatische Generierung von Parsern gut verstanden ist, eröffnet die Beschreibung und Generierung der restlichen Komponenten ein großes Feld der Forschung. Insbesondere die generierten Komponenten Interpretierer oder Compiler besitzen heute höchstens die Qualität eines Prototypen. Ergebnis des GIPE[1] Projekts war CENTAUR, ein System zur Generierung sprachspezifischer Programmierumgebungen. Ziel von Workpackage 6 des ESPRIT-Projekts GIPE II ist die Bewertung von CENTAUR. Als Fallbeispiel zur Beurteilung dient hier LOTOS eine Sprache zur Beschreibung paralleler Prozesse. Ein Interpretierer für LOTOS ist ein Simulator für das durch die LOTOS Spezifikation beschriebene *structured-labelled-transition-system*. CENTAUR bietet für die Generierung eines Simulators erforderliche Spezifikation der dynamischen Semantik zwei verschiedene Ansätze:

- ASF - Algebraic Specification Formalism
- TYPOL - basierend auf NATURAL SEMANTICS

Um eine vergleichende Bewertung zu erhalten, wird der LOTOS Simulator zweimal spezifiziert, mit ASF von PTT-Research in Holland und mit TYPOL von der TH-Darmstadt. Die vorliegende Spezifikation für den Prototypen des geplanten LOTOS Simulators beschreibt BASIC-LOTOS, ein Subset von LOTOS. LOTOS ist zusammengesetzt aus einem Prozeßteil und einem Datenteil, die sich vollständig separieren lassen. BASIC-LOTOS entspricht dem Prozeßteil und ist eine in sich abgeschlossene, mächtige Sprache zur Beschreibung von parallelen Prozessen. Der Simulator in CENTAUR zerfällt in eine Spezifikation der dynamischen Semantik von LOTOS in TYPOL und eine Beschreibung der Funktionalität der Benutzeroberfläche. Eine TYPOL Spezifikation besteht aus Inferenzregeln sowohl über den syntaktischen Aufbau einer Sprache als auch über den Inhalt von dynamischen Datenstrukturen. Die Funktionalität der Benutzeroberfläche wird (noch) in LISP festgelegt. Ein anderer Ansatz zu Generierung eines Simulators besteht darin, die Sprachsemantik auf einen Formalismus zu Beschreibung paralleler Prozesse abzubilden, der bereits eine konfigurierbare Simulationsumgebung bereitstellt. Aspekte der Abbildung LOTOS $\rightarrow$ TSK und ein Vergleich mit der Vorgehensweise innerhalb CENTAUR werden im letzten Kapitel behandelt.

[1] Generation of Interactive Programming Environments

2 LOTOS

Um die speziellen Probleme des Entwurfs und der Spezifikation eines BASIC-LOTOS-Simulators nachvollziehbar zu machen, wird zuerst die zugrunde liegende Sprache vorgestellt. Zur Beschreibung der Herkunft und Eigenschaften von LOTOS folgt anschließend eine Erläuterung der, sich daraus ergebenden, Aspekte eines LOTOS-Simulators.

2.1 Sprache

Zur Beschreibung informationsverarbeitender Systeme ist eine natürliche Sprache wie Deutsch oder Englisch wegen mangelnder Präzision ungeeignet. Bei Verwendung einer geeigneten formalen Spezifikationssprache lassen sich Systeme beschreiben, die vollständig, konsistent und eindeutig[2] sind. Eine solche Spezifikationssprache, die durch einen fundierten mathematischen Unterbau mathematische Analysen und Korrektheitsbeweise zuläßt, nennt man FDT[3]. Sie unterscheidet sich von einer Implementierungssprache durch einen höheren Abstraktionsgrad und das zugrunde liegende mathematische Modell. Somit ist es zum einem möglich, das Verhalten eines Systemes und nicht dessen Realisierung zu beschreiben, und zum anderen erlaubt das mathematische Fundament Korrektheitsbeweise und die Anwendung von Werkzeugen wie Verifikatoren.

Ein Vertreter dieser Sprachklasse ist LOTOS[4]. LOTOS wurde aus folgenden Gründen zur Bewertung von CENTAUR ausgewählt:

- LOTOS ist komplex genug für einen realistischen Test von CENTAUR.
- Eine parallele Sprache benötigt möglicherweise besondere Behandlung.
- LOTOS hat industrielle Anwendungen.
- Als Spezifikationssprache fordert LOTOS die Entwicklung von Werkzeugen wie Transformatoren und Verifikatoren heraus, die möglicherweise spezielle Anforderungen an CENTAUR stellen.
- LOTOS ist standardisiert. Syntax und Semantik sind formal festgelegt in [ISO87].

Die Entwicklung von LOTOS (1981-1988) zielte auf die formale Beschreibung von verteilten parallelen Systemen, insbesondere der OSI[5] Architektur, ein Beispiel für den Einsatz in der Telekommunikation. Die Grundidee von LOTOS ist die Beschreibung von Systemen durch die zeitliche Abfolge von Ereignissen im beobachtbaren Verhalten. In dieser Hinsicht unterscheidet sich LOTOS von dem der Historie nach sehr verwandten ESTELLE. Während ESTELLE auf dem Modell der erweiterten endlichen Automaten basiert, hat LOTOS seine Wurzeln in der Prozeßalgebra. Der Unterschied besteht in der Definition des Zustands einer interpretierten Spezifikation. In Estelle besteht der Zustand aus den Variableninhalten der Prozesse und deren Kommunikationsstruktur. In LOTOS wird der Zustand durch die bereits erfolgten Ereignisse definiert. Die Tatsache, daß die beiden Spezifikationen in Abbildung 2 in LOTOS unterschiedliches Verhalten realisieren verdeutlicht den Unterschied zur Interpretation als Stringakzeptoren. Die entsprechenden, endlichen Automaten sind Trace-Äquivalent, akzeptieren also die selbe Sprache, nämlich $ab|ac$. In [Mil80] wird gezeigt, daß die *Observation equivalence* von LOTOS feiner ist als die *Trace equivalence* der Automatentheorie. Der spezielle Unterschied im Verhalten der LOTOS-Terme wird noch im Abschnitt Simulation erläutert.

[2] "Eindeutig" kann in diesem Zusammenhang auch "eindeutig Mehrdeutig" bedeuten. Durch nichtdeterministische Spezifikationsmethoden kann Mehrdeutigkeit exakt verwendet werden.

[3] Formal Description Technique

[4] Language of Temporal Ordering Specification

[5] Open Systems Interconnection

Zur Beschreibung von Systemverhalten stellt LOTOS im wesentlichen drei Konzepte bereit:

- Der Prozeß als Systemkomponente
- Die Kommunikation als Ereignissträger
- Der abstrakte Datentyp zur Vereinbarung von Daten und Operationen

Hieraus ergibt sich die orthogonale Zweiteilung von LOTOS in

Prozeßteil Beschreibung der Abfolge der Ereignisse, basierend auf Milner's CCS[6] und Hoar's CSP[7]. Ergibt als LOTOS-Subset BASIC-LOTOS .

Datenteil Beschreibung der Struktur der Ereignisse, basierend auf ACT ONE von der Technischen Universität Berlin, einer Sprache zur Beschreibung von abstrakten Datentypen.

Eine LOTOS-Spezifikation beschreibt eine Hierachie von Prozessen. Das beobachtbare Verhalten des obersten Prozesses entspricht dem äußeren Systemverhalten. Dieses ist zusammengesetzt aus dem i.a. parallelen beobachtbaren Verhalten der Unterprozesse, das durch den darüberliegenden Prozeß versteckt wird. In Abbildung 1 stellen die Rechtecke Prozesse dar, die mit einer festgelegten Menge von Ereignissen mit ihrer Umwelt kommunizieren können. Die Ereignisnamen, die ein Prozeß als Schnittstelle an seine Umgebung veröffentlicht, heißen Gates. Die beiden Prozesse werden durch einen Operator verknüpft, der festlegt, in welchen Ereignissen, in diesem Fall b, die Prozesse untereinander kommunizieren. Die Gesamtheit Operator und Prozesse stellt einen neuen Prozeß dar, der seiner Umwelt ein neues Verhalten anbietet. Der neue Prozeß entspricht, in dem Verhalten der unabhängigen Gates a, c und d, jeweils dem Einzelverhalten der Prozesse P_1 und P_2. Die Synchronisation im Ereignis b erzeugt ein neues Verhalten, die Kombination der Einzelverhalten von P_1 und P_2.

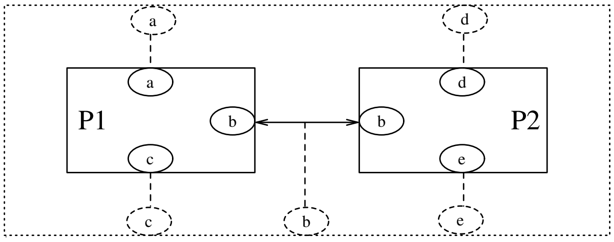

Abbildung 1: Zwei kommunizierende Prozesse

Die Schnittstelle der Prozesse gibt nur potentielle Ereignisse an. Die tatsächliche Bereitschaft zur Kommunikation hängt vom aktuellem Verhalten des Prozesses ab. Dieses wird durch eine sogenannte Behaviour-Expression definiert. Um in einem Prozeß ein Ereignis bereitzustellen, bedient man sich des Action-prefix. Dieses Konstrukt besteht aus einem Ereignisnamen und einer Behaviour-Expression und stellt wieder eine Behaviour-Expression dar.

```
a; stop
```

Das hier verwendete `stop` steht für den inaktiven Prozeß. Der angegebene Prozeß kann also einmal das Ereignis a anbieten und dann an keinem weiteren Ereignis mehr teilnehmen. Ein elementarer Operator ist der sogenannte Choice.

```
a; c; stop [] b; d; stop
```

meint sowohl a und b werden als Ereignisse angeboten, aber nur wahlweise exklusiv. Weiterhin bedeutet die Teilnahme an einem Ereignis das Verschwinden des Verhaltens der Alternative. Das heißt, nach einem Ereignis a ist nur noch das Ereignis c möglich. Die vorgestellten Operatoren Action-prefix und Choice sind dem Prinzip nach ausreichend, um das Verhalten der anderen LOTOS-Operatoren nachzuahmen. Natürlich sind aber die restlichen LOTOS-Operatoren zur Strukturierung einer Spezifikation und zur Verbesserung der Problemorientiertheit unerläßlich. Tabelle 1 bietet eine Übersicht der BASIC-LOTOS-Operatoren.

[6] Calculus of Communicating Systems
[7] Communicating Sequential Processes

Mit Hinzunahme der Prozeßinstantiierung in die Behaviour-Expressions läßt sich bereits ein sinnvolles Beispiel angeben:

```
process GetMoney[Nickel, Dime, Back, GotMoney, NickelBack]:=
  Nickel; (Nickel; GotMoney; GetMoney[Nickel, Dime, Back, GotMoney, NickelBack]
           []
           Back; NickelBack; GetMoney[Nickel, Dime, Back, GotMoney, NickelBack]
          )
  []
  Dime; GotMoney; GetMoney[Nickel, Dime, Back, GotMoney, NickelBack]
endproc
```

Dieser Prozeß könnte als Teilprozeß eines Getränkeautomaten dienen. Die Ereignisse *Nickel*, *Dime*, *Back* sind Eingaben der Umwelt an den Prozeß, während *GotMoney* und *NickelBack* Ausgaben des Prozesses an die Umwelt sind. Bietet die Umwelt das Ereignis *Dime* an so antwortet der Prozeß mit dem Ereignis *GotMoney* und steht nach Konsumierung dieses Ereignisses mit seinem ursprünglichen Verhalten wieder zu Verfügung. Teilnahme an dem Ereignis *Nickel* unterbindet die Möglichkeit des Ereignisses *Dime* erlaubt aber neben Einwurf eines weiteren Nickels die Rückgabe des bereits eingeworfenen Nickels. Je nachdem heißt es entweder *GotMoney* oder *NickelBack*.

Name	Syntax	Semantik
inaction	stop	nimmt an keinem Ereignis teil
termination	exit	beendet einen Prozeß (exit $\equiv \delta$[8] ; stop)
choice	B_1 [] B_2	exklusive Wahl zwischen den Ereignissen von B_1 und B_2
action-prefix	g; B	Ereignis g wird angeboten, danach folgt Verhalten B
synchronization	B_1 || B_2	B_1 und B_2 können nur an einem Ereignis teilnehmen, das sie gleichzeitig anbieten
interleave	B_1 ||| B_2	B_1 und B_2 bieten unabhängig von einander ihre Ereignisse an
in-parallel-over	B_1 |[$g_1,\ldots,g_n$]| B_2	B_1 und B_2 müssen sich in den Ereignissen δ + $g_1,\ldots,g_n$ synchronisieren und bieten andere unabhängig voneinander an
hiding	hide $g_1,\ldots,g_n$ in B	alle Ereignise $\in g_1,\ldots,g_n$ in B werden in i[9] umbenannt
instantiation	p[$g_1,\ldots,g_n$]	Erzeugung einer Prozeßinstanz von p mit den aktuellen Gates $g_1,\ldots,g_n$
disabling	B_1 [> B_2	Ereignisse von B_1 werden solange angeboten bis, entsprechend einer Unterbrechung, ein Ereignis von B_2 ausgewählt wird. Danach existiert nur noch B_2
enabling	B_1 >> B_2	Nachdem Prozeß B_1 terminiert hat[10], bietet B_2, entsprechend einer Hintereinanderausführung, seine Ereignisse an

Tabelle 1: Basic-Lotos Operatoren.

[8] das δ-Ereignis, auch "tick" genannt, steht für die Prozeßterminierung

[9] internes Ereignis; benötigt keinen Kommunikationspartner $\rightsquigarrow$ internes Verhalten

[10] Das externe δ-Ereignis des terminierenden Prozesses wird durch den `enable`-Operator in ein internes δ-Ereignis umbenannt

Eine Besonderheit des parallel-Operators ist die Mehrwegesynchronisation. Im Prozeß

```
P1 || P2
```

müssen sich `P1` und `P2` in allen Ereignissen synchronisieren. Bestehen die Prozesse `P1` und `P2` nun aus Unterprozessen der Art

```
Q1 || Q2,
```

so bedeutet dies, daß sich ingesamt vier Prozesse in ihren Ereignissen synchronisieren müssen. Demnach bewirkt die Wahl eines Ereignisses des Hauptprozesses die Verhaltensänderung von insgesamt vier Prozessen. Eine Änderung des Verhaltenausdrucks durch Ereigniswahl ist also nicht auf einen kleinen Teilbereich des Ausdrucks lokalisiert, sondern kann sich über den gesamten Ausdruck erstrecken. Die Anforderung des parallel-Operators an beide Operanden, ein spezifisches Ereignis gleichzeitig bereitzustellen, läßt sich nicht nur im Sinne der zeitlichen Synchronisation, das Warten des Schnelleren auf den Langsameren, interpretieren. Hierdurch wird auch ein Constraint etabliert. Wenn ein Operand die erlaubten und der andere die augenblicklich möglichen Ereignisse liefert, ist das Ergebnis des parallel-Operators die Menge der aktuell legalen Ereignisse. In diesem Sinn wird zwischen Zustands- und Constraintorientierten Spezifikationen unterschieden.

Als Beispiel einer BASIC-LOTOS Spezifikation findet sich im Anhang die Beschreibung eines Tokenrings.

2.2 Simulation

Ein Interpretierer für LOTOS-Spezifikationen dient

- dem Sprachentwickler, seine Erwartungen zu überprüfen,
- einem Implementierer, das Verständnis der Spezifikation zu erleichtern und
- einem Tester, die Implementierung gegen die Spezifikation zu prüfen.

In [Hog89] heißt es:

> Die Erfahrung hat gezeigt, daß es fast unmöglich ist, eine korrekte Spezifikation realistischen Ausmaßes ohne die Hilfe von Werkzeugen zu erstellen.

Ein Simulator für LOTOS-Spezifikationen unterscheidet sich grundlegend von einem Interpretierer oder Debugger für eine imperative Sprache wie z.B. Pascal. Durch eine LOTOS Spezifikation wird ein *structured labeled transition system* definiert. *transition system* meint die Übergänge von einem Verhaltem zu einem anderem. Es ist *labeled*, weil die Übergänge mit benannten Ereignissen markiert sind. Durch die Möglichkeit, über diese Ereignisse Daten zu transportieren, wird das System *structured*. Ein LOTOS Interpretierer ist demnach ein Simulator für das beschriebene System. Wie oben erläutert, kommunizieren Prozesse mit ihrer Umwelt. Die Umwelt der Spezifikation ist der Benutzer. Das bedeutet, der Benutzer muß eines[11] der aktuell möglichen Ereignisse auswählen und erhält dadurch ein neues Verhalten der Spezifikation. Dies gilt ohne Einschränkungen für die beobachtbaren Ereignisse der Spezifikation, aber auch die versteckten Ereignisse der Unterprozesse *müssen* durch den Benutzer ausgelöst werden, da es i.a. keine sinnvolle Strategie gibt, automatisch die Reihenfolge der möglichen Ereignisse zu bestimmen. Bei einer imperativen Sprache beantwortet der Benutzer eventuelle Eingabeanforderungen und im Falle des Debuggens kann er die schrittweise Ausführung überwachen, die jedoch an sich autark abläuft. Bei der Ausführung einer LOTOS Spezifikation bestimmt der Benutzer durch seine Ereignisauswahl das letztendliche Resultat. In diesem Zusammenhang steht auch die Auflösung von Nichtdeterminismus in LOTOS. Die beiden Behaviour-Expressions in Abbildung 2

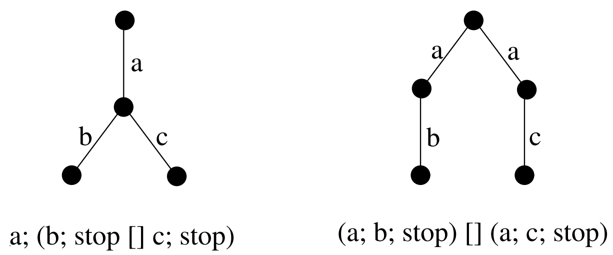

Abbildung 2: Nichtdeterminismus in LOTOS

unterscheiden sich nach der Auswahl von a, das zunächst beide anbieten. Während im ersten Fall nach Ausführung von a sowohl b als auch c möglich sind, hängt es im zweiten Fall davon ab, welches der beiden a konsumiert wurde, ob entweder nur b oder nur c möglich ist[12]. Der Benutzer muß also, um den Nichtdeterminismus im zweiten Fall aufzulösen, die Möglichkeit erhalten, zwischen den zwei möglichen Folgeverhalten auszuwählen. Diese Entscheidung kann nach der Auswahl von a getroffen werden oder es werden von vorneherein zwei verschiedene a's angeboten. Durch die Wahl eines der beiden a's entscheidet sich der Benutzer für das nachfolgende Verhalten. Diese Möglichkeit, ein Ereignis durch sein nachfolgendes Verhalten einzigartig zu machen, wird auch im vorgestellten BASIC-LOTOS Simulator verwendet.

[11] In LOTOS findet beobachtbares Verhalten immer hintereinander, nie gleichzeitig statt. Gleichwohl kann die Ausführung eines Ereignisses die gleichzeitige Änderung des Verhaltens mehrerer Prozesse bedeuten.

[12] Der LOTOS-Operator [] ist kommutativ aber nicht distributiv

Eine andere Möglichkeit, nichtdeterministischen Verhalten zu spezifizieren, ist die Verwendung des speziellen Action-prefix i. Er steht für das interne Ereignis und entspricht einem ϵ-Übergang in einem Automaten. In dem Ausdruck

```
(a; stop) [] (i; b; stop)
```

ist nicht festgelegt, ob als nächste Ereignisse a und b oder nur exklusiv b angeboten werden, da nicht eindeutig ist, ob das interne Ereignis eine Vorauswahl zugunsten von b treffen wird. Eine Anwendung findet sich bspw. in der Spezifikation einer langen Leitung, die möglicherweise korrekt, aber auch fehlerhaft übertragen kann. In der Simulation wird der Nichtdeterminismus durch explizite Auswahl von internen Ereignissen aufgelöst.

Die Benutzung des LOTOS Interpretierers besteht also in der wiederholten Auswahl der jeweils für das aktuelle Verhalten angebotenen Ereignisse. Zur dieser Minimalfunktionalität sollte auch noch die sukzessive Rücknahme von Ereignissen durch den Benutzer gehören. Eine umfassende Diskussion der Aspekte eines LOTOS-Simulators findet sich in [TJW+92].

3 CENTAUR

Im weiteren werden, nach einer allgemeinen Einführung, die Formalismen von CENTAUR zur Beschreibung der oben genannten Werkzeuge vorgestellt. Insbesondere auf TYPOL, das in dieser Arbeit zur Semantikspezifikation Verwendung fand, wird besonders eingegangen.

3.1 Allgemein

Ziel des GIPE Projekts war die Entwicklung von Spezifikations-Formalismen zur automatischen Generierung von Programmierumgebungen. Ein Ergebnis dieses Projekts ist der Programmierumgebungsgenerator CENTAUR. Aufgabe eines solchen Generators ist es, anhand der Beschreibung einer beliebigen Sprache, Werkzeuge zum Umgang mit dieser Sprache zu erzeugen. Zum Grundstock einer im Prinzip beliebigen Art von Werkzeugen gehören

Syntax directed Editor Texteditor mit Kenntnis der verwendeten Sprache. Üblich sind automatische Vorgabe von Programmschemata, Auswahl von gültigen Ausdrücken aus einem generiertem Menü usw.

Scanner/Parser Verwandelt den Programmtext in eine abstrakte Darstellung. Dieser Übersetzungsvorgang ist Grundlage für alle Werkzeuge, die sich mit den inhaltlichen Aspekten des Programmtextes befassen.

Pretty Printer Auch als Unparser bezeichnet, weil er die Umkehrung der Parserfunktion darstellt. Die abstrakte Darstellung wird durch ein sprachspezifisches Layout wieder zurück in einen Programmtext verwandelt. Einerseits kann so schluddrige Eingabe in eine einheitliche Form gebracht werden und andererseits können beispielsweise die Schlüsselwörter einer Sprache hervorgehoben werden.

Type Checker Überprüft die statische Semantik einer Sprache.

Interpreter Ist oft auch als Simulator oder Debugger ausgeprägt. Realisiert die dynamische Semantik einer Sprache und ermöglicht so deren Ausführung.

Compiler Übersetzt die Quellsprache in eine oft effizienter ausführbare Zielsprache.

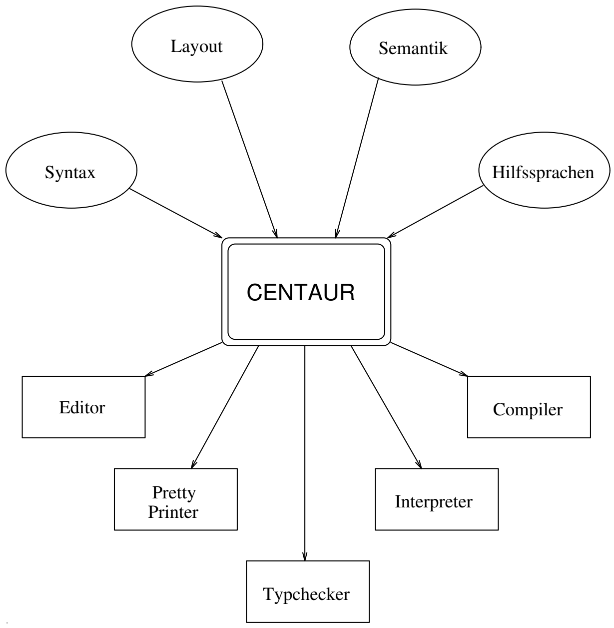

Abbildung 3: Ein- Ausgabe eines Generators

3.2 Formalismen

Der Schlüssel zur Generierung dieser Werkzeuge ist die exakte und vollständige Beschreibung der neuen Sprache. Hierzu die folgende Aufstellung:

Sprachdefinitionsteil	generierte Komponente
lexikalische Syntax	→ Scanner
konkrete Syntax	→ Parser
abstrakte Syntax	→ abstrakter Syntaxbaum Manager
Layout Regeln	→ Pretty Printer
statische Semantik	→ Typechecker
dynamische Semantik	→ Interpreter

Die Sprachdefinitionsteile lassen sich in zwei Teilbereiche, Syntax und Semantik, aufteilen. Für jeden Teilbereich bietet CENTAUR zwei Alternativen zur Beschreibung an.

Entwickler	Syntax	Semantik	Ausführung	Bemerkung
Niederlande	SDF	ASF	Term rewriting	Beliebige kontextfreie, aber möglicherweise mehrdeutige Grammatiken möglich. Automatische Generation der abstrakten aus der konkreten Syntax. Separate Angabe von Prioritätsregeln. Verwendung der konkreten Syntax in der Beschreibung der dynamischen Semantik.
Frankreich	METAL	TYPOL	Inferenz System	LALR(1) Grammatik. Eigene Abbildung von konkreter auf abstrakte Syntax möglich. Nur noch abstrakte Syntax in der Beschreibung der dynamischen Semantik verfügbar.

SDF Spezifikationen werden in einem speziellem Editor so angezeigt, wie sie der Benutzer eingegeben hat. Für METAL Spezifikationen ist die Angabe eines Pretty Printers in PPML obligatorisch. Allerdings kann auch für SDF Spezifikationen ein Pretty Printer angegeben werden und die dynamische Semantik in TYPOL formuliert werden. Eine PPML Spezifikation ist eine Ansammlung von Regeln der Art:

tree → format

`tree` ist ein abstrakter Syntaxbaum und `format` das dazugehörige Layout. Es wird in einer Box-Sprache angegeben:

plus(∗op1, ∗op2) → [<h> ∗op1 "+" ∗op2];

Hier wird festgelegt, daß der Operator `plus` als die horizontale Ausrichtung der Elemente linker Operator, + und rechter Operator angezeigt werden soll. Die Angabe von `*op1` bewirkt die rekursive Anwendung der Formatregeln auf den linken Teilbaum von `plus`. Es ist möglich, für die Ausgabe von Teilbäumen Kontexte zu definieren, mit denen ein spezielles Aussehen des Textes oder ein bestimmtes Auswahlverhalten bei Selektion von Teilbäumen mit der Maus erzielbar ist. Ein weiteres Feature, auf das hier nicht näher eingegangen werden kann, ist das des Print-Levels. Formatregeln können mit einem Detaillierungslevel versehen werden, der angibt, wann diese Regel zum Einsatz kommt. Im Editor gibt es die Möglichkeit den Detaillevel zu verändern, so daß von Übersichtsdarstellungen über Halbansichten bis Vollanzeigen alle Spielarten einstellbar sind. Abbildung 4 veranschaulicht das Box-Konzept von PPML anhand eines LOTOS-Specification-Header.

Die dazugehörige Formatregel lautet:

```
spec-header(*spec-identifier, *formal-parameter-list, *functionality) ->
[<h> "specification" *spec-identifier [<v> *formal-parameter-list
                                            *functionality
                                       ]
];
```

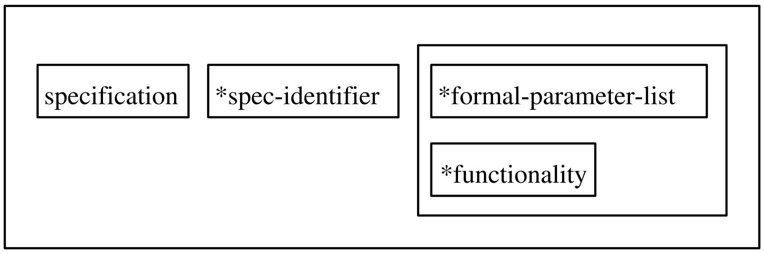

Abbildung 4: Anordnung der Boxen für einen LOTOS Specification Header

Zur Verdeutlichung des Unterschiedes zwischen SDF und METAL und zur Erklärung der im Anhang angegebenen Syntax-Spezifikationen folgt nun ein vergleichendes Beispiel. Gegeben sei eine Teilgrammatik für mathematische Ausdrücke in BNF Notation:

```
EXP ::= EXP "+" FACTOR
FACTOR::= TERM "*" FACTOR
```

Diese Teilgrammatik verwirklicht die Priorität von '*' über '+' und weist '+' Links- bzw. '*' Rechtsassoziativität zu. Dies kann in SDF so ausgedrückt werden:

context-free syntax

```
EXP "+" EXP -> EXP left
EXP "*" EXP -> EXP right
```

priorities

```
EXP "+" EXP -> EXP <
EXP "*" EXP -> EXP
```

Wie man sieht, ist die Notation, angelehnt an eine algebraische Schreibweise, umgekehrt zur BNF Schreibweise. Ein bestimmter Ausdruck ergibt einen bestimmten Typ. Wichtiger ist jedoch die Befreiung von den ursprünglich zusätzlich eingführten Nonterminalen `TERM` und `FAKTOR`. Durch die separate Angabe der Operatorprioritäten ist das Einführen von Nonterminalen, die nur Prioritäten einführen, überflüssig geworden. Da sich auch die Assoziativität angeben läßt und die abstrakte Syntax komplett mit den Baumaufbauregeln automatisch generiert wird, ergibt sich eine sehr kompakte Formulierung der Syntax einer Sprache.

Dasselbe Beispiel in METAL:

```
abstract syntax

plus -> EXP FACTOR;
times -> TERM FACTOR;
EXP ::= plus;
TERM ::= times;

rules

<exp> ::= <exp> "+" <factor>;
  plus(<exp>, <factor>)
<factor> ::= <term> "*" <factor>;
  times(<term>, <factor>)
```

Zunächst wird die abstrakte Syntax festgelegt. Es gibt zwei Operatoren `plus` und `times` mit jeweils zwei Nachfolgern. Für die Nachfolger wird ein Typ definiert, der nur bestimmte Operatoren erlaubt. In den `rules` erfolgt die Angabe der konkreten Syntax mit den dazugehörenden Baumaufbauregeln. Dieser Mehraufwand enthält die Möglichkeit, die Abbildung von konkreter auf abstrakte Syntax frei zu gestalten. Beispielsweise läßt sich so das unterschiedliche Erscheinungsbild eines `if-statements` mit und ohne `else` Zweig in der konkreten Syntax auf einen allgemeinen abstrakten Syntaxknoten abbilden.

```
<stat> ::= "if" <bexp> "then" <statement> "else" <statement>;
  if-stat(<bexp>, <statement>.1, <statement>.2)

<stat> ::= "if" <bexp> "then" <statement>;
  if-stat(<bexp>, <statement>, <null-stat()>)
```

Zur Komplettierung einer Programmierumgebungskomponente gehört die Beschreibung der Benutzerschnittstelle, also die Festlegung, welche Funktionen auf welche Art und Weise ausgelöst werden können. Augenblicklich erfolgt dies in CENTAUR mit Lisp unter Verwendung von CENTAUR-Funktionen. Dies entspricht mehr dem *Programmieren* denn dem *Spezifizieren* einer Benutzerschnittstelle. Die sich noch in der Entwicklung befindliche Sprache SEAL soll dem Abhilfe verschaffen.

Eine fertige Programmierumgebung in CENTAUR besteht zunächst aus einem sprachspezifischen Editor. Ein Standard-Editor bekommt durch Laden eines Programmtextes sein spezifisches Aussehen und Verhalten. Durch hinzugefügte Buttons oder Pop-up Menüs und spezielles Mausverhalten können Aktionen ausgelöst oder Tools gestartet werden. Es ist möglich, andere Editoren zur Anzeige von Ergebnissen oder als zusätzliche Tools zu erzeugen und mit dem Ursprungseditor zu einer Umgebungs-Einheit zu verbinden. Für jeden Editor kann ein individueller Pretty Printer, von denen es mehrere Versionen für eine Sprache geben kann, angegeben werden. Durch Standard-Editor-Funktionen können Teilbäume unter den Editoren, Verträglichkeit vorausgesetzt, ausgetauscht werden.

3.3 TYPOL

Zur Spezifikation von Sprachsemantik bietet CENTAUR, wie bereits erwähnt, die Formalismen ASF und TYPOL an. Innerhalb GIPE II werden zwei LOTOS-Simulatoren zum Vergleich beider Formalismen entstehen. Vorgabe dieser Arbeit war die Verwendung von TYPOL. Der Ausgangspunkt von TYPOL ist Natural Semantics [Kah87]. Eine Ähnlichkeit zu PROLOG oder allgemein zur logischen Programmierung ergibt sich durch die Verwendung von Inferenzregeln. Eine Inferenzregel besteht aus zwei Teilen, die durch einen Schlußstrich[13] getrennt werden: Die Prämissen und den Schluß. Wenn alle Prämissen bewiesen werden können, so ist auch der Schluß bewiesen. Prämisse und Schluß besitzen beide die Form:

$$\text{Hypothese} \vdash \text{Aussage}$$

Unter Annahme bestimmter Voraussetzungen, der Hypothese, wird ein bestimmter Sachverhalt behauptet, die Aussage. Die axiomatische Regel

$$\vdash \text{true} : \text{boolean};$$

ist folgendermaßen zu lesen: Ohne weitere Voraussetzungen hat `true` den Typ `boolean`. Der Doppelpunkt in der Aussage trägt keine Bedeutung. Sonderzeichen in Aussagen dienen nur zur Trennung von TYPOL-Bezeichnern. Es ist allerdings sinnvoll, eine Benutzungskonvention einzuhalten. So ist es üblich "`:`" als "hat den Typ" und "`->`" als "daraus folgt" zu lesen. Eine Erklärung der in der Simulator Spezifikation verwendeten Zeichenketten findet sich im Kapitel Entwurf. Als Beispiel für eine Schlußregel:

$$\frac{\begin{array}{l}\vdash \text{Exp1} : \text{boolean}\ \&\\ \vdash \text{Exp2} : \text{boolean}\end{array}}{\vdash \text{or}(\text{Exp1}, \text{Exp2}) : \text{boolean};}$$

Der Operator `or` hat den Typ `boolean`, wenn seine Operanden vom Typ `boolean` sind. Auch dieses Mal waren keine Hypothesen nötig. Um die Interpretation von Hypothesen zu verdeutlichen, nehmen wir Variablen mit einem zugeordnetem Typ in unsere Ausdruckssprache auf. Der Typ eines Ausdrucks, der Variablen enthält, läßt sich jetzt nur noch mit Kenntnis der Variablentypen bestimmen. Also:

$$\_ \vdash \text{true} : \text{boolean};$$
$$\_ \vdash \text{false} : \text{boolean};$$

$$\frac{\text{LOOKUP}(\text{Env} \vdash \text{Name} : \text{VarTyp})}{\text{Env} \vdash \text{variable}(\text{Name}) : \text{VarTyp};}$$

$$\frac{\begin{array}{l}\text{Env} \vdash \text{Exp1} : \text{boolean}\ \&\\ \text{Env} \vdash \text{Exp2} : \text{boolean}\end{array}}{\text{Env} \vdash \text{or}(\text{Exp1}, \text{Exp2}) : \text{boolean};}$$

`true` und `false` sind Typaxiome, benötigen daher kein Typenvironment zur Typbestimmung. Sie sind aber Teil einer Regelsammlung, die als Hypothese ein Typenvironment annimmt. Deshalb erscheint das Typenvironment als anonyme Variable in den Axiomen. Die erste Regel trifft zu, wenn der Ausdruck, dessen Typ bestimmt werden soll, eine Variable ist. Sein Typ `VarTyp` ergibt sich dann aus dem Variablentyp, der durch die Regelsammlung LOOKUP bestimmt wird. Diese benutzt das als Hypothese vorangestellte Typenvironment `Env`, um den Variablentyp von `Name` zu bestimmen. Die letzte Regel ist dahingehend erweitert, daß `Exp1` und `Exp2` nun unter der Voraussetzung eines Typenvironments den Typ `boolean` haben müssen.

In der Terminologie der attributierte Grammatiken handelt es sich bei `or(Exp1, Exp2)` um das Subjekt, den zu bewertenden Ausdruck, bei `Env` um ein ererbtes und bei `boolean` um ein synthetisiertes Attribut. Im allgemeinen Fall werden ererbte Attribute an die Prämissen weiter vererbt und die synthetisierten Attribute des Schlusses ergeben sich aus den synthetisierten Attributen der Prämissen. Mit einer speziellen Art TYPOL-Regeln lassen sich also attributierte Grammatiken implementieren. TYPOL in seiner Allgemeinheit hat mehr Ähnlichkeit zu den *Relational Attribute Grammars* [DM85].

[13] vom logischen Schließen

Obiges Beispiel verdeutlicht auch das Konzept der Regelsammlung, in Typol `SET` genannt. Regeln in einem `SET`, daß sozusagen ein Thema behandelt, genügen alle einem Typschema, das innerhalb des `SET`s angegeben wird. Für das Beispiel lautet es:

judgement ENVTYP $\vdash$ EXP : TYP

`ENVTYP`, `EXP` und `TYP` sind Operandentypen aus den jeweiligen Sprachdefinitionen. In einer Typol-Regel können an dieser Stelle Variablen wie `Exp1` oder abstrakte Syntaxbäume wie `or(Exp1, Exp2)` oder `true` stehen. Als Komplettbeispiel eines `SET`s nun noch die Definition von LOOKUP:

set LOOKUP is

judgement ENVTYP $\vdash$ NAMETYP : TYP;

env[] $\vdash$ Name : unknown;

env[entry(Name, Typ).Tail] $\vdash$ Name : Typ;

$$\frac{\text{Tail} \vdash \text{Name : Typ}}{\text{env[entry(Name}', \text{Typ).Tail]} \vdash \text{Name : Typ;}}$$
provided diff(Name', Name);

end LOOKUP;

Hier ist `ENVTYP` eine Liste `env` mit Einträgen vom Typ `entry`, der als Nachfolger den Variablennamen und den Variablentyp enthält. Das erste Axiom ordnet einem Variablennamen `Name` den Typ `unknown` zu, falls das Typenvironment aus der leeren Liste besteht. Aus Gründen der Lesbarkeit wurde in diesem Beispiel von der möglichen Verwendung von anonymen Variablen abgesehen. Das zweite Axiom ordnet einem Variablennamen `Name` genau den Typ `Typ` zu, der in dem `entry` für diesen Namen im Typenvironment steht, falls er sich an erster Stelle in der Liste befindet. Für den anderen Fall, daß sich der Eintrag für den Namen irgendwo im `Tail` der Liste befindet, ist die erste Regel zuständig. Gibt es eine Liste mit einem Anfangseintrag, dessen `Name'` sich von `Name` unterscheidet, so ergibt sich der Variablentyp aus dem Ergebnis der Suche in der Restliste. Diese Verkleinerung der Liste findet solange statt, bis entweder das erste oder das zweite Axiom zutreffen. Die `provided` Klausel in der ersten Regel trennt die Anwendungsfälle von dem zweiten Axiom und der ersten Regel. Ohne diesen Zusatz würde die erste Regel auch bei Gleichheit von `Name'` und `Name` zutreffen, woraus eine falsche Semantik resultiert.

In Typol wird also die Semantik einer Sprache in typisierte Regelsysteme aufgeteilt, die jeweils Teilaspekte der Semantik spezifizieren. Eine typische Aufteilung ist die in Induktion über den syntaktischen Aufbau einer Sprache und diverse Hilfsfunktionen, die bspw. mathematische Konzepte verwirklichen. Die Berechnung eines semantischen Wertes S eines Syntaxterms T mit den Voraussetzungen H erfolgt durch Beweis des Zieles H $\vdash$ T : S. Der Beweis erfolgt durch Anwendung von Axiomen und Regeln des Regelsystems. Die Schlußfolgerungen von Regeln werden durch den Beweis der Prämissen der Regel durchgeführt. Hierbei ist auch die Reerfüllung von Regeln möglich. Besitzt eine Prämisse mehrere Lösungen, so werden alle Lösungen erzeugt, bis die Regel, die die Prämisse enthält, erfüllbar wird. Die Konstrukte, aus denen Hypothesen und Aussagen bestehen, sind Bäume und Listen. Ihre Verwendung setzt die Definition in einer Sprachspezifikation voraus. Für beide bestehen Möglichkeiten zur Dekomposition und damit zur Untersuchung ihrer Inhalte. Durch Unifikation sind hierbei beliebig komplizierte Strukturen erlaubt. Als Beispiel einer Programmtransformation:

$\vdash$ statements[ifthenelse(true, IfStat, ElseStat).Stats] $\rightarrow$ statements[IfStat.Stats]

Für die Berechnungen, die sich in Typol nur umständlich ausdrücken lassen, wie z.B. einfache Arithmetik, sind Prolog-Prädikate importierbar. Auch eine sehr unhandliche Verwendung von Lisp Funktionen ist möglich. Am häufigsten werden jedoch Sprachen zur Definition von Daten und andere Typol-Module importiert. Typol-Funktionen, die mit einer `export` Anweisung öffentlich gemacht werden, sind in anderen Modulen oder in der Lisp Umgebung für die Realisierung von Toolfunktionen verfügbar. Eine vollständige Behandlung von Typol erfolgt in [Cen92, Des88].

4 Entwurf

Der Entwurf des BASIC-LOTOS-Simulators läßt sich in zwei Teile gliedern:

- Festlegen der Simulatorfunktionalität und deren Repräsentation zum Benutzer mit den Mitteln von CENTAUR.
- Entscheiden zwischen den Alternativen der Realisierung

Die Simulatorfunktionalität wurde auf ein Minimum reduziert. Es können Ereignisse ausgeführt, wieder zurückgenommen und der Zustand in einer Darstellung beobachtet werden. Nachfolgend werden zunächst die Simulatorkomponenten motiviert, um nach Klärung der gegebenen Voraussetzungen für den Entwurf, die Alternativen zur Realisierung der Komponenten zu erörtern. Dazu werden zunächst die elementaren Simulatordaten und -funktionen aufgezählt und anschließend in einzelnen Abschnitten diskutiert.

4.1 Komponenten

Da schon die Signatur der Simulatorfunktionen Aufschluß über die Simulatorkomponenten zuläßt, folgt vor der Identifikation der Komponenten ein kurzer Vorgriff auf die Simulatorfunktionen. Die im Kapitel Lotos erwähnte Vorgehensweise zur Simulation einer LOTOS-Spezifikation ermöglicht die Identifizierung von zwei elementaren Simulatorfunktionen: Die *menu*-Funktion berechnet alle möglichen Ereignisse eines gegebenen Zustands. In einem *transition system*, wie es durch eine LOTOS-Spezifikation definiert wird, meint man mit Zustand die Punkte zwischen den Ereignissen. In Abbildung 2 entsprechen den Graphenknoten Zuständen einer Simulation. Eine mögliche Zustandsdefinition für einen BASIC-LOTOS-Simulator ist eine Behaviour-Expression, die durch Ereignisausführung modifiziert wird. Die *next*-Funktion dient zur Berechnung des Folgezustands B' in $B \xrightarrow{a} B'$. Wie bereits angedeutet, sind hier zwei Alternativen der Arbeitsaufteilung zwischen *menu* und *next* zu nennen:

1. $menu(B) = \{a \mid \exists B' : B \xrightarrow{a} B'\}$. *menu* berechnet die Menge aller Ereignisse a, die vom Zustand B auf einen Zustand B' führen. Folglich: $next(B, a) = B' \, if B \xrightarrow{a} B'$. Zu einem gegebenen Zustand B und einem Ereignis a wird der Folgezustand B' berechnet.
2. $menu(B) = \{(a, B') \mid \exists B' : B \xrightarrow{a} B'\}$. *menu* berechnet nun die Menge der Tupel möglicher Ereignisse, kombiniert mit dem Folgezustand der Ereignisse. Der Definitionsbereich der *next*-Funktion ist jetzt eine Menge von Ereignis-Zustands Tupeln und eine Tupelauswahl. Das Funktionsergebnis ist die Zustandskomponente des ausgewählten Tupels.

Die erste Alternative ordnet *next* die Berechnung des Folgezustands und eine zweifache Auswahl zu. Nach der Auswahl eines Mengenelementes a besteht i.a. die Möglichkeit von mehreren Folgezuständen, wie der folgende LOTOS-Term zeigt:

```
a; B1 [] a; B2
```

Nach der Auswahl von a muß also noch zwischen den Folgezuständen B1 und B2 ausgewählt werden, um systematisches Erschließen von Nichtdeterminismus zu ermöglichen. In der zweiten Alternative sind alle Elemente der Ergebnismenge von *menu* eindeutig. Für das Beispiel entstehen die Mengenelemente (a, B1) und (a, B2). Der *next*-Funktion kommt in BASIC-LOTOS[14] also nur noch die Aufgabe der Auswahl aus den Mengenelementen zu.

Die zweite Alternative empfiehlt sich aber nicht nur durch die höhere Eleganz der Darstellung, sondern auch durch Implementierungsaspekte. Bei der Berechnung der möglichen Ereignisse bedeutet es kaum Mehraufwand, den Folgezustand mitzuberechnen. In einer einzigen Baumtraversierung können beide Ergebnisse gleichzeitig bestimmt werden. Gerade in TYPOL, in dem eine Top-down - Bottom-up Auswertungsrichtung vorgegeben ist, entfallen unnötige Baumnavigationen.

[14] Die zweite Tupelkomponente eines Mengenelementes ist in LOTOS kein Folgezustand, sondern eine parametrisierte Funktion eines Folgezustands. *next* erfüllt hier zusätzlich die Funktion des Anforderns und des Einsetzens der aktuellen Parameter mit anschließender Auswertung der Funktion auf den Folgezustand.

Betrachtet man Definitionsbereich (Zustand) und Funktionsbereich (Menge möglicher Ereignisse) der *menu*-Funktion und berücksichtigt die Notwendigkeit eines Behälters für die LOTOS-Spezifikation, so ergeben sich automatisch drei Komponenten eines Simulators:

- Editor der Spezifikation
- Repräsentation des Zustands der Simulation
- Auswahlmöglichkeit der möglichen Ereignisse

Diese werden durch drei CENTAUR-Fenster repräsentiert. Zur Illustration des Erscheinungsbildes des Simulators in CENTAUR folgt nun eine Beschreibung des Beginns einer Simulation anhand der Abbildung 5. Zunächst ist nur der BASIC-LOTOS-Editor vorhanden, der durch sein sprachspezifisches pop-up Menü den Start der Simulation ermöglicht (großes Fenster links unten). Daraufhin erscheint ein Fenster für die Zustandsrepräsentation (breites Fenster oben) und ein Fenster, das die für diesen Zustand möglichen Ereignisse anzeigt (Fenster rechts). Durch *clicken* mit der Maus auf ein Ereignis wird dieses ausgeführt, d.h. der Folgezustand in das Fenster mit der Zustandsrepräsentation übernommen und eine neue Menge möglicher Ereignisse berechnet. Rücknahme von Ereignissen erfolgt durch ein pop-up im Auswahlfenster oder durch *clicken* mit der Maus auf den Anfang der Liste mit den möglichen Ereignissen. Notwendigerweise muß die Zustandsrepräsentation eine Filterung der Informationsmenge vornehmen. Die Ausgabe eines monolithischen LOTOS-Terms, eine Behaviour-Expression, bei der alle Prozeßaufrufe durch ihre Definitionen ersetzt worden sind, ist zum einem wenig hilfreich bei der Analyse und zum anderen völlig unüberschaubar selbst für Spezifikationen mittleren Umfangs. Für den BASIC-LOTOS-Simulator wurde die Informationsmenge auf die im Augenblick aktiven Prozesse beschränkt. Für jeden Prozeß, der potentiell an einer Kommunikation teilnehmen kann, wird sein Prozeßkopf, einschließlich der Ereignisumbenennungen, entsprechend seiner Position in der hierachischen Aufrufstruktur angezeigt. Wahlweise kann aber auch der vollständige Zustand zur Anzeige gebracht werden.

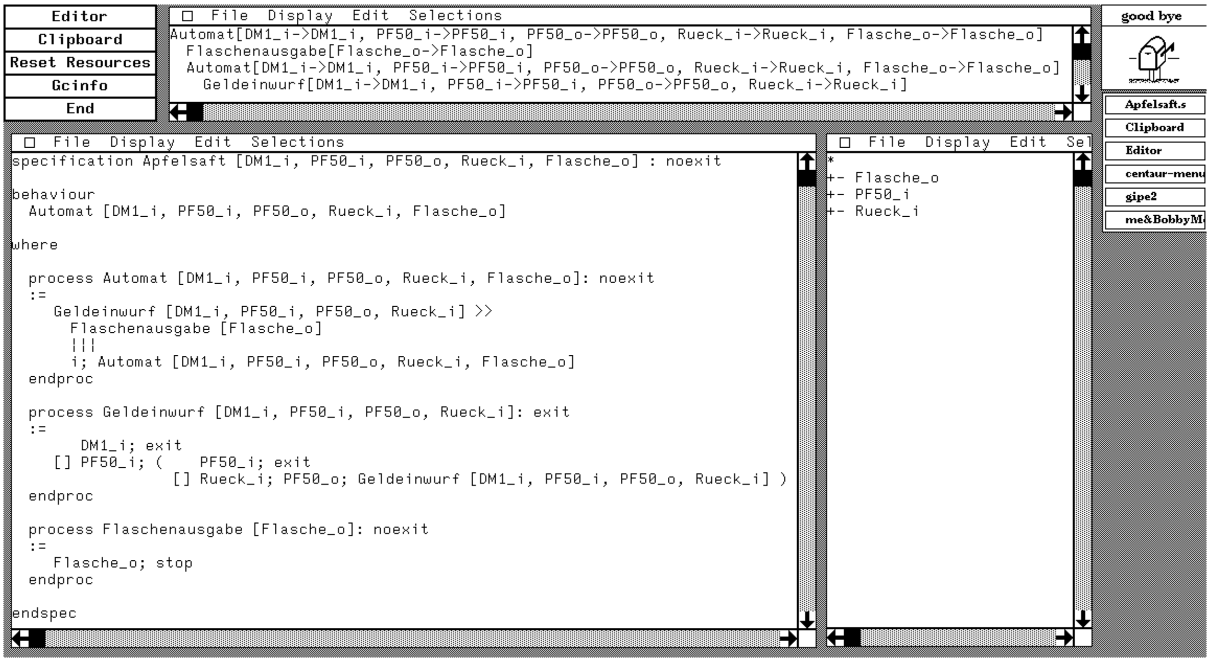

Abbildung 5: Simulatorsitzung

4.2 Voraussetzungen

Im Rahmen der Arbeiten an einem Typchecker für BASIC-LOTOS bei PTT-Research in den Niederlande entstanden die Sprachbeschreibungen für einen sprachspezifischen Editor. Die Definition der Syntax in SDF entspricht in Form und Inhalt nicht exakt der des Standards in [ISO87]. Zum einem ist durch die Ausdrucksweise in SDF eine kompaktere Darstellung möglich, zum anderen ist die formale Beschreibung der Syntax nicht nur Grundlage zur Generation eines Parsers, sondern auch eines Editors. Um ein natürliches Selektionsverhalten des Editors zu erreichen, werden oft zusätzliche Nonterminale eingeführt, die die Menge der Satzformen der Grammatik zwar nicht ändern, aber relevante Komponenten (Teilgrammatiken) identifizieren. Statt beispielsweise durch

```
"specification" spec-identifier formal-param-list data-type-defs*
"behaviour"     behaviour-exp opt-local-defs
"endspec"                                                 -> specification
```

alle Keywords der ersten Grammatikregel zuzuschlagen, macht es Sinn, die Nonterminale `spec-header` und `behaviour-part` einzuführen.

```
spec-header
data-type-defs*
behaviour-part
opt-local-defs
"endspec"                                                 -> specification

"specification" spec-identifier formal-param-list       -> spec-header
"behaviour" behaviour-exp                                 -> behaviour-part
```

Auswählen von `"specification"` und `"behaviour"` selektiert nun nicht mehr die gesamte Spezifikation, sondern nur noch einen sinnvollen Teilbereich.

Eine Differenz inhaltlicher Art ergibt sich durch die Unfähigkeit, innerhalb SDF die Assoziativität des parallel-Operators auszudrücken[15]. Im Anhang befindet sich die SDF Spezifikation von BASIC-LOTOS.

Da TYPOL auf der abstrakten Syntax aufsetzt, muß bei der Verwendung von TYPOL, anstatt ASF, als Semantikdefinitionssprache die SDF Syntax-Spezifikation um die Angabe von Namen der abstrakten Syntaxknoten angereichert werden. Nur so kann sich eine TYPOL-Regel auf ein BASIC-LOTOS-Konstrukt beziehen.

Auch bei der Definition des Pretty Printers muß auf das spätere Selektionsverhalten des Editors geachtet werden. Damit die Selektion einer Klammer die Selektion des Teilausdrucks innerhalb des Klammerpaares hervoruft und nicht die des Ausdrucks, deren Operand der Klammerausdruck ist, müssen besondere Vorkehrungen getroffen werden. In PPML geschieht das mithilfe von sogenannten Kontexten, mit denen sich auch durch das Parsen weggefallene Prioritätsklammern wieder erzeugen lassen. Für LOTOS gibt es keine Vorgaben bezüglich der Formatierung von Spezifikationen. Somit ist es abgesehen vom Aspekt der Lesbarkeit dem Geschmack des Sprachdefinierers überlassen, welche Darstellung er wählt. Hier zeigt sich positiv die Flexibilität von CENTAUR, mehrere Pretty Printer zu einer Sprachdefinition zuzulassen, so daß ein Benutzer ggf. eine eigene modifizierte Darstellung wählen kann.

[15] siehe Kapitel Bewertung

4.3 Daten

In diesem Abschnitt werden die Datenstrukturen, auf denen der Simulator fußt, motiviert und erläutert. Die oben genannte *menu*-Funktion impliziert durch ihren Definitions- und Wertebereich Repräsentationen für Zustand und Ereignismenge. In CENTAUR werden alle Daten durch eine abstrakte Syntax definiert. Somit sind folgende Sprachen, mit denen der Simulator arbeitet, gegeben:

Slotos Die abstrakte Syntax von BASIC-LOTOS ist der Ausgangspunkt der Simulation. In diesem Format ist die aktuelle Spezifikation und damit auch die Information über alle Bezeichner gespeichert.

State repräsentiert den Zustand der Simulation. Diese Sprache muß Operatoren bereitstellen, die den dynamischen Zustand der Simulation möglichst kompakt repräsentieren können.

Menu hält die Menge der zum aktuellen Zustand möglichen Ereignisse. Die Struktur eines LOTOS-Ereignisses muß hier, um Eindeutigkeit zu erreichen, um das Folgeverhalten des Ereignisses erweitert werden.

Eine weitere Datenstruktur wird benötigt, um die Konstruktion eines State aus Slotos zu ermöglichen.

Env ist ein Environment für Prozeßbezeichner. Der Übergang von der statischen Spezifikation zur dynamischen Zustandsrepräsentation verlangt, speziell in LOTOS, den komplexen Zwischenschritt des Flattenings[16]. Für BASIC-LOTOS ist es ausreichend, die Gültigkeitsbereiche von Prozeßbezeichnern zu berechnen.

4.4 Funktionen

Zur Begründung der Wahl von Alternativen bei dem Entwurf des Simulators zunächst noch einmal eine Übersicht der benötigten Grundfunktionen:

Funktion	Signatur	Beschreibung
annotate	spec $\rightarrow$ spec'	Flattening
expand	spec $\rightarrow$ state	Erzeugung des initialen Zustands
menu	state $\rightarrow$ set of events	Berechnung der möglichen Ereignisse
next	state $\stackrel{event}{\rightarrow}$ state'	Folgezustandberechnung
follow	state $\rightarrow$ state'	Zustandsvereinfachungen

In dieser Aufteilung steckt bereits die Entscheidung das Flattening, also die Eindeutigkeit der Prozeßbezeichner und Zugriff auf deren Prozeßdefinition zu gewährleisten, als Transformation der Spezifikation durchzuführen.

[16]Eindeutigkeit aller Bezeichner, Auflösung von Datentypabhängigkeiten und eindeutige Typisierung gewährleisten. Definition in [ISO87].

4.5 Flattening

Bei der Simulation eines Prozeßaufrufs muß der Aufruf formal, unter Berücksichtigung der aktuellen Parameter, durch die Prozeßdefinition ersetzt werden. Es ist denkbar ein Prozeßenvironment dynamisch während der Simulation mitzuführen und auszuwerten. Dies bedingt aber Nachteile bezüglich

- Effizienz - Während der interaktiven Simulation muß ständig das Environment mitgeführt und modifiziert werden. Bei jeder Prozeßinstantiierung muß das aktuelle Environment nach dem Prozeßbezeichner durchsucht werden.
- Übersichtlichkeit - Die Spezifikation der dynamischen Semantik wird mit der Spezifikation von Gültigkeitsbereichen gemischt. Durch die explizite Mitführung eines Environments vergrößert sich die Spezifikation der dynamischen Semantik insgesamt, um nur in kleinen Teilbereichen an Informationsgehalt zu gewinnen.

Deshalb findet am Beginn einer Simulation eine einmalige Vorverarbeitung der Spezifikation statt. Eine spezielle TYPOL-Regelsammlung besucht alle Behaviour-Expressions und Prozeßdefinitionen der Spezifikation, wobei mithilfe der Prozeßdefinitionen ein aktuelles Environment bereitgestellt wird und alle Prozeßaufrufe in den Behaviour-Expressions mit den zugehörigen Prozeßdefinitionen aus dem aktuellen Environment kommentiert werden. CENTAUR bietet die Möglichkeit, abstrakte Syntaxknoten mit Zusatzinformation zu versorgen. Wenn während der Simulation ein Prozeßaufruf zu einem Prozeß instantiiert werden soll, kann sofort über die im abstrakten Syntaxknoten des Prozeßaufrufs gespeicherte Zusatzinformation auf die entsprechende Prozeßdefinition zugegriffen werden. Handelt es sich bei der Zusatzinformation nur um einen Verweis auf die originale Prozeßdefinition innerhalb der Spezifikation entsteht hierdurch nicht einmal unnötiger Speicherplatzverlust durch das mehrfache Bereitstellen gleicher Prozeßdefinitionen[17]. Die Vorteile Geschwindigkeit des Zugriffs und Separation der Spezifikation der Gültigkeitsbereiche werden mit einem Nachteil erkauft. Auf die Zusatzinformation kann nur mit Wissen der Wurzel des betreffenden abstrakten Syntaxbaums zugegriffen werden. Die Zusatzinformation muß daher zusätzlich zur Prozeßdefinition auch deren Herkunft enthalten. Aus diesem Grund enthält das Prozeßenvironment nicht nur Bezeichner und Definition, sondern einen Bezeichner und einen Zugriff auf die Definition, der die Definition und deren Herkunft enthält. Die zusätzliche Verwaltung der Herkunft bei Speicherung im Environment und beim Zugriff während der Prozeßinstantiierung steht jedoch in keinem Verhältnis zu den Vorteilen dieser Vorgehensweise.

Die Sprache ENV, deren METAL-Spezifikation sich im Anhang befindet, definiert die Datenstruktur des Prozeßenvironments. Ein lokales Environment, das einem Gültigkeitsbereich entspricht, ist eine Liste von Tupeln der Form Prozeßbezeichner $\mapsto$ Definitionszugriff. Um die Verdeckung von Bezeichnern durch lokale Gültigkeitsbereiche zu modellieren, wurde das globale Environment als Liste von lokalen Environments angelegt. Beginnt in der Spezifikation ein neuer Gültigkeitsbereich durch lokale Prozeßdefinitionen, wird dem globalen Environment ein lokales hinzugefügt. Wird zu einem Prozeßbezeichner die zugehörige Definition gesucht, werden die lokalen Environments sukzessiv, in der umgekehrten Reihenfolge des Aufbaus, nach dem Prozeßbezeichner durchsucht.

[17] Obwohl die Spezifikation Referenzierung zuließe, wird kopiert statt referenziert

4.6 Zustandsrepräsentation

Vor der Behandlung der zweiten Grundfunktion *expand*, Erzeugung des initialen Zustands, muß die Zustandsrepräsentation vorgestellt werden. Der Zustand ist darstellbar als monolithischer Term, eine einzige Behaviour-Expression, die zusammen mit den Prozeßdefinitionen alles zukünftige Verhalten festlegt. Zustandsübergänge bestehen u.a. aus Entfernen von Teilausdrücken wie `action_prefix` oder die Eliminierung der nicht gewählten Seite eines `choice` und dem Einsetzen von Prozeßdefinitionen anstelle der Prozeßaufrufe. Berücksichtigt man jedoch, daß

- schon ein einzelner Zustand sehr groß werden kann,
- für die *undo*-Funktion alte Zustände gespeichert werden müssen und
- jedes berechnete mögliche Ereignis einen Folgezustand enthält,

ist eine weniger speicherintensive Repräsentation wünschenswert. Dazu ist es sinnvoll, den Zustand in einen dynamischen und einen statischen Teil zu gliedern. Der statische Anteil, wie Prozeßdefinitionen und Ausdrücke, deren Söhne sich nicht verändern, ist in der Originalspezifikation enthalten. Der dynamische Teil bildet den Zustand und enthält Verweise auf den statischen Teil. Eine erste Aufteilung der Zustandsoperatoren ergibt:

dynamisch	statisch
enable	choice
disable	action_prefix
par-op	exit
hide	stop
proc_inst	
behaviour	

Gegenüber BASIC-LOTOS-Behaviour-Expressions ist der `proc_call` durch den `proc_inst` ausgetauscht und der Operator `behaviour`, der im dynamischen Zustand den Verweis auf den statischen Anteil realisiert, hinzugekommen. Der `choice` muß nicht im Zustand enthalten sein, solange sein Menü, die Menge der möglichen Ereignisse berechnet werden kann. Bei der Ausführung eines Ereignisses kann der `choice` sofort durch den entsprechenden Folgezustand des Operanden, der das Ereignis angeboten hat, ersetzt werden. Leider ist die Voraussetzung der Berechenbarkeit des Menüs nur durch hohen Aufwand zu erreichen. Die Operanden eines statischen `choice` sind BASIC-LOTOS-Behaviour-Expressions und nicht State-Zustandoperatoren, für die die *menu*-Funktion definiert ist. Soll die Zustandsrepräsentation für den einen Operanden des `choice` erst dann in den Zustand aufgenommen werden, wenn der andere Operand durch eine Ereignisauswahl bereits weggefallen ist, muß man die *menu*-Funktion auch für Behaviour-Expressions definieren. Im vorliegenden Entwurf wurde dieser Mehraufwand nicht unternommen, da er einzig und alleine dem Zweck dient, das Speichern der Zustandsrepräsentationen der `choice` Operanden bis zur ersten Teilnahme des `choice` an einer Interaktion aufzuschieben, denn natürlich ist die Zustandrepräsentation nach der Interaktion in beiden Fällen identisch. Aus diesem Grund wird der `choice`-Operator in die Menge der dynamischen Operatoren übernommen. Abbildung 6 veranschaulicht die Repräsentation des Zustands der im Anhang befindlichen Spezifikation `two_slot_buffer`, nach der Ausführung der Ereignisse *inp* und (*middle*).

Aus Gründen, die erst im nächsten Abschnitt ersichtlich werden, wurden für die Ausdrücke

```
behaviour(stop)
behaviour(exit)
```

die Abkürzungen

```
s_stop
s_exit
```

eingeführt. Der Prefix `"s_"` ist allen dynamischen Zustandsoperatoren vorangestellt und ist die Abkürzung für state.

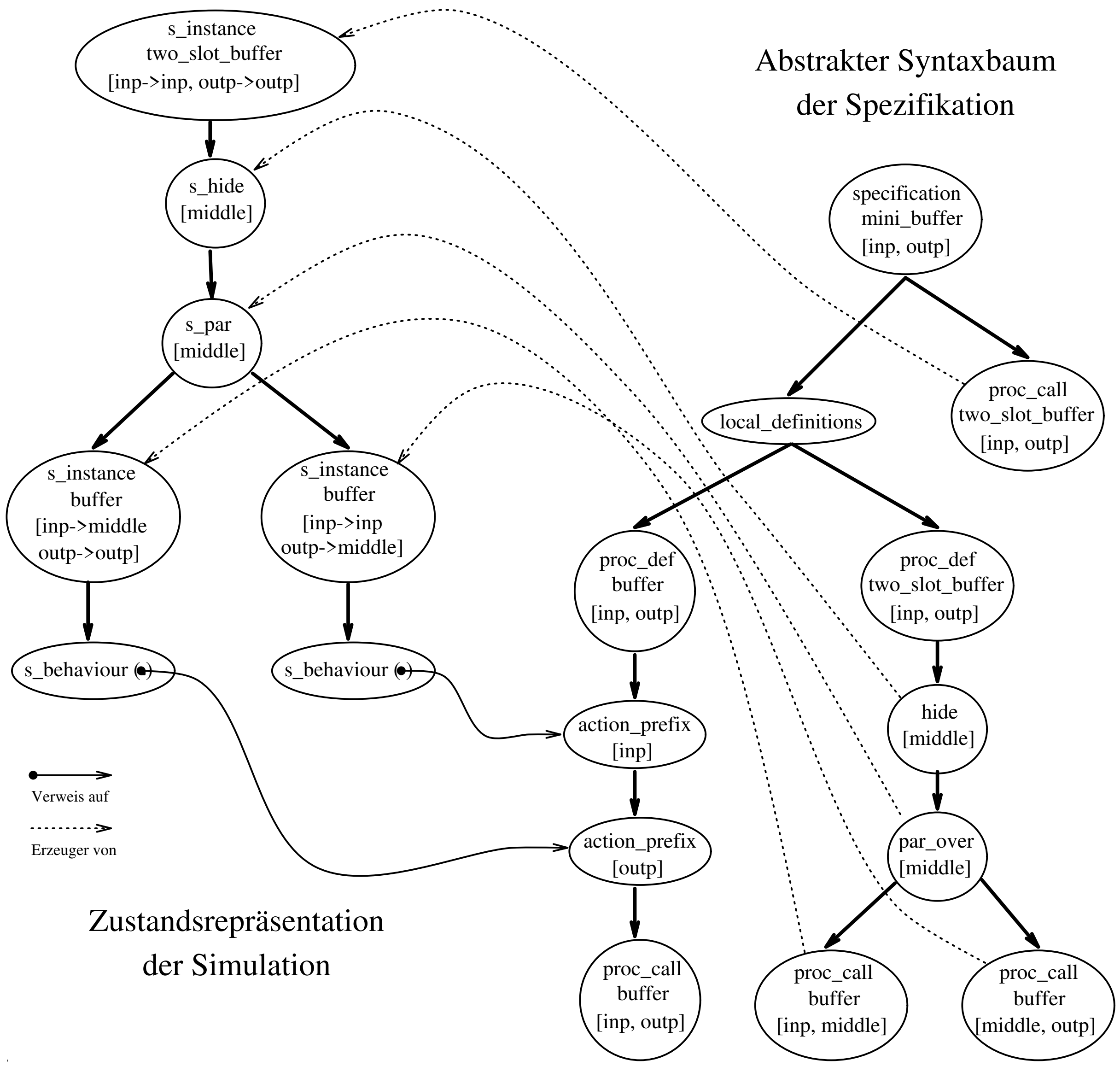

Abbildung 6: Aufteilung in dynamischen und statischen Zustand

Daß die endgültige Aufzählung der Zustandsoperatoren

```
s_choice, s_enable, s_disable, s_par, s_hide, s_instance,
s_behaviour, s_stop, s_exit
```

nur noch den `action_prefix` und die Prozeßdefinitionen im statischen Anteil beläßt soll nicht darüber hinwegtäuschen, daß diese Zustandsrepräsentation erheblich kompakter ist als ein monolithischer Term. Der `action_prefix` verschiebt die Transformation seines Operanden in einen Zustand und die Menü-Berechnung dieses Zustands bis zu dem Zeitpunkt, an dem das Ereignis, das der `action_prefix` anbietet, ausgeführt wird. Solange verwenden alle Instantiierungen desselben Prefixes auch die gleiche statische Repräsentation des Folgeverhaltens, nämlich die entsprechende Stelle in der Originalspezifikation. Des weiteren wird der zweite Operand des `enable` solange nicht benötigt, bis der erste Operand ein δ-Ereignis[18] anbietet und dieses ausgeführt wird. Bis dahin bleibt der zweiter Operand ein untransformierter Verweis auf die Originalspezifikation.

Die Behandlung der Zustandsrepräsentation wird abgeschlossen durch die Darstellung der Übersetzung von vier SDF-Instanzen des parallel-Operators auf eine einzige State-Instanz. Die vier Ausprägungen des parallel-Operators innerhalb der SDF Definition lassen sich wie folgt auf den allgemeinen parallel-Operator zurückführen:

Name	Syntax	Repräsentation
synchronise	B_1 `||` B_2	s_par(S_1, S_2, all_gids()[19])
interleave	B_1 `|||` B_2	s_par(S_1, S_2, empty_gid_list())
par	B_1 `|[ ]|` B_2	s_par(S_1, S_2, empty_gid_list())
par_over	B_1 `|[`$g_1,\ldots,g_n$ `]|` B_2	s_par(S_1, S_2, gid_list($g_1,\ldots,g_n$))

Tabelle 2: Zusammenfassung der parallel-Operatoren

Somit muß die sehr ähnliche Semantik der vier parallel-Operatoren nur einmal für den allgemeinen Fall fomuliert werden. Der Pretty Printer, der den Zustand ausführlich anzeigt[20], ist sogar in der Lage, die Rücktransformationen vorzunehmen. Der Zustandsoperator s_par(S_1, S_2, all_gids()) wird also als S_1 `||` S_2 angezeigt. Wie leicht ersichtlich ist, wird allerdings eine Angabe von `|[ ]|` in der Spezifikation im Zustand als `|||`,die im Prinzip angemessenere Variante der äquivalenten Formulierungen, ausgegeben. Die METAL-Spezifikation von State befindet sich im Anhang.

[18] `exit` $\xrightarrow{\delta}$ `stop`

[19] Steht für die Liste, die jeden Ereignisnamen enthält

[20] Besitzt den Namen `std`, während der Default Pretty Printer, der nur die Prozeßinstanzen anzeigt, den Namen `small` besitzt

4.7 Transformationsfunktion

Nachdem die Zustandsrepräsentation bekannt ist, kann nun auf die Erzeugung des initialen Zustands eingegangen werden. Hierbei ist zu berücksichtigen, daß auch später in der Simulation für die Berechnung von Folgezuständen noch nicht transformierte `s_behaviour` Operatoren in Zustände transformiert werden müssen. Da per Definition `s_behaviour(Bexp)` auch ein Zustand ist, könnte die Transformation von `Bexp` bis zur Berechnung des Menüs aufgeschoben werden. Damit ergäbe sich der Initialzustand als `s_behaviour(B)`, wobei `B` die Behaviour-Expression des behaviour-parts der Spezifikation darstellt. Im vorgestellten Simulator wird die Zustandsrepräsentation aber direkt als Grundlage für die Anzeige des aktuellen Zustands verwendet[21]. Das bedeutet, daß der Initialzustand überhaupt nicht sichtbar wäre und generell Prozessinstantiierungen, die erst durch Transformation eines `s_behaviour`-Operators entstehen, nicht in der Anzeige des Zustands sichtbar wären, obwohl sie potentiell Ereignisse anbieten. Da während der Simulation nach der Übernahme eines neuen Zustands automatisch dessen Menü berechnet wird, d.h. das Aufschieben der Transformation dauert gerade bis zu diesem Zeitpunkt, kann man auch äquivalent sofort transformieren, so daß die *menu*-Funktion für den `s_behaviour`-Operator nur noch auf `s_behaviour(action_prefix($\dots$))` definiert werden muß. Damit ist eine Invariante der Zustandsrepräsentation bezüglich der Anwendung der *menu*-Funktion formuliert. Die *menu*-Funktion muß nur auf folgenden Operatoren definiert werden:

Klasse A: `s_choice, s_enable, s_par, s_hide, s_instance`

wenden die *menu*-Funktion rekursiv auf ihre Operanden an und kombinieren bzw. transformieren deren Ergebnisse zu ihrem Resultat.

Klasse B: `s_exit, s_stop`

liefern sofort das leere Menü, bzw. das Menü mit dem δ-Ereignis.

Klasse C: `s_behaviour(action_prefix(ActionDenotation, Bexp))`

liefert das Menü mit dem Ereignis `ActionDenotation`, das die Transformation von `Bexp` als Folgezustand enthält. Das Argument der *menu*-Funktion ist also immer ein Zustand, dessen innere Knoten aus Operatoren der Klasse A bestehen und dessen Blätter aus Operatoren der Klassen B und C, also Ereignisangeboten[22], bestehen. Zur Erhaltung dieser Invarianten muß der Folgezustand des δ-Ereignisses vom ersten Operanden des `enable`-Operators bereits die Transformierte des zweiten Operanden sein. Für die *expand*-Funktion, welche die Transformation von BASIC-LOTOS-Behaviour-Expression nach State-Zustandsrepräsentation vornimmt, gilt demnach:

- Behaviour-Expressions, die den State-Operatoren der Klasse A entsprechen, werden in ihre entsprechenden State-Operatoren übersetzt und die *expand*-Funktion wird rekursiv auf ihre Operanden angewendet.
- Die Behaviour-Expressions `stop` und `exit` werden nach `s_stop` bzw. `s_exit` übersetzt, beenden aber die Transformation.
- Bei der Behaviour-Expression `action_prefix` stoppt die Transformation mit der Übersetzung `s_behaviour(action_prefix($\dots$))`.

Wie bereits erwähnt, besteht der einzige, aber signifikante Unterschied des Zeitpunktes der Transformation darin, daß in der gewählten Alternative für jede berechnete Ereignismenge, inklusive der ersten, der volle Zustand mit allen Prozeßinstantiierungen, die nur potentiell oder tatsächlich Ereignisse beisteuern, darstellbar ist.

[21] Es ist auch möglich, eine separate Datenstruktur, die nur zur Anzeige dient, aus der Zustandsrepräsentation zu berechnen. Durch die flexible Formulierbarkeit von Pretty Printern in CENTAUR bietet es sich jedoch an, den Zustand direkt darzustellen

[22] Mit Ausnahme von `s_stop`, das die leere Ereignismenge anbietet

4.8 Menüfunktion

Vor der Behandlung der *menu*-Funktion wird deren Ergebnis, die Menge der möglichen Ereignisse, beschrieben. Ein Mengenelement besteht, wie schon erwähnt, aus einem Ereignisnamen und dem zugehörigen Folgezustand des Ereignisses. Für den Ereignisnamen gibt es zwei ausgezeichnete Belegungen:

- `_tick` steht für das δ-Ereignis. Der vorangestellte Underscore stellt sicher, daß diese Repräsentation des δ-Ereignis nicht durch Angabe einer gleichlautenden Action_Denotation in der Spezifikation erreichbar ist[23].

- `i` steht für das interne Ereignis. Benutzt der Spezifikator die Action_Denotation `"i"`, was zur gleichen Repräsentation führt, so hat er auch das interne Ereignis gemeint.

Der in Tabelle 1 definierte `hide`-Operator sieht die Umbenennung aller Ereignisse seines ersten Operanden, die in der Liste seines zweiten Operanden vorkommen, in das Ereignis i vor. Der Benutzer des Simulators hat aber möglicherweise ein Interesse daran, zwischen den in der Spezifikation explizit angegebenen internen Ereignissen und den erst durch Umbenennung entstandenen zu unterscheiden. Ferner wäre es nützlich, wenn die vor der Umbenennung gültigen Ereignisnamen noch verfügbar wären. Aus diesem Grund wird ein Mengenelement des Menüs durch einen Sichtbarkeitseintrag, der die Werte `external` und `internal` annehmen kann, erweitert. Die Funktion des `hide`-Operators besteht nun nicht darin, ggf. den Ereignisnamen zu ändern, sondern die Sichtbarkeit auf `internal` zu setzten. Demzufolge besitzen auch das interne Ereignis i und das δ-Ereignis des `enable`-Operators, falls sein erster Operand ein δ-Ereignis anbietet, die Sichtbarkeit `internal`. Der Pretty Printer für das Menü versieht Ereignise mit der Sichtbarkeit `internal` mit Klammern, so daß sowohl die Sichtbarkeit als auch der ursprüngliche Name erkennbar sind.

Die Repräsentation der Menge der möglichen Ereignisse ist also eine Liste von Tupeln mit den Komponenten Ereignisname, Sichtbarkeit und Folgezustand des Ereignisses. Die Eindeutigkeit jedes Mengenelements erlaubt die Darstellung als Liste mit der Übersetzung des Elementhinzunahme-Operators der Menge in den cons-Operator der Liste. Die METAL-Spezifikation von Menu befindet sich im Anhang.

Die zentrale Simulatorfunktion ist die *menu*-Funktion. Aus der Zustandsrepräsentation müssen alle möglichen Ereignisse mit deren Folgeverhalten berechnet werden. Läßt man sich durch die Ähnlichkeiten zwischen Typol und PROLOG inspirieren [BFLO86, vE86], ergibt sich ein erster Ansatz durch das Prädikat

```
menu(B, Event, B')
```

Der Zustand `B` wird vorgegeben und `Event` und `B'` werden durch das Ergebnis instantiiert. Für `menu` werden per Induktion über den syntaktischen Aufbau von State Klauseln definiert, die zu einem gegebenen Zustand `B` ein Ereignis `Event` und dessen Folgezustand `B'` berechnen. Der PROLOG Auswertungsmechanismus inferiert dann durch Backtracking alle Folgezustände zu einem Ereignis und alle Ereignisse des Ausgangszustand. Diese Vorgehensweise hat den Vorteil, daß man sich bei der Formulierung der Klauseln sehr nahe an der Definition der dynamischen Semantik des ISO-Standards halten kann, ist jedoch viel zu teuer, wie sich am parallel-Operator zeigen läßt. In

```
menu(s_par(B1, B2, Gids), Event, B'):-menu(B1, Event, B1'),
                                      is_in_Gids(Event, Gids),
                                      menu(B2, Event, B2'),
                                      combine_par(B1', B2', B').
```

wird für jedes Ereignis `Event` aus dem ersten Term, das die Synchronisationsbedingung des zweiten Terms erfüllt, *alle* möglichen Ereignisse und Folgeverhalten von `B2` im dritten Term neu berechnet. Darüber hinaus ist die gezeigte nur eine von drei Klauseln des parallel-Operators, die alle zum Inferenzprozeß herangezogen werden. Um dieses katastrophale Laufzeitverhalten zu umgehen, muß eine konstruktive Defintion der *menu*-Funktion gefunden werden. Der Satz der Inferenzregeln des ISO-Standards kann durch eine formale Transformation in eine algorithmische Definition umgewandelt werden [vE88].

[23] Action_Denotations in der Spezifikation müssen mit einem Buchstaben anfangen

Beispielhaft wird hier der `disable`-Operator transformiert, mit den Inferenzregeln:

$$\frac{B_1 \xrightarrow{a} B_1'}{B_1[> B_2 \xrightarrow{a} B_1'[> B_2} \quad , a \neq \delta$$

$$\frac{B_1 \xrightarrow{\delta} B_1'}{B_1[> B_2 \xrightarrow{\delta} B_1'}$$

$$\frac{B_2 \xrightarrow{a} B_2'}{B_1[> B_2 \xrightarrow{a} B_2'}$$

Einsetzen in die *menu*-Defintion ergibt mit $\cup$ als Mengenvereinigung:

$$menu(B_1[> B_2) = \{(a, B'[> B_2) \mid \exists B' : B_1 \xrightarrow{a} B', a \neq \delta\} \cup$$
$$\{(\delta, B') \mid \exists B' : B_1 \xrightarrow{\delta} B'\} \cup$$
$$\{(a, B') \mid \exists B' : B_2 \xrightarrow{a} B'\}$$

Werden die Ergebnismengen durch Applikationen von *menu* ersetzt, erhält man:

$$menu(B_1[> B_2) = \{(a, B[> B_2) \mid (a, B) \in menu(B_1), a \neq \delta\} \cup$$
$$\{(a, B) \mid (a, B) \in menu(B_1), a = \delta\} \cup$$
$$menu(B_2)$$

Selbstverständlich wird in einer Implementierung das zweifache Auftauchen der Menüberechnung von B_1 in einen einmaligen Aufruf der *menu*-Funktion übersetzt, dessen Ergebnis dann in einem Arbeitsgang in die spezifizierten Teilmengen aufgeteilt werden kann. Tabelle 3 enthält die vollständige Definition der *menu*-Funktion für die Zustandsrepräsentation. Interne Ereignisse werden geklammert dargestellt.

Argument	Ergebnis
s_stop	$\{\}$
s_exit	$\{(\delta, \mathrm{s\_stop})\}$
s_behaviour(s_prefix(i, Bexp)) s_behaviour(s_prefix(Event, Bexp))	$\{((i), expand(\mathrm{Bexp}))\}$ $\{(\mathrm{Event}, expand(\mathrm{Bexp}))\}, \mathrm{Event} \neq i$
s_choice(S_1, S_2)	$menu(\mathrm{S}_1) \cup menu(\mathrm{S}_2)$
s_enable(S_1, S_2)	$\{(a, \mathrm{s\_enable}(\mathrm{S}, \mathrm{S}_2)) \mid (a, \mathrm{S}) \in menu(\mathrm{S}_1), a \neq \delta\} \cup$ $\{(a, expand(\mathrm{S}_2)) \mid (a, \mathrm{S}) \in menu(\mathrm{S}_1), a = \delta\}$
s_disable(S_1, S_2)	$\{(a, \mathrm{s\_disable}(\mathrm{S}, \mathrm{S}_2) \mid (a, \mathrm{S}) \in menu(\mathrm{S}_1), a \neq \delta\} \cup$ $\{(a, \mathrm{S}) \mid (a, \mathrm{S}) \in menu(\mathrm{S}_1), a = \delta\} \cup$ $menu(\mathrm{S}_2)$
s_par(S_1, S_2, GidList)	$\{(a, \mathrm{s\_par}(\mathrm{S}, \mathrm{S}_2, \mathrm{GidList})) \mid (a, \mathrm{S}) \in menu(\mathrm{S}_1),$ $a \notin \{\mathrm{GidList}, \delta\}\} \cup$ $\{(a, \mathrm{s\_par}(\mathrm{S}_1, \mathrm{S}, \mathrm{GidList})) \mid (a, \mathrm{S}) \in menu(S_2),$ $a \notin \{\mathrm{GidList}, \delta\}\} \cup$ $\{(a, \mathrm{s\_par}(\mathrm{S}_1', \mathrm{S}_2', \mathrm{GidList})) \mid$ $(a, \mathrm{S}_1') \in menu(\mathrm{S}_1), (a, \mathrm{S}_2') \in menu(\mathrm{S}_2),$ $a \in \{\mathrm{GidList}, \delta\}\}$
s_hide(S, GidList)	$\{((a), \mathrm{s\_hide}(\mathrm{S}', \mathrm{GidList})) \mid (a, \mathrm{S}') \in menu(\mathrm{S}),$ $a \in \{\mathrm{GidList}\}\} \cup$ $\{(a, \mathrm{s\_hide}(\mathrm{S}', \mathrm{GidList})) \mid (a, \mathrm{S}') \in menu(\mathrm{S}),$ $a \notin \{\mathrm{GidList}\}\}$
s_instance(S, RelabelMap)	$\{(a', \mathrm{s\_instance}(\mathrm{S}', \mathrm{RelabelMap})) \mid (a, \mathrm{S}') \in menu(\mathrm{S}),$ $a' = relabel(a, \mathrm{RelabelMap})\}$[24]

Tabelle 3: Konstruktive Definition der *menu*-Funktion

[24] Mithilfe der RelabelMap werden die formalen Ereignisse a in die aktuellen a' umbenannt

4.9 Folgezustandsberechnung

Die gewählte Definition der *menu*-Funktion bedingt, daß die einzige Aufgabe der *next*-Funktion darin besteht, die Zustandskomponente des Argumentes in den aktuellen Zustand zu übertragen. Abbildung 7 ordnet alle Simulatorfunktionen in den Gesamtzusammenhang ein.

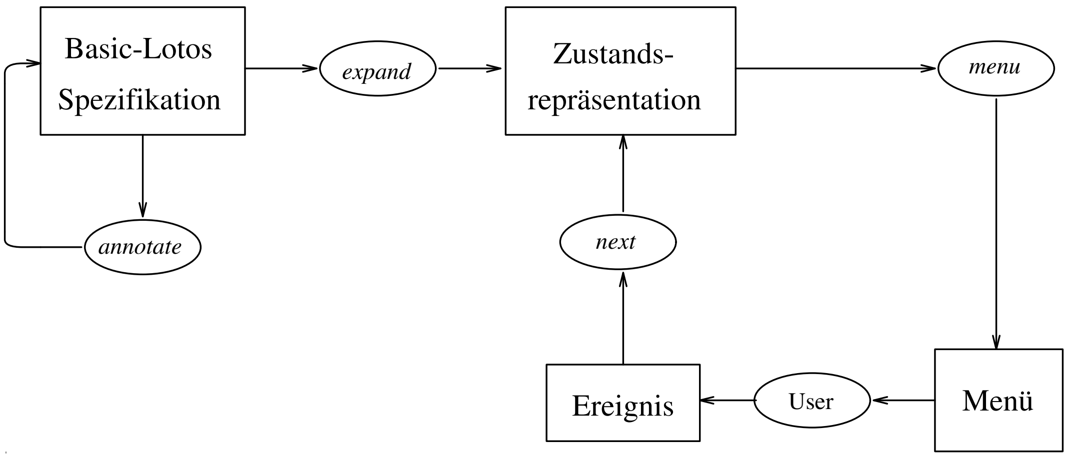

Abbildung 7: Systemfunktionen des Simulators

Die Pfeile geben die Datenflußrichtung an. Die *follow*-Funktion ist in dieser Darstellung in der *menu*-Funktion enthalten.

Abbildung 8 verdeutlicht die Benutzung der vorgestellten Sprachen untereinander als Teilsprachen.

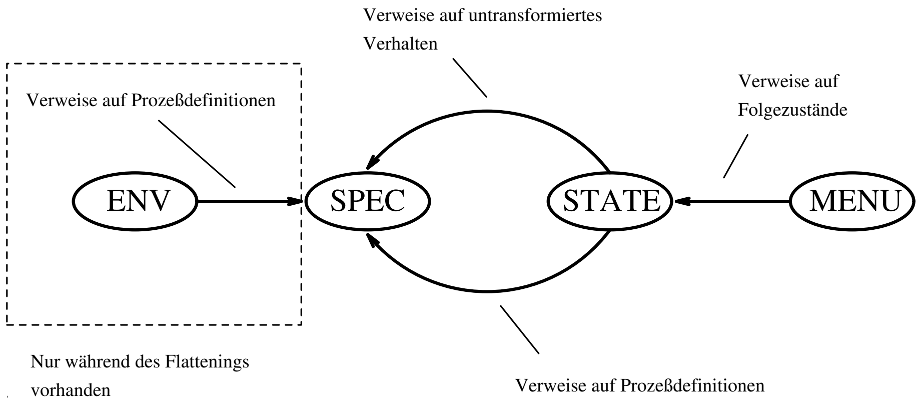

Abbildung 8: Abhängigkeit der Simulatordaten

Die Pfeile visualisieren die Benutzt-Relation der abstrakten Syntaxen. Beispielsweise ist eine Komponente eines Ereignisses der Sprache Menu ein Folgezustand aus der Sprache State. In METAL, das keine direkte modulare Sprachentwicklung erlaubt, gibt es dafür ein Terminal, das einen Baum speichern kann. Der betreffende Operator wird `implemented as tree`.

4.10 Optimierungen

Die Definition der *menu*-Funktion in Tabelle 3 ist noch verbesserungsfähig bezüglich der Berechnung der Folgezustände. In der folgenden Illustration wird der Übersichtlichkeit halber konkrete BASIC-LOTOS-Syntax statt der abstrakten Syntax der Zustandsrepräsentation verwendet.

> Gemäß
>
> (hide a in b; stop) $\xrightarrow{b}$ (hide a in stop)
>
> ist
>
> menu(hide a in b; stop) = (b, hide a in stop).
>
> Jedoch gilt
>
> (hide a in stop) $\equiv$ stop.

Ein `hide`-Operator, dessen Operand keine Ereignisse ermöglicht, kann selbst auch keine Ereignisse ermöglichen. Weitere Beobachtungen dieser Art kann man sich zu Nutze machen, um die Folgezustände auf äquivalente Zustände zu vereinfachen. Gleich zwei Vorteile ergeben sich durch genauere Beobachtung der Operator Verknüpfungen bei der Berechnung von Folgezuständen:

1. Die Zustände werden kleiner und beanspruchen somit weniger Speicherplatz. Im Extremfall verkürzt sich der Zustand einer Spezifikation auf `s_stop`, was einem Deadlock entspricht.
2. Die Menüberechnung für die Folgezustände wird beschleunigt, da keine überflüssigen Operatoren betrachtet werden müssen.

Die möglichst prägnante Formulierung der Zustandsvereinfachungen ist auch der Grund für die im vorherigen Abschnitt vorwegenommene Einführung der Abkürzungen `s_stop` und `s_exit`.
Die Lesbarkeit der TYPOL-Regeln erhöht sich drastisch, wenn

```
s_hide(s_behaviour(action_prefix(s_exit())), GidList) ->
s_behaviour(action_prefix(s_exit()))
```

durch

```
s_hide(s_exit(), GidList) -> s_exit()
```

abgekürzt werden kann. Tabelle 4 zeigt die in der Spezifikation realisierten Zustandsvereinfachungen.

Folgezustand	Vereinfachung
stop >> B	stop
stop [> B	B
stop par[25]stop	stop
exit par stop	stop
stop par exit	stop
exit par exit	exit
stop || B	stop
B || stop	stop
hide GidList in stop	stop
hide GidList in exit	exit
hide GidList_1 in (hide GidList_2 in B)	hide GidList in B, GidList=*union*(GidList_1, GidList_2)
proc_inst(stop, Id, RelabelMap)	stop
proc_inst(proc_inst(B, Id_2, Map_2), Id_1, Map_1)	proc_inst(B, Id_2, Map), Map=*combine*(Map_1, Map_2)

Tabelle 4: Äquivalente Zustandsvereinfachungen

[25]par steht für alle Ausprägungen des parallel-Operators: `||`, `|||`, `|[`$a_1,\ldots,a_n$`]|`

Es fällt auf, daß bei der unabhängigen Komposition durch den `|||`-Operator der Ausdruck A:= `stop ||| B` nicht zum Ausdruck A':= `B` vereinfacht wird. Der Grund dafür liegt darin, daß sich bei allen Ausprägungen des parallel-Operators beide Operanden im δ-Ereignis synchronisieren müssen. Der Unterschied zwischen den Ausdrücken A und A' ist also, daß A alle Ereignisse von A' bis auf die δ-Ereignisse anbietet, da diese sich mit dem `stop` nicht synchronisieren können. Kann auf die Information eines Typcheckers zugegriffen werden, der für `B` die Funktionalität `noexit` ermittelt, d.h. `B` kann nie ein δ-Ereignis anbieten, ist die Vereinfachung von A nach A' zulässig. Eine weitere nicht realisierte Vereinfachung, die sich anbietet ist

`proc_inst(s_exit, Id, RelabelMap)` $\rightarrow$ `s_exit`.

Damit würde aber die Prozessinstanz von `Id` aus der Zustandsdarstellung verschwinden, obwohl sie noch das δ-Ereignis des `exit` anbietet. Die Vereinbarung für die Zustandsdarstellung ist aber, daß alle Instanzen, die potentiell Ereignisse anbieten können, dargestellt werden.

Quasiverträglich mit dieser Formulierung ist die Zusammenfassung von direkt aufeinander folgenden Prozeßinstanzen. Tatsächlich bietet `Mother` in

`proc_inst(proc_inst(a; stop, Child, [a->b]), Mother, [b->c])`

ein Ereignis an und ist sogar essentiell für die Umbenennung des Ereignisses von b nach c zuständig. Auf der anderen Seite kann `Mother` zu keinen eigenen Ereignissen mehr beitragen. Weit wichtiger als die Unterscheidung der Herkunft von Ereignissen ist jedoch die Vermeidung der Zustandsvergrößerung durch unbeschränkte Rekursionen. Der unerschöpliche Wasservorrat der Prozeßdefinition

`Quelle[Wasser]:= Wasser; Quelle[Wasser]`

führt in der Simulation zu Ketten von `Quelle`-instanzen. Natürlich ist die Kettenlänge davon abhängig, wie oft der Benutzer des Simulators das Ereignis `Wasser` auswählt, aber *eine* Repräsentation von `Quelle`, die sich ggf. selbst neu instantiieren kann, ist prinzipiell ausreichend. Gleichzeitig wird durch das Zusammenfassen von direkt aufeinander folgenden Prozeßinstanzen eine Verhaltensäquivalenz auf eine Zustandsäquivalenz abbgebildet. Für das Verhalten einer LOTOS-Spezifikation ist es ohne Bedeutung, wieviele Instanzen von `Quelle` sich das Wasser reichen. Ohne Zusammenfassen der Instanzen ergeben sich jedoch lauter verschiedene Folgezustände bei der Ausführung von `Wasser`. Entspricht das Folgeverhalten einer Spezifikation nach einer Prozeßinstantiierung genau dem Verhalten davor, so spiegelt sich dieser Sachverhalt, bei Zusammenfassung der direkt aufeinander folgenden Prozeßinstanzen, in der Zustandsdarstellung wieder. Diesem Gewinn steht der Verlust der Vorfahren von Prozeßinstanzen, die maximal[26] noch Umbenennungen vornehmen, gegenüber. In der Realisierung wurde auf die Vorfahren verzichtet jedoch genügt die Änderung einer TYPOL-Regel und die Entfernung einer zweiten, um das alternative Verhalten zu erzielen.

Werden Prozeßinstanzen zusammengefaßt, deren Umbenennungen nicht der Identitäts-Funktion entsprechen, muß die in Tabelle 4 erwähnte *combine*-Funktion aus den beiden Umbenennungen eine berechnen, die beide ersetzt. Die *combine*-Funktion, die im allgemeinen Fall die vier Fallunterscheidungen

1. Ereignis a wird nicht umbenannt,
2. Ereignis a wird nur von der ersten Umbenennung nach a' umbenannt,
3. Ereignis a wird nur von der zweiten Umbenennung nach a' umbenannt,
4. Ereignis a wird nach a' und danach von a' nach a'' umbenannt,

berücksichtigen muß, wird durch folgende Beobachtung vereinfacht: Der Mutterprozeß kann ja nur daher durch Zusammenfassung gestrichen werden, da er selbst keine Ereignisse mehr anbieten kann. Damit ist die dritte Fallunterscheidung nicht mehr notwendig. Es genügt also, in der Umbenennung des Kindprozesses

$$[alt_1 \mapsto neu_1, \ldots, alt_n \mapsto neu_n]$$

die rechten Seiten $neu_1 - neu_n$ durch die Umbenennung des Mutterprozesses zu transformieren, um die Kombination der Umbenennungen zu erhalten. Durch obige Argumentation müssen danach keine Umbenennungskomponenten des Mutterprozesses mehr berücksichtigt werden.

[26] Eine Instanz `Bach` der Prozeßdefinition `Bach[Wasser]:=Quelle[Wasser]`, die ihre Existenz dem Aufruf in `Fluß[Wasser]:=Bach[Wasser]` verdankt, nimmt nur die identische Umbenennung Wasser $\mapsto$ Wasser vor

Zusammenfassend läßt sich festhalten, daß der Zustand folgenden Transformationen unterworfen ist:

- Er wächst durch
  - Prozeßinstantiierungen
  - Einsetzen von Folgezuständen des `action_prefix`
  - Einsetzen von Folgezuständen des `enable`.
- Er schrumpft durch
  - Konsumierung von Action_Denotations des `action_prefix`
  - Semantische Reduktionen, wie das Ersetzen des `disable`-Operators mit den Operanden Prozeß und Unterbrechungsprozeß durch den Unterbrechungsprozeß alleine, wenn ein Ereignis des Unterbrechungsprozesses ausgeführt wird.
  - Vereinfachung von Folgezuständen.

Die Tatsache, daß nur Folgezustände vereinfacht werden, hat für den initialen Zustand die Konsequenz, daß Redundanzen der Spezifikation noch sichtbar sind. So gibt es den *Initial*zustand

```
hide a in (hide b in a; b; c; stop)
```

mit den aufeinanderfolgenden `hide`-Operatoren, aber der *Folge*zustand von Ereignis (a) ist

```
hide a,b in b; c; stop.
```

Der besseren Lesbarkeit wegen wurde im Beispiel, und im folgenden, konkrete Syntax für die Zustandsdarstellung benutzt. Es gibt auch redundante Ausdrücke, die nie vereinfacht werden, weil sie keine Ereignisse anbieten und daher nie ein Folgezustand für sie berechnet wird. Das führt dazu, daß in

```
B || stop
```

u.U. das Menü von `B` immer wieder berechnet wird. Wäre der Teilausdruck `stop` während der Simulation entstanden, so hätte der Ausdruck nur zwischenzeitlich existiert und wäre sofort zu `stop` vereinfacht worden. Das folgende Beispiel dient zur Veranschaulichung der Folgezustandsvereinfachungen.

Der Folgezustand für das Ereignis a von

```
(a; stop ||| b; stop) || ((exit ||| a; stop) >> (i; exit))
```

wird wie folgt zu `stop` reduziert:

```
Ausgangszustand               (a; stop ||| b; stop)  ||  ((exit ||| a; stop)  ≫  i; exit)
nach Ausführung von a           stop                        exit |||  stop
nach interleave-Reduktion                                        stop         ≫  i; exit
nach enable-Reduktion          (stop  ||| b; stop)  ||                        stop
nach synchronise-Reduktion                          stop
```

Die Darstellung soll deutlich machen, daß die Vereinfachungen Bottom-up und in einem Durchgang erfolgen, im Gegensatz zur Methode der wiederholten Anwendung elementarer Vereinfachungen auf den ganzen Zustand.

5 Spezifikation

In diesem Abschnitt wird die Umsetzung des Entwurfs in die erstellte TYPOL-Spezifikation nachvollziehbar gemacht. Darüber hinaus finden einige spezielle Aspekte der Spezifikation mit TYPOL und deren Behandlung Erwähnung. Im folgenden wird zunächst auf die Form und dann auf den Inhalt der Spezifikation eingegangen.

5.1 Syntax

Die Spezifikation des Simulators enthält zwei formgebende Grundlagen:

- Den syntaktischen Aufbau der verwendeten Sprachen, der im Kapitel Entwurf schon vollständig motiviert wurde.
- Die typografische Gestaltung der Spezifikation innerhalb der in TYPOL vorgegebenen Grenzen.

Inhaltlich ohne Bedeutung, spielen das Layout und die Wahl der Symbolik für die Lesbarkeit der Spezifikation eine große Rolle. Isoliert betrachtet, besitzt

plus(Exp1, Exp2)

zwei Interpretationen. Es ist möglicherweise ein abstrakter Syntaxknoten `plus` mit den Söhnen `Exp1` und `Exp2`. Durch die Möglichkeit, statt Setname(Hypothese $\vdash$ Subjekt *rel-op* Resultat) im Falle des Fehlens einer Hypothese nur Setname(Subjekt *rel-op* Resultat) zu formulieren, ergibt sich die zweite Interpretation zu: Auf das Subjekt `Exp1` wird die Relation Komma in der Regelsammlung `plus` angewendet, um das Resultat `Exp2` zu erhalten. Der Fall erscheint konstruiert und leicht entscheidbar, in einer unbekannten Spezifikation kann die Identifizierung der Komponenten einer TYPOL-Regel gleichwohl sehr schwer fallen. Durch genaue Beobachtung der umgebenden Syntax läßt sich jedoch immer eine Fallentscheidung treffen. Unter einem gewissen Gesichtspunkt mag die Uniformität zwischen abstrakten Syntaxtermen und der Benutzung von Regelsammlungen sogar erwünscht sein. Warum soll zwischen einer Funktion (dem Setnamen) und einem Datum (dem abstrakten Syntaxknoten), das aufgrund seiner Interpretation eine Funktion impliziert, unterschieden werden? Da auch abstrakte Syntaxbäume dynamisch konstruierbar und zerlegbar sind, sind sie direkt mit den Aufrufhierachien der Regelsammlungen vergleichbar. In der vorliegenden Spezifikation wurde allerdings eine klar erkennbare Trennung zwischen den zu interpretierenden und manipulierenden Daten und den dazu benutzten Regelsammlungen angestrebt. Die METAL-Syntax gibt die Kleinschreibung der abstrakten Syntaxknoten vor, so daß alle Setnamen großgeschrieben wurden. Die dritte Bezeichnergruppe in TYPOL-Regeln, die Variablen, werden auch kleingeschrieben, beginnen aber mit einem Großbuchstaben. Damit sind alle drei Gruppen sofort optisch zu identifizieren. Die Typen von METAL-Operatoren müssen mit einem Großbuchstaben beginnen. Damit die eng mit den Operatortypen zusammenhängenden Variablentypen von den Variablen unterscheidbar sind, wurde die übliche vollständige Großschreibung für Typen gewählt.

Wo immer möglich wurde versucht, bei der Namensgebung von Variablen Zusatzinformation zu vermitteln. Statt eine Variable vom Typ `PMAP` nur `Map` zu nennen, wird durch `Local_Process_Map` der Verwendungszweck illustriert. Eine ebensolche Informationsvermittlung kommt auch den *rel-ops* in Typol zu. Der Relationsoperator *rel-op*, der zwischen Subjekt und Resultat steht, trägt keinerlei inhaltliche Bedeutung. Er dient nur dazu, Typol-Variablen von einander zu trennen. Trotzdem ist es sinnvoll, einen die Semantik der Aussage Subjekt *rel-op* Resultat wiederspielgelnden Operator zu definieren. Dies erfolgt zusammen mit der Festlegung der Typen einer Typol-Regel im `judgement` einer Regelsammlung. Durch den Umstand, daß keine Buchstaben für den Operator verwendet werden können und zudem einige Sonderzeichen bereits als Typol-Token belegt sind, ist die Auswahl der Operatoren sehr eingeschränkt. Immerhin darf der Operator Variablen enthalten, so daß z.B. `A  - Event >  A'` für den Operator, der den Übergang von einem Verhalten `A` zu einem Verhalten `A'` durch das Ereignis `Event` definieren soll, formulierbar ist. In der Spezifikation wurde folgende Vereinbarung festgelegt:

Name	Operator	Beschreibung
transform	`->`	Modifikation, Erweiterung eines Datums
translate	`=>`	Übersetzung in eine andere Sprache
compose	`>-`	Konstruktion eines Verbundes
retrieve	`-<`	Zugriff auf eine Verbundkomponente
results_in	`:`	Wert- oder Typermittlung
yield	`- arg >`	Liefert Ergebnis mit Berücksichtigung von arg
at	`@`	Kombination zwischen Ausdruck und Herkunft

Tabelle 5: Beschreibung der relational-Operatoren

Da die relational-Operatoren ohne Zusatzinformation bei weitem nicht selbst erklärend sind, wurde jede Regelform mit einer Beschreibung ihrer Interpretation versehen. Es ist eine Eigenheit (nicht nur) von Typol-Regeln, daß deren unmittelbarer Zugang nur über kommentierende Erläuterung möglich ist. Diese Interpretationshinweise sind Teil eines eigens entworfenen Dokumentationsformulars, das für jede Regelsammlung ausgefüllt wurde. Durch die Beschreibung der Regelsysteme in Funktion und Einordnung in das Gesamtsystem wurde auch dem letzten Formaspekt, der Dokumentation, genüge getan. Die oben genannten Formaspekte sollen zum Schluß noch an der Beispielregel

```
|- State => Menu &
HIDE_EVENTS(Gids |- Menu -> Menu')
-----------------------------------------
|- s_hide(State, gid_list(Gids)) => Menu';
```

aus der Spezifikation erläutert werden. Abstrakte Syntaxoperatoren wie `s_hide` und `gid_list` sind kleingeschrieben. Davon leicht zu unterscheiden sind die großgeschriebenen Setnamen wie `HIDE_EVENTS`. Die verbleibenden groß/klein-geschriebenen Ausdrücke `State`, `Menu`, `Menu'`, `Gids` und `State` sind Variablen. Die Regelsammlung MENU, aus der diese Regel stammt, nimmt eine Übersetzung eines Zustands in ein Menü vor. Deshalb besitzt sie den relational-Operator `=>`. `HIDE_EVENTS` verwendet `->`, weil es eine Menütransformation durchführt. Daraus ergibt sich die Namensgebung für `Menu'`, eine Modifikation von `Menu`. Die Beispielregel und das zu dieser Regelsammlung gehörige Dokumentationsformular ist im Anhang enthalten.

5.2 Semantik

Die im Kapitel Entwurf vorgestellten METAL-Spezifikationen lassen sich wie folgt aufteilen:

- Datensprachen - Sie definieren die Menge der wohlgeformten Simulatordaten, die nur Information aufbewahren, aber keiner Interpretation unterliegen. Hierzu zählen Env und Menu.
- Operatorsprachen - Durch sie werden die dem allgemeinen Verständnis einer Sprache entsprechenden abstrakten Grammatiken mit Operatoren und Operanden konstituiert. Im Simulator sind das Slotos[27] und State.

Der erste Sprachentyp dient zur Zwischenspeicherung von Ergebnissen, wie Env, und zur Repräsentation von Ergebnissen, wie Menu. Zwar sind auch für Env Funktionen definiert, jene zur Erweiterung und zum Durchsuchen des Environments, diese arbeiten jedoch auf einer starren abstrakten Syntax. Der zweite Typ der Sprachen vermindert die Voraussagbarkeit eines abstrakten Baumknotens auf einer der Typen, die die Klassen[28] einer Sprache definieren. Auf diesem Sprachtyp sind Funktionen definiert, die die Bedeutung der Operatoren auswerten, um ihre Ergebnisse zu konstruieren. Im Simulator sind das u.a. *expand* für Slotos und *menu* für State. Daraus läßt sich eine Strukturierung der dynamischen Spezifikation für die Operatorsprachen ableiten:

1. Induktion über den syntaktischen Aufbau der Sprache, d.h. für die zu findende Funktion wird für jeden Operator der Sprache das die Bedeutung des Operators reflektierende Ergebnis spezifiziert.
2. Bereitstellung der dazu notwendigen Hilfsfunktionen.

In der Übersicht von Tabelle 6 fehlen zur vollständigen Aufzählung noch zwei Dateien. **`member`** und **`follow_state`**, enthalten, wie aus der Importhierachie in Abbildung 9 hervorgeht, sekundäre Hilfsfunktionen. **`member`** enthält Hilfsfunktionen für die Funktionen in **`split`**, **`intersect`** und **`hide`**, die Testfunktionen für das Enthaltensein eines Ereignisses in einer Aufzählung oder Menge benötigen. **`follow_state`** ermöglicht den Funktionen in **`intersect`** und **`merge`**, den Folgezustand ihrer ermittelten Ereignisse zu berechnen.

Funktion	Argument	Definitionsdatei	primäre Hilfsfunktionen in
annotate	Slotos	`annot_spec`[29]	`pdef_access`, `process_env`
expand	Slotos	`expand`	`pdef_access`, `relabel`
menu	State	`menu`	`split`, `intersect`, `merge`, `expand`, `enable`, `hide`, `relabel`
state	Slotos	`state`	`expand`, `annot_spec`

Tabelle 6: Zuordnung der Spezifikationsdateien zu Funktionen

Die Funktionen *menu* und *state* sind die Schnittstellen zum Simulator. Die *state*-Funktion erzeugt den intialen Zustand, indem sie nacheinander *annotate* und *expand* auf die BASIC-LOTOS-Spezifikation anwendet.

[27] Small-LOTOS, definiert das LOTOS-Subset BASIC-LOTOS
[28] wie Statement und Expression
[29] Die Dateinamen werden ohne den Suffix `ty`, für typol, aufgeführt

Die Datei `follow_state` enthält nicht für alle Operatoren deren Spezifikation des Folgezustands. Soweit möglich wurden nämlich alle Hilfsfunktionen eines Operators in seine spezifische Datei konzentriert. So enthält die Datei `relabel`, die das Verhalten der *menu*-Funktion auf den `s_instance`-Operator spezifiziert, auch die Hilfsfunktionen, die eine Umbenennung erzeugen, Umbenennungen vornehmen, Umbenennungen zusammenfassen und jene, die das Folgeverhalten des `s_instance`-Operators definiert. Auf diese Weise sind singuläre Konzepte ihren Operatoren zugeordnet. Andererseits gibt es auch Gemeinsamkeiten der Operatoren, die eine Faktorisierung der Spezifikation dieser Gemeinsamkeiten motivieren. Betrachtet man die Definition der *menu*-Funktion *ohne* Folgeverhalten, so ergibt sich bspw. für die Operatoren `s_choice` und `s_disable` der gleiche Ausdruck $menu(op(S_1, S_2)) = menu(S_1) \cup menu(S_2)$. Es erscheint also sinnvoll, eine Funktion zum Vereinen von Menüs zu definieren, die durch den Operator *op* parametrisiert wird, um auch das operatorspezifische Folgeverhalten berechnen zu können. Genau das leisten die Regelsammlungen MERGE und INTERSECT, die in den gleichnamigen Dateien definiert sind. Für INTERSECT gibt es nur einen Anwendungsfall, den parallel-Operator, während in der Datei `merge` die restlichen zweistelligen Operatoren, bis auf `s_enable`, definiert sind. Der Operator `s_enable` wird gesondert behandelt, da er durch die Verzögerung der Auswertung seines zweiten Menüs der Gruppe der einstelligen Operatoren zugeordnet werden kann. Berücksichtigt man weiterhin, daß die Semantik des `s_choice`-Operators derart einfach ist, daß dafür eine simplere Hilfsfunktion, die auch an anderer Stelle zur Vereinigung von Teilmenüs Verwendung findet, benutzt werden kann, ergibt sich die Menge der Operatoren, deren Folgeverhalten in `follow_state` definiert ist.

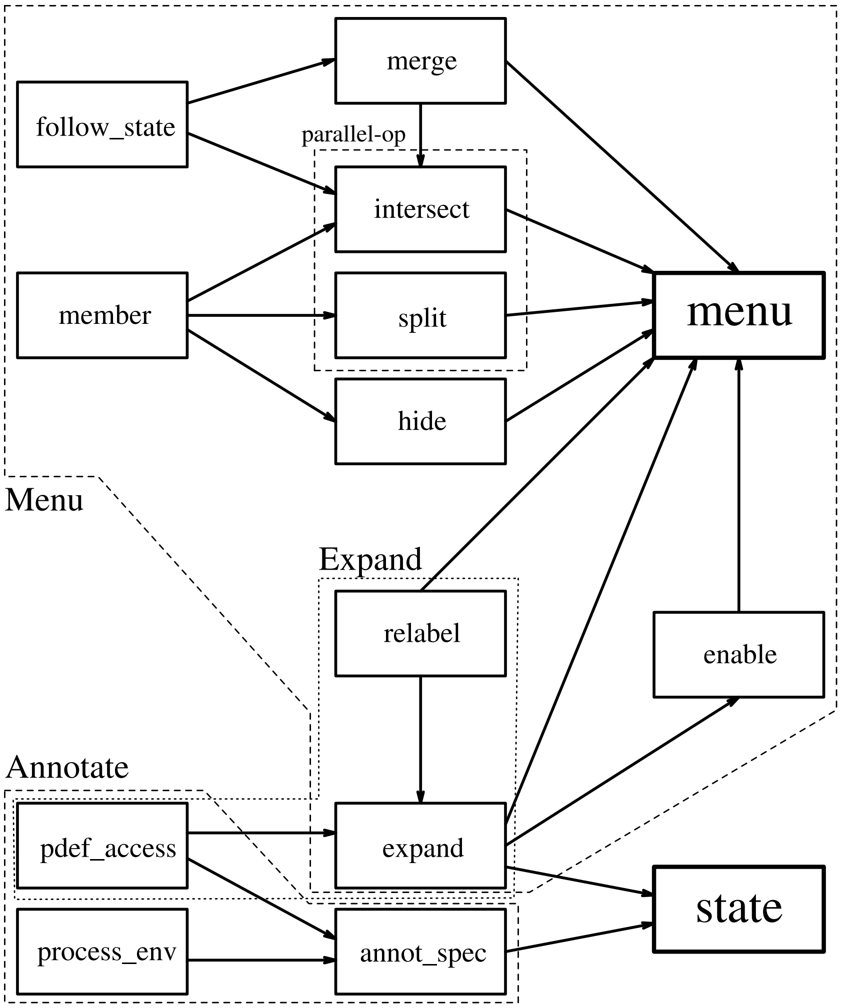

Abbildung 9: Importabhängigkeiten der Spezifikationsdateien

Innerhalb der Struktur der Importabhängigkeiten von Tabelle 9, die gleichzeitig eine Modulaufteilung widerspiegelt, da jede Datei ein Modul bildet, gibt es die Struktur der lokalen SETS. Eine Regelsammlung wurde immer so lokal wie möglich deklariert, damit schon durch die Deklarationsstelle die Menge der potentiellen Verwender hervorgeht. Zur globalen Modulstruktur bleibt zu vermerken, daß `process_env` die Verwaltung des Prozeßenvironments für `annot_spec` vornimmt und dafür `pdef_access` verwendet, das die Schnittstelle zur Speicherung von Zusatzinformation in einem abstrakten Syntaxbaum definiert. Daher wird `pdef_access` auch von `expand` zur Wiedergewinnung der Zusatzinformation, den Prozeßdefinitionen, importiert. `split` und `intersect` bilden zusammen ein virtuelles Modul für den parallel-Operator. Die Regelsammlung SPLIT unterteilt die Argumentmenüs des parallel-Operators in die unabhängigen und die zu synchronisierenden Ereignismengen.

Die Regelsammlung INTERSECT berechnet die Menge der Ereignisse des parallel-Operators, die sich synchronsieren müssen. Verkürzt dargestellt muß der Durchnitt aus den beiden Operandenmenüs und der Menge der zu synchronisierenden Ereignisse gebildet, die Schnittmenge in Mengen mit gleichen Ereignisnamen aufgeteilt und die Ereignisse in den aufgeteilten Mengen, die aus verschiedenen Menüs stammen, zu Ereignissen des parallel-Operators kombiniert werden. Dabei ist jedoch zu beachten, daß durch die Projektion der Ereignisse auf die Ereignisnamen zum Vergleich der Synchronisationsbedingung aus den Menümengen Multisets werden. In einer Teilmenge mit gleichen Ereignisnamen muß also jedes Ereignis aus einem Menü mit jedem aus dem anderen Menü kombiniert werden. Daraus folgt, daß die Menüberechnung bereits eines einzelnen parallel-Operators die Komplexität $O(n*m)$ besitzt, angenommen die Operanden bieten n respektive m kombinierbare Ereignisse gleichen Namens an. Werden x gleichartige parallel-Operatoren kombiniert, beträgt die Komplexität unter den oben genannten Bedingungen $O((m*n)^x)$. Damit besitzt die *menu*-Funktion potentiell expotentielle Komplexität, obgleich alle anderen Operatoren lineare Komplexität bezüglich der Anzahl der Ereignisse in den Argumentmengen haben. Im Anhang befindet sich eine Spezifikation, deren Ereignismenge durch dynamisches Instantiieren von parallel-Operatoren expotentiell mit der Anzahl bereits ausgeführter Ereignisse wächst. Die oben angegebene Vorgehensweise zur Ermittlung des Menüs des parallel-Operators dient nur zur Illustration und wurde viel kürzer durch zwei sich gegenseitig benutzende Regelsammlungen realisiert.

Die Spezifikation enthält Bestandteile, die nicht notwendig zur Beschreibung der dynamischen Semantik sind. Diese sind:

- Textausgaben zur Information des Benutzers, welche Teilaufgabe gerade bearbeitet wird.
- Überprüfungen der statischen Semantik.

In der Abwesenheit eines Typcheckers, der die Korrektheit der statischen Semantik im Voraus hätte garantieren können, werden die elementarsten Voraussetzungen, die zur Fortsetzung einer Simulation unabkömmlich sind, vom Simulator überprüft. Die Überprüfung erfolgt ohne Mehraufwand, nur durch Abfangen des Fehlschlagens des Beweises einer Regelsammlung. Durch Textausgaben gemeldet werden wenn

- ein verwendeter Prozeß keine Definition besitzt,
- sich die aktuelle und formale Parameterliste eines Prozesses in der Länge unterscheiden.

Ob dagegen ein mit der Funktionalität `noexit` deklarierter Prozeß diese auch besitzt oder doch mit `exit` terminiert, ist eine Frage der Typisierung, die für die Ausführung unerheblich ist.

5.3 Zusatzinformationszugriff

Vor der Behandlung spezieller Eigenschaften der Spezifikation bezüglich der Ausführung soll zunächst auf eine Problematik bei der Verwendung von Zusatzinformationen eingegangen werden. Im konkreten Fall besteht die Zusatzinformation aus den Prozeßdefinitionen, auf welche bei einer Prozeßinstantiierung zugegriffen werden muß. Wie schon im vorherigen Kapitel erläutert, muß beim Zugriff auf die Zusatzinformation die Wurzel des abstrakten Syntaxbaums, der die Zusatzinformationen enthält, bekannt sein. Im allgemeinen verursacht dieser Umstand keine Probleme, da zu Anfang der Ausführung einer Spezifikation die Wurzel bekannt ist und während des Beweises der Pfad, der von der Wurzel zum aktuellen abstrakten Syntaxknoten führt, anhand der Regelsubjekte konstruiert wird. In der Spezifikation befindet sich sinngemäß die Regel

$$\text{Map} \vdash \text{Pdefs} \rightarrow \text{Map'},$$

welche der üblichen Konvention der Induktion über den syntaktischen Aufbau entspricht. Das Vorhandensein von Prozeßdefinitionen unter der Voraussetzung eines Environments erzeugt ein verändertes Environment. Interpretiert man den Anwendungsbereich dieser Regel dagegen, daß ein Environment eine Transformation erfährt unter der Voraussetzung des Vorhandenseins von Prozeßdefinitionen, liest sich die äquivalente Regel:

$$\text{Pdefs} \vdash \text{Map} \rightarrow \text{Map'}.$$

Der entscheidende Unterschied zwischen den Alternativen ist das Subjekt. In der ersten Form entspricht der Ableitungsbaum des Beweises dem abstrakten Syntaxbaum, der auch die Zusatzinformationen enthält. Im zweiten Fall wird solange ein Environment bewiesen, bis die Voraussetzung aus der leeren Prozeßdefinition besteht. Der Pfad für den aktuellen abstrakten Syntaxknoten wird also entlang des sich verändernden Environmentes konstruiert und es besteht keine Möglichkeit, auf die Zusatzinformationen im aktuellen Teil der Prozeßdefinitionen zuzugreifen. Das gleiche Problem des falschen Subjektes das sich hier noch durch das zudem üblichere, andere Formulieren lösen ließ, erfordert im folgenden Fall spezielle Behandlung. In der sinngemäß formulierten Regel der *menu*-Funktion für den Fall des `action_prefix`

$$\frac{\text{withsubject(BexpLocation : EXPAND(Bexp} \rightarrow \text{State))}}{\begin{array}{l}\vdash \text{s\_behaviour(prefix(Name, Bexp), BexpLocation)} \rightarrow \\ \text{events[event(Name, external(), State];}\end{array}}$$

wird explizit mit dem TYPOL-Konstrukt `withsubject` das Subjekt für die weitere Auswertung vorgegeben. Die Ausführung arbeitet bis zu dieser Regel auf dem abstrakten Syntaxbaum eines States. In dem Fall, in dem die angegebenen Regel zutrifft, muß für das ermittelte Ereignis der Nachfolgezustand berechnet werden, wobei dazu der Operand `Bexp` des `prefix` mithilfe der Regelsammlung EXPAND aus der Sprache Slotos in die Sprache State transformiert werden muß. Bei dem Wechsel des abstrakten Syntaxbaumes geht das Subjekt verloren und muß deshalb explizit angegeben werden. Aus diesem Grund speichert der Operator `s_behaviour` nicht nur das Verhalten `Bexp`, sondern auch die Herkunft des Verhaltens `BexpLocation`.

5.4 Eigenschaften

Notwendigerweise ist die vorliegende Spezifikation eindeutig. Diese Eigenschaft wird sichergestellt durch die Bedingungen:

- Alle Regeln haben unterschiedliche Subjekte oder
- Regeln mit gleichen Subjekten besitzen disjunkte Prämissen.

Dadurch wird die mehrdeutige Interpretation gleicher Anfangsbedingungen ausgeschlossen.

In [AC90, ACG92] werden einige Eigenschaften von TYPOL-Spezifikationen definiert, die u.a. die Voraussetzungen für funktionale und inkrementelle Auswertung von TYPOL-Spezifikationen angeben. In den selben Texten wird die Verwendung der Terminologie der attributierten Grammatik auf TYPOL-Spezifikationen vereinbart, auf die einige Eigenschaften Bezug nehmen.

Nicht notwendig für die funktionale Auswertbarkeit, aber ebenfalls aussagekräftig sind die Eigenschaften *link-less* und *constraint-less*. Eine TYPOL-Spezifikation ist *link-less*, wenn für alle Regeln gilt, daß die Ergebnisse der Prämissen und das Subjekt der Regel weder Variablen untereinander teilen noch durch ein Prädikat miteinander verknüpft sind. Dadurch wird bspw. vermieden, daß an ein Ergebnis einer Prämisse durch das Ergebnis einer anderen Prämisse Anforderungen gestellt werden.

Eine TYPOL-Spezifikation ist *constraint-less*, wenn alle Eingabeattribute der Regeln an Variablen gebunden sind. Hiermit ist sichergestellt, daß die Prämissen keine Anforderungen an ihre Ergebnisse stellen, indem sie strukturierte Terme vorgeben und somit die Anwendbarkeit einer Regel zurückweisen können. Beide Eigenschaften verhindern also für bestimmte Fälle, daß die Nichtanwendbarkeit einer Regel erst dann festgestellt wird, wenn bereits Prämissen ausgewertet wurden.

Alle Regeln der vorgelegten Spezifikation sind *link-less* und die meisten sind *constraint-less*. Die Regeln, die nicht *constraint-less* sind, besitzen strukturierte Terme in ihren Hypothesen. Dadurch entfernt sich die Spezifikation von einer reinen attributierten Grammatik, aber auch diese Regeln sind unmittelbar anwendbar oder nicht.

Eine TYPOL-Spezifikation ist funktional auswertbar, wenn sie *Strongly Non Circular* und *safe* ist. Eine Spezifikation ist *Non Circular*, wenn für alle möglichen Beweisbäume gilt, daß die dazugehörigen Attributabhängigkeitsgraphen keine Zyklen enthalten. Dadurch wird garantiert, daß es immer eine Auswertungsreihenfolge für die Attribute gibt. Die Eigenschaft *Strongly Non Circular* fordert dies auch für die Überlagerung aller möglichen Beweisbäume, d.h. die Auswertungsreihenfolge der Attribute darf nicht vom Beweisbaum abhängen. Leider gibt es noch kein Werkzeug, das feststellt, ob eine TYPOL-Spezifikation *Strongly Non Circular* ist und die Auswertung per Hand ist viel zu aufwendig. Um zu zeigen, daß die vorgelegte Spezifikation funktional auswertbar ist, genügt es aber nur diejenigen Eigenschaften nachzuweisen, die aus den Eigenschaften *Strongly Non Circular* und *safe* für die funktionale Auswertbarkeit abgeleitet werden.
Die wichtigste aus der Eigenschaft *Strongly Non Circular* gefolgerte Forderung ist:

- Für jede Regel existiert eine Ordnung der Prämissen, so daß für alle $i<j$ das Ergebnis der Prämisse_i nicht vom Ergebnis der Prämisse_j abhängt.

Von der *safe* Eigenschaft wird benutzt, daß für jede Regel gilt:

- Die Menge der Ausgabevariablen ist in der Menge der Eingabevariablen enthalten.

Ohne diese Einschränkung ist es nicht möglich die Unifikation der PROLOG-Auswertung durch das Pattern Matching der funktionalen Auswertung zu ersetzen.

Die vorgelegte Spezifikation erfüllt also alle Voraussetzungen zur Ausführbarkeit durch einen funktionalen Auswertungsmechanismus, weil die spezielle Funktionalität der PROLOG Implementierung, die Unifikation und Backtracking ermöglicht, gar nicht verwendet wird. Steht einmal eine funktionale oder sogar inkrementelle Auswertungsstrategie für eine besondere Klasse von TYPOL-Spezifikationen zur Verfügung, die u.a. die Unifikation durch simples Pattern Matching ersetzt, ist die vorgelegte Spezifikation unter Gewinn an Ausführungseffizienz weiter verwendbar.

Im weiteren soll gezeigt werden welche Vorteile eine *constraint-less* Regel bringt und wie eine Regel zu einer *constraint-less* Regel transformiert werden kann. Dazu ein Beispiel aus der Regelsammlung für das Prozeßenvironment. Die übliche Formulierung für das Finden eines Prozeßdefinitionszugriffs in einem Environment mit mehreren Gültigkeitsbereichen lautet:

$$\frac{\text{FROM\_LOCAL\_ENV}(\text{LocalEnv} \vdash \text{Pid} \rightarrow \text{Paccess}, \text{true}())}{\text{penv}[\text{LocalEnv.GlobalEnv}] \vdash \text{Pid} \rightarrow \text{Paccess};}$$

$$\frac{\begin{array}{l}\text{FROM\_LOCAL\_ENV}(\text{LocalEnv} \vdash \text{Pid} \rightarrow \text{Paccess}, \text{false}())\ \&\\ \text{FROM\_GLOBAL\_ENV}(\text{GlobalEnv} \vdash \text{Pid} \rightarrow \text{Paccess}')\end{array}}{\text{penv}[\text{LocalEnv.GlobalEnv}] \vdash \text{Pid} \rightarrow \text{Paccess'};}$$

Der zum Prozeßbezeichner `Pid` gehörende Prozeßdefinitionszugriff `Paccess` ist gefunden, wenn eine Suche im lokalen Environment `LocalEnv` den Prozeßdefinitionszugriff `Paccess` fand und erfolgreich (`true()`) war. Ist die Suche jedoch nicht erfolgreich, wird die erste Regel zurückgewiesen und die zweite kommt zur Anwendung. Da beide Regeln das gleiche Subjekt haben, müssen die Prämissen disjunkt sein, d.h. die Prämisse, die sicherstellt, daß die Suche im lokalen Environment erfolglos war, *muß* mitaufgenommen werden. Diese Forderung ergibt sich auch alleine schon aus der Tatsache, daß die Reihenfolge der Regeln keine Bedeutung hat und eine Auswertungsreihenfolge nicht vorgegeben ist.

Am Beispiel wird deutlich, daß *constraint-less* TYPOL-Regeln schnellere Ausführung gewährleisten. Im obigen Beispiel muß das lokale Environment *zweimal* vollständig durchsucht werden, um das Ergebnis `false()` zu ermitteln. Im ersten Fall beim Versuch der Anwendung der ersten Regel, im zweiten Fall beim Beweis der ersten Prämisse der zweiten Regel, die deren Anwendbarkeit sichert. Es ist daher geschickter, nur einmal zu suchen und das Ergebnis sofort für eine Entscheidung zu verwenden. Dazu wird nur eine Regel für die Suche des Prozeßdefinitionszugriffs angegeben und eine neue Regelsammlung SEARCH eingeführt. In

$$\frac{\begin{array}{l}\text{FROM\_LOCAL\_ENV}(\text{LocalEnv} \vdash \text{Pid} \prec \text{Paccess} : \text{Found})\ \&\\ \text{SEARCH}(\text{GlobalEnv} \vdash \text{Pid}, \text{Paccess} \text{ -Found}\!\rightarrow \text{Paccess}')\end{array}}{\text{penv}[\text{LocalEnv.GlobalEnv}] \vdash \text{Pid} \prec \text{Paccess'};}$$

wird das lokale Environment `LocalEnv` nur einmal durchsucht und das Ergebnis der Suche, sowohl der Prozeßdefinitionszugriff `Pacces` als auch das Erfolgsflag `Found`, Variablen zugewiesen. In der Regelsammlung SEARCH wird nun abhängig vom Erfolgsflag der bereits gefundene Prozeßdefinitionszugriff `Pacces` weitergereicht oder, mithilfe des globalen Environments `GlobalEnv`, weitergesucht.
Demnach enthält die Regelsammlung SEARCH:

$$\_ \vdash \text{Pid}, \text{Paccess} \text{ -true}()\!\rightarrow \text{Paccess};$$

$$\frac{\text{FROM\_GLOBAL\_ENV}(\text{Env} \vdash \text{Pid} \prec \text{Paccess})}{\text{Env} \vdash \text{Pid}, \_ \text{ -false}()\!\rightarrow \text{Paccess};}$$

War die vorherige Suche erfolgreich (`-true()`→), wird das Environment nicht benötigt und der bereits gefundene Prozeßdefinitionszugriff nicht verändert. Im anderen Fall ist der gefundene Prozeßdefinitionszugriff ohne Belang und die Suche muß im globalen Environment fortgesetzt werden. Über den Vorteil des einmaligen Durchsuchens des lokalen Environments hinaus wird durch die neu eingeführte Regelsammlung SEARCH ein semantisches Konzept etabliert. Der Sachverhalt der lokalen Verdeckung bzw. die Erreichbarkeit von älteren Gültigkeitsbereichen wird explizit formuliert im Gegensatz zur impliziten, ersten Version, die nicht *constraint-less* ist. In einer speziellen Regelsammlung, wie SEARCH, sind nur die für die Entscheidung relevanten Attribute vorhanden und zum Verständnis muß nicht aus i.a. komplizierteren Fallunterscheidungen die eigentliche Essenz extrahiert werden.

Im Beispiel des if-then-else Konstrukts aus einer Sprache mit Statements

$$\frac{\begin{array}{l}\text{EVAL(Exp : Result) \&}\\ \text{IF(IfStat, ElseStat} \vdash \text{Result} \rightarrow \text{Stat)}\end{array}}{\text{ifstat(Exp, IfStat, ElseStat)} \rightarrow \text{Stat;}}$$

mit der Regelsammlung IF:

$$\text{IfStat, ElseStat} \vdash \text{true()} \rightarrow \text{IfStat;}$$

$$\text{IfStat, ElseStat} \vdash \text{false()} \rightarrow \text{ElseStat;}$$

spiegelt sich exakt die Statementauswahl der Expression `Exp` wieder. Dagegen wird in

$$\frac{\text{EVAL(Exp : true())}}{\text{ifstat(Exp, IfStat, ElseStat)} \rightarrow \text{IfStat;}}$$

$$\frac{\text{EVAL(Exp : false())}}{\text{ifstat(Exp, IfStat, ElseStat)} \rightarrow \text{ElseStat;}}$$

die Regelsammlung der Induktion über den syntaktischen Aufbau der Statements erweitert (durch die Belegung des Teilterms `Exp`) und wiederum wird der möglicherweise komplizierte Ausdruck `Exp` u.U. *zweimal* ausgewertet.

6 Erweiterungen

Inhalt dieses Kapitels ist die Beschreibung der Unzulänglichkeiten sowohl der Spezifikation als auch des daraus entstandenen Simulators. Es werden Ausblicke gegeben, wie diese zu beheben sind und durch welche Maßnahmen sinnvolle Erweiterungen der Spezifikation zu erreichen sind.

6.1 Simulator

Die Realisierung nur eines Teiles der Simulatorfunktionen, die in [TJW+92] beschrieben sind, wie z.B. *Automatic Movement* oder *Coverage Analysis*, ginge über den Rahmen dieser Arbeit weit hinaus. Trotzdem sollen hier einige Funktionen genannt werden, die einem Benutzer in der praktischen Arbeit nützlich sind.

Ohne Veränderung der spezifizierten Semantik, nur durch kleine Zusätze, könnte ohne weiteres, mithilfe der CENTAUR-Funktionalität, die Funktion *find-action-prefix* realisiert werden. Sie erlaubt dem Benutzer, zu einem Ereignis aus dem Menü die Herkunft innerhalb der Spezifikation zu bestimmen.

Während die Zustandsdarstellung einen Einblick in die Zukunft ermöglicht und das Menü die Gegenwart festlegt, fehlt dem Benutzer noch die Kontrolle der Vergangenheit. Ein *Simulation tree*, der die während der Simulation gewählten Ereignisse strukturiert aufbewahrt, könnte dem Abhilfe verschaffen. Eine Realisierung beträfe nur die Programmierung der Umgebung mit LISP.

Eine geringfügige lokale Erweiterung der Spezifikation könnte die einzige Auflage des Simulators an die Spezifikation beseitigen. Gegenwärtig muß die zu simulierende Spezifikation *guardly well defined* sein. Ein Prozeß darf sich nicht selbst instantiieren, bevor er nicht an einem Ereignis teilgenommen hat. Dies gilt natürlich nicht nur für die direkte Rekursion, sondern auch für die gegenseitige Rekursion von mehreren Prozessen. Das Beispiel

```
p[a]:= a; stop ||| p[a]
```

führt den Simulator in eine nicht terminierende Expansion der Prozeßinstantiierung `p[a]`. Wie man sieht, müßte das Menü allerdings auch unendlich viele Ereignisse a enthalten. Während des Flattenings könnten solche neuralgischen Stellen durch das Mitführen einer Aufrufhierachie identifiziert und durch ein spezielles Expansionsereignis ϵ entschärft werden. Das obige Beispiel wird also nach

```
p[a]:= a; stop ||| ϵ; p[a]
```

transformiert. Es ist jetzt dem Benutzer überlassen, wieviele Instanzen von `p` er durch Auswahl von ϵ zuläßt. Das Ereignis ϵ müßte, analog zum δ-Ereignis, keinen Kommunikationspartner benötigen, damit die Semantik von

```
p[a]:= a; stop || p[a]
```

nicht zu stark verändert wird. Unverändert produziert das Beispiel unendlich viele Prozeßinstanzen, aber nur ein Ereignis a. Müßte sich das ϵ-Ereignis durch den parallel-Operator synchronisieren wäre in

```
a; stop || ϵ; p[a]
```

von vorneherein kein Ereignis möglich. Im Beispiel ist auch im anderen Fall nie ein Ereignis außer ϵ möglich, da der Benutzer nicht unendlich oft expandieren kann. Bei einem realistischeren LOTOS Beispiel jedoch, in dem die Prozeßinstantiierung bspw. von Daten abhängen kann, ist dem Benutzer die Möglichkeit gegeben, bis zum Punkt der Rekursionsterminierung zu expandieren, während der Simulator ohne ϵ-Mechanismus stur und damit endlos expandieren müßte, da er über die Daten keine Annahmen machen kann. Will man also LOTOS-Spezifikationen, die nicht *guardly well defined* sind, simulieren, ist das durch eine gezielte Modifikation der Spezifikation möglich, ohne daß die Semantik der Spezifikation dabei zu stark verändert wird.

Eine weitere nützliche Information für den Benutzer eines Simulators bietet die Funktion *Recursion Detection*. Wann immer sich ein Zustand während der Simulation in einer Ereignisfolge wiederholt, wird das durch den Simulator gemeldet. Dazu ist eine Normalisierung des Zustands nötig, um möglichst viele äquivalente Zustände auch durch einen Vergleich zu erkennen. Eine Art Normalisierung stellen schon die vorhandenen Zustandsvereinfachungen da. Um aber beispielsweise die Assoziativität des parallel-Operators zu berücksichtigen, bietet sich eine Repräsentation des parallel-Operators der Form

(|| P_1, P_2, P_3, ...)

an, wobei es auf die Reihenfolge der Prozesse nicht ankommt. Damit sind die Prozesse

(P_1 || P_2) || P_3 und P_1 || (P_2 || P_3)

als äquivalent erkennbar. Die konkrete Realisierung einer Normalisierungfunktion ist allerdings mit beträchtlichen Aufwand verbunden.

Eine entscheidende Aufwertung erfährt der Simulator durch die Verallgemeinerung von Basic-Lotos nach Lotos. Hier eröffnen sich die Arbeitgebiete

- Erstellen eines Funktionspakets für die abstrakten Datentypen aus ACT_ONE. Dieses beinhaltet Funktionen zur Normalisierung von abstrakten Datentypen, Finden von Variablenbelegungen, um einen Ausdruck nach `true` reduzieren zu können, usw.,
- Anpassung der Datenstrukturen des Simulators an die erweiterte Funktionalität,
- Erweiterung der Spezifikation der dynamischen Semantik.

Die abstrakten Datentypen an sich sollen hier nicht weiter behandelt werden, da das *term rewriting* ein abgeschlossenes Thema für sich darstellt. Bezüglich der Datenstrukturen, die neu für das Flattening zu entwerfen wären, empfiehlt es sich, zunächst einen Typchecker bereitzustellen, dessen Ausgabe die Grundlage der Simulation ist. In [Eer90] wird ein Zwischenformat, die *Common Representation*, vorgeschlagen, das natürlich auch Rückbezüge zur Originalspezifikation zulassen muß. Hierdurch wird ein weiteres Themengebiet, das der Typauflösung und Bezeichneridentifikation, abgegrenzt.

In Lotos sind die Ereignisse strukturiert, d.h. sie enthalten zusätzlich zum Ereignisnamen eine Liste von *experiment offers*. Ein *experiment offer* besteht entweder aus dem Senden $a!E$ eines Wertes, der sich durch Auswerten der Expression E ergibt, oder aus dem Empfangen $a?x : t$ eines Wertes vom Typ t, der der Variablen x zugewiesen wird. Für die Kombination von Ereignissen durch den parallel-Operator gilt:

Prozeß A	Prozeß B	Interaktionstyp	Bedingung & Effekt
$a!E_1$	$a!E_2$	value matching	value(E_1)=value(E_2) Synchronisation
$a!E$	$a?x : t$	value passing	value(E) $\in$ domain(t) x=value(E)
$a?x : t$	$a!E$	value passing	value(E) $\in$ domain(t) x=value(E)
$a?x : t$	$a?y : u$	value generation	$t = u$ $x = y = v,\ v \in domain(t)$

Tabelle 7: Interaktionstypen in Lotos

Die Simulatorereignisse müssen also durch die *experiment offers* ergänzt werden. Darüber hinaus können durch den allgemeinen `choice`-Operator Variablen eingeführt werden. In

```
choice x:int [] a!2*x; B
```

sind die Ereignisse `<a, 2>, <a, 4>, <a, 6>, ...` möglich. Deshalb muß ein Ereignis die *experiment offers* und die sogenannnten *choice parameters* enthalten.

Die letzte Erweiterung der Simulatorereignisse betrifft die Möglichkeit, in LOTOS Wächter zu verwenden. Das Ereignis a in

```
[x>5] -> a?x:int; B
```

wird nur dann angeboten, wenn die Bedingung des Wächters, in diesem Fall $x > 5$, wahr ist. Aus diesem Grund enthält ein Simulatorereignis für LOTOS zusätzlich ein Liste von Bedingungen. Zusammenfassend gilt jetzt

$$\text{Ereignis} = \begin{pmatrix} \text{Ereignisname} \\ \text{extern/intern Kennung} \\ \textit{experiment offers} \\ \textit{choice parameters} \\ \text{Bedingungen} \\ \text{Funktion auf einen Folgezustand} \end{pmatrix}$$

Die letzte Komponente ist jetzt eine *Funktion* auf einen Folgezustand, da in

```
g?x:t; B
```

das Folgeverhalten `B` erst durch die Belegung von x festgelegt wird. Die Wertzuweisung an x erfolgt aber nicht während der Menüberechnung, sondern erst später, nach der Ereignisauswahl durch die *next*-Funktion. Hiermit ist auch die Erweiterung der Funktionalität der *next*-Funktion angesprochen. Sie muß nach der Auswahl eines Ereignisses ggf. den Benutzer zur Angabe von Werten auffordern, die Erfüllbarkeit der Ereignisbedingungen durch diese Werte prüfen und bei Erfolg die Funktion auf den Folgezustand auswerten.

Das Auswerten der Folgezustandsfunktion könnte bspw. bedeuten, in

```
a?x:int; P(x)
```

nach der Ausführung von $< a, 0 >$ den Folgezustand

```
P(0),
```

durch Ersetzen aller x nach 0 zu berechnen. Um das sehr kostspielige Ersetzen zu vermeiden, kann man *Value Environments* einführen. Der `s_behaviour`-Operator wird durch ein *Value Environment* erweitert, um die aktuellen Variablenbelegungen bereitzustellen. Während der Zustandskonstruktion wird ein *Value Environment* aufgebaut, daß dann in den Verweisen auf den statischen Zustand aufbewahrt wird, weil im dynamischen Zustand selbst keine Variablen mehr vorhanden sind. In der Notation (VE, *Bexp*) wird aus

$([x \mapsto 2]$, `a?y:int;` $\mathtt{B}_1$ `|||` `b?y:int;` $\mathtt{B}_2)$

nach der Ausführung von $< a, 2 >$ und $< b, 3 >$

$([x \mapsto 2, y \mapsto 2]$, $\mathtt{B}_1)$ `|||` $([x \mapsto 2, y \mapsto 3]$, $\mathtt{B}_2)$.

Zusammenfassend werden zur Erweiterung des Simulators für LOTOS benötigt:

- Modul für die abstrakten Datentypen
- Typchecker mit Schnittstelle zum Simulator
- Erweiterung der Simulatorereignisse
- Erweiterung der Zustandsrepräsentation durch *Value Environments*
- Erweiterung der Funktionalität von *menu* und *next*.

Eine vollständige Abhandlung über LOTOS-Werkzeuge allgemein, einen LOTOS-Simulator inbegriffen, findet sich in [vE88].

6.2 Spezifikation

Es ist eine leichte Übung, die *menu*-Funktion auch für Behaviour-Expressions zu definieren, um so den `choice`-Operator aus dem dynamischen in den statischen Zustand zu überführen. Allerdings ergibt sich dann das Problem, daß in der Zustandsdarstellung möglicherweise Prozeßinstanzen fehlen, die zum Menü des `choice`-Operators beigetragen haben.

Schon früher, im Abschnitt Optimierungen, wurde die Möglichkeit erwähnt, den Ausdruck

```
stop ||| B
```

zu `B` zu reduzieren, falls sichergestellt werden kann, daß `B` die Funktionalität `noexit` besitzt. Steht kein Typchecker zu Verfügung, der diese Information liefern könnte, besteht die Alternative, einen neuen State-Operator einzuführen, dessen Menü das Menü seines Operanden, bis auf die δ-Ereignisse, ist. Das Beispiel würde zu

```
delta_filter(B)
```

reduzieren. Der Gewinn an Ausführungsgeschwindigkeit, im Hinblick auf eine gesamte Spezifikation, ist eher gering bis minimal einzuschätzen.

Noch zweifelhaftere Bedeutung für die Praxis besitzt der Ansatz, schon den Initialzustand zu vereinfachen, statt dies nur mit Folgezuständen zu tun. Ein Gewinn entsteht nur dann, wenn sich im Initialzustand viele oder große Ausdrücke befinden, die nie zu einem Ereignis beitragen können. Für diese wird nämlich nie ein Folgezustand berechnet und damit verbleiben sie immer unnötigerweise im Zustand. Tatsächlich tritt der Fall aber nur bei fehlerhaften Basic-Lotos-Spezifikationen auf.

Eine meßbare Verbesserung der *menu*-Funktion bezüglich der Ausführungsgeschwindigkeit würde die Aufteilung des Menüs in interne und externe Ereignisse bewirken. Bei jeder Menüberechnung eines parallel-Operators werden zunächst beide Operandenmenüs aufgeteilt in Menüs mit externen Ereignissen, die sich eventuell synchronisieren müssen, und Menüs mit internen Ereignissen, die keine Kommunikationspartner mehr benötigen. Diese wiederholte Auftrennung und die Zusammenmischung nach dem Berechnen der Teilmenüs könnte eingespart werden, wenn das Menü in ein externes und ein internes Untermenü aufgeteilt wäre. Gleichzeitig könnte die intern/extern Kennung aus den Ereignissen entfernt werden, da diese Information jetzt in der Untermenüzugehörigkeit steckt. Die Realisierung dieser Idee wurde nur deshalb nicht unternommen, da

- sich die Datenstrukturen nicht zu weit vom ISO-Standard entfernen sollten,
- in eine formale Spezifikation der dynamischen Semantik für einen Prototypen nur dann Implementierungsspezifische Optimierungen einfliessen sollten solange sie die Klarheit der Spezifikation nicht beeinträchtigen.

In die gleiche Klasse von Optimierungen fällt der Gedanke, Menüs nicht einfach als Liste von Ereignissen, sondern mit faktorisierten Submenüs zu repräsentieren. Das Menü des Ausdrucks

```
(a; stop [] a; stop) || (a; stop [] a; stop) || (a; stop [] a; stop)
```

enthält acht Ereignisse, die jeweils drei Folgeverhalten kombinieren. In komprimierter Darstellung mit Submenüs wären nur sechs Einträge zu speichern. Das Menü

```
{a a {a a {a a}}}
```

enthält die Ereignisse, die den Synchronisations-Operatoren im Beispiel entsprechen. Ein solchermaßen faktorisiertes Menü müßte zur Anzeige *ausmultipliziert* werden und bei der Auswahl eines seiner Ereignisse müßte die aufgeschobene Berechnung des Folgeverhaltens einsetzen. Hieraus folgt die Notwendigkeit, für ein Menü mit aufgeschobener Kombination seiner Ereignisse Zusatzinformation über die Folgezustandsrekonstruktion von der Zustandswurzel bis zur Ereigniskombination, mitzuführen. Normalerweise wird der Folgezustand eines Ereignis durch jeden Operator transformiert. Für aufgeschobene Kombinationen müßte die gesamte Transformation nur einmal für alle zu kombinierenden Ereignisse gespeichert werden. Gelingt die konsequente Durchführung dieses Ansatzes, ist neben der Speicherplatzersparnis für die Menürepräsentation auch die Komplexität der *menu*-Funktion, bezüglich der Anzahl Operatoren im Zustand, von expotentiellem auf polynomiales Verhalten reduziert.

Das aufwendige Kombinieren der Ereignisse eines parallel-Operators mit dem Aufwand $O(n*m)$, bei respektive m und n gleichartigen Ereignissen der Operanden, reduziert sich zu O(1), da einfach ein Submenü als Submenü niedrigster Priorität des anderen Submenüs eingetragen werden kann. Das gilt allerdings nur für die bereits identifizierten Teilmenüs mit gleichartigen Ereignissen. Da im ersten Schritt erst alle Ereignisse aus den Operandenmenüs in interne, unabhängige und zu synchronisierende Ereignissse aufgeteilt werden müssen, reduziert sich die Komplexität der *menu*-funktion des parallel-Operators nur auf $O(n+m)$ und entspricht damit der Komplexität aller anderen Operatoren. Erst wenn bei der Anzeige des resultierenden Menüs alle Ereignisse einzeln erscheinen sollen, ergibt sich ein expotentieller Aufwand für die Anzeige des letzten Menüs. Selbstverständlich entsteht auch für jedes ausgewählte Ereignis aus einem Submenü expontentieller Aufwand zur Folgezustandsberechnung.

Wesentlich einfacher im Ansatz und in der Realisierung ist die Implementierung der *menu*-Funktion als Memo-Funktion. Wenn für einen Teilbaum des Zustands ein Menü berechnet wurde, kann dieses, zu diesem Teilbaum gehörig, abgespeichert werden und muß erst dann wieder neu berechnet werden, wenn sich der Teilbaum durch Ereignisausführung geändert hat. Dazu müßte die Menüberechnung, neben dem Folgezustand, auch ein Folgemenü berechnen. Sind zu den Operanden eines Operators alle Folgemenüs vorhanden, so ist auch sein bereits berechnetes Folgemenü noch aktuell. Wird ein Teilbaum durch Ereignisausführung verändert, werden alle auf dem Transformationspfad liegenden Folgemenüs gelöscht. Die nächste Menüberechnung muß dann für die Operanden ohne Folgemenü das Menü neuberechnen und mit den u.U. noch vorhandenden Folgemenüs des betreffenden Operators neu kombinieren. Auch diese Vorgehensweise ist eher für eine effiziente Kodierung in einer Implementierungssprache geeignet.

Im Abschnitt Optimierungen des Kapitels Entwurf wurden die Vorzüge der Zusammenfassung von aufeinander folgenden Prozeßinstanzen ausgeführt. Es gibt jedoch Spezifikationen, bei denen die Verkürzung prinzipiell möglich ist, aber nicht durch den spezifizierten Simulator durchgeführt werden kann. Dazu der Beispielzustand

```
proc_inst(hide b in (b; stop [] a; p[a]),
          p, [a->a]).
```

Das Ereignis a hat den Folgezustand

```
proc_inst(hide b in (proc_inst(hide b in (b; stop [] a; p[a]),
                              p, [a->a])),
          p, [a->a]).
```

Es ist leicht zu erkennen, daß der `hide`-Operator der Zusammenfassung im Wege steht, obwohl er überflüssig ist, weil sein Operand nur noch Ereignisse a, aber nie b liefern kann. Eine Möglichkeit ist es, auf diesen Spezialfall zu testen und ggf. das überflüssige `hide` zu entfernen. Ein anderer allgemeinerer Ansatz ersetzt die Vorkommen von

```
hide GidList in (proc_inst B, ProcId, RelabelMap)
```
durch
```
proc_inst(hide GidList' in B, ProcId, RelabelMap')
```

Das bewirkt, daß alle `hide`-Operatoren in dem Zustand nach unten wandern, während alle `proc_inst`-Operatoren nach oben befördert werden. Auf diese Art sammeln sich gleichartige Operatoren und werden dann, durch die bereits vorhandenen Vereinfachungen, zusammengefaßt. Nebenbei wird hierdurch eine weitere Zustandsnormalisierung definiert, die aber den Zustand für den Benutzer, möglicherweise nicht nachvollziehbar, verändert. `GidList'` entsteht aus `GidList`, indem alle linken Seiten der Umbenennungen in `RelabelMap` in `GidList'` eingetragen werden, deren rechte Seiten in `GidList` vorkommen. In `RelabelMap'` können genau diese Umbenennungen, gegenüber `RelabelMap`, eingespart werden, da interne Ereignisse nicht mehr umbenannt werden. Durch diese Transformation ändern sich allerdings die mitgeführten Namen der Ereignisse, die im unveränderten Zustand noch durch `RelabelMap` umbenannt und dann erst, durch den `hide`-Operator, versteckt wurden. Legt man Wert auf die Originalnamen der internen Ereignisse[30] verbietet sich dieser Ansatz oder er muß, durch komplizierte Maßnahmen derart erweitert werden, daß *nur* die Ereignisse, die explizit durch `GidList'` versteckt werden, doch noch in `RelabelMap'` umbenannt werden.

[30] Streng genommen haben sowieso alle internen Ereignisse den Namen i

Abschließend läßt sich noch anmerken, daß sich die Ausführungsgeschwindigkeit des Simulators durch Ersetzen einiger grundlegender Regelsammlungen mit schnellen Primitiven sicher steigern ließe. Aber die Primitive, in einer Implementierungssprache geschrieben, müßten bereitgestellt werden und außerdem sollte die Spezifikationssprache TYPOL verwendet und nicht ein Simulator mit Produktqualität erstellt werden. Zu diesem Thema nur noch die Diskussion der Regelsammlung JOIN. Sie hat die Aufgabe, zwei Menüs zu einem Menü zusammenzumischen. Dazu wird von einem Menü solange ein Ereignis zu dem anderen Menü transferiert, bis das Quellenmenü leer ist. Die Methode erweist sich als sehr unglücklich, wenn das Quellenmenü sehr viele und das andere Menü nur sehr wenige Ereignisse enthält. Wäre der Fall zu erkennen, könnten Quellen- und Zielmenü vorher vertauscht werden. Dies ist jedoch in TYPOL nicht möglich ohne den Menütyp durch einen Elementzähler zu erweitern, genauso wenig wie die beiden Listen einfach hintereinander zu hängen. Diese von der Spezifikation her uninteressante Operation könnte durch ein schnelles Primitiv durchgeführt werden. Das für TYPOL zugängliche PROLOG Prädikat `appendtree` ist allerdings nicht schneller, da es keine direkte Manipulation der Listen vornimmt, sondern über PROLOG Klauseln definiert ist. Eine andere Methode, die Menüs zu mischen, besteht darin, jeweils ein Ereignis aus jedem Menü zu entnehmen und einem dritten zuzuordnen. Ist irgendwann eines der beiden Menüs leer, so kann das andere als Listende des dritten Menüs verwendet werden und durch Auflösen der Rekursion werden die zuvor aus den beiden Menüs entnommenen Ereignisse dem dritten Menü zugeführt. Diese Vorgehensweise ist zwar schneller, hat aber den Nachteil, daß die Menüs wirklich vermischt werden, d.h. aus der Reihenfolge der Ereignisse im Menü ist nicht mehr zu erahnen, aus welchem Teil der Spezifikation die Ereignisse stammen. Werden die Menüs tatsächlich hintereinander gemischt, so kann den Ereignissen im Menü $\{aaaa\}$ sofort angesehen werden, woher sie in

```
(a; stop [] a; stop) ||| (a; stop [] a; stop)
```

stammen, indem sie einfach von links nach rechts den Ereignisangeboten zugeordnet werden. Auch bei vermischten Ereignissen ist es mithilfe der im Abschnitt Simulator erwähnten Funktion *find-action-prefix* möglich, die Ereigniszuordnung vorzunehmen, jedoch muß die Funktion zur Verfügung stehen. Zudem muß bei der Benuzung der Funktion solange ausprobiert werden, bis das gewünschte Ereignis gefunden ist, während geordnete Ereignisse zielsicheres Auswählen ermöglichen. Ein Kompromiß stellt die letzte vorzustellende Variante dar. Auch hier wird den Menüs jeweils ein Ereignis entnommen, bis ein Menü leer ist. Dann wird das andere *vollständige* Menü als Listenende des Ergebnismenüs verwendet und durch Rekursionsauflösung wird sukzessive das richtige der vorher entnommenen Ereignisse zugefügt. Jetzt bleiben die Teilmenüs in ihrer Reihenfolge erhalten, aber im Ergebnismenü steht immer das kürzere vor dem längeren Teilmenü. Als Veranschaulichung der Spezifikationsmethode TYPOL für eine effizientere Listenvereinigung und zum Vergleich mit der Teilmenüfolge erhaltenen Regelsammlung JOIN in der Spezifikation folgt nun die Regelsammlung FAST_JOIN:

Regelsammlung JOIN:

$$\frac{\mathrm{FAST\_JOIN(Left, Right, Left, Right} \rightarrow \mathrm{\_, Menu)}}{\vdash \mathrm{Left, Right} \succ \mathrm{Menu;}}$$

Regelsammlung FAST_JOIN:

$\vdash$ events[], _, _, Right $\rightarrow$ left(), Right;

$\vdash$ _, events[], Left, _ $\rightarrow$ right(), Left;

$$\frac{\vdash \mathrm{Ltail, Rtail, Left, Right} \rightarrow \mathrm{Shorter, Menu}\ \&\quad \mathrm{ADD\_RIGHT(Lhead, Rhead} \vdash \mathrm{Shorter} \rightarrow \mathrm{Event)}}{\vdash \mathrm{events[Lhead.Ltail], events[Rhead.Rtail], Left, Right} \rightarrow \mathrm{Shorter, events[Event.Menu];}}$$

Regelsammlung ADD_RIGHT:

LeftEvent, _ $\vdash$ left() $\rightarrow$ LeftEvent;

_, RightEvent $\vdash$ right() $\rightarrow$ RightEvent;

Die Variablen `Left` und `Right` enthalten konstant die beiden ursprünglichen Teilmenüs, um in den zwei Axiomen von FAST_JOIN als Listenende des neuen Menüs dienen zu können. In diesen beiden Axiomen wird durch `left()` und `right()` indiziert, welches Teilmenü das kleinere war und damit welches von den zuvor extrahierten Ereignissen anschließend zum Aufbau des neuen Ergebnismenüs, in ADD_RIGHT, benutzt werden muß.

Die geführte Diskussion verlangt noch eine Nachbetrachtung. Es konnten überhaupt nur deshalb verschiedene Versionen entworfen werden, weil das Menü prinzipiell als Ereignismenge behandelt wird, aber als Liste implementiert wurde. Es ergeben sich zwei Interpretationen der Menü-Implementierung:

- Menü als Menge - Da es auf die Ereignisreihenfolge nicht ankommt, sind alle drei Varianten gleichwertig und die Entscheidung muß zugunsten der Lesbarkeit der Spezifikation oder der Effizienz der Ausführung getroffen werden. Durch das Fehlen eines Mengentyps in TYPOL ist es nicht möglich, die Irrelevanz der Ereignisreihenfolge zu spezifizieren! Auch wenn erst ein eigener Mengentyp in TYPOL spezifiziert wird, kann dem Leser der Spezifikation nur über Kommentare mitgeteilt werden, daß die sich durch die Mengenoperatoren ergebende Reihenfolge keine Bedeutung hat.

- Menü als Liste - Es wird dem Umstand Rechnung getragen, daß der Benutzer die Reihenfolge der Ereignisse im Ergebnismenü als Hilfestellung für die Ereignisauswahl benutzen kann. Für die vollständige Erhaltung der Reihenfolge kommt nur die erste ineffizienteste Version in Frage. Diese Spezifikation ist in diesem Fall auch wörtlich zu lesen, d.h. es ist eine spezielle Vorgenhensweise des Zusammenhängens von zwei Listen angegeben, die nicht durch eine andere ersetzt werden darf.

Eine Spezifikation sollte so wenig wie möglich und so viel wie nötig spezifizieren. Eine ausführbare Spezifikation hat jedoch nicht nur beschreibenden Charakter, sondern ist auch direkt verantwortlich für die Effizienz der Ausführung. Beim Menü als Menge steht der Spezifikator vor Wahl, eine sehr lesbare oder eine sehr schnell ausführbare Version zu erstellen. Auch wenn die Datenstruktur Menge zur Verfügung stünde, könnte sich der Spezifikator für die Liste entscheiden, wenn mit ihr schnellere Algorithmen möglich sind. In TYPOL *kann* er gar nicht so spezifizieren, daß ein möglicher Implementierer ohne Zuhilfenahme von Kommentaren die Reihenfolge, die für ihn am günstigsten ist, wählen kann[31]. Für das Menü als Liste entstehen keine Probleme, da hier das gewünschte Verhalten angegeben werden muß.

[31] Es soll hier nicht behauptet werden, daß es prinzipiell unmöglich ist, in TYPOL eine Menge mit nicht spezifizierter Reihenfolge der Mengenelemente zu spezifizieren. Eine mehrdeutige Regelsammlung würde genügen, *explizit* die Irrelevanz der Reihenfolge auszudrücken. Unter dem Ausführungsaspekt macht das aber keinen Sinn. Es ist also genauer gesagt nicht möglich, durch eine eindeutige Formulierung die Reihenfolge der Implentierung von TYPOL zu überlassen.

7 Bewertung

Es folgt eine bewertende Betrachtung

- der CENTAUR-Formalismen,
- der Unterstützung von CENTAUR bei der Sprachdefinition und
- über die Qualität des entstandenen BASIC-LOTOS-Simulators.

Hierbei werden nur die Kritikpunkte erörtert, die sich bei der eigenen Arbeit mit CENTAUR ergaben. Einige andere Kommentare zur Sprachentwicklung mit CENTAUR, insbesondere über das ASF+SDF System, sind in [JJWW91] nachzulesen.

7.1 METAL/SDF

Positiv:

Der Sprachentwickler, der frei zwischen METAL und SDF entscheiden kann, hat den Vorteil, zwischen der kurzen modularen Formulierung mit SDF und der flexibleren mit METAL wählen zu können.

METAL erlaubt die unabhängige Konstruktion von abstrakter und konkreter Syntax. Somit kann der Sprachentwickler maßgeblich auf die Gestalt der abstrakten Syntaxbäume Einfluß nehmen, die ja bei der Verwendung von TYPOL Grundlage der Semantikdefinition ist.

Durch das vordefinierte Terminal `TREE` ist es möglich, verschiedene Sprachen miteinander zu kombinieren. Im Simulatorentwurf wurden so die Verweise des Zustands auf die Spezifikation realisiert.

Negativ:

Ist er auf SDF angewiesen, weil er ASF als Semantikformalismus benutzen möchte, hat er die Kontrolle über die abstrakte Syntax abgegeben[32]. Es ist anzunehmen, daß dieser Nachteil mehr und mehr an Bedeutung verlieren wird, je weniger die abstrakte Syntax in der Sprachentwicklung eine Rolle spielen wird. Im ASF+SDF System wird komplett mit der konkreten Syntax gearbeitet, es sei denn, es soll zusätzlich ein Pretty Printer spezifiziert werden[33]. Mit SDF war es nicht möglich, die Assoziativität des parallel-Operators zu spezifizieren. Es handelt sich im allgemeinen Fall um einen dreistelligen Operator, für den die knappen SDF Formulierungen (noch) nicht ausgelegt sind. Sie erlauben die Angabe von Operatorassoziativitäten nur für zweistellige Operatoren.

Der Spezifikationsformalismus PPML für den Pretty Printer benutzt als Ausgangspunkt für das Layout die abstrakte Syntax. Soll daher für eine SDF Syntax ein Pretty Printer spezifiziert werden, muß die SDF Spezifikation mit abstrakten Knotennamen erweitert werden. Der gleiche Sachverhalt gilt für die Erstellung einer TYPOL Semantik für eine SDF Syntax. Auch hier wird die Zuordnung zwischen Sprachkonstrukt und seiner Semantik über die Namen der abstrakten Syntaxknoten vorgenommen. Dieser Zustand ist zwar tragbar, aber verbesserungsbedürftig.

Im Gegensatz zu SDF ist mit METAL kein modularer Sprachentwurf möglich. Alle Sprachkonstrukte einer Sprache müssen in einer Spezifikation enthalten sein. Es ist zwar möglich, Sprachen zu mischen, wie beim Simulatorentwurf geschehen, jedoch nur über einen Mechanismus der abstrakten Syntax, bei dem der Typ der verwendeten Sprache verloren geht. Ein Operand eines `assign` ist dann nur ein abstrakter Syntaxbaum, ob letztlich eine Expression oder ein Statement ist allein die Verantwortung des Sprachentwicklers, der die Regeln für die Konstruktion des `assign` angibt. Der Mechanismus erlaubt also das Mischen von abstrakten Syntaxen, ist aber nicht zur modularen Sprachentwicklung geeignet.

[32] siehe Kapitel CENTAUR

[33] Im Normalfall bleibt im ASF+SDF System die Benutzerformatierung erhalten

Da METAL- und PPML-Spezifikationen nicht nur einen Parser und eine textuelle Ausgabe generieren, sondern auch einen interaktiven Editor, bestimmen beide gleichermaßen das Selektionsverhalten des Editors. Es hängt bspw. von der PPML Spezifikation ab, welcher Ausdruck beim *clicken* mit der Maus auf eine öffnende Klammer selektiert wird. Durch die Verwendung von Kontexten in PPML-Spezifikationen läßt sich erreichen, daß der Ausdruck zwischen dem angewählten Klammerpaar selektiert wird. Ohne zusätzliche Maßnahmen wird der Ausdruck selektiert, der den geklammerten Ausdruck als Operand enthält. Es ist erstens eine Einschränkung, bei der Formulierung der Grammatik auf das Selektionsverhalten des Editors Rücksicht nehmen zu müssen, und zweitens umständlich, sich beim darauf folgenden Spezifizieren des Pretty Printer mit der gleichen Problematik nochmals beschäftigen zu müssen.

Verbesserungsvorschläge:
Ein Ausweg hieraus böte ein spezieller Formalismus für das Selektionsverhalten des Editors. Syntaxdefinition und Pretty Printer Spezifikation könnten völlig unabhängig von dieser Thematik erfolgen. Erst in einem dritten Schritt müßten von einem Default-Verhalten abweichende Selektionsregeln angegeben werden.

Das Problem der zusätzlich erforderlichen abstraken Syntaxknotennamen bei der gleichzeitigen Verwendung von SDF und PPML bzw. TYPOL könnte durch eine andere Aufteilung der Sprachdefinition gelöst werden. Statt eine Sprache in Syntax, Semantik und Layout zu gliedern, könnte die Strukturierung anhand der Sprache selbst erfolgen. Teilsprachen innerhalb der Sprache würden einzelne Module bilden und die kleinste Einheit einer Sprachdefinition wäre die *vollständige* Beschreibung eines Sprachkonstruktes. An einer Stelle lokalisiert, würden sowohl Syntax, Semantik als auch das Layout eines Sprachkonstruktes spezifiziert. Damit wäre keine künstliche Verbindung zwischen den Spezifikationsteilen mehr nötig. Für einen Überblick bspw. der Semantik einer Sprache könnten die entsprechenden Anteile, immer noch extrahiert und zusammen angezeigt werden.

7.2 PPML

Positiv:

Eine sehr nützliche Eigenschaft von CENTAUR ist die Möglichkeit, mehrere Pretty Printer unter verschiedenen Namen für eine Sprache definieren zu können. So kann nicht nur jeder Benutzer einer Sprache sein persönliches Lieblingslayout festlegen, sondern auch zu einer gegebenen Situation kann bspw. die Ausführlichkeit einer Sprachdarstellung geändert werden. Für den BASIC-LOTOS-Simulator existieren zwei Pretty Printer:

- `small` erzeugt eine Übersicht der aktuellen Prozesse.
- `std` zeigt die vollständige Zustandsrepräsentation an.

Im Normalfall enthält die Darstellung von `std` zu viel Information. Interessiert jedoch ein bestimmter Zustandsübergang, so kann für diesen Zweck kurzzeitig auf den `std` Pretty Printer gewechselt werden.

Die Pretty Printer sind aber nicht nur durch den Benutzer austauschbar, auch CENTAUR wechselt ggf. den Pretty Printer automatisch im Fall von gemischten Sprachen. In der Zustandsrepräsentation des BASIC-LOTOS-Simulators gibt es einen Operator, dessen erster Operand ein abstrakter Syntaxbaum aus der Spezifikation ist. Wird dieser Operator in einem Fenster ausgegeben, wechselt CENTAUR automatisch vom Pretty Printer des State-Formalismus auf den des SLOTOS-Formalismus. In der Zustandsdarstellung sind also die Verweise auf die Spezifikation im Klartext sichtbar. Weil auch Daten in CENTAUR als Sprachen repräsentiert werden, erhält man, durch Angabe einer PPML-Spezifikation, eine beliebige Präsentation seiner Daten. Es ist ein großer Vorteil, bei der Entwicklung eines Werkzeuges die zugrundeliegenden Daten in einem selbst gewählten Format präsentiert zu bekommen, statt sie einer internen Darstellung entnehmen zu müssen.

Auch ein Pretty Printer allein ermöglicht die Wahl des Detaillierungsgrads der Anzeige. Durch einen Printlevel läßt sich die Tiefe des angezeigten abstrakten Syntaxbaums beschneiden. In der PPML Spezifikation

- sind für verschiedene Printlevel unterschiedliche Regeln angebbar
- ist die Zunahme des Printlevels bei der Ausgabe von Operanden spezifizierbar.

Damit ist bspw. ein Procedure-Folding sofort realisierbar.

Ein weiteres mächtiges Konzept sind die Kontexte. Für die Ausgabe eines Operanden kann ein Kontext spezifiziert werden. In unterschiedlichen Kontexten können gleichen abstrakten Syntaxknoten verschiedene Ausgaben zugeordnet werden. Beispielsweise könnte eine Bezeichnerliste in einer Typvereinbarung in einer anderen Schriftart als in einer Variablenvereinbarung angezeigt werden.

Negativ:

Wie schon oben erwähnt, sollten die Layout- und Selektionsspezifikationen eine Trenunng erfahren.

Da ein abstrakter Syntaxbaum keine Prioritätsklammern enthält, müssen diese bei der Ausgabe wieder hinzugefügt werden. Dazu müssen in PPML explizit die Prioritäten von Operator und Operanden verglichen werden.

Verbesserungsvorschläge:

Eine PPML-Spezifikation könnte stark vereinfacht werden, würden diese Vergleiche automatisch von der Ausführungskomponente der Spezifikation durchgeführt werden. In der Spezifikation müßte dann nur in geeigneter Weise angegeben werden, wie im jeweiligen Fall zu verfahren ist. Könnte die PPML-Ausführung auf die Prioritätsvereinbarungen der Syntaxdefinition zugreifen, würde sogar die zweifache Angabe der Operatorprioritäten entfallen. Es ist auch denkbar aus der METAL-Syntaxdefinition einen Default Pretty Printer zu generieren. Abstrakte Syntaxknoten werden einfach in die Zeichenketten zurückübersetzt, durch die sie erzeugt wurden. Der Default Pretty Printer wäre Ausgangspunkt von Änderungen durch den Benutzer, die zum endgültigen Pretty Printer führen.

7.3 TYPOL

Positiv:

Der grundlegende Vorteil einer logischen Spezifikationssprache wie TYPOL ist, daß mehr das *Was* und weniger das *Wie* einer Vorgehensweise beschrieben wird. Der Spezifikator kann sich darauf konzentrieren, welche Zusammenhänge in seiner Spezifikation existieren und muß sich weder über die Ausführung seiner Regeln noch über Speicherverwaltung Gedanken machen. Es sind so zwar noch Endlosrekursionen formulierbar, aber andere Abstürze, die sich in Implementierungssprachen auf vielfältige Art und Weise herbeiführen lassen, sind, bei fehlerfreier Implementierung der Ausführungskomponente ausgeschlossen.

Durch die allgemeinen Datentypen Baum und Liste sind TYPOL-Regeln in einem gewissen Sinn sogar polymorph, d.h. es kommt nicht darauf an, wie eine Variable von einem Typ T letzendlich strukturiert ist. Dadurch läßt sich bspw. ein Datentyp durch lokale Änderungen erweitern, ohne daß die ganze Spezifikation überarbeitet werden muß. Dagegen ist es nicht möglich, ein und dieselbe Regelsammlung gleichzeitig für verschieden Typen zu verwenden.

Das TYPOL Konstrukt `recovery` erlaubt die Separation der Spezifikationen des Normal- und des Fehlerfalls. Entsprechend einer Ausnahmebehandlung können mit `recovery` Regeln angegeben werden, die dann benutzt werden, wenn die Regeln für den Normalfall fehlgeschlagen haben.

Obwohl eigentlich zum abstrakten Syntaxbaum-Manager gehörig, sollen hier die Zusatzinformationen erwähnt werden. Durch sie ist es möglich, daß ein TYPOL-Werkzeug, z.B. ein Typchecker, einem anderen TYPOL-Werkzeug, z.B. einem Compiler, Informationen im abstrakten Syntaxbaum hinterläßt. Diese Anreicherung des abstrakten Syntaxbaums ermöglicht die Kommunikation zwischen Werkzeugen, ohne daß ein spezielles Zwischenformat ausgetauscht werden muß. Alle Werkzeuge können so auf dem originalen abstrakten Syntaxbaum arbeiten und müssen nicht auf Informationen im Zwischenformat zurückgreifen, an welcher Stelle im originalen abstrakten Syntaxbaum ein bestimmtes Konstrukt stand. Kann ein Compiler eine typkorrekte Variablenvereinbarung aus Dimensionsgründen nicht übersetzten, kann er die betreffende Stelle gleich selektieren, ohne den beschriebenen Umweg gehen zu müssen. Dadurch lassen auch sich die Ausführungen von allen Werkzeugen im gleichen Fenster nachvollziehen, was noch im folgenden verständlich wird.

Dem TYPOL-Debugger liegt das 4 Port-Modell von PROLOG zugrunde. Jede Regel, die im aktuellen Beweis Verwendung findet, wird angezeigt, kann zum Fehlschlagen gebracht werden, kann kurz oder ausführlich bewiesen werden. Existiert zum augenblicklichen Subjekt einer Regel ein Editor, der die Sprache anzeigt, die das zu beweisende Subjekt enthält, positioniert der Editor den Quelltext an die entsprechende Stelle und selektiert das Subjekt. So läßt sich der Beweis eines TYPOL-Ziels anhand des Quelltextes sehr übersichtlich verfolgen.

Backtracking und Unifikation sind zwei mächtige Konzepte in TYPOL. Sie können u.U. die Angabe eines Algorithmus überflüssig machen. In einem Typchecker für LOTOS muß der Typ eines Ausdrucks in der Umgebung ermittelt werden, die durch ihn selbst mitdefiniert wird. In ASF muß ein Algorithmus formuliert werden, der nacheinander den Typ von Ausdrücken ermittelt, deren Umgebung völlig definiert ist und andere Ausdrücke solange zurückstellt, bis sie dieser Bedingung genügen. Unter Verwendung von CIRCULAR-TYPOL, eine besondere Art der Spezifikation, genügt die Angabe von wenigen Regeln, die direkt die Rückbezüglichkeit der Typbestimmung reflektieren.

Negativ:
Nicht immer wird diese Mächtigkeit benötigt. In vielen Fällen ist das Backtracking überflüssig und die Unifikation könnte durch das simplere Pattern Matching abgelöst werden. Dadurch könnte die Ausführungseffizienz von TYPOL-Spezifikationen verbessert werden.

Während die Operationen auf den elementaren Einheiten einer Sprache, wie Bäumen und Listen, nicht mehr spezifiziert werden müssen, trifft das bspw. nicht auf die Gültigkeitsbereiche einer Sprache zu. In diesem Zusammenhang müssen immer noch Algorithmen, z.B. zum Aufbau und Durchsuchen des Environments, angegeben werden. Das Ziel einer kompakten Sprachdefinition wäre es, nur über die Sprachkonstrukte zu reden und für die restlichen Zusammenhänge Formalismen oder Bibliotheken der Sprachentwicklungsumgebung zu benutzen.

Die wenigen Sonderzeichen, die für die Zusammensetzung von relationalen Operatoren in TYPOL erlaubt sind, ermöglichen keine selbsterklärende Formulierung der Regeln. Im allgemeinen muß eine Regelform einer Regelsammlung immer mit Kommentaren begleitet werden, um sie einem Leser verständlich zu machen. Dieser Mangel wurde in ASF behoben, indem nicht nur die konkrete Syntax der Sprachkonstrukte verwendet wird, sondern auch, durch SDF-Spezifikationen, kontextfreie Formulierungen der Regeln möglich ist. Schlüsselwörter und Variablen können beliebig zu einer selbsterklärenden Regel zusammengestellt werden.

Leider sind die Operatoren einer Sprache nicht nachträglich für semantische Zwecke erweiterbar. Es existiert zwar ein `define` Konstrukt, damit lassen sich aber nur *neue* kleine Teilsprachen vereinbaren. Das spezielle Ereignis `no_event`, das in der METAL-Spezifikation des Menüs vorkommt, wird nur für semantische Zwecke benötigt und sollte daher nicht in der Sprachdefinition von Menu auftauchen. Die Datenstruktur Menu muß dieses Ereignis in keinem Fall aufnehmen können, es muß allerdings vom Typ EVENT sein, damit es in einer bestimmten Regelsammlung verwendet werden kann. Wäre es möglich, `no_event` nachträglich als vom Typ EVENT zu vereinbaren, könnte entsprechend sauberer spezifiziert werden.

Es ist möglich, PROLOG-Prädikate polymorph zu verwenden, aber auf eine Art und Weise, die keine Typüberprüfung mehr zuläßt. Entweder ein PROLOG-Prädikat wird typisiert importiert und ist dann nur noch in dieser Ausprägung verwendbar oder anstelle der Typen wird die anonyme Variable[34] angegeben. Allerdings ist dann jeder Typ erlaubt und eine fehlerhafte Verwendung kann nicht mehr aufgezeigt werden.

Da TYPOL-Regeln typisiert sind, wäre es prinzipiell möglich, für Operatornamen auch Variablen zuzulassen. Die Regelform für die Regelsammlung MERGE könnte dann so formuliert werden:

Menu1, Menu2 $\vdash$ OpName(State1, State2) => Menu;

Unter der Voraussetzung, daß die Operanden des Operators `OpName` die Menüs `Menu1` und `Menu2` liefern, besitzt der Operator `OpName` das Menu `Menu`. An der Stelle von `OpName` muß jedoch immer ein konkreter Operatorname stehen. Das in der Dokumentation beschriebene `where` Konstrukt könnte hier Abhilfe schaffen, ist aber leider nicht implementiert. Um trotzdem eine MERGE Regelsammlung für mehrere Operatoren zu verwenden, wurde die Interpretation geändert nach:

Operator $\vdash$ Menu1, Menu2 => Menu;

Die Menüs sind jetzt das Subjekt und der Operator, mit seinen Operanden, wird zur Voraussetzung.

Unbequem und zudem formal logisch nicht einsichtig ist die Versorgung eines TYPOL-Ziels mit einem Subjekt. Im Abschnitt Zusatzinformationszugriff des Kapitels Spezifikation wurde diese Problematik ausführlich erläutert. Es würde einerseits das Spezifizieren erleichtern und andererseits die Lesbarkeit der Spezifikation erhöhen, wenn auf diese Details keine Rücksicht genommen werden müßte.

[34] In TYPOL wird die anonyme Variable durch _ gekennzeichnet

Ambivalent:
TYPOL-Spezifikation sind nicht leicht zu lesen. Abgesehen von der aufgezwungenen kryptischen Syntax der relationalen Operatoren muß der Leser einzelne Variablen einander zuordnen, um einen Informationsfluß zu identifizieren. Andererseits ist so eine statische Formulierung von Gesetzmäßigkeiten möglich, die in einer imperativen Beschreibung in dieser Form nicht sichtbar werden würden.

Im Abschnitt Syntax des Kapitels Spezifikation wurde bereits auf die Verwechslungsmöglichkeiten zwischen SET-Namen und Namen von abstrakten Syntaxknoten hingewiesen. In einer gewissen Hinsicht mag diese Uniformität erwünscht sein, es wäre jedoch besser, schon von der Syntax eine Unterscheidung zu ermöglichen.

Spezifikation der Art von JOIN im Modul `merge` sind zwar knapp und präzise, aber, bei heutiger Rechnerleistung, direkt dafür verantwortlich, daß die generierten Werkzeuge weit von einer Produktqualität entfernt sind. Es bleibt weiterhin ein Betätigungsfeld der Forschung, wie allgemeine Spezifikationen effizient ausgeführt werden können bzw. wie man in effizient ausführbaren Spezifikationssprachen allgemeine Formulierungen erlaubt.

Verbesserungsvorschläge:
Hinsichtlich der Ausführungseffizienz wäre ein Werkzeug optimal, das Eigenschaften von TYPOL-Regelsammlungen feststellt und einem Ausführungsmechanismus zuordnet, der gerade noch die geforderte Funktionalität besitzt. TYPOL-Spezifikation, die weder Backtracking noch Unfikation benutzen, könnten schneller funktional ausgeführt werden. Sehr interessant wäre auch die inkrementelle Ausführung von TYPOL-Spezifikationen gerade hinsichtlich Simulatoren, deren große Zustände sich bei Zustandsübergängen nur wenig verändern.

Bei den negativen Kritikpunkten wurde schon die Nützlichkeit von Variablen für Operatornamen angesprochen. Auch für TYPOL-Ausdrücke im allgemeinen könnte man Aliase vergeben. Oft werden Terme mehrmals verwendet und müssen an einer Stelle expandiert geschrieben werden, um eine Komponente benennen zu können. Das Vergeben von Aliasen würde die Wiederholung der expandierten Terme vermeiden und gleichzeitig die Identifizierung von gleichen Termen erleichtern.

Statt

$$\frac{\begin{array}{l}\text{ANNOT\_SPEC}(\vdash \text{spec\_def(SpecHeader, Bexp, OptPdefs)}) \ \& \\ \text{EXPAND(Bexp => State)}\end{array}}{\vdash \text{spec\_def(SpecHeader, Bexp, OptPdefs) => State;}}$$

könnte

$$\frac{\begin{array}{l}\text{ANNOT\_SPEC}(\vdash \text{Spec}) \ \& \\ \text{EXPAND(Bexp => State)}\end{array}}{\vdash \text{Spec} = \text{(spec\_def(SpecHeader, Bexp, OptPdefs)) => State;}}$$

formuliert werden.

Wird eine TYPOL-Variable in einer Regel nur einmal verwendet, d.h. steht sie nur als Platzhalter für einen Term und wird nicht zur Informationsübermittlung verwendet, ist sie in jedem Fall durch die anonyme Variable ersetzbar. Dadurch wird die Regel etwas unleserlicher, weil durch geschickte Wahl von Variablennamen die Bedeutung der Regel illustriert werden kann. Allerdings könnten durch konsequente Nutzung der anonymen Variablen alle einzeln auftretenden Variablen durch den TYPOL-Typchecker gemeldet werden. Alle diese Meldungen wären dann Hinweise auf falsch geschriebene Variablennamen oder unvollständige Regeln. Unter Umständen kann dies eine erhebliche Erleichterung bei der Fehlersuche darstellen.

Das Nachvollziehen eines Beweises wird erleichtert durch die Selektion der Subjekte in einem Editor und der Möglichkeit, die Regeln zu benennen. Trotzdem könnte die Übersichtlichkeit noch erhöht werden. Beim Beweis der Regel

$$\frac{\begin{array}{l}\vdash \mathrm{A}\ \&\\ \vdash \mathrm{B}\end{array}}{\vdash \mathrm{op}(\mathrm{A},\mathrm{B});}$$

zeigt der TYPOL-Debugger zunächst die Regel in dieser Form an. Wird darauf der Beweis der Regel angestoßen, zeigt der Debugger die erste passende Regel zur Erfüllung von `A` an. Ist der Beweis von `A` beendet, *springt* der Debugger zur ersten passenden Regel zur Erfüllung von `B`. Durch dieses Verhalten verliert man leicht die Übersicht, aus welchem Grund eine Regel bewiesen werden soll. Dauert der Beweis von `A` sehr lange, muß man eine sehr gute Kenntnis der Spezifikation besitzen, um sich daran zu erinnern, daß nach `A`, zur Erfüllung von `op(A, B)`, `B` bewiesen werden muß. Nimmt man an, der Ausführungsmechanismus müßte nach dem Beweis von `A` im Beweisbaum bis `op(A, B)` zurückwandern, um sich das neue Ziel `B` zu holen, wäre es sinnvoll, jeden Regel-Knoten im Beweisbaum, den der Ausführungsmechanismus mit seinem Beweispfad besucht, anzuzeigen. Dadurch würde der Debugger nicht von `A` nach `B` springen, sondern würde erst zur angegebenen Ursprungsregel zurückkehren, um den Beweis von `B` zu motivieren.

Ein Beitrag zur Modularisierung der Sprachentwicklung und damit eine Entlastung der Spezifikation von statischer und dynamischer Semantik wäre die Einführung eines Formalismus für Gültigkeitsbereiche. Die Spezifikation der Gültigkeitsbereiche, auch unter dem Namen Bezeichneridentifikation bekannt, wäre von den Semantik-Spezifikationen separiert und könnte die automatische Generierung eines Bezeichner-Environments bewirken. Dieses könnte, mit bereitgestellten Hilfsfunktionen, in den Spezifikationen der Semantik benutzt werden. Dieses Vorgehen erlaubt eine effiziente Implementierung der Bezeichneridentifikation und befreit die Semantik-Spezifikationen von Algorithmen, die nicht direkt durch die Sprachkonstrukte induziert werden.

7.4 Benutzerschnittstelle

Positiv:

Die mit CENTAUR erstellbaren Benutzerschnittstellen bieten eine Vielfalt von Möglichkeiten der Benutzerinteraktion. Grundlage der Kommunikation mit dem Benutzer sind immer Fenster, von denen mehrere als zusammengehörige Einheit zusammenwirken können. Fenster können mit Buttons, pop-up Menüs und drop-down Menüs Menüs versehen werden. Auch das Verhalten auf Selektion von Fensterinhalten mit der Maus kann angegeben werden. Da alle mit CENTAUR generierten Umgebungen die gleichen Resourcen nutzen, sollten sie einem einmal eingeübten Benutzer leichte Einarbeitung gewährleisten.

Negativ:

Die Verwendung der genannten Resourcen ist nicht grundsätzlich kompliziert, allerdings sind Kenntnisse der Programmiersprache LISP unerläßlich. Auch einige Interna des CENTAUR-Systems offenbaren sich bei der Modellierung der Benutzerschnittstelle. Tatsächlich wird die Benutzerschnittstelle nicht *generiert*, sondern muß *programmiert* werden. Es fehlt ein Formalismus, der das jetzt noch notwendige Einlöten der Kommunikationselemente durch Abstraktion auf eine logische Ebene hebt. Genau ein solcher ist unter dem Namen SEAL bereits in der Entwicklung.

7.5 Handhabung

Grundsätzlich entspricht das Arbeiten mit CENTAUR dem Umgang mit den inzwischen etablierten grafischen Benutzeroberflächen. Es beinhaltet Selektionen von Teilbäumen und die Auswahl von Funktionen mit der Maus. Über ein *Clipboard* können Teilbäume zwischen Fenstern ausgetauscht und ersetzt werden.

Positiv

Der Umgang mit CENTAUR ist sehr bequem, weil häufig nur Mausbedienung erforderlich ist. Änderungen an Spezifikationen fließen sofort in das generierte Werkzeug ein. Ein aufwendiges neucompilieren des Werkzeuges entfällt. Insgesamt ist die Interaktivität der Sprachdefinition mit CENTAUR als sehr gut zu beschreiben.

Negativ:

Gegenüber dem Benutzer einer mit CENTAUR generierten sprachspezifischen Umgebung kommt der Sprachentwickler jedoch nicht umhin, die LISP-Console zu benutzen. Das französische LeLisp ist sozusagen der Kern von CENTAUR. Es besitzt unter anderem objektorientierte Sprachkonstrukte, durch die die CENTAUR Ressourcen gut strukturiert werden. Sollen sich aber Sprachentwickler nicht nur aus Programmierern rekrutieren, muß eine abstraktere Form der Kommunikation gefunden werden.

Ein CENTAUR-Editor dient mehr als Behälter eines abstrakten Syntaxbaums und als Träger der Kommunikationselemente denn als Werkzeug zur Erstellung und Modifikation von Quelltexten. Die Möglichkeiten in dieser Hinsicht sind derart eingeschränkt, daß bezeichnender Weise seit der CENTAUR-Version 1.2 eine Kommunikation zum Standard Editor `emacs` besteht. Auch der aktuelle CENTAUR-Editor bricht beim ersten Syntaxfehler eines einzuladenden Quelltexts das Parsen ab und gibt lediglich die entsprechende Zeilennummer auf der Console aus. Seitdem in der CENTAUR-Version 1.2 externe Parser verwendet werden, benötigt der Editor eine sehr lange Initialisierungszeit beim ersten Einladen eines Quelltextes. Alle weiteren Quelltextwechsel erfolgen danach aber unverzögert.

Im Verlauf einer Sprachentwicklung werden von CENTAUR zahlreiche Dateien generiert. Es ist der Übersicht des Sprachentwicklers wenig förderlich, daß keine Möglichkeit besteht, die von CENTAUR generierten Dateien von den originalen Spezifikationen des Entwicklers getrennt zu speichern.

Ein so großes[35] System wie CENTAUR stellt erhebliche Ansprüche an die Dokumentation, um es auch optimal einsetzen zu können. Trotz, oder gerade wegen, eines ca. 8cm dicken Ordners fällt die tiefergehende Einarbeitung in CENTAUR nicht unbedingt leicht. Immerhin liegt die Dokumentation auch als ASCII Dateien vor, so daß eine grobe Stichwortsuche ermöglicht ist. Dem Umfang und den Möglichkeiten von CENTAUR angemessen wäre aber ein rechnergestütztes Retrievalsystem mit Stichwortsuche und Querverweisen. In einem solchem interaktiven Nachschlagewerk ließe sich eine den komplexen Inhalten angemessene Strukturierung unterbringen.

Ein im ASF+SDF System integrierter Projektmananger fehlt für die Entwicklung von TYPOL-Spezifikationen vollständig. Der Sprachentwickler muß selbst die Übersicht über das Compilieren seiner Versionen behalten und besitzt als einzige Arbeitserleichterung einen Batchbetrieb über die LISP-Console.

Als letzter, weil gravierendster Punkt, sei hier der Garbage Collector des LeLisp erwähnt. Da fast allen Aktionen in CENTAUR das LeLisp zugrunde liegt und dieses von Zeit zu Zeit seinen Speicher aufräumen muß, stockt die Interaktion mit CENTAUR zu diesen Zeitpunkten von mehreren Sekunden bis zu einer Minute[36]. Für ein interaktives System ist diese schubweise völlige Blockade untragbar. Selbstverständlich läßt sich mit CENTAUR trotzdem gut arbeiten, aber ein auf Eis gelegter Benutzer sollte die unfreiwilligen Kunstpausen zu anderen Zwecken schätzen lernen, um Frustrationen vorzubeugen.

[35] Plattenspeicherbedarf $\approx$ 52 Mbyte

[36] Die Zeitdauer der Speicherreorganisation ist stark vom Hauptspeicherausbau des Rechners und der aktuellen Auslastung abhängig

7.6 Simulator

Positiv:

Der generierte Simulator läßt sich als ansprechender Prototyp beschreiben. Für die zu Testzwecken eingesetzten kleinen und mittelkleinen Basic-Lotos-Spezifikationen ist er ohne Einschränkungen einsetzbar. Die Benutzung ist einfach und die Ausführung der Funktionen erfolgt meist unmittelbar oder bleibt in erträglichen Zeiträumen. Mit relativ geringem Aufwand konnte eine drei Fenster umfassende Simulationsumgebung generiert werden.

Negativ:

Wie bereits weiter oben benannt, ist der Basic-Lotos-Editor nicht unbedingt als solcher zu gebrauchen. Fairerweise soll jedoch angemerkt werden, daß die Möglichkeit durch einen speziellen Formalismus Sprachkonstrukte aus Menüs auszuwählen nicht untersucht wurde.

Selbst für die kleineren Basic-Lotos-Spezifikationen wird erheblicher Speicherplatzbedarf angemeldet. Auch die Ausführungsgeschwindigkeit ist noch verbesserungsbedürftig. Obwohl die Ausführung der Typol-Spezifikation und die Anzeige des Ergebnises, bei kleinen Anwendungsbeispielen, ansprechend schnell erfolgen, wird die Ausführung der Spezifikation durch PROLOG-Interpretation größeren Lotos-Spezifikation mit sehr großen Zuständen nicht gerecht. Die Kombination von PROLOG, LISP und Centaur-eigener Baumdarstellung erfordert Datenkonvertierungen, die signifikant die Ausführungsgeschwindigkeit beeinflußen.[37] Der übermäßige Speicherplatzbedarf und die Engpässe bezüglich der Ausführungsgeschwindigkeit stellen die Simulation von Lotos-Spezifikation mit einigermaßen realistischem Umfang sehr in Frage.

7.7 Gesamtbewertung

Die zum Teil zahlreiche negative Kritik an Centaur soll nicht den Eindruck erwecken Centaur besäße mehr schlechte als gute Seiten. Ganz im Gegenteil ermöglicht erst der solide Standard von Centaur den hohen Anspruch, der vielen Kritikpunkten zugrunde liegt. Die Qualität der vorhanden Kritikpunkte und die Abwesenheit vieler anderer Kritikpunkte ist demnach als Plus für Centaur zu werten. Centaur ist sehr stabil und ermöglicht hervorragend interaktive Sprachentwicklung mit einer grafischen Benutzeroberfläche. Mit geringen Aufwand sind einfache Werkzeuge schnell erstellt. Ein Tutorial zeigt schnell den Umgang mit den Centaur-Formalismen und verdeutlicht das gelungene Konzept, auch Daten von Spezifikationen als Sprachen zu formulieren. Durch höheren Einsatz können sehr komfortable Sprachumgebungen generiert werden. Die vielfältigen Möglichkeiten und die absolute Funktionsfähigkeit des Systems entschädigen dafür, daß Centaur zeitweise etwas langsam ist.

[37] Für eine bestimmte Situation beträgt der Anteil 85%! Die Centaur Entwickler führen dieses vom Autor ermittelte Ergebnis auf einen Fehler im Konvertierungsalgorithmus zurück, bis zum heutigen Datum ist allerdings der versprochene Patch nicht ausgeliefert worden.

8 CENTAUR - TSK

Im folgenden wird der TSK-Formalismus vorgestellt und anschließend in einem Vergleich CENTAUR gegenübergestellt. Da beide Ansätze recht unterschiedlich sind,

- CENTAUR - der LOTOS-Simulator wird vollständig als Werkzeug einer Programmierumgebung spezifiziert
- TSK - eine vorhandene Simulationsumgebung wird durch Konfiguration und die Abbildung der LOTOS-Semantik auf TSK zum LOTOS-Simulator,

ist ein direkter struktureller Vergleich nicht möglich. Es wird jedoch der Versuch unternommen Gemeinsamkeiten und die jeweiligen Vor- und Nachteile aufzuzeigen.

8.1 TSK

TSK[38] ist ein Kalkül für kommunikationsbasierte Sprachen mit Prozeßkonzepten, deren Semantik sich mit Zustandsübergängen der Form

$$(term, store) \xrightarrow{a} (term', store')$$

beschreiben läßt[39]. TSK orientiert sich stark an CCS, enthält aber signifikante Erweiterungen ohne die z.B. eine Abbildung LOTOS $\rightarrow$ TSK nur sehr umständlich möglich wäre. Dazu gehören der explizite Speicher *store* und bspw. der `refine`-Operator. Andere Erweiterungen erklären sich dadurch, daß TSK das Ziel eines Übersetzungsprozesses ist und die Darstellung von TSK-Ergebnissen in einer der Quellsprache angemessenen Weise erfolgen soll.

TSK entstand aus dem Gedanken eine universelle Simulationumgebung bereitzustellen, die durch Konfigurieren sofort einen prototypischen Simulator für eine neue Sprache liefert. Dieser Ansatz ist eine erhebliche Abkürzung gegenüber der vollständigen Spezifikation eines neuen Simulator bspw. mit CENTAUR. Der Grund dafür ist die spezielle Eignung der TSK-Simulationsumgebung nur für einen bestimmten Sprachtyp mit Kommunikationen und Prozeßkonzept. Da die Konzepte für die Kommunikation, die Prozesse und den Zustandsübergang bereits im TSK-Kalkül modelliert sind und auch die Benutzerschnittstelle schon entsprechende Kommunikationselemente bereitstellt, entfällt die Spezifikation dieser Anteile. Die Tätigkeiten des Sprachentwicklers beschränken sich also auf:

1. Transformation der Sprachsemantik nach TSK.
2. Ausfüllen vorgegebener Funktionen, die die Rückübersetzung der TSK-Ergebnisse in die Benutzersprache vornehmen.

Der prinzipielle Ablauf einer Sprachdefinition mit TSK läßt sich in Abbildung 10 nachvollziehen. Hier werden, am Beispiel einer LOTOS-Umgebung, zwei mögliche Sichten auf TSK deutlich: Der Benutzer einer TSK-Simulationsumgebung sieht nur seinen Quelltext und die Simulationsergebnisse. Der Sprachdefinierer sorgt für die Übersetzungen

LOTOS-Spezifikation $\rightarrow$ TSK-Spezifikation und
TSK-Simulation $\rightarrow$ LOTOS-Simulation.

Eine dritte Sicht, die des TSK-Systempflegers, wird in der Abbildung nicht explizit dargestellt. Sie ist aber essentiell für alle Benutzer und alle zukünftigen Sprachdefinierer, die von Erweiterungen und Verbesserungen profitieren.

[38] Thiesscher Simulations Kalkül
[39] Sprachdefinitionen mit TSK sind für LOTOS und Occam in Vorbereitung

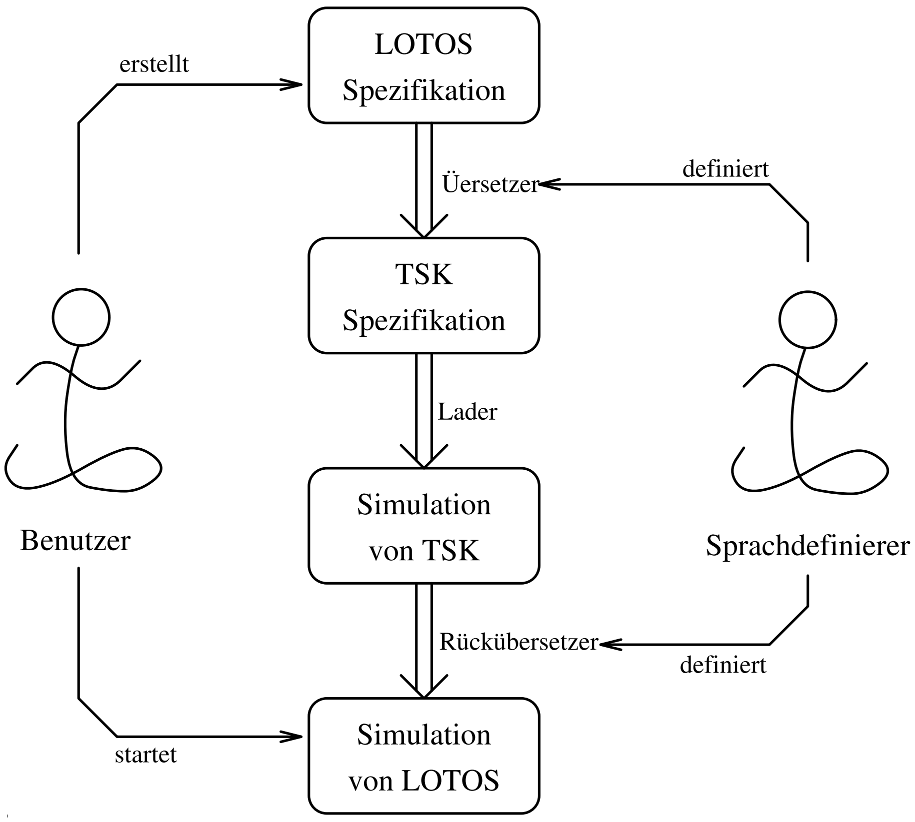

Abbildung 10: Sprachdefinition mit Tsk

Bemerkenswert ist, daß es sich bei der Semantikdefinition um eine sehr direkte Abbildung auf eine Sprache mit bereits definierter Semantik handelt. Im allgemeinen wird durch wenige Tsk-Operatoren direkt die Semantik des Quellsprachenoperators ausdrückbar sein, ohne weitere Hilfsfunktionen definieren zu müssen. Auch im Entwurf des Basic-Lotos-Simulators nahm die *expand*-Funktion eine Übersetzung vor. Danach schloß sich jedoch noch die Definition der Bedeutung der Zielsprache unter Verwendung zahlreicher Hilfsfunktionen an. Der Transformationsansatz hingegen ermöglicht sehr kompakte Semantikdefinitionen, erfordert aber eine dritte Sprache, mit der die Übersetzung

Quellsprachen-Operator → Tsk-Operator

spezifiziert wird. Dies bedeutet keinen Nachteil, wenn die Übersetzungssprache im wesentlichen nur aus einem Zuordnungsoperator besteht. Benötigen die Übersetzungen jedoch komplexere Operationen gebietet der Transformationsansatz diese nicht mit Tsk zu formulieren, sondern sie mithilfe der Übersetzungssprache bereitzustellen. Auf der einen Seite erhält man so eine klare Trennung zwischen Übersetzungsfunktionen und Konstruktion von Tsk-Termen. Auf der anderen Seite muß man neben den Tsk-Termen auch noch die Übersetzungssprache handhaben und beherrschen. In Tabelle 8 sind die Zielkonstrukte der Semantikdefinition, die Tsk-Operatoren, aufgelistet.

Zusätzlich zur Angabe der Semantikübersetzung muß in Tsk noch die Aktualisierung der Tsk-Parameter Speicher und Quellsprachenwerte vorgenommen werden. Hierbei wird definiert was der Speicher enthält und welche Wertetypen zulässig sind.

Für jeden Variablentyp müssen Funktionen definiert werden für

- die Wertzuweisung
- den Vergleich auf Gleichheit
- eine Wertegenerierung

Mit den vorgestellten Maßnahmen ist die Transformation der Quellsprache nach TSK komplett. Der Benutzer eines TSK-Simulators wünscht sich zusätzlich zu der Simulation noch eine, seinen Bedürfnissen entsprechende, Darstellung der Simulationsergebnisse und einen Bezug zu seinem Quellsprachentext. Deshalb hält TSK Funktionen zur Rückübersetzung der Ergebnisse bereit und besitzt Komponenten in seinen Operatoren, die einen Bezug zum Quellsprachentext ermöglichen. Die Rücktransformation der TSK Ergebnisse beinhaltet Funktionen für

- die Darstellung von Werten der Quellsprache
- das Auffinden von speziellen Stellen im originalen Quelltext, um sie ggf. anzeigen und hervorheben zu können.
- ...

Der TSK-Prototyp benutzt PSG und SAMPλE, die beide an der Technischen Hochschule Darmstadt entwickelt wurden, sowohl als Implementierungsumgebung und -sprache als auch für die Simulationsumgebung und die Semantikübersetzungssprache.

TSK als Kalkül und dessen Benutzung wird vollständig in [Thing] beschrieben, aus dem auch Abbildung 10 (modifiziert) und Tabelle 8 stammen.

leerer Term	NIL
einfache Komm.	δ pos term
Kommunikationspunkt	ag data pos term
Beschränkung	$\{ag_1, \ldots, ag_n\}\setminus$ term
Umbenennung	$[ag_1 \mapsto ag_1^{'}]$ term
Verfeinerung	$<ag \leftarrow ag_1, \ldots, ag_n >$ term
Alternativen	$term_1 + \ldots + term_n$
Nebenläufigkeit	$term_1 \mid \ldots \mid term_n$
Unabhängigkeit	$term_1 \parallel \ldots \parallel term_n$
Sequenzen	$term_1 ; \ldots ; term_n$
Unterbrechungen	$term_1$! $term_2$
Prozeßtyp	name form–parm: term
Prozeßinstantiierung	name act–parm
Konditional	p_1 $term_1$? ... ? p_n $term_n$
Unvollständigkeit	BLACK
imperative Anbindung	Λ trans term
Instanzen	Π show term

Tabelle 8: TSK-Operatoren

8.2 Aufwandsvergleich

Beide Verfahren, die Erstellung eines Simulators mit CENTAUR und die Konfiguration der TSK-Simulationsumgebung, beruhen auf dem Generator-Prinzip. Konsequenzen daraus sind, daß beide

- Spezifikationssprachen zur Semantikdefinition benutzen und die
- Simulatoren spezielle Eigenschaften der Generatoren erben.

Bevor diese Eigenschaften, und damit die jeweils erzielten Ergebnisse, verglichen werden soll zunächst der jeweilige Aufwand betrachtet werden.

Bereits weiter oben wurde die TSK-Methode als erhebliche Abkürzung gegenüber der der herkömmlichen Simulatorgenerierung bezeichnet. Wie im Kapitel Entwurf nachzuvollziehen muß gewöhnlich eine Zustandsrepräsentation entworfen und eine Darstellung für die Ereignisse gefunden werden. Zusätzlich muß sowohl auf dem Quelltext als auch auf der Zustandsrepräsentation eine Semantik definiert werden. Darüber hinaus beinhaltete die Erstellung einer Simulationsumgebung mit CENTAUR die Spezifikation von Pretty Printern für Zustandsdarstellung bzw. Ereignisauswahl und die Programmierung der Komponenteninteraktion mit LISP. Alle diese Arbeitsschritte entfallen bei der Benutzung von TSK, obwohl das Endergebnis, wie später noch zu beschreiben ist, durchaus hochwertiger ausfallen kann.

Der zentrale Akt bei der Benutzung von TSK ist die Übersetzung der Quellsprachenkonzepte in die der TSK-Operatoren. Exemplarisch wird hier die Mehrpunkt-Kommunikation von LOTOS herausgegriffen. In LOTOS ist es möglich, beliebig viele Ereignisse synchronisierend zu einem Ereignis zu vereinigen, während TSK nur die Sender $\leftrightarrow$ Empfänger Situation vorsieht. Der `refine`-Operator von TSK erlaubt aber eine elegante Abbildung der Mehrpunktkommunikation auf die Zweipunktkommunikation. Oft werden Kombinationen von TSK-Operatoren verwendet werden müssen, um einen Operator der Quellsprache zu modellieren. Obwohl diese Übersetzung sicherlich von Fall zu Fall einige Überlegung erforderlich macht, steht der Aufwand in keinem Verhältnis zur herkömmlichen Methode der vollständigen Semantikdefinition ohne die TSK-Abstraktionen. Zudem sind die Gedanken des Übersetzers stärker auf die Sprachsemantik konzentriert und nicht durch sekundäre Probleme abgelenkt. Er braucht sich nur auf die Semantik der TSK-Operatoren abzustützen, ohne sich überlegen zu müssen, wie letztendlich die möglichen und parallel möglichen Kommunikationen ausgerechnet werden.

Zur Vermeidung der Spezifikation von Gültigkeitsbereichen und Zugriff auf Definitionen macht TSK eine vernünftige, aber starke Voraussetzung: Das Vorhandensein eines Typcheckers der Quellsprache. Die Ergebnisse des Typcheckers entheben den Sprachdefinierer der dynamischen Semantik der Spezifikation von Environments und zugehörigen Zugriffsfunktionen. Die Situation auf diesem Gebiet entspricht also der herkömmlichen Problematik. Entweder wird ein Typchecker erstellt, dessen Ergebnisse verwendet werden, oder die statische Semantik wird mit der dynamischen Semantik vermischt. Das bedeutet, daß auch TSK-Übersetzungen nicht frei von Hilfsfunktionen sind, die statische Semantik realisieren bzw. zur Realisierung von mathematischen Konzepten dienen, die zur Übersetzung benötigt werden. Der Aufwand der Bereitstellung von Hilfsfunktionen wird aber gegenüber anderen Formalismen durch die Verwendung von SAMPλE als Übersetzungssprache stark gemindert. SAMPλE besitzt viele vordefinierte mächtige Funktionen und eine umfangreiche Bibliothek, so daß auch dieser Spezifikationsanteil sehr knapp ausfällt. Die Verfügbarkeit von Funktionalen und verzögerter Auswertung steigern die Abstraktionsmöglichkeiten des Sprachdefinierers, stellen jedoch gleichzeitig höhere Anforderungen zur adäquaten Nutzung der Übersetzungssprache an ihn. In jedem Fall ist für eine mächtige Sprache wie SAMPλE die Einarbeitungszeit länger und die Lesbarkeit hängt stark von der Disziplin des Sprachdefinierers ab. Damit ist ein neuralgischer Punkt des TSK-System in seiner jetzigen prototypischen Ausführung benannt. Die Eigenverantwortlichkeit des Sprachdefinierers ist sehr hoch. Für die TSK-Terme ist keine statische Semantik definiert, u.a. können Ereignisnamen ohne Vereinbarung benutzt werden. Die wahlfreie Übersetzung erlaubt zwar eine unkomplizierte Vorgehensweise, erfordert dafür aber einen höchst eigenverantwortlichen Umgang, um sinnvolle Übersetzungen zu erhalten. Zu den Fallstricken dieser Art gehört die Tatsache, daß nicht nur die TSK-Terme, sondern auch die Konzepte der Übersetzungssprache beherscht werden müssen.

Auf den ersten Blick scheint es aussagekräftig, die *logische* Spezifikation mit TYPOL der *funktionalen* Übersetzung mit SAMPλE gegenüberzustellen. Für die Induktion über den syntaktischen Aufbau einer Sprache sind aber beide Ansätze gleichwertig. Dazu ein Beispiel, das zum besseren Verständnis aus dem Bereich der mathematischen Ausdrücke entstammt. Ob in der Regelsammlung EVAL

$$\frac{\begin{array}{l}\mathrm{Vars} \vdash \mathrm{E}_1 \rightarrow \mathrm{Result}_1 \ \& \\ \mathrm{Vars} \vdash \mathrm{E}_2 \rightarrow \mathrm{Result}_2 \ \& \\ \mathrm{PLUS}(\mathrm{Result}_1, \mathrm{Result}_2, \mathrm{Result})\end{array}}{\mathrm{Vars} \vdash \mathrm{plus}(\mathrm{E}_1, \mathrm{E}_2) \rightarrow \mathrm{Result};}$$

formuliert wird oder die Funktion eval wie folgt definiert wird

$$\mathrm{eval}(\mathrm{Vars}, \mathrm{plus}(\mathrm{E}_1, \mathrm{E}_2)) = \mathrm{eval}(\mathrm{Vars}, \mathrm{E}_1) + \mathrm{eval}(\mathrm{Vars}, \mathrm{E}_2),$$

ist letztendlich, abgesehen von Ausführungsaspekten, eine Frage der Syntax. Auch viele der benötigten Hilfsfunktionen können äquivalent formuliert werden. Sollten für die Hilfsfunktionen der Übersetzung tatsächlich die generell größeren Abstraktionsfähigkeiten des funktionalen Ansatzes nötig sein, stellt sich die Frage, ob die Verwendung von TSK überhaupt die richtige Wahl war. Ist eine komplexe Übersetzung nicht vermeidbar, sollte eine Konstruktion im herkömmlichen Sinn in Erwägung gezogen werden.

Erwähnenswert im Vergleich der Arbeitsaufwände ist die Tatsache, daß in TSK nur einige Rückübersetzungsfunktionen definiert werden müssen, um eine komplette komfortable Simulationsumgebung zu erhalten. Im Gegensatz zur LISP-Programmierung mit CENTAUR wird die Umgebung bei der Benutzung von TSK tatsächlich generiert. Um die Bereitstellung von Ausgabefenstern, das Anbieten von Eingabelementen und die Definition des Mausverhaltens braucht sich der TSK Benutzer nicht zu kümmern. Nur die Anteile der Oberfläche, die das Erscheinungsbild der Quellsprache direkt betreffen, müssen durch Definition der Rückübersetzungsfunktionen spezifiziert werden.

Zusammenfassend läßt sich festhalten, daß die TSK-Methode eine wesentlich stärkere Konzentration auf die Quellsprachensemantik zuläßt und durch geringen Aufwand (Übersetzung der Semantik und Aktualisierung von Speicher und Werten) die meisten herkömmlichen Arbeitschritte überflüssig macht. Wie man an der Verwendung einer mächtigen Übersetzungssprache und der Voraussetzung eines Typcheckers sieht, ist die gewünschte Konzentration noch nicht vollständig ermöglicht. Abschließend soll nochmals betont werden, daß die genannten Vorteile hauptsächlich durch die Spezialisierung von TSK auf einen bestimmten Sprachtyp erzielt werden und der Vergleich nur in diesem Rahmen Gültigkeit besitzt.

8.3 Ergebnisvergleich

Da die Arbeiten an der TSK-Simulationsumgebung noch laufen und bis zu diesem Zeitpunkt noch keine fertige Umgebung zur Ansicht vorliegt, ist der folgende Vergleich grundsätzlich hypothetischer Natur. Im weiteren wird von einer geplanten Ausbaustufe ausgegangen. Das Hauptaugenmerk liegt auch nicht auf dem Vergleich von zwei generierten Simulatoren, sondern auf den generellen Eigenschaften, die von den Generatoren auf ihre Produkte vererbt werden.

Beide Ansätze liefern für verschiedene Produkte uniforme Benutzeroberflächen. Die standardisierte Benutzung ist bei der Entwicklung mit CENTAUR allerdings eingeschränkter, weil hier die Umgebung viel flexibler gestaltet werden kann. Der Vorteil, für ein spezielles Produkt eine individuell maßgeschneiderte Umgebung generieren zu können, erweist sich als Nachteil, wenn das Produkt nur eines unter vielen zu Verwendenden ist. Allerdings bleibt durch die Umgebungsfunktionalität von CENTAUR ein allgemeinerer Benutzungsstandard immer erhalten. Es wird sich zeigen müssen, ob die Vorgaben der TSK-Umgebung nicht zu wenig allgemein sind. Der Benutzer ist bei der Ereignisauswahl auf eine bestimmte Methode festgelegt, unabhängig davon, ob er eine andere für sinnvoller hält. Die geplante Beeinflußung der Zustandsdarstellung wird nie alle denkbaren Auswahlkriterien für die Darstellung von Zustandsgrößen erfassen können. Denkbar sind die Darstellungen:

- Detaillierte Anzeige aller Zustandsgrößen
- Alleinige Anzeige der Prozesse
- Anzeige der Prozesse, die aktiv an einem Ereignis teilnehmen
- Spezielles Hervorheben von Prozessen in bestimmten Zuständen wie Deadlock, Warten auf Ereignisteilnahme und Teilnahme an einem Ereignis
- Anzeige nur der Blätter des Baums der Prozeßhierachien
- ...

Da der Phantasie und den Ansprüchen an dieser Stelle keine Grenzen gesetzt sind, wird es wahrscheinlich immer wieder vorkommen, daß die TSK-Alternativen nur einen Kompromiß darstellen.

Den Produkten von beiden Ansätzen ist gemeinsam, daß sie von nachträglichen Verbesserungen der Generatoren profitieren können. Seitdem in der letzten CENTAUR-Version die Pretty Printer inkrementell arbeiten, besitzt auch jede von CENTAUR generierte Umgebung dieses Feature. Ebenso ist es denkbar, in Zukunft, TSK-Terme parallel auf mehreren Rechnern auswerten zu lassen oder auf eine Maschinensprache für eine spezielle TSK-Hardware abzubilden.

Zu den Eigenschaften, die momentan von den Generatoren auf die Produkte vererbt werden, zählen:

- CENTAUR
  - + graphische Benutzeroberfläche mit komfortablen Bedienelementen
  - + Zwischenablage für Teilbäume
  - + inkrementelle Anzeige von Ergebnissen in mehreren Fenstern
  - + Darstellung von Ergebnissen in verschiedenen Farben, Größen und Schriftarten.
  - + Möglichkeit der Spezifikation von inkrementellen Tools
  - + Interaktives Ändern der Produkt-Spezifikationen
  - - Produkt ohne CENTAUR nicht lauffähig
  - - Unterbrechungen durch Garbage Collector möglich
- TSK
  - + Kommunikationsberechnung berücksichtigt echte Parallelität[40]
  - + Verschiedene Strategien für die automatisierte Ausführung von Ereignissen wählbar
  - + Allgemeine Analysefunktionen denkbar[41].
  - + Verarbeitung von unvollständigen Quelltexten
  - + Ausblenden von Prozeßinstanzen
  - + Benutzerfunktionen wie *find-action-prefix*[42] vorhanden
  - + Kontrolle der Simulation durch Simulationsgraph[43]
  - - Konfiguration der Umgebung nur eingeschränkt möglich
  - - Berechnung der Kommunikation erfolgt immer in vollster Allgemeinheit[44]

Als Fazit läßt sich festhalten, daß die Entwicklung einer Simulationsumgebung mit TSK unter folgenden Voraussetzungen entscheidende Vorteile aufweist:

- Die Quellsprache ist gut geeignet für eine Abbildung auf die TSK-Terme
- Die vorgegebene TSK-Simulationsumgebung erfüllt die Anforderungen des Sprachdefinierers

Sind diese gegeben, ermöglicht die TSK-Methode dem Sprachentwickler eine größere Konzentration auf die Quellsprachensemantik bei insgesamt stark reduzierten Arbeitsaufwand. Will man bspw. mit CENTAUR eine ähnlich komfortable Umgebung generieren, wie sie für TSK geplant ist, vergrößert sich die momentane Aufwandsdifferenz nochmals. Über die Interaktivität der Sprachentwicklung mit TSK kann leider, mangels Erfahrung, keine Aussage gemacht werden.

Das Ergebnis der Betrachtung gibt Anlaß zu der Überlegung, ob die Vorteile der TSK-Methode nicht auch durch Sprachdefinierungsbibliotheken ereichbar wären. Könnte der CENTAUR Sprachdefinierer auf eine TYPOL-Bibliothek mit Regelsammlungen für Sprachen mit Kommunikations- und Prozeßkonzept zurückgreifen ständen ihm außerdem eine Auswahl entsprechender vordefinierter und konfigurierbarer Umgebungen zu Verfügung, so besäße er die gleiche Ausgangsituation wie der TSK Sprachdefinierer. In diesem Fall wäre sogar keine dritte Übersetzungssprache notwendig. Diese Vision läßt sich treffend als *Wiederverwendung von Spezifikationen für Programmierumgebungen* bezeichnen. Auf diese Art wären die Arbeitsersparnis der TSK-Methode mit der Flexibilität der Entwicklung mit CENTAUR vereinbar.

[40] Diese Möglichkeit verführt dazu, bspw. die Semantik von LOTOS um echte parallele Ausführung von Ereignissen zu erweitern

[41] Da alle mit TSK durchgeführten Sprachdefinitionen davon profitieren könnten, motiviert sich die Bereitstellung von allgemeinen Analysetechniken wie *Deadlock-* und *Coverage-Analysis*

[42] Diese Funktion findet für ein angebotenes Ereignis die Stelle(n) im Quelltext, die für das Ereignis verantwortlich war(en)

[43] Eigentlich Simulationsbaum. Die Grapheigenschaft resultiert nur aus dem Zusammenführen von Zuständen, die durch echt parallele Ereignisse entstanden, und nicht aus dem allgemeinen Erkennen von identischen Zuständen.

[44] Dieser Umstand ist wahrscheinlich die Ursache dafür, daß der TSK-LOTOS-Simulator dem CENTAUR Kontrahenten in der Ausführungsgeschwindigkeit unterlegen ist

Literatur

[AC90] Isabelle Attali and Jacques Chazarain. Functional evaluation of natural semantics specifications. Technical report, INRIA research report 1218, May 1990.

[ACG92] Isabelle Attali, Jacques Chazarain, and Serge Gilette. Incremental evaluation of natural semantics specifications. Technical report, INRIA research report, 1992.

[BFLO86] J. P. Briand, M.C. Fehri, L. Logrippo, and A. Obaid. Executing lotos specifications. In *Proceedings of the sixth international workshop on protocol specification, testing and verification,*, 1986.

[Cen92] Typol reference manual, 1992.

[Des88] Thierry Despeyroux. Typol: a formalism to implement natural semantics. Technical report, INRIA research report 94, 1988.

[DM85] P. Deransart and J. Maluszynski. Relating logic programs and attribute grammars. In *J. Logic Programming vol 2*, pages 119–155, 1985.

[Eer90] Henk Eertink. Lite common representation, 1990.

[Hog89] Dieter Hogrefe. *Estelle, LOTOS und SDL.* Springer Verlag, 1989.

[ISO87] Lotos: Language for the temporal ordering specification of observational behaviour, International Standard ISO/IS 8807, 1987.

[JJWW91] P. Jager, W. Jonker, A. Wammes, and J. Wester. On the generation of interactive programming environments. *GIPE II*, 1991.

[Kah87] Gilles Kahn. Natural semantics. In *Lecture Notes in Computer Science 247*, 1987.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[Thing] Christoph Thies. *TSK - Eine Methode zur Generierung von interaktiven Simulationsumgebungen.* PhD thesis, Technische Hochschule Darmstadt, In Vorbereitung.

[TJW+92] V. Tijssen, H. Joosten, A. Wammes, J. Wester, T. Kühne, and C. Thies. User-requirements for a lotos simulator. Technical report, PTT Research Tele-Informatics, TU Darmstadt, FG Praktische Informatik, January 1992.

[vE86] Peter van Eijk. A comparison of behavioural language simulators. In *Proceedings of the sixth international workshop on protocol specification, testing and verification*, 1986.

[vE88] Peter van Eijk. *Software tools for the specification language LOTOS.* PhD thesis, Universiteit Twente, 1988.

A Basic-Lotos-Spezifikationen

A 1 Tokenring Spezifikation

```
specification token_ring[f1t2, f2t3, f3t1] : noexit

behaviour
    (node[f3t1, f1t2] |[f1t2]| node[f1t2, f2t3])
    |[f2t3, f3t1]| (node[f2t3, f3t1] ||| new_token[f3t1])

where
  process new_token[new] : noexit
  :=
    new; new_token[new]
    []
    (
      hide kill_token_gen in
      kill_token_gen; stop)
  endproc

  process node[data_in, data_out] : noexit
  :=
    data_in;
    ( data_out; node[data_in, data_out]
      []
      (cpu[data_in, data_out] >> data_out; node[data_in, data_out])
    )
  where
     process cpu[data_in, data_out] : exit
     :=
       hide calc_data, calcing in
         calc_data; (calcing; exit ||| bypass[data_in, data_out])
     endproc

     process bypass[data_in, data_out] : exit
     :=
        data_in; data_out; bypass[data_in, data_out]
        []
        exit
     endproc

  endproc
endspec
```

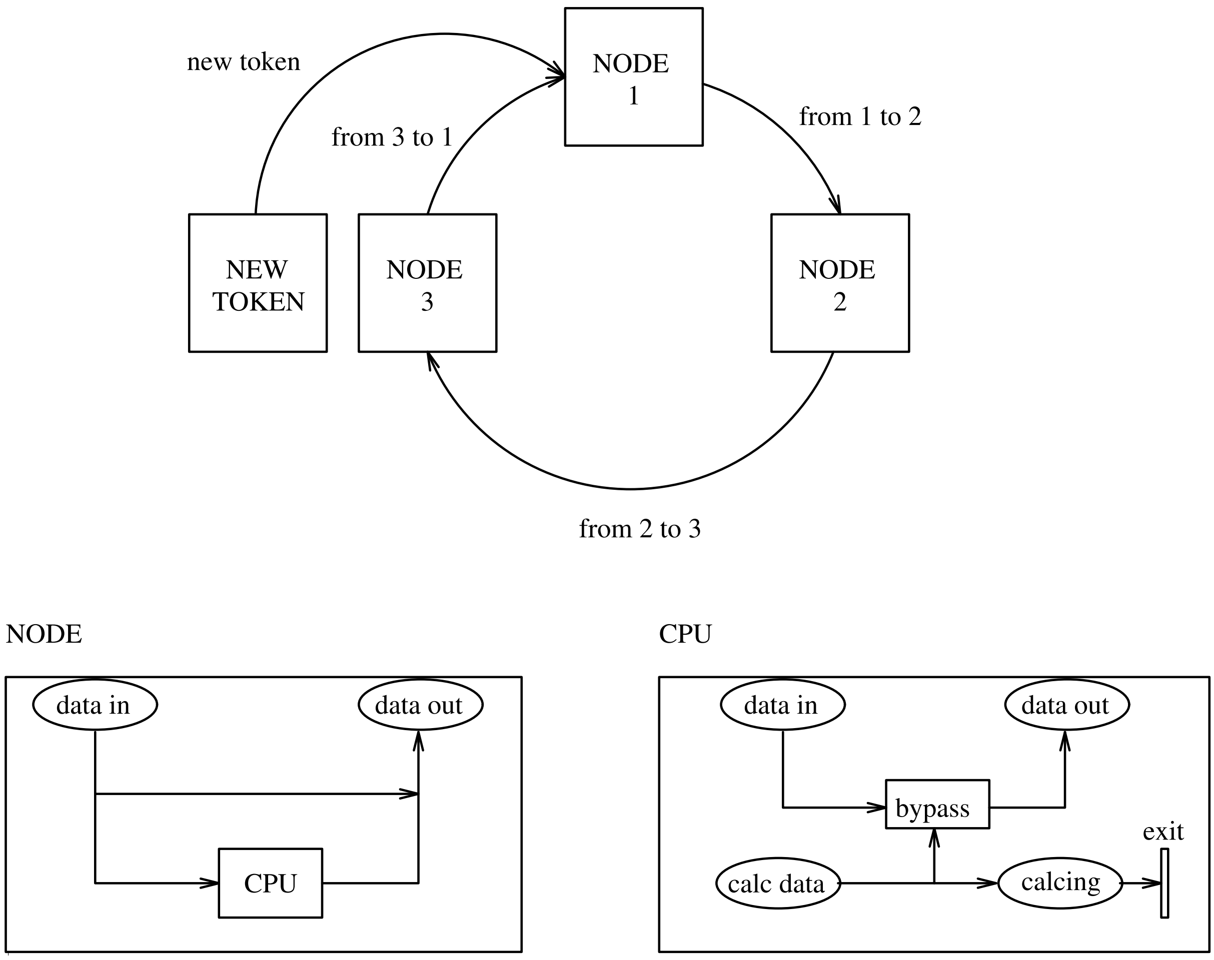

Abbildung 11: Schematische Darstellung eines Tokenrings

A 2 Spezifikation mit expotentiellen Menüwachstum

```
specification tower_of_power[a] : noexit

behaviour
  new[a] || (a; tower_of_power[a])

where

  process new[a] : noexit
  := (a; new[a])
     []
     (a; new[a])
  endproc

endspec
```

A 3 TwoSlotBuffer Spezifikation

```
specification two_slot_buffer_test[inp, outp] : noexit

behaviour
  two_slot_buffer[inp, outp]

where
  process two_slot_buffer[inp, outp]: noexit :=
    hide middle in
      buffer[inp, middle] |[ middle ]| buffer[middle, outp]

  where
    process buffer[inp, outp]: noexit :=
      inp; outp; buffer[inp, outp]
    endproc
  endproc
endspec
```

B Basic-Lotos-Syntaxdefinition

```
module SLOTOS
sorts
  Letter Digit Normal_Character Special_Character
  Underscore_Symbol Normal_Or_Underscore Identifier
  Special_Identifier

  Action_Denotation Behaviour_Expression
  Opt_Gate_List Functionality
  Gate_Identifier Gate_Identifier_List
  Opt_Local_Defs Process_Definition_List
  Process_Definition Process_Header Process_Identifier
  Specification
  Specification_Header Specification_Identifier

lexical syntax
  [ \t\n]                                        ->     LAYOUT
  [a-zA-Z]                                       ->     Letter
  [0-9]                                          ->     Digit
  Letter                                         ->     Normal_Character
  Digit                                          ->     Normal_Character
  [#%&*+\_./<=>@\\\^~{}]                         ->     Special_Character
  [_]                                            ->     Underscore_Symbol
  Normal_Character                               ->     Normal_Or_Underscore
  Underscore_Symbol                              ->     Normal_Or_Underscore
  Letter                                         ->     Identifier
  Letter Normal_Or_Underscore* Normal_Character  ->     Identifier
  Special_Character+                             ->     Special_Identifier
  Digit                                          ->     Special_Identifier
  Digit Normal_Or_Underscore* Normal_Character   ->     Special_Identifier

context-free syntax

spec_def:          Specification_Header
                   "behaviour" Behaviour_Expression
                   Opt_Local_Defs
                   "endspec"                     -> Specification

spec_header:       "specification"
                   Specification_Identifier
                   Opt_Gate_List
                   ":" Functionality             -> Specification_Header

gate_list:        "[" Gate_Identifier_List "]"   -> Opt_Gate_List
empty_gate_list:  % empty %                      -> Opt_Gate_List

f_exit:           "exit"                         -> Functionality
f_noexit:         "noexit"                       -> Functionality
```

```
local_def:        "where"
                  Process_Definition_List        -> Opt_Local_Defs
empty_local_def:  % empty %                      -> Opt_Local_Defs

                  Process_Definition+: proc_defs -> Process_Definition_List

proc_def:         Process_Header
                  ":=" Behaviour_Expression
                  Opt_Local_Defs
                  "endproc"                      -> Process_Definition

proc_header:      "process" Process_Identifier
                  Opt_Gate_List
                  ":" Functionality              -> Process_Header

hide:             "hide" Gate_Identifier_List
                  "in" Behaviour_Expression      -> Behaviour_Expression

enable:           Behaviour_Expression ">>"
                  Behaviour_Expression           -> Behaviour_Expression

disable:          Behaviour_Expression "[>"
                  Behaviour_Expression           -> Behaviour_Expression right

synchronise:      Behaviour_Expression "||"
                  Behaviour_Expression           -> Behaviour_Expression right

interleave:       Behaviour_Expression "|||"
                  Behaviour_Expression           -> Behaviour_Expression right

par:              Behaviour_Expression "|[" "]|"
                  Behaviour_Expression           -> Behaviour_Expression right

par_over:         Behaviour_Expression
                  "|[" Gate_Identifier_List "]|"
                  Behaviour_Expression           -> Behaviour_Expression

choice:           Behaviour_Expression "[]"
                  Behaviour_Expression           -> Behaviour_Expression right

prefix:           Action_Denotation ";"
                  Behaviour_Expression           -> Behaviour_Expression

stop:             "stop"                         -> Behaviour_Expression
exit:             "exit"                         -> Behaviour_Expression
proc_call:        Process_Identifier
                  Opt_Gate_List                  -> Behaviour_Expression

                  "(" Behaviour_Expression ")"   -> Behaviour_Expression bracket


action:           Gate_Identifier                ->  Action_Denotation
i:                "i"                            ->  Action_Denotation
```

```
                  {Gate_Identifier ","}+:gids    -> Gate_Identifier_List

                  Identifier                     ->  Specification_Identifier
                  Identifier                     ->  Process_Identifier
                  Identifier                     ->  Gate_Identifier

priorities
 {"hide" Gate_Identifier_List "in" Behaviour_Expression                    ->
 Behaviour_Expression} <
 Behaviour_Expression ">>" Behaviour_Expression                            ->
 Behaviour_Expression  <
 Behaviour_Expression "[>" Behaviour_Expression                            ->
 Behaviour_Expression  <
 {Behaviour_Expression "||" Behaviour_Expression                           ->
  Behaviour_Expression,
  Behaviour_Expression "|||" Behaviour_Expression                          ->
  Behaviour_Expression,
  Behaviour_Expression "|[" Gate_Identifier_List "]|" Behaviour_Expression ->
  Behaviour_Expression} <
 Behaviour_Expression "[]" Behaviour_Expression                            ->
 Behaviour_Expression  <
 Action_Denotation ";" Behaviour_Expression                                ->
 Behaviour_Expression
```

C METAL-Definitionen

C 1 Menürepräsentation

```
definition of MENU is

abstract syntax

  events        -> EVENT * ...;                -- MENU = List of EVENTS
  event         -> EVENT_NAME VISIBLE FOLLOW_STATE;

  event_name    -> implemented as IDENTIFIER;   -- action denotation

  external      -> implemented as SINGLETON;
  internal      -> implemented as SINGLETON;

  follow_state  -> implemented as TREE;

  no_event      -> implemented as SINGLETON;

  EVENTS        ::= events;
  EVENT         ::= event no_event;
  EVENT_NAME    ::= event_name;
  VISIBLE       ::= external internal;
  FOLLOW_STATE  ::= follow_state;

end definition
```

C 2 Prozeßenvironment

```
definition of ENV is

abstract syntax

  penv          -> PMAP * ...;                 -- Environment = List of Maps
  pmap          -> PACCESS * ...;              -- Lookupmap for one level

  paccess       -> PDEF PPATH;                 -- Process definition&location

  pdef          -> implemented as TREE;        -- SLOTOS Process definition
  ppath         -> implemented as TREE;        -- PSP Path to location

  PENV          ::= penv;                      -- Process environment
  PMAP          ::= pmap;                      -- Local environment

  PACCESS       ::= paccess;                   -- Process definition access

  PDEF          ::= pdef;                      -- Process definition
  PPATH         ::= ppath;                     -- Location of definition

end definition
```

C 3 Zustandsrepräsentation

```
definition of STATE is

abstract syntax

  s_choice      -> S_NODE S_NODE;                -- SE [] SE
  s_enable      -> S_NODE S_NODE;                -- SE >> SE
  s_disable     -> S_NODE S_NODE;                -- SE [> SE
  s_par         -> S_NODE S_NODE GIDS;           -- SE |[a, ...]| SE
  s_hide        -> S_NODE GIDS;                  -- hide a, ... in SE
  s_instance    -> S_NODE PID RELABEL_MAP;       -- ProcessInstance(PID, [a->b])
  s_behaviour   -> BEHAVIOUR LOCATION;           -- BehaviourExpression(BEXP)
  s_stop        -> implemented as SINGLETON;     -- shortcut for Bexp(stop)
  s_exit        -> implemented as SINGLETON;     -- shortcut for Bexp(exit)

  behaviour     -> implemented as TREE;          -- Subtree of Specification
  location      -> implemented as TREE;          -- Accesspath to Subtree

  gid_list      -> implemented as TREE;          -- SLOTOS Identifier list
  all_gids      -> implemented as SINGLETON;     -- represents all Identifiers

  gid           -> implemented as IDENTIFIER;    -- Gate Identifier
  pid           -> implemented as IDENTIFIER;    -- Process Identifier

  relabel_map   -> GID_RELABEL * ...;            -- relabeling of GateIds
  gid_relabel   -> GID GID;                      -- relabeling of GateId


  S_NODE        ::= s_choice s_enable s_disable
    s_par
    s_hide s_instance s_behaviour
                    s_stop s_exit;

  BEHAVIOUR     ::= behaviour;
  LOCATION      ::= location;

  GIDS          ::= gid_list all_gids;
  RELABEL_MAP   ::= relabel_map;
  GID_RELABEL   ::= gid_relabel;
  GID           ::= gid;
  PID           ::= pid;

end definition
```

D TYPOL-Spezifikation

```
program state is

use             SLOTOS;
use             STATE;

import          ANNOT_SPEC(|- Specification)
                from annot_spec;

import          EXPAND(Behaviour_Expression => S_NODE)
                from expand;

import          prmess(string, string, integer)
                from prolog;

export          INITIALIZE(spec => state)
                as init_state(spec) = state;


set INITIALIZE is

-- used by      Simulator
--
-- taking       the original specification
--
-- for          creating the inital state from the given specification.
--              The initial state is the expanded
--              extended-behaviour-expression of the behaviour part
--              of the specification.
--              In order to expand process calls, access to the
--              process definitions has to be provided.
--              This is done by ANNOT_SPEC, that annotates the behaviour
--              expression and recursively all process definitions.
--
-- returning    the expanded initial state, which is not simplyfied.


judgement |- Specification => S_NODE;

-- translate Lotos specification into simulator state.


ANNOT_SPEC(|- spec_def(SpecHeader, Bexp, OptPdefs)) &
prmess("reports", "SLOTOS", 1) &
EXPAND(Bexp => State)
-----------------------------------------------------
|- spec_def(SpecHeader, Bexp, OptPdefs) => State;
   do prmess("reports", "SLOTOS", 3);

end INITIALIZE;

end state;
```

```
program expand is

use             SLOTOS;
use             STATE;
use             PSP;


import          MAKE_RELABEL(Opt_Gate_List, Opt_Gate_List => RELABEL_MAP)
                from relabel;

import          GET_PDEF(Behaviour_Expression -<
                         Opt_Gate_List, Behaviour_Expression @ PATH)
                from pdef_access;


set EXPAND is

-- used by      INITIALIZE in state,
--              CALC_MENU in menu,
--              ENABLE_FOLLOW in enable
--
-- taking       a behaviour expression
--
-- for          calculating the corresponding state of the given
--              behaviour expression.
--              Behaviour expression nodes are translated into
--              state nodes until all leafs of the generated state are
--              experiment offers.
--              In order to enable the later expansion of state leaves
--              their location within the specification are memorized.
--              Process calls are expanded into process instances using
--              the process definitions previously stored at the process calls.
--              The various incarnations of the SLOTOS parallel operator
--              are expressed through the general state operator 's_par'.
--
-- returning    the state corresponding to the given behaviour expression.


judgement |- Behaviour_Expression => S_NODE;

-- translate Lotos behaviour expression into simulator state


|- stop() =>
   s_stop();

|- exit() =>
   s_exit();

|- prefix(ActionDenotaion, Bexp) =>
   s_behaviour(behaviour(prefix(ActionDenotaion, Bexp)),
               location(s(subject, int 2))
              );
```

```
|- Bexp1 => State1 &
|- Bexp2 => State2
----------------------------
|- choice(Bexp1, Bexp2) =>
   s_choice(State1, State2);

|- Bexp1 => State
--------------------------------------------------------------------------------
|- enable(Bexp1, Bexp2) =>
   s_enable(State, s_behaviour(behaviour(Bexp2), location(s(subject, int 2))));

|- Bexp1 => State1 &
|- Bexp2 => State2
-----------------------------
|- disable(Bexp1, Bexp2) =>
   s_disable(State1, State2);

|- Bexp1 => State1 &
|- Bexp2 => State2
-------------------------------------
|- synchronise(Bexp1, Bexp2) =>
   s_par(State1, State2, all_gids());

|- Bexp1 => State1 &
|- Bexp2 => State2
-------------------------------------------
|- interleave(Bexp1, Bexp2) =>
   s_par(State1, State2, gid_list(gids[]));

|- Bexp1 => State1 &
|- Bexp2 => State2
-------------------------------------------
|- par(Bexp1, Bexp2) =>
   s_par(State1, State2, gid_list(gids[]));

|- Bexp1 => State1 &
|- Bexp2 => State2
--------------------------------------------------------
|- par_over(Bexp1, SynchronizationGids, Bexp2) =>
   s_par(State1, State2, gid_list(SynchronizationGids));

|- Bexp => State
---------------------------------------
|- hide(HiddenGids, Bexp) =>
   s_hide(State, gid_list(HiddenGids));

GET_PDEF(proc_call(identifier Pid, A_Gids) -< F_Gids, Bexp @ BexpPath) &
MAKE_RELABEL(F_Gids, A_Gids => RelabelMap) &
withsubject(BexpPath: EXPAND(Bexp => State))
-------------------------------------------------------------------------
|- proc_call(identifier Pid, A_Gids) =>
   s_instance(State, pid Pid, RelabelMap);

end EXPAND;

end expand;
```

```
program menu is

use             SLOTOS;
use             STATE;
use             MENU;


import          MERGE(S_NODE |- EVENTS, EVENTS >- EVENTS),
                JOIN(EVENTS, EVENTS >- EVENTS)
                from merge;

import          INTERSECT(S_NODE |- EVENTS, EVENTS >- EVENTS)
                from intersect;

import          SPLIT_SYNCED_INTERLEAVED(GIDS |- EVENTS -< EVENTS, EVENTS)
                from split;

import          HIDE_EVENTS(Gate_Identifier_List |- EVENTS -> EVENTS)
                from hide;

import          ENABLE(S_NODE |- EVENTS -> EVENTS)
                from enable;

import          RELABEL_MENU(PID, RELABEL_MAP |- MENU -> MENU)
                from relabel;

import          EXPAND(Behaviour_Expression => S_NODE)
                from expand;

import          prmess(string, string, integer)
                from prolog;

export          CALC_MENU(|- state => menu) &
                prmess("reports", "SLOTOS", 5)
                as calc_menu(state) = menu;


set CALC_MENU is

-- used by      Simulator
--
-- taking       a state
--
-- for          calculating a set of events possible with the given state.
--              The leaves of the given state are experiment offers, that
--              are directly translated into events.
--              Inner nodes of the state are operators, that specifically
--              combine or modify the menus of their descendents.
--              Because the calculated events already contain their
--              resulting states, in some cases state leaves, that
--              represent behaviour expressions, are expanded to states.
--              This ensures that the resulting state of an event is
--              a state, which leaves are all experiment offers.
--
-- returning    the menu of events, representing possible state transitions


judgement |- S_NODE => EVENTS;

-- translate simulator state into menu of events.


|- s_stop() =>
   events[];

|- s_exit() =>
   events[event(event_name "_tick", external(), follow_state(s_stop()))];

withsubject(BexpLocation: EXPAND(Bexp => State))
-----------------------------------------------------------------------
|- s_behaviour(behaviour(prefix(i(), Bexp)), location(BexpLocation)) =>
   events[event(event_name "i", internal(), follow_state(State))];

withsubject(BexpLocation: EXPAND(Bexp => State))
-------------------------------------------------------------------------------
|- s_behaviour(behaviour(prefix(identifier ActionDenotation, Bexp)),
               location(BexpLocation)) =>
   events[event(event_name ActionDenotation, external(), follow_state(State))];
```

```
|- State1 => Menu1 &
|- State2 => Menu2 &
JOIN(Menu1, Menu2 >- Menu)
------------------------------------
|- s_choice(State1, State2) => Menu;

|- State1 => Menu &
ENABLE(State2 |- Menu -> Menu')
-------------------------------------
|- s_enable(State1, State2) => Menu';

|- State1 => Menu1 &
|- State2 => Menu2 &
MERGE(s_disable(_, State2) |- Menu1, Menu2 >- Menu)
---------------------------------------------------
|- s_disable(State1, State2) => Menu;

|- State1 => Menu1 &
|- State2 => Menu2 &
SPLIT_SYNCED_INTERLEAVED(Gids |- Menu1 -< Sync1, Inter1) &
SPLIT_SYNCED_INTERLEAVED(Gids |- Menu2 -< Sync2, Inter2) &
INTERSECT(s_par(State1, State2, Gids) |- Sync1, Sync2 >- Menu) &
MERGE(s_par(State1, State2, Gids) |- Inter1, Inter2 >- Menu') &
JOIN(Menu, Menu' >- Menu'')
-----------------------------------------------------------------
|- s_par(State1, State2, Gids) => Menu'';

|- State => Menu &
HIDE_EVENTS(Gids |- Menu -> Menu')
------------------------------------------
|- s_hide(State, gid_list(Gids)) => Menu';

|- State => Menu &
RELABEL_MENU(Pid, RelabelMap |- Menu -> Menu')
-----------------------------------------------
|- s_instance(State, Pid, RelabelMap) => Menu';

end CALC_MENU;

end menu;
```

```
program merge is

use             STATE;
use             MENU;

import          FOLLOW_STATE(S_NODE |- EVENT, EVENT >- EVENT)
                from follow_state;

define true:    () -> BOOLEAN;
define false:   () -> BOOLEAN;

set JOIN is

-- used by      INTERSECT in intersect
--              CALC_MENU in menu
--
-- taking       two menus
--
-- for          joining the given menus to one menu.
--              The method preserves the order of the events in the
--              given menus.
--              Apart from combining menus that were calculated
--              seperately, the operation join exactly reflects the
--              semantics of the state operator 's_choice'.
--
-- returning    the menu containing both given menus


judgement |- EVENTS, EVENTS >- EVENTS;

-- join two menus to one menu.

|- events[], Menu >- Menu;

|- LeftMenuRest, RightMenu >- MenuRest
--------------------------------------------------------------------
|- events[Event.LeftMenuRest], RightMenu >- events[Event.MenuRest];

end JOIN;



set MERGE is

-- used by      CALC_MENU in menu
--
-- taking       a state operator, two menus
--
-- for          calculating the menu of the given state operator having the
--              descendents that yield the given menus.
--              The method used, preserving the order of the events in the
--              given menus, to combine the given menus is the same
--              for all operators. The specific resulting behaviour
--              is calculated in FOLLOW_STATE.
--              Some state operators yield different resulting states,
--              depending on the desendent the event came from.
--              As a result the event position not used in the judgement
--              of FOLLOW_STATE is identified by the operator 'no_event'.
--
-- returning    the menu of the given state operator


judgement S_NODE |- EVENTS, EVENTS >- EVENTS;

-- given a state operator, compose the menu of the given state operator
-- that has descendents which yield the two menus.


_ |- events[], events[] >- events[];

FOLLOW_STATE(Operator |- no_event(), Event >- Event') &
Operator |- events[], RightMenuRest >- MenuRest
------------------------------------------------------------------------------
Operator |- events[], events[Event.RightMenuRest] >- events[Event'.MenuRest];

FOLLOW_STATE(Operator |- Event, no_event() >- Event') &
Operator |- LeftMenuRest, RightMenu >- MenuRest
------------------------------------------------------------------------------
Operator |- events[Event.LeftMenuRest], RightMenu >- events[Event'.MenuRest];

end MERGE;

end merge;
```

```
program intersect is

use             STATE;
use             MENU;

import          JOIN(EVENTS, EVENTS >- EVENTS)
                from merge;

import          MEMBER_OF_EVENTS(EVENTS |- EVENT : EVENTS)
                from member;

import          FOLLOW_STATE(S_NODE |- EVENT, EVENT >- EVENT)
                from follow_state;

import          diff(EVENTS, EVENTS)
                from prolog;

set INTERSECT is

-- used by      CALC_MENU in menu
--
-- taking       a state operator, two menus
--
-- for          calculating the menu of the given state operator having the
--              descendents that yield the given menus.
--              Two events are combined to one event, that has the
--              resulting state of the combination of the given state operator
--              and the resulting states of the two given events.
--              The specific resulting behaviour is calculated in FOLLOW_STATE.
--              When projected on event names the given menus become
--              multisets. Therefore each event in the 'left' menu has
--              to be combined with each event in the 'right' menu,
--              if there event names are equal.
--
-- returning    the menu of the given state operator

judgement S_NODE |- EVENTS, EVENTS >- EVENTS;

-- given a state operator, compose the menu of the given state operator
-- that has descendents which yield the two menus.

_ |- events[], _ >- events[];
_ |- Menu, events[] >- events[];
  provided diff(Menu, events[]);

MEMBER_OF_EVENTS(RightMenu |- Event : RightMenuRest) &
Operator |- LeftMenuRest, RightMenu >- LeftMenu' &
MULTIPLY(Operator |- Event, RightMenuRest, LeftMenu' -> Menu'')
---------------------------------------------------------------
Operator |- events[Event.LeftMenuRest], RightMenu >- Menu'';

set MULTIPLY is

-- used by      INTERSECT
--
-- taking       a state operator, an event, two menus
--
-- for          calculating all possible combinations of the given
--              event, with the first of the given menus.
--              The second of the given menus is extended by the
--              menu resulting from the combination of the given
--              event with the first of the given menus.
--
-- returning    the menu of the given state operator

judgement S_NODE |- EVENT, EVENTS, EVENTS -> EVENTS;

-- given a state operator, extend the second menu with all combinations
-- combining the event and the first menu.

Operator |- _, events[], LeftMenu' -> LeftMenu';

FOLLOW_STATE(Operator |- Event, Event' >- Event'') &
INTERSECT(Operator |- events[Event], RightMenuRest >- RightMenu') &
JOIN(RightMenu', LeftMenu' >- Menu)
-------------------------------------------------------------------
Operator |- Event, events[Event'.RightMenuRest], LeftMenu' ->
            events[Event''.Menu];

end MULTIPLY;

end INTERSECT;

end intersect;
```

```
program split is

use             SLOTOS;
use             STATE;  -- GIDS
use             MENU;


define true:    () -> BOOLEAN;
define false:   () -> BOOLEAN;


import          MEMBER_OF_GIDS(GIDS |- EVENT_NAME : BOOLEAN)
                from member;

import          diff(EVENT_NAME, EVENT_NAME)
                from prolog;


set SPLIT_SYNCED_INTERLEAVED is

-- used by      CALC_MENU in menu
--
-- taking       a gate identifier list, a menu
--
-- for          seperating the given menu, considering the given
--              gate identifier list and the visibility of the events
--              in the given menu.
--              The menu is splitted in two menus:
--              One containing events that must synchronize with other events.
--              One containing events that do not have to synchronize
--              with other events.
--              Internal events are never synchronized.
--              External events are checked against membership in
--              the given gate identifier list.
--              The final decision, whether the event has to synchronize or
--              not, is made in SYNCHRONIZE.
--
-- returning    a menu of events to be synchronised,
--              a menu of independend events


judgement GIDS |- EVENTS -< EVENTS, EVENTS;

-- given a synchronizesation gate list, retrieve the menus of events to be
-- synchronized and of independent events from the menu guided by
-- the given synchronizesation gate list.


_ |- events[] -< events[], events[];

MEMBER_OF_GIDS(Gids |- EventName : Synchronize) &
Gids |- MenuRest -< SyncRest, IntRest &
SYNCHRONIZE(Synchronize |- SyncRest, IntRest
                           - event(EventName, external(), FollowState) >
                           SyncRest', IntRest')
----------------------------------------------------------------------
Gids |- events[event(EventName, external(), FollowState).MenuRest] -<
        SyncRest', IntRest';

Gids |- MenuRest -< SyncRest, IntRest
--------------------------------------------------------------------------
Gids |- events[event(EventName, internal(), FollowState).MenuRest] -<
        SyncRest, events[event(EventName, internal(), FollowState).IntRest];
```

```
set SYNCHRONIZE is

-- used by      SPLIT_SYNCED_INTERLEAVED
--
-- taking       a boolean 'membership' indicator to the the list of events
--              to be synchronized,
--              a menu of events to be synchronized,
--              a menu of indenpendend events,
--              an event, corresponding to the boolean indicator.
--
-- for          updating the given menus with the given event.
--              If the given event is a 'successful process termination'
--              it is added to the menu of events to be synchronized.
--              Otherwise the boolean indicator determines to which of
--              the given menus the given event is added.
--
-- returning    a menu of events to be synchronized,
--              a menu of indenpendend events


judgement BOOLEAN |- EVENTS, EVENTS  - EVENT >  EVENTS, EVENTS;

-- given a boolean 'membership' indicator, transform the menus into
-- menus extended by the event, sustaining the synchronizesation properties.


_ |-
  Synced, Interleaved
  - event(event_name "_tick", external(), FollowState) >
  events[event(event_name "_tick",external(),FollowState).Synced], Interleaved;

true()  |-
  Synced, Interleaved
  - event(EventName, external(), FollowState) >
  events[event(EventName, external(), FollowState).Synced], Interleaved;
  provided diff(EventName, event_name "_tick");

false() |-
  Synced, Interleaved
  - event(EventName, external(), FollowState) >
  Synced, events[event(EventName, external(), FollowState).Interleaved];
  provided diff(EventName, event_name "_tick");


end SYNCHRONIZE;

end SPLIT_SYNCED_INTERLEAVED;

end split;
```

```
program follow_state is

use             STATE;
use             MENU;

import          not_both_in(S_NODE, S_NODE, S_NODE, S_NODE)
                from logic;

import          diff(_, _)
                from prolog;


set FOLLOW_STATE is

-- used by      MERGE in merge
--              INTERSECT in intersect
--
-- taking       a state operator, two menus
--
-- for          determing the resulting state of the given state
--              operator, under the execution of the two given events.
--              The menu operator 'no_event' indicates that there is
--              just one event executed and simultanously determines
--              the descendent that will stay unchanged.
--              In the case of the state operator 's_par' the form
--              containing 'no_event' is transformed into a form
--              containing one event with a changed resulting state and
--              one event with a unchanged resulting state.
--              If possible, the resulting state of the given state operator
--              will be simplyfied individually for each state operator.
--
-- returning    the event with the resulting state of the given state operator


judgement S_NODE |- EVENT, EVENT >- EVENT;

-- given a state operator, compose the event of the given state operator
-- that has descendents which offer the two events.


s_disable(_, _) |-
  event(event_name "_tick", external(), follow_state(State1')), no_event() >-
  event(event_name "_tick", external(), follow_state(State1'));

s_disable(_, State2) |-
  event(EventName, Visible, follow_state(s_stop())), no_event() >-
  event(EventName, Visible, follow_state(State2));
  provided diff(EventName, event_name "_tick");

s_disable(_, State2) |-
  event(EventName, Visible, follow_state(State1')), no_event() >-
  event(EventName, Visible, follow_state(s_disable(State1', State2)));
  provided diff(EventName, event_name "_tick") &
           diff(State1', s_stop());

s_disable(_, _) |-
  no_event(), Event >-
  Event;


s_par(_, _, GidList) |-
  event(EventName, Visible, follow_state(State1')),
  event(EventName, Visible, follow_state(State2)) >- Event
----------------------------------------------------------------------
s_par(State1, State2, GidList) |-
  event(EventName, Visible, follow_state(State1')), no_event() >- Event;

s_par(_, _, GidList) |-
  event(EventName, Visible, follow_state(State1)),
  event(EventName, Visible, follow_state(State2')) >- Event
------------------------------------------------------------------------
s_par(State1, State2, GidList) |-
  no_event(), event(EventName, Visible, follow_state(State2')) >- Event;
```

```
s_par(_, _, _) |-
  event(EventName, Visible, follow_state(s_stop())),
  event(EventName, Visible, follow_state(s_stop())) >-
  event(EventName, Visible, follow_state(s_stop()));

s_par(_, _, _) |-
  event(EventName, Visible, follow_state(s_stop())),
  event(EventName, Visible, follow_state(s_exit())) >-
  event(EventName, Visible, follow_state(s_stop()));

s_par(_, _, _) |-
  event(EventName, Visible, follow_state(s_exit())),
  event(EventName, Visible, follow_state(s_stop())) >-
  event(EventName, Visible, follow_state(s_stop()));

s_par(_, _, _) |-
  event(EventName, Visible, follow_state(s_exit())),
  event(EventName, Visible, follow_state(s_exit())) >-
  event(EventName, Visible, follow_state(s_exit()));

s_par(_, _, all_gids()) |-
  event(EventName, Visible, follow_state(s_stop())),
  event(EventName, Visible, follow_state(State)) >-
  event(EventName, Visible, follow_state(s_stop()));
  provided diff(State, s_stop()) &
           diff(State, s_exit());

s_par(_, _, all_gids()) |-
  event(EventName, Visible, follow_state(State)),
  event(EventName, Visible, follow_state(s_stop())) >-
  event(EventName, Visible, follow_state(s_stop()));
  provided diff(State, s_stop()) &
           diff(State, s_exit());

s_par(_, _, all_gids()) |-
  event(EventName, Visible, follow_state(State1)),
  event(EventName, Visible, follow_state(State2)) >-
  event(EventName, Visible, follow_state(s_par(State1, State2, all_gids())));
  provided diff(State1, s_stop()) &
           diff(State2, s_stop()) &
           not_both_in(State1, State2, s_stop(), s_exit());

s_par(_, _, gid_list(GL)) |-
  event(EventName, Visible, follow_state(State1)),
  event(EventName, Visible, follow_state(State2)) >-
  event(EventName, Visible, follow_state(s_par(State1, State2, gid_list(GL))));
  provided not_both_in(State1, State2, s_stop(), s_exit());

end FOLLOW_STATE;

end follow_state;
```

```
program enable is

use             SLOTOS;
use             STATE;
use             MENU;

import          EXPAND(Behaviour_Expression => S_NODE)
                from expand;

import          diff(_, _)
                from prolog;


set ENABLE is

-- used by      CALC_MENU in menu
--
-- taking       a state to be enabled, a menu
--
-- for          iterating over the events of the given menu in order
--              to change the resulting states of the events in the given menu
--              to the resulting state of the state operator 'enable',
--              that has an enabling descendent yielding the given menu and
--              as the enabled descendent the given state.
--
-- returning    the menu of the state operator 'enable'.


judgement S_NODE |- EVENTS -> EVENTS;

-- given a state to be enabled, transform the menu into the menu
-- of the given state operator 'enable'.


_ |- events[] -> events[];

ENABLE_FOLLOW(AcceptState |- Event -> Event') &
AcceptState |- MenuRest -> MenuRest'
-----------------------------------------------------------------
AcceptState |- events[Event.MenuRest] -> events[Event'.MenuRest'];
```

```
set ENABLE_FOLLOW is

-- used by      ENABLE
--
-- taking       a state to be enabled, an event
--
-- for          calculating the event with the resulting state of
--              the state operator 'enable' under the execution of
--              the given event.
--              If the given event is the 'successful process termination'
--              the given state will be expanded and will be used for
--              the resulting state of the event returned.
--              Otherwise the event returned has the simplified
--              resulting state of the state operator 'enable' under
--              the execution of the given event.
--
-- returning    the event with the resulting state of
--              the state operator 'enable'.


judgement S_NODE |- EVENT -> EVENT;

-- given a state to be enabled, transform the event into the event of
-- the state operator 'enable'.


withsubject(BexpLocation: EXPAND(Bexp => AcceptState))
-------------------------------------------------------------------
s_behaviour(behaviour(Bexp), location(BexpLocation)) |-
  event(event_name "_tick", external(), _) ->
  event(event_name "_tick", internal(), follow_state(AcceptState));

_ |-
  event(event_name "_tick", internal(), follow_state(s_stop())) ->
  event(event_name "_tick", internal(), follow_state(s_stop()));

AcceptState |-
  event(event_name "_tick", internal(), follow_state(State)) ->
  event(event_name "_tick", internal(),
        follow_state(s_enable(State, AcceptState)));
  provided diff(State, s_stop());

_ |-
  event(EventName, Visible, follow_state(s_stop())) ->
  event(EventName, Visible, follow_state(s_stop()));
  provided diff(EventName, event_name "_tick");

AcceptState |-
  event(EventName, Visible, follow_state(State)) ->
  event(EventName, Visible, follow_state(s_enable(State, AcceptState)));
  provided diff(EventName, event_name "_tick") &
           diff(State, s_stop());

end ENABLE_FOLLOW;

end ENABLE;

end enable;
```

```
program hide is

use             SLOTOS;
use             STATE;
use             MENU;

import          not_hide(S_NODE)
                from logic;

import          diff(S_NODE, S_NODE)
                from prolog;


define true:    () -> BOOLEAN;
define false:   () -> BOOLEAN;

import          MEMBER_OF_GIDS(GIDS |- EVENT_NAME : BOOLEAN)
                from member;


set HIDE_EVENTS is

-- used by      CALC_MENU in menu
--
-- taking       a gate identifier list, a menu
--
-- for          changing the visibilty of those events specified in
--              the given gate identifier list to internal.
--              The given gate identifier list is also used to create
--              the resulting state, that normally contains the state
--              operator 'hide' which caused the hiding of events.
--              Internal events are not hidden again.
--
-- returning    the given menu with the specified events made internal.


judgement Gate_Identifier_List |- EVENTS -> EVENTS;

-- given list of gates to be hidden, transform the menu into the menu
-- of the state operator 'hide'.


_ |- events[] -> events[];

MEMBER_OF_GIDS(gid_list(Gids) |- EventName : HasToBeHidden) &
HIDE(HasToBeHidden => Visible) &
HIDE_FOLLOW(Gids |- event(EventName, Visible, follow_state(State)) -> Event) &
Gids |- MenuRest-> MenuRest'
------------------------------------------------------------------------------
Gids |- events[event(EventName, external(), follow_state(State)).MenuRest] ->
        events[Event.MenuRest'];

HIDE_FOLLOW(Gids |- event(EventName,internal(),follow_state(State)) -> Event) &
Gids |- MenuRest-> MenuRest'
-------------------------------------------------------------------------------
Gids |- events[event(EventName, internal(), follow_state(State)).MenuRest] ->
        events[Event.MenuRest'];


set HIDE is

-- used by      HIDE_EVENTS
--
-- taking       a boolean indicator for the need of hiding.
--
-- for          translating the need for hiding into the corresponding
--              status of visibility.
--
-- returning    the new status of visibilty.


judgement |- BOOLEAN => VISIBLE;

-- translate the boolean 'membership' indicator into an event visibilty.


|- true()  => internal();
|- false() => external();


end HIDE;
```

```
set HIDE_FOLLOW is

-- used by      HIDE_EVENTS
--
-- taking       a gate identifier list, an event
--
-- for          calculating the resulting state of the state operator
--              'hide' under the execution of the given event.
--              Normally this is the state operator 'hide' followed by
--              the resulting state of the given event.
--              As an exception 'stop' and 'exit' do not offer events
--              to be hidden. If the following operator is also a 'hide'
--              the two 'hides' can be replaced by one 'hide' provided
--              with the union of both gate identifier lists.
--
-- returning    the event with the resulting state of the 'hide' operator.


judgement Gate_Identifier_List |- EVENT -> EVENT;

-- given a list of gates to be hidden, transform the event into
-- the event of the state operator 'hide'.


Gids |- event(EventName, Visible, follow_state(s_stop())) ->
        event(EventName, Visible, follow_state(s_stop()));

Gids |- event(EventName, Visible, follow_state(s_exit())) ->
        event(EventName, Visible, follow_state(s_exit()));

UNION_GIDS(Gids2 |- Gids -> Gids')
------------------------------------------------------------------------
Gids |- event(EventName, Visible, follow_state(s_hide(State, Gids2))) ->
        event(EventName, Visible, follow_state(s_hide(State, Gids')));

Gids |- event(EventName, Visible, follow_state(State)) ->
        event(EventName, Visible, follow_state(s_hide(State, gid_list(Gids))));
        provided diff(State, s_stop()) &
                 diff(State, s_exit()) &
                 not_hide(State);


set UNION_GIDS is

-- used by      HIDE_FOLLOW
--
-- taking       two gate identifier lists
--
-- for          calculating the gate identifier list that
--              is needed for the single 'hide' operator,
--              which substitutes two succeding 'hide' operators.
--
-- returning    the union of the given gate identifier lists.


judgement GIDS |- Gate_Identifier_List -> GIDS;

-- given a list of gates to be hidden, extend the gate identifier list
-- with the given list of gates to be hidden.


Gids |- gids[] -> Gids;

MEMBER_OF_GIDS(Gids |- event_name Gid : AlreadyHidden) &
Gids |- GidsRest -> GidsRest' &
ADD_GID(AlreadyHidden |- identifier Gid, GidsRest' -> Gids')
------------------------------------------------------------
Gids |- gids[identifier Gid . GidsRest] -> Gids';
```

```
set ADD_GID is

-- used by      UNION_GIDS
--
-- taking       a boolean indicator for the need of adding the gate identifier,
--              a gate identifier, a gate identifier list
--
-- for          updating the given gate identifier list with the
--              given gate identifier depending on the given boolean indicator.
--              If the given gate identifier already belongs to the given
--              gate identifier list it will not be added to the returned
--              gate identifier list.
--
-- returning    the gate identifier list with unique members including
--              the given gate identifier list and the given gate identifier.


judgement BOOLEAN |- Gate_Identifier, GIDS -> GIDS;

-- given a 'to be added' indicator, extend the gate identifier list
-- with the gate identifier, depending on the given indicator.


false() |- Gid, gid_list(Gids) -> gid_list(gids[Gid.Gids]);
true()  |- _, Gids -> Gids;

end ADD_GID;

end UNION_GIDS;

end HIDE_FOLLOW;

end HIDE_EVENTS;

end hide;
```

```
program relabel is

use             SLOTOS;
use             STATE;
use             MENU;

import          not_instance(S_NODE)
                from logic;

import          diff(_, _),
                prmess(string, string, integer)
                from prolog;


set MAKE_RELABEL is

-- used by      EXPAND in expand
--
-- taking       two optional gate identifier lists
--
-- for          building a relabel map from formal to actual
--              gate parameters. If there are no actual and formal
--              gate parameters, an empty relabel map will be build.
--
-- returning    the relabel map from formal to actual gate parameters
--              corresponding to the given gate identifier lists.


judgement |- Opt_Gate_List, Opt_Gate_List => RELABEL_MAP;

-- translate the optional gate parameter lists into a relabel map.


|- empty_gate_list(), empty_gate_list() => relabel_map[];

MAKE_GIDS_RELABEL(FormalGids, ActualGids => RelabelMap)
--------------------------------------------------------------
|- gate_list(FormalGids), gate_list(ActualGids) => RelabelMap;


set MAKE_GIDS_RELABEL is

-- used by      MAKE_RELABEL
--
-- taking       two gate identifier lists
--
-- for          building a relabel map from formal to actual
--              gate parameters. Each element of the formal gate
--              parameter list is assembled with the corresponding
--              element of the actual gate parameter list to the function:
--              formal_gate -> actual_gate. The list of these functions
--              forms the relabel map.
--              If the formal- and actual gate paramerter lists differ
--              in length an error message will be reported.
--
-- returning    the relabel map from formal to actual gate parameters.


judgement |- Gate_Identifier_List, Gate_Identifier_List => RELABEL_MAP;

-- translate the gate identifier lists into a relabel map.


|- gids[], gids[] => relabel_map[];

|- F_Rest, A_Rest => RelabelRest
--------------------------------------------------------------------------------
|- gids[identifier FormalGid . F_Rest], gids[identifier ActualGid . A_Rest] =>
   relabel_map[gid_relabel(gid FormalGid, gid ActualGid) . RelabelRest];

recovery
|- F_Gids, A_Gids => relabel_map[];
   do prmess("reports", "SLOTOS", 6);

end MAKE_GIDS_RELABEL;

end MAKE_RELABEL;
```

```
set RELABEL_MENU is

-- used by      CALC_MENU in menu
--
-- taking       a process identifier, a relabel map, a menu
--
-- for          relabeling the given menu with the given relabel map
--              in order to translate the given menu of a specification
--              process into the menu of the process instance corresponding
--              to the given process identifier.
--              The menu given might contain events resulting from
--              experiment offers with formal gate parameter names.
--              These has to be renamed into actual gate parameter names.
--              The process identifier is needed in order to create the
--              resulting state, that normally contains the process
--              the relabeling took place for.
--
-- returning    the menu of the process instance corresponding to the
--              given process identifier.


judgement PID, RELABEL_MAP |- MENU -> MENU;

-- given a process identifier and a relabel map, transform the menu
-- into the menu of the process instance corresponding to the given
-- process identifier and relabel map.


_, _ |- events[] -> events[];


RELABEL_EVENT(RelabelMap |- Event -> Event') &
RELABEL_FOLLOW(Pid, RelabelMap |- Event' -> Event'') &
Pid, RelabelMap |- MenuRest -> MenuRest'
----------------------------------------------------------------------------
Pid, RelabelMap |- events[Event . MenuRest] -> events[Event'' . MenuRest'];


set RELABEL_EVENT is

-- used by      RELABEL_MENU
--
-- taking       a relabel map, an event
--
-- for          relabeling the given event with the given relabel map.
--              Internal events are not relabeld.
--
-- returning    the relabeled event


judgement RELABEL_MAP |- EVENT -> EVENT;

-- given a relabel map, transform the event from a process in the
-- specification into the event of the process instance corresponding
-- to the given relabel map.


_ |- event(EventName, internal(), FollowState) ->
     event(EventName, internal(), FollowState);

RELABEL_GID(RelabelMap |- gid FormalGid -> gid ActualGid)
--------------------------------------------------------------------
RelabelMap |- event(event_name FormalGid, external(), FollowState) ->
              event(event_name ActualGid, external(), FollowState);

end RELABEL_EVENT;
```

```
set RELABEL_FOLLOW is

-- used by      RELABEL_MENU
--
-- taking       a process identifier, a relabel map, an event
--
-- for          calculating the resulting state of the given event.
--              A process instance with the resulting state "stop"
--              is eqvuivalent to the state "stop".
--              Two succeding processes can be reduced to the called process,
--              provided its relabel map is modified.
--              In any other case the resulting state of the returned
--              event is a process instance with the resulting state
--              of the given event.
--              Process instances with the resulting state "exit" are not
--              simplyfied to "exit" in order to keep the process instance
--              alive until it is asked to terminate.
--
-- returning    the event with the simplyfied resulting state


judgement PID, RELABEL_MAP |- EVENT -> EVENT;

-- given a process identifier and a relabel map, transform the resulting
-- state of the event into the resulting state of the process instance
-- corresponding to the given process identifier and relabel map.

_, _ |-
  event(EventName, Visible, follow_state(s_stop())) ->
  event(EventName, Visible, follow_state(s_stop()));

COMBINE_RELABELS(Relabel1 |- Relabel2 -> Relabel)
------------------------------------------------------------------------------
_, Relabel1 |-
  event(EventName, Visible, follow_state(s_instance(Snode, Pid, Relabel2))) ->
  event(EventName, Visible, follow_state(s_instance(Snode, Pid, Relabel)));

Pid, Relabel |-
  event(EventName, Visible, follow_state(State)) ->
  event(EventName, Visible, follow_state(s_instance(State, Pid, Relabel)));
  provided diff(State, s_stop()) &
           not_instance(State);


set COMBINE_RELABELS is

-- used by      RELABEL_FOLLOW
--
-- taking       two relabel maps
--
-- for          calculating one relabel map, that performs the
--              sequential two step relabeling of the given relabel maps
--              in one step.
--              The resulting map does not suit the most general case of
--              two-map to one-map reduction.
--              This is not neccessary since only the actual gate parameters
--              present in the map of the called process need to be relabeld
--              by the map of the calling process.
--              Other events cannot occur because the calling process is empty
--              except for the called process.
--
-- returning    one relabel map, that substitutes the given two relabel maps


judgement RELABEL_MAP |- RELABEL_MAP -> RELABEL_MAP;

-- given the relabel map of the calling process, transform the relabel map
-- of the called process into the relabel map that substitutes both
-- relabel maps.


_ |- relabel_map[] -> relabel_map[];

RELABEL_GID(Relabel |- ActualGid -> ActualGid') &
Relabel |- RelabelRest -> RelabelRest'
--------------------------------------------------------------------------
Relabel |- relabel_map[gid_relabel(FormalGid, ActualGid) . RelabelRest] ->
           relabel_map[gid_relabel(FormalGid, ActualGid') . RelabelRest'];


end COMBINE_RELABELS;

end RELABEL_FOLLOW;
```

```
set RELABEL_GID is

-- used by      RELABEL_EVENT
--              COMBINE_RELABELS
--
-- taking       a relabel map, a gate identifier
--
-- for          renaming the given gate identifier with the given relabel map
--              If the gate identifier is not in the domain of the relabel map,
--              that is, it was not an experiment offer from a formal
--              parameter gate, it will be returned unchanged.
--              Otherwise it is renamed to the corresponding actual parameter
--              gate.
--
-- returning    the given gate identifier, relabeld with the given relabel map.


judgement RELABEL_MAP |- GID -> GID;

-- given a relabel map, transform the formal gate identifier into
-- an actual gate identifier.


relabel_map[] |- Gid -> Gid;

relabel_map[gid_relabel(FormalGid, ActualGid)._] |- FormalGid -> ActualGid;

RelabelRest |- FormalGid -> ActualGid
--------------------------------------------------------------------------
relabel_map[gid_relabel(Gid, _) . RelabelRest] |- FormalGid -> ActualGid;
  provided diff(FormalGid, Gid);

end RELABEL_GID;

end RELABEL_MENU;

end relabel;
```

```
program member is

use             SLOTOS;
use             STATE;
use             MENU;

import          diff(_, _)
                from prolog;

define true:    () -> BOOLEAN;
define false:   () -> BOOLEAN;


set MEMBER_OF_EVENTS is

-- used by      INTERSECT in intersect
--
-- taking       a menu, an event
--
-- for          determing the membership of the given event in the given menu.
--              If false the result will be an empty menu.
--              If true the result will be the the menu containing the
--              given event succeded by the unsearched rest of the given menu.
--              As a result in addition to testing membership it is possible
--              to extract every single, of multiple occurences, of the given
--              event, by using the returned menu again for membership testing.
--
-- returning    the given menu, reduced by the front list not containing
--              the given event

judgement EVENTS |- EVENT : EVENTS;

-- given a menu, return the menu reduced by the front list not containing
-- the event.


events[] |- _ : events[];

events[event(EventName, Visible, FollowState).MenuRest] |-
  event(EventName, Visible, _) :
  events[event(EventName, Visible, FollowState).MenuRest];

MenuRest |- event(EventName, Visible, _) : Result
-------------------------------------------------
events[event(EventName', _, _).MenuRest] |-
  event(EventName, Visible, _) : Result;
  provided diff(EventName, EventName');

end MEMBER_OF_EVENTS;

set MEMBER_OF_GIDS is

-- used by      HIDE_EVENTS in hide
--              SPLIT_SYNCED_INTERLEAVED in split
--
-- taking       a gate identifier list, an event name
--
-- for          determing the membership of the given event name in
--              the given gate identifier list.
--              There is a special list "all_gids", which will yield
--              true for the membership of any event name.
--
-- returning    the boolean 'membership' indicator for the event name

judgement GIDS |- EVENT_NAME : BOOLEAN;

-- given a gate identifier list, return the membership of the event name
-- in the given gate identifier list.


all_gids() |- _ : true();

gid_list(gids[]) |- _ : false();

gid_list(gids[identifier EventName._]) |- event_name EventName : true();

gid_list(GLRest) |- event_name EventName : Result
-----------------------------------------------------------------------------
gid_list(gids[identifier EventName'.GLRest]) |- event_name EventName : Result;
  provided diff(EventName, EventName');

end MEMBER_OF_GIDS;

end member;
```

```
program annot_spec is

use             SLOTOS;
use             ENV;
use             PSP;

import          ADD_LOCAL_ENV(PENV |- Process_Definition_List => PENV),
                FROM_GLOBAL_ENV(PENV |- Process_Identifier -< PACCESS)
                from process_env;

import          PUT_PDEF(PACCESS @ PATH)
                from pdef_access;

import          diff(Process_Definition_List, Process_Definition_List),
                prmess(string, string, integer)
                from prolog;


set ANNOT_SPEC is

-- used by      INITIALIZE in state
--
-- taking       the original specification
--
-- for          providing access to the process_definition for each
--              process call in the specification.
--              This is done by annotating each process call node with
--              a process definition access node.
--              During the annotation of the specification a process
--              environment is build to enable access to the
--              process definitions in the correct scope.
--              No local process environment will be calculated for
--              specifications without process definitions, but in either
--              case the specification itself is supplied as a process.
--
-- returning    Side effect (annotation of specification).


judgement |- Specification;

-- annotate Lotos specification.


ADD_LOCAL_ENV(penv[] |-
              proc_defs[proc_def(proc_header(Sid, GateList, Func), Bexp, _)] =>
              Env) &
ANNOT_BEXP(Env |- Bexp)
--------------------------------------------------------------------------------
|- spec_def(spec_header(Sid, GateList, Func), Bexp, empty_local_def());

ADD_LOCAL_ENV(penv[] |-
              proc_defs[proc_def(proc_header(Sid, GateList, Func), Bexp, _)] =>
              Env) &
ADD_LOCAL_ENV(Env |- Pdefs => Env') &
ANNOT_PDEFS(Env' |- Pdefs) &
ANNOT_BEXP(Env' |- Bexp)
--------------------------------------------------------------------------------
|- spec_def(spec_header(Sid, GateList, Func), Bexp, local_def(Pdefs));
   provided prmess("reports", "SLOTOS", 2);
```

```
set ANNOT_BEXP is

-- used by      ANNOT_SPEC
--
-- taking       an process environment, a behaviour expression
--
-- for          annotating process calls with their process definitions
--              by navigating through the behaviour expression while
--              stopping at stop and exit,
--              annotating process calls and
--              skipping the other nodes.
--              The process environment is used to access the process
--              definition with the correct scope for a given process call.
--
-- returning    Side effect (annoted process calls).


judgement PENV |- SLOTOS;

-- given a process environment, annotate Lotos behaviour expression.


Env |- stop();

Env |- exit();

Env |- Bexp
----------------------
Env |- prefix(_, Bexp);

Env |- Bexp1 &
Env |- Bexp2
---------------------------
Env |- choice(Bexp1, Bexp2);

Env |- Bexp1 &
Env |- Bexp2
---------------------------
Env |- enable(Bexp1, Bexp2);

Env |- Bexp1 &
Env |- Bexp2
----------------------------
Env |- disable(Bexp1, Bexp2);

Env |- Bexp1 &
Env |- Bexp2
--------------------------------
Env |- synchronise(Bexp1, Bexp2);

Env |- Bexp1 &
Env |- Bexp2
-------------------------------
Env |- interleave(Bexp1, Bexp2);

Env |- Bexp1 &
Env |- Bexp2
------------------------
Env |- par(Bexp1, Bexp2);

Env |- Bexp1 &
Env |- Bexp2
--------------------------------
Env |- par_over(Bexp1, _, Bexp2);

Env |- Bexp
--------------------
Env |- hide(_, Bexp);

FROM_GLOBAL_ENV(Env |- Pid -< Paccess) &
PUT_PDEF(Paccess @ subject)
---------------------------------------
Env |- proc_call(Pid, _);

end ANNOT_BEXP;
```

```
set ANNOT_PDEFS is

-- used by      ANNOT_SPEC
--
-- taking       a process environment, a process definition list
--
-- for          annotating the process definitions, that served
--              to build the process environment, by
--              iterating over the list of process definitions.
--
-- returning    Side effect (annotated process definitions).


judgement PENV |- Process_Definition_List;

-- given a process environment, annotate the Lotos process definition list.


ANNOT_PDEF(Env |- Pdef)
-----------------------
Env |- proc_defs[Pdef];

ANNOT_PDEF(Env |- Pdef) &
Env |- PdefsRest
---------------------------------
Env |- proc_defs[Pdef.PdefsRest];
  provided diff(PdefsRest, proc_defs[]);


set ANNOT_PDEF is

-- used by      ANNOT_PDEFS
--
-- taking       a process environment, a process definition
--
-- for          annotating the process definition.
--              A process definition without local process definitions
--              is simply annotated.
--              Local process definitions are first completely added to
--              the process environment and then are annoted themselves.
--
-- returning    Side effect (annotated process definition).


judgement PENV |- Process_Definition;

-- given a process environment, annotate the Lotos process definition.


ANNOT_BEXP(Env |- Bexp)
--------------------------------------------
Env |- proc_def(_, Bexp, empty_local_def());

ADD_LOCAL_ENV(Env |- Pdefs => Env') &
ANNOT_PDEFS(Env' |- Pdefs) &
ANNOT_BEXP(Env' |- Bexp)
-------------------------------------------
Env |- proc_def(_, Bexp, local_def(Pdefs));

end ANNOT_PDEF;

end ANNOT_PDEFS;

end ANNOT_SPEC;

end annot_spec;
```

```
program process_env is

use             SLOTOS;
use             ENV;

import          diff(_, _),
                prmess(string, string, integer, string)
                from prolog;

define true:    () -> BOOLEAN;
define false:   () -> BOOLEAN;


set ADD_LOCAL_ENV is

-- used by      ANNOT_SPEC, ANNOT_PDEF in annot_spec
--
-- taking       a process environment, a process definition list
--
-- for          building a new environment, containing the given environment
--              plus the processes defined in the given
--              process definition list.
--              First a local environment is build from the given
--              process definition list and second it is added to the given
--              environment.
--
-- returning    the given environment extended with the given
--              process definition list.


judgement PENV |- Process_Definition_List => PENV;

-- given a global process environment, translate the process definition list
-- into the given global environment extended with the local process
-- environment, containing the processes in the process definition list.


MAKE_LOCAL_ENV(pmap[] |- Pdefs => Map)
--------------------------------------
Env |- Pdefs => penv[Map.Env];


set MAKE_LOCAL_ENV is

-- used by      ADD_LOCAL_ENV
--
-- taking       a process definition list, a local environment
--
-- for          building a local environment containing the process
--              defined in the process definition list.
--              An empty local environment has to be supplied as
--              a starting environment which is extended by
--              iterating over the processes in the process definition list.
--
-- returning    the local environment containing the processes defined
--              in the given process definition list.


judgement PMAP |- Process_Definition_List => PMAP;

-- given a local process environment, translate the process definition list
-- into the given local process environment extended with the processes
-- defined in the process definition list.


ADD_PDEF(Map |- Pdef => Map')
-------------------------------
Map |- proc_defs[Pdef] => Map';

ADD_PDEF(Map |- Pdef => Map') &
Map' |- PdefRest => Map''
-----------------------------------------
Map |- proc_defs[Pdef.PdefRest] => Map'';
```

```
set ADD_PDEF is

-- used by      MAKE_LOCAL_ENV
--
-- taking       a local environment, a process definition
--
-- for          adding the given process definition to the given
--              local environment.
--              In order to retrieve the process definition annotations
--              within the behaviour expressions of the process definitions
--              a process definition access is added to the local environment.
--              In addition to the process definition the process definition
--              access contains the location the process definition in
--              the original specification.
--              There is no need to store optional local process definitions.
--              Therefore each process definition is stored with
--              'empty_local_def'initions.
--
-- returning    the given local environment extended by the
--              given process definition.


judgement PMAP |- Process_Definition => PMAP;

-- given a local process environment, translate the  process definition into
-- the given local process environment extended with the process definition.


Map |-
  proc_def(proc_header(identifier Pid, GateList, Func), Bexp, _) =>
  pmap[paccess(pdef(proc_def(proc_header(identifier Pid, GateList, Func),
                             Bexp, empty_local_def())),
               ppath(subject)
              ) . Map];

end ADD_PDEF;

end MAKE_LOCAL_ENV;

end ADD_LOCAL_ENV;


set FROM_GLOBAL_ENV is

-- used by      ANNOT_BEXP in annot_spec
--
-- taking       an global environment, a process identifier
--
-- for          retrieving the process definition of the given
--              process identifier with the correct scope.
--              First the local environment within the given global
--              environment is searched for the process identifier.
--              If it does not contain the process identifier
--              the surrounding global environment will be searched.
--              Undeclared processes will cause an error message.
--
-- returning    the process definition access


judgement PENV |- Process_Identifier -< PACCESS;

-- given a global process environment, retrieve the process definition access
-- corresponding to the process identifier.


penv[] |- identifier Pid -< _; do prmess("reports", "SLOTOS", 4, Pid);

FROM_LOCAL_ENV(LocalEnv |- Pid -< Paccess : Found) &
SEARCH(GlobalEnv |- Pid, Paccess  - Found ->  Paccess')
-------------------------------------------------------
penv[LocalEnv.GlobalEnv] |- Pid -< Paccess';
```

```
set SEARCH is

-- used by      FROM_GLOBAL_ENV
--
-- taking       a global environment, a process identifier,
--              a boolean indicator for the need of further search,
--              a process definition access.
--
-- for          yielding the process definition of the given
--              process identifieraccess with the correct scope.
--              If the given process definition access has been found in
--              the local environment it will be returned.
--              Otherwise the given global surrounding environment is
--              searched for the process definition access with the
--              correct scope.
--
-- returning    the process definition access of the given process identifier
--              with the correct scope.


judgement PENV |- Process_Identifier, PACCESS  - BOOLEAN ->  PACCESS;

-- given a global process environment, depending on the boolean
-- 'found' indicator return the process definition accesss or
-- retrieve the process definition access corresponding to
-- the process identifier.


_ |- _, Paccess  - true() ->  Paccess;

FROM_GLOBAL_ENV(Env |- Pid -< Paccess)
--------------------------------------
Env |- Pid, _  - false() ->  Paccess;

end SEARCH;

set FROM_LOCAL_ENV is

-- used by      FROM_GLOBAL_ENV
--
-- taking       a local environment, a process identifier
--
-- for          yielding the process definition access of the given
--              process identifier with the correct scope.
--              The local environment is searched for an process definition
--              with the given process identifier.
--              If the process definition is found the process definition
--              access will be returned and the boolean indicator will be true.
--              Otherwise an undefined process definition access will be
--              returned and the boolean indicator will be false.
--
-- returning    the process definition access of the given process identifier,
--              a 'found/not_found' indicator.


judgement PMAP |- Process_Identifier -<  PACCESS : BOOLEAN;

-- given a local process environment, retrieve the process definition access
-- corresponding to the process identifier and indicate the success of
-- this operation with the boolean 'found' indicator.


pmap[] |- _ -< _ : false();

pmap[paccess(pdef(proc_def(proc_header(Pid, GateList, Func),
                           Bexp, empty_local_def())),
             PdefPath
            )._] |-
  Pid -< paccess(pdef(proc_def(proc_header(Pid, GateList, Func),
                               Bexp, empty_local_def())),
                 PdefPath
                ) : true();

MapRest |- Pid -< Paccess : Result
-------------------------------------------------------------------------
pmap[paccess(pdef(proc_def(proc_header(Pid', _, _), _, _)), _).MapRest] |-
  Pid -< Paccess : Result;
  provided diff(Pid', Pid);

end FROM_LOCAL_ENV;

end FROM_GLOBAL_ENV;

end process_env;
```

```
program pdef_access is

use             SLOTOS;
use             ENV;
use             PSP;

import          sendannot(path, string, PACCESS),
                getannot(path, string, PACCESS)
                from prolog;


set PUT_PDEF is

-- used by      ANNOT_BEXP in annot_spec
--
-- taking       a process definition access, a location
--
-- for          annotating the process call at the given location
--              with the given process definition access.
--
-- returning    Side effect (annoted process call).


judgement |- PACCESS @ PATH;

-- Store the process definition access at the location of the process call.


|- Pacces @ PcallPath;
   do sendannot(PcallPath, "pdef", Pacces);

end PUT_PDEF;


set GET_PDEF is

-- used by      EXPAND in expand
--
-- taking       a process call
--
-- for          retrieving the process definition access, that was
--              previously stored in the process call.
--
-- returning    the formal gate identifiers,
--              the process body, a behaviour expression.
--              the location of the process body,
--              all of the given process definition.
--              The process body is the second son of a process definition.

judgement |- Behaviour_Expression -<
             Opt_Gate_List, Behaviour_Expression @ PATH;

-- retrieve the optional gate parameter list and process body together with
-- the location of the process body from the process call.


|- proc_call(_, _) -< F_Gids, Bexp @ s(PdefPath, int 2);
   provided getannot(subject, "pdef",
                     paccess(pdef(proc_def(proc_header(_, F_Gids, _), Bexp,_)),
                             ppath(PdefPath)
                            )
                    );

end GET_PDEF;

end pdef_access;
```

Ich erkläre an Eides statt, die vorliegende Arbeit selbständig und unter ausschließlicher Zuhilfenahme der angegebenen Literatur erstellt zu haben.

Darmstadt, 29. September 1992