KfK 5279
Januar 1994

Algebraische Spezifikation und Modellierung in Höheren Petri-Netzen mit zustandsabhängiger Schaltregel

C. Düpmeier
Institut für Angewandte Informatik

Kernforschungszentrum Karlsruhe

KERNFORSCHUNGSZENTRUM KARLSRUHE

Institut für Angewandte Informatik

KfK 5279

Algebraische Spezifikation und Modellierung in Höheren Petri-Netzen mit zustandsabhängiger Schaltregel*)

Clemens Düpmeier

*) vom Fachbereich 4: Informatik der Universität Koblenz-Landau genehmigte Dissertation

Kernforschungszentrum Karlsruhe GmbH, Karlsruhe

Algebraische Spezifikation und Modellierung in Höheren Petri-Netzen mit zustandsabhängiger Schaltregel

Zusammenfassung

Zur Spezifikation und Modellierung komplexer Softwaresysteme benötigt man formale Verfahren, die sowohl komplexe sequentielle Aktionen (Funktionen auf Objekten) als auch komplexe parallele Aktionen formal beschreibbar machen. Dabei sollte ein Anwender solcher Methoden durch ein computergestütztes Werkzeug unterstützt werden, das ihm neben der reinen Eingabe auch die automatische Analyse von Modellen erlaubt. Zur Beschreibung komplexer sequentieller Aktionen auf Datenstrukturen kann man die Methode der algebraischen Spezifikation abstrakter Datentypen verwenden. Komplexe parallele Aktionen lassen sich mit der Petri-Netz-Theorie beschreiben. Unser Spezifikations- und Modellierungsansatz und das zugehörige Werkzeug SMARAGD (Specification, Modelling and Reachability Analysis Graphical Development System) basieren daher auf einer Kombination von algebraischen Spezifikationsmethoden und der Petri-Netz-Theorie.

Als Netz-Modell zur Modellierung komplexer verteilter Softwaresysteme erschien uns dabei ein Höheres Petri-Netz-Modell (z.B. PrT- oder Colored Petri-Netze) sinnvoll, wobei es sich bei der Werkzeugkonzeption zeigte, daß vorhandene Höhere Petri-Netz-Modelle nur unzureichend die Eigenschaften aufwiesen, die für eine Integration von algebraischer Spezifikation und Petri-Netz-Theorie mit dem Ziel einer computergestützten Analyse nötig sind. Daher wird in dieser Arbeit ein Petri-Netz-Modell vorgestellt, das einerseits die Allgemeinheit vorhandener Ansätze zur Kombination von algebraischer Spezifikation und Petri-Netz-Theorie erhält, andererseits aber durch eine neuartige Interpretation der Beschriftung des Netz-Modells eine effiziente computergestützte Analyse ermöglicht.

Die formale Beschreibung des Netz-Modells erfolgt zunächst durch die Definition der Sprache SNL (SMARAGD Net Language) und diese wird dann anschließend auf eine durch graphische Konstrukte unterstützte Beschreibungsform abgebildet, die eine weitgehend natürliche Eingabe der Netz-Modelle über einen graphischen Editor erlaubt, wie er dann auch im Werkzeug SMARAGD implementiert wurde.

Nach der Einleitung wird zunächst die rein lexikalische Struktur der Sprache SNL vorgestellt. Im Anschluß daran wird der Spezifikationsteil der Sprache definiert, wobei die Einführung einzelner Komponenten der Sprache durch theoretische Definitionen und Ergebnisse der Spezifikationstheorie begründet werden.

Hierauf wird die Petri-Netz-Seite der Sprache vorgestellt und theoretisch begründet. Die Verknüpfung der Spezifikationstheorie und Petri-Netz-Theorie erfolgt dabei dadurch, daß die Anschriften eines Petri-Netzes zunächst rein algebraisch spezifiziert werden. Dies führt zu dem Begriff einer SNL-Netz-Spezifikation. Eine Realisierung so einer SNL-Netz-Spezifikation führt dann zu einem ausführbaren SNL-Modell. SNL-Modelle lassen sich dann vielfältigen Analysen unterziehen.

Die durch graphische Konstrukte unterstützte Beschreibungsform des Netz-Modells und die Konzeption des SMARAGD-Editors ergibt sich im Anschluß an die Sprachdefinition in natürlicher Weise, wobei an dieser Stelle auf die Oberfläche und einige Implementierungsgesichtspunkte ebenfalls eingegangen wird. Zum Abschluß ziehen wir einen Vergleich von SMARAGD mit anderen Höheren Petri-Netz-Werkzeugen und geben schließlich ein Resumee der bisherigen Arbeiten und zeigen zukünftige Entwicklungen auf.

Algebraic Specification and Modelling in High Level Petri Nets with a State Dependent Firing Rule

Abstract

For the specification and modelling of complex distributed software systems are formal methods needed, which allow to describe both the complex sequential (i.e. functions on objects) and the complex parallel behaviour of the systems. Users should be guided in the application of such methods by computer tools, which beside the pure description tasks should also allow automatic analysis of the interesting behaviour of the specified systems. For the formal description of sequential behaviour of data structures one can use an abstract data type specification language. Complex parallel behaviour can be best described in the intuitive graphical formalism called petri nets. Our specification and modelling approach and the SMARAGD System (Specification, Modelling and Reachability Analysis Graphical Development System) are therefore based on a combination of abstract data type specification and petri net theory.

Of the many different flavours of petri nets seemed for us an high level net formalism (like Predicate/Transition-Nets (PrT-Nets) or Colored Petri Nets) most suitable to model complex distributed systems but it showed up early in the design phase of the SMARAGD tool that such existing high level net models are for sure theoretically interesting but lack nearly all the capabilities needed for an integration of algebraic specification and petri net theory with the goal of an efficient computer based simulation and analysis of the constructed models.This thesis presents therefore an high level petri net model with a new state dependent firing rule, which preserves the theoretical interesting features of other high level net types but gives us also a much more deterministic algorithm to handle the firing rule which is needed for efficient simulation and analysis of net models.

The formal description of our net model is first done by defining a context free language which we call SNL (SMARAGD Net Language). This language is later translated into a second description which contains graphical elements to define the net semantic as usual in terms of boxes, ellipses, arcs and their special text annotations. The latter form is then used by system designers to construct their system models with the graphical editor of the SMARAGD system.

After the introduction we will first define the lexical structure of SNL. Then we will present the abstract data type specification part of the language. Each new component of the language will be motivated here with some definitions and results from specification theory.

The main part of the thesis is then the definition of the petri net part of the language within a sound an complete theory. The abstract data type specification and petri net theory is tied together through special net inscriptions which are based on terms and formulas of the first order abstract data type calculus. This leeds to the notion of a SNL Net Specification. This syntactic construct is then transformed by an interpretation of the data type specifications within algebraic systems and with our new state dependent firing rule into an in the SMARAGD environment executable system model, which we call SNL system (or SNL model). SNL models can then easily be simulated and analysed within the SMARAGD environment.

As mentioned above, the context free language SNL gets at the end of this thesis translated into a more graphical description method, which gives users a more natural view of their model in terms of diagrams containing boxes describing actions, ellipses to describe data states and directed arcs to show the data flow of a system. The presentation of this graphical formalism is embedded into the discussion of the graphical editor of the SMARAGD system, its feature and its implementation.

Inhaltsverzeichnis

1. Einleitung 1
2. Lexikalische Bauelemente der Spezifikations- und Modellierungssprache SNL 5
3. Spezifikation abstrakter Datentypen mit SNL 7
   - 3.1 Grundsyntax zur Spezifikation abstrakter Datentypen in SNL 7
     - 3.1.1 Der Begriff der Signatur 8
     - 3.1.2 Spezifikationen und Signaturen 13
     - 3.1.3 Formeln 20
       - 3.1.3.1 Variablen und Terme 21
       - 3.1.3.2 Atomare Formeln 23
       - 3.1.3.3 Allgemeine Formeln 24
   - 3.2 Semantik des Spezifikationsteils von SNL 27
     - 3.2.1 Realisation von Termen 28
     - 3.2.2 Realisation von Formeln 29
   - 3.3 Erweiterungen des Spezifikationsteils von SNL 34
     - 3.3.1 Alternative Schreibweisen und Abkürzungen 34
     - 3.3.2 Partielle Operatoren und Prädikate 35
     - 3.3.3 Parametrisierte Spezifikationen 37
     - 3.3.4 Standardtypen in SNL 42
       - 3.3.4.1 Der Standarddatentyp BOOL 42
       - 3.3.4.2 Der Standarddatentyp INT 43
       - 3.3.4.3 Der Standarddatentyp STRING 44
     - 3.3.5 Konstruktion von Tupeln, Rekords, Unionen und Listen von Spezifikationen 44
   - 3.4 Realisation von SNL-Spezifikationen in ML 53
     - 3.4.1 SNL-Signaturen und ML-Signaturen 53
     - 3.4.2 SNL-Spezifikationen und ML-Strukturen 56
4. Das Petri-Netz-Modell 63
   - 4.1 Formale Summen und Multimengen 63
   - 4.2 SNL-Netz-Spezifikationen 66
   - 4.3 SNL-Systeme (SNL-Modelle) 75
   - 4.4 Das Schalten von SNL-Systemen 77
     - 4.4.1 Ein kurzer Exkurs in die lineare Algebra 77
     - 4.4.2 Lineare Algebra von SNL-Systemen 81
   - 4.5 Modularisierung 88
     - 4.5.1 Netz-Klassen und Netz-Objekte 88
     - 4.5.2 Netz-Module 96
5. Ein graphischer Editor für SNL-Modelle 103
   - 5.1 Benutzerinterface des Editors 103
     - 5.1.1 Dialogfenster und ihre Elemente 104
     - 5.1.2 Seiten- und Diagramm-Objekte 106
     - 5.1.3 Elementare Netzelemente im Editor 109
       - 5.1.3.1 Deklarationsboxen 109
       - 5.1.3.2 Stellen, Transitionen und Kanten im Editor 113
       - 5.1.3.3 Beschriftung von Stellen, Transitionen und Kanten 114
     - 5.1.4 Hierarchisierungselemente im Editor 120
       - 5.1.4.1 Erzeugung von Portknoten 120

5.1.4.2 Erzeugung von Substitutionsknoten 123
5.1.4.3 Zuweisung von Socketknoten zu Portknoten 124
5.1.4.4 Auslagern von Teilnetzen 124
5.1.4.5 Ersetzen eines Substitutionsknotens durch ein Teilnetz 125
5.1.5 Die Menueleiste des Editors 125
6. Vergleich mit anderen Petri-Netz-Modellen und -Werkzeugen 133
7. Fazit und Ausblick 139
Literatur 143

1. Einleitung

In den letzten Jahren führte der Trend in der Softwareentwicklung zu immer komplexeren Anwendungen in Bereichen, die ein hohes Maß an Fehlerfreiheit verlangen. Dabei hat die Erfahrung gezeigt, daß computergestützte Werkzeuge, die den Entwickler in allen Phasen des Entwicklungsprozesses unterstützen, zur Sicherung der Qualität der Software unerläßlich sind.

Werkzeuge der ersten Generation unterstützten den Entwickler vorwiegend während der Codierungsphase. Moderne Sprachen mit strengen Typenkonzepten und deren Compiler erlauben heutzutage die Detektion der meisten Syntaxfehler bereits während der Übersetzung eines Programms. Andere Werkzeuge ermöglichen ein intensives Testen einer Anwendung vor dem eigentlichen Einsatz, um weitere erst während der Laufzeit bestimmbare Fehler aufzudecken.

Wenn man mit solchen Werkzeugen auch eine große Anzahl von Fehlern während der Codierungsphase verhindern oder detektieren kann, bieten sie in Bezug auf Fehler beim Design der Software kaum oder viel zu späte Hilfe. Aus diesem Grunde wurden eine Reihe von Methoden, wie SADT, entwickelt, die speziell die Designphase im Softwareentwicklungsprozeß unterstützen sollen. Diese Verfahren basieren aber in der Regel nur auf informellen Methoden, so daß eine computergestützte Verifikation von Eigenschaften, die einer zu entwickelnden Anwendung in der Designphase zugeordnet werden, nicht oder nur wenig möglich ist.

Aus diesem Grunde konzentriert sich ein Teil der Forschung im Softwareengineeringbereich zur Zeit verstärkt auf die Entwicklung formaler Methoden zur Unterstützung der Designphase. Dabei finden algebraische Ansätze zur Spezifikation von Software große Beachtung (siehe z.B. [Ehrig85b, Ehrig90, Partsch90]). Diese erlauben es, die Eigenschaften eines zu entwickelnden Softwaresystems zunächst einmal in einem algebraisch-logischen Kalkül auf mathematisch exakte Weise zu formalisieren. Werkzeuge sollen dann eine Umsetzung der formalen Spezifikation in die betreffende Anwendung so unterstützen, daß alle Eigenschaften des Systems, die in der Spezifikation festgelegt wurden, auch in der Anwendung garantiert werden. Ein wichtiges Teilgebiet im Rahmen der algebraischen Spezifikation ist die algebraische Spezifikation abstrakter Datentypen. Sie beschäftigt sich mit der Spezifikation von Datentypen, wobei ein Datentyp als eine Menge von Datenbereichen (Wertemengen für Operationen) und auf diesen Datenbereichen definierten Operationen aufgefaßt wird (siehe z.B. [Goguen75, Ehrig82, Ehrig85a]).

Algebraische Spezifikationen liefern eine gute Spezifikationsmethode für die funktionalen Eigenschaften einer Software. Sie liefern jedoch keinerlei Hilfsmittel zur Formalisierung und Verifikation von Eigenschaften, die mit der Parallelität eines Anwendungssystems zusammenhängen. Dieser Aspekt gewinnt aber im Zeitalter der verteilten Systeme zunehmend an Bedeutung. Zur Untersuchung des parallelen Verhaltens von nebenläufigen Systemen verwendet man häufig Petri-Netze. Da Petri-Netze keine Mechanismen zur Spezifikation der Funktionalität einer Anwendung bereitstellen, liegt eine Kombination von algebraischen Spezifikationsmethoden und Petri-Netz-Methoden nahe. Die Durchführbarkeit eines solchen Ansatzes wurde schon in [Berthomieu86, Vautherin87, Reisig91, Schmidt89] gezeigt.

In dieser Arbeit wird eine Spezifikations- und Modellierungssprache SNL (SMARAGD Net Language) vorgestellt, die auf dieser Kombination von Methoden zur Spezifikation abstrakter Datentypen und der Theorie von Petri-Netzen basiert. Die Sprache SNL bildet die Grundlage für das rechnergestützte Werkzeug SMARAGD (Specification, Modelling And Reachability Analysis Graphical Development-System), das seit einiger Zeit im Institut für Datenverarbeitung in der Technik (IDT) des Kernforschungszentrums Karlsruhe (KfK) in der Abteilung Mikrosystem-Informatik zur Modellierung komplexer Softwaresysteme entwickelt wird. Eine weitere Arbeit von W. Süß (siehe [Süß92]) behandelt die Analyse von Modellen in unserer Spezifikations- und Modellierungssprache.

Im Kapitel 2 führen wir zunächst die lexikalischen Bauelemente der Sprache ein. Hier werden die Schlüsselwörter der Sprache und andere Token der Sprache festgelegt.

Im Kapitel 3 definieren wir dann den Spezifikationsteil der Sprache. Hier wird zunächst der Begriff der Signatur definiert. Diese legt für einen abstrakten Datentyp im Sinne einer Schnittstellenbeschreibung die Namen verwendeter Wertebereiche (*Sorten*) für Operationen und Relationen und die Namen von Operationen und Relationen auf diesen Wertebereichen unter Angabe der Argument- und Rückgabewertebereiche fest.

Danach wird der Begriff der Formel eingeführt. Formeln sind innerhalb unserer Definition abstrakter Datentypen geschlossene Formeln einer formalen Sprache 1. Ordnung. Ihre Bauelemente sind zunächst Terme über der Signatur des abstrakten Datentyps bestehend aus Konstanten, Variablen, Operationssymbolen und Klammern als Hilfsgrößen. Aus Termen und Prädikaten kann man dann elementare Formeln bilden. Diese lassen sich wiederum mit logischen Konnektoren der formalen Sprache (wie **and**, **or** und **not**) zu einfachen Formeln der Sprache zusammenbinden. Die Einbeziehung von Quantoren führt schließlich zu allgemeinen geschlossenen Formeln der formalen Sprache 1. Ordnung.

Die Spezifikation eines abstrakten Datentyps besteht dann aus der Signatur des abstrakten Datentyps und einer Menge von Formeln, die die Semantik des abstrakten Datentyps festlegen. Modelle einer Spezifikation sind algebraische Systeme zur Signatur des abstrakten Datentyps, die alle Formeln erfüllen. Ein abstrakter Datentyp ist schließlich die Äquivalenzklasse aller Modelle der Spezifikation des abstrakten Datentyps.

Unsere Spezifikationssprache enthält neben den Basiskonstrukten, die auf Grund der Theorie unbedingt erforderlich sind, Konstrukte zur Modularisierung von abstrakten Datentypspezifikationen. Ein Vererbungsmechanismus erlaubt den hierarchischen Aufbau von Spezifikationen auf der Grundlage bereits vorhandener Spezifikationen. Der Begriff einer parametrisierten Spezifikation (Funktor) erlaubt es einem Benutzer, Spezifikationen mit formalen Parametern zu definieren, aus denen durch Konkretisierung der Parameter Elemente ganzer Klassen von abstrakten Datentypen gewonnen werden können.

Auf der Grundlage eingebauter Standardspezifikationen, wie BOOL, INT und STRING wird weiter der konstruktive Entwurf von Spezifikationen über Tupel-, Rekord-, Unions- und Listenkonstruktionsmechanismen innerhalb der Spezifikationssprache unterstützt.

Im Kapitel 4 wird der Petri-Netz-Teil der Sprache vorgestellt. Als Netz-Modell erschien uns dabei das PrT-Netz-Modell nach Genrich/Lautenbach (siehe [Genrich80a, Genrich86, Düpmeier90]) sinnvoll, wobei es sich bei der Werkzeugkonzeption zeigte, daß zum Bau eines computergestützten Werkzeuges einige wichtige Änderungen der klassischen PrT-Netz Definition nach Genrich und Lautenbach notwendig werden, damit zum einen eine sinnvolle Kopplung zwischen Spezifikation und Petri-Netz-Modell und zum anderen eine effiziente computergestützte Analyse möglich wird.

Die Verknüpfung der Spezifikationstheorie mit der Petri-Netz-Theorie erfolgt dadurch, daß die Anschriften im Petri-Netz-Modell aus Elementen der Signatur der gegebenen algebraischen Spezifikationen zusammengesetzt werden. Dabei erhält man zunächst nur eine rein syntaktische Beschreibung des Netzes, die wir SNL-Netz-Spezifikation nennen. Eine Realisierung der algebraischen Spezifikation durch ein zugehöriges Modell führt dann zu einem ausführbaren SNL-Modell, das wir auch SNL-System nennen.

Unser Begriff der Realisierung einer SNL-Netz-Spezifikation weicht in der Hinsicht vom üblichen Begriff ab, daß die Auswertung von Formeln in der Netz-Beschriftung von der aktuellen Markierung des betreffenden SNL-Systems abhängt. Dies liegt daran, daß wir Quantoren in Formeln der Netzbeschriftung nicht über die Wertebereichsmengen im zugehörigen Modell des algebraischen Systems, sondern nur über die Objekte, die in der Umgebung einer betreffenden Stelle oder Transition als Marken auf den Umgebungsstellen liegen, auswerten.

Neben den Grundelementen zur Definition von SNL-Netz-Spezifikationen und zugehörigen SNL-Systemen bietet unsere Sprache weiter zwei Konzepte zur Modularisierung von Netzen, die an die Begriffe der Substitutionsstellen und Substitutionstransitionen in [Huber89] angelehnt sind. Dies sind die Konzepte Netz-Klasse und Netz-Modul.

Eine Netz-Klasse ist ein Modul-Konzept im Sinne einer Substitutionsstelle, das an den Begriff der Klasse in objektorientierten Sprachen angelehnt ist. Eine Netz-Klasse stellt einem Benutzer eine Menge von Port-Transitionen bereit, die wie Methoden einer Klasse im Sinne einer objektorientierten Sprache verwendet werden können, um Operationen auf intern in der Klasse verborgenen Datenstrukturen durchzuführen.

Ein Netz-Modul kann als eine Black-Box betrachtet werden, die definierte Eingabeobjekte über Eingangsstellen entgegennimmt und dann definierte Ausgabeobjekte über Ausgangsstellen an den Benutzer des Moduls zurückgibt.

Wir erweitern in unserer Arbeit die Konzepte von Jensen, indem wir eine Parametrisierung von Netz-Klassen und Netz-Modulen sowie der zugehörigen Port-Transitionen und Port-Stellen erlauben, die es uns analog zum Begriff der parametrisierten Spezifikation gestattet, ganze Klassen von Teilnetzen durch eine Netz-Klasse oder einen Netz-Modul zu beschreiben.

Im Kapitel 5 bilden wir die Netzsprache SNL auf eine graphische Beschreibungsform ab und stellen das Konzept des SMARAGD-Editors vor, der es einem erlaubt, SNL-Spezifikationen und -Modelle über eine graphische Benutzeroberfläche einzugeben. Neben der Konzeption der Benutzeroberfläche des Editors enthält das Kapitel 5 die Beschreibung einiger wichtiger Implementierungsaspekte des Editors.

Kapitel 6 vergleicht das SNL-Modell mit anderen wichtigen Höheren Petri-Netz-Modellen. Dabei werden wir vor allem auch die Gesichtspunkte von SNL herausstellen, die diese Sprache zur Spezifikation und Modellierung komplexer Softwaresysteme auszeichnen. Weiter werden wir einen Vergleich von SMARAGD mit anderen Höheren Petri-Netz-Werkzeugen vornehmen.

Kapitel 7 gibt ein kurzes Fazit zur Sprachdefinition und zeigt, welche Arbeiten im Hinblick auf das Werkzeug SMARAGD noch nötig sind, damit wir ein Werkzeug erhalten, das über eine prototypische Realisierung hinaus in realen Softwareentwicklungen eingesetzt werden kann.

2. Lexikalische Bauelemente der Spezifikations- und Modellierungssprache SNL

Die in den folgenden Kapiteln beschriebene Petri-Netz-Sprache SNL (SMARAGD-Net-Language) verwendet einige lexikalische Grundbauelemente, die wir im folgenden kurz beschreiben wollen.

Wir wollen Namen, die innerhalb von SNL auftreten, in vier Gattungen unterscheiden: reservierte Schlüsselwörter, Namen vom Tokentyp **id**, Namen vom Tokentyp **con** und Namen vom Tokentyp **symbol**. Schlüsselwörter dürfen nur innerhalb der ihnen durch die Sprachdefinition zugewiesenen Semantik verwendet werden. Sie stehen damit nicht für den Benutzer als frei wählbare Namen zur Verfügung. Die Schlüsselwörter von SNL sind:

> **all, andalso, arc, bvar, class, const, def, end, eqn, exist, for, functor, fvar, if, in, include, infix, infixr, list, marking, module, net, netclass, netmodule, not, of, opn, orelse, out, param, petrinet, place, placedef, port, pred, ptl, return, sig, signature, sort, spec, specification, then, trans, transition, union, var** und **with.**

Da wir SNL-Spezifikationen in der Sprache ML realisieren werden, sind die folgenden Schlüsselwörter von ML ebenfalls reserviert.

> **abstraction, abstype, and, as, case, do, datatype, exception, fn, fun, handle, let, local, nonfix, of, open, raise, rec, sharing, struct, structure, type, val** und **while.**

Neben den reservierten Schlüsselwörtern gibt es noch eine Reihe von Sonderzeichen und Sonderzeichenfolgen, die reserviert sind. Diese sind:

> **( ) [ ] , { } : ; ` . ... | = => <=> -> <- <-> _ ? <> ***

Das Gleichheitszeichen nimmt dabei eine Sonderstellung ein, da wir es erlauben, daß = als benutzerdefinierter Name für das Gleichheitsprädikat beliebiger Spezifikationen verwendet werden darf und muß.

Benutzerdefinierte Namen vom Tokentyp **id** werden durch den regulären Ausdruck

[a-zA-Z][a-zA-Z0-9'_]*

beschrieben. Sie beginnen stets mit einem Buchstaben. Im Anschluß daran folgt eine beliebige Folge von Zeichen bestehend aus Buchstaben und/oder Zahlen und/oder den Sonderzeichen ',_. Wir wollen hierzu ein paar Beispiele betrachten.

Beispiel 1 **(Namen vom Typ id)**

(a) Einfache Namen sind z.B. *x, y, x1, y2* als Variablennamen und *insert, push, pop* als Operatornamen.

(b) Bei komplizierteren Netz-Spezifikationen sind zusammengesetzte Namen der Gestalt *fifo_insert* oder *natstack'push* sinnvoll.

(c) Voranstellen von Ziffern (z.B. *3inst*) ist bei Namen vom Tokentyp **id** nicht erlaubt.

Eine Längenbegrenzung der Namen gibt es auf Grund der Spezifikation nicht. Sie kann aber implementierungsabhängig eingeführt werden.

Benutzerdefinierte Namen vom Tokentyp **con** (Konstanten) sind entweder benutzerdefinierte Namen vom Tokentyp **id** oder numerische Konstanten vom Typ **num.**

Unter numerischen Konstanten verstehen wir dabei Zahlenfolgen, die natürliche Zahlen bezeichnen und durch den regulären Ausdruck

num: [0-9] [0-9]*

beschrieben werden. Unter Verwendung von **id** und **num** können wir nun **con** durch

con: **id | num**

beschreiben, d.h. ein Token vom Typ **con** (Konstante) ist entweder ein benutzerdefinierter Name vom Typ **id** oder eine numerische Konstante vom Typ **num.**

Wir verwenden numerische Konstanten in Situationen, in denen wir natürliche Zahlen als Angabe der Anzahl von Objekten unserer Sprache angeben müssen. Token vom Typ **con** verwenden wir dagegen als Konstantensymbole innerhalb von SNL-Spezifikationen. Im Fall der Spezifikation von numerischen Datentypen machen hier durchaus numerische Konstanten als Namen für Konstanten Sinn. Betrachten wir hierzu ein paar Beispiele.

Beispiel 2 (Namen vom Typ con und numerische Konstanten)

(a) Token, wie sie typischerweise für numerische Konstanten verwendet werden, sind z.B. *234*, *175* oder *0*. Diese können auch als Namen vom Tokentyp **con** aufgefaßt werden.

(b) Als Namen vom Tokentyp **con** sind ebenfalls *true* oder *empty_stack* erlaubt.

Weiter erlauben wir benutzerdefinierte Namen, die durch Folgen von Sonderzeichen beschrieben werden. Diese sind vom Tokentyp **symbol** und über den folgenden regulären Ausdruck definiert:

symbol: [!%&$+-/:<=>?@\~`^|*] [!%&$+-/:<=>?@\~`^|*]*

Dabei sind wiederum die reservierten Sonderzeichenfolgen ausgeschlossen. Wir wollen einmal ein paar Beispiele für Namen vom Tokentyp **symbol** zeigen.

Beispiel 3 (Token vom Typ symbol)

(a) *??*, |=|, &&, ||, != sind Token vom Typ **symbol**

(b) =>, <=>, ->, und : sind keine Token vom Typ **symbol**, da sie reserviert sind.

Eine Reihe weiterer Wörter und Symbolfolgen sind nicht unbedingt reserviert aber innerhalb von Standardspezifikationen und Konstruktionsmechanismen, die SMARAGD bereitstellt, vordefiniert, und sollten deshalb nicht für andere Zwecke verwendet werden. Hierzu gehören einmal die Wörter **div**, **mod**, **succ**, **pred** als Operatoren und die Zeichenfolgen <, <=, >=, > und ~ als Prädikate und Operatoren der Standardspezifikation INTEGER, das Wort **size** und das Symbol ^ als Operatoren der Standardspezifikation STRING und die Wörter **hd**, **tl**, **rev**, **length**, **nth**, **nthtail** und **nil** sowie die Zeichenfolgen ::, @ und [] als Operatoren des SNL-Listenkonstruktes.

Die SNL-Standardspezifikation STRING erlaubt es weiter, Stringkonstanten in der Form einer Zeichenkette, die in Anführungszeichen eingeschlossen ist, anzugeben, sodaß Zeichenketten, die in Anführungszeichen eingeschlossen sind, syntaktisch als Stringkonstanten (Elemente der Sorte *string*) interpretiert werden.

Die SNL-Tupel- und Rekordkonstrukte definieren weiter Standardprojektionsoperatoren als Zeichenketten der Gestalt # *num*, bzw. # *label*, so daß wir # als reserviert zur Bildung der Namen solcher Projektionsoperatoren ansehen.

Ein Kommentar ist in SNL eine beliebige Zeichenfolge, die durch (* *) geklammert ist und nicht die Zeichenfolge (* oder *) enthält. Wir erlauben hier also keine Schachtelung von Kommentaren.

Damit haben wir die wesentlichen lexikalischen Aspekte unserer Sprache SNL erörtert. Im nächsten Kapitel werden wir uns mit den Sprachkonstrukten zur Definition von SNL-Spezifikationen befassen.

3. Spezifikation abstrakter Datentypen mit SNL

In diesem Kapitel wollen wir den Spezifikationsteil unserer Spezifikations- und Modellierungssprache SNL (SMARAGD Net Language), die dem Werkzeug SMARAGD zu Grunde liegt, näher erläutern. Da der Schwerpunkt dieser Arbeit auf die Beschreibung eines neuartigen Petri-Netz-Modells ausgerichtet ist und sowohl die der Petri-Netz- als auch Spezifikations-Teile der Sprache zugehörige Theorie sehr umfangreich ist, werden wir uns in diesem Kapitel auf die Beschreibung der zum Verständnis von SNL nötigen grundlegenden theoretischen Aspekte der algebraischen Spezifikation beschränken und einige Konzepte der Theorie der Spezifikation abstrakter Datentypen nur informell an Hand ihrer SNL-Syntax beschreiben.

Leser, die eine tiefergehende theoretische Beschreibung aller für den Spezifikationsteil von SNL relevanten Konzepte suchen, finden umfangreiche Einführungen in die Theorie der Spezifikation abstrakter Datentypen u.a. in den Büchern von Ehrich/Gogolla/Lipeck ([Ehrig82]), Ehrig und Mahr ([Ehrig85b,Ehrig90]) und Klaeren ([Klaeren83]). Eine weiterführende Theorie algebraischer Spezifikationen mit partiellen Operatoren findet man z.B. in [Reichel84, Broy80, Schmidt89] beschrieben. Zum Vergleich des Spezifikationsteils von SNL mit anderen Spezifikationssprachen sei auf das CIP-System verwiesen, dessen Spezifikationssprache ähnliche Konzepte zur Spezifikation abstrakter Datentypen enthält (man siehe hier z.B. [CIP85, Partsch90]).

Der Entwurf des Spezifikationsteils von SNL wurde im wesentlichen von drei Randbedingungen geprägt.

- Der Spezifikationsteil von SNL sollte die wesentlichen Konstrukte anderer Spezifikationssprachen in Bezug auf die Spezifikation abstrakter Datentypen enthalten, so daß ein einfacher Austausch von Spezifikationen zwischen SMARAGD und anderen Spezifikationswerkzeugen möglich ist.
- Da uns ML als Interpretationssprache unseres SNL-Netz-Modells am geeignesten erschien, sollten SNL-Spezifikationen zum einen leicht in ML-Implementierungen transformiert werden können, und weiter ein Umschreiben der auf den Spezifikationselementen basierenden Netzbeschriftungen vermieden werden, was sich am einfachsten dadurch realisieren läßt, daß SNL-Terme und Formeln in identischer Form als ML-Terme erkannt werden.
- Da das System von Praktikern und nicht Theoretikern verwendet werden soll, ist das gesamte SMARAGD-System so konzipiert, daß konstruktive Methoden gegenüber mathematischen Methoden, die auch unterstützt werden, im Vordergrund stehen sollen. Für den Spezifikationsteil heißt das u.a., daß der Spezifikationsteil von SNL eine Reihe eingebauter Typen, gegeben durch Standardspezifikationen, enthält, deren Implementierungen durch die Standardtypen von ML, wie *bool*, *int* und *string*, gegeben sind. Weiter gibt es Standardkonstrukte, wie Tupel, Rekord, Union und Liste, die es dem Benutzer im konstruktiven Sinne erlauben, aus den eingebauten Standardspezifikationen neue Spezifikationen ohne großen theoretischen Aufwand zu erzeugen.

Diese drei Randbedingungen bestimmen im wesentlichen den Umfang und die Syntax des Spezifikationsteils von SNL, den wir im folgenden vorstellen wollen.

3.1 Grundsyntax zur Spezifikation abstrakter Datentypen in SNL

Die Grundsyntax zur Spezifikation von abstrakten Datentypen sieht in SNL wie folgt aus:

spec_decls:	**specification** *spec_decl* { ; *spec_decl* }$_0^*$
spec_decl:	*spec_id* [: *sig*] = *spec*
spec_id:	**id**
spec:	*spec_id* [**with** *rename_list* **end**]
	spec *spec_body* **end**

Innerhalb eines Spezifikationsdeklarationsteils können eine oder mehrere Spezifikationsdeklarationen angegeben werden, wobei ein Spezifikationsdeklarationsteil einer Spezifikation *spec* einen Namen *spec_id* zuordnet. Die Spezifikation ist dabei durch den Namen einer bereits definierten Spezifikation oder durch eine Spezifikationsdefinition von der Gestalt **spec** *spec_body* **end** bestimmt. Wenn eine Spezifikation durch den Namen einer bereits existierenden Spezifikation angegeben wird, kann man (einzelne) Elemente dieser Spezifikation noch durch den optionalen **with** *rename_list* **end** Teil umbenennen.

Optional hinter dem Spezifikationsnamen kann man durch Doppelpunkt getrennt eine Angabe über die Elemente der Spezifikation machen, die von der Spezifikation für eine externe Benutzung exportiert werden (*sig*).

Im Rumpf einer Spezifikation müssen die zu der Spezifikation gehörigen Sorten, Konstanten, Operatoren und Prädikate, sowie die Formeln, denen sie genügen müssen, definiert werden. *spec_body* hat daher die Gestalt:

spec_body:	{ *spec_body_decls* }$_0^*$
spec_body_decls:	*spec_sig_decls*
	spec_eqn_decls
spec_sig_decls:	*spec_sort_decls*
	spec_const_decls
	spec_opn_decls
	spec_pred_decls
	.....
spec_eqn_decls:	*spec_var_decls*
	spec_formula_decls
	...

In der Theorie zur Spezifikation abstrakter Datentypen werden die Angaben der Sortennamen, der Konstanten-, Operator- und Prädikatsdeklarationen unter dem Begriff der *Signatur* (*spec_sig_decls*) zusammengefaßt. Der Gleichungsteil (*spec_eqn_decls*) bestimmt die Semantik eines Datentyps. Beide Teile zusammen bilden die Spezifikation eines abstrakten Datentyps (*spec_body*).

Auf die Form und Bedeutung der einzelnen Teile werden wir nun in einzelnen Unterabschnitten genauer eingehen.

3.1.1 Der Begriff der Signatur

Unter dem Blickwinkel der Spezifikation von Datentypen können wir eine *Signatur* als "rein syntaktische Spezifikation" so eines Datentyps ansehen. Sie legt im Sinne einer Schnittstellenbeschreibung die Namen verwendeter Wertebereiche (*Sorten*) für Operationen und Relationen und die Namen von Operationen und Relationen auf diesen Wertebereichen unter Angabe der Argument- und Rückgabewertebereiche fest. Die Namen von Operationen bzw. Relationen bezeichnen wir auch als *Operatoren* (*Operationssymbole*) bzw. *Prädikate* (*Relationssymbole*). Die Angabe der Argument- und Rückgabewertebereiche eines Operations- oder Relationssymbols nennen wir auch den *Typ eines Operators* (*Operationssymbols*) oder *Prädikats* (*Relationssymbols*). Mathematisch können wir daher eine *Signatur* wie folgt definieren:

Def. 1 (Signatur)

Eine Signatur über einer Menge S ist ein 5-Tupel $SIG = (S, \Omega, \Pi, typ_\Omega, typ_\Pi)$, für das gilt:

(a) S, Ω, Π sind Mengen, die wir *Menge der Sorten*, *Menge der Operatoren* (*Operationssymbole*) und *Menge der Prädikate* (*Relationssymbole*) nennen wollen.
Elemente dieser Mengen bezeichnen wir daher auch als *Sorten*, *Operatoren* und *Prädikate*.

(b) $typ_\Omega: \Omega \rightarrow S^* \times S$ ist eine Abbildung, die jedem Operator $\omega \in \Omega$ seinen *Typ* $typ_\Omega(\omega) = (w, s)$ mit $w \in S^*$, $s \in S$ zuordnet. Operatoren vom Typ (ε, s), wobei ε für das leere Wort steht, nennt man auch *nullstellige Operatoren* oder *Konstantensymbole* von der *Sorte s*.

(c) $typ_\Pi: \Pi \rightarrow S^+$ ist eine Abbildung, die jedem Prädikat $\pi \in \Pi$ seinen *Typ* $typ_\Pi(\pi) = w \in S^+$ zuordnet.

Die explizite Angabe von Prädikaten innerhalb einer Signatur, wie wir sie in der obigen Definition beschrieben haben, findet man in dieser Form nicht in der Standardliteratur zur Spezifikation von abstrakten Datentypen. Es ist hier vielmehr üblich, Prädikate als Operatoren aufzufassen, denen als Sorte für den Wertebereich stets eine speziell zu interpretierende Sorte (z.B. mit Namen *bool*) der Spezifikation eines booleschen Datentyps zugeordnet ist, die in der Realisierung als Wertebereich eines booleschen Datentyps im Fall einer prädikatenlogischen Deutung der Spezifikationen interpretiert wird. Auch wenn diese Darstellung praktisch ist, so erlaubt sie unseres Erachtens keine genaue Trennung zwischen der Semantik der Spezifikationssprache, die ja durchaus auch nicht immer durch eine Prädikatenlogik 1. Ordnung beschrieben werden muß, und der Semantik benutzerdefinierter Datentypen innerhalb der Spezifikationssprache. Dies hat uns dazu bewogen, hier einen etwas anderen Ansatz zur theoretischen Beschreibung der Konzepte von abstrakten Datentypen zu wählen.

In der Praxis ist es häufig üblich, nullstellige Operatoren explizit von den Operatoren mit anderen Typen zu unterscheiden. In SNL unterscheiden wir entsprechend zwischen Konstantendeklarationsteil und Operatordeklarationsteil. Eine Signatur besteht daher in SNL zunächst aus Sortendeklarationen, die alle Sorten der Sortenmenge S definieren, Konstantendeklarationen, die Konstantensymbole und ihre Typen definieren, Operatordeklaratio-

nen, die Operatoren und ihre Typen definieren, und Prädikatdeklarationen, die Prädikate und ihre Typen definieren. Eine Signaturdeklaration hat in unserer Sprache daher die Gestalt:

sig_decls:	**signature** *sig_decl* { ; *sig_decl* }$_0^*$
sig_decl:	*sig_id* = *sig*
sig_id:	**id**
sig:	*sig_id*
	sig *sig_body* **end**
sig_body:	{ *sig_body_decls* }$_0^*$
sig_body_decls:	*sig_sort_decls*
	sig_const_decls
	sig_opn_decls
	sig_pred_decls

In einer Signaturdeklaration (*sig_decl*) wird einem Signaturnamen *sig_id* eine Signatur *sig* zugeordnet. Mehrere Signaturdeklarationen können dabei zu einem Signaturdeklarationsteil (*sig_decls*) zusammengefaßt werden. Eine Signatur wird entweder durch den Namen einer bereits definierten Signatur oder durch eine Signaturdefinition beschrieben.

Eine Signaturdefinition (**sig** *sig_body* **end**) besteht aus einer Reihe von Deklarationsteilen, die die Sorten, Konstanten, Operatoren und Prädikate mit ihren jeweiligen Typen definieren. Jeder Deklarationsteil beginnt mit einem Schlüsselwort wie **sort**, **const** oder **opn**.

Sorten werden innerhalb von Sortendeklarationsteilen *sig_sort_decls* definiert. Die Syntax hierfür ist:

sig_sort_decls:	**sort** *sig_sort_decl* { ; *sig_sort_decl* }$_0^*$ [;]
sig_sort_decl:	*sort_id*
sort_id:	**id**

Konstantensymbole (nullstellige Operatoren) werden innerhalb von Konstantendeklarationsteilen definiert. Die Syntax für Konstantendeklarationsteile ist:

sig_const_decls:	**const** *sig_const_decl* { ; *sig_const_decl* }$_0^*$ [;]
sig_const_decl:	*const_id* : *const_type*
const_id:	**con**
const_type:	*sort_name*
sort_name:	**id**

In einer Konstantendeklaration wird einem Konstantensymbol *const_id* als Typ ein Sortenname in der Syntax *const_id* : *sort_name* zugeordnet. Der Sortenname muß dabei vorher in einer Sortendeklaration als Sorte definiert worden sein. Ein Konstantenname ist dabei ein Token vom Typ **con**, d.h. auch Zahlensymbole wie *0* sind als Konstantennamen erlaubt.

Operatoren werden in Operatordeklarationsteilen deklariert. Dabei ist ihr "Typ", d.h. die Sorten der Argumente des Operators und des Resultates eines Operators, anzugeben. Wir haben daher:

sig_opn_decls:	**opn** *sig_opn_decl* { ; *sig_opn_decl* }$_0^*$ [;]
sig_opn_decl:	*opn_id* : *opn_type*
opn_type:	*sort_name* { * *sort_name* }$_0^*$ -> *sort_name*
opn_id:	**id**
	symbol

Ein Operatordeklarationsteil (*sig_opn_decls*) besteht aus dem Schlüsselwort **opn** und einer Folge von durch Semikolon getrennten Operatordeklarationen (*sig_opn_decl*). Jede Operatordeklaration beginnt mit dem Operatornamen, dem durch Doppelpunkt getrennt die Typangabe folgt. Der Typ des Operators ist dabei definiert durch eine Folge von durch * getrennten Sortennamen, die die "Argumente" des Operators bilden, der Symbolfolge -> und einem Sortennamen, der den "Ergebnistyp" des Operators benennt. Wie bei der Konstantendeklaration müssen auch die Sortennamen, die in einer Operatordeklaration auftreten, vorher in Sortendeklarationen als Sorten deklariert worden sein. Ein Operatorname ist entweder ein selbstdefinierter Name vom Tokentyp **id** oder ein Symbolname vom Tokentyp **symbol**.

Prädikate werden in Prädikatdeklarationen hinter dem Schlüsselwort **pred** deklariert. Wie bei Operatordeklarationen ist dabei der Typ des Prädikates (die Liste der Argumentbereiche) anzugeben. Wir haben daher die folgende Syntax:

sig_pred_decls:	**pred** *sig_pred_decl* { ; *sig_pred_decl* }$_0^*$ [;]
sig_pred_decl:	*pred_id* : *pred_type*
pred_type:	*sort_name* { * *sort_name* }$_0^*$
pred_id:	**id**
	symbol

Ein Prädikatdeklarationsteil (*pred_decls*) besteht aus einer Folge von durch Semikolon getrennten Prädikatdeklarationen. Jede Prädikatdeklaration beginnt mit dem Prädikatnamen (*pred_id*) gefolgt von der Typangabe für das Prädikat getrennt durch einen Doppelpunkt. Die Typangabe ist dabei eine Folge von Sortennamen, die die einzelnen "Argumente" des Prädikats bezeichnen und durch * getrennt sind. Wie bei der Konstantendeklaration und Operatordeklaration müssen auch die Sortennamen, die in einer Prädikatdeklaration auftreten, vorher in Sortendeklarationen als Sorten deklariert worden sein. Ein Prädikatname ist entweder ein selbstdefinierter Name vom Tokentyp **id** oder ein Symbolname vom Tokentyp **symbol**.

Für jede Sorte eines abstrakten Datentyps werden implizit (d.h. ohne explizite Angabe) stets ein Prädikat = und ein Prädikat <> als zweistellig definierte Prädikate auf dieser Sorte vorausgesetzt (Diese werden bei einer späteren Realisation des Typs als Gleichheits- und Ungleichheitsrelation in dieser Sorte interpretiert). Diese Einschränkung auf Sorten mit Gleichheit ist in SNL nötig, da wir zur Analyse von SNL-Netz-Modellen Marken im Netz auf Gleichheit überprüfen müssen.

Innerhalb einer Signaturdefinition müssen benutzerdefinierte Namen für Sorten, Konstanten, Operatoren und Prädikate eindeutig sein, d.h. es darf keine zwei benutzerdefinierte Sorten, Konstanten, Operatoren und Prädikate mit gleichem Namen geben (Kein Overloading). Außerdem benutzen Operatoren und Prädikate den gleichen Namensraum, so daß es auch kein benutzerdefiniertes Paar Operator, Prädikat mit gleichem Namen geben darf. Es können aber durchaus eine Konstante und ein Operator oder eine Sorte und ein Operator oder eine Sorte und eine Konstante gleichbenannt sein.

Die Namensräume verschiedener Signaturen sind disjunkt. Es können daher Sorten, Konstanten, Operatoren oder Prädikate in verschiedenen Signaturen gleichbenannt sein. Weiter bilden die Namen für Signaturen selbst einen eigenen Namensraum, so daß z.B. eine Signatur und eine Sorte (wie das häufig gemacht wird) gleichbenannt werden können.

Wir wollen nun zum Begriff der Signatur noch einige Beispiele betrachten:

Beispiel 4 (Beispiele zum Begriff der Signatur)

(a) Betrachten wir als erstes Beispiel einmal die Signatur eines abstrakten Datentyps, wie sie typischerweise zur Spezifikation eines "booleschen Datentyps" verwendet werden könnte.

```
signature BOOL_A = sig
     sort      bool;
     const     true: bool;
               false:bool;
     opn       !  : bool -> bool;
               || : bool * bool -> bool;
               && : bool * bool -> bool ;
end
```

Wir haben hier eine Signatur für einen abstrakten Datentyp `BOOL_A` vor uns, die eine Sorte `bool` definiert. Weiter gibt es zwei Konstantensymbole `true` und `false` vom Typ `bool` und die drei Operatoren `!` (für nicht), `||` (für oder) und `&&` (für und). `!` ist ein einstelliges Operationssymbol für eine Operation, die ein Argument von der Sorte `bool` erwartet und ein Ergebnis von der Sorte `bool` liefert. `||` und `&&` sind zweistellige Operationssymbole für Operationen, die Argumente von der Sorte `bool` erwarten und ein Ergebnis von der Sorte `bool` zurückgeben. Man beachte, daß defaultmäßig weiter stets die implizit vorhandenen Prädikate = und <> als zweistellige Prädikate auf der Sorte `bool` zur Signatur gehören. Wir haben hier übrigens nicht die Standardbezeichnungen `and`, `or` und `not` für die Operatoren von `BOOL_A` gewählt, da `and` als Schlüsselwort von ML in SNL reserviert ist und damit nicht als benutzerdefinierter Name verwendet werden darf.

(b) Eine mögliche Signatur für einen abstrakten Datentyp, der eine "Art natürlicher Zahlen" beschreibt, könnte wie folgt aussehen:

```
signature NAT = sig
     sort      nat;
     const     0 : bool;
     opn       succ : nat -> nat;
               + : nat * nat -> nat;
     pred      <= : nat * nat;
end
```

Wir haben hier eine Sorte `nat`, ein Konstantensymbol `0` vom Typ `nat`, zwei Operatoren + und `succ`, sowie ein Prädikat <=. Die Operatoren + und `succ` stehen dabei für Operationen Nachfolger (`succ`) und Addiere (+) auf natürlichen Zahlen. Das <= Prädikat steht für die Kleiner-Gleich-Relation auf den natürlichen Zahlen. Auch hier sind standardmäßig = und <> vordefiniert.

(c) Betrachten wir als ein weiteres Beispiel einmal die Signatur eines abstrakten Datentyps, wie sie typischerweise zur Spezifikation eines Datentyps "Stack" dienen könnte.

```
signature STACK = sig
     sort      stack; item;
     const     empty : stack;
     opn       push  : stack * item -> stack;
               pop   : stack -> stack;
               top   : stack -> item;
     pred      isempty : stack;
end
```

Die Signatur `STACK` definiert zwei Sorten `stack` und `item`, auf denen typische Operationen zur Manipulation eines Stacks von Elementen des Typs `item` aufsetzen. Solche Operationen wären eine Push-Operation (Operationssymbol `push`), die ein Element vom Typ `item` auf einen Stack bringt, eine Pop-Operation (Operationssymbol `pop`), die das oberste Element von einem Stack entfernt und eine Top-Operation (`top`), die das oberste Element auf dem Stack zurückgibt. Eine Relation (Relationssymbol `isempty`) könnte vorhanden sein, die angibt, ob der Stack leer ist oder Elemente enthält.

Die obigen Beispiele zeigen einfache Signaturen, die unabhängig voneinander sind. Bei komplexeren Anwendungen beobachtet man aber häufig Abhängigkeiten zwischen verschiedenen Spezifikationen in der Weise, daß die eine Spezifikation Elemente einer anderen Spezifikation verwendet, so daß die Signatur der einen Spezifikation als Bestandteil der anderen erscheint. So könnte man im Beispiel 4c den abstrakten Datentyp `STACK` in der Weise präzisieren, daß man die "generische" Sorte `item` durch die "konkretere" Sorte `nat` der Spezifikation `NAT` aus Beispiel 4b ersetzt. Man erhält dann die Signatur für einen abstrakten Datentyp `STACKNAT`, der einen Stack von natürlichen Zahlen spezifiziert. Dabei kann es weiter sinnvoll sein, auch alle (oder einige) der Operatoren und/oder Prädikate des abstrakten Typs `NAT` auf `STACKNAT` zu übertragen, um z.B. verschiedene Stackelemente addieren oder vergleichen zu können.

Es wäre nun lästig, wenn man bei solchen Abhängigkeiten zwischen abstrakten Datentypen stets alle gewünschten Elemente eines bereits definierten Datentyps, die man innerhalb eines neu zu definierenden abstrakten Datentyps verwenden will, noch einmal spezifizieren muß. Viel günstiger wäre es, wenn man über eine spezielle Deklaration den bereits definierten Datentyp auf einfache Weise als Bestandteil in die neu zu definierende algebraische Spezifikation integrieren kann. Ein solcher Mechanismus, den wir in SNL bereitstellen, erlaubt es, Spezifikationen in andere Spezifikationen als Unterspezifikationen einzubinden. Der Name solcher Unterspezikationen und deren Signatur muß dann ebenfalls ein Bestandteil der Signatur des betreffenden abstrakten Datentyps sein. Wir erweitern daher die Menge unserer Deklarationen, die im Rumpf einer Signaturdefinition stehen können, um einen Spezifikationsdeklarationsteil:

sig_body_decls: *sig_sort_decls*
sig_const_decls
sig_opn_decls
sig_pred_decls
sig_spec_decls

Der Spezifikationsdeklarationsteil selbst hat die folgende Syntax:

sig_spec_decls:	**specification** *sig_spec_decl* { ; *sig_spec_decl* }$_0^*$ [;]
sig_spec_decl:	*spec_id* : *sig*
spec_id:	**id**

Innerhalb eines Spezifikationsdeklarationsteils (*sig_spec_decls*) hinter dem Schlüsselwort **specification** kann ein Benutzer bei der Definition der Signatur eines abstrakten Datentyps Namen und Signaturen für bereits vorhandene abstrakte Datentypen angeben, die er als Unterspezifikationen mit dem gewählten Namen und der gewählten Signatur in die betreffende Spezifikation einbinden will. Eine Spezifikationsdeklaration ordnet dabei einer Signatur *sig* einen Namen *spec_id* zu, der zur Referenz der Signaturelemente von *sig* innerhalb der übergeordneten Signatur verwendet wird.

Die Elemente der Signatur von Unterspezifikationen können dann in der Form eines Namens

spec_id_1.spec_id_2. ... .spec_id_n.name_of_element_in_signature_spec_id_n

referenziert werden, wie man das von Strukturen in Sprachen wie PASCAL oder C kennt. Wir müssen daher die Syntax für Namen (Token vom Typ *.._name* nicht vom Typ *.._id*) von Elementen innerhalb einer Signatur auf solche "Pfadnamen" verallgemeinern. Wir definieren daher:

spec_name:	*spec_id* { . *spec_id* }$_0^*$
sort_name:	[*spec_name* .] *sort_id*
const_name:	[*spec_name* .] *const_id*
opn_name:	[*spec_name* .] *opn_id*
pred_name:	[*spec_name* .] *pred_id*

In den meisten Spezifikationssprachen werden im Gegensatz zu unserem Ansatz die Elemente eingebetteter Spezifikationen direkt über den Namen, den sie in der eingebetteten Spezifikation besitzen, angesprochen. Dies hat jedoch den Nachteil, daß es leicht zu Namenskollisionen kommen kann, die durch unseren Ansatz vermieden werden. Weiter sind bei uns Unterspezifikationen bzgl. ihrer Semantik automatisch vor Veränderungen dieser Semantik durch Gleichungen in der die Unterspezifikation einbindenden Spezifikation geschützt, da wir so eine Beeinflussung per Definition nicht erlauben. Diese Eigenschaft, die man Hierarchieerhaltung (hierarchy preserving) nennt, fordern die meisten Spezifikationssprachen für die Sprachkonstrukte, die einen hierarchischen Aufbau von Spezifikationen unterstützen. Sie ist jedoch für den Benutzer nicht offensichtlich, wenn die Namensräume von eingebundener und einbindender Spezifikation nicht streng getrennt sind.

Die Form des Signaturbegriffs einschließlich des Konzepts der Unterspezifikationen, wie wir ihn verwenden, findet man in fast identischer Form als Bestandteil des Modulkonzeptes der Sprache ML. Diese Übereinstimmung ist nicht zufällig, sondern liegt daran, daß wir ML als die Sprache für die Realisation von abstrakten Datentypen gewählt haben. Wie wir noch genauer darlegen werden, kann man unseren Begriff der Signatur als Schnittstellenbeschreibung ansehen, die sowohl die externe Schnittstelle einer algebraischen Spezifikation als auch die Schnittstelle eines ML-Moduls, der diese algebraische Spezifikation realisiert, beschreibt.

Beispiel 5 (Beispiel zum Import von Signaturen)

Betrachten wir als Beispiel für die Angabe einer Unterspezifikation den schon erwähnten abstrakten Datentyp `STACKNAT`. Wir gehen hier davon aus, daß wir eine Signatur `NAT` wie im Beispiel 4 bereits definiert haben.

```
signature STACKNAT = sig
    specification N : NAT;
    sort          stack;
    const         empty : stack;
    opn           push : stack * N.nat -> stack;
                  pop : stack -> stack;
                  top : stack -> N.nat;
    pred          isempty : stack;
end
```

Die Signatur von STACKNAT könnte man durch Einsetzen von NAT auch wie folgt beschreiben:

```
signature STACKNAT = sig
     sort      N.nat; stack;
     const     N.0 : N.nat;
               empty : stack;
     opn       N.succ : N.nat -> N.nat;
               N.+ : N.nat * N.nat -> N.nat;
               push : stack * N.nat -> stack;
               pop : stack -> stack;
               top : stack -> N.nat;
     pred      <= : N.nat * N.nat;
               isempty : stack;
end
```

3.1.2 Spezifikationen und Signaturen

Die Grundsyntax zur Angabe von abstrakten Datentypspezifikationen haben wir bereits am Anfang dieses Kapitels vorgestellt. Sie lautet:

spec_decls: **specification** *spec_decl* { ; *spec_decl* }$_0^*$ [;]
spec_decl: *spec_id* [: *sig*] = *spec*
spec_id: **id**
spec: *spec_id* [**with** *rename_list* **end**]
spec *spec_body* **end**

Wenn wir die optionalen Teile zunächst einmal ignorieren, wird innerhalb einer Spezifikationsdeklaration einem Namen *spec_id* eine Spezifikation zugeordnet, die entweder durch den Namen einer bereits definierten Spezifikation oder durch eine Spezifikationsdefinition von der Gestalt **spec** *spec_body* **end** gegeben ist. Wir haben hier also eine Syntax, die analog zu der für Signaturen ist. Die Analogie läßt sich nun noch weiter ausdehnen. Nach der am Anfang gegebenen Syntax kann man sich den Rumpf einer Spezifikationsdefinition in einen Signatur- und einen Gleichungsteil unterteilt denken:

spec_body: { *spec_body_decls* }$_0^*$
spec_body_decls: *spec_sig_decls*
spec_eqn_decls

Der Signaturdeklarationsteil im Spezifikationsrumpf besteht dabei aus Deklarationsteilen, die analog zu den Deklarationsteilen im Rumpf einer Signaturdefinition benannt sind.

spec_sig_decls: *spec_sort_decls*
spec_const_decls
spec_opn_decls
spec_pred_decls
.....

Dies hat auch seinen guten Grund, denn der Signaturteil innerhalb einer Spezifikationsdefinition beschreibt gerade die mathematische Signatur des abstrakten Datentyps und enthält daher wie eine Signaturdefinition auch die Definition der Sorten, Konstanten, Operatoren und Prädikate des abstrakten Datentyps mit ihren Typangaben. Im einfachsten Fall sind also zunächst die Signaturdeklarationsteile in einer Spezifikations- und Signaturdefinition syntaktisch gleich definiert. Wir haben damit im einfachsten Fall zunächst:

spec_sort_decls: *sig_sort_decls*
spec_const_decls: *sig_const_decls*
spec_opn_decls: *sig_opn_decls*
spec_pred_decls: *sig_pred_decls*

Durch Austauschen der Schlüsselworte **signature** und **sig** im Beispiel 4 durch **specification** und **spec** erhalten wir also gültige Formen von Spezifikationsdeklarationen, die allerdings noch keinerlei Formeln zur Spezifikation der Semantik beinhalten.

Beispiel 6 (Die Signatur BOOL_A als Spezifikation geschrieben)

Die Signatur `BOOL_A` kann man als Spezifikation `BOOL_A` schreiben, indem man im Beispiel 4 das Schlüsselwort **signature** durch **specification** und das Schlüsselwort **sig** durch **spec** ersetzt.

```
specification BOOL_A = spec
     sort      bool;
     const     true: bool ;
               false:bool;
     opn       !  : bool -> bool;
               || : bool * bool -> bool;
               && : bool * bool -> bool ;
end
```

Wie bei Signaturen gibt es auch bei Spezifikationen die Möglichkeit, bereits definierte Spezifikationen in neue Definitionen einzubeziehen. Im Gegensatz zu Signaturen bieten wir bei Spezifikationen hierfür mehrere Möglichkeiten an, wobei sich nur die Methode der expliziten Hierarchisierung auch nach außen sichtbar in der Signatur niederschlägt.

Unter einer Hierarchisierung der Definition von Spezifikationen verstehen wir dabei die Einbindung bereits definierter Spezifikationen in Form von Unterspezifikationen in andere Spezifikationen in der Weise, daß erstens die transitive Abhängigkeitsrelation (Spezifikation 1 hängt von der eingebunden Spezifikation 2, diese von einer eingebundenen Spezifikation 3 usw. ab) keine Zyklen enthält und zweitens die Semantik der eingebundenen Spezifikationen nicht durch Gleichungen der einbindenden Spezifikationen beeinflußt werden darf. Weiter verlangt man für die Realisierungen solcher hierarchisch aufgebauter Spezifikationen, daß die Realisation einer übergeordneten Spezifikation durch ein algebraisches System auch eine Realisation der eingebundenen Unterspezifikationen in Form von algebraischen Teilsystemen enthält und umgekehrt eine Realisation des Untersystems durch Hinzufügen von Wertebereichen, Konstanten, Funktionen und Relationen zu einer Realisation der übergeordneten Spezifikation ausgebaut werden kann, so daß hier ein modulartiger Aufbau von Realisationen und Spezifikationen möglich ist.

Prinzipiell erlauben wir, daß innerhalb einer Spezifikation Elemente einer anderen Spezifikation durch Angabe von vollqualifizierten Namen (*path_name.element_name*) verwendet werden können, ohne daß die zugehörigen Spezifikationen explizit als Unterspezifikationen eingebunden werden müssen. Im Fall einer solchen *impliziten* Referenz einer anderen Spezifikation wird diese automatisch als Unterspezifikation mit von außen nicht sichtbaren und damit nicht verwendbaren Namen eingebunden. Die Anwendung einer solchen *impliziten Hierarchisierung* beschränken wir aber, wie wir etwas später noch ausführlich erörtern werden, auf Fälle, in denen die implizit eingebundenen Spezifikationen völlig nach außen verdeckt sind, d.h. die referenzierten Elemente werden innerhalb der Signatur der übergeordneten Spezifikation durch alternative Namen beschrieben.

Sollen Signaturelemente einer Spezifikation, die als Unterspezifikation eingebunden werden soll, von einem externen Benutzer verwendet werden können, so muß eine explizite Deklaration der einzubindenden Spezifikation als Unterspezifikation erfolgen, wobei der Name, über den Elemente der Unterspezifikation angesprochen werden, und die exportierte Signatur der eingebundenen Spezifikation auch in einer Signatur der übergeordneten Spezifikation auftreten muß. Die Syntax zur expliziten Angabe von Unterspezifikationen ist in Spezifikationsdefinitionen etwas anders als bei Signaturen, da wir zwischen Einbindung einer Signatur (im Fall der Signaturdefinition) und Einbindung einer Spezifikation, die ja auch Formeln enthalten kann, (im Fall einer Spezifikation) unterscheiden wollen. Wir erweitern daher unsere Syntax für Spezifikationen wie folgt:

spec_sig_decls: *spec_sort_decls*
spec_const_decls
spec_opn_decls
spec_pred_decls
spec_decls

Neu ist hier also, daß wir es nun im Signaturteil einer Spezifikation (beschrieben durch *spec_sig_decls*) zulassen, daß hier ebenfalls Spezifikationsdeklarationsteile stehen dürfen, die nun allerdings Unterspezifikationen beschreiben. Während man auf oberster Ebene bei der Deklaration einer Spezifikation in der Regel die Syntax *spec_id* = **spec** *spec_body* **end** zur Definition der Spezifikation verwendet, wird man zur Deklaration von Unterspezifikationen in der Regel die Syntax *spec_id* [: *sig*]= *spec_name* [**with** *rename_list* **end**] verwenden, da man ja eine bereits auf oberster Ebene definierte Spezifikation als Unterspezifikation einbinden will. Man beachte dabei, daß durch die Verwendung von *spec_name* auf der rechten Seite des Gleichheitszeichens es auch erlaubt ist, Unterspezifikationen anderer auf oberster Ebene definierter Spezifikationen einzubinden. Die Elemente einer so

definierten Unterspezifikation (sofern sie zur Benutzung freigegeben sind, wie wir etwas später erläutern) können dann (wie im Abschnitt über Signaturen schon erwähnt) über qualifizierte Pfadnamen der Gestalt *spec_id_1.spec_id_2. ... .element* referenziert werden. Wir wollen hierzu das Beispiel unserer NATSTACK-Signatur einmal als Spezifikation schreiben.

Beispiel 7 (Verwendung der Spezifikation NAT als Unterspezifikation in STACKNAT)

Wir gehen davon aus, daß wir eine Spezifikation NAT wie folgt gegeben haben:

```
specification NAT = spec
     sort      nat;
     const     0 : nat;
     opn       succ : nat -> nat;
               + : nat * nat -> nat;
     pred      <= : nat * nat;
end
```

Unsere Spezifikation STACKNAT definieren wir dann wie folgt:

```
specification STACKNAT = spec
     specification N = NAT;
     sort      stack;
     const     empty : stack;
     opn       push : stack * N.nat -> stack;
               pop  : stack -> stack;
               top  : stack -> N.nat;
     pred      isempty : stack;
end
```

Man beachte, daß wir hier im Gegensatz zur Signaturdefinition = anstatt : in der Deklaration der Unterspezifikation verwendet haben, da es sich hier um die Einbindung einer Spezifikation handelt.

Es stellt sich nun noch die Frage, wofür man das optionale Konstrukt **with** *rename_list* **end** verwendet. Durch dieses Konstrukt kann man bei der Referenzierung einer bereits definierten algebraischen Spezifikation Signaturelementen der referenzierten Spezifikation einen neuen Namen innerhalb des Kontextes der referenzierenden Spezifikation geben.

Auf oberster Ebene kann man z.B. die Elemente einer Spezifikation auf einfache Weise innerhalb einer neuen Spezifikation neu benennen, wie dies das folgende Beispiel verdeutlicht.

Beispiel 8 (Alternative Benennung der Elemente von BOOL_A)

Wir wollen einmal eine neue Spezifikation BOOL_B einführen, die die Elemente von BOOL_A alternativ durch bool, true, false, und, oder und nicht benennt. Dies läßt sich wie folgt realisieren.

```
specification BOOL_B = BOOL_A
     with      und   for &&,
               oder  for ||,
               nicht for !
     end;
```

Die Syntax für die *rename_list* ist:

rename_list: *rename_decl* {[,] *rename_decl* }$_0^*$
rename_decl: *sort_id* **for** *sort_id*
const_id **for** *const_id*
opn_id **for** *opn_id*
pred_id **for** *pred_id*
spec_id **for** *spec_id*

Innerhalb einer *rename_decl* wird dabei ein Element der referenzierten Spezifikation *name_2* in ein Element der neuen Spezifikation mit Namen *name_1* nach der Syntax *name_1* **for** *name_2* umbenannt. Man beachte dabei, daß nach wie vor Pfadnamen verwendet werden müssen, um die Elemente der neuen Spezifikation anzusprechen. Elemente der Signatur, die nicht in der *rename_list* auftreten, aber exportiert werden, können in der neuen Spezifikation weiter mit ihrem alten Namen angesprochen werden.

Wir wollen uns nun der Erklärung der optionalen : *sig* Konstrukte innerhalb der bisherigen Syntax von Spezifikationen zuwenden. Häufig steht man bei der Angabe von Spezifikationen von abstrakten Datentypen vor dem

Problem, daß man zur Definition der Eigenschaften eines Datentyps einige Hilfsgrößen verwenden muß oder möchte, die der Benutzer einer abstrakten Datentypspezifikation nicht als von ihm verwendbare Elemente der Signatur der abstrakten Datentypspezifikation ansehen soll.

Für den oben beschriebenen Fall sieht man in der Regel vor, daß man bei der Definition einer Spezifikation angeben kann, welche der intern in der Spezifikation verwendeten Symbole man benutzen darf. Wir spezifizieren diese "Exportangabe" über die Angabe einer Signatur *sig*. Betrachten wir hierzu ein Beispiel:

Beispiel 9 (Beispiel zur Verwendung von Signaturen mit Spezifikationen)

(a) Wir wollen eine Signatur für natürliche Zahlen mit einer Konstanten `0`, der Operation `succ` (Inkrementierung um 1) und der Multiplikation `*` angeben.

```
signature NAT'MULT = sig
     sort   nat;
     const  0 : nat;
     opn    succ : nat -> nat;
            * : nat * nat -> nat;
end
```

Wenn wir für diese Signatur eine Semantik im Sinne der natürlichen Zahlen spezifizieren wollten, würden wir es mit einem Ansatz versuchen, der die Gleichungen `0*m=0` und `succ(n ) * m = succ( ... (succ (n*m)) ... )` beinhalten würde, wobei `succ` in der letzten Gleichung auf der rechten Seite *m*-mal auftritt und damit die Anzahl der Auftreten von `succ` in dieser Gleichung abhängig von *m* wäre. Wir würden mit dieser Methode unendlich viele Gleichungen benötigen, um die Semantik von `*` zu spezifizieren, was nicht Sinn der Sache sein kann.

Wenn wir nun als eine weitere Hilfsoperation + als Addition auf den natürlichen Zahlen einführen, können wir die Semantik von `*` über die Gleichungen `0*m=0` und `succ(n) * m = m + (n*m)` spezifizieren. Wir wollen also in der Spezifikation die Operation + als Hilfsgröße einführen, aber garantieren, daß ein Anwender die Spezifikation nur gemäß der obigen Signatur `NAT'MULT` verwenden kann. Dies schreiben wir in der Form:

```
specification NAT'MULT :NAT'MULT = spec
     sort   nat;
     const  0 : nat;
     opn    succ : nat -> nat;
            +  : nat * nat -> nat;
            *  : nat * nat -> nat;

..... die Gleichungen .....

end
```

Der Zusatz : `NAT'MULT` zum Namen der Spezifikation `NAT'MULT` sagt nun aus, daß für die externe Benutzung der Spezifikation `NAT'MULT` nur die Elemente der Signatur `NAT'MULT` zur Verfügung stehen. Die Verwendung von + ist damit bei der Anwendung der Spezifikation `NAT'MULT` verboten.

Es sollte dabei klar sein, daß die Signaturangaben in der Signatur und in der Spezifikation gleich sein müssen. Die Spezifikation darf allerdings Signaturelemente enthalten, die nicht in der Signaturdefinition auftreten. Genau genommen müssen wir daher zwischen zwei Bedeutungen des Signaturbegriffs unterscheiden. Der Signaturbegriff im Sinne der Mathematik würde im Fall unserer Spezifikation `NAT'MULT` die verdeckte Operator + als Operation von `NAT'MULT` enthalten. Die Signatur `NAT'MULT` im Sinne unserer Definition enthält dagegen nur die Elemente der mathematischen Signatur von `NAT'MULT`, die von der Spezifikation `NAT'MULT` zur externen Verwendung exportiert werden. Es gibt aber auch noch eine andere Erklärungsmöglichkeit des Umgangs mit Spezifikationen, die verdeckte Elemente enthalten. Wir können die Signatur `NAT'MULT` als die mathematische Signatur des spezifizierten Datentyps ansehen, der entsteht, wenn wir die Realisierungen der Spezifikation `NAT'MULT` auf die Signaturelemente der Signatur `NAT'MULT` einschränken. Wir werden diese Interpretation für Realisationen von Spezifikationen mit verdeckten Elementen übernehmen.

Auf Grund der Syntax für Signaturen kann man im obigen Beispiel für den Namen der Signatur `NAT'MULT` auch die Signaturdefinition von `NAT'MULT` selbst einsetzen. Wir könnten also schreiben

```
specification NAT'MULT :
sig (* Signatur von NAT'MULT *)
     sort      nat;
     const     0 : nat;
     opn       succ : nat -> nat;
               * : nat * nat -> nat;
end =spec (* Spezifikation von NAT'MULT *)
     sort      nat;
     const     0 : nat;
     opn       succ : nat -> nat;
               + : nat * nat -> nat;
               * : nat * nat -> nat;
(*....... die Gleichungen .....*)
end
```

Da man in diesem Fall alle Signaturdeklarationen doppelt hat, was zu unnötiger Schreibarbeit führt, kann man in der Spezifikation Signaturdeklarationen, die in der Signaturdefinition schon vorhanden sind, auch weglassen und braucht innerhalb der Spezifikation nur noch die Elemente der Signatur zu deklarieren, die nicht bereits in der Signaturdeklaration vorhanden sind. Das obige Beispiel verkürzt sich also zu:

```
specification NAT'MULT :
sig (* Signatur von NAT'MULT *)
     sort      nat;
     const     0 : nat;
     opn       succ : nat -> nat;
               * : nat * nat -> nat;
end = spec (* Spezifikation von NAT'MULT *)
           (* verdeckte Operation + *)
     opn       + : nat * nat -> nat;
(*....... die Gleichungen .....*)
end
```

Da auch in der Signatur NAT'MULT bereits nat, 0, succ und * deklariert sind, kann man auch kurz schreiben

```
specification NAT'MULT : NAT'MULT = spec
     (* Spezifikation von NAT'MULT *)
           (* verdeckte Operation + *)
     opn       + : nat * nat -> nat;

....... die Gleichungen .....

end
```

wobei alle Deklarationen aus der Signatur NAT'MULT in die Spezifikation NAT'MULT übernommen werden. Welche der drei angegebenen Darstellungen man wählt, ist im Prinzip egal. Wenn man getrennt stehende Definitionen der Signatur und Spezifikation eines Datentyps hat, ist es aber in der Regel sinnvoll, die Signaturdeklarationen innerhalb der Spezifikation zu duplizieren, um sie in der Spezifikationsdefinition sichtbar vor Augen zu haben. Dies reduziert Fehler, die dadurch entstehen, daß man Signaturelemente falsch verwendet, weil man ihre Definition nicht vor Augen hat. Das Werkzeug, das die SNL-Spezifikation interpretiert, sollte dann Signatur- und Spezifikationsdefinition darauf überprüfen, ob in diesen duplizierte Signaturelemente gleich definiert sind.

Da die Signaturangabe in einer Spezifikationsdeklaration optional ist, müssen wir noch festlegen, was im Fall des Fehlens der Signaturangabe passiert. Die Defaultregel ist hier, daß bei Fehlen der Signaturangabe alle innerhalb der Spezifikation definierten Signaturelemente extern verwendbar sind.

Wir wollen noch einmal unser Beispiel STACKNAT betrachten. Die Spezifikation STACKNAT verwendet die Spezifikation NAT in Form einer Unterspezifikation, so daß die Signatur STACKNAT aus Beispiel 5 die externe Schnittstelle von STACKNAT beschreibt. Damit sind alle Elemente der Spezifikation NAT auch für Benutzer von STACKNAT in der Form qualifizierter Namen verwendbar.

STACKNAT benötigt von der Spezifikation NAT aber nur die Sorte N.nat, so daß es durchaus sinnvoll wäre, die Signatur von STACKNAT bzgl. der verwendeten Spezifikation NAT auf das Element N.nat einzuschränken. Dies kann man dadurch erreichen, daß man in der Unterspezifikationsanweisung für NAT zum Spezifikationsna-

men N eine Signatur explizit angibt, die nur die Sorte nat enthält. Ein externer Benutzer kann dann nur die Elemente von N verwenden, die innerhalb dieser Signatur spezifiziert sind. Man beachte dabei, daß die angegebene Signatur nur eine Einschränkung der exportierten Signatur von NAT sein darf und keine Elemente enthalten kann, die nicht auch in NAT definiert sind und von NAT exportiert werden. Das folgende Beispiel zeigt eine Spezifikation von STACKNAT, bei der die Benutzung der NAT Unterkomponente auf die Sorte nat eingeschränkt wurde.

Beispiel 10 (Einschränkung der Benutzung von Unterspezifikationen)

Wir beschränken die Benutzung der Unterspezifikation N=NAT in STACKNAT, indem wir N eine Signatur als Typ zuordnen, die nur die Sorte nat enthält.

```
specification STACKNAT = spec
     specification N : sig sort nat; end = NAT;
     sort      stack;
     const     empty : stack;
     opn       push : stack * N.nat -> stack;
               pop  : stack -> stack;
               top  : stack -> N.nat;
     pred      isempty : stack;
end
```

Nun wäre es noch günstig, wenn man N.nat noch innerhalb von STACKNAT so umbenennen könnte, daß die Unterspezifikation N von außen ganz verdeckt wäre. Wir erlauben daher, daß man beliebige Elemente einer Spezifikation mit einem weiteren Namen versehen kann. In unserem Beispiel könnte man N.nat z.B. einen weiteren Namen nat über die Sortendeklaration

```
sort        nat=N.nat
```

zuordnen. Wenn wir nun anstatt N nur noch nat in der Signatur der Spezifikation aufführen, so ist die Benutzung der Spezifikation NAT nach außen völlig verdeckt.

Beispiel 11 (Alternative Namen und Verdecken von Hierarchien)

Wir wollen in der Spezifikation STACKNAT einen alternativen Namen nat für N.nat einführen und die Signatur von STACKNAT so umdefinieren, daß sie keine explizite Referenz mehr auf die Unterspezifikation N enthält. Das sieht dann wie folgt aus:

```
specification STACKNAT :
sig (* Signatur von STACKNAT *)
     sort      stack; nat;
     const     empty : stack;
     opn       push : stack * nat -> stack;
               pop : stack -> stack;
               top : stack -> nat;
     pred      isempty : stack;
end =
spec (* Spezifikationsteil *)
     specification N : NAT = NAT;
     sort nat=N.nat;

.... Hier folgen die Gleichungen ...

end
```

Wie wir schon angedeutet haben, erlauben wir eine implizite Einbettung einer Spezifikation als Unterspezifikation, wenn die externe Schnittstelle (d.h. die Signatur der einbettenden Spezifikation) keine expliziten Referenzen auf die inkludierte Spezifikation enthält. Im obigen Beispiel können wir daher die Unterspezifikationsdeklaration von NAT auch weglassen und einfach schreiben:

Beispiel 12 (Implizite Referenz und deren Verdeckung)

Das Beispiel 11 schreibt sich mit einer impliziten Referenz von NAT wie folgt:

```
specification STACKNAT :
sig (* Signatur von STACKNAT *)
     sort      stack; nat;
     const     empty : stack;
     opn       push : stack * nat -> stack;
               pop : stack -> stack;
               top : stack -> nat;
     pred      isempty : stack;
end =
spec (* Spezifikationsteil *)
     sort nat=NAT.nat;

.... Hier folgen die Gleichungen ...

end
```

Damit die Verwendung alternativer Namen für bereits definierte Sorten, Konstanten, Operatoren und Prädikate erlaubt ist, müssen wir unsere Sprachsyntax wie folgt verallgemeinern.

spec_sort_decls:	**sort** *spec_sort_decl* { ; *spec_sort_decl* }$_0^*$ [;]
spec_const_decls:	**const** *spec_const_decl* { ; *spec_const_decl* }$_0^*$ [;]
spec_opn_decls:	**opn** *spec_opn_decl* { ; *spec_opn_decl* }$_0^*$ [;]
spec_pred_decls:	**pred** *spec_pred_decl* { ; *spec_pred_decl* }$_0^*$ [;]
spec_sort_decl:	*sort_id* [= *sort_name*]
spec_const_decl:	*const_id* [: *sort_name*] [= *const_name*]
spec_opn_decl:	*opn_id* [: *opn_type*] [= *opn_name*]
spec_pred_decl:	*pred_id* [: *pred_type*] [= *pred_name*]

Die Typangaben in den Signaturdeklarationen einer Spezifikation sind nun optional und müssen im Fall einer alternativen Namensdeklaration nicht unbedingt vorhanden sein, da ja in diesem Fall der Typ durch das rechts vom Gleichheitszeichen folgende Signaturelement, das vorher mit Typ definiert sein muß, eindeutig bestimmt ist. Man beachte weiter, daß die ..._name* Konstrukte Pfadnamen erlauben, während die ..._id* Konstrukte keine Pfadkomponenten enthalten dürfen.

Alternativ zur Methode der Unterspezifikationen möchte man oftmals eine bereits definierte Spezifikation so in eine neu zu definierende Spezifikation einbinden, als ob der Rumpf dieser einzubindenden Spezifikation innerhalb der einbindenden Spezifikation selbst definiert wäre. Hierzu führen wir eine weitere Deklarationsart innerhalb von Spezifikationsdefinitionen ein, die wir Inkludedeklaration nennen. Die Wirkung einer Inkludedeklaration der Gestalt

```
include     NAT
```

ist dann so definiert, daß man sich diese Inkludedeklaration durch den Inhalt des Rumpfes der Spezifikation NAT ersetzt denken muß. Wir haben also:

spec_sig_decls:	*spec_sort_decls*
	spec_const_decls
	spec_opn_decls
	spec_pred_decls
	spec_decls
	spec_include_decls

Den Inkludedeklarationsteil definieren wir wie folgt:

spec_include_decls:	**include** *spec_include_decl* { ; *spec_include_decl* }$_0^*$ [;]
spec_include_decl:	*spec_name* [**with** *rename_list* **end**]

Damit könnten wir unser STACKNAT Beispiel auch wie folgt umschreiben:

Beispiel 13 (Alternative Namen und Verdecken von Hierarchien mit der Inkludedeklaration)

Unter Verwendung der Inkludedeklaration können wir STACKNAT wie folgt beschreiben:

```
specification STACKNAT :
sig (* Signatur von STACKNAT *)
     sort      stack; nat;
     const     empty : stack;
     opn       push : stack * nat -> stack;
               pop : stack -> stack;
               top : stack -> nat;
     pred      isempty : stack;
end =

spec (* Spezifikationsteil *)
     include   NAT;
(* Alle Elemente aus NAT sind ab hier ohne Angabe eines *)
(* Pfadnamens verwendbar *)

.... Hier folgen die Gleichungen ...

end
```

Man beachte dabei, daß die Semantik einer Inkludedeklaration im allgemeinen von der einer analogen Unterspezifikationsdeklaration verschieden ist, da die Semantik einer Inkludedeklaration einfach als abkürzende Schreibweise für die Angabe aller Deklarationen im Rumpf der inkludierten Spezifikation anstatt der Inkludedeklaration definiert ist. Das bedeutet insbesondere, daß nachfolgende Gleichungen die Semantik der durch den Rumpf der inkludierten Spezifikation gegebenen Signaturelemente verändern können, ohne daß dies wie im Fall einer Unterspezifikationsdeklaration verboten ist. Realisierungen der inkludierenden Spezifikation enthalten daher im allgemeinen auch nicht Realisationen der inkludierten Spezifikation in Form von algebraischen Untersystemen, wie dies bei Unterspezifikationsdeklarationen der Fall war. Wenn man aber als Benutzer dafür sorgt, daß hinter einer Inkludedeklaration keine Gleichungen folgen, die die Semantik der durch die Inkludedeklaration gegebenen Elemente verändern, so kann man sich ein Inkludedeklaration durchaus äquivalent zu einer Unterspezifikationsdeklaration denken, in der alle Elemente der eingebundenen Spezifikation ohne Angabe von Pfadnamen, d.h. nur über ihre Elementnamen referenziert werden können.

Bei der Inkludierung von Spezifikationen erlauben wir es wie auch bei der Einbindung von Spezifikationen als Unterspezifikationen, daß ein und dieselbe Spezifikation mehrfach eingebunden werden darf. Während dies im Fall einer Unterspezifikation nicht problematisch ist, da hier jede Unterspezifikation in der Regel einen eindeutigen Namen besitzt, hätten wir bei der mehrfachen Inkludierung einer Spezifikation die Deklarationen der einzelnen Elemente der inkludierten Spezifikationen mehrfach, was nicht sinnvoll ist. Generell gilt daher in SNL die Regel, daß weitere Deklarationen eines bereits definierten Signaturelementes die Definition von diesem überschreiben, aber nur dann erlaubt sind, wenn nicht vorher definierte andere Elemente das betreffende Element auf Grund der neuen Deklaration syntaktisch falsch verwenden würden. Um Namenskollisionen zwischen gleichbenannten, aber verschiedenen Elementen von Spezifikationen, die inkludiert werden sollen, zu vermeiden, ist es daher ähnlich wie bei Unterspezifikationsdeklarationen erlaubt, die Elemente einer zu inkludierenden Spezifikation bei der Inklusion umzubenennen. Die Syntax ist hierbei äquivalent zu der von Unterspezifikationsdeklarationen.

3.1.3 Formeln

Formeln sind innerhalb unserer SNL-Definition abstrakter Datentypen geschlossene Formeln einer prädikatenlogischen Sprache 1. Ordnung, wie dies bei der Spezifikation abstrakter Datentypen üblich ist. Aus theoretischer Sicht beschreiben wir im folgenden ein etwas allgemeineres Konzept, das beliebige formale Sprachen 1. Ordnung zuläßt, da sich die Definitionen hierfür nicht wesentlich von dem Spezialfall einer klassischen Prädikatenlogik unterscheiden. Dabei ist uns klar, daß die Einbeziehung von Quantoren, wie wir sie auch im Fall einer Prädikatenlogik und damit in SNL erlauben, so allgemeine Spezifikationen zulassen, daß eine computergestützte Analyse oder Transformation solch allgemeiner Spezifikationen unmöglich ist. Wir werden daher in SMARAGD und auch in SNL Mechanismen zur konstruktiven Spezifikation von Datentypen einbinden, die theoretisch auf einem einfacheren gleichungsorientierten Kalkül beruhen, der eine weitgehende automatische Transformation von solchen Spezifikationen, die konform zu den Konstruktionsregeln sind, in ML-Implementierungen möglich macht.

Gleichwohl halten wir es durchaus für sinnvoll, auch allgemeine prädikatenlogische Formeln zuzulassen, wobei dann Spezifikationen, die nicht automatisch transformierbar sind, mit manueller Unterstützung implementiert werden müssen. Die Intention dabei ist, daß ein normaler Anwender von SMARAGD seine Modelle nur gemäß der angegebenen Konstruktionsregeln erstellt, die eine weitgehend automatische Transformation in Implementierungen ermöglicht, weiter aber ein Expertenmodus vorhanden ist, der es in algebraischer Spezifikation versierten Personen erlaubt, komplexere Spezifikationen, die nicht gemäß der erwähnten Konstruktionsregeln gebildet sind, anderen Anwendern innerhalb von Libraries bereitzustellen, wobei die zugehörigen Standardimplementierungen gegebenenfalls manuell erstellt werden. Man beachte dabei, daß die genaue Festlegung der oben erwähnten Konstruktionsregeln nicht Gegenstand dieser Arbeit sind, sondern zu einem späteren Zeitpunkt im Zusammenhang mit der Realisierung der Transformationskomponente spezifiziert werden. Der Leser wird daher in dieser Arbeit in Zusammenhang mit dazugehörigen SNL-Konstrukten nur einige Hinweise finden, wie solche Konstruktionsregeln aussehen könnten, aber keine vollständige Spezifikation von Ihnen finden.

Ein weiterer Grund dafür, daß wir allgemeine prädikatenlogische Formeln innerhalb von SNL zulassen ist es, daß wir, wie wir später sehen werden, innerhalb der Beschriftungen eines SNL-Petri-Netz-Modells beliebige prädikatenlogische Formeln interpretieren können und auch wollen.

Die Bauelemente von Formeln einer formalen Sprache 1. Ordnung sind zunächst Terme bestehend aus Konstanten, Variablen, Operationssymbolen und Klammern als Hilfsgrößen. Aus Termen und Prädikaten kann man dann elementare Formeln bilden. Diese lassen sich wiederum mit logischen Konnektoren der formalen Sprache (wie **andalso**, **orelse** und **not**) zu einfachen Formeln der Sprache zusammenbinden. Die Einbeziehung von Quantoren führt schließlich zu allgemeinen geschlossenen Formeln der formalen Sprache 1. Ordnung.

Zum Aufbau der Terme benötigt man in der Regel Variablen, denen eine Sorte als Typ zugeordnet ist. Der Gleichungsteil *spec_eqn_decls* besteht daher zunächst aus Variablen- und Formeldeklarationsteilen:

spec_eqn_decls: *spec_var_decls*
spec_formula_decls
...

Im folgenden Abschnitt wollen wir zunächst Variablen und Terme näher erläutern.

3.1.3.1 Variablen und Terme

Terme sind nach gewissen Konstruktionsregeln gebildete Wörter (Zeichenketten von Symbolen) über einem Alphabet, das aus Symbolen einer Signatur eines abstrakten Datentyps, zu dieser Signatur gehörenden Variablensymbolen und runden Klammern als Hilfssymbolen besteht. Die Konstruktionsregeln beschreiben dabei, wie man "syntaktisch sinnvoll" Operatoren auf Argumentterme anwenden kann, um dadurch neue Terme zu gewinnen. Die Regeln lauten: (1) Variablen- und Konstantensymbole sind bereits Terme eines definierten Typs (d.h. von einer gewissen Sorte). (2) Wenn ich einen Operator f vom Typ $typ_\Omega(f)=(w,s)$ mit $w=s_1s_2...s_k$ und eine Folge von Termen $t_1, t_2, ... t_k$ mit $typ_\Omega(t_i)=s_i$ für $i=1,2,..k$ habe, so ist die Symbolfolge $f(t_1,t_2,..,t_k)$ wiederum ein Term vom Typ s. Wir definieren daher:

Def. 2 (Terme und Typ von Termen)

Sei $SIG=(S, \Omega, \Pi, typ_\Omega, typ_\Pi)$ eine Signatur und $V_S=(V_S(s))_{s\in S}$ eine Familie von Variablen der Sorten $s\in S$. Die *Sprache der Terme* $TERM(SIG,V_S)$ der Signatur SIG mit Variablen aus der Familie V_S und der *Typ von Termen* $typ(t)$ sind wie folgt rekursiv definiert:

(1) für jedes $s\in S$ und jedes Variablensymbol $v\in V_S(s)$ ist das Wort v ein Element von $TERM(SIG,V_S)$ und es ist $typ(v)=s$,

(2) für jedes $\omega\in\Omega$ mit $typ_\Omega(\omega)=(w,s)$ und jede Folge von Termen $t_1, t_2, ..., t_k\in TERM(SIG,V_S)$ mit $typ(t_1)typ(t_2)...typ(t_k)=w$ gehört das Wort $t=\omega(t_1, t_2, ..., t_k)$ der Menge $TERM(SIG,V_S)$ an und es ist $typ(t)=s$.

Während Konstanten in ihrer Syntax bereits innerhalb der Signatur eines abstrakten Datentyps spezifiziert worden sind, müssen wir vor Bildung von Termen zunächst die hierzu verwendeten Variablen und ihren Typ spezifizieren. Dies geschieht in unserer Spezifikationssprache hinter dem Schlüsselwort **var** in Variablendeklarationsteilen. Diese haben die Gestalt:

spec_var_decls: **var** *spec_var_decl* { ; *spec_var_decl* }$_0^*$ [;]
spec_var_decl: *var_id* : *sort_name*
var_id: **id**

Die obige Definition von Termen läßt sich nun einfach auf unsere Spezifikationssprache übertragen, indem wir definieren, daß ein Term entweder ein Konstantensymbol *const_name*, ein Variablensymbol *var_id* oder ein Wort der Gestalt *opn_name*(*term_list*) ist. In der Praxis werden jedoch aus historischen Gründen einige häufig verwendete Operatoren in einer etwas anderen Schreibweise verwendet. So schreibt man zweistellige Operatoren, wie + oder -, meistens in der Infixschreibweise *term opn_name term* und einstellige Operatoren, wie das logische Nicht (hier im Zeichen !), in der Präfixschreibweise ! *term* (Man beachte das Fehlen der Klammern). Um solchen historisch bedingten Standardschreibweisen Rechnung zu tragen, erlauben wir es, daß man innerhalb einer Spezifikation die Infix- oder Präfixschreibweise für zweistellige Operatoren und Prädikate, die Assoziativität eines Infixoperators oder -prädikats und die Priorität des Operators oder Prädikats explizit angeben kann.

Als Default wird jeder definierte Operator und jedes definierte Prädikat in Präfixform verwendet, wobei einstellige Operatoren oder Prädikate in der Form *opn_name term*, *pred_name term* oder *opn_name*(*term*), *pred_name* (*term*) verwendet werden können. Bei mehreren Argumenten müssen die Argumentterme explizit in runden Klammern eingeschlossen werden, wobei die einzelnen Argumentterme durch Komma zu trennen sind.

Um einen zweistelligen Operator oder ein zweistelliges Prädikat in Infixschreibweise benutzen zu können, muß man ihn oder es als Infix in einer Infix-Deklaration deklarieren. Wir erweitern unseren Gleichungsteil daher noch um Infix-Deklarationsteile:

spec_eqn_decls:	*spec_var_decls*
	spec_formula_decls
	spec_infix_decls
	spec_infixr_decls

Die einzelnen Deklarationsteile haben die Gestalt:

spec_infix_decls:	**infix** *spec_infix_decl* { ; *spec_infix_decl* }$_0^*$ [;]
spec_infixr_decls:	**infixr** *spec_infix_decl* { ; *spec_infix_decl* }$_0^*$ [;]
spec_infix_decl:	*opn_id* [**num**]
	pred_id [**num**]

Der Unterschied zwischen *spec_infix_decls* und *spec_infixr_decls* ist nur, daß Operatoren und Prädikate im ersten Fall linksassoziativ und im zweiten Fall rechtsassoziativ sind. Das optionale **num** Argument gibt die Präzedenz des Operators oder Prädikats im Bereich von 0 bis 9 an. Fehlt es, so wird als Defaultpräzedenz 0 gewählt.

Bei der Vergabe der Präzedenz sollte man sich am folgenden Schema orientieren:

- Präfixoperatoren und -prädikate haben höchste Präzedenz (8).
- Arithmetische Infix-Operatoren mit höherer Präzedenz (wie Multiplikation * und Division /) haben die nächsthöchste Präzedenz (7).
- Arithmetische Infix-Operatoren mit geringerer Präzedenz (wie Addition + und Subtraktion -) haben Präzedenz 6.
- Infix-Vergleichsoperatoren (wie <, <=, =, >= und <>) haben Präzedenz 4.
- Der logische Operator **andalso** (logisches Und) hat die Präzedenz 2
- Der logische Operator **orelse** (logisches Oder) hat die Präzedenz 1

Die Definition eines Terms unserer Sprache wird durch folgende Grammatik bestimmt:

term:	*opn_name* (*term_list*)
	opn_name term
	term opn_name term
	const_name
	var_id
	(*term*)
term_list:	*term* { , *term* }$_0^*$

Gemäß der mathematischen Definition sind zunächst einmal alle Konstanten- und Variablensymbole (*const_name* bzw. *var_id*) Terme. Aus Konstanten und Variablen als Basistermen und den Operationssymbolen kann man nun allgemeinere Terme durch iterative Anwendung von Operationssymbolen auf Terme zusammensetzen. Dabei erlauben wir für einstellige Operatoren die Präfixschreibweise *opn_name term* (ohne Klammern), bei zweistelligen Operatoren die Infixschreibweise *term opn_name term*, sofern der Operator vorher in einer Infixdeklaration als Infix definiert ist oder für beliebige Präfixoperatoren (auch einstellige) die funktionale Schreibweise *opn_name* (*term_list*). Durch Klammerung von Termen ist es weiter möglich, die implizit durch die

Präzedenzen der verwendeten Operatoren gegebene Abarbeitungsreihenfolge von Termen zu verändern, wobei geklammerte Ausdrücke höchste Präzedenz haben.

Bei Anwendung eines Operationssymbols auf Terme muß die Anzahl der Terme mit der Stelligkeit des Operators übereinstimmen, wie dies aus der mathematischen Definition ersichtlich ist. Die Stelligkeit eines Operators ist dabei die Anzahl der Argumente, die diesem Operator bei der Definition zugeordnet wurde. Weiter muß der Typ jedes Terms der Sorte des jeweiligen Argumentes in der Definition des Operators entsprechen. Wir wollen einmal einige Beispiele für Terme betrachten.

Beispiel 14 (Beispiel für Terme)

(a) Beispiele für Terme der Signatur `BOOL_A` mit Infixoperatoren `&&` der Präzedenz 2 und `||` der Präzedenz 1 und Präfixoperator `!` wären `!(x||y&&z)` oder `x && y || !z`. Man beachte, daß bzgl. der Prioritäten diese Ausdrücke zu lesen sind als `!(x||(y&&z))` und `(x&&y)||(!z)`.

(b) Beispiele für Terme der Signatur `NAT` mit Präfixoperator `succ` und Infixoperator + sind `succ(0)+x` und `(succ x)+(succ(succ 0)+y)`.

(c) Beispiele für Terme der Signatur `STACK` sind `top(push(push(st,x),y))` und `pop(push(empty,x))`. Hierbei sind `x` und `y` Variablen vom Typ `item` und `st` vom Typ `stack`.

3.1.3.2 Atomare Formeln

Atomare Formeln (atomare Ausdrücke) definieren die Basiselemente zum Aufbau von Formeln einer Sprache 1. Ordnung. Sie ergeben sich durch Anwendung von Prädikaten eines abstrakten Datentyps auf Terme dieses Datentyps. Mathematisch läßt sich dies wie folgt formulieren:

Def. 3 (atomare Formeln)

Sei $SIG=(S, \Omega, \Pi, typ_\Omega, typ_\Pi)$ eine Signatur und $V_S=(V_S(s))_{s\in S}$ eine Familie von Variablen der Sorten $s\in S$. Die *Sprache der atomaren Formeln* $FORM(SIG,V_S)$ der Signatur *SIG* mit Variablen aus der Familie V_S ist die kleinste Sprache, die der folgenden Bedingung genügt:

für jedes Prädikat $\pi\in\Pi$ und jede Folge von Termen $(t_1, t_2, ..., t_k)$ mit $typ(t_1)...typ(t_k)=typ_\Pi(\pi)$ gehört das Wort $\pi(t_1,t_2,...,t_k)$ der Sprache $FORM(SIG,V_S)$ an.

Die Syntax zur Bildung von atomaren Formeln ist:

elementary_formula:	*pred_name* (*term_list*)
	pred_name term
	term pred_name term
term_list:	*term* { , *term* }$_0^*$

Wie bei Operatoren hat man auch bei der Anwendung von Prädikaten die Möglichkeit einer Präfixanwendung der Gestalt *pred_name term* bei einstelligen Prädikaten, der Infixschreibweise *term pred_name term* bei zweistelligen Prädikaten, die als Infix deklariert sind oder der funktionalen Anwendung *pred_name* (*term_list*), die für Prädikate mit beliebiger Stelligkeit gilt.

Wie auch bei Termen ist bei Anwendung eines Prädikates auf die Anzahl und die Typen der Argumente zu achten. Die Anzahl der Argumente und ihre Typen müssen mit der in der Definition des Prädikates gegebenen Anzahl und Typspezifikation übereinstimmen. Betrachten wir einmal ein paar Beispiele für atomare Formeln:

Beispiel 15 (Beispiele für atomare Formeln)

(a) Atomare Formeln der Signatur `BOOL_A` mit zweistelligem Prädikat =, das als Infixprädikat stets vordefiniert ist, sind unter anderem `true = false`, `true && x =true` und `x || y = ! ( !x && !y )`.

(b) Atomare Formeln zu `NAT` sind `succ(x) + y =succ (x + y)` oder `succ x < x`.

(c) Atomare Formeln für `STACK` sind `isempty(push(st,x))` oder `x = top(push(st,x))`, wobei hier `x` eine Variable vom Typ `item` und `st` eine Variable vom Typ `stack` ist.

3.1.3.3 Allgemeine Formeln

Aus atomaren Formeln kann man nun durch Verknüpfung mit "logischen" Konnektoren und Quantifizierung von Variablen durch Quantorensymbole Formeln einer formalen Sprache 1. Ordnung zusammensetzen. Mathematisch läßt sich dieser Prozeß wie folgt beschreiben:

Def. 4 (Formeln einer formalen Sprache 1. Ordnung)

Sei $SIG=(S,\Omega,\Pi,typ_\Omega,typ_\Pi)$ eine Signatur. L_1 sei eine Menge, deren Elemente wir im Kontext dieser Definition *einstellige logische Konnektoren* nennen werden. L_2 sei eine Menge, deren Elemente wir im Kontext dieser Definition *zweistellige logische Konnektoren* nennen werden. Weiter sei $V_S=(V_s)_{s\in S}$ eine Familie von paarweise zueinander disjunkten Mengen, wobei jede Menge in zwei Mengen $V_S^f(s)$ und $V_S^b(s)$ partitioniert ist. Im Kontext dieser Definition nennen wir Elemente der Menge $V_S^f(s)$ *freie Variablen der Sorte* $s\in S$ und Elemente der Menge $V_S^b(s)$ *gebundene Variablen der Sorte* $s\in S$. $Q_S=(Q_s)_{s\in S}$ sei weiter eine Familie von Mengen, deren Elemente wir im Kontext dieser Definition *Quantoren der Sorte* $s\in S$ nennen werden. Die Menge U bestehe nur aus den Symbolen '(' und ')' als Hilfszeichen.

(a) $FORM(SIG,L_1,L_2,V_S,Q_S)$ sei die kleinste Menge, die die folgenden Bedingungen erfüllt.

(1) $FORM(SIG,V_S^f) \subseteq FORM(SIG,L_1,L_2,V_S,Q_S)$.

(2) Für jeden einstelligen Konnektor $o\in L_1$ und jedes Element $\alpha\in FORM(SIG,L_1,L_2,V_S,Q_S)$ gehört das Wort $o(\alpha)$ der Menge $FORM(SIG,L_1,L_2,V_S,Q_S)$ an.

(3) Für jeden zweistelligen Konnektor $o\in L_2$ und alle $\alpha,\beta\in FORM(SIG,L_1,L_2,V_S,Q_S)$ gehört das Wort $(\alpha)o(\beta)$ der Menge $FORM(SIG,L_1,L_2,V_S,Q_S)$ an.

(4) Für jedes $s\in S$, jede freie Variable $x\in V_s^f(s)$, jede gebundene Variable $\xi\in V_s^b(s)$, jeden Quantor $q\in Q_s$ und jedes Element $\alpha\in FORM(SIG,L_1,L_2,V_S,Q_S)$ gehört das Wort $q_\xi\alpha(x/\xi)$ der Menge $FORM(SIG,L_1,L_2,V_S,Q_S)$ an. Dabei ist $\alpha(x/\xi)$ das Wort, daß aus α dadurch entsteht, daß man jedes Vorkommen der freien Variablen x durch ξ ersetzt.

(b) Die Mengenfolge $ALPH=(V_S,\Omega,\Pi,L_1,L_2,Q_S,U)$ nennt man im Kontext dieser Definition ein *Alphabet 1. Ordnung* über der Signatur SIG mit einstelligen Konnektoren aus L_1, zweistelligen Konnektoren aus L_2, Variablen aus der Familie V_S und Quantoren aus der Familie Q_S.

(c) Das Tripel $L=(ALPH,TERM(SIG,V_S),FORM(SIG,L_1,L_2,V_S,Q_S))$ wird eine *formale Sprache 1. Ordnung* genannt.

(d) Elemente der Menge $FORM(SIG,L_1,L_2,V_S,Q_S)$ nennen wir *Formeln der Sprache L.*

(e) Eine Formel $\alpha\in FORM(SIG,L_1,L_2,V_S,Q_S)$ heißt *geschlossen*, wenn sie keine freien Variablensymbole (d.h. Symbole aus den Mengen $V_S^f(s)$ für alle $s\in S$) enthält.

In SNL beschränken wir uns auf den Fall einer klassischen Prädikatenlogik 1. Ordnung, d.h. die Menge der einstelligen Konnektoren besteht bei uns nur aus einem Konnektor **not** (logisches nicht) und die Menge der zweistelligen Konnektoren aus **andalso** (logisches und), **orelse** (logisches oder) und => (logische Implikation). Weiter haben wir für jede Sorte genau zwei Quantoren **all** und **exist**, die wir als All- und Existenzquantor interpretieren werden.

Für die Anwendung der Konnektoren auf ihre Argumente verwenden wir die bekannte Präfix- bzw. Infixnotation für **not** bzw. **andalso**, **orelse** und =>.Wir geben dabei **not** die höchste Priorität(3) vor **andalso** und =>(2). **orelse** hat die niedrigste Priorität(1). Auf diese Weise werden zunächst alle Terme und dann alle atomaren Formeln gebildet, bevor diese dann durch die logischen Konnektoren verbunden werden.

Für die Anwendung der Quantoren auf eine Formel verwenden wir die Schreibweisen **all x formel** oder **exist x formel**. Da wir uns von der Theorie aus auf geschlossene Formeln beschränken, erlauben wir es, daß nicht alle Variablen explizit quantorisiert werden müssen. Vielmehr ist implizit jede Variable, die nicht durch einen Quantor gebunden ist, per Default durch einen Allquantor gebunden.

Ein Formelteil einer algebraischen Spezifikation eines abstrakten Datentyps hat daher die Gestalt:

spec_formula_decls:	**eqn** *formula* { ; *formula* }$_0^*$
formula:	**all** *var_id formula*
	exist *var_id formula*
	formula **andalso** *formula*
	formula **orelse** *formula*

formula => *formula*
not *formula*
(*formula*)
elementary_formula

Man beachte dabei, daß die Präzedenz von **all** und **exist** entsprechend der von anderen Präfixoperatoren höher als die aller Infixoperatoren ist. SNL faßt daher den nächsten SNL-Formelausdruck hinter dem Variablennamen als Formelangabe zu einem Quantor auf. Formeln als zweites Argument zu einem **all** oder **exist** Quantor müssen daher in der Regel geklammert werden. Nachdem wir nun wissen, was Formeln sind, können wir formal definieren, was wir unter einer Spezifikation verstehen.

Def. 5 (Spezifikation eines abstrakten Datentyps)

Sei $SIG=(S,\Omega,\Pi,typ_\Omega,typ_\Pi)$ eine Signatur, V_S eine Familie von Variablen zu SIG und L eine formale Sprache 1. Ordnung über SIG unter den Voraussetzungen von Def. 4. F sei weiter eine Menge von Formeln aus $FORM(SIG,L_1,L_2,V_S,Q_S)$. Ein 3-Tupel der Gestalt $SPEC=(SIG,V_S,F)$ heißt dann eine Spezifikation mit Signatur SIG, Variablen aus V_S und Formeln F der Sprache L.

Oftmals läßt man auch in der obigen Definition die Angabe der Variablen weg und schreibt einfach $SPEC=(SIG,F)$ für eine Spezifikation. In diesem Fall meint man einfach die Spezifikation mit den Variablen, die in F vorkommen. Diese Sichtweise entspricht unserer Syntax des Spezifikationsteils als Folge von *sig_part* und *eqn_part*.

Wir wollen uns nun einmal einige Beispiele für Spezifikationen abstrakter Datentypen ansehen.

Beispiel 16 (Beispiele für Spezifikationen mit Formeln in SNL)

(a) Im ersten Beispiel fügen wir nun unserer Signatur des Datentyps `BOOL_A` Formeln hinzu, die die "Semantik" von `BOOL_A` genauer festlegen. Eine mögliche Spezifikation von `BOOL_A` ist:

```
specification BOOL_A = spec
     sort      bool;
     const     true : bool;
               false : bool;
     opn       ! : bool -> bool;
               || : bool * bool -> bool;
               && : bool * bool -> bool;
     infix     || 1; && 2;
     var       x : bool;
               y : bool;
     eqn       true <> false;
               !true = false;
               !false = true;
               true && x = x;
               false && x = false;
               x || y = !(!x && !y);
end
```

Wir wollen uns die einzelnen Formeln einmal genauer ansehen. Die erste Formel drückt offensichtlich aus, daß in unserem Datentyp `BOOL_A` die beiden Konstanten `true` und `false` verschiedene Werte vom Typ `bool` sein sollen. Die zweite Formel legt fest, daß die Negation (bzgl. des Negationsoperators `!` in `BOOL_A`) von `true` den booleschen Wert `false` ergibt. Die dritte Formel legt entsprechend fest, daß die Negation von `false` der Wert `true` ist. Die Semantik der Operation `&&` wird in den beiden darauffolgenden Gleichungen festgelegt. Sie geben an, daß der Ergebniswert einer Und-Operation nur dann `true` ist, wenn beide Argumentausdrücke den Wert `true` haben. Die letzte Formel legt schließlich die Semantik der Oder-Operation fest. Die Oder-Operation wird hier in bekannter Weise durch die Und-Operation und die Negation beschrieben. Man beachte, daß in den letzten drei Formeln die Variablen `x` und `y` implizit durch den Allquantor gebunden sind!

(b) Eine Spezifikation für unseren Datentyp natürliche Zahlen könnte lauten:

```
specification NAT = spec
     sort      nat;
     const     0 : nat;
     opn       succ : nat -> nat;
               + : nat * nat -> nat;
     pred      <= : nat * nat;
     infix     + 6; <= 4;
     var       m : nat; n : nat;
     eqn       n + 0 = n;
               n + succ m = succ(n+m);
               m <= m + n;
               not (n + (succ m) <= n);
end
```

Die ersten zwei Formeln legen die Semantik der Operationen + und `succ` fest. Die letzten beiden Formeln bestimmen die Semantik der <= Relation.

(c) Betrachten wir als weiteres Beispiel unseren Typ `STACK`. Hier könnte eine Spezifikation so aussehen:

```
specification STACK = spec
     sort      stack; item;
     const     empty : stack;
     opn       push : stack * item -> stack;
               pop: stack -> stack;
               top : stack -> item;
     pred      isempty : stack;

     var       st : stack;
               x : item;
     eqn       pop(push(st,x))=st;
               top(push(st,x))=x;
               isempty(empty);
               not isempty(push(st,x));
end
```

Die erste Formel präzisiert unsere Vorstellung von einem Stack, daß, wenn man das oberste Element von einem Stack entfernt, der Stack dabei herauskommt, der bestand, bevor man das oberste Element auf den Stack gebracht hat. Dies ist gerade die "Last In First Out" Eigenschaft eines Stacks. Die Formel zwei besagt, daß, nachdem man ein Element `x` auf den Stack gebracht hat, dieses `x` als oberstes Element auf dem Stack liegt. Die beiden letzten Formeln definieren das Prädikat `isempty` nur dann als erfüllt, wenn der Wert des Stackargumentes die Konstante `empty` ist. Man beachte, daß mit diesen Gleichungen das Verhalten eines Stacks bezogen auf die Anwendung der `pop` und `top` Operationen auf die leere Warteschlange nicht festgelegt ist. Hierauf kommen wir etwas später noch zurück.

(d) Die Formeln in den obigen Beispielen sind alle implizit durch den Allquantor quantifiziert. Als ein Beispiel für einen explizit vorhandenen Existenzquantor betrachten wir einmal die Spezifikation GROUP, die die Eigenschaften einer Gruppe als Formeln formuliert.

```
specification GROUP = spec
     sort      group;
     const     e : group;
     opn       + : group * group -> group;
     infix     + 6;
     var       x : group; y : group;
     eqn       x + e = x;
               exist y (x + y = e);
end
```

Die erste Formel gibt hier an, daß e das Einselement der Gruppe ist. Die zweite Formel spezifiziert, daß es zu jedem Element x aus `group` ein inverses Element y gibt. Beide Formeln könnte man übrigens in eine Formel der Gestalt

```
all x (x+e = x andalso exist y (x+y = e))
```

zusammenfassen. Diese Formel zeigt die explizite Verwendung des Allquantors sowie die Schachtelung von quantorisierten Formeln.

3.2 Semantik des Spezifikationsteils von SNL

Ein adequates mathematisches Modell für Datentypen liefern algebraische Systeme. Ein algebraisches System über einer Signatur $SIG=(S,\Omega,\Pi,typ_\Omega,typ_\Pi)$ definiert zu jeder Sorte eine Menge von Werten dieser Sorte, zu jedem Operator eine Operation vom äquivalenten Typ und zu jedem Prädikat eine Relation von einem äquivalentem Typ. Ein algebraisches System über einer Signatur wird daher durch drei Abbildungen definiert, die Sortensymbole auf Mengen, Operatoren auf Funktionen und Prädikate auf Relationen abbildet.

Def. 6 (algebraisches System)

Ein *algebraisches System* über der Signatur $SIG=(S,\Omega,\Pi,typ_\Omega,typ_\Pi)$ ist ein Tripel $A=(A_S,A_\Omega,A_\Pi)$ mit den Eigenschaften:

(1) $A_S:S \rightarrow$ **Set**, $A_\Omega:\Omega \rightarrow$ **Func** und $A_\Pi:\Pi \rightarrow$ **Rel** sind injektive Abbildungen. Dabei fassen wir **Set**, **Func** und **Rel** als Menge aller Mengen, Menge aller Funktionen und Menge aller Relationen (über einem gewissen Universum) auf.

(2) Für jeden Operator $\omega \in \Omega$ gilt: $typ(A_\Omega(\omega))=A_S(typ_\Omega(\omega))$.

(3) Für jedes Prädikat $\pi \in \Pi$ gilt: $typ(A_\Pi(\pi))=A_S(typ_\Pi(\pi))$.

Mit $ALS(SIG)$ bezeichnen wir die Klasse aller algebraischen Systeme über einer Signatur SIG.

Bedingung (1) ordnet offenbar jeder Sorte eine Menge von "Werten dieser Sorte", jedem Operator eine Funktion und jedem Prädikat eine Relation zu. Bedingung (2) sagt, daß der Typ einer Operation des algebraischen Systems (d.h. die Folge der Argumentwerte und der Wertebereich der Funktion) zum Typ des Operators innerhalb der Signatur passen muß. Bedingung (3) gibt ein entsprechendes Konsistenzkriterium für die Relationen an.

Wie wir auch für Signaturen in unserer Spezifikationstheorie annehmen, daß wir implizit stets für jede Sorte zwei Prädikate = und <> gegeben haben, nehmen wir auch an, daß jedes algebraische System über einer Signatur in unserer Spezifikationstheorie ohne explizite Angabe die Gleichheits- und Ungleichheitsrelationen enthält. Wir haben also stets für jede Sorte $A_\Pi(=)$ als Gleichheitsrelation und $A_\Pi(<>)$ als Ungleichheitsrelation implizit gegeben.

Ein algebraische System, für das die Menge der Relationen (ohne die impliziten Prädikate = und <>) leer ist (d.h. $\Omega=\emptyset$), nennt man auch *Algebra über einer Signatur SIG*. Man kann in diesem Fall innerhalb der Signatur sowohl Ω als auch typ_Ω und im algebraischen System A_Π wegfallen lassen und einfach $SIG=(S,\Omega,typ_\Omega)$ bzw. $A=(A_S,A_\Omega)$ schreiben.

Beispiel 17 (Beispiel zu algebraischen Systemen)

(a) Wir wollen als erstes Beispiel einmal unsere Signatur `BOOL_A` betrachten. Wohl jeder Betrachter verbindet mit dieser Signatur eine boolesche Algebra. Ein typisches Beispiel ist definiert durch:

(1) A_S(`bool`)={*true*,*false*}.

(2) A_Ω(`true`)=*true* und A_Ω(`false`)=*false*

(3) A_Ω(`!`) sei definiert durch: A_Ω(`!`)(*true*)=*false* und A_Ω(`!`)(*false*)=*true*.

(4) A_Ω(`&&`) sei definiert durch: A_Ω(`&&`)(*true*,*true*)=*true* sonst *false*.

(5) A_Ω(`||`) sei definiert durch: A_Ω(`||`)(*false*,*false*)=*false* sonst *true*.

(b) Das algebraische System, das man mit der Spezifikation `NAT` typischer Weise verbindet, ist gegeben durch:

(1) A_S(`nat`) sei die Menge der natürlichen Zahlen mit 0.

(2) A_Ω(`0`) sei die natürliche Zahl 0.

(3) A_Ω(`+`) sei die Addition in den natürlichen Zahlen.

(4) A_Ω(succ) sei die Inkrementoperation, die eine natürliche Zahl um 1 erhöht.

(5) A_Π(<=) sei die übliche Kleiner-Gleich-Relation zwischen natürlichen Zahlen.

(c) Mit der Signatur STACK wollen wir einmal das folgende algebraische System definieren:

(1) A_S(item)=$\{a,b,c\}$ sei die Menge der drei Buchstaben a, b und c.

(2) A_S(stack)=$\{a,b,c\}^*$ sei die Menge aller Wörter über dem Alphabet $\{a,b,c\}$.

(3) A_S(empty) sei das leere Wort ε über dem Alphabet $\{a,b,c\}$.

(4) A_S(push)(w,s) sei für ein Wort w aus $\{a,b,c\}^*$ und ein Element s aus $\{a,b,c\}$ definiert als das Wort sw.

(5) A_S(pop)(w) mit $w=s_1...s_k$ aus $\{a,b,c\}^*$, $k>=1$ und s_i aus $\{a,b,c\}$ für i=1,..,k, sei das Wort $w'=s_2...s_k$. A_S(pop)(ε)=ε.

(6) Mit $w=s_1...s_k$ aus $\{a,b,c\}^*$ und s_i aus $\{a,b,c\}$ für i=1,..,k sei A_S(top)(w)=s_1. A_S(top)(ε) definieren wir willkürlich als a.

(7) A_Π(isempty)=$\{\varepsilon\}$.

Das oben definierte algebraische System zur Signatur STACK entspricht sicherlich unserer Vorstellung eines Stacks, wenn wir von der willkürlichen Definition von A_S(top)(ε) einmal absehen, worauf wir später noch zurückkommen. Wenn wir aber die Operation A_S(push)(w,s) für ein Wort w aus $\{a,b,c\}^*$ und ein Element s aus $\{a,b,c\}$ durch das Wort ws definieren würden, so erhielten wir ebenfalls ein algebraisches System über der Signatur STACK, das nun aber nicht mehr unserer Vorstellung von einem Stack, sondern eher unserer Vorstellung von einer FIFO (First In First Out) entspricht. Dieses Beispiel soll hier verdeutlichen, daß eine Signatur zwar die Syntax eines algebraischen Systems (Datentyps), nicht aber dessen Semantik festlegt.

Das stackähnliche Verhalten als (Last-in-First-Out-) Speicher wird erst dann erzwungen, wenn wir verlangen, daß die Formel `pop(push(st,x))=st` im unter Beispiel 17c definierten algebraischen System gelten soll. Hierzu müssen wir aber zunächst einmal sauber definieren, was es heißt, daß eine Formel in einem algebraischen System erfüllt ist. Wie schon bei der Definition von Formeln werden wir auch die Semantik von Formeln iterativ über die Begriffe Realisation von Termen, Realisation von atomaren Formeln und Realisation von Formeln aufbauen. Wir beginnen zunächst mit dem Begriff der Realisation von Termen in einem algebraischen System.

3.2.1 Realisation von Termen

Wir haben Terme als syntaktische Zeichenfolgen definiert, die die wohldefinierte Anwendung von Operatoren auf Argumentzeichenfolgen beschreiben. Wenn wir nun die Operatoren als Funktionen in einem zu einer Signatur gehörigen algebraischen System interpretieren, Konstanten als Elemente der Wertemengen dieses Systems auffassen und Variablen einen definierten Wert aus einer zugehörigen Trägermenge zuweisen, so kann man einen Term als wohldefinierte Anwendung von Funktionen auf zugehörige Argumente interpretieren. Dies führt zu der folgenden Definition:

Def. 7 (Auswertung von Termen)

Sei $SIG=(S,\Omega,\Pi,typ_\Omega,typ_\Pi)$ eine Signatur, $V_S=(V_s)_{s\in S}$ eine Familie von Variablen und $TERM(SIG,V_S)$ die zugehörige Menge der Terme. Weiter sei $A=(A_S,A_\Omega,A_\Pi)$ ein algebraisches System über der Signatur SIG.

(a) Eine *Auswertung der Variablen aus* V_S in dem algebraischen System A ist eine Familie von Abbildungen $v=(v_s:V_S(s)\rightarrow A_S(s))_{s\in S}$.

(b) $VAL(V_S,A)=VAL(V_S,A_S)=VAL(V,A)$ sei die Menge aller Auswertungen $v=(v_s:V_S(s)\rightarrow A_S(s))_{s\in S}$ von Variablen aus der Familie V_S in dem algebraischen System A über SIG.

(c) Die *Auswertung der Terme* $t\in TERM(SIG,V_S)$ in dem algebraischen System A über SIG mit Variablenauswertung v ist die eindeutig bestimmte Abbildung $\bar{v}_T:TERM(SIG,V_S)\rightarrow\cup_{s\in S}A_S(s)$, die wie folgt definiert ist:

(1) für $t=x\in V_S(s)$ ist $\bar{v}_T(t)=v_s(t)$,

(2) für einen Term $t=\omega(t_1,t_2,...,t_k)$ ist $\bar{v}_T(t)=A_\Omega(\omega)(\bar{v}_T(t_1),\bar{v}_T(t_2),...,\bar{v}_T(t_k))$.

Eine Auswertung von Variablen ist damit offensichtlich einfach eine Zuweisung von Werten an die Variablen. Jeder Variablen x vom Typ s wird ein Wert $v_s(x)$ zugeordnet. Die Auswertung von Termen ergibt sich nun aus der Auswertung von Variablen in kanonischer Form, wenn wir die Operatoren als die zugehörigen Funktionen in dem algebraischen System A über SIG interpretieren. Konstantensymbole werden hierbei als Werte innerhalb ei-

ner Trägermenge des algebraischen Systems interpretiert. Die Anwendung eines Operators auf Konstantensymbole und Variablen vom entsprechenden Typ ergibt nach der Interpretation der Konstantensymbole und Variablen als Werte von Trägermengen und des Operators als Funktion auf den zugehörigen Trägermengen wiederum einen Wert aus der zum Ergebnistyp des Operators gehörigen Trägermenge. Iterativ erhält man somit bei einer Auswertung von Termen ein Element der Trägermenge des algebraischen Systems, die zu dem Typ des Terms gehört.

Man beachte dabei, daß bei gegebener Auswertung der Variablen die Auswertung der Terme bzgl. eines algebraischen Systems A schon eindeutig bestimmt ist. Allgemeiner bestimmt die obige Konstruktion eindeutig eine Abbildung, die jedem Term $t \in TERM(SIG,V_S)$ eine Abbildung $t_R:VAL(V_S,A) \rightarrow \cup_{s\in S}A_S(s)$ zuordnet, d.h.: für jede Auswertung $v=(v_s:V_S(s) \rightarrow A_S(s))_{s\in S}$ von Variablen in A ist $t_R(v)$ eine Auswertung des Terms t. Die soeben eingeführte Abbildung nennt man die *Realisation der Terme* $t \in TERM(SIG,V_S)$ *in* A.

Def. 8 (Realisation von Termen)

Sei $SIG=(S,\Omega,\Pi,typ_\Omega,typ_\Pi)$ eine Signatur, $V_S=(V_s)_{s\in S}$ eine Familie von Variablen und $TERM(SIG,V_S)$ die zugehörige Menge der Terme. Weiter sei $A=(A_S,A_\Omega,A_\Pi)$ ein algebraisches System über der Signatur SIG. Die durch A bestimmte *Realisation der Terme* $t \in TERM(SIG,V_S)$ ist eine Abbildung $R_T:TERM(SIG,V_S)\text{->}[VAL(V_S,A) \rightarrow \cup_{s\in S}A_S(s)]$, die durch folgende Bedingung bestimmt ist:

> Für jeden Term $t \in TERM(SIG,V_S)$ ist $R_T(t)=t_R:VAL(V_S,A) \rightarrow \cup_{s\in S}A_S(s)$ definiert durch $t_R(v)=\bar{v}_T(t)$ für alle $v \in VAL(V_S,A)$.

Wir wollen einmal ein Beispiel zur Auswertung und Realisation von Termen betrachten:

Beispiel 18 (Beispiele zur Auswertung von Termen)

Betrachten wir als Beispiel wieder die Signatur `BOOL_A` und die Algebra A aus dem Beispiel 17a. Den Term `t=!((x||y)&&z)` aus Beispiel 14a können wir dann mit der Variablenauswertung `x->`*true*, `y->`*false* und `z->`*false* wie folgt auswerten.

$$\begin{aligned}\bar{v}_T(\texttt{t}) &= A_\Omega(\texttt{!})\,(\,(\textit{true}\ A_\Omega(\texttt{||})\ \textit{false})\ A_\Omega(\texttt{\&\&})\ \textit{false}) \\ &= A_\Omega(\texttt{!})\,(\textit{true}\ A_\Omega(\texttt{\&\&})\ \textit{false}) \\ &= A_\Omega(\texttt{!})\,(\textit{false}) = \textit{true}\end{aligned}$$

3.2.2 Realisation von Formeln

Bevor wir die Auswertung von atomaren Formeln definieren können, müssen wir uns zunächst einmal überlegen, welche möglichen "Werte" eine atomare Formel annehmen kann. Wie aus der allgemeinen Definition von Formeln ersichtlich ist, stellen die atomaren Formeln die Grundbauelemente dar, die durch logische Konnektoren und Quantorisierung zu allgemeinen Formeln zusammengesetzt werden. Ihr "Wert" sollte daher ein "logischer Wert" in dem Sinne sein, daß sie Elemente der Trägermenge einer Algebra sind, die die Semantik der logischen Konnektoren der formalen Sprache definiert. Dies führt zu der folgenden Definition:

Def. 9 (logische Matrizen)

Sei $LSIG=(\bar{S}=\{\bar{s}\},\bar{\Omega}=L_0\cup L_1\cup L_2,typ_{\bar{\Omega}})$ eine Signatur mit $L_0=\{\mathsf{v},e_1,...,e_k\}$, $L_1=\{o^1,...,o^l\}$, $L_2=\{o_1,...,o_m,\text{->}\}$ und $typ_{\bar{\Omega}}(\mathsf{v})=(\varepsilon,\bar{s})$, $typ_{\bar{\Omega}}(e_i)=(\varepsilon,\bar{s})$ für alle $i=1,..,k$, $typ_{\bar{\Omega}}(o^j)=(\bar{s},\bar{s})$ für alle $i=1,..,l$, $typ_{\bar{\Omega}}(o_i)=(\overline{ss},\bar{s})$ für alle $i=1,..,m$ und $typ_{\bar{\Omega}}(\text{->})=(\overline{ss},\bar{s})$. Eine Algebra $\bar{A}=(A_{\bar{S}},A_{\bar{\Omega}})$ über der Signatur $LSIG$ heißt *logische Matrix*, wenn die Operation $A_{\bar{\Omega}}(\text{->})$ für alle $a,b,c \in A_{\bar{S}}(\bar{s})$ die folgenden Bedingungen erfüllt:

(1) $a\ A_{\bar{\Omega}}(\text{->})\ a = A_{\bar{S}}(\mathsf{v})$

(2) $(a\ A_{\bar{\Omega}}(\text{->})\ b = A_{\bar{S}}(\mathsf{v})) \wedge (b\ A_{\bar{\Omega}}(\text{->})\ c = A_{\bar{S}}(\mathsf{v})) \Rightarrow (a\ A_{\bar{\Omega}}(\text{->})\ c = A_{\bar{S}}(\mathsf{v}))$

(3) $(a\ A_{\bar{\Omega}}(\text{->})\ b = A_{\bar{S}}(\mathsf{v})) \wedge (b\ A_{\bar{\Omega}}(\text{->})\ a = A_{\bar{S}}(\mathsf{v})) \Rightarrow a=b$

(4) $a\ A_{\bar{\Omega}}(\text{->})\ A_{\bar{S}}(\mathsf{v}) = A_{\bar{S}}(\mathsf{v})$

Im folgenden werden wir mit $LSIG$ stets eine Signatur, wie sie unter Def. 9 definiert ist, bezeichnen. Diese enthält stets nur endlich viele null-, ein- und zweistellige Operatoren, wobei ein nullstelliger Operator (mit Zeichem v und als Wahrheitssymbol bezeichnet) und ein zweistelliger Operator (mit Zeichem -> und Implikationssymbol genannt) ausgezeichnet sind. $\bar{A}$ sei im folgenden stets eine logische Matrix zu der Signatur $LSIG$.

Wir nennen dann die zu v in $\bar{A}$ gehörende Konstante den Wert wahr (v von verum) und die zu -> gehörende Funktion Implikation. Die einstelligen Operatoren von $LSIG$ bezeichnen wir als einstellige logische Konnektoren und die zweistelligen Operatoren als zweistellige logische Konnektoren. Nullstellige Operatoren nennen wir logische

Wertesymbole und die zugehörigen Elemente der Trägermenge von $\overline{A}$ nennen wir logische Werte der logischen Matrix $\overline{A}$.

Wenn wir eine zweistellige Relation $\neg$ auf $A_{\overline{S}}(\overline{s})$ durch $a \neg b$ genau dann, wenn $a\ A_{\overline{\Omega}}(->)\ b = A_{\overline{S}}(\vee)$ ist, definieren, stellt die Bedingung (1) von Def. 9 die Reflexivität, die Bedingung (2) die Transitivität und die Bedingung (3) die Antisymmetrie dieser Relation dar. $\neg$ definiert damit eine Halbordnung auf $A_{\overline{S}}(\overline{s})$. Immer wenn wir über Ordnungseigenschaften (z.B. Infimum und Supremum) auf der Menge $A_{\overline{S}}(\overline{s})$ sprechen, setzen wir diese Ordnung voraus. Bedingung (4) garantiert uns übrigens dabei, daß jedes Element von $A_{\overline{S}}(\overline{s})$ mit $\vee$ bzgl.$\neg$ vergleichbar ist.

In unserer Definition einer formalen Sprache 1. Ordnung treten auch einstellige und zweistellige logische Konnektoren auf. Es dürfte klar sein, daß man diese als entsprechende Elemente einer Signatur *LSIG* der obigen Gestalt auffassen kann. Im folgenden fassen wir daher eine formale Sprache 1. Ordnung als Sprache über einer Signatur *SIG* und einer Signatur *LSIG* (d.h. die ein- und zweistelligen logischen Konnektoren der formalen Sprache sind die ein- und zweistelligen Operatoren aus *LSIG*) auf.

Es liegt dann nahe, daß die "logischen Werte" der Formeln unserer formalen Sprache Elemente der Trägermenge einer zu *LSIG* gehörenden logischen Matrix sind. Eine Auswertung einer Formel ist insbesondere dann gültig, wenn sie zu dem Wert $A_{\overline{\Omega}}(\vee)$ (wahr) ausgewertet wird. Die logischen Konnektoren kann man weiter als die entsprechenden Operationen in einer zu *LSIG* gehörenden logischen Matrix interpretieren. Zur Verdeutlichung betrachten wir einmal den Fall einer klassischen Logik. In diesem Fall wird eine für die Realisierung der Formeln der Sprache zu verwendende logische Matrix eine boolesche Algebra sein. Wir wollen hier einmal die für unsere Spezifikation günstigste Form so einer klassischen booleschen Matrix vorstellen.

Beispiel 19 (Klassische boolesche Matrix)

Für die Signatur $LBOOL=(\overline{S}=\{bool\},\overline{\Omega}=\{true,false,not,and,or,=>\},typ_{\overline{\Omega}})$ sei $typ_{\overline{\Omega}}(true)=(\varepsilon,bool)$, $typ_{\overline{\Omega}}(false)=(\varepsilon,bool)$, $typ_{\overline{\Omega}}(not)=(bool,bool)$ und $typ_{\overline{\Omega}}(and)=typ_{\overline{\Omega}}(or)=typ_{\overline{\Omega}}(=>)=(bool\ bool,bool)$. Die Algebra $A\text{-}BOOL=(A_{\overline{S}},A_{\overline{\Omega}})$ sei definiert durch:

(1) Die Trägermenge der Algebra ist gegeben durch $A_{\overline{S}}(bool)=\{1,0\}$.

(2) $A_{\overline{\Omega}}(true)=1$ und $A_{\overline{\Omega}}(false)=0$.

(3) Die Funktion $A_{\overline{\Omega}}(not)$ ist gegeben durch $A_{\overline{\Omega}}(not)(1)=0$ und $A_{\overline{\Omega}}(not)(0)=1$.

(4) Die Funktion $A_{\overline{\Omega}}(and)$ ist gegeben durch $A_{\overline{\Omega}}(and)(1,1)=1$ sonst 0.

(5) Die Funktion $A_{\overline{\Omega}}(or)$ ist gegeben durch $A_{\overline{\Omega}}(or)(0,0)=0$ sonst 1.

(6) Die Funktion $A_{\overline{\Omega}}(=>)$ ist gegeben durch $A_{\overline{\Omega}}(=>)(1,0)=0$ sonst 1.

A-BOOL ist dann eine logische Matrix, die wir *klassische boolesche Matrix* nennen wollen. Wir werden sie zur Auswertung der Formeln unserer Spezifikationssprache verwenden.

Im Fall der booleschen Matrix haben wir nur zwei Wahrheitswerte, die wir gemäß der Signatur mit *true* und *false* bezeichnen. *true* ist dabei der ausgezeichnete Wert für wahr. Prädikate können wir nun in diesem Fall (im Fall einer zweiwertigen Logik) als Relationen eines zugehörigen algebraischen Systems auffassen, wobei wir bei einer Realisation von Formeln einer klassischen Prädikatenlogik einer atomaren Formel den Wert 1, wenn die zugehörige Relation bzgl. der Argumentwerte "erfüllt" ist, und sonst 0 zuordnen. Im Fall einer mehrwertigen Logik (mehr als zwei Wahrheitswerte) funktioniert dies leider nicht. Nun kann man aber jede Relation $\rho=A_{\Pi}(\pi)$ in äquivalenter Weise durch ihre charakteristische Funktion $\rho_{\pi}: A_S^{typ_{\Pi}(\pi)} -> \{0, 1\}$, $\rho_{\pi}(a)=1$ falls $a \in \rho$ sonst 0, beschreiben. Diese Interpretation von Prädikaten durch charakteristische Funktionen (anstatt von Relationen) läßt sich nun in natürlicher Weise auf mehrwertige Logiken übertragen, wenn wir die Menge $\{0,1\}$ durch die Menge der "logischen Werte" einer mehrwertigen Logik ersetzen. Wir definieren daher:

Def. 10 (Realisation von atomaren Formeln)

Seien *SIG*, *LSIG* und V_S wie in vorherigen Definitionen gegeben. $A=(A_S,A_\Omega)$ sei eine Algebra über der Signatur *SIG*, und $\overline{A}=(A_{\overline{S}},A_{\overline{\Omega}})$ sei eine logische Matrix zu der Signatur *LSIG*. $\rho_\Pi=(\rho_\pi: A_S^{typ_\Pi(\pi)} \to A_{\overline{S}}(\overline{s}))_{\pi\in\Pi}$ sei eine Familie von Funktionen.

(a) Für jede Auswertung von Variablen $v=(v_s:V_S(s)\to A_S(s))_{s\in S}$ ist dann die *Auswertung von atomaren Formeln* $\overline{v}_F:FORM(SIG,V_S)\to A_{\overline{S}}(\overline{s})$ durch die folgende Regel definiert:

(1) Ist $\alpha=\pi(t_1,t_2,..,t_k)$ mit $\pi\in\Pi$, $t_1,t_2,...,t_k\in TERM(SIG,V_S)$ und $typ_\Pi(\pi)=typ(t_1)...typ(t_k)$, so ist $\overline{v}_F(\alpha)=\rho_\pi(\overline{v}_T(t_1),...,\overline{v}_T(t_k))$, wobei $\overline{v}_T$ die durch v eindeutig bestimmte Auswertung der Terme aus $TERM(SIG,V_S)$ ist.

(b) Die durch die Algebren A und $\overline{A}$ und die Abbildungsfamilie ρ_Π eindeutig bestimmte *Realisation der atomaren Formeln* ist die Abbildung

$$\mathbf{R}_{EF}:FORM(SIG,V_S)\to[VAL(V_S,A)\to A_{\overline{S}}(\overline{s})],$$

die für jede Formel $\alpha\in FORM(SIG,V_S)$ durch $\mathbf{R}_{EF}(\alpha)=\alpha_{\mathbf{R}}$ mit $\alpha_{\mathbf{R}}(v)=\overline{v}_F(\alpha)$ für jede Auswertung $v\in VAL(V_S,A)$ definiert ist.

Wenn man die Algebren A und $\overline{A}$ sowie die Familie von Funktionen ρ_Π gegeben hat, ist die Realisation von atomaren Formeln über der Signatur *LSIG* eindeutig bestimmt. Im Fall einer zweiwertigen Logik kann man darüberhinaus die Funktionen der Familie ρ_Π nach dem oben gesagten als charakteristische Funktionen von Relationen eines algebraischen Systems A auffassen. Aus dieser Sicht ist damit die Realisation von atomaren Formeln über einer Signatur *SIG* schon durch das algebraische System A und die boolesche Matrix $\overline{A}$ bestimmt. Wenn wir uns auf eine klassische Logik beschränken und voraussetzen, daß als boolesche Matrix für die Interpretation der Logik stets *A-BOOL* aus Beispiel 19 verwendet wird, so ist unter dieser Bedingung eine Realisation der atomaren Formeln im Rahmen der klassischen Prädikatenlogik schon eindeutig durch die Angabe eines algebraischen Systems A über *SIG* bestimmt. Dies führt zu der folgenden Definition und Bemerkung.

Def. 11 (Realisation der atomaren Formeln einer klassischen Prädikatenlogik)

Sei L eine formale Sprache 1. Ordnung über einer Signatur *SIG* mit Konnektoren aus *LBOOL* (d.h. mit Konnektoren einer klassischen Prädikatenlogik). Jedes algebraische System $A=(A_S,A_\Omega,A_\Pi)$ mit Signatur SIG bestimmt dann eine eindeutig bestimmte Realisation der atomaren Formeln von L, wenn wir verlangen:

(1) Die Familie $\rho_\Pi=(\rho_\pi: A_S^{typ_\Pi(\pi)} \to \{0,1\})_{\pi\in\Pi}$ sei die Familie der charakteristischen Funktionen der Relationen von A, d.h. für $\pi\in\Pi$ ist ρ_π definiert durch: $\rho_\pi(a)=1$ genau dann, wenn $a\in A_\Pi(\pi)$ ist, sonst 0.

(2) Die zu *LBOOL* gehörige logische Matrix sei stets *A-BOOL* aus dem Beispiel 19.

Im Rahmen der klassischen Prädikatenlogik (insbesondere im Rahmen von SNL) verwenden wir stets die durch (1) und (2) bestimmte Realisation von Formeln innerhalb eines algebraischen Systems A und sprechen kurz von einer Realisation der atomaren Formeln in A.

Wir wollen einmal ein Beispiel zur Auswertung von atomaren Formeln einer prädikatenlogischen Sprache ansehen:

Beispiel 20 (Beispiel zur Auswertung atomarer Formeln)

Betrachten wir als Beispiel wieder die Signatur `BOOL_A` mit der Algebra A aus Beispiel 17a. Da wir implizit stets das Gleichheitsprädikat = gegeben haben, können wir atomare Formeln α der Gestalt `x || y = !( !x && !y )` definieren. Wenn wir nun die Auswertung von Variablen der Gestalt `x->`*true*, `y->`*false* betrachten, so ergibt eine Auswertung der obigen atomaren Formel als klassische prädikatenlogische Formel in *A-BOOL*:

$$\begin{aligned}
\overline{v}_F(\alpha) &= \overline{v}_T(\texttt{x || y})\ \rho_= \ \overline{v}_T(\texttt{!(!x \&\& !y)})\\
&= (\mathit{true}\ A_\Omega(\texttt{||})\ \mathit{false})\ \rho_=\ (A_\Omega(\texttt{!})\,(A_\Omega(\texttt{!})\,(\mathit{true})\ A_\Omega(\texttt{\&\&})\ A_\Omega(\texttt{!})\,(\mathit{false}))\\
&= \mathit{true}\ \rho_=\ (A_\Omega(\texttt{!})\,(\mathit{false}\ A_\Omega(\texttt{\&\&})\ \mathit{true}))\\
&= \mathit{true}\ \rho_=\ (A_\Omega(\texttt{!})\,(\mathit{false}))\\
&= \mathit{true}\ \rho_=\ \mathit{true}\\
&= 1
\end{aligned}$$

Die atomare Formel ist also bei der gegebenen Auswertung von Variablen wahr. Sie ist sogar bei allen möglichen Auswertungen von Variablen in A wahr, wie man leicht durch Einsetzen der anderen möglichen Auswertungen nachweisen kann.

Wir haben jetzt alle Bauelemente in der Hand, um den Begriff der Realisation von Formeln einer formalen Sprache 1. Ordnung anzugeben, wobei wir uns allerdings im folgenden stets auf Sprachen beschränken, die für jede Sorte nur zwei Quantorsymbole ($\exists$ und $\forall$) besitzen. Diese Quantoren werden wir dabei als Existenz- und Allquantor interpretieren.

Def. 12 (Realisation der Formeln einer formalen Sprache 1. Ordnung)

Seien *SIG*, *LSIG*, V_S wie in den anderen Definitionen gegeben. L sei eine formale Sprache 1. Ordnung über *SIG* und *LSIG*, wobei Q_s für jede Sorte $s \in S$ nur die Quantoren $\exists$ und $\forall$ enthält. $A=(A_S,A_\Omega)$ sei eine Algebra über der Signatur *SIG*, und $\overline{A}=(A_{\overline{S}},A_{\overline{\Omega}})$ sei eine logische Matrix zu der Signatur *LSIG*. Weiter sei $\rho_\Pi = (\rho_\pi : A_S^{typ_\Pi(\pi)} \to A_{\overline{S}}(\overline{s}))_{\pi \in \Pi}$ eine Familie von Funktionen,

$\mathbf{R}_T$:$TERM(SIG,V_S) \to [VAL(V_S,A) \to \cup_{s \in S} A_S(s)]$ die durch A eindeutig bestimmte Realisation von Termen und $\mathbf{R}_{EF}$:$FORM(SIG,V_S) \to [VAL(V_S,A) \to A_{\overline{S}}(\overline{s})]$ die durch A, $\overline{A}$ und ρ_Π eindeutig bestimmte Realisation von atomaren Formeln.

Die durch A, $\overline{A}$ und ρ_Π eindeutig bestimmte *Realisation der Formeln der Sprache L* ist die Abbildung $\mathbf{R}_F$:$FORM(SIG, LSIG, V_S, Q_S) \to [VAL(V_S,A) \to A_{\overline{S}}(\overline{s})]$, die wie folgt rekursiv definiert ist:

Sei im folgenden $\alpha \in FORM(SIG, LSIG, V_S, Q_S)$ eine beliebige aber feste Formel und $\nu=(\nu_s : V_S(s) \to A_S(s))_{s \in S}$ eine beliebige Auswertung von Variablen. Dann ist $\mathbf{R}_F(\alpha)=\alpha_\mathbf{R}$:$VAL(V_S,A) \to A_{\overline{S}}(\overline{s})$ definiert durch:

(1) Wenn $\alpha=\pi \in FORM(SIG,V_S)$ eine atomare Formel ist, dann ist $\alpha_\mathbf{R}=\mathbf{R}_{EF}(\alpha)$.

(2) Wenn α eine Formel der Gestalt $o^i(\beta)$ $(i \le l)$ mit $o^i \in L_l$ ist, und der Wert $\beta_\mathbf{R}(\nu)$ definiert ist, dann ist $\alpha_\mathbf{R}(\nu)=A_{\overline{\Omega}}(o^i)(\beta_\mathbf{R}(\nu))$.

(3) Wenn α eine Formel der Gestalt $(\beta)o_i(\gamma)$ $(i \le m)$ mit $o_i \in L_2$ ist, und die Werte $\beta_\mathbf{R}(\nu)$ und $\gamma_\mathbf{R}(\nu)$ definiert sind, dann ist $\alpha_\mathbf{R}(\nu)=A_{\overline{\Omega}}(o_i)(\beta_\mathbf{R}(\nu),\gamma_\mathbf{R}(\nu))$.

(4) Wenn α eine Formel der Gestalt $\forall_\zeta \beta$ ist, so gilt:

  (1) wenn die gebundene Variable $\zeta \in V_S^b(s)$ in der Formel β nicht vorkommt, so ist $(\forall_\zeta \beta)(\nu)=\beta_\mathbf{R}(\nu)$;

  (2) wenn die Variable $\zeta \in V_S^b(s)$ in der Formel β vorkommt, dann ist $(\forall_\zeta \beta)(\nu)=inf_{a \in A_S(s)}\{\beta_\mathbf{R}(\zeta/a)\}(\nu)$, wobei $\beta(\zeta/a)$ eine Formel ist, die aus β durch die Substitution $\zeta \to a$ mit $a \in A_S(s)$ entsteht, und *inf* die Operation der unteren Schranke in der Algebra $\overline{A}$ ist.

(5) Wenn α eine Formel der Gestalt $\exists_\zeta \beta$ ist, so gilt:

  (1) wenn die gebundene Variable $\zeta \in V_S^b(s)$ in der Formel β nicht vorkommt, so ist $(\exists_\zeta \beta)(\nu)=\beta_\mathbf{R}(\nu)$;

  (2) wenn die Variable $\zeta \in V_S^b(s)$ in der Formel β vorkommt, dann ist $(\exists_\zeta \beta)(\nu)=sup_{a \in A_S(s)}\{\beta_\mathbf{R}(\zeta/a)\}(\nu)$, wobei $\beta(\zeta/a)$ eine Formel ist, die aus β durch die Substitution $\zeta \to a$ mit $a \in A_S(s)$ entsteht, und *sup* die Operation der oberen Schranke in der Algebra $\overline{A}$ ist.

Die Operationen Infimum und Supremum sind hier als Operationen bzgl. der Halbordnung auf $\overline{A}$ zu verstehen, die wir als $\neg$ hinter der Definition einer logischen Matrix definiert hatten. Dies ist eine Ordnungsrelation, die sich in natürlicher Weise aus der Implikation in $\overline{A}$ ergibt.

Die Realisationen der Terme und Formeln einer formalen Sprache 1. Ordnung bilden die Realisation der Sprache.

Def. 13 (Realisation einer Sprache 1. Ordnung)

Seien L, $\mathbf{R}_\mathbf{T}$ und $\mathbf{R}_\mathbf{F}$ wie in Def. 12 gegeben. Das Paar $\mathbf{R}=(\mathbf{R}_\mathbf{T}, \mathbf{R}_\mathbf{F})$ nennen wir dann eine *Realisation der Sprache L*.

Da aus dem Kontext stets ersichtlich ist, welche der Abbildungen $\mathbf{R}_\mathbf{T}$ bzw. $\mathbf{R}_\mathbf{F}$ anzuwenden ist, werden wir im folgenden kurz $\mathbf{R}$ für $\mathbf{R}_\mathbf{T}$ und $\mathbf{R}_\mathbf{F}$ schreiben.

Wenn wir eine Sprache L über der Signatur *LBOOL* haben, können wir diese in Bezug auf die klassische boolesche Matrix *A-BOOL* realisieren. Eine Sprache über der Signatur *LBOOL* mit so einer Realisierung nennen wir dann eine klassische Prädikatenlogik.

Def. 14 (Klassische Prädikatenlogik 1. Ordnung)

Sei L eine formale Sprache 1. Ordnung über den Signaturen *SIG* und *LBOOL*. A sei ein algebraisches System über *SIG*. Dann definiert A eindeutig eine Realisation **R** *von L*, für die gilt:

(1) Die für die Realisation verwendete logische Matrix ist *A-BOOL*.

(2) Die Realisation der atomaren Formeln ist gemäß Def. 11 über die charakteristischen Funktionen der Relationen von A realisiert.

Eine Sprache L mit so einer Realisierung nennen wir eine *klassische Prädikatenlogik 1. Ordnung*.

Die Formeln in SNL-Spezifikationen sind stets geschlossene Formeln einer Prädikatenlogik 1. Ordnung, d.h. wir betrachten nur Realisierungen unserer Formeln gemäß der Def. 14. Die Semantik von Spezifikationen basiert nun auf algebraischen Systemen zu der zugehörigen Signatur, bzgl. denen die eindeutig bestimmte Realisation der Formeln innerhalb so eines algebraischen Systems nur gültige Aussagen liefert. Um dies genauer zu formulieren, definieren wir zunächst, wann eine Formel bei gegebener Realisation erfüllt ist.

Def. 15 (Erfüllbarkeit von Formeln)

Seien die Voraussetzungen wie in Def. 12 gegeben.

(a) Eine Realisation **R** *erfüllt* die Formel $\alpha \in FORM(SIG,LSIG,V_S,Q_S)$ *vermöge* einer Auswertung $v=(v_s:V_S(s)\text{->}A_S(s))_{s\in S}$ von Variablen, wenn $\alpha_{\mathbf{R}}(v)=v$ gilt.

(b) Die Formel $\alpha \in FORM(SIG,LSIG,V_S,Q_S)$ heißt *erfüllbar* in der Realisation **R**, wenn es eine Auswertung $v=(v_s:V_S(s)\text{->}A_S(s))_{s\in S}$ von Variablen gibt, die α in der Realisation **R** erfüllt.

(c) Eine Realisation **R** *erfüllt* die Formel $\alpha \in FORM(SIG,LSIG,V_S,Q_S)$, wenn für alle Auswertungen $v=(v_s:V_S(s)\text{->}A_S(s))_{s\in S}$ von Variablen $\alpha_{\mathbf{R}}(v)=v$ gilt. **R** nennt man dann auch ein *L-Modell* (oder kurz *Modell*) der Formel α.

(d) Eine Realisation **R** *erfüllt* eine Menge von Formeln $F \in FORM(SIG,LSIG,V_S,Q_S)$, wenn sie jede Formel $\alpha \in F$ erfüllt. **R** nennt man dann auch ein *L-Modell* (oder kurz *Modell*) der Formelmenge F.

Man beachte, daß bei einer geschlossenen Formel das Erfülltsein der Formel nicht mehr von einer Auswertung abhängt, da die Formel ja keine freien Variablen enthält.

Im Fall einer klassischen Prädikatenlogik hängt die Realisation der Sprache nur von dem gewählten algebraischen System über der Signatur *SIG* ab. Man kann daher in diesem Falle davon sprechen, daß ein algebraisches System eine Formel oder eine Menge von Formeln erfüllt.

Def. 16 (Algebraische Systeme als Modelle von Formeln)

Sei L eine prädikatenlogische Sprache über den Signaturen *SIG* und *LSIG* und A ein algebraisches System über *SIG* und **R** die Realisation von L in A.

(a) Das algebraische System A *erfüllt* die Formel $\alpha \in FORM(SIG,LSIG,V_S,Q_S)$ *vermöge* einer Auswertung $v=(v_s:V_S(s)\text{->}A_S(s))_{s\in S}$ von Variablen, wenn $\alpha_{\mathbf{R}}(v)=v$ gilt. Wir schreiben dies im Zeichen $A \models_v \alpha$.

(b) Die Formel $\alpha \in FORM(SIG,LSIG,V_S,Q_S)$ heißt *erfüllbar* im algebraischen System A, wenn es eine Auswertung $v=(v_s:V_S(s)\text{->}A_S(s))_{s\in S}$ von Variablen gibt, so daß A α vermöge v erfüllt.

(c) Das algebraische System A *erfüllt* die Formel $\alpha \in FORM(SIG,LSIG,V_S,Q_S)$, wenn für alle Auswertungen $v=(v_s:V_S(s)\text{->}A_S(s))_{s\in S}$ von Variablen $\alpha_{\mathbf{R}}(v)=v$ gilt. A nennt man dann auch ein *L-Modell* (oder kurz *Modell*) der Formel α. Wir schreiben hierfür $A \models \alpha$.

(d) Das algebraische System A *erfüllt* eine Menge von Formeln $F \in FORM(SIG,LSIG,V_S,Q_S)$, wenn sie jede Formel $\alpha \in F$ erfüllt. A nennt man dann auch ein *L-Modell* (oder kurz *Modell*) der Formelmenge F. Wir schreiben hierfür $A \models F$.

(e) Sei $SPEC=(SIG,F)$ eine Spezifikation mit Formeln aus L. Das algebraische System A *erfüllt* die Spezifikation *SPEC*, wenn A die Formelmenge F erfüllt. A nennen wir in diesem Falle ein *Modell der Spezifikation SPEC*.

Wenn wir von einem Modell einer Spezifikation $SPEC=(SIG,F)$ bzgl. der Spezifikationstheorie, die unserer Sprache SNL zu Grunde liegt, sprechen, verstehen wir gemäß der obigen Definition darunter stets ein algebra-

isches System A, das die Menge der Formeln F der Spezifikation erfüllt. Die Klasse aller SIG-Algebren, die die Spezifikation erfüllen, bezeichnen wir dann als den zu $SPEC$ gehörigen *abstrakten Datentyp.*

Def. 17 (Begriff des abstrakten Datentyps)

Sei $SPEC=(SIG,V_S,F)$ eine Spezifikation. Die Klasse aller algebraischen Systeme über SIG, die die Spezifikation erfüllen, heißt der durch $SPEC$ *spezifizierte abstrakte Datentyp.* Wir schreiben für diese Klasse $MOD(SPEC)$. Es gilt also:

$$MOD(SPEC) := \{ A \in ALS(SIG) \mid A \models F \}$$

Für die praktische Anwendung innerhalb von SNL ist der bisher eingeführte Begriff eines Modells einer Spezifikation zu allgemein. Er läßt insbesondere Wertebereiche für Modelle zu, die unendlich und damit nicht computergestützt realisierbar sind. Wir beschränken uns daher in SNL auf Modelle, die *termerzeugt* sind, d.h. alle Elemente der Wertebereiche eines solchen Modells können aus den Konstanten, die die Realisierungen der Konstantensymbole der Spezifikation sind, durch endlich viele Anwendungen von Operationen des algebraischen Systems erzeugt werden.

Anders als in unserem Ansatz wird häufig mit der Semantik eines Typs, der durch eine Spezifikation beschrieben wird, nicht ein beliebiges sondern ein durch bestimmte Eigenschaften (z.B. initial oder terminal) gekennzeichnetes Modell verbunden. Man vergleiche hierzu z.B. [Wirsing83]. Wir bevorzugen hier aber unsere Art der Definition, da sie näher an der intuitiven Vorstellung eines zu einer Spezifikation gehörigen Typs ist.

3.3 Erweiterungen des Spezifikationsteils von SNL

Die im folgenden vorgestellten Erweiterungen von SNL dienen im wesentlichen dazu, Spezifikationen komfortabler und flexibler anzugeben bzw. Mechanismen zur konstruktiven Entwicklung von Spezifikationen bereitzustellen. Viele der angegebenen Erweiterungen stellen dabei keine Erweiterung der zugrunde liegenden Theorie dar, sondern lassen sich vielmehr im Rahmen der bisher entwickelten Theorie und SNL-Syntax durch Transformationsregeln beschreiben.

Die Einführung von partiellen Operatoren stellt hier eine Ausnahme dar, da man für sie die meisten der bislang vorgestellten theoretischen Konzepte im Hinblick auf die mögliche Partialität der Operatoren verändern müßte. Da die Bedeutung von partiellen Operatoren aber intuitiv leicht verständlich ist und die oben erwähnten Erweiterungen der Theorie im wesentlichen in kanonischer Form vorgenommen werden können, verzichten wir hier auf eine theoretische Beschreibung der Theorie von Spezifikationen mit partiellen Operatoren und werden die Konzepte von partiellen Operatoren nur informell vorstellen.

Interessierte Leser können die nötigen Erweiterungen der bisher vorgestellten Theorie u. a. in [Broy80, Reichel84] nachlesen. Die Verwendung von partiellen Operatoren in unserem Sinne findet man u.a. auch im CIP-System (siehe [CIP85, Partsch90]) und innerhalb der Spezifikationssprache SEGRAS des ESPRIT Projekts GRASPIN (siehe hierzu [Schmidt89]).

Die im folgenden vorgestellten Erweiterungen kann man in drei Kategorien unterteilen. Die eine Kategorie definiert andere Schreibweisen für schon definierte Konstrukte und Abkürzungen zur einfacheren Formulierung von Formeln. Die zweite Kategorie beschreibt die schon erwähnte Erweiterung bzgl. partieller Operatoren. Die dritte Kategorie stellt Konstruktionsmechanismen bereit, um abstrakte Datentypen aus Standardspezifikationen, die in SNL eingebaut sind, mit Hilfe von parametrisierten Spezifikationen, den Konzepten Tupel, Rekord, Union und Liste in einfacher Weise gewinnen zu können.

3.3.1 Alternative Schreibweisen und Abkürzungen

Als eine alternative Schreibweise erlauben wir es zunächst einmal, daß ein Benutzer eine Implikationsformel in einer für Informatiker gewohnteren an einer Programmiersprache angelehnten Schreibweise angeben kann. Wir erlauben daher, daß man die Formel α=>β auch in der Form **if** α **then** β angeben kann.

Entprechend einem if-then-else Konstrukt einer Programmiersprache definieren wir weiter, daß das Konstrukt **if** α **then** β **else** γ als Abkürzung für die Formel (α => β) **and** (**not** α => γ) steht.

Als weitere Abkürzung definieren wir, daß α <=> β für die Formel (α => β) **andalso** (β => α) steht.

Damit erhalten wir für die Syntax von Formeln die erweiterte Form:

formula: **all** *var_id formula*
exist *var_id formula*
formula **andalso** *formula*
formula **orelse** *formula*
formula => *formula*
if *formula* **then** *formula* [**else** *formula*]
formula <=> *formula*
not *formula*
(*formula*)
elementary_formula

3.3.2 Partielle Operatoren und Prädikate

Bei der Definition der Spezifikation `STACK` haben wir bislang semantisch nicht definiert, welches Verhalten die Modelle dieser Spezifikationen zeigen sollen, wenn man die Operationen `pop` und `top` auf einen leeren Stack anwendet. In der Regel wird man es als einen Fehler oder Ausnahmezustand ansehen, wenn man eine `top` Operation auf einen leeren Stack durchführen will.

Dabei gibt es prinzipiell zwei Möglichkeiten, um das Verhalten von Operatoren und Prädikaten auf solchen Ausnahmewerten zu beschreiben. Im ersten Fall würde man in jeder Sorte, in der man sie benötigt, spezielle Fehlerwerte einführen, und das Ergebnis einer Operation auf einem Ausnahmewert dann durch solch einen Fehlerwert beschreiben. So könnte man bei der Spezifikation `STACK` z.B. eine Konstante `error` von der Sorte `elem` einführen und dann eine Gleichung `top(empty)=error` angeben, die definiert, daß der Wert von `top` angewendet auf die leere Warteschlange der Wert `error` der Sorte `elem` sein soll. Das hat aber den Nachteil, daß man nun das spezielle Element `error` in der Sorte `elem` hat, und damit wiederum `push(w,error)` sinnvoll definiert werden muß. Ist dies z.B ein Stack, dessen oberstes Element das Element `error` ist oder definiert man vieleicht `push(w,error)=w`?

Die Alternative zur Einführung spezieller Fehlerwerte wäre, daß wir in unserem theoretischen Konzept von Operationen auf Datentypen zulassen, daß diese Operationen nicht für alle möglichen Werte, die bei einer Realisierung einer Sorte zugeordnet sind, definiert sein müssen. Man spricht bei Operationen, die nicht auf dem gesamten möglichen Wertebereich definiert sind, von *partiellen Operationen*. So könnten wir im Beispiel `STACK` definieren, daß der Wert von `top` auf der Warteschlange `empty` nicht definiert ist, und damit eine Anwendung von `top` auf `empty` verboten ist. Die letzte Methode hat nicht nur den Vorteil, daß Spezifikationen von Datentypen in einer Sprache, die partielle Operatoren zuläßt, in der Regel eine einfachere Gestalt haben, da man nicht alle Ausnahmezustände explizit mit ihren Werten definieren muß, sondern bietet stets auch die Möglichkeit die erste Methode, d.h. die Verwendung von Ausnahmewerten, zu verwenden, da totale Operatoren nur Spezialfälle partieller Operatoren sind.

In SNL lassen wir es nun zu, daß man den Wertebereich, auf dem eine Operation eines (partiellen) algebraischen Systems definiert sein muß und definiert ist, innerhalb einer Spezifikation durch Formeln einschränkt.

Hierzu führen wir einen Deklarationsteil für partielle Operatoren (*spec_ptl_decls*) ein, dessen Syntax wie folgt definiert ist.

spec_ptl_decls: **ptl** *spec_ptl_decl* { ; *spec_ptl_decl* }$_0^*$ [;]
spec_ptl_decl: *ptl_id* [: *ptl_type*] [= *ptl_name*]
ptl_id: **id**
symbol
ptl_type: *type_comp* { * *type_comp*}$_0^*$ -> *sort_name*
type_comp: (*var_tuple* : *sort_name* { * *sort_name* }$_0^*$ | *formula*)
sort_name
var_tuple: *var_id*
(*var_id* { , *var_id* }$_0^*$)

Zur Angabe von partiellen Operatoren in Signaturen führen wir entsprechend einen *sig_ptl_decls*-Teil ein:

sig_ptl_decls: **ptl** *sig_ptl_decl* { ; *sig_ptl_decl* }$_0^*$
sig_ptl_decl: *sig_opn_decl*

Die Syntax (*var_tuple* : *sort_name* { * *sort_name* }$_0^*$ | *formula*) zur Angabe eingeschränkter Argumentwertebereiche für partielle Operatoren ist an die mathematische Mengenschreibweise
{*var_tuple* ∈ *sort_name* { * *sort_name* }$_0^*$ | *formulas*} angelehnt. Mit dieser Syntax kann man die Spezifikation STACK wie folgt definieren:

Beispiel 21 (Spezifikation STACK mit partiellen Operatoren)

Wir schränken hier den Argumentwertebereich von pop und top auf die Stacks ein, die nicht leer sind.

```
specification STACK = spec
     sort      stack; item;
     const     empty : stack;
     opn       push : stack * item -> stack;
     ptl       pop: (st:stack | st <> empty) -> stack;
               top :(st:stack | st <> empty) -> item;
     var       st : stack;
               x : item;
     eqn       pop(push(st,x))=st;
               top(push(st,x))=x;
end
```

Man beachte, daß der Formelteil zur Angabe eines eingeschränkten Wertebereiches einer partiellen Operation eine beliebige geschlossene Formel enthalten kann. Variablen, die dabei in einer *var_tuple* Deklaration bereits eindeutig vom Typ her festgelegt sind, müssen nicht innerhalb einer Variablendeklaration vorab deklariert werden, während alle anderen verwendeten Variablen in Variablendeklarationen vorab deklariert werden müssen.

Die obige Spezifikation STACK zeigt nur Beispiele von partiellen Operatoren, deren Argumentbereichsspezifikationen nur eine Variable von einer Sorte enthalten. Im allgemeinen benötigt man hier aber Einschränkungen, die sich nur bzgl. Tupeln von Variablen bzgl. mehrerer Sorten formulieren lassen. Wenn wir z.B. unsere STACK Spezifikation durch einen Operator `select : fifo * nat ->item` anreichern wollen, der das i-te Element auf dem Stack selektieren soll, so wird man den Argumentbereich bzgl. der Sorte nat auf die folgende Weise einschränken wollen:

```
ptl  select : ((w,i):fifo*nat | succ(0)<=i andalso
                                i<=length(w)) -> E.elem
```

Bei längeren und komplizierteren Einschränkungen wie bei der obigen Operation select kann es auf Grund der Übersichtlichkeit der Darstellung einer Spezifikation sinnvoll sein, die einschränkenden Bedingungen ganz aus der Operatordeklaration herauszunehmen, und gesondert im Formelteil festzuhalten.

Hierzu führen wir in SNL ein Sprachkonstrukt **def** *term* ein, daß wir als Abkürzung der Formel *exist y* (*y=term*) ansehen, die wir in der Form im Kontext der Semantik partieller algebraischer Systeme interpretieren können, daß es ein *y* aus dem Wertebereich gibt, dem auch der Term *term* angehört, so daß der Wert von Term definiert und gleich *y* ist. Wir können dann z.B. die Semantik von select auf die folgende Art und Weise beschreiben.

```
ptl   select : fifo * nat -> item;
var   w : fifo; i : nat;
eqn   def select(w,i) <=> succ(0) <=i andalso
                          i<=length(w);
```

Um die obige Form einer Formel in SNL zuzulassen, müssen wir die Syntax zur Bildung von Formeln in SNL durch ein def-Konstrukt erweitern:

formula:
- **all** *var_id formula*
- **exist** *var_id formula*
- *formula* **andalso** *formula*
- *formula* **orelse** *formula*
- *formula* **=>** *formula*
- **if** *formula* **then** *formula* [**else** *formula*]
- *formula* **<=>** *formula*
- **not** *formula*
- **def** *term*
- **(** *formula* **)**
- *elementary_formula*

Man beachte, daß innerhalb von SNL Einschränkungen der Wertebereiche von Operatoren stets durch notwendige und hinreichende Bedingungen beschrieben werden. Im Gegensatz hierzu ist es z.B. im CIP-System ([CIP85, Partsch90]) erlaubt, solche Einschränkungen durch Formeln der Gestalt *def term => formula* anzugeben, die zwar *formula* als notwendige, nicht aber als hinreichende Bedingung zur Existenz des Wertes von Term angeben. In SNL verlangen wir dagegen im Sinne von [Reichel84], daß sowohl eine notwendige als auch hinreichende Bedingung angegeben werden muß.

In Hinblick auf partielle Operatoren ist die Semantik von Termen gegeben durch den Begriff der Realisation von Termen neu zu überdenken. Da Operatoren vom Typ (w,s) nun nicht mehr auf dem gesamten Kreuzprodukt A_w bzgl. eines algebraischen Systems A definiert sein müssen, kann es vorkommen, daß der Wert eines Argumentterms eines solchen Operators innerhalb eines Terms außerhalb des Definitionsbereichs der zu dem Operator gegebenen Operation liegt. Die Anwendung des Operators auf so einen Term ist dann gewissermaßen *undefiniert*, bzw. nicht sinnvoll definierbar. Die Realisation von Termen in Bezug auf Spezifikationen mit partiellen Operatoren wird daher eine Abbildung sein, die nicht auf der gesamten Menge der Terme zu einer Signatur und einer gegebenen Familie von Variablen gegeben ist, sondern nur auf der Teilmenge der Terme, die interpretierbar sind, d.h. jeder Subterm eines solchen Terms ist rekursiv definierbar in dem Sinne, daß jeder Operator nur auf Terme angewendet wird, die definiert sind, und in der Interpretation liegen die Werte dieser Terme im Wertebereich der zum Operator gehörigen (partiellen) Operation. Wir gehen dabei hier davon aus, daß partielle Operationen *strikt* sind, d.h. die Anwendung einer partiellen Operation auf Argumente, die nicht alle definiert sind, ist ebenfalls undefiniert.

Analog zu der Interpretation von Termen müssen wir auch die Interpretation von atomaren Formeln überdenken, d.h. welche Semantik hat ein Prädikat angewendet auf Argumentterme, wenn nicht alle Argumentterme definiert sind. Hier definieren wir, daß der Wert einer atomaren Formel bzgl. einer gegebenen Variablenauswertung wahr ist, wenn entweder alle Argumentterme des beteiligten Prädikats undefiniert sind oder der Wert aller Argumentterme definiert ist und die zum Prädikat gehörige Relation von diesen Werten erfüllt wird. Andernfalls ist der Wert der atomaren Formel falsch. Für unser implizit stets vorhandenes Prädikat = bezeichnet man die mit der oben gegebenen Interpretation gegebene Semantik in der Literatur als *starke Gleichheit* (*strong equality*).

Bei allgemeinen Formeln müssen wir die Semantik nun nicht weiter redefinieren, da ja gemäß unserer obigen Definition der Semantik von atomaren Formeln diese stets einen definierten Wert besitzen. Man beachte dabei, daß wir keine partiellen Prädikate zulassen.

Bei der Angabe von Formeln in Spezifikationen mit partiellen Operatoren muß man in der Regel vorsichtig sein, um nicht Inkonsistenzen zwischen Formeln einzuführen. Bei unserem Beispiel `STACK` würde z.B. die Formel `pop(push(st,top(st)))=st` bewirken, daß einerseits da `top(empty)` undefiniert ist, `push(empty, top(empty))` und damit `pop(push(empty,top(empty)))` auf Grund der Striktheit ebenfalls undefiniert ist, andererseits dieser Wert gemäß der angegebenen Gleichung aber gleich `empty` sein sollte. Es ist daher besser, Gleichungen der obigen Gestalt stets durch eine if-Bedingung zu sichern, wie z.B. in

```
if (st <> empty) then pop(push(st,top(st)))=st
```

3.3.3 Parametrisierte Spezifikationen

Häufig beobachtet man bei abstrakten Datentypen analoge Strukturen in der Weise, daß sie bis auf Stellen, an denen Symbole eines eingebundenen Datentyps verwendet werden, gleich sind. Durch eine Parametrisierung der Spezifikationen kann man solche Gemeinsamkeiten extrahieren und in einer parametrisierten Datenspezifikation unterbringen. Die konkreten Datentypen entstehen dann durch Instantiierung aus der parametrisierten Spezifikation. Wir könnten z.B. eine parametrisierte Spezifikation "Stack von irgendeinem abstrakten Typ" haben, aus der wir dann durch Instantiierung die Spezifikationen für Typen "Stack von natürlichen Zahlen", "Stack von Zeichenketten" oder "Stack von Mengen" gewinnen könnten. In der Theorie läßt sich eine parametrisierte Spezifikation als eine Abbildung beschreiben, die als Argument abstrakte Datentypen mit einer vorgegebenen Spezifikation (insbesondere einer vorgegebenen Signatur) nimmt und diese auf abstrakte Datentypen mit einer anderen vorgegebenen Spezifikation abbildet. Solche Abbildungen nennt man in der Kategorientheorie Funktoren. Wir wollen hier auf die Theorie von Funktoren, wie sie u.a. in der Theorie der algebraischen Spezifikation von Datentypen verwendet werden, nicht näher eingehen (Man siehe hierzu z.B. [Ehrig90, Herrlich73, MacLane72]), sondern hier einen konkreten Kalkül zur Parametrisierung von algebraischen Spezifikationen angeben der auf dem Funktorkalkül beruht. Hierzu führen wir den Begriff eines Funktors innerhalb unserer Spezifikationssprache ein.

functor_decls:	**functor** *functor_decl* { ; *functor_decl* }$_0^*$
functor_decl:	*functor_id* ([*param_specs*]) [: *sig*] = *spec*
functor_id:	**id**
param_specs:	*param_spec* { , *param_spec* }$_0^*$
param_spec:	*param_id* : *spec*
param_id:	**id**

Eine Funktordeklaration besteht aus dem Schlüsselwort **functor** gefolgt von dem Namen des Funktors *functor_id*, eine in runden Klammern eingeschlossenen Parameterliste des Funktors, wobei Parameter Namen für Spezifikationen sind, dem Schlüsselwort = und dem Funktorrumpf, der durch eine Spezifikation definiert wird. Man beachte dabei, daß die Parameterliste auch leer sein darf und daß im Fall der Angabe von Parametern die Spezifikation, die dem Parameternamen als "Typ" zugeordnet ist, mit angegeben werden muß. Innerhalb der Spezifikationsdefinition, die den Funktorrumpf beschreibt, können dann die Elemente der formalen Parameterspezifikationen, die exportiert werden, wie bei eingebundenen Unterspezifikationen in der Form *param_id.name_of_element_of_spec* referenziert werden. Man beachte dabei aber, daß die formalen Parameterspezifikationen normaler Weise von dem Funktor nicht exportiert werden, sofern sie nicht explizit durch eine Unterspezifikationsdeklaration, die exportiert wird, indirekt exportiert werden. Betrachten wir hierzu einmal drei Beispiele:

Beispiel 22 (Beispiel zu parametrisierten Spezifikationen)

(a) Wir wollen als erstes Beispiel einmal eine parametrisierte Fassung unseres Stacktyps vorstellen.

```
functor STACK ( E : spec sort elem; end) = spec
     sort     stack;
              elem=E.elem;
     const    empty : stack;
     opn      push : stack * elem -> stack;
     ptl      pop : (w:stack | w <> empty) -> stack;
              top : (w:stack | w <> empty) -> elem;
     var      st : stack; e: elem;
     eqn      pop(push(st,e))=st;
              top(push(st,e))=e;
end
```

Hier haben wir nun die Sorte `elem` als Sorte der Parameterspezifikation verwendet.

(b) Als nächstes betrachten wir eine parametrisierte Version eines abstrakten Datentyps Menge.

```
specification MONOID = spec
     sort     elem;
     const    e : elem;
     opn      op : elem * elem -> elem;
end

functor SET ( M : MONOID ) : sig
     sort     set;
              elem;
     const    emptyset : set;
     opn      incorp : set * elem -> set;
              total : set -> elem;
     pred     contains : set * elem;
end = spec
     sort     elem=M.elem;
     var      s : set; y: elem; x : elem;
     eqn      not contains(empty,x);
              contains(incorp(s,x),x);
              (x <> y) => (contains(incorp(s,x),y) <=>
                                   contains(s,y));
              total(emptyset)= M.e;
              contains(s,x) =>
```

```
                                    total(incorp(s,x))=total(s);
                    not contains(s,x) => total(incorp(s,x)=
                                                    M.op(total(s),x);
    end
```

SET ist ein typisches Beispiel für einen Funktor, der in der Argumentspezifikation nicht nur Sorten, sondern auch Operatoren enthält. Die Semantik des Operators `total` der parametrisierten Spezifikation SET kann hier nur mit Hilfe des Operators `op`, der in der Parameterspezifikation übergeben wird, definiert werden. Man beachte, daß die Konstante `e` ebenfalls nur als Hilfsgröße zur Definition von `total` verwendet wird. Daher werden diese Operationen von SET auch nicht exportiert und sind damit für den Benutzer von SET verdeckt.

(c) Wir wollen noch eine parametrisierte Version eines Warteschlangendatentyps (first-in-first-out) angeben.

```
            specification ELEMENT = spec
                sort   elem;
            end
functor FIFO ( E : ELEMENT ) : sig
      sort   fifo;
             elem;
             nat;
      const  empty : fifo;
      opn    put : fifo * elem -> fifo;
      ptl    get : fifo -> elem;
             rest : fifo -> fifo;
      opn    length : fifo -> nat;
= spec
      sort   elem=E.elem;
             nat=NAT.nat;
      var    w : fifo;
             n : elem;
      eqn    def get(w) <=> w <> empty;
             if (w <> empty) then get(put(w,n))=get(w)
                             else get(put(w,n))=n;
             def rest(w) <=> w <> empty;
             if (w <> empty) then
                          rest(put(w,n))=put(rest(w),n)
                         else rest(put(w,n))=w;
             length(put(w,n))= NAT.succ(length(w));
             length(empty)=0;
end
```

Wir haben bisher nur Beispiele gesehen, bei denen die Parameter von Funktoren nur Signaturdefinitionen enthielten. Das innerhalb einer formalen Parameterspezifikation auch Formeln sinnvoll sind, zeigt das folgende Beispiel:

Beispiel 23 (Bedingungen an formale Parameter)

Wenn wir in unserer parametrisierten Spezifikation SET verlangen wollten, daß die Operation des Einfügens von Elementen in die Menge kommutativ ist, d.h. daß die Gleichung

```
incorp(incorp(s,x),y)=incorp(incorp(s,y),x)
```

erfüllt ist, so müssen wir von der Operation `op` ebenfalls verlangen, daß sie kommutativ ist. Andernfalls würden wir einen inkonsistenten Typ bzgl. der Operation `total` erhalten. In diesem Fall würde man die Kommutativität von `op` in der formalen Parameterspezifikation von SET verlangen, z.B. so:

```
specification COMM_MONOID = spec
     sort   elem;
     const  e : elem;
     opn    op : elem * elem -> elem;
     var    x : elem; y : elem;
     eqn    op(x,y)=op(y,x);
end
```

Wir verbinden mit parametrisierten Datentypen keine unabhängige Semantik. Die Semantik der auf parametrisierte Datentypen aufgebauten Typen wird vielmehr einfach durch die Instantiierung der Parametrisierung bestimmt. Die Instantiierung wollen wir nun näher erläutern.

Bei der Instantiierung eines parametrisierten abstrakten Datentyps werden die formalen Parameterspezifikationen durch aktuelle Parameterspezifikationen ersetzt (d.h. die Spezifikation, die den Typ des Parameters bestimmt, wird durch den aktuellen Parameter, eine Spezifikation, ersetzt), wobei man sich letztere durch eine Unterspezifikationsdeklarationdeklaration in den Rumpf der parametrisierten Spezifikation eingebunden denken kann. Es entsteht also nach der Instantiierung eine Spezifikation, die der Rumpfspezifikation des parametrisierten Datentyps erweitert um Unterspezifikationsdeklarationen für jede aktuelle Parameterspezifikation von der Gestalt

specification *param_id* : *param_sig* = *actual_param_spec*

entspricht. Aus der obigen Syntax ersieht man schon, daß die aktuellen Parameterspezifikationen von ihrer Signatur her nicht identisch zu den Signaturangaben der formalen Parameter sein müssen, sondern ihre Signatur die Signatur des formalen Parameters nur als Einschränkung enthalten müssen. Weiter muß die aktuelle Parameterspezifikation die Gleichungen der formalen Parameterspezifikation erfüllen.

Um die Instantiierung von parametrisierten Datentypen in unsere Sprache einzubauen, verallgemeinern wir die Syntax für eine Spezifikation wie folgt:

spec:	*spec_name* [**with** *rename_list* **end**] **spec** *spec_body* **end** *functor_id* ([*spec_param_list*]) [**with** *rename_list* **end**]
spec_param_list:	*spec* { , *spec* }$_0^*$

Eine Spezifikation wird jetzt nicht nur durch den Spezifikationsnamen einer bereits definierten Spezifikation oder durch eine Spezifikationsdefinition sondern auch durch einen Funktoraufruf beschrieben. In einem Funktoraufruf folgt dem Funktornamen die aktuelle Parameterliste zur Instantiierung des Funktors und optional eine Liste von Umbenennungen, wie wir dies schon von der Referenz einfacher Spezifikationen kennen.

Betrachten wir einmal ein paar Beispiele für mögliche Instantiierungen:

Beispiel 24 (Instantiierung von parametrisierten Datentypen)

(a) Bzgl. unseres parametrisierten Datentyps Stack wäre nun die Instantiierung einer Spezifikation "Stack von natürlichen Zahlen" wie folgt denkbar:

```
specification NATSTACK=STACK(spec sort elem=NAT.nat end);
```

Wir haben hier einfach die Sorte `elem` von `STACK` durch die Sorte `nat` von `NAT` instantiiert. Äquivalent hierzu wäre die Instantiierung

```
specification NATSTACK=STACK(NAT with elem for nat end);
```

(b) Auf ähnliche Weise können wir auch `SET` durch `NAT` instantiieren. Wir wollen dabei jedoch die Sorte `set` nach `natset` und die Operation `total` nach `sum` umbenennen. Als aktuelle Parameterspezifikation wählen wir nun:

```
specification NAT_MONOID = spec
     sort      elem=NAT.nat;
     const     e=NAT.0;
     opn       op=NAT.+;
end
```

Die Signatur von `NATSET` beschreiben wir durch:

```
signature NATSET = sig
     sort      natset; nat;
     const     emptyset : natset;
     opn       incorp : natset * nat -> natset;
               sum : natset -> nat;
     pred      contains : natset * nat;
end
```

NATSET erhalten wir dann durch die folgende Instantiierung von SET:

```
specification NATSET : NATSET = spec
     include   SET(NAT_MONOID);
     sort      natset=set;
               nat=elem;
     opn       sum=total;
end
```

Man beachte, daß wir hier set in natset und total in sum umbenannt haben. sum addiert jetzt offensichtlich die Werte aller natürlichen Zahlen, die auf dem Stack sind, auf. Mit der Möglichkeit, Elemente der Signatur eines Funktors bei der Instantiierung noch umzubenennen, können wir die Spezifikation NATSET auch schreiben als:

```
specification NATSET : NATSET = SET(NAT_MONOID)
     with      natset for set,
               nat for elem,
               sum for total
     end;
```

Man beachte, daß gemäß der Syntax von SNL eine Funktorinstantiierung ein Sprachelement vom Typ *spec* ist, d.h. Funktorinstantiierungen können überall da verwendet werden, wo Spezifikationen verlangt werden. Eine Funktorinstantiierung kann damit innerhalb der Definitionen anderer Spezifikationen oder Funktoren verwendet werden.

Beispiel 25 (Funktorinstantiierung in einem Funktorrumpf)

Wir wollen einmal den Funktor SET zu einen Funktor MSET ausdehnen, der eine Operation zur Bestimmung des kleinsten Elementes innerhalb einer Menge enthält. Die Parameterspezifikation des Funktors MSET muß dann als zusätzliches Element gegenüber der formalen Parameterspezifikation MONOID von SET ein Prädikat zum Vergleich der Elemente innerhalb der betrachteten Mengen enthalten. Wir definieren daher zunächst:

```
specification MSET_PARAM = spec
     include   MONOID;
     pred      le  : elem * elem;
end
```

Die Spezifikation von MSET lautet dann:

```
specification MSET(M : MSET_PARAM) :
sig
     sort      set;
               elem;
     const     emptyset : set;
     opn       incorp : set * elem -> set;
               total : set -> elem;
     ptl       min : set -> elem;
     pred      contains : set * elem;
end = spec
     include   SET(M);
     ptl       min : (s:set | s <> emptyset) -> elem;
     var       s : set; x : elem;
     eqn       if (s=empty) then min(incorp(s,x))=x
```

```
                    else if M.le(x,min(s)) then
                         min(incorp(s,x))=x
                    else min(incorp(s,x))=min(s);
end
```

Eine Funktorinstantiierung kann weiter als aktueller Parameter eines anderen Funktors auftreten.

Beispiel 26 (Funktorinstantiierung als aktueller Parameter eines Funktors)

Die folgende Spezifikationsdeklaration definiert eine Warteschlange, deren Elemente Mengen natürlicher Zahlen sind.

```
specification FIFO_NATSET=
     FIFO(SET(NAT_MONOID) with elem for set end);
```

3.3.4 Standardtypen in SNL

Um eine Basis für die Konstruktion von Spezifikationen zu bieten, ist es sinnvoll, einige Standarddatentypen in SMARAGD bereitzustellen, so daß der Modellierer auf diese bereits eingebauten Typen aufbauen kann. Wir stellen daher in SNL Spezifikationen von Standarddatentypen (BOOL, INT und STRING) bereit, für die es zugehörige Standardmodelle und Standardimplementierungen in der Sprache ML gibt. Wir wollen diese Typen im folgenden im einzelnen vorstellen.

3.3.4.1 Der Standarddatentyp BOOL

Zur Angabe der prädikatenlogischen Semantik von Formeln in SNL haben wir einen booleschen Datentyp mit Signatur *LBOOL* verwendet. Wenn wir für die folgende Spezifikation in SNL die SNL-Schlüsselworte **not**, **andalso** und **orelse** als Operatorsymbole und nicht als Schlüsselworte interpretieren, so läßt sich die zu *LBOOL* gehörige Spezifikation des booleschen Datentyps in SNL wie folgt beschreiben:

```
specification BOOL = spec
     sort   bool;
     const  true      : bool;
            false     : bool;
     opn    not       : bool -> bool;
            andalso   : bool * bool -> bool;
            orelse    : bool * bool -> bool;
     infix  andalso   2;
            orelse    1;
     var    x : bool; y: bool;
     eqn    true <> false;
            not true = false;
            not false = true;
            true andalso x = x;
            false andalso x = false;
            x orelse y = not (not x andalso not y);
end
```

Dies entspricht einer funktionalen Sicht der prädikatenlogischen Sprache 1.ter Ordnung, über die SNL Formeln interpretiert werden. Wir können daher (da die obige Spezifikation von BOOL als Standardtyp in SNL enthalten ist) die Schlüsselworte **andalso**, **orelse** und **not** in SNL einmal als Elemente der SNL zu Grunde liegenden prädikatenlogischen Sprache 1. Ordnung auffassen, wie dies hinter Def. 4 im Text beschrieben wird, oder äquivalenter Weise als die Operatoren der SDL-Standardspezifikation BOOL, wie sie oben definiert ist, ansehen. Ein Prädikat *predicate : s1 * s2 * ... * sk* kann man daher auch in äquivalenter Weise in SNL als eine Funktion *predicate : s1 * s2 * ... * sk -> bool* schreiben.

Im folgenden erlauben wir es, daß die Elemente der oben definierten Spezifikation BOOL überall in SNL-Spezifikationen ohne eine explizite Inkludedeklaration der Spezifikation BOOL verwendet werden dürfen, da BOOL per Definition in jede SNL-Spezifikation implizit inkludiert wird. Die Namen der Elemente von BOOL sind daher, um Namenskonflikte zu vermeiden, als Schlüsselwörter von SNL anzusehen, wie dies schon bzgl. der nichtfunktionalen Sicht von SNL definiert wurde.

Wir verbinden mit der obigen Spezifikation BOOL stets eine Standardrealisierung, die minimal und termerzeugt ist. Eine solche Realisierung beinhaltet als Wertemenge für die Sorte `bool` genau zwei Werte für die Konstanten `true` und `false`. Je zwei solcher Realisierungen sind isomorph. In ML ist BOOL über den ML-Standarddatentyp bool in SMARAGD realisiert, wie wir dies etwas später noch genauer darlegen werden.

3.3.4.2 Der Standarddatentyp INT

Der Standarddatentyp INT wird in SNL durch die folgende Spezifikation beschrieben.

```
specification INT = spec
     sort   int;
     const  0 :int;
     opn    succ  : int -> int;
            pred  : int -> int;
            ~     : int -> int;
            +     : int * int -> int;
            -     : int * int -> int;
            *     : int * int -> int;
            div   : int * int -> int;
            mod   : int * int -> int;
     pred   <=    : int * int;
            <     : int * int;
            >=    : int * int;
            >     : int * int;
     infix  * 5; div 5; mod 5;
            + 4; - 4; * 5;
            <= 3; < 3; >= 3; > 3
     var    x : int; y : int;
     eqn    succ(pred(x))=x; pred(succ(x))=x;
            ~(0) = 0; ~(succ(y)) = pred(~y);
            ~(pred(y)) = succ(~y);
            x+0=x; x+(succ(y)) = succ(x+y);
            x+(pred(y)) = pred(x+y);
            x-y = x + (-y);
            x*0=0; x*(succ y)=x*y+x;
            x*(pred y)=x*y-x;
            x<=x;(x<=y) => (x<=(succ(y)));
            not (succ(x)<=x);
            not (x<=y) => not (succ(x)<=y);
            (x<y) <=> (not (x=x) andalso x<=y;
            (x>y) <=> not (x<=y);
            (x>=y) <=> not (x<y);
            ((x>=0)andalso(y>0)) =>
              if (x<=y) then x div y = 0
              else x div y = (x-y) div y +1;
            ((x<0)andalso(y>0)) =>
              x div y = (x+y) div y -1;
            (y<>0) => x div y = ~x div ~y;
            x mod y = x - (x div y) * y;

end
```

Wir verbinden mit dieser Spezifikation stets das mathematische Modell der ganzen Zahlen mit der Addition +, der Subtraktion -, der Multiplikation *, der ganzzahligen Division `div`, der Modulooperation `mod`, der Negation ~, der Inkrementoperation `succ` und der Dekrementoperation `pred`. Weiter haben wir die Standardvergleichsrelationen auf den ganzen Zahlen <, <=, =, <>, >= und >.

Damit wir die Werte der Sorte `int` gemäß den üblichen mathematischen Konventionen benennen können, führen wir in SNL als Standardschreibweisen für ganze Zahlen `1`, `2`, ... als Abkürzungen für die Terme `succ(0)`,

`succ(succ(0))`, ... ein. Man beachte, daß man negative Zahlen damit über den Negationsoperator als `~1`, `~2`, etc. schreiben kann.

3.3.4.3 Der Standarddatentyp STRING

Der Standarddatentyp STRING hat die folgende Spezifikation:

```
specification STRING = spec
    sort   string;
    const  "A"; .... "Z";
           "a"; ...; "z";
           "0"; ... ; "9";
           "";
    opn    ^ : string * string -> string;
    infix  ^ 6;
    opn    size : string -> int;
    pred   < : string * string;
           <= : string * string;
           > : string * string;
           >= : string * string;
    var    x1 : string; y1 : string;
           x2 : string; y2 : string;
    eqn    "A" < "B"; ...; "A" < "9"; "A" > "";
           "B" < "C"; ...; "B" < "9"; "B" > "";
           ...
           "8" < "9"; "8" > "";
           "9" > "";
           x1 ^ "" = x1;
           "" ^ x1 = x1;
           x1 ^ y1 = x2 ^ y2 andalso size (y1)=size(y2)
           <=> x1 = x2 andalso y1=y2;;
           size(x1 ^ y1)=size(x1) + size(y1);
           size("A")=1; ..., size("9")=1;
           size("")=0;
           a <> "" => a ^ b > nil;
           not (nil > a ^ b);
           size(i)=1 andalso size(j)=1 =>
             ((i^a) > (j^b) <=>
                (i>j) orelse (i=j andalso a>b));
           a <= b <=> not (a>b);
           a <b <=> b > a;
           a >= b <=> b <= a;
end
```

Für "a_1" ^"a_2" ^ ... ^"a_k" schreiben wir abkürzend auch "$a_1a_2...a_k$".

Wir verbinden mit der Spezifikation `STRING` als Modell stets ein algebraisches System, das als Wertemenge der Sorte `string` die Menge der Wörter enthält, die aus den Zeichen des Alphabets {"A", ..., "Z", "a", ..., "z", "0", ...,"9" } enthält, wobei "" das leere Wort beschreibt. ^ ist in diesem algebraischen System die Operation der Konkatenation zweier Wörter und `size` die Längenfunktion, die beschreibt, wieviel Buchstaben ein Wort enthält. Die Ordnungsrelationen sind durch die lexikographische Ordnungsrelation bestimmt. In ML wird dieses Modell durch den ML-Standarddatentyp *string* bereitgestellt.

3.3.5 Konstruktion von Tupeln, Rekords, Unionen und Listen von Spezifikationen

In der Praxis verwendet man zur Bildung komplexerer Datentypen häufig Konstruktionsverfahren, die einem einzelne Datentypen als Komponenten eines weiteren (man sagt strukturierten) Datentyps erscheinen lassen. Aus mathematischer Sicht bieten sich für die Bildung solcher zusammengesetzter Datentypen insbesondere die Konzepte kartesisches Produkt und disjunkte Summe an, die ein Programmierer in der Regel als Tupel- bzw. Rekord und Union kennt.

Im folgenden wollen wir zeigen, daß man die Bildung von kartesischen Produkten und disjunkten Summen in Form von Tupeln bzw. Rekords oder Unionen auf einfache Art und Weise in einem erweiterten Kontext der Spezifikationssprache SNL über die Instantiierung geeigneter (SMARAGD-interner) Funktoren beschreiben kann. Aus diesem Grunde kann man in SNL allgemeine Sprachkonstrukte bereitstellen, über die ein Anwender auf einfache Art und Weise Tupel, Rekords bzw. Unionen innerhalb von SNL bilden kann, wobei diese Sprachkonstrukte dann über spezielle Transformationsregeln, die wir im folgenden noch näher vorstellen werden, auf die Instantiierung der zugehörigen SMARAGD-internen Funktoren zurückgeführt werden. Wir folgen dabei einer Methode, wie sie für die Breitbandsprache CIP-L des CIP Systems in [CIP85] vorgestellt wird, wählen allerdings eine etwas andere Notation und unterscheiden weiter die Produkte in die zwei Formen Tupel und Rekord.

Damit der Anwender eine Basis (siehe in diesem Zusammenhang auch den Abschnitt über Standardspezifikationen in SNL) für Implementierungen seiner Spezifikationen hat, gehen wir dabei davon aus, daß die beteiligten Funktoren weiter mit Standardrealisierungen verbunden sind.

Bei der Definition der folgenden SMARAGD-internen Funktoren und Transformationsregeln erlauben wir als Namen von internen bzw. durch Schlüsselworte von SNL noch zu definierenden Sorten und Operatoren spezielle, sonst auf Grund der Syntax nicht verwendbare Zeichenketten, wobei wir solche sonst als Namen nicht erlaubte Zeichenketten der Übersichtlichkeit in Hochkomma (') einschließen. Dies führt zu keinen Schwierigkeiten, da es sich ja nur um SMARAGD-interne Konstrukte handelt, deren für den Benutzer sichtbaren und verwendbaren Elemente über noch zu definierende syntaktische Erweiterungen von SNL bereitgestellt werden.

Die Bildung eines kartesischen Produkts aus $k>1$ Komponentensorten läßt sich für jedes $k>1$, $k \in N$, durch einen SMARAGD-internen Funktor beschreiben, der durch das folgende verallgemeinerte Konstruktionsschema definiert ist.

```
functor PRODUCT_k(A:spec sort s1;...;sk end)=
spec
sort   's1*s2...*sk';
opn    '(.,.)' : A.s1 * A.s2 * .. *A.sk -> 's1*s2*...*sk';
       '#1' : 's1*s2*...*sk' -> A.s1;
       '#2' : 's1*s2*...*sk' -> A.s2;
       ......
       '#k' : 's1*s2*...*sk' -> A.sk;
var    x1 : A.s1; ... ; xk : A.sk;
       y1 : A.s1; ... ; yk : A.sk;
eqn    '(.,.)'(x1,...,xk)='(.,.)'(y1,...,yk) <=>
                x1=y1 andalso x2=y2 ... andalso xk=yk;
       '#1'('(.,.)'(x1,...,xk))=x1;
       ........
       '#k'('(.,.)'(x1,...,xk))=xk;
end;
```

Für jedes $k>1$, $k \in N$, definiert der Funktor PRODUCT_k zunächst eine Sorte mit Namen `s1*s2...*sk` und Operatoren mit den Namen `(.,.)`, `#1`, `#2` ... `#k`. Der Sortenname `s1*s2...*sk` symbolisiert, daß die Werte dieser Sorte bzgl. der in SNL vordefinierten Standardrealisierung der Funktoren alle Tupel der Gestalt `(x1,...,xk)` mit `x1` aus der Wertemenge von s1, x2 aus der Wertemenge von s2, usw. sind. Der Operator mit dem Namen `(.,.)` ist der Konstruktor für die Werte der Sorte `s1*s2...*sk`, d.h. in der Standardrealisation erzeugt die zugehörige Operation bei gegebenen Argumentwerten x1, x2, .., xk das Tupel (x1,x2,..,xk) und da wir nur termerzeugte Modelle betrachten, läßt sich jedes Element der zu `s1*s2...*sk` in der Standardrealisierung gehörigen Wertemenge auch so darstellen. Aus diesem Grunde liegt es nahe, wenn wir es in SNL erlauben, daß wir Terme der Gestalt (.,.) (x1,...,xk) einfach als Tupel (x1,..,xk) schreiben dürfen, was der Name (.,.) des Operators schon symbolisieren soll. Die Operatoren #1, #2, ..., #k stellen die Projektionsoperatoren dar. In der Standardrealisierung liefern die zugehörigen Operationen aus einem k-Tupel die jeweils i-te Komponente, was durch die Gleichungen innerhalb des Funktor PRODUCT_k explizit für alle Realisierungen festgeschrieben ist.

Es läßt sich leicht zeigen, daß die Funktoren PRODUCT_k für alle $k>1$, $k \in N$ monomorph relativ zu den Parametersorten sind, so daß die oben angegebene Standardrealisierung einen geeigneten Prototyp für andere Realisierungen darstellt. Weiter ist (.,.), wie alle anderen Operatoren in SNL, per Definition strikt, so daß die Anwendung von (.,.) auf Terme nur dann einen definiertes Tupel liefert, wenn alle Argumentterme definiert sind. Die stets für jede Sorte implizit vorhandene Gleichheits- und Ungleichheitsrelation ist so definiert, daß zwei Tupel genau dann gleich sind, wenn all ihre korrespondierenden Komponenten gleich sind.

Nachdem wir nun die Syntax und Semantik der SMARAGD-internen Funktoren PRODUCT_k geklärt haben, können wir uns nun ansehen, über welche Sprachkonstrukte wir Tupel und Rekords in SNL mit Hilfe dieser Funktoren und geeigneter Transformationsregeln einführen können. Wir wollen zunächst die Definition von Tupeln und die zugehörige Transformationsregel angeben.

Def. 18 (Tupeldefinition)

Ein Tupel wird in SNL über ein Sprachkonstrukt der Gestalt

```
sort p = m1 * m2 * ... * mk;
```

definiert, wobei ein solches Konstrukt per Definition äquivalent zu dem folgenden SNL-internen Codeteil ist:

```
include   PRODUCT_k(spec sort s1=m1; ...; sk=mk end)
          with  'm1*m2*...*mk' for 's1*s2*...*sk'
          end;
sort      p = 'm1*m2*...*mk';
```

Bei der obigen Definition ist zu beachten, daß ein Benutzer von SNL nur die Namen als Sortennamen verwenden darf, die innerhalb des Konstruktes

```
sort p = m1 * m2 * ... *mk;
```

angegeben werden, und diese müssen den für SNL üblichen Konventionen genügen. Gleichwohl ist, wie wir dies für Sortendefinitionen der Gestalt *sort name_1 = name_2* definiert haben, der Name `p` nur ein alternativer Name der intern definierten Sorte `m1*m2*...*mk`, so daß eine weitere Deklaration der Gestalt

```
sort p1 = m1 * m2 * ... *mk;
```

keine neue Sorte in SNL, sondern nur einen weiteren alternativen Namen `p1` für die Sorte `m1*m2*...*mk` definiert.

Weiter ist zu beachten, daß die standardmäßig als Projektions- und Konstruktionsoperatoren von Tupeln in SNL vordefinierten Operatornamen (.,.), #1, #2, etc in ihrer Bedeutung überladen sind, d.h. auf beliebige Tupel angewendet werden können, und daß eine Anwendung von (.,.) stets in der Tupelform (x1,..,xk) geschrieben werden muß.

Die Syntax von SNL erweitern wir damit wie folgt:

spec_sort_decl:	*sort_id* [= *sort_type*]
sort_type:	*sort_name*
	sort_type { * *sort_type* }$_0^*$
term:	*opn_name term*
	term opn_name term
	const_name
	var_id
	(*term* { , *term* }$_0^*$)

Die Analogie des *sort_type* Konstruktes zu der linken Seite einer Operator- oder Prädikattypdefinition ist kein Zufall. Man kann nämlich, wie dies auch in ML definiert ist, jede Operation mit mehreren Argumenten s1, ..,sk als eine Operation mit nur einem Argument deuten, wobei dieses eine Argument vom Typ her ein Tupeltyp der Gestalt s1 * s2 * .. sk ist. Wenn wir diese Deutung auf unsere Spezifikationssprache übertragen, können wir sagen, daß ein Operator op mit Typ s1 * s2 * .. *sk -> t äquivalent zu einem Operator op mit Typ s -> t ist, wobei s durch eine Deklaration der Gestalt sort s = s1 * s2 * .. * sk definiert ist.

Man beachte weiter, wie sich die Syntax für Terme der obigen Deutung von Operatoren als Operationen mit nur einem Argument angepaßt hat. Die Anwendung eines nicht als Infix definierten Operators auf sein (einziges) Argument ist nun definiert durch die Syntax *opn_name term*, wobei das Argument *term* nun ein Tupel von der Gestalt (*term* { , *term* }$_0^*$) sein kann.

Bei der praktischen Anwendung von Tupeln ist es oftmals störend, daß innerhalb eines Tupelterms ein Komponententerm nur an Hand seiner Position im Tupel eindeutig einer Sorte zugeordnet werden kann. Weiter stehen die Komponenten eines Tupels in der Regel für reale Objekte, die, auch wenn sie von der gleiche Sorte sind, eine verschiedenen Bedeutung haben, so daß man sie unterschiedlich kennzeichnen möchte. So könnte das folgende Tupel vom Typ string * string ("Hans", "Walter") in der ersten Komponente den Vornamen und in der zweiten Komponente den Nachnamen einer Person enthalten. Dies ist aber aus der Form des Tupels und der Elemente nicht ersichtlich (Ist "Walter" nun ein Vorname oder Nachname?). Aus diesem Grunde stellen Programmierspra-

chen in der Regel einen zusammengesetzten Typ Rekord bereit, der es erlaubt, seine einzelnen Komponenten über Namen anzusprechen.

Wir können unsere Konstruktion von Tupeln in SNL auf leichte Weise auf die Bildung von Rekords übertragen, wenn wir eine benutzergesteuerte Umbenennung der Projektionsoperatoren der Funktoren PRODUCT_k innerhalb eines neuen SNL *sort_type* Konstruktes erlauben. Die Definition solcher Rekords und der zugehörigen Transformationsregel lautet:

Def. 19 (Rekorddefinition)

Ein Rekord wird in SNL über ein Sprachkonstrukt der Gestalt

```
sort r = { label1 : m1, label2 : m2, ..., labelk : mk }
```

definiert, wobei ein solches Konstrukt per Definition äquivalent zu dem folgenden SNL-internen Codeteil ist:

```
include   PRODUCT_k(spec sort s1=m1; ...; sk=mk end)
          with     'm1*m2*...*mk' for 's1*s2*...*sk',
                   '{.,.}' for '(.,.)',
                   '#label1' for '#1',
                   '#label2' for '#2',
                   ...
                   '#labelk' for '#k'
          end;
sort      r = 'm1*m2*...*mk';
```

Wie auch bei Tupeln kann ein Benutzer von SNL zur Referenzierung der Sorte `'m1*m2*...*mk'` nur den Namen `r` verwenden. Im Gegensatz zu den immer gleich benannten Projektionsoperatoren von Tupeln, sind die Projektionsoperatoren von Rekords jetzt in der Syntax `#label1`, `#label2`, ..., `#labelk` über die Namen der Label der Rekordkomponenten, wie sie in der Definition eines Rekords gegeben sind, in SNL definiert. Hierzu erweitern wir die SNL-Syntax in der Weise, daß Namen, denen ein # vorangestellt ist, explizit als Projektionsoperatoren bzgl. gegebener Rekorddefinitionen aufgefaßt werden.

Weiter führen wir entsprechend unserer Vorgehensweise bei Tupeln zur Angabe der Terme, die entstehen, wenn wir die von ihren Namen in SNL nicht benutzbare Konstruktorfunktion `{.,.}` auf zugehörige Argumente `x1`, `x2`, .., `xk` anwenden, d.h. für Terme der Gestalt `{.,.} (x1,...,xk)`, in SNL die Schreibweise `{ label1=x1, label2=x2, ..., labelk=xk }` ein. Da innerhalb dieser Schreibweise eine Komponente eines Rekords durch die Labelangabe eindeutig identifiziert wird, ist die Reihenfolge der `labeli=xi` Konstrukte nicht wichtig, so daß wir hier eine beliebige Reihenfolge zulassen, wobei allerdings jede Komponente genau einmal auftreten muß.

Zur Einbindung von Rekords in SNL erweitern wir damit unsere Syntax durch:

spec_sort_decl:	*sort_id* [= *sort_type*]
sort_type:	*sort_name*
	sort_type { * *sort_type* }$_0^*$
	{ *label* : *sort_type* { , *label* : *sort_type* }$_0^*$ }
label:	**id**
term:	*opn_name term*
	term opn_name term
	const_name
	var_id
	(*term* { , *term* }$_0^*$)
	{ *label* = *term* { , *label* = *term* }$_0^*$ }

Wir wollen einmal ein Beispiel zu der Verwendung von Rekords und Tupeln betrachten.

Beispiel 27 (Verwendung von Tupeln und Rekords)

(a) Eine Spezifikation, die Paare von natürlichen Zahlen definiert, läßt sich in SNL wie folgt erzeugen:

```
specification PAIR'OF'NAT = spec
     include  NAT;
     sort pair = nat * nat;
end;
```

Wir wollen diese Spezifikation einmal mit einem Prädikat zum Vergleich von solchen Paaren anreichern:

```
specification PAIR'OF'NAT = spec
     include   NAT;
     sort      pair = nat * nat;
     pred      lt : pair * pair;
     infix     lt 4;
     var       x1 : nat; y1 : nat;
               x2 : nat; y2 : nat;
     eqn       (x1,y1) lt (x2,y2) <=> x1<=x2
                                      andalso
                                      y1<=y2;
end;
```

Wir haben hier bei der Angabe der Gleichung zur Spezifikation der Semantik von `lt` auf Paaren ausgenützt, daß wir gemäß der Semantik des Tupelkonstruktes alle Paare in der Form `(x,y)` schreiben können. Wir hätten alternativ aber auch die Projektionsoperatoren von Tupeln verwenden können. Mit diesen würde eine Gleichung zur Angabe der Semantik von `lt` lauten:

```
var   p1 : pair; p2 : pair;
eqn   p1 lt p2 <=> #1(p1) <= #1(p2) andalso
                   #2(p1) <= #2(p2);
```

Als nächstes wollen wir uns mit der Bildung von Unionen in SNL befassen. Hier ist der Grundgedanke, daß man aus mathematischer Sicht die disjunkte Summe der Wertebereiche mehrerer Typen bilden möchte. Hierzu führen wir für jedes $k \in N$, $j \in N$ die folgenden SMARAGD-internen Funktoren ein.

```
functor SUM_K+J(A:spec sort s1; ... ; sk end) = spec
   sort       sum;
   const      c1 : sum; c2 : sum; ...;cj : sum;
   opn        make1 : A.s1 -> sum;
              make2 : A.s2 -> sum;
              ...
              makek : A.sk -> sum;
   var        x1 : A.s1; x2 : A.s2; ... ; xk : A.sk;
   eqn        c1 <>c2; c1<>c3; ...; c1<>cj;
              c2<>c3;...;c2<.cj;
              ...
              ck-1<>ck;
              make1(x1)<>c1; ...; make1(x1)<>cj;
              make2(x2)<>c1; ...; make2(x1)<>cj;
              ...
              makek(xk)<>c1; ...; makek(xk)<>cj;
              make1(x1)<>make2(x2); ... make1(x1)<>makek(xk);
              make2(x2)<>make3(x3); ...; make2(x2)<>makek(xk);
              ...
              makek-1(xk-1)<>makek(xk);
end
```

Im Fall von *k=0* habe dabei SUM_K+J eine leere Parameterspezifikation und damit auch keine Sorten *si* und keine Operatoren *makei*. Im Fall von *j=0* habe die Spezifikation keine Konstantensymbole *ci*. Die Gleichungen definieren als Semantik nur, daß die durch die Konstantensymbole und die *make1*, .., *makek* erzeugten Elemente der Sorte *sum* alle verschieden voneinander sind, so daß wir im Fall termerzeugter Modelle davon ausgehen können, daß die zu *sum* gehörige Wertemenge isomorph zu der disjunkten Vereinigung der Mengen von Termen ist, die von den Konstanten bzw. den Konstruktoroperatoren *make1*, .., *makek* erzeugt werden. Wir verbinden in der Standardrealisierung daher stets mit Unionen die Semantik der durch diese disjunkte Summe gegebenen Termalgebra. Das Gleichheitsprädikat ist dann auf dieser Menge offensichtlich so definiert, daß zwei Elemente genau dann gleich sind, wenn ihre Darstellung als Terme (sie werden durch den gleichen String beschrieben) gleich sind.

Mit Hilfe der obigen SMARAGD-internen Funktoren definieren wir nun die Semantik von Unionen innerhalb von SNL durch die folgende Transformationsregel.

Def. 20 (Definition von Unionen)

Eine Union wird in SNL über ein Sprachkonstrukt der Gestalt

```
union u = a1 | a2 | ... | aj | con1 of t1 | ..| conk of tk;
```

definiert, wobei ein solches Konstrukt per Definition äquivalent zu dem folgenden SNL-internen Codeteil ist:

```
include   SUM_k+j(spec sort s1=t1; ...; sk=tk end)
          with      u for sum,
                    a1 for c1, ..., aj for cj,
                    con1 for make1,...,conk for makek
          end;
```

Um die Benutzung von Unionen innerhalb unserer Sprache zuzulassen, führen wir eine neue Form von Deklarationsteilen *spec_union_decls* in SNL ein. Wir erweitern die Syntax von SNL daher um die folgenden Konstrukte:

spec_sig_decls:	*spec_sort_decls* *spec_const_decls* *spec_opn_decls* *spec_ptl_decls* *spec_pred_decls* *spec_decls* *spec_include_decls* *spec_union_decls*
spec_union_decls:	**union** *spec_union_decl* { ; *spec_union_decl* }$_0^*$ [;]
spec_union_decl:	*sort_id* = *union_alternative* { | *union_alternative* }$_0^*$
union_alternative:	*const_id* *con_id* **of** *sort_type*
con_id:	**id**

Da über eine Unionsdeklaration neue Elemente der Signatur einer Spezifikation (z.B. die Konstanten und Konstruktoroperationen) definiert werden, die extern innerhalb der Signatur der Spezifikation sichtbar sind, führen wir auch einen Unionsdeklarationsteil innerhalb von Signaturen ein.

sig_decls:	*sig_sort_decls* *sig_const_decls* *sig_opn_decls* *sig_pred_decls* *sig_decls* *sig_include_decls* *sig_union_decls*
sig_union_decls:	**union** *sig_union_decl* { ; *sig_union_decl* }$_0^*$
sig_union_decl:	*sort_id* = *union_alternative* { | *union_alternative* }$_0^*$

Wir wollen uns einmal ein Beispiel zur Verwendung von Unionen ansehen.

Beispiel 28 (Verwendung von Unionen)

(a) Unsere Spezifikation BOOL läßt sich unter Verwendung einer Union wie folgt vereinfachen:

```
specification BOOL = spec
     union  bool = true | false;
     opn    not       : bool -> bool;
            andalso:  : bool * bool -> bool;
            orelse    : bool * bool -> bool;
     infix  andalso 2; orelse 1;
     var    x : bool; y : bool;
     eqn    not true = false;
            not false = true;
```

```
                    true andalso x = x;
                    false andalso x = false;
                    x orelse y = not (not x andalso not y);
        end
```

(b) Wir nehmen einmal an, daß wir einen Datentyp spezifizieren wollen, der Personen durch Angabe ihrer Namen, ihres Geschlechtes, ihres Alters und ihres Gewichtes beschreiben soll. Grundlage für eine solche Datentypspezifikation könnten dann die folgenden Unionsdeklarationen bieten.

```
union NAME      = Name of string;
union SEX       = Male | Female;
union AGE       = Age of int;
union WEIGHT    = Weight of int;
```

Mit diesen Unionen könnte man nun eine Person durch die folgende Union beschreiben:

```
union PERSON    Person of NAME * SEX * AGE * WEIGHT
```

Personen könnten dann z.B. durch die folgenden Terme beschrieben werden:

```
Person(Name "Mary", Female, Age 27, Weight 58)

Person(Name "John", Male, Age 23, Weight 73)
```

Innerhalb der Struktur PERSON könnte man dann weitere Operationen und Prädikate zum Umgang mit Personen einführen. Hier wären vor allem Projektionsoperatoren sinnvoll, die bei gegebener Person ihre einzelne Attribute extrahieren. Eine Spezifikation könnte z.B. lauten:

```
specification PERSON = spec
    union   NAME = Name of string;
    union   SEX = Male | Female;
    union   AGE = Age of int;
    union   WEIGHT = Weight of int;
    union   PERSON = Person of NAME*SEX*AGE*WEIGHT;
    opn     give_name: PERSON -> NAME;
            give_sex : PERSON -> SEX;
            give_age : PERSON -> AGE;
            give_weight : PERSON -> WEIGHT;
    var     name : NAME; age : AGE;
            sex : SEX; weight : WEIGHT;
    eqn     give_name(Person(name,sex,age,weight))=name;
            give_sex(Person(name,sex,age,weight))=sex;
            give_age(Person(name,sex,age,weight))=age;
            give_weight(Person(name,sex,age,weight))=weight;
end
```

Unsere Transformationsregel zur Erklärung von Unionen reicht weiter aus, um selbst rekursive Unionsdeklarationen zu erklären, die auf der rechten Seite als Argument von Konstruktoroperatoren der zu definierenden Unionssorte diese Sorte selbst zulassen. Das folgende Beispiel zeigt die Verwendung solcher rekursiven Unionsdeklarationen.

Beispiel 29 (Spezifikation natürlicher Zahlen über eine rekursive Union)

(a) Die natürlichen Zahlen lassen sich durch die folgende Union beschreiben.

```
specification NAT = spec
     union      nat = 0 | succ of nat;
     opn        + : nat * nat -> nat;
     pred       <= : nat * nat;
     infix      + 6; <= 4;
     var        m : nat; n : nat;
     eqn        n + 0 = n;
```

```
                    n + succ m = succ(n+m);
                    m <= m + n;
                    not n + (succ m) <= n;
        end
```

Gemäß der Transformationsregel für Unionen ist die obige Unionsdeklaration äquivalent zu dem folgenden Codesegment.

```
sort        nat;
const       0 : nat;
opn         succ : nat -> nat;
var         x : nat; y : nat;
eqn         0 <> succ x;
            succ x = succ y <=> x = y;
```

Man sieht hier, daß rekursive Unionsdeklarationen durch rekursive Konstruktoren (hier `succ`) beschrieben werden, was zu keinerlei Problemen führt.

(b) Weiter können wir unseren Funktor `STACK` mit Hilfe einer Union wie folgt beschreiben:

```
functor STACK ( E : spec sort elem; end) = spec
     sort    elem = E.elem;
     union   stack = empty | push of stack * elem;
     ptl     pop : (w:stack | w <> empty) -> stack;
             top : (w:stack | w <> empty) -> elem;
     var     st : stack; e : elem;
     eqn     pop(push(st,e))=st;
             top(push(st,e))=e;
end
```

Analog zu `STACK` kann man auch den Funktor `FIFO` mit Hilfe einer Union

```
union fifo = empty | put of fifo * elem
```

definieren.

Als weiteren Mode wollen wir nun noch in SNL Listen einführen. Ausgangspunkt zur Angabe von Listen innerhalb von SNL ist der folgende SNL interne Funktor.

```
functor LIST(A : spec sort s1 end) = spec
     union   'A.s1 list' = nil | :: of (A.s1 * 'A.s1 list');
     infixr  5 ::;
     opn     @ : 'A.s1 list' * 'A.s1 list' -> 'A.s1 list';
     infixr  5 @;
     ptl     hd : (x : 'A.s1 list' | x <> nil) -> 'A.s1 list';
             tl : (x : 'A.s1 list' | x <> nil) -> 'A.s1 list';
     opn     rev : 'A.s1 list' -> 'A.s1 list';
             length : 'A.s1 list'' -> int;
     ptl     nth : ((x,i) : 'A.s1 list' * int | 0 <= i andalso
                            length(x) < i) -> A.s1;
             nthtail : ((x,i) : 'A.s1 list' * int | 0 <= i andalso
                            length(x) < i) -> 'A.s1 list';

     var     a : A.s1; b : A.s1;
             x : 'A.s1 list'; y : 'A.s1 list';
             n : int;
     eqn     nil @ y = y;
             (a :: x) @ y = a :: (x @ y);
             hd (a :: x) = a;
             tl (a :: x) = x;
             rev (a :: nil) = a :: nil;
             rev (a :: x) = rev (x) @ (a :: nil);
             length (nil) = 0;
             length ( a :: x) = 1 + length(x);
```

```
        nth (a :: x,0) = a;
        nth (a :: x, n) = nth(x,n-1);
        nthtail (a :: x,0) = x;
        nthtail (a :: x,n) = nthtail(x,n-1);
end;
```

Die Verwendung von Listen in SNL wird dann über die folgende Transformationsregel geregelt.

Def. 21 (Definition von Listen)

Eine Liste wird in SNL über ein Konstrukt der Gestalt

```
sort mlist = m1 list
```

eingeführt. Dieses Konstrukt ist äquivalent zu dem Codeteil

```
include LIST(spec sort s1=m1 end)
     with 'm1 list' for 'A.s1 list' end;
sort mlist = `m1 list;
```

Man beachte dabei, daß die in der Signatur des Funktors LIST vorkommenden Symbole per Definition wiederum in ihrer Bedeutung überladen sind, d.h. für verschiedene Listen die gleichen Operationssymbole verwendet werden können.

Zur einfacheren Schreibweise von Listentermen führen wir weiter die Schreibweisen $[a_1,a_2,...,a_k]$ für $a_1 :: a_2 :: ... :: a_k$ und [] für nil ein.

Wir erweitern daher die SNL-Syntax wie folgt:

spec_sort_decl:	*sort_id* [= *sort_type*]
sort_type:	*sort_name*
	sort_type { * *sort_type* }$_0^*$
	{ *label* : *sort_type* { , *label* : *sort_type* }$_0^*$ }
	sort_type **list**
term:	*opn_name term*
	term opn_name term
	const_name
	var_id
	(*term* { , *term* }$_0^*$)
	{ *label* = *term* { , *label* = *term* }$_0^*$ }
	[*term* { , *term* }$_0^*$]
	[]

Wir wollen uns die Verwendung des Listenmodes einmal im folgenden Beispiel ansehen.

Beispiel 30 (Verwendung von Listen)

Unter Verwendung von Beispiel 28 können wir jetzt Listen von Personen durch die folgende Sortendefinition definieren.

```
sort PERSONS = PERSON list;
```

Eine solche Liste von Personen könnte dann gemäß der SNL Syntax für Terme z.B. in der Form

```
[Person(Name "Mary", Female, Age 27, Weight 58),
Person(Name "John", Male, Age 23, Weight 73) ]
```

angegeben werden. Wenn wir nun einen Operator einführen wollen, der aus einer Liste von Personen alle Einträge, die sich auf Frauen beziehen, extrahiert, könnten wir dies wie folgt realisieren:

```
opn  girls : PERSONS -> PERSONS;
var  a : PERSON; x : NAME; y : AGE;
     z : WEIGHT; lst : PERSONS;
eqn  girls (nil) = nil;
     if (give_sex(a) = Female)
           then girls(a :: lst)=a :: girls(lst)
     else girls(a :: lst)=girls(lst);
```

3.4 Realisation von SNL-Spezifikationen in ML

Zur Analyse von SNL-Netz-Spezifikationen, deren Beschriftung auf der Grundlage abstrakter Datentypspezifikationen in SNL definiert sind, benötigen wir konkrete Realisationen der zugehörigen SNL-Spezifikationen innerhalb einer interpretativen Programmiersprache, damit wir zur Bestimmung von Schaltvorgängen eines Netzes dessen Beschriftungselemente über einen Interpreter der gewählten Sprache auswerten können.

Die gewählte Programmiersprache sollte dabei eine möglichst natürliche Realisation von SNL-Spezifikationen erlauben. Diese Bedingung wird von der Programmiersprache ML hervorragend erfüllt, da das Modulkonzept von ML an die mathematische Theorie algebraischer Systeme angelehnt ist, was sich schon in den Namen der Modulkonstrukte von ML, nämlich Signatur, Struktur und Funktor, zeigt (Man siehe hierzu [MacQueen85]).

Der Begriff der ML-Signatur entspricht im wesentlichen unserer Definition der Signatur von abstrakten Datentypen und ist ebenfalls an den mathematischen Begriff der Signatur eines algebraischen Systems angelehnt. Der Begriff der ML-Struktur basiert auf dem mathematischen Konzept eines algebraischen Systems. Eine ML-Struktur enthält Datentypdefinitionen (Definitionen der Wertebereiche eines algebraischen Systems) und Definitionen von Funktionen, die auf diesen Datentypen arbeiten. Die ML-Funktionen, die innerhalb einer ML-Struktur definiert sind, entsprechen den Operationen und Relationen eines algebraischen Systems, wobei man (wie üblich) boolesche Funktionen als Relationen auffaßt. ML-Funktoren sind schließlich an den Begriff eines Funktors zwischen Kategorien von algebraischen Systemen angelehnt.

Wir können uns im folgenden nicht jede Einzelheit der Syntax und Semantik der Sprache ML ansehen (man siehe hierzu z.B. [Harper86, Harper88, Harper89, Harper90]), werden aber die für uns wichtigen Teile der ML-Syntax vorstellen, damit wir dann die Regeln beschreiben können, mit deren Hilfe wir SNL-Spezifikationen in natürlicher Weise in ML-Strukturen transformieren können. Eine Reihe dieser Transformationsregeln kann man automatisieren, so daß in einer späteren Version unseres Werkzeuges SMARAGD der Benutzer durch solche automatischen Transformationsregeln bei der Realisation von SNL-Spezifikationen unterstützt werden kann. Wir halten eine automatische Transformation insbesondere dann für möglich, wenn sich der Benutzer bei der Spezifikation an vorgegebene Konstruktionsregeln hält. Im Verlaufe dieses Abschnittes werden wir einige mögliche Konstruktionsregeln aufzeigen und informell erläutern. Man beachte aber, daß in einer ersten Version von SMARAGD (und nur diese ist im wesentlichen Gegenstand dieser Arbeit) der Benutzer jedoch gezwungen sein wird, diese Realisierungen komplett noch von Hand durchzuführen, so daß das Ziel dieses Abschnittes nicht die Konzeption und theoretische Fundierung der Konstruktions- und Transformationsregeln ist, sondern nur eine Anleitung zur manuellen Transformation von SNL-Spezifikationen in zugehörige ML-Strukturen geben soll, die auch keineswegs vollständig sein wird. Die Konzeption der Transformationskomponente wird einer späteren Arbeit vorbehalten bleiben.

3.4.1 SNL-Signaturen und ML-Signaturen

Wir beginnen bei der Vorstellung der ML-Modulkonzepte mit dem Begriff der ML-Signatur. Die Syntax für eine ML-Signatur ist:

ml_sig_decl:	**signature** *sig_id* = *ml_sig* { **and** *sig_id* = *ml_sig* }$_0^*$
sig_id:	**id**
ml_sig:	*sig_id*
	sig *ml_sig_body* **end**
ml_sig_body:	{ *ml_sig_body_decl* }$_0^*$
ml_sig_body_decl:	*ml_sig_datatype_decls*
	ml_sig_type_decls
	ml_sig_val_decls
	ml_sig_exception_decls
	ml_sig_structure_decls

Aus der obigen Syntax ersieht man, daß sich die äußere Form einer SNL-Signatur in fast identischer Weise auf eine ML-Signatur abbilden läßt. Man muß hier nur das Trennzeichen ; von SNL durch das Schlüsselwort **and** von ML ersetzen.

Die Deklarationsteile im Rumpf einer SNL-Signatur lassen sich einfach auf zugehörige Deklarationsteile einer ML-Signatur abbilden. Ein SNL-Signatursortendeklarationsteil (*sig_sort_decls*) kann man in ML einen ML-Signaturtypdeklarationsteil (*ml_sig_type_decls*) zuordnen. Ein SNL-Signaturunionsdeklarationsteil (*sig_union_dels*) kann man in ML auf einen ML-Signaturdatentypdeklarationsteil (*ml_sig_datatype_decls*) abbilden. *sig_const_decls*, *sig_opn_decls* und *sig_pred_decls* werden in ML durch *ml_sig_val_decls* beschrieben. Die An-

gabe partieller Operatoren (*sig_ptl_decls*) werden in ML auf Ausnahmedeklarationen (*ml_sig_exception_decls*) und zugehörige Wertedeklarationen (*ml_sig_val_decls*) abgebildet. Unterspezifikationsdeklarationen (*sig_spec_decls*) werden schließlich auf ML-Strukturdeklarationen (*ml_sig_structure_decls*) abgebildet.

Bei den einzelnen Deklarationsteilen (bis auf *sig_ptl_decls*) sind nur Schlüsselwörter und das Trennzeichen ; umzusetzen bzw. wegzulassen. Ein ML-Signaturtypdeklarationsteil hat die Gestalt:

ml_sig_type_decls:	**type** *ml_sig_type_decl* { **and** *ml_sig_type_decl* }$_0^*$
ml_sig_type_decl:	[*type_var_seq*] *type_id*
type_id:	**id**
type_var_seq:	*type_var*
	(*type_var* { , *type_var* }$_0^*$)
type_var:	**type_var_id**

Wenn wir hier die optionale Typvariablensequenz, die zur Angabe polymorpher Typen dient, ignorieren, entspricht eine SNL-Sortendeklaration einer ML-Typdeklaration, wenn wir das Schlüsselwort **sort** durch das Schlüsselwort **type** und das Trennzeichen ; in SNL durch das Schlüsselwort **and** in ML ersetzen und das optionale ; am Ende von SNL-Sortendeklarationsteilen weglassen.

ML-Signaturdatentypdeklarationsteile haben die Gestalt:

ml_sig_datatype_decls:	*ml_datatype_decls*
ml_datatype_decls:	**datatype** *ml_datatype_decl* { **and** *ml_datatype_decl* }$_0^*$
ml_datatype_decl:	[*type_var_seq*] *type_id* = *constrs*
constrs:	*constr* { | *constr* }$_0^*$
constr:	[**op**] **con** [**of** *ml_type*]
ml_type:	*type_var*
	[*type_seq*] *type_id*
	{ *label* : *ml_type* { , *label* : *ml_type* }$_0^*$ }
	ml_type { * *ml_type* }$_0^*$
	ml_type -> *ml_type*
	(*ml_type*)
type_seq:	*type* { *type* }$_0^*$

Wenn man beachtet, daß sich die SNL-Sortentypangaben (in *sort_type*) alle in identischer Form einer ML-Typangabe (in *ml_type*) zuordnen lassen, so sieht man leicht, daß man einen SNL-Unionsdeklarationsteil durch Umsetzen des Schlüsselwortes **union** nach **datatype** und durch Ersetzen von ; durch **and** in einen ML-Datentypdeklarationsteil konvertieren kann. Die Semantik stimmt bei dieser Konvertierung ebenfalls, da die beteiligten internen SMARAGD-Funktoren für Tupel, Unionen, Rekords und Listen in ML konsistent durch die eingebauten Tupel-, Rekord-, Unions- und Listenkonstruktoren implementiert werden.

ML-Signaturwertedeklarationsteile (*ml_val_decls*) haben die folgende Syntax:

ml_sig_val_decls:	**val** *ml_sig_val_decl* { **and** *ml_sig_val_decl* }$_0^*$
ml_sig_val_decl:	[**op**] *val_id* : *ml_type*

Eine SNL-Konstanten, -Operator- oder -Prädikatdeklaration *const_id* : *const_type*, *opn_id* : *opn_type* oder *pred_id* : *pred_type* entspricht abgesehen von dem optionalen ML-Schlüsselwort **op** einer ML-Wertedeklaration *val_id* : *ml_type*. Da die Syntax für einen ML-Typ (*ml_type*) insbesondere die Angabe eines Typnamens *type_id* als Typ zuläßt, kann eine SNL-Konstantendeklaration der Gestalt *const_id* : *sort_name* daher in der Form *val_id* : *type_id* in äquivalenter Weise in ML dargestellt werden. Die ML-Syntax läßt über die Regeln *ml_type* : *ml_type* { * *ml_type* }$_0^*$ und *ml_type* : *ml_type* -> *ml_type* weiter Typangaben der Gestalt *type_id* * *type_id* * ... *type_id* -> *type_id* zu, so daß eine SNL-Operatordeklaration in identischer Form in ML dargestellt werden kann. SNL-Prädikate fassen wir in Bezug auf eine funktionale prädikatenlogische Realisation unserer SNL-Spezifikationen in ML als Funktionen mit dem Wertebereich **bool** auf. Den Typ *type_id* * *type_id* * ... *type_id* eines Prädikates *pred_id* notieren wir daher in ML in der Form *type_id* * *type_id* * ... *type_id* -> *bool*, wobei **bool** in ML standardmäßig als boolescher Datentyp mit den Werten **true** und **false** vordefiniert ist. SNL-Konstanten-, -Operator- und -Prädikatdeklarationen überträgt man also in äquivalente ML-Wertedeklarationsteile, indem man die Schlüsselworte **const**, **opn** und **pred** durch das ML-Schlüsselwort **val** und das Trennzeichen ; durch das ML-Schlüsselwort **and** ersetzt, hinter der Typangabe von Prädikaten die Zeichenfolge -> bool anfügt und das letzte Semikolon eines Deklarationsteils wegläßt.

ML-Signaturausnahmedeklarationsteile (*ml_sig_exception_decls*) haben die folgende Gestalt:

ml_sig_exception_decls: **exception** *ml_sig_exception_decl* { *ml_sig_exception_decl* }$_0^*$
ml_sig_exception_decl: *exception_id* : *ml_type*
exception_id: **id**

Da wir in SNL keine Typangabe für die Angabe der Partialität von Operatoren haben (Wir definieren hier nur, daß ein Operator partiell ist, geben aber nicht an, wie diese Ausnahme konkret innerhalb einer Interpretation der Spezifikation als Ausnahme behandelt werden soll), bilden wir solche partielle Operatoren stets auf Ausnahmen mit dem generischen Typ *unit* ab, d.h. eine Vereinbarung der Gestalt

```
ptl get : fifo -> item; rest:fifo -> fifo;
```

bilden wir in ML auf

```
exception get : unit; rest : unit;
val get : fifo -> item; rest : fifo -> fifo;
```

ab.

Wir müssen jetzt noch klären, wie man einen Spezifikationsdeklarationsteil von SNL-Signaturen auf Strukturdeklarationsteile in ML-Signaturen überträgt. Die Syntax für einen ML-Signaturstrukturdeklarationsteil ist:

ml_sig_structure_decls: **structure** *ml_sig_structure_decl* { **and** *ml_sig_structure_decl* }$_0^*$
ml_sig_structure_decl: *structure_id* : *ml_sig*

Wiederum ist offensichtlich nur das Schlüsselwort **include** in SNL durch das Schlüsselwort **structure** in ML und das Trennzeichen ; durch das ML-Schlüsselwort **and** zu ersetzen und ein abschließendes ; ganz wegzulassen.

Wir wollen einmal ein paar Beispiele für die Konvertierung von SNL-Signaturen in ML-Signaturen betrachten.

Beispiel 31 (Beispiele für die Konvertierung von SNL-Signaturen in ML-Signaturen)

(a) Die SNL-Signatur `BOOL_A` hat in ML die Gestalt:

```
signature BOOL = sig
     type   bool
     val    true : bool
      and   false : bool
     val    !     : bool -> bool
      and   ||    : bool * bool -> bool
      and   &&    : bool * bool -> bool
end
```

Wenn wir eine Konstruktorform der Spezifikation wählen, können wir diese in eine ML-Signatur der Form

```
signature BOOL = sig
     datatype  bool = true | false
     val       ! : bool -> bool
      and      || : bool * bool -> bool
      and      && : bool * bool -> bool
end
```

übertragen.

(b) Die SNL-Signatur `NAT` hat in ML die Gestalt

```
signature NAT = sig
     type   nat
     val    0     : nat
     val    succ  : nat -> nat
      and   op +  : nat * nat -> nat
     val    op <= : nat * nat -> bool
end
```

Man beachte hierbei, daß die Verwendung des Schlüsselwortes **op** hier nötig ist, da sonst ML die Zeichenfolgen + und <=, da sie standardmäßig als *Infixoperatoren* definiert sind, nicht als Präfixoperatornamen erkennt. Die Konstruktorform der SNL-Signatur `NAT` kann man in ML auf die folgende Weise darstellen:

```
signature NAT = sig
     datatype  nat = 0 | succ of nat
     val       succ  : nat -> nat
      and      op +  : nat * nat -> nat
     val       op <= : nat * nat -> bool
end
```

(c) Die Konstruktorform von STACKNAT hat in ML die Gestalt:

```
signature STACKNAT = sig
     structure  N : NAT
     datatype   stack = empty | push of (stack,N.nat)
     exception  top : unit and pop : unit
     val        pop     : stack -> stack
      and       top     : stack -> N.nat
     val        isempty : stack -> bool
end
```

Damit haben wir vollständig beschrieben, wie wir SNL-Signaturen in ML-Signaturen konvertieren. Wir wollen uns nun ansehen, wie wir SNL-Spezifikationen durch ML-Strukturen realisieren können.

3.4.2 SNL-Spezifikationen und ML-Strukturen

Die Grundsyntax zur Angabe von ML-Strukturen lautet:

ml_structure_decls:	**structure** *ml_structure_decl* { **and** *ml_structure_decl* }$_0^*$
ml_structure_decl:	*structure_id* [: *ml_sig*] = *ml_structure*
structure_id:	**id**
ml_structure:	*structure_name*
	struct *structure_body* **end**
	functor_id (*actual_param_list*)
actual_param_list:	*ml_structure* { , *ml_structure* }$_0^*$
structure_name:	[*path_name*.]*structure_id*

Damit entspricht die äußere Form einer SNL-Spezifikationsdeklaration einem ML-Strukturdeklarationsteil (*ml_structure_decls*) mit genau einer Strukturdeklaration (*ml_structure_decl*), wenn man die SNL-Schlüsselworte **specification** und **spec** durch die ML-Schlüsselworte **structure** und **struct** ersetzt. Man beachte weiter, daß innerhalb einer ML-Strukturdeklaration die externe Schnittstelle der Struktur durch eine ML-Signatur angegeben werden kann. Die Semantik ist dabei analog zu der für SNL-Spezifikationen.

In einem ersten Schritt zur Realisierung einer SNL-Spezifikation in ML bietet es sich nun an, zunächst die sichtbare Schnittstelle einer SNL-Spezifikation in Form einer ML-Strukturdefinition unter Angabe einer ML-Signatur für die Struktur anzugeben und die Gleichungen, die die Semantik der einzelnen Elemente der Signatur beschreiben in Form von Kommentaren in dem zunächst leeren Rumpf der ML-Struktur festzuhalten. Das folgende Beispiel verdeutlicht diesen Schritt.

Beispiel 32 (Transformation einer SNL-Spezifikation in eine leere ML-Struktur mit korrekter Signatur)

Wir wollen einmal die Konstruktorform der Spezifikation BOOL_A in eine zunächst leere ML-Struktur mit korrekter ML-Signatur umwandeln und die semantischen Formeln im leeren Rumpf als Kommentare festhalten. Man erhält dann:

```
structure BOOL_A : sig
     datatype  bool = true | false
     val       ! : bool -> bool
      and      ||: bool * bool -> bool
      and      &&: bool * bool -> bool
end = struct
     (* semantic of sort bool is given as

           constructor pattern true | false
     datatype bool = true | false
     *)
     (* semantic of ! is defined in formulas
```

```
        eqn     ! (true) = false
                ! (false) = true
        *)
        (* semantic of && is defined through
        infix   && 2
        var     x : bool; y : bool
        eqn     true && x = x;
                false && x = false
        *)
        (* semantic of || is defined through
        infix   || 1
        var     x: bool; y : bool
        eqn     x || y = !(!x && !y)
        *)
end
```

Wir haben im obigen Beispiel die Formeln schon so gruppiert, daß die einzelnen Kommentare die Semantik einzelner Teile der SNL-Signatur beschreiben. Wir müssen nun nur noch die einzelnen Kommentare in geeignete ML-Konstrukte umsetzen.

Dazu betrachten wir zunächst einmal die Syntax für einen ML-Strukturrumpf genauer. Für unsere Zwecke können wir uns vorstellen, daß er die folgenden Kategorien von ML-Definitionen enthält: Datentypdefinitionen (*ml_type_decls* und *ml_datatype_decls*), Wertedefinitionen (*ml_funct_decls* und *ml_val_decls*), ML-Strukturdefinitionen (*ml_structure_decls*) und ML-Opendefinitionen (*ml_open_decls*), ML-Infixdeklarationsteile (*ml_infix_decls* und *ml_infixrdecls*), ML-Ausnahmedefinitionsteile (*ml_exception_decl*) und andere uns nicht näher interessierende Definitionen (*other_ml_defs*). Wir haben damit:

structure_body:	{ *structure_body_decls* }$_0^*$
structure_body_decls:	*ml_type_decls*
	ml_datatype_decls
	ml_val_decls
	ml_funct_decls
	ml_structure_decls
	ml_open_decls
	ml_infix_decls
	ml_infixr_decls
	ml_exception_decls
	other_ml_decls

Die *ml_type_decls* und *ml_datatype_decls* beinhalten die Möglichkeiten von ML Datentypen zu definieren. Wir benutzen sie daher zur Realisierung der Sorten und Unionen. *Die ml_funct_decls* und *ml_val_decls* beinhalten die Möglichkeiten von ML, Namen an Funktionen oder allgemeiner ML-Werten zu binden. Sie bieten uns daher Möglichkeiten zur Realisation von Konstanten, Operatoren und Prädikaten. *ml_funct_decls* erlauben uns in Verbindung mit *ml_exception_decls* weiter die Realisation partieller Operatoren.

ml_structure_decls und *ml_open_decls* stellen in ML Mechanismen bereit, andere Strukturen in ML-Strukturen als Unterstrukturen einzubinden bzw. zu öffnen. Wir benutzen sie daher zur Realisation der Unterspezifikations- bzw. Inkludedeklarationen. *ml_infix_decls* bzw. *ml_infixr_decls* korrespondieren in ihrer Semantik in ML schließlich mit den *spec_infix_decls* bzw. *spec_infixr_decls* von SNL.

Wir wollen uns nun die für die Realisation von Sorten relevanten Formen der Datentypdefinitionen ansehen. Diese haben die Syntax:

ml_type_decls:	**type** *ml_type_decl* { **and** *ml_type_decl* }$_0^*$
ml_type_decl:	[*type_var_seq*] *type_id* = *ml_type*
ml_datatype_decls:	**datatype** *ml_datatype_decl* { **and** *ml_datatype_decl* }$_0^*$
ml_datatype_decl:	[*type_var_seq*] *type_id* = *constrs*
constrs:	*constr* { | *constr* }$_0^*$
constr:	[**op**] **con** [**of** *ml_type*]

Wenn wir mal das optionale *type_var_seq* Konstrukt, daß man zur Angabe polymorpher Typen verwendet, ignorieren, so wird sowohl in einer ML-Typdefinition (*ml_type_decl*) als auch in einer ML-Datentypdefinition (*ml_datatype_decl*) einem Typnamen (*type_id*) ein ML-Typ (*ml_type* oder *constrs*) zugewiesen.

Unter die Klasse *ml_type*, deren Syntax wir bereits vorher angegeben haben, fallen zunächst einmal die Standardtypen von ML: bool, int, real, string, `a list, und unit. Die Klasse *ml_type* beinhaltet weiter die standardmäßig in ML vorhandenen abgeleiteten Datentypen Rekord (Syntax {*label*:*ml_type*,...,*label*:*ml_type*}), Tupel (Syntax *ml_type**...**ml_type*) und den Funktionstyp (Syntax *ml_type*->*ml_type*).

Mit Hilfe des Sprachkonstruktes *constrs* innerhalb einer ML-Datentypdeklaration (*ml_datatype_decl*) kann ein Benutzer neue Typen in ML definieren, deren Wertebereiche wir als Menge der Terme deuten können, die von den einzelnen Konstruktorpatterns (*constr*) innerhalb des *constrs* Teils frei erzeugt werden. Dies entspricht aber gerade der Semantik der SNL-Unionsdeklarationen.

Wenn wir uns bei der Definition von SNL-Spezifikationen auf Spezifikationen beschränken, deren Sorten nur aus den eingebauten Standardtypen bool, int und string durch die Rekord-, Tupel-, Listen- und Unionskonstruktoren konstruierbar sind, so lassen sich die zugehörigen Sorten- und Unionsdeklarationen auf zugehörige Typ- und Datentypdeklarationen in ML durch Umsetzen von Schlüsselwörtern abbilden, die diese Sorten und Unionen realisieren, da die Syntax und Semantik dieser eingebauten Datenkonstrukte so gewählt ist, daß die zugehörigen ML-Konstrukte passende Modelle liefern. Wie einfach die Umsetzung ist, zeigt das folgende Beispiel.

Beispiel 33 (Realisation von Sorten- und Unionsdeklarationen)

(a) Die Unionsdeklaration der Spezifikation BOOL

```
union bool = true | false;
```

läßt sich durch die Datentypdefinition

```
datatype bool = true | false
```

in ML realisieren.

(b) Die Unionsdeklaration innerhalb der NAT Spezifikation

```
union nat = 0 | succ of nat;
```

ergibt in ML die Realisation

```
datatype nat = 0 | succ of nat
```

(c) Die Liste der SNL-Deklarationen

```
union NAME = Name of string;
union SEX = Male | Female;
union AGE = Age of int;
union WEIGHT = Weight of int;
union PERSON = Person of NAME * SEX * AGE * WEIGHT;
sort PERSONS = PERSON list;
```

läßt sich in ML durch die Deklarationssequenz

```
datatype NAME = Name of string
datatype SEX = Male | Female
datatype AGE = Age of int
datatype WEIGHT = Weight of int
datatype PERSON = Person of NAME * SEX * AGE * WEIGHT
type PERSONS = PERSON list
```

realisieren.

Ist eine Sorte innerhalb einer Spezifikation nicht, wie oben beschrieben, aus den Standardtypen und Typkonstruktoren konstruiert, so läßt sie sich nicht so leicht auf eine ML-Typ- bzw. Datentypdeklaration abbilden. In diesem Fall muß man die Semantik der Gleichungen auswerten, die Konstruktoren für die Werte der betreffenden Sorte darstellen, was sehr schwierig bzw. unmöglich sein kann. Wir werden daher in SMARAGD in einem nächsten Prototyp zunächst nur die automatische Transformation solcher Sortendeklarationen implementieren, die aus den eingebauten Standardtypen bool, int und string durch die Rekord-, Tupel-, Listen- und Unionskonstruktoren konstruierbar sind. Ein reiner Anwender des SMARAGD Systems sollte sich bei der Spezifikation auf solche Sortendeklarationen beschränken können. Weiter werden wir einen Expertenmodus zulassen, der es gestattet, daß ein Anwender in Spezifikationen zu Sorten, die nicht automatisch realisiert werden können, deren ML-Realisierung per Hand eingibt.

Nachdem wir nun die Realisation der Sorten- und Unionsdeklarationen erläutert haben, wollen wir auf die Realisation der Konstanten, Operatoren und Prädikate eingehen. Eine erste Idee zur Automatisierung ist hierbei, daß

für Spezifikationen, die nur nach den oben beschriebenen Regeln konstruierte Sorten enthalten, die Semantik der Konstanten, Operatoren und Prädikate in natürlicher Weise durch Gleichungen beschreibbar ist, die sich an der Konstruktionsweise der Sorten anlehnen. Wenn ich z. B. die Semantik eines Operators `f` auf der Sorte `nat` definiert durch das Konstruktionsschema `union nat = 0 | succ of nat` beschreiben will, so muß ich einmal die Semantik des Operators auf dem Konstruktor `0` und zum zweiten die Semantik des Operators bzgl. der Werte von `nat`, die durch das Konstruktionsschema `succ of nat` beschrieben werden, angeben. Dies liefert mir in natürlicher Weise ein rekursives Gleichungsschema zur Definition der Semantik von `f` von der Gestalt:

```
eqn f(0) = ?
var x : nat
eqn f(succ(x)) = ?.
```

Allgemeiner kann man oftmals die Semantik einer Funktion, die auf einem Datentyp mit n-Konstruktoren definiert ist, durch n-Gleichungen beschreiben, die jede für sich die Semantik des Operators bzgl. einem Konstruktor beschreiben. In ML gibt es nun die Möglichkeit, Funktionen durch Angabe von Alternativen für ihre Werte auf verschiedenen Wertebereichen, die durch Konstruktionsschema (Pattern) für die Werte beschrieben werden, zu definieren. Dies geschieht durch eine Definition von der Gestalt

```
fun f( pattern_1) = body_1
     | f(pattern_2) = body_2
     | .....
     .....
     | f(pattern_n) = body_n
```

Auf diese Weise läßt sich nun leicht die Semantik von solchen SNL-Operatoren und Prädikaten in ML übertragen, deren Semantik durch an die Konstruktionsschema zugehöriger Argumentsorten angelehnte Gleichungen beschrieben werden. Wir wollen dies einmal in den folgenden Beispielen verdeutlichen.

Beispiel 34 (Realisation von Operatoren und Prädikaten in ML)

(a) Ein Beispiel für die soeben beschriebene Vorgehensweise liefert die folgende Realisation von `BOOL`.

```
structure BOOL = struct
     datatype bool = true | false
     fun       not true = false
             | not false = true
     infix     andalso 2 and orelse 1
     fun       true andalso x = x
             | false andalso x = false
     fun       x orelse y = not (not x andalso not y)
end
```

(b) In der Konstruktorform der `NAT` Spezifikation ist die Semantik des Operators + über die Gleichungen

```
n + 0 = n;
n + succ(m)= succ(n + m);
```

beschrieben. Man beachte dabei, daß die erste Gleichung sich für das zweite Argument von + auf das Konstruktorpattern 0 und die zweite Gleichung auf das Konstruktorpattern `succ(m)` bezieht. In ML ergibt sich daher eine Realisierung von + durch den Codeteil

```
infix + 6;
fun n + 0 = n
     | n + succ(m) = succ(n+m)
```

Die beschriebene Vorgehensweise läßt sich auch auf Spezifikationen mit partiellen Operatoren übertragen, wenn die Ausnahmebereiche der partiellen Operatoren über ein Konstruktorpattern der zu Grunde gelegten Sorte definiert sind. In diesem Fall realisiert man den "Wert" eines Operators `f` auf seinem Ausnahmebereich(en) als Aufruf einer Ausnahmebedingung von der Gestalt `raise f`. Dies führt beim Aufruf einer solchen partiellen Funktion auf einem Ausnahmewert dazu, daß vom ML-Interpreter kontrolliert eine Fehlermeldung ausgegeben wird, die besagt, daß der Wert von `f` auf dem angegebenen Argument nicht definiert ist. Das folgende Beispiel verdeutlicht die Behandlung von partiellen Operatoren in Bezug auf Ausnahmebereiche, die durch Konstruktorpatterns definiert sind, am Beispiel des Funktors `STACK`.

Beispiel 35 (Realisation von STACK)

Die Konstruktorform des Funktors `STACK` läßt sich wie folgt realisieren.

```
functor STACK (E: sig type elem end) = struct
     type        elem = E.elem
     datatype    stack = empty | push of stack * elem
     exception   pop and top
     fun         pop(empty) = raise pop
               | pop(push(st,e))=st
     and         top(empty) = raise top
               | top(push(st,e)) = e
end
```

Wenn wir die bislang entwickelte Methode zur automatischen Realisation von Operatoren weiter ausbauen wollen, müssen wir nun Wege finden, wie wir die Gleichungen der Spezifikation in eine solche Form bringen können, daß sie als Alternativen innerhalb einer ML-Funktionsdefinition dienen können. Die zur Realisation in ML für uns geeignete Form einer ML-Funktionsalternative (wir betrachten hier der Übersichtlichkeit halber keine Infixoperatoren) ist von der Gestalt:

```
f (pattern_1, pattern_2, ..., pattern_n) = expr
```

Es sollte an dieser Stelle klar sein, daß unser prädikatenlogischer Kalkül mit Quantoren im allgemeinen keine solche Transformation erlaubt, vielmehr hier Einschränkungen nötig sind. Gleichungen im gleichungsorientierten Kalkül sollten dagegen unter nicht zu restriktiven Voraussetzungen in die obige Form transformiert werden können. Weiter kann man jede Formel der Gestalt:

```
condition => f (pattern_1, ..., pattern_n) = expr
```

als andere Form einer Funktionsdeklaration der Gestalt

```
f (pattern_1, ..., pattern_n) = if ( condition )
                    then expr
```

ansehen. Das folgende Beispiel verdeutlicht die Möglichkeiten, die mit dieser Transformation einhergehen.

Beispiel 36 (Realisation der Funktion isgirl aus Beispiel 30)

Die Semantik des Operators `girls` aus Beispiel 30 läßt sich mit der oben beschriebenen Funktion wie folgt beschreiben:

```
fun girls(nil) = nil
 |girls(a :: lst) = if (give_sex(a) = Female)
                     then a :: girls(lst)
                     else girls(lst)
```

Die Unterspezifikationsdeklarationen und Inkludedeklarationen innerhalb von SNL entsprechen weitgehend den Konzepten von Unterstrukturen und der Open-Deklaration von ML. Die einzige Erweiterung ist das Konzept der Umbenennung von Elementen einer Struktur innerhalb einer Unterspezifikations- oder Inkludedeklaration, für die es kein analoges Konstrukt in ML gibt. Umbenennung läßt sich aber stets auf die Weise in ML realisieren, daß man einer Unterstruktur- oder Open-Deklaration eine Reihe von Wertedeklarationen folgen läßt, die die neuen Namen an die bereits gegebenen Elemente der eingebetteten Struktur binden. Wir wollen auch hierzu ein paar Beispiele betrachten.

Beispiel 37 (Realisation von Unterspezifikations- und Inkludedeklarationen)

(a) Der Funktor `SET` aus Beispiel 22 läßt sich wie folgt realisieren. Man beachte, daß hier die implizite Einbettung der Parameterstruktur `M` wie in SNL auch erfolgt.

```
functor SET ( M : MONOID) : sig
    type          elem
    datatype      set emptyset | incorp of set * elem
    val           total : set -> elem
    val           contains : set * elem -> bool
end = struct
    type          elem = M.elem
    fun           contains(emptyset,x)=false
          |       contains(incorp(s,x),x)= true
```

```
          |     contains(incorp(s,x),y) = contains(s,y)
    fun           total(emptyset) = M.e
          |     total(incorp(s,x)= if (contains(s,x))
                                then total(s)
                                else M.op(total(s),x)
end
```

Wir setzen dabei voraus, daß vorher bereits eine ML-Signatur MONOID definiert wurde, die eine Realisation der Parameterspezifikation MONOID der Spezifikation SET darstellt. Man beachte weiter, daß wir hier die Sorte set durch eine Datentypdefinition mit Konstruktoren emptyset und incorp ersetzt haben.

(b) MSET aus Beispiel 25 läßt sich mit dem Funktor nach (a) wie folgt realisieren:

```
functor MSET (M : MSET_PARAM) : sig
  type          set and elem
  val           emptyset : set
  val           incorp : set * elem -> set
      and       total : set -> elem
  exception     min
  val           min : set -> elem
  val           contains : set * elem -> bool
end =
struct
  open          SET(M)
  exception     min
  fun           min (emptyset) = raise min
          |     min(incorp(s,x)) = if ( s = emptyset) then x
                                  else if (M.le(x,min(s))then x
                                  else min(s)
end
```

(c) Beispiel 24 (b) ergibt dann bei entsprechender Realisierung der Spezifikation NAT_MONOID und der Signatur NATSET für die Spezifikation NATSET die Realisation:

```
structure NATSET : NATSET = struct
     open    SET(NAT_MONOID)
     type    natset = set
     and     nat = elem
     val     sum = total
end
```

Damit wollen wir unsere kurzen Ausführungen über die Realisation von SNL-Spezifikationen beenden. Wir hoffen, daß dieser kurze Überblick dem Leser einen Eindruck vermittelt hat, wie ein zukünftiges Transformationssystem aufgebaut werden kann und welche Randbedingungen an es geknüpft sind. Im folgenden Kapitel wollen wir nun auf die Petri-Netz-Seite der Sprache SNL eingehen.

4. Das Petri-Netz-Modell

In diesem Kapitel wollen wir das unserer Spezifikations- und Modellierungssprache SNL zugrunde liegende Petri-Netz-Modell vorstellen und den Petri-Netz-Teil von SNL formal durch eine kontextfreie Grammatik beschreiben.

Wir müssen zunächst definieren, was wir unter formalen Summen verstehen, denn diese werden wir als Beschriftung von Kanten und zur Angabe von Markierungen des Netzes verwenden. Im Anschluß daran beschreiben wir, was wir unter einer SNL-Netz-Spezifikation verstehen. Die Realisierung von SNL-Netz-Spezifikationen liefert uns dann SNL-Systeme (SNL-Modelle).

4.1 Formale Summen und Multimengen

Die Markierung eines traditionellen Petri-Netzes besteht aus einer Anzahl von ununterscheidbaren Marken (black token), die auf Stellen liegen können. In Netzen mit individuellen Marken können wir nun Objekte als Individuen unterscheiden. Trotzdem ist es sinnvoll, daß man in einer Stellenmarkierung mehr als ein Exemplar ein und desselben Individuums auf einer Stelle zuläßt. Eine geeignete Formalisierung des Begriffs des Mehrfachvorkommens von Individuen innerhalb einer Menge bietet der Begriff der Multimenge (Man siehe z.B. [Hickman80])

Def. 22 (Multimenge)

Sei A im folgenden eine beliebige Menge.

(a) Eine Funktion $\mu: A \to \mathbb{Z}$ nennen wir im folgenden eine (*verallgemeinerte*) *Multimenge*. Die Menge aller Multimengen über der Menge A bezeichnen wir mit $MULT(A)$.

(b) Eine Funktion $\mu: A \to \mathbb{N}$ nennen wir (*eigentliche*) *Multimenge*. Die Teilmenge aller eigentlichen Multimengen von $MULT(A)$ bezeichnen wir mit $MULT^+(A)$.

(c) Für $\mu \in MULT(A)$ bezeichnen wir mit $Tr(\mu)$ die Trägermenge von μ, d.h. die Menge aller $x \in A$, für die $\mu(x)$ ungleich 0 ist.

(d) $MULT_{fin}(A)$ bzw. $MULT^+_{fin}(A)$ seien die Teilmengen von $MULT(A)$ bzw. $MULT^+(A)$, die nur Funktionen mit endlicher Trägermenge enthalten.

Man kann sich eine eigentliche Multimenge $\mu: A \to \mathbb{N}$ als eine Verallgemeinerung des Mengenbegriffes so vorstellen, daß μ jedes Element $a \in A$ $\mu(a)$ mal enthält. Auf Multimengen lassen sich die folgenden Operationen einführen.

Def. 23 (Operationen auf Multimengen)

Sei A wiederum eine beliebige Menge.

(a) Die *Addition von Multimengen* ist eine Operation $+: MULT(A) \times MULT(A) \to MULT(A)$, die wie folgt definiert ist: Für je zwei $\mu, \nu \in MULT(A)$ und jedes $a \in A$ ist $(\mu+\nu)(a)=\mu(a)+\nu(a)$.

(b) Die *Subtraktion von Multimengen* ist eine Operation $-: MULT(A) \times MULT(A) \to MULT(A)$, die wie folgt definiert ist: Für je zwei $\mu, \nu \in MULT(A)$ und jedes $a \in A$ ist $(\mu-\nu)(a)=\mu(a)-\nu(a)$.

(c) Die *Skalarmultiplikation einer Multimenge mit einer ganzen Zahl* ist eine Operation $\cdot: \mathbb{Z} \times MULT(A) \to MULT(A)$, die für jedes $\mu \in MULT(A)$, $d \in \mathbb{Z}$ und alle $a \in A$ durch $(d\,\mu)(a)=d\,\mu(a)$ definiert ist.

Wir wollen einmal einige Rechenregeln im Umgang mit Multimengen zusammenstellen.

Bemerkung 1: (Rechenregeln im Umgang mit Multimengen)

Sei A eine Menge und $MULT(A)$ die Menge der Multimengen über A.

(a) $MULT(A)$ ist mit der oben eingeführten Addition und Skalarmultiplikation ein $\mathbb{Z}$-Modul, d.h. es gelten für alle $\mu, \nu, \delta \in MULT(A)$ und $d, e \in \mathbb{Z}$ die Regeln:

(1) $d(\mu+\nu)=(d\mu)+(d\nu)$, $(d+e)\mu=(d\mu)+(e\mu)$

(2) $(\mu+\nu)+\delta=\mu+(\nu+\delta)$

(3) $\mu+\nu=\nu+\mu$

(4) Mit $\overline{0}$ als die Multisumme mit $\overline{0}(a)=0$ für alle $a \in A$ haben wir $\overline{0}+\mu=\mu=\mu+\overline{0}$, $0\,\mu=\overline{0}$, $\mu+(-\mu)=(-\mu)+\mu=\overline{0}$

(b) Die Subtraktion läßt sich über die Addition durch $\mu-\nu=\mu+(-1)\nu$ darstellen.

$MULT_{fin}(A)$, $MULT^+(A)$ bzw. $MULT^+_{fin}(A)$ sind in natürlicher Weise in den $\mathbb{Z}$-Modul $MULT(A)$ eingebettet. $MULT_{fin}(A)$ ist dabei ein Untermodul von $MULT(A)$. Die Restriktionen von + und $\cdot$ bilden entprechend Operatio-

nen auf $MULT^+(A)$ bzw. $MULT^+_{fin}(A)$, die nicht aus $MULT^+(A)$ bzw. $MULT^+_{fin}(A)$ herausführen; $MULT^+(A)$ bzw. $MULT^+_{fin}(A)$ sind allerdings keine Untermodule von $MULT(A)$, da es in der Regel kein inverses Element zu einer Multimenge μ aus $MULT^+(A)$ bzw. $MULT^+_{fin}(A)$ in $MULT^+(A)$ bzw. $MULT^+_{fin}(A)$ gibt.

Wir wollen noch eine Ordnungsrelation auf Multimengen einführen.

Def. 24 (Halbordnung auf Multimengen)

Die Ordnungsrelation $\leq$ auf $MULT(A)$ sei wie folgt definiert: Für alle $\mu, \nu \in MULT(A)$ gilt $\mu \leq \nu$ genau dann, wenn $\mu(a) \leq \nu(a)$ für alle $a \in A$ ist. Die so eingeführte Ordnungsrelation überträgt sich analog auf $MULT_{fin}(A)$, $MULT^+(A)$ bzw. $MULT^+_{fin}(A)$.

$\leq$ ist nur eine Halbordnung auf $MULT(A)$, da nicht alle Elemente von $MULT(A)$ miteinander vergleichbar sind. Ansonsten gelten die für Ordnungsrelationen bekannten Rechenregeln.

Bemerkung 2:

Für alle $\mu, \nu, \delta \in MULT(A)$ gelten die Regeln:

(1) $\mu \leq \nu$ und $\nu \leq \mu \Rightarrow \mu = \nu$ (antisymmetrisch)

(2) $\mu \leq \nu$ und $\nu \leq \delta \Rightarrow \mu \leq \delta$ (transitiv)

(3) $\mu \leq \mu$ (reflexiv)

Wir wollen nun einmal spezielle Multimengen betrachten, die wir für ein zugehöriges Element $a \in A$ mit $`a \in MULT(A)$ bezeichnen und durch $`a(a)=1$ und $`a(b)=0$ für alle $b \in A$ mit $a \neq b$ definieren wollen. Jede Multimenge aus $MULT_{fin}(A)$ läßt sich dann eindeutig in der Form $\mu = \sum_{a \in Tr(\mu)} d_a \cdot `a$ darstellen, wobei $Tr(\mu)$ die endliche Trägermenge, d.h. die Menge aller $a \in A$ mit $\mu(a) \neq 0$, ist.

Def. 25 (Formale Summenschreibweise für Multimengen)

Sei A eine Menge.

(a) Für jedes $a \in A$ bezeichnen wir mit $`a$ die Multimenge über $A$, die durch $`a(a)=1$ sonst 0 definiert ist.

(b) Die Darstellung einer Multimenge $\mu \in MULT_{fin}(A)$ in der Form $\mu = \sum_{a \in Tr(\mu)} d_a \cdot `a$ mit $d_a \in \mathbb{Z}$ für alle $a \in Tr(\mu)$ bezeichnen wir als *formale Summenschreibweise* der Multimenge μ.

Offensichtlich läßt sich jedes Element $\mu \in MULT_{fin}(A)$ in so einer formalen Summenschreibweise darstellen. Wir werden diese Darstellung im folgenden häufig verwenden.

In den nachfolgenden Kapiteln benötigen wir Multimengen über den Trägermengen eines algebraischen Systems. Elemente solcher Trägermengen werden dabei zunächst nur rein syntaktisch als geschlossene Terme einer zugehörigen Signatur dargestellt und als Elemente der Trägermengen eines zugehörigen algebraischen Systems interpretiert. Diese Trennung von Syntax und Semantik wollen wir nun auf Multimengen über Trägermengen eines algebraischen Systems übertragen.

Hierzu führen wir den Begriff einer *formalen Summe* ein, der an die formale Summenschreibweise angelehnt ist.

Def. 26 (Formale Summen)

Sei $SIG=(S,\Omega,\Pi,typ_\Omega,typ_\Pi)$ eine Signatur, $V_S=(V_S(s))_{s \in S}$ eine Familie von Variablen zu den Sorten $s \in S$ und $TERM(SIG,V_S)$ die Menge der Terme über der Signatur SIG mit Variablen aus der Familie V_S.

(a) Die *Menge der formalen Summen über* $TERM(SIG,V_S)$ ist die kleinste Sprache, für die gilt:

(1) Für alle $t \in TERM(SIG,V_S)$ ist die Zeichenfolge $`t$ in $FSM(SIG,V_S)$. Dabei bindet der Operator $`$ stärker, als alle Operatoren und Prädikate innerhalb des Terms t.

(2) Für beliebige $\omega_1, \omega_2 \in FSM(SIG,V_S)$ ist die Zeichenfolge $\omega_1+\omega_2$ in $FSM(SIG,V_S)$.

(b) Wir wollen noch einige abkürzende Schreibweisen einführen:

(1) $`t+....+`t$, wobei $`t$ n-mal (mit $n \in \mathbb{N}$) wiederholt wird, schreiben wir abkürzend als $n`t$.

(2) Für $\omega_1+\omega_2+.....+\omega_n$ schreiben wir abkürzend $\sum_{i=1}^{n} \omega_i$.

(c) Den *Typ einer formalen Summe* wollen wir rekursiv wie folgt definieren:

(1) Für $`t \in FSM(SIG,V_S)$ sei $typ(`t)=\{ typ(t) \} \subseteq S$.

(2) Für $\omega = \sum_{i=1}^{n} \omega_i \in FSM(SIG,V_S)$ sei $typ(\omega)= \bigcup_{i=1}^{n} typ(\omega_i) \subseteq S$.

(3) Sei $\omega = \sum_{i=1}^{n} d_i `t_i \in FSM(SIG,V_S)$. Dann ist die *Menge der Variablen von* μ definiert durch

$var(\mu)= \bigcup_{i=1}^{n} var(t_i)$.

(4) Sei $\omega = \sum_{i=1}^{n} d_i `t_i \in FSM(SIG,V_S)$. Dann heißt $card(\omega)= \sum_{i=1}^{n} d_i$ die *Kardinalität von* μ. Für jede Sorte $s \in S$ heißt $card_s(\omega)= \sum_{typ(t_i)=s} d_i$ die *Kardinalität von* μ *bzgl. der Sorte s.*

Die Syntax von formalen Summen ist damit analog zur formalen Summenschreibweise von Multimengen. Wie interpretieren wir nun aber formale Summen?

Nun, wenn wir ein algebraisches System $A=(A_S,A_\Omega,A_\Pi)$ als Modell von *SIG* haben, so können wir einen geschlossenen Term t mit $typ(t)=s$ als ein Element der Trägermenge $A_S(s)$ des algebraischen Systems A interpretieren. Ist dieses Element a_t, d.h. $t_R(v)=a_t$ für alle $v \in VAL(V_S,A_S)$, so liegt es nahe, daß wir $`t$ als die Multimenge $`a$ interpretieren. Eine formale Summe $\omega = \sum_{i=1}^{n} d_i `t_i \in FSM(SIG,V_S)$, wobei alle $t_i$ für $i=1,..,n$ geschlossen sind, können wir dann analog als Multisumme $\sum_{i=1}^{n} d_i `a_{t_i}$ interpretieren.

Im allgemeinen enthalten aber Terme innerhalb von formalen Summen Variablen, so daß sie nicht geschlossen sind. Was ist nun in diesem Fall? Zunächst bemerken wir hierzu, daß im Fall eines geschlossenen Terms der Wert t_R nicht von der Valuation abhängt und wir diesen daher als eine konstante Funktion $t_R:VAL(V_S,A_S) \rightarrow \bigcup_{s \in S} A_S(s)$ auffassen können. Wir können daher, wie dies in der Spezifikationstheorie üblich ist, die konstante Abbildung t_R mit der Konstante a_t identifizieren und unsere formale Summe über geschlossenen Termen als $\sum_{i=1}^{n} d_i `(t_i)_R$ schreiben. Den Ausdruck $`t_R$ können wir dann (auch wenn t nicht geschlossen ist) als eine Abbildung von der Menge der Valuationen $VAL(V_S,A_S)$ in die Menge der Multimengen über der Vereinigung der Trägermengen von $A \bigcup_{s \in S} A_S(s)$ interpretieren. Um dies zu definieren, führen wir zunächst die folgende abkürzende Schreibweise ein.

Bezeichnung:

Sei $A=(A_S,A_\Omega,A_\Pi)$ ein algebraisches System über einer Signatur *SIG*.

Wenn es zu keinen Mißverständnissen führt, schreiben wir im folgenden einfach A für die Vereinigung der Trägermengen von A. Insbesondere steht $MULT^+_{fin}(A)$ für $MULT^+_{fin}(\bigcup_{s \in S} A_S(s))$.

Wir definieren nun:

Def. 27 (Realisation von formalen Summen)

Sei $SIG=(S,\Omega,\Pi,typ_\Omega,typ_\Pi)$ eine Signatur, $V_S=(V_S(s))_{s \in S}$ eine Familie von Variablen zu den Sorten $s \in S$, $TERM(SIG,V_S)$ die Menge der Terme über der Signatur *SIG* mit Variablen aus der Familie V_S und $FSM(SIG,V_S)$ die Menge der formalen Summen von Termen aus $TERM(SIG,V_S)$. Weiter sei $A=(A_S,A_\Omega,A_\Pi)$ ein algebraisches System zur Signatur *SIG*. Die *Realisation der formalen Summen* aus $FSM(SIG,V_S)$ im algebraischen System A ist eine Abbildung $\mathbf{R}_{FSM}:FSM(SIG,V_S)->[VAL(V_S,A_S)->MULT^+_{fin}(A)]$, die wie folgt definiert ist: Für jede formale Summe $\omega \in FSM(SIG,V_S)$ ist $\mathbf{R}_{FSM}(\omega)=\omega_R:VAL(V_S,A_S) \rightarrow MULT^+_{fin}(A)$ durch die folgenden Regeln rekursiv bestimmt:

(1) Für $\omega = `t$ ist $\omega_R(v) = `t_R(v)$ für alle $v \in VAL(V_S,A_S)$.

(2) Für $\omega = \omega_1 + \omega_2$ ist $\omega_R(v) = \omega_{1R}(v) + \omega_{2R}(v)$ für alle $v \in VAL(V_S,A_S)$. Dabei steht das +-Zeichen auf der rechten Seite für die Addition von Multimengen.

Eine formale Summe $\omega = \sum_{i=1}^{n} d_i `t_i \in FSM(SIG,V_S)$ interpretieren wir also als Abbildung

$$\omega_R = \sum_{i=1}^{n} d_i `t_i\ _R : VAL(V_S,A_S) \rightarrow MULT^+_{fin}(A).$$

Falls alle Terme innerhalb der formalen Summe geschlossen sind, ist die Abb. ω_R konstant, und wir können sie in diesem Falle als Element von $MULT^+_{fin}(A)$, d.h. als Multimenge, auffassen.

Die formalen Summen von Termen sind "Ausdrücke" (im wesentlichen Terme einer Signatur) mit "Argumenten" einer Operation `, die selbst Terme über einer Signatur sind. Die Semantik dieser "Ausdrücke" haben wir dabei im wesentlichen unabhängig von der speziellen Semantik der "Argumentterme" beschrieben. Um eine solche Konstruktion wirklich formal in einem Logikkalkül durchführen zu können, müßten wir Variablen haben, deren Werte Terme sind, und einen Logikkalkül benützen, der auf solchen "Höheren Variablen" basiert. Dies wäre aber eine Sprache 2. Ordnung.

Wir wollen dies hier nicht näher präzisieren, sondern nur auf dieses unangenehme Merkmal der Theorie der Multimengen hinweisen. Wir begnügen uns hier mit der intuitiv verständlichen Beschreibung der Semantik von formalen Summen, wie sie durch Def. 27 gegeben ist.

Formale Summen bilden die Grundlage für die Spezifikation der Kantenbeschriftungen und Markierungen in unserem Netz-Modell. In SNL sind formale Summen über die folgende kontextfreie Grammatik definiert:

formal_sum:	*base_sum* { + *base_sum* }$^*_0$
base_sum:	[**num**] ` *term*

Als Basis dienen formale Summen der Gestalt `*term*, wobei *term* ein beliebiger Term über einer gegebenen Signatur und den Variablen einer Spezifikation ist. Diese Grundelemente für formale Summen können dann optional mit einem Faktor versehen werden, der eine natürliche Zahl bezeichnet. Einzelne Faktoren können schließlich durch das Summensymbol verknüpft werden. Der Operator `, der aus einem Term eine formale Summe erzeugt, hat dabei eine höhere Priorität, als jeder Operator innerhalb von Termen.

Beispiel 38

Wenn wir als algebraische Spezifikation `FIFO` für ein Beispiel wählen und `w` als eine Variable vom Typ `fifo` auffassen, so sind `1`rest(w)` und `1`get(w)+2`get(rest(w))` zwei formale Summen vom Typ {`fifo`} bzw. {`data`}.

4.2 SNL-Netz-Spezifikationen

In diesem Unterkapitel wollen wir das grundlegende Petri-Netz-Modell beschreiben. Es basiert (wie alle Petri-Netzformen) auf dem Begriff des Netzes (Eine gute Einführung findet man in [Reisig86]).

Def. 28 (Netz)

Ein Netz ist ein Tupel $N=(P,T,F)$, wobei P und T Mengen von *Stellen* und *Transitionen* mit der folgenden Eigenschaft sind:

(i) $P \cap T = \emptyset$

(ii) $P \cup T \neq \emptyset$

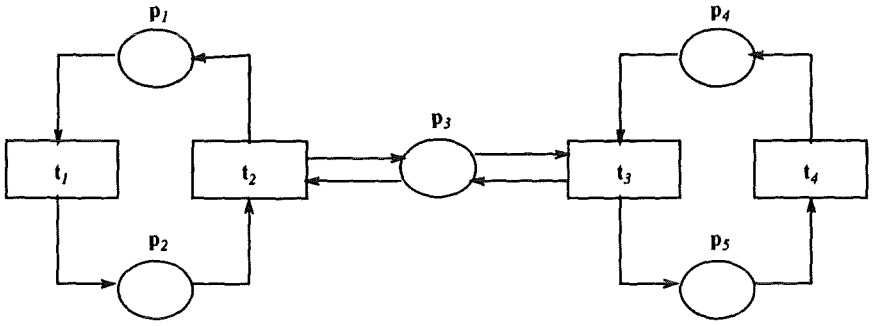
(iii) $F \subseteq P \times T \cup T \times P$

(iv) $dom(F) \cup cod(F) = P \cup T$

Man veranschaulicht sich ein Netz, indem man die Stellen als Kreise, die Transitionen als Rechtecke und die Elemente der Flußrelation $F \subseteq P \times T \cup T \times P$ als gerichtete Kanten darstellt. Betrachten wir hierzu ein Beispiel.

Beispiel 39

Sei $P=\{p_1,p_2,p_3,p_4,p_5\}$ die Menge der Stellen, $T=\{t_1,t_2,t_3,t_4\}$ die Menge der Transitionen und $F=\{(t_1,p_2),(p_2,t_2),(t_2,p_1),(p_1,t_1),(t_2,p_3),(p_3,t_2),(t_3,p_3),(p_3,t_3),(t_3,p_5),(p_5,t_4),(t_4,p_4),(p_4,t_3)\}$ die Flußrelation. Das folgende Bild zeigt dann das Netz in der Veranschaulichung:



Def. 29 (Umgebungsstellen und Umgebungskanten)

Sei $N=(P,T,F)$ ein Netz und $X=P\cup T$.

(a) Für jedes Element $x\in P\cup T$ definieren wir

$$\bullet x = \{y\in P\cup T \mid yFx\},\quad x\bullet = \{y\in P\cup T \mid xFy\}$$

Wenn $x\in T$ ist, nennen wir Elemente aus $\bullet x$ bzw. $x\bullet$ *Vor-* bzw. *Nachstellen der Transition x.*

(b) Für jedes $x\in X$ definieren wir die *Menge der Umgebungsstellen von x* durch

$$penv(x) = \begin{cases} \{p\} & falls\ x=p\in P \\ \bullet t\bullet & falls\ x=t\in T \end{cases}$$

(c) Für jedes $x\in X$ heißt $x^{>}=\{f\in F \mid \text{es gibt ein } y\in X \text{ mit } f=(x,y)\}$ die *Menge der Ausgabekanten* und $^{<}x=\{f\in F \mid \text{es gibt ein } y\in X \text{ mit } f=(y,x)\}$ die *Menge der Eingabekanten von x.* $^{<}x^{>}=^{<}x\cup x^{>}$ heißt die *Menge der Umgebungskanten von x.*

Im folgenden werden nur solche Netze betrachtet, die die folgende Bedingung erfüllen:

- Die Mengen P und T sind endlich.

In der Praxis wünscht man sich Netze, die eine problemnahe Modellierung erlauben. Da der Anwender es mit konkreten Objekten und Abläufen auf diesen Objekten zu tun hat, wünscht er sich eine Netzform, in der "individuelle Objekte" aus seiner Problemwelt und die zugehörigen Aktionen auf den Objekten zur Modellierung verwendet werden können. Dies führte zu einer Reihe von Petri-Netzformen mit individuellen Marken (Objekten) und Beschriftungen, die Operationen und Bedingungen bzgl. der Objekte beinhalten. Beispiele für solche Netzformen findet man u.a. in [Battiston86, Berthomieu86, Hummert87, Reisig91, Schmidt89] und für die bekanntesten dieser Höheren Petri-Netzformen, den Prädikat/Transitionsnetzen und den Colored Petrinetzen, in [Genrich79, Genrich86] und [Jensen81a, Jensen86]. Eine Beschreibung der Eigenschaften dieser verschiedenen Netzformen wollen wir auf ein späteres Kapitel verschieben, in dem wir dann unser Netzmodell mit einigen der in den oben erwähnten Artikeln vorgestellten Netzformen vergleichen wollen. An dieser Stelle wollen wir vielmehr zunächst einmal mit der Beschreibung unseres Netzmodells beginnen.

In diesem werden Objekte und die auf ihnen durchführbaren Operationen durch abstrakte Datentypen beschrieben. Hierbei ist es sinnvoll, die Spezifikation eines abstrakten Datentyps von einer Realisierung des Datentyps zu unterscheiden. Entprechend wollen wir in unserem Netz-Modell zwischen der Spezifikation eines Netzes und möglicher ausführbarer Modelle dieser Spezifikation (Realisierungen der Netz-Spezifikation) unterscheiden. Wir wollen daher zunächst einmal den Begriff einer SNL-Netz-Spezifikation einführen. Diese versieht ein Netz mit einer Beschriftung, die als Bauelemente Elemente der Spezifikation eines abstrakten Datentyps verwendet.

Def. 30 (SNL-Netz-Spezifikation)

Es sei $N = (P,T,F)$ ein Netz, $SIG=(S,\Omega,\Pi,typ_\Omega,typ_\Pi)$ eine Signatur, $V_S=(V_S(s))_{s\in S}$ eine Familie von Variablen, $L = (ALPH, TERM(SIG,V_S), FORM(LSIG,SIG,V_S))$ eine Sprache der ersten Ordnung über SIG

mit Variablen aus V_S und $SPEC(SIG,E)$ eine algebraische Spezifikation über SIG mit Formeln aus L (d.h. $E \subseteq FORM(LSIG,SIG,V_S)$).

(a) Eine Beschriftung des Netzes N in der Sprache L ist ein 3-Tupel $A_N=(A_P,A_T,A_F)$, wobei gilt:

(1) $A_P=(cap, vars, eqn)$ ist ein 3-Tupel von Abbildungen $cap:P\to[S\to\mathbb{N}_0]$, $vars:P\to\wp(V_S)$ und $eqn:P\to FORM(LSIG,SIG,V_S)$. *cap* ordnet jeder Stelle $p\in P$ eine *Kapazitätsfunktion* $cap(p)=cap_p:S\to\mathbb{N}_0$ zu, wobei $cap(p)(s)=cap_p(s)=cap(p,s)$ die *Kapazität der Stelle p bzgl. der Sorte* $s\in S$ ist. *vars* ordnet jeder Stelle $p\in P$ eine *Familie von Variablen* $vars(p)=(vars_s(p))_{s\in S}\subseteq \wp(V_S)=(\wp(V_S(s))_{s\in S}$ zu. *eqn* ordnet schließlich jeder Stelle eine *Formel* $eqn(p)$ zu.

(2) $A_T=(fvars, bvars, eqn)$ ist ein 3-Tupel von Abbildungen $fvars:T\to\wp(V_S)$, $bvars:T\to\wp(V_S)$ und $eqn:T\to FORM(LSIG,SIG,V_S)$. *fvars* ordnet jeder Transition $t\in T$ eine *Familie von freien Variablen* $fvars(t)=(fvars_s(t))_{s\in S}\subseteq \wp(V_S)$ und *bvars* ordnet jeder Transition $t\in T$ eine *Familie von gebundenen Variablen* $bvars(t)=(bvars_s(t))_{s\in S}\subseteq \wp(V_S)$ zu. *eqn* ordnet jeder Transition eine *Transitionsformel* $eqn(t)\in FORM(LSIG,SIG,V_S)$ zu.

(3) $A_F:F\to FSM(SIG,V_S)$ ordnet jeder Kante des Netzes eine formale Summe zu.

(4) $typ(vars(p)) \subseteq \{ s\in S \mid cap_p(s)>0 \}$ für alle $p\in P$.

(5) $eqn(p)$ ist für alle $p\in P$ eine geschlossene Formel mit $vars(eqn(p)) \subseteq vars(p)$. Dabei sei $vars(eqn(p))$ die Menge der Variablen in der Formel $eqn(p)$.

(6) $\forall p\in P$ und $f\in {}^{\circ}p^{\circ}$ gilt: $typ(A_F(f)) \subseteq \{ s\in S \mid cap_p(s)>0 \}$.

(7) $\forall t\in T$ und $f\in {}^{\circ}t^{\circ}$ gilt: $fvars(A_F(f))\subseteq fvars(t)$.

(8) $\forall t\in T$ gilt: $fvars(t)\cap bvars(t)=\emptyset$.

(9) $\forall t\in T$ gilt: $vars(t):=fvars(t)\cup bvars(t)\subseteq \bigcup_{p\in penv(t)} vars(p)$.

(10) $\forall t\in T$ gilt: $fvars(eqn(t))\subseteq fvars(t)$ und $bvars(eqn(t))\subseteq bvars(t)$. Dabei sei $fvars(eqn(t))$ die Menge der freien Variablen und $bvars(eqn(t))$ die Menge der gebundenen Variablen in der Formel $eqn(t)$.

(b) Sei A_N eine Beschriftung wie in (a) definiert. Eine Familie von formalen Summen $M_0=(m_{0p})_{p\in P}$ heißt eine *(formale) Markierung eines beschrifteten Netzes* (N,A_N), falls gilt:

(1) Für alle $p\in P$ ist $m_{0p}\in FSM(SIG,V_S)$.

(2) $vars(m_{0p}) = \emptyset$ für alle $p\in P$.

(3) $0\le card_s(m_{0p}) \le cap_p(s)$ für alle $s\in S$ und $p\in P$.

(c) Seien A_N eine Beschriftung und M_0 eine Markierung wie in (a) und (b) definiert. Das Tripel $NSPEC=(N,A_N,M_0)$ heißt dann eine *SNL-Netz-Spezifikation mit Anfangsmarkierung* M_0 *über der algebraischen Spezifikation SPEC.*

Wir wollen im Kontext der obigen Definition noch einige Bezeichnungen einführen:

Def. 31 (Der Begriff der Umgebung)

Es seien die Voraussetzungen der vorigen Definition gegeben, und es sei $X=P\cup T$.

(a) Für alle $p\in P$ definieren wir die *Menge der Sorten von p* durch $sorts(p) = \{ s\in S \mid cap_p(s)>0 \}$.

(b) Für jedes $x\in X$ sei $senv(x)= \bigcup_{p\in penv(x)} sorts(p)$ die *Menge der Sorten in der Umgebung von x.*

(c) Für jedes $x\in X$ sei $venv(x)= \bigcup_{p\in penv(x)} vars(p)$ die *Familie der Variablen in der Umgebung von x.*

(d) Für jedes $p\in P$ sei *die Familie der freien Variablen von p* definiert durch $fvars(p):=\emptyset$, da Formeln auf Stellen per Definition geschlossen sind.

(e) Für alle $t\in T$ sei die Familie der Variablen von t definiert durch $vars(t)=(vars_s(t))_{s\in S}=(fvars_s(t)\cup bvars_s(t))_{s\in S}$.

(f) Für jedes $x\in X$ nennen wir die durch $penv(x)$, ${}^{\circ}x^{\circ}$, $svenv(x)$ und $venv(x)$ gegebenen Informationen die *Umgebung von x in der SNL-Netz-Spezifikation NSPEC.*

Wir wollen Def. 30 nun ein wenig näher erläutern. Der Punkt (a) der Definition legt die Beschriftung eines Netzes fest, die aus einer Reihe von Abbildungen besteht. Jeder Stelle $p\in P$ ist zunächst über die Abbildung *cap* eine Menge von Sorten $sorts(p)$ (siehe Def. 31 (a)) zugeordnet. Wir können uns eine Stelle p als Speicher vorstellen, der Multimengen von Objekten enthält, wobei der Typ jedes Objektes einer der Sorten aus $sorts(p)$ entsprechen

muß. Die maximale Anzahl der Objekte einer Sorte s auf der Stelle p wird durch die Kapazität $cap_p(s)$ bestimmt. Eine Kapazität 0 bedeutet, daß Objekte dieser Sorte nicht auf der Stelle vorkommen dürfen.

Weiter ist jeder Stelle p eine Menge von Variablen *vars*(p) zugeordnet, deren Typen in *sorts*(p) enthalten sein müssen (Bedingung 4). Diese Variablen können dann in Formeln bzgl. der Stelle p bzw. innerhalb der Formeln von Transitionen, zu deren Umgebung die Stelle gehört, verwendet werden (siehe etwas weiter unten).

In der Praxis benötigt man in SNL-Netzen in der Regel abstrakte Datentypen, deren Individuenbereiche in einer Realisierung ebenfalls endlich sind. Dies garantiert man durch Formeln innerhalb der Spezifikation, die die Individuenbereiche beschränken. In einem Beispiel könnte die Länge der erlaubten Warteschlangen z.B. durch eine Formel *length*(w)<*maxf* beschränkt sein. Dabei erscheint es natürlich, daß man so eine Formel in einem Netz einer Stelle analog der Kapazitätsfunktion zuordnet. Wir erlauben es daher, daß man Stellen p optional eine geschlossene Formel *eqn*(p) zuordnet, die nur Variablen enthalten darf, die dieser Stelle zugeordnet sind (Bedingung 5). In einer Realisierung des Netzes sind dann nur Markierungen erlaubt, die die Formeln auf den Stellen erfüllen.

Jeder Transition $t \in T$ ist eine Formel *eqn*(t) zugeordnet. Diese Formel stellt eine Bedingung dar, die erfüllt werden muß, damit eine Transition schalten kann. Die Menge der freien Variablen einer Transitionsformel *eqn*(t) muß dabei in der Familie der freien Variablen der Transition *fvars*(t) und die Menge der gebundenen Variablen der Transitionsformel muß in der Familie der gebundenen Variablen *bvars*(t) enthalten sein. Dies ist Bedingung 10. Die Familien der freien und gebundenen Variablen müssen weiter disjunkt sein, und jede Variable (frei und gebunden) einer Transition muß mindestens einer Umgebungsstelle der Transition zugeordnet sein (Bedingungen 8 und 9).

Jede Kante ist mit einer formalen Summe beschriftet, deren Variablen in der Menge aller freien Variablen der zugehörigen Transition enthalten sein müssen (Bedingung 7). Diese formalen Summen definieren den Fluß der Objekte durch das Netz, d.h. nach Substitution der Variablen liefern sie in der Realisation der Netzspezifikation Multimengen, die beschreiben, welche Objekte von einer Stelle abzuziehen bzw. welche Objekte auf eine Stelle zu legen sind. Da beim Schalten einer Transition nur Objekte auf eine Stelle gelegt bzw. von einer Stelle abgezogen werden, die zu Sorten gehören, für die die Stelle auch einen Speicher darstellt, muß Bedingung 6 gelten.

Bedingung (9) stellt eine Anforderung an das Schalten eines Netzes dar, die man als Lokalitätsprinzip bezeichnen kann. Dieses Prinzip verlangt, daß das Schalten einer Transition (also das "ob" und "wie" eine Transition schaltet) nur vom Zustand des Systems in der Umgebung der Transition, nicht aber von anderen Faktoren abhängen darf. Wir verlangen daher in (9), daß die Familien der Variablen einer Transition t in der Menge der Umgebungsvariablen *venv*(t) von t enthalten sein muß. Wir werden die Variablen der Transitionen dann über den Inhalt der Umgebungsstellen der betreffenden Transition auswerten. Die Werte der Transitionsbeschriftungen und Kantenbeschriftungen hängen damit nur von dem Inhalt der Umgebungsstellen, also vom aktuellen Zustand (der aktuellen Markierung) in der Umgebung der betreffenden Transition, ab.

(10) und (7) geben an, daß das Schalten einer Transition t nur durch eine Auswertung (Zuweisung von Werten) der freien Variablen einer Transition t (*fvars*(t)) bestimmbar ist. Bezüglich so einer Auswertung ist die Formel *eqn*(t) entweder *true* oder *false* und bestimmt damit, ob die Bedingung für das Schalten der Transition bezüglich dieser Auswertung erfüllt ist oder nicht. Einsetzen der Auswertung in die Kantenbeschriftung ergibt die Multimenge von Objekten, die abzuziehen bzw. auf die Nachfolgestellen zu legen sind.

Als Beispiel für eine SNL-Netz-Spezifikation versehen wir unser Netz aus Beispiel 39 mit einer geeigneten Beschriftung.

Beispiel 40

Sei $N = (P,T,F)$ das Netz aus Beispiel 39. Wir setzen voraus, daß wir die folgende leicht modifizierte Spezifikation des `FIFO`-Funktors gegeben haben.

```
specification FIFO_PARAM = spec
        sort  elem;
end

functor FIFO(FP:FIFO_PARAM) :
sig
        sort  elem;
        union fifo = empty | put of fifo * elem;
        ptl   get : fifo -> elem;
              rest : fifo -> fifo;
```

```
        opn   length : fifo -> int;
end
= spec
        sort  elem = FP.elem;
        var   w : fifo; n : elem;
        eqn   def get(w) <=> w <> empty;
              if (w <> empty) then get(put(w,n))=get(w)
                              else get(put(w,n))=n;
              def rest(w) <=> w <> empty;
              if (w <> empty) then rest(put(w,n))=put(rest(w),n)
                              else rest(put(w,n))=w;
              length(put(w,n))=length(w)+1;
              length(empty)=0;
end
```

Wir definieren weiter ein System von zwei Erzeugerprozessen, die Daten erzeugen, und einem Verbraucherprozeß, der diese Daten konsumiert, über die folgende Spezifikation.

```
specification PROCESS = spec
      union     process = a1 | a2 | b;
      union     data = c1 | c2;
      ptl       mdata : (pr:process | pr <> b) -> data;
      opn       proc : data -> process;
      var       pr : process;
                n : data;
      eqn       mdata(proc(n))=n;
                if (pr <> b) proc(mdata(pr))=pr;
end
```

Auf Basis dieser Spezifikation und dem Funktor `FIFO` wollen wir dann die Spezifikation für unser Beispielnetz wie folgt definieren:

```
specification ERZEUGER-VERBRAUCHER = spec
      include   PROCESS;
                FIFO(spec sort elem=PROCESS.data; end);
      const     maxf : int;
end
```

Eine mögliche Beschriftung des Netzes ist dann:

(1) Die Beschriftung der Transitionen sei stets die Formel `true`.

(2) Die Beschriftung der Stellen p_1 und p_4 sei gegeben durch die Sorte `process`, die Variable `pr` der Sorte `process` und der Kapazität 2 für p_1 und 1 für p_4.

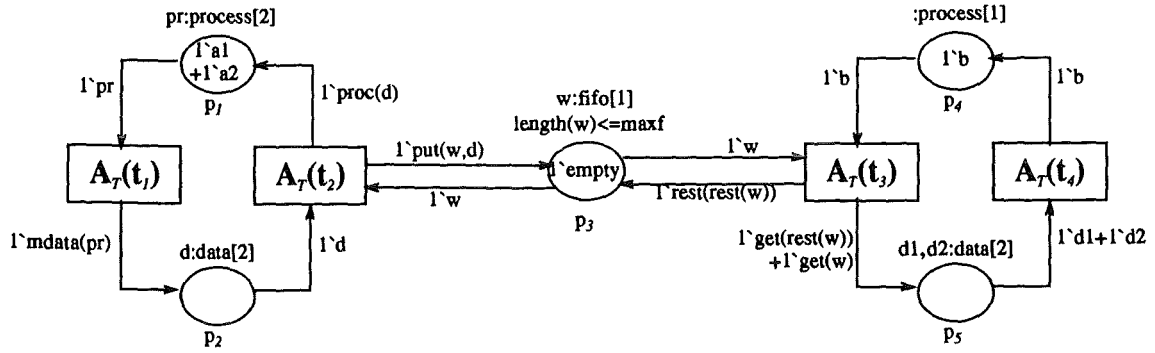
(3) Die Beschriftung der Stelle p_2 sei gegeben durch die Sorte `data`, die Variable `d` der Sorte `data` und eine Kapazität 2.

(4) Die Beschriftung der Stelle p_5 sei gegeben durch die Sorte `data`, Variablen `d1` und `d2` der Sorte `data` und eine Kapazität 2.

(5) Die Beschriftung der Stelle p_3 sei gegeben durch die Sorte `fifo`, die Variable `w` der Sorte `fifo`, die Kapazität 1 der Sorte `fifo` und eine Formel `length(w)<=maxf`.

(6) Die Beschriftung der Kanten sei gegeben durch $A_F((t_1,p_2))$=`1`mdata(pr)`, $A_F((p_2,t_2))$=`1`d`, $A_F((t_2,p_1))$=`1`proc(d)`, $A_F((p_1,t_1))$=`1`pr`, $A_F((t_2,p_3))$=`1`put(w,d)`, $A_F((p_3,t_2))$=`1`w`, $A_F((t_4,p_4))$=`1`b`, $A_F((p_3,t_3))$=`1`w`, $A_F((t_3,p_3))$=`1`rest(rest(w))`, $A_F((p_5,t_4))$=`1`d1+1`d2`, $A_F((t_3,p_5))$=`1`get(w)+1`get(rest(w))`, $A_F((p_4,t_3))$=`1`b`.

Eine Anfangsmarkierung ist durch die Abbildung $M_0(p_1)$=`1`a1+1`a2`, $M_0(p_4)$=`1`b`, $M_0(p_3)$=`1`empty` gegeben, wenn wir alle weiteren Stellen mit der formalen Summe 0, die wir nie explizit angeben werden, versehen denken. Das folgende Bild zeigt das Netz mit der Anfangsmarkierung.



Wir haben dabei die Beschriftungen des Netzes schon teilweise in SNL angegeben, wie wir im folgenden genauer erläutern wollen.

Wir wollen nun die mathematische Definition einer SNL-Netz-Spezifikation in die Sprache SNL transformieren.

net_decl:	**petrinet** *net_id* = *net*
net_id:	**id**
net:	**net** *net_body* **end**
net_body:	{ *net_body_decls* }$_0^*$
net_body_decls:	*spec_sig_decls*
	spec_eqn_decls
	net_decls

Das Aussehen der Spezifikationsdeklarationsteile *spec_sig_decls* und *spec_eqn_decls* wurde dabei schon im Kapitel über den Spezifikationsteil von SNL festgelegt. Wir müssen hier noch die Netzdeklarationen *net_decls* näher definieren.

Zur Angabe der Netzteile liegt eigentlich eine Vierteilung in Stellendeklarationen, Transitionsdeklarationen, Kantendeklarationen und eine die Anfangsmarkierung des Netzes beschreibende Deklaration nahe. Auf Grund der starken Umgebungsabhängigkeit der Transitionen ist es jedoch günstiger, die Umgebungskanten einer Transition als Teil der Transitionsdefinition zu definieren. Weiter fassen wir die Anfangsmarkierung einer Stelle als Teil einer Stellendeklaration auf. Wir haben daher nur eine Zweiteilung der Deklarationen:

net_part:	*place_decls*
	trans_decls

Ein Stellendeklarationsteil (*place_decls*) beginnt mit dem Schlüsselwort **placedef**. Anschließend folgt eine Folge von Stellendeklarationen, die durch Semikolon getrennt sind.

place_decls:	**placedef** *place_decl* { ; *place_decl* ; }$_0^*$ [;]

Eine Stellendeklaration bindet einen Stellennamen (*place_name*) an eine Stellendefinition (*place*).

place_decl:	*place_name* = *place*
place:	*place_name*
	place *place_body* **end**

Eine Stellendefinition wird entweder durch den Namen einer bereits deklarierten Stelle oder durch eine explizite Definition der Eigenschaften der Stelle innerhalb eines Konstruktes **place ... end** beschrieben. Im ersten Fall erbt die deklarierte Stelle die Eigenschaften der Stelle, deren Name rechts vom Gleichheitszeichen steht, wobei diese bereits deklariert sein muß. Im anderen Fall werden die Eigenschaften einer Stelle innerhalb des Stellenrumpfs (*place_body*) in verschiedenen Deklarationsteilen beschrieben, die die Sorten und Variablen beschreiben, die einer Stelle zugeordnet sind, die Kapazität der Stelle bzgl. jeder Sorte festlegen, eventuell eine Formel angeben, die von den individuellen Objekten auf der Stelle erfüllt sein muß, und die Anfangsmarkierung einer Stelle beschreiben. *place_body* hat daher die folgende Gestalt:

place_body:	{ *place_body_decls* }$_0^*$
place_body_decls:	*sort_var_cap_decls*
	spec_formula_decls
	marking_decl

sort_var_cap_decls:	**var** *sort_var_cap_decl* { ; *sort_var_cap_decl* }$_0^*$ [;]
sort_var_cap_decl:	[*var_list*] : *sort_name* [[*capacity*]]
var_list:	*var_id* { , *var_id* }$_0^*$
var_id:	**id**
capacity:	**num**
marking_decl:	**marking** *marking* [;]
marking:	*formal_sum*

Die *sort_var_cap_decls* fassen die Deklarationen der Sorten einer Stelle, der zugehörigen Kapazitäten und Variablen zusammen. Einer Liste von Variablen wird innerhalb einer *sort_var_cap_decl* ein Sortenname als Typ zugeordnet, dem optional eine Kapazitätsangabe in rechteckigen Klammern folgt. Fehlt diese Kapazitätsangabe, so ist die Kapazität der Stelle bzgl. der angegebenen Sorte defaultmäßig 1. Man beachte dabei, daß eine Sorte nur in genau einer solchen *sort_var_cap_decl* auftreten darf. Weiter beachte man, daß die Variablenliste auch leer sein darf. In diesem Fall wird einer Stelle eine Sorte zugeordnet, bzgl. der keine Variablen definiert sind. Gleichwohl können auf so einer Stelle Individuen der entsprechenden Sorte liegen, die dann in Transitionen oder Kantenbeschriftungen nicht über Variablen, sondern über Konstanten der jeweiligen Sorte referenziert werden können.

Innerhalb von *spec_formula_decls* können einer Stelle weiter eine oder mehrere Formeln zugeordnet werden, wobei die Syntax zur Angabe der Formeln *spec_formula_decls* der Syntax der Formeln im algebraischen Spezifikationsteil entspricht. Die Variablen innerhalb der Formeln müssen allerdings in den *sort_var_cap_decl* definiert sein. Weiter ist eine zu einer Stelle gehörige Formel stets geschlossen, d.h. Variablen, die nicht explizit quantorisiert sind, werden stets als durch einen Allquantor quantorisierte Variablen aufgefaßt.

Optional folgt weiter die Angabe einer eventuell vorhandenen Anfangsmarkierung der Stelle (*marking_decl*). Diese muß eine formale Summe von geschlossenen Termen sein, deren Typ zu den Sorten passen muß, die der Stelle zugeordnet sind. Weiter darf die Anzahl der Elemente zu einer Sorte innerhalb der formalen Summe nicht die Kapazität der Stelle für diese Sorte überschreiten.

Da die formale Summe nur geschlossene Terme enthält, definiert sie in der Realisierung eine Multimenge von Individuen, die der Stelle zugeordnet ist. Es sind dabei nur solche Realisierungen erlaubt, die garantieren, daß von der Platzmarkierung eventuell zu der Stelle spezifizierte Formeln erfüllt werden.

Ein Transitionsdeklarationsteil (*trans_decls*) beginnt mit dem Schlüsselwort **transition**. Anschließend folgt eine Reihe von Transitionsdeklarationen, die durch Semikolon getrennt sind.

trans_decls:	**transition** *trans_decl* { ; *trans_decl* }$_0^*$ [;]

Eine Transitionsdeklaration bindet einen Transitionsnamen (*trans_id*) an eine Transitionsdefinition (*trans*).

trans_decl:	*trans_id* = *trans*
trans:	*trans_id*
	trans *trans_body* **end**

Eine Transitionsdefinition wird entweder durch den Namen einer bereits deklarierten Transition oder durch eine explizite Definition der Eigenschaften der Transition innerhalb eines Konstruktes **trans ... end** beschrieben. Im ersten Fall erbt die deklarierte Transition die Eigenschaften der Transition, deren Name rechts vom Gleichheitszeichen steht, wobei diese bereits deklariert sein muß. Im anderen Falle werden die Eigenschaften einer Transition innerhalb des Transitionsrumpfs (*trans_body*) in verschiedenen Deklarationsteilen beschrieben, die die freien und gebundenen Variablen beschreiben, die einer Transition zugeordnet sind, Formeln angeben, die erfüllt sein müssen, damit die Transition schalten kann, und die Umgebungskanten der Transition beschreiben. *trans_body* hat daher die folgende Gestalt:

trans_body:	{ *trans_body_decls* }$_0^*$
trans_body_decls:	*trans_var_decls*
	spec_formula_decls
	arc_decls
trans_var_decls:	**fvar** *trans_var_decl* { ; *trans_var_decl* }$_0^*$ [;]
	bvar *trans_var_decl* { ; *trans_var_decl* }$_0^*$ [;]
trans_var_decl:	*var_id* : *place_list*
place_list:	*place_id* { , *place_id* }$_0^*$
var_id:	**id**

place_id:	**id**
arc_decls:	**arc** *arc_decl* { ; *arc_decl* }$_0^*$ [;]
arc_decl:	*place_id* -> *formal_sum*
	place_id <- *formal_sum*

In Deklarationen der Variablenumgebung einer Transition (*trans_var_decl*) wird für eine Transition zunächst definiert, welche freien Variablen (hinter dem Schlüsselwort **fvar**) und welche gebundenen Variablen (hinter dem Schlüsselwort **bvar**) eine Transition benützt. Diese Variablen sind dabei bzgl. ihrer Auswertung an Umgebungsstellen der Transition gebunden, so daß ihnen in einer Variablendeklaration (*trans_var_decl*) nicht eine Sorte als Typ, sondern eine Liste von Stellen zugeordnet wird. Die entsprechende Variable muß dann auf jeder dieser Stellen vom gleichen Typ definiert sein. Bei einer Auswertung der Variablen bzgl. einer vorgegebenen Markierung des Netzes kann dann der Wert einer freien Variablen ein beliebiges Element der Menge von Objekten von der zugehörigen Sorte sein, die sich als Vereinigung der Menge von Objekten dieser Sorte ergibt, die auf den der Variablen innerhalb der Variablendeklaration zugeordneten Stellen liegen. Eine gebundene Variable wird entsprechend über diese Mengen quantifiziert. Man beachte dabei, daß wir hier auch erlauben, daß innerhalb der Stellenliste zu einer Variablen eine Stelle vorkommt, zu der für die Transition keine Kante existiert, die die Transition mit der Stelle verbindet. Unser Begriff der Umgebung ist innerhalb von SNL daher etwas allgemeiner, als er in der Theorie beschrieben wurde.

In Kantendeklarationen *arc_decl* gibt man die Kanten mit ihrer Beschriftung an, die die Transition mit ihren Umgebungsstellen verbindet. Eine Kantenangabe besteht zunächst einmal aus der Angabe der zugehörigen Stelle. Dann folgt die Spezifikation der Richtung der Kante. -> bezeichnet eine Kante, die von der angegebenen Stelle zur Transition führt (also eine Eingabekante). <- bezeichnet eine Kante, die von der Transition zur angegebenen Stelle führt (also eine Ausgabekante). Zum Abschluß folgt die Beschriftung der Kante. Diese ist eine formale Summe (*formal_sum*), die nur freie Variablen enthalten darf, die innerhalb einer Variablendeklaration *trans_var_decl* der Transition als freie Variablen deklariert wurden.

In *spec_formula_decls* kann weiter die Angabe von Formeln erfolgen, die erfüllt sein müssen, damit die Transition schalten kann. Die Syntax für die Angabe solcher Formeln haben wir bereits im Kapitel über die Spezifikation von abstrakten Datentypen definiert. Die Formeln dürfen nur freie bzw. gebundene Variablen enthalten, die als freie bzw. gebundene Variablen innerhalb einer *trans_var_decl* der Transition deklariert worden sind. Falls keine Formel angegeben wurde, wird als Default stets die Formel *true* angenommen.

Wir wollen einmal unser Erzeuger-Verbraucher System in leicht abgewandelter Form innerhalb unserer Netz-Sprache in einem Beispiel spezifizieren.

Beispiel 41 (Erzeuger-Verbraucher-System in SNL)

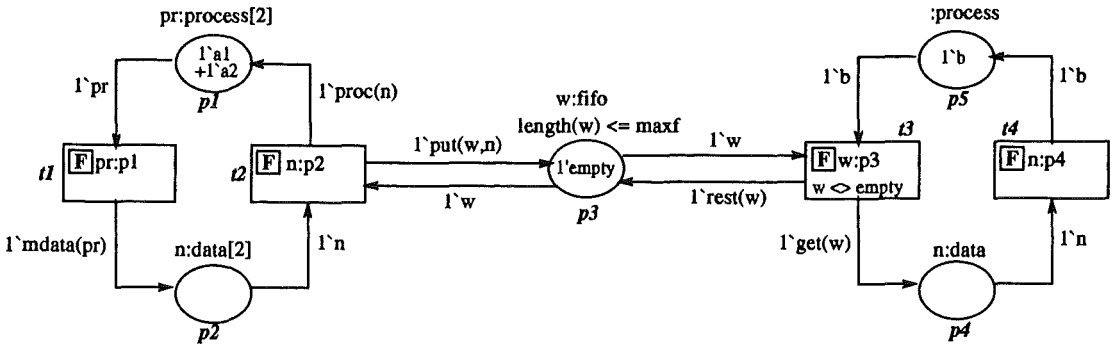
In diesem Beispiel beschränken wir uns auf ein Modell, das jeweils nur ein Datenpaket verbraucht, so daß sich die Beschriftung im Verbraucherteil vereinfacht. Das Modell verwendet die Spezifikation `PROCESS`, sowie den Funktor `FIFO`. Wir haben dann:

```
petrinet ERZEUGER-VERBRAUCHER = net
     include      PROCESS;
                  FIFO(spec sort elem=PROCESS.data; end);
     const        maxf : int;
     placedef     p1=place
                      var pr:process[2];
                      marking 1`a1+1`a2;
                  end;
                  p2=place
                      var n:data[2];
                  end;
                  p3=place
                      var w:fifo;
                      eqn length(w) <= maxf;
                      marking 1`empty;
                  end;
                  p4=place
                      var n:data;
                  end;
                  p5=place
```

```
                    var: process;
                    marking 1`b;
                end;
        transition  t1=trans
                    fvar pr:p1;
                    arc p1 -> 1`pr;
                        p2 <- 1`mdata(pr); end;
                t2=trans
                     fvar n:p2; w:p3;
                     arc p2 -> 1`n;
                         p1 <- 1`proc(n);
                         p3 -> 1`w;
                         p3 <- 1`put(w,n); end;
                t3=trans
                    fvar w:p3;
                    arc p3 -> 1`w;
                        p3 <- 1`rest(w);
                        p4 <- 1`get(w);
                        p5 -> 1`b;
                    eqn w <> empty; end;
                t4=trans
                    fvar n:p4;
                    arc p4 -> 1`n;
                        p5 <- 1`b; end;
end
```

Im Erzeugerzyklus haben wir zwei Prozesse `1`a1` und `1`a2`, die Daten `1`mdata(a1)` und `1`mdata(a2)` erzeugen können. Diese werden über die Transition *t2* in die Warteschlange auf der Stelle *p3* über einen Aufruf von *put* eingetragen. Auf der Verbraucherseite gibt es einen Prozeß `b`, der die Daten über einen *get* Aufruf über die Transition *t3* aus der Warteschlange auf *p3* entfernt und dann in *t4* verarbeitet.

Man beachte dabei die Formel `length(w) <= maxf` auf der Stelle *p3*. Diese blockiert automatisch das Schalten der Transition *t2*, wenn die Warteschlange `maxf` Elemente besitzt. Weiter beachte man die Formel `w <> empty` auf der Transition *t3*. Diese verhindert, daß *t3* schaltet, wenn die Warteschlange leer ist. Die folgende Abbildung verdeutlicht die Netz-Spezifikation ERZEUGER-VERBRAUCHER:



In diesem Bild haben wir die Beschriftung des Netzes in der Form angegeben, die wir in unserer Sprachdefinition definiert haben. Die Angabe der freien Variablen haben wir zusätzlich durch ein Quadrat mit der Inschrift F markiert, um sie eindeutig von einer möglichen Angabe von gebundenen Variablen zu unterscheiden.

4.3 SNL-Systeme (SNL-Modelle)

In diesem Unterkapitel wollen wir nun die Semantik beschreiben, die durch eine SNL-Netz-Spezifikation gegeben ist. Die Idee dabei ist, daß wir die Netzbeschriftung bzgl. algebraischer Systeme, die Modelle der Spezifikationen innerhalb der SNL-Netz-Spezifikation sind, interpretieren. Dies wollen wir im folgenden genauer beschreiben.

Um die semantischen Eigenschaften eines SNL-Systems definieren zu können, müssen wir zunächst die Semantik der Beschriftung der SNL-Netz-Spezifikation definieren. Hierzu ist zunächst eine Realisation der algebraischen Spezifikation in der SNL-Netz-Spezifikation über ein algebraisches System A und eine logische Matrix $\overline{A}$ notwendig. Auf deren Grundlage interpretieren wir dann die Terme, formalen Summen und elementaren Formeln innerhalb der SNL-Netz-Spezifikation über die uns schon bekannten Realisierungen $\mathbf{R}_T$, $\mathbf{R}_{FSM}$ und $\mathbf{R}_{EF}$. Die Formeln im Spezifikationsteil einer SNL-Netz-Spezifikation interpretieren wir über die schon bekannte Realisation $\mathbf{R}_F$.

Im Gegensatz zur Realisation von Formeln mit Quantoren im Spezifikationsteil einer SNL-Netz-Spezifikation, die wir schon im Spezifikationsteil dieses Berichtes definiert haben, wollen wir die Quantoren in Formeln auf den Transitionen und Stellen nicht als All- und Existenzquantor auf den zugehörigen Wertebereichen von A, sondern gemäß unserem Lokalitätsprinzip als All- und Existenzquantor über der Menge der Individuen der jeweiligen Sorte, die bezüglich der aktuellen Markierung des Netzes auf den Umgebungsstellen einer Transition liegen, auffassen. Wir erhalten hiermit eine Interpretation der Formeln, die von der Markierung im Netz abhängt.

Bevor wir diese Interpretation von Netz-Formeln formal definieren, wollen wir zunächst noch einige Hilfsbegriffe einführen.

Def. 32 (Hilfsbegriffe)

Sei $NSPEC=(N,A_N,M_0)$ eine SNL-Netz-Spezifikation und $A=(A_S,A_\Omega,A_\Pi)$ ein Modell der zu $NSPEC$ gehörenden algebraischen Spezifikation $SPEC$. $M=(m_p)_{p\in P}$ sei weiter eine Familie von Multimengen, die jeder Stelle $p\in P$ des Netzes eine Multisumme $m_p\in MULT_{fin}(A)$ über den Trägermengen von A zuordnet, wobei $typ(m_p)\subseteq sorts(p)$ ist. Weiter sei $X=P\cup T$.

(a) Mit $Tr_s(m_p)$, $s\in sorts(p)$, bezeichnen wir die Teilmenge der Elemente von $Tr(m_p)$, die vom Typ s sind.

(b) Für $t\in T$, $v\in vars(t)$ sei $penv_v(t):=\{\, p\in\cdot t \mid v\in vars(p)\,\}$. Für $p\in P$, $v\in vars(p)$ sei $penv_v(p)=\{p\}$.

(c) Sei $x\in X$ und $v\in vars(x)$. Mit $obj_{v,x}(M)$ bezeichnen wir die Menge aller Objekte der Sorte $typ(v)$, die bzgl. der *Familie M in der Umgebung von x* liegen, d.h. $obj_{v,x}(M)=\bigcup_{p\in penv_v(x)} Tr_s(m_p)$.

(d) Die Menge der Formeln in der Netz-Beschriftung einer SNL-Netz-Spezifikation $NSPEC$ bezeichnen wir mit $NEQN(NSPEC)$. Die Menge der Formeln im Spezifikationsteil bezeichnen wir mit $EQN(NSPEC)$.

(e) Sei $t\in T$. Die Menge der Auswertungen der freien Variablen von t in A bezeichnen wir mit $VAL_t(NSPEC,A):=\{\, v:=(v_s:fvars_s(t)\to A_S(s))_{s\in S}\,\}$. Für $p\in P$ definieren wir $VAL_p(NSPEC,A):=\emptyset$.

(f) Sei $t\in T$ und $v\in VAL_t(NSPEC,A)$. v heißt *Auswertung der freien Variablen von t unter M in A*, falls für alle $y\in fvars(t)$ gilt: $v(y)\in obj_{y,t}(M)$. Wir bezeichnen die Menge aller Auswertungen der freien Variablen von t unter M in A mit $VAL_t(NSPEC,M)$. Für jedes $p\in P$ definieren wir $VAL_p(NSPEC,A):=\emptyset$.

Wir definieren nun die Realisation von Formeln in der Netzbeschriftung einer SNL-Netz-Spezifikation wie folgt:

Def. 33 (Realisation der Formeln einer SNL-Netz-Spezifikation)

Seien SIG, $LSIG$, V_S wie in den anderen Definitionen gegeben. L sei eine formale Sprache 1. Ordnung über SIG und $LSIG$. Sei $NSPEC=(N,A_N,M_0)$ eine SNL-Netz-Spezifikation über SIG und $LSIG$ mit Formeln aus L, $A=(A_S,A_\Omega,A_\Pi)$ ein algebraisches System über der Signatur SIG und $\overline{A}=(A_{\overline{S}},A_{\overline{\Omega}})$ eine logische Matrix zu der Signatur $LSIG$. $M=(m_p)_{p\in P}$ sei eine Familie von Multimengen, die jeder Stelle des Netzes eine Multimenge $m_p\in MULT_{fin}(A)$ über den Trägermengen von A zuordnet, wobei für alle $p\in P$ $typ(m_p)\subseteq sorts(p)$ ist. Weiter sei $\rho_\Pi=(\rho_\pi: A_S^{typ_\Pi(\pi)} \to A_{\overline{S}}(\overline{s}))_{\pi\in\Pi}$ eine Familie von Funktionen, $\mathbf{R}_T:TERM(SIG,V_S)\to[VAL(V_S,A)\to\cup_{s\in S}A_S(s)]$ die durch A eindeutig bestimmte Realisation von Termen und $\mathbf{R}_{EF}:FORM(SIG,V_S)\to[VAL(V_S,A)\to A_{\overline{S}}(\overline{s})]$ die durch A, $\overline{A}$ und ρ_Π eindeutig bestimmte Realisation von atomaren Formeln.

Die durch A, $\overline{A}$ und ρ_Π eindeutig bestimmte *Realisation der Formeln der Netzbeschriftung einer SNL-Netz-Spezifikation NSPEC bzgl. M* ist eine Abbildung $\mathbf{R}_N$, die jeder Formel $\alpha \in NEQN(NSPEC)$ zu $x \in X$ eine Abbildung $\mathbf{R}_N(\alpha)=\alpha_\mathbf{R}:VAL_x(NSPEC,M)\text{->}A_{\overline{S}}(\overline{s})$ zuordnet, die wie folgt definiert ist:

Im folgenden sei $v \in VAL_x(NSPEC,M)$ eine beliebige Auswertung der freien Variablen von $x \in X$ unter M in A.

(1) Wenn $\alpha = \pi \in FORM(SIG,V_S)$ eine atomare Formel ist, dann ist $\alpha_\mathbf{R}=\mathbf{R}_{EF}(\alpha)|_{VAL_x(NSPEC,A)}$.

(2) Wenn α eine Formel der Gestalt $o^i(\beta)$ $(i \le l)$ mit $o^i \in L_1$ ist, und der Wert $\beta_\mathbf{R}(v)$ definiert ist, dann ist $\alpha_\mathbf{R}(v)=A_{\overline{\Omega}}(o^i)(\beta_\mathbf{R}(v))$.

(3) Wenn α eine Formel der Gestalt $(\beta)o_i(\gamma)$ $(i \le m)$ mit $o_i \in L_2$ ist, und die Werte $\beta_\mathbf{R}(v)$ und $\gamma_\mathbf{R}(v)$ definiert sind, dann ist $\alpha_\mathbf{R}(v)=A_{\overline{\Omega}}(o_i)(\beta_\mathbf{R}(v),\gamma_\mathbf{R}(v))$.

(4) Wenn α eine Formel der Gestalt $\forall_\zeta\beta$ ist, so gilt:

  (1) wenn die gebundene Variable $\zeta \in bvars(x)$ in der Formel β nicht vorkommt, so ist $(\forall_\zeta\beta)(v)=\beta_\mathbf{R}(v)$;

  (2) wenn die Variable $\zeta \in bvars(x)$ in der Formel β vorkommt, dann ist $(\forall_\zeta\beta)(v)=inf_{a \in obj_{\zeta,x}(M)}\{\beta_\mathbf{R}(\zeta/a)(v)\}$, wobei $\beta(\zeta/a)$ eine Formel ist, die aus β durch die Substitution $\zeta \rightarrow a$ mit $a \in obj_{\zeta,x}(M)$ entsteht, und *inf* die Operation der unteren Schranke in der Algebra $\overline{A}$ ist.

(5) Wenn α eine Formel der Gestalt $\exists_\zeta\beta$ ist, so gilt:

  (1) wenn die gebundene Variable $\zeta \in bvars(x)$ in der Formel β nicht vorkommt, so ist $(\exists_\zeta\beta)(v)=\beta_\mathbf{R}(v)$;

  (2) wenn die Variable $\zeta \in bvars(x)$ in der Formel β vorkommt, dann ist $(\exists_\zeta\beta)(v)=sup_{a \in obj_{\zeta,x}(M)}\{\beta_\mathbf{R}(\zeta/a)(v)\}$, wobei $\beta(\zeta/a)$ eine Formel ist, die aus β durch die Substitution $\zeta \rightarrow a$ mit $a \in obj_{\zeta,x}(M)$ entsteht, und *sup* die Operation der oberen Schranke in der Algebra $\overline{A}$ ist.

Man beachte, daß damit die Interpretation der Quantoren und damit der Formeln der Netzbeschriftung von der Familie $M=(m_p)_{p \in P}$ abhängt. Diese Interpretation der Formeln ist unseres Wissens nach neu in der Petri-Netz-Theorie, ist jedoch nur eine konsequente Anwendung des Lokalitätsprinzips, da der Zustandsübergang nur vom momentanen Zustand des Systems in der Umgebung einer Transition abhängig sein soll. Der Zustand des Systems in der Umgebung einer Transition wird aber gerade durch die Markierung der Umgebungsstellen bestimmt, und die einzigen Individuen, die die Transition kennt, sind daher die Objekte, die auf den Umgebungsstellen liegen. Logischerweise sollten die Quantoren nur über diese Objekte interpretiert werden.

Def. 34 (Realisation einer SNL-Netz-Spezifikation)

Es seien die Voraussetzungen wie in Definition 30 gegeben. Das Viertupel $\mathbf{R}=(\mathbf{R}_T, \mathbf{R}_F, \mathbf{R}_{FSM}, \mathbf{R}_N)$ nennen wir dann eine *Realisation der SNL-Netz-Spezifikation NSPEC.*

Da aus dem Kontext stets ersichtlich ist, welche der Abbildungen $\mathbf{R}_T$, $\mathbf{R}_F$, $\mathbf{R}_{FSM}$ bzw. $\mathbf{R}_N$ anzuwenden ist, werden wir im folgenden kurz $\mathbf{R}$ für $\mathbf{R}_T$, $\mathbf{R}_F$, $\mathbf{R}_{FSM}$ bzw. $\mathbf{R}_N$ schreiben.

Wenn wir eine Sprache L über der Signatur *LBOOL* haben, können wir diese in Bezug auf die klassische boolesche Matrix *A-BOOL* realisieren. Die Realisation von Formeln aus *EQN(NSPEC)* und *NEQN(NSPEC)* hängt dann nur noch von dem algebraischen System A und bei Formeln aus *NEQN(NSPEC)* noch von der aktuellen Markierung ab. Wir sprechen dann von einer *(prädikatenlogischen) SNL-Netz-Spezifikation (1. Ordnung)* und einer Realisierung bzgl. der Prädikatenlogik 1. Ordnung. Die Formeln von SNL-Spezifikationen sind stets solche prädikatenlogischen Formeln 1. Ordnung und Realisierungen sind bzgl. unserer Sprache stets schon eindeutig durch das algebraische System A bestimmt.

Def. 35 (Erfüllbarkeit von Formeln einer SNL-Netz-Spezifikation)

Es seien die Voraussetzungen wie in Definition 29 gegeben, und $\mathbf{R}=(\mathbf{R}_T, \mathbf{R}_F, \mathbf{R}_{FSM}, \mathbf{R}_N)$ sei eine Realisation der SNL-Netz-Spezifikation *NSPEC* über $A, \overline{A}$ und ρ_Π.

(a) Sei $\alpha \in NEQN(NSPEC)$ eine beliebige Formel von $x \in X$. Die Formel α ist *vermöge* einer Auswertung $v \in VAL_x(NSPEC,M)$ *bzgl. der Realisation* **R** *unter M erfüllt*, wenn $\alpha_{\mathbf{R}}(v)=v$ gilt. Wir schreiben dies als $M \models_{v,\mathbf{R}} \alpha$.

(b) Die Formel $\alpha \in NEQN(NSPEC)$ aus der Umgebung von $x \in X$ heißt *erfüllbar unter M bzgl. der Realisation* **R**, wenn es eine Auswertung $v \in VAL_x(NSPEC,M)$ gibt, vermöge der α bzgl. **R** erfüllt ist.

(c) Die Formel $\alpha \in NEQN(NSPEC)$ aus der Umgebung von $x \in X$ heißt *gültig unter M bzgl. der Realisation* **R**, wenn für jede Auswertung $v \in VAL_x(NSPEC,M)$ gilt: α ist vermöge v bzgl. **R** unter M erfüllt. Wir schreiben dies als $M \models_{\mathbf{R}} \alpha$.

(d) *M erfüllt die SNL-Netz-Spezifikation NSPEC bzgl. der Realisation* **R** (im Zeichen $M \models_{\mathbf{R}} NSPEC$), falls gilt:

(1) Für alle $p \in P$ gilt: $M \models_{\mathbf{R}} eqn(p)$ (Formeln auf Stellen erfüllt).

(2) $0 \leq card_s(m_p) \leq cap_p(s)$ für alle $s \in S$ und $p \in P$.

Wir lassen im folgenden bei der Schreibweise $\models$ den Suffix **R** weg, wenn die Realisation einer SNL-Netz-Spezifikation *NSPEC* aus dem Kontext ersichtlich ist.

Mit der obigen Definition können wir nun definieren, was wir unter einem SNL-System (auch SNL-Modell) und Markierungen solcher SNL-Systeme verstehen.

Def. 36 (SNL-Systeme (SNL-Modelle) und deren Anfangsmarkierung)

Es seien die Voraussetzungen wie in der Definition 29 gegeben und $\mathbf{R}=(\mathbf{R}_T, \mathbf{R}_F, \mathbf{R}_{FSM}, \mathbf{R}_N)$ eine Realisation der SNL-Netz-Spezifikation *NSPEC* über $A, \overline{A}$ und ρ_Π.

Das Tupel $SNL\text{-}SYS=(NSPEC,A)$ heißt dann ein *SNL-System* (oder *SNL-Modell*) *mit der Realisierung* **R** *über* $A, \overline{A}$ und ρ_Π, falls gilt: $M_0 := \mathbf{R}_{FSM}(M_0^f) \models_{\mathbf{R}} NSPEC$. $M_0 = \mathbf{R}_{FSM}(M_0^f)$ heißt in diesem Falle *Anfangsmarkierung des SNL-Systems* (*SNL-Modells*) *SNL-SYS*. Dabei ist $\mathbf{R}_{FSM}(M_0^f)$ gegeben durch die Familie $\mathbf{R}_{FSM}(M_0^f)=(\mathbf{R}_{FSM}(m_{0,p}^f))_{p \in P}$.

Wir werden im folgenden mit einem SNL-System *SNL-SYS* stets eine Realisierung **R** über A, einer logischen Matrix $\overline{A}$ und Familie von Funktionen ρ_Π voraussetzen, ohne daß wir $\overline{A}$, ρ_Π oder **R** explizit angeben werden, sofern wir nicht verschiedene Realisierungen ein und desselben SNL-Systems über verschiedene $\overline{A}$ und/oder ρ_Π betrachten, was kaum vorkommt. Insbesondere unterdrücken wir daher auch im allgemeinen das Subscript **R** beim Begriff des Erfülltsein von Formeln. Bei SNL-Netz-Spezifikationen ist die Realisation sowieso durch die Angabe von A in der SNL-System-Schreibweise $SNL\text{-}SYS=(NSPEC,A)$ eindeutig festgelegt, weil wir hier nur Realisierungen der Prädikate über die Relationen in A betrachten. Netz-Spezifikationen in unserer Spezifikations- und Modellsprache SNL sind daher stets schon durch Angabe eines Modells A zur Spezifikation *SPEC* eindeutig als SNL-System interpretierbar!

4.4 Das Schalten von SNL-Systemen

Bevor wir uns den Begriffen des Schaltens von Schritten bzw. Transitionen in SNL-Systemen zuwenden, wollen wir zunächst einen linear-algebraischen Kalkül im Umgang mit Familien von Modulen über einen Ring (in unserem Fall Multimengen über den Ring $\mathbb{Z}$) vorstellen, der es uns erlaubt, Schritt- bzw. Schaltvorgänge auf mathematisch einfache und elegante Weise zu beschreiben.

4.4.1 Ein kurzer Exkurs in die lineare Algebra

Familien von Modulen über einen gemeinsamen Ring kann man in natürlicher Weise selbst als ein Modul auffassen, den man direktes Produkt der Familie von Modulen nennt.

Def. 37 (kartesisches Produkt einer Familie von Mengen)

Sei R ein Ring, I eine beliebige Indexmenge und $A=((A_i,+_i,\cdot_i))_{i \in I}$ eine Familie von R-Modulen. Wir definieren dann:

(a) Das *kartesische Produkt der Familie von Mengen* $(A_i)_{i \in I}$ ist definiert durch:

$$A_I := \prod_{i \in I} A_i := \{ (a_i)_{i \in I} \mid a_i \in A_i \; \forall i \in I \}$$

Ein I-Tupel $a_I := (a_i)_{i \in I} \in A_I$ nennen wir im folgenden auch einen *I-Vektor* (über der Familie A).

(b) Auf A_I definieren wir eine Operation $+ : A_I \times A_I \to A_I$ durch $a_I + b_I = (a_i)_{i \in I} + (b_i)_{i \in I} := (a_i +_i b_i)_{i \in I} \; \forall a_I, b_I \in A_I$. Wir nennen $+$ die Addition auf A_I.

(c) Weiter definieren wir eine *Skalarmultiplikation* $\cdot : R \times A_I \to A_I$ durch $\lambda \cdot a_I = \lambda \cdot (a_i)_{i \in I} := (\lambda \cdot_i a_i)_{i \in I}$ $\forall \lambda \in R, \forall a_I \in A_I$.

Es ist dann leicht nachzuweisen, daß die folgenden Rechenregeln gelten:

Bemerkung 3:

Es seien die Voraussetzungen wie in der vorhergehenden Definition gegeben. Für alle $a_I = (a_i)_{i \in I}$, $b_I = (b_i)_{i \in I}$, $c_I = (c_i)_{i \in I} \in A_I$ und $r, s \in R$ gelten die folgenden Regeln:

(a) $r \cdot (a_I + b_I) = (r \cdot a_I) + (r \cdot b_I), (r + s) \cdot a_I = (r \cdot a_I) + (s \cdot a_I)$

(b) $(a_I + b_I) + c_i = a_I + (b_I + c_i)$

(c) Mit $0_I = (0_i)_{i \in I}$, wobei 0_i das Nullelement des Moduls $(A_i, +_i, \cdot_i)$ ist, gilt:
$0_I + a_I = a_I = a_I + 0_I, 0 \cdot a_I = 0_I$

(d) Sei $1 \in R$ das Einselement des Ringes R und -1 das inverse Element von 1 in R. Dann gilt:
$a_I + (-1) \cdot a_I = (-1) \cdot a_I + a_I = 0_I$
d.h. jedes $a_I \in A_I$ besitzt ein inverses Element $-a_I := (-1) \cdot a_I$

(e) Es läßt sich eine Operation *Subtraktion* $- : A_I \times A_I \to A_I$ durch $a_I - b_I := a_I + (-1) \cdot b_I$ einführen.

(f) Sind die R-Module $((A_i, +_i, \cdot_i))_{i \in I}$ alle kommutativ, so ist auch $(A_I, +, \cdot)$ kommutativ, d.h. es gilt dann:
$a_I + b_I = b_I + a_I$.

Die oben angegebenen Rechenregeln besagen gerade, daß $(A_I, +, \cdot)$ ebenfalls ein R-Modul ist. Dies führt zu der folgenden Definition.

Def. 38 (direktes Produkt einer Familie von Modulen)

Sei R ein Ring, I eine beliebige Indexmenge und $A = ((A_i, +_i, \cdot_i))_{i \in I}$ eine Familie von R-Modulen. Dann ist $(A_I, +, \cdot)$ ebenfalls ein R-Modul, den wir das *direkte Produkt der Familie von R-Modulen A* nennen.

Der einfacheren Schreibweise wegen bezeichnen wir im folgenden die Operationen aller beteiligten Module stets mit $+$ und $\cdot$. Wir lassen also die Indizes weg und schreiben eine Familie von Modulen kurz als $A = (A_i)_{i \in I}$. Wenn aus dem Kontext ersichtlich ist, wo eine Skalarmultiplikation zu stehen hat, lassen wir der mathematischen Konvention zu Folge oft auch den Punkt ($\cdot$) ganz weg. Weiter schreiben wir kurz A_I für $(A_I, +, \cdot)$, d.h. A_I steht einmal für den Modul $(A_I, +, \cdot)$ und ein anderes Mal nur für die Menge $A_I := \prod_{i \in I} A_i$.

Wenn die Indexmenge $I = \{ i_1, ..., i_k \}$, $k \in \mathbb{N}, 0 \leq k$, endlich ist, können wir sie willkürlich anordnen, wie dies in der obigen Mengenschreibweise schon implizit durch die Indizierung der Elemente erfolgt ist. Wir können dann einen I-Vektor $a_I = (a_i)_{i \in I}$ in der Form

$$\begin{pmatrix} a_{i_1} \\ \dots \\ a_{i_k} \end{pmatrix} \begin{array}{l} \text{<- Komponente } i_1 \\ \\ \text{<- Komponente } i_k \end{array}$$

darstellen, wie man dies für Vektoren aus der Schule gewöhnt ist. Addition und Skalarmultiplikation verstehen sich dann als die bekannten komponentenweise durchgeführten Operationen innerhalb dieser Schreibweise.

Nachdem wir nun wissen, wie wir mit Familien von R-Modulen umgehen, wollen wir uns nun ansehen, wie wir Familien von Modulhomomorphismen behandeln. Hierzu definieren wir zunächst:

Def. 39 (Der Begriff einer Matrix von Modulhomomorphismen)

Sei R ein Ring, I und J Indexmengen und $A=(A_i)_{i\in I}$ und $B=(B_j)_{j\in J}$ Familien von R-Modulen.

(a) Eine $A\times B$- (bzw. $A_I\times B_J$-) *Matrix* ist eine Familie von Modulhomomorphismen $\Theta=(\Theta_{ij}:A_i\to B_j)_{i\in I,j\in J}$ (d.h. jede Abbildung $\Theta_{ij}:A_i\to B_j$ ist ein Modulhomomorphismus).

(b) Die Menge aller $A\times B$-Matrizen bezeichnen wir mit $[A,B]$ (bzw. $[A_I,B_J]$).

(c) Auf $[A,B]$ definieren wir eine Operation *Addition* $+:[A,B]\times[A,B]\to[A,B]$ durch $\Theta+\Psi=(\Theta_{ij})_{i\in I,j\in J}+(\Psi_{ij})_{i\in I,j\in J}:=(\Theta_{ij}+\Psi_{ij})_{i\in I,j\in J}\ \forall\Theta,\Psi\in[A,B]$.

(d) Wir definieren eine Operation *Skalarmultiplikation* $\cdot:R\times[A,B]\to[A,B]$ durch $\lambda\cdot\Theta=\lambda\cdot(\Theta_{ij})_{i\in I,j\in J}:=(\lambda\cdot\Theta_{ij})_{i\in I,j\in J}\ \forall\lambda\in R,\ \forall\Theta\in[A,B]$.

Man sieht dann leicht, daß für $([A,B],+,\cdot)$ die gleichen Rechenregeln gelten wie für direkte Produkte. Wir haben also:

Bemerkung 4:

$[A,B]$ ist mit den oben eingeführten Operationen + und $\cdot$ ebenfalls ein R-Modul.

Wenn I und J endliche Indexmengen sind und wir wiederum eine willkürliche Ordnung der Elemente von I und J, wie sie z.B. durch die Mengenschreibweisen $I=\{i_1,\ldots,i_n\}$, $n\geq 1$, $n\in\mathbb{N}$, und $J=\{j_1,\ldots,j_m\}$, $m\geq 1$, $m\in\mathbb{N}$ nahegelegt wird, voraussetzen, können wir ein Element $\Theta\in[A_I,B_J]$ eindeutig in einer Matrixschreibweise

$$\begin{array}{l} i_1 \text{ ----------} \\ \\ \\ i_n \text{ ----------} \end{array} \begin{pmatrix} \Theta_{i_1,j_1} & \cdots & \Theta_{i_1,j_m} \\ \cdots & \Theta_{i_l,j_k} & \cdots \\ \Theta_{i_n,j_1} & \cdots & \Theta_{i_n,j_m} \end{pmatrix}$$

$$\qquad\quad j_1 \qquad\qquad j_n$$

darstellen. Die Addition zweier Matrizen $\Theta,\Psi\in[A,B]$ ergibt sich dann in dieser Schreibweise als die Addition korrespondierender Elemente der Matrix innerhalb der Matrixschreibweise. Die Skalarmultiplikation bedeutet die Multiplizierung jedes Elementes innerhalb der Matrixschreibweise mit dem gegebenen Faktor.

Als nächstes wollen wir nun definieren, was es heißt, eine Matrix $\Theta\in[A,B]$ auf einen I-Vektor $a_I\in A_I$ anzuwenden.

Def. 40 (Anwendung von Matrizen auf I-Vektoren)

Sei R ein Ring, I und J endliche Indexmengen und $A=(A_i)_{i\in I}$ und $B=(B_j)_{j\in J}$ Familien von R-Modulen. Wir definieren dann eine Operation $\cdot:[A_I,B_J]\times A_I\to B_J$ durch

$$\Theta\cdot a_I:=((\Theta\cdot a_I)_j)_{j\in J}:=\Big(\sum_{i\in I}\Theta_{i,j}(a_i)\Big)_{j\in J}\qquad\forall\Theta=(\Theta_{ij})_{i\in I,j\in J}\in[A_I,B_J],\ \forall\, a_I=(a_i)_{i\in I}\in A_I$$

Wir sagen: $\Theta\cdot a_I$ ist die *Anwendung der Matrix* Θ *auf den I-Vektor* a_I.

Wenn wir bei der oben eingeführten Operation $\cdot:[A_I,B_J]\times A_I\to B_J$ ein Element $\Theta=(\Theta_{ij})_{i\in I,j\in J}\in[A_I,B_J]$ als konstant festhalten, so erhalten wir eine Abbildung $\Theta=\cdot(\Theta,\_):A_I\to B_J$, $\Theta(a_I)=\Theta\cdot a_I$. Auf Grund der Homomorphismuseigenschaft sämtlicher Θ_{ij} ist auch die so eingeführte Abbildung Θ ein Modulhomomorphismus, denn es gilt:

$$\begin{aligned}(\Theta(\lambda_1\cdot a_I^1+\lambda_2\cdot a_I^2))_j &= \sum_{i\in I}\Theta_{i,j}(\lambda_1\cdot a_i^1+\lambda_2\cdot a_i^2)\\ &= \sum_{i\in I}(\lambda_1\cdot\Theta_{i,j}(a_i^1)+\lambda_2\cdot\Theta_{i,j}(a_i^2))\\ &= \lambda_1\cdot\sum_{i\in I}\Theta_{i,j}(a_i^1)+\lambda_2\cdot\sum_{i\in I}\Theta_{i,j}(a_i^2)\\ &= \lambda_1\cdot(\Theta\cdot a_I^1)_j+\lambda_2\cdot(\Theta\cdot a_I^2)_j\end{aligned}$$

Damit haben wir also: $\Theta(\lambda_1\cdot a_I^1+\lambda_2\cdot a_I^2)=\lambda_1\cdot(\Theta\cdot a_I^1)+\lambda_2\cdot(\Theta\cdot a_I^2)$.

Analog erhalten wir bei Festhalten von a_I eine Abbildung $a_I:[A_I,B_J]\to B_J$, $a_I(\Theta)=\Theta\cdot a_I$, die ebenfalls ein Modulhomomorphismus ist. Es gilt also die folgende Bemerkung:

Bemerkung 5:

Sei R ein Ring, I und J endliche Indexmengen und $A=(A_i)_{i\in I}$ und $B=(B_j)_{j\in J}$ Familien von R-Modulen.

(a) Für jede Matrix $\Theta=(\Theta_{i,j})_{i\in I,j\in J}\in [A_I,B_J]$ ist die Abbildung $\Theta= \cdot (\Theta,\_) : A_I \to B_J$, $\Theta(a_I)=\Theta \cdot a_I$ $\forall\, a_I=(a_i)_{i\in I}\in A$ ein R-Modulhomomorphismus zwischen A_I und B_J, d.h. es gilt die Beziehung

$\Theta(\lambda_1 \cdot a_I^1 + \lambda_2 \cdot a_I^2)=\lambda_1 \cdot (\Theta \cdot a_I^1) + \lambda_2 \cdot (\Theta \cdot a_I^2)$ $\forall a_I^1, a_I^2\in A_I$ und $\forall \lambda_1, \lambda_2\in R$

(b) Für jedes $a_I\in A_I$ ist die Abbildung $a_I : [A_I,B_J] \to B_J$, $a_I(\Theta)=\Theta \cdot a_I$ $\forall \Theta\in [A_I,B_J]$ ein R-Modulhomomorphismus zwischen $[A_I,B_J]$ und B_J, d.h. es gilt die Gleichung

$(\lambda_1 \cdot \Theta + \lambda_2 \cdot \Psi) \cdot a_I = \lambda_1 \cdot (\Theta \cdot a_I) + \lambda_2 \cdot (\Psi \cdot a_I)$ $\forall \Theta, \Psi\in [A_I,B_J]$ und $\forall \lambda_1, \lambda_2\in R$.

Wir wollen nun noch ein Produkt von Matrizen von Familien von Modulhomomorphismen einführen. Dazu sei R wiederum ein Ring, I, J und K seien endliche Indexmengen und $A=(A_i)_{i\in I}$, $B=(B_j)_{j\in J}$ und $C=(C_k)_{k\in K}$ seien drei Familien von R-Modulen. Weiter sei $\Theta=(\Theta_{i,j})_{i\in I,j\in J}\in [A_I,B_J]$ und $\Psi=(\Psi_{j,k})_{j\in J,k\in K}\in [B_J,C_K]$. Die Summe $\sum_{j\in J} \Psi_{j,k} o \Theta_{i,j}$, wobei o die Komposition von Abbildungen ist, ist dann eine wohldefinierte Abbildung $\sum_{j\in J} \Psi_{j,k} o \Theta_{i,j} : A_i \to C_k$ und es gilt weiter

$$
\begin{aligned}
(\sum_{j\in J} \Psi_{j,k} o \Theta_{i,j})\,(\lambda_1 \cdot a_i^1 + \lambda_2 \cdot a_i^2) &= \sum_{j\in J} ((\Psi_{j,k} o \Theta_{i,j})\,(\lambda_1 \cdot a_i^1 + \lambda_2 \cdot a_i^2)) \\
&= \sum_{j\in J} \Psi_{j,k}(\Theta_{i,j}(\lambda_1 \cdot a_i^1 + \lambda_2 \cdot a_i^2)) \\
&= \sum_{j\in J} \Psi_{j,k}(\lambda_1 \cdot \Theta_{i,j}(a_i^1) + \lambda_2 \cdot \Theta_{i,j}(a_i^2)) \\
&= \sum_{j\in J} (\lambda_1 \cdot \Psi_{j,k}(\Theta_{i,j}(a_i^1)) + \lambda_2 \cdot \Psi_{j,k}(\Theta_{i,j}(a_i^2))) \\
&= \lambda_1 \cdot \sum_{j\in J} (\Psi_{j,k} o \Theta_{i,j})\,(a_i^1) + \lambda_2 \cdot \sum_{j\in J} (\Psi_{j,k} o \Theta_{i,j})\,(a_i^2)
\end{aligned}
$$

d.h. aber, daß $\sum_{j\in J} \Psi_{j,k} o \Theta_{i,j} : A_i \to C_k$ für alle $i\in I$ und $k\in K$ ein Modulhomomorphismus ist.

Damit ist $\Psi \cdot \Theta := (\sum_{j\in J} \Psi_{j,k} o \Theta_{i,j})_{i\in I,k\in K}\in [A_I,C_K]$. Wir definieren daher:

Def. 41 (Produkt von Matrizen von Familien von Modulhomomorphismen)

Sei R wiederum ein Ring, I, J und K seien endliche Indexmengen und $A=(A_i)_{i\in I}$, $B=(B_j)_{j\in J}$ und $C=(C_k)_{k\in K}$ seien drei Familien von R-Modulen. Die Operation $\cdot : [B_J,C_K]\times[A_I,B_J] \to [A_I,C_K]$ sei definiert durch

$$\Psi \cdot \Theta := ((\Psi \cdot \Theta)_{i,k})_{i\in I,k\in K} := (\sum_{j\in J} \Psi_{j,k} o \Theta_{i,j})_{i\in I,k\in K}\ \forall \Psi=(\Psi_{j,k})_{j\in J,k\in K}\in [B_J,C_K],\ \Theta=(\Theta_{i,j})_{i\in I,j\in J}\in [A_I,B_J]$$

Für $\Psi\in [B_J,C_K]$ und $\Theta\in [A_I,B_J]$ heißt $\Psi \cdot \Theta$ das *Produkt der Matrizen* Ψ *und* Θ.

Für die Matrixmultiplikation gelten die folgenden Rechenregeln, wie man leicht durch Nachrechnen zeigt.

Bemerkung 6:

Sei R ein Ring, I, J, K und L seien endliche Indexmengen und $A=(A_i)_{i\in I}$, $B=(B_j)_{j\in J}$, $C=(C_k)_{k\in K}$ und $D=(D_l)_{l\in L}$ seien vier Familien von R-Modulen. Dann gilt:

(a) Für jedes $\Psi\in [B_J,C_K]$ ist die Abbildung $\Psi : [A_I,B_J] \to [A_I,C_K]$, $\Psi(\Theta):=\Psi \cdot \Theta$ $\forall \Theta\in [A_I,B_J]$, ein R-Modulhomomorphismus, d.h. es gelten die Gleichungen

(1) $\Psi \cdot (\Theta^1+\Theta^2) = (\Psi \cdot \Theta^1)+(\Psi \cdot \Theta^2)$ $\forall \Theta^1, \Theta^2\in [A_I,B_J]$

(2) $\Psi \cdot (\lambda \cdot \Theta) = \lambda \cdot (\Psi \cdot \Theta)$ $\forall \lambda\in R$ und $\forall \Theta\in [A_I,B_J]$

(b) Für jedes $\Theta\in [A_I,B_J]$ ist die Abbildung $\Theta : [B_J,C_K] \to [A_I,C_K]$, $\Theta(\Psi):=\Psi \cdot \Theta$ $\forall \Psi\in [B_J,C_K]$, ein R-Modulhomomorphismus, d.h. es gelten die Gleichungen

(1) $(\Psi^1+\Psi^2) \cdot \Theta = (\Psi^1 \cdot \Theta)+(\Psi^2 \cdot \Theta)$ $\forall \Psi^1, \Psi^2\in [B_J,C_K]$

(2) $(\lambda \cdot \Psi) \cdot \Theta = \lambda \cdot (\Psi \cdot \Theta)$ $\forall \lambda\in R$ und $\forall \Psi\in [B_J,C_K]$

(c) Für alle $\Theta\in [A_I,B_J]$, $\Psi\in [B_J,C_K]$ und $\Phi\in [C_K,D_L]$ gilt: $\Phi \cdot (\Psi \cdot \Theta) = (\Phi \cdot \Psi) \cdot \Theta$.

(d) Für jedes $\Psi\in [B_J,C_K]$, $\Theta\in [A_I,B_J]$ und $a_I\in A_I$ gilt: $(\Psi \cdot \Theta)(a_I) = \Psi \cdot (\Theta \cdot a_I)$.

Damit haben wir alle Begriffe definiert, die wir benötigen, um Schritt- bzw. Schaltvorgänge in SNL-Systemen mit linear-algebraischen Mitteln zu beschreiben. Dieser Aufgabe wollen wir uns im nächsten Abschnitt widmen.

4.4.2 Lineare Algebra von SNL-Systemen

Im Abschnitt über SNL-Systeme haben wir zur Beschreibung der Anfangsmarkierung eines SNL-Systems Familien von Multimengen $M=(m_p)_{p\in P}$ mit $m_p\in MULT^+_{fin}(A)$, $typ(m_p)\subseteq sorts(p)\ \forall p\in P$, verwendet. Wie wir schon im Kapitel über Multimengen festgehalten haben, können wir $MULT_{fin}(A)$ und damit die Mengen $M_p(NSPEC,A):=\{m_p\in MULT_{fin}(A)\mid typ(m_p)\subseteq sorts(p)\}\ \forall p\in P$ als $\mathbb{Z}$-Module auffassen. Gemäß den Definitionen im vorherigen Unterkapitel sind daher Familien $M_P=(m_p)_{p\in P}$, $m_p\in M_p(NSPEC,A)$, P-Vektoren bzgl. der Familie $M(NSPEC,A):=M_P(NSPEC,A):=(M_p(NSPEC,A))_{p\in P}$ von $\mathbb{Z}$-Modulen. Dies führt zu der folgenden Definition und Bemerkung.

Def. 42 (P-Vektoren von SNL-Systemen)

Sei *SNL-SYS*=(*NSPEC*,*A*) ein SNL-System.

(a) $M_p(NSPEC,A):=\{m_p\in MULT_{fin}(A)\mid typ(m_p)\subseteq sorts(p)\}$ ist $\forall p\in P$ ein $\mathbb{Z}$-Modul.

(b) Das direkte Produkt $M(NSPEC,A):=M_P(NSPEC,A):=(M_p(NSPEC,A))_{p\in P}$ ist ebenfalls ein $\mathbb{Z}$-Modul, dessen Elemente wir *P-Vektoren des SNL-Systems SNL-SYS* nennen.

(c) Analog zu (a) definieren wir $M^+_p(NSPEC,A):=\{m_p\in MULT^+_{fin}(A)\mid typ(m_p)\subseteq sorts(p)\}\subseteq M_p(NSPEC,A)$.

(d) Analog zu (b) definieren wir $M^+(NSPEC,A)=M^+_P(NSPEC,A)=(M^+_p(NSPEC,A))_{p\in P}$.

Gemäß der obigen Definition sind die erlaubten Objektverteilungen innerhalb eines SNL-Systems *SNL-SYS*=(*NSPEC*,*A*) offensichtlich *P*-Vektoren, die zusätzlich die Netz-Spezifikation erfüllen (d.h. $M\models_{\mathbb{R}} NSPEC$).

Def. 43 (Markierungen von SNL-Systemen)

Sei *SNL-SYS*=(*NSPEC*,*A*) ein SNL-System. Ein *P*-Vektor $M\in M(NSPEC,A)$ heißt (*mögliche*) *Markierung des SNL-Systems SNL-SYS*, falls gilt: *M* erfüllt *NSPEC* (d.h. $M\models_{\mathbb{R}} NSPEC$).

Die Menge aller möglichen Markierungen bezeichnen wir mit $MARK(NSPEC,A)$.

Wir haben die natürlichen Inklusionen $MARK(NSPEC,A)\subseteq M^+(NSPEC,A)\subseteq M(NSPEC,A)$. Dabei übertragen sich die Moduloperationen in natürlicher Weise von $M(NSPEC,A)$ auf $M^+(NSPEC,A)$, nicht aber auf $MARK(NSPEC,A)$. Letzteres scheitert nicht nur an den Kapazitätsbedingungen $0\le cap_s(m_p)\le cap_p(s)\ \forall p\in P,\ s\in S$, sondern in der Regel auch an den Formeln der Stellen $p\in P$. Die Formel (all x : (x>4)) orelse (all x : (x<2)) wird z.B. bzgl. der üblichen Semantik von natürlichen Zahlen von den Multimengen 1`5 und 1`1, nicht aber von der Multimenge 1`5+1`1 erfüllt.

Jeder Transition $t\in T$ eines SNL-Systems *SNL-SYS*=(*NSPEC*,*A*) ist vermöge der Netzbeschriftung eine Menge von freien Variablen *fvars*(*t*) zugeordnet. Dabei hängt die Formel der Transition und die Beschriftung der Kanten in der Umgebung der Transition *t* nur von diesen Variablen als freien Variablen ab. Jede Auswertung $v\in VAL_t(NSPEC,A)$ dieser freien Variablen im algebraischen System *A* bietet daher die Möglichkeit, die Transitionsformel auf Erfülltheit zu überprüfen und die Kantenbeschriftungen in der Umgebung der Transition auszuwerten, und stellt damit einen potentiellen Modus dar, in dem die Transition *t* schalten kann.

Wenn wir es zulassen wollen, daß eine Transition parallel (nebenläufig) zu sich selbst schalten darf, so lassen sich solche nebenläufigen Schaltvorgänge von *t* allgemeiner durch Multimengen von Auswertungen $MULT^+_{fin}(VAL_t(NSPEC,A))$ beschreiben. Lassen wir noch allgemeiner nicht nur die Nebenläufigkeit einer Transition sondern das nebenläufige Schalten von Transitionen, die ihrerseits jede für sich nebenläufig zu sich selbst schalten kann, zu, so werden solche allgemeinen Schaltvorgänge durch Familien von Multimengen von Auswertungen der freien Variablen der beteiligten Transitionen, d.h. durch Familien $\mu=(\mu_t)_{t\in T}$, $\mu_t\in MULT^+_{fin}(VAL_t(NSPEC,A))$ für alle $t\in T$, beschrieben. Dies sind aber gerade *T*-Vektoren bzgl. der Familie von $\mathbb{Z}$-Modulen $(MULT_{fin}(VAL_t(NSPEC,A)))_{t\in T}$. Wir definieren daher:

Def. 44 (T-Vektoren)

Sei *SNL-SYS*=(*NSPEC*,*A*) ein SNL-System.

(a) Die Menge $S_t(NSPEC,A):=MULT_{fin}(VAL_t(NSPEC,A))$ der Multimengen aller Auswertungen der freien Variablen einer Transition $t\in T$ ist ein $\mathbb{Z}$-Modul.

(b) Das direkte Produkt $S(NSPEC,A):=S_T(NSPEC,A):=(S_t(NSPEC,A))_{t\in T}$ ist ebenfalls ein $\mathbb{Z}$-Modul, dessen Elemente wir *T-Vektoren des SNL-Systems SNL-SYS* nennen.

(c) Analog zu (a) definieren wir die Menge $S_t^+(NSPEC,A):=MULT_{fin}^+(VAL_t(NSPEC,A))$.

(d) Analog zu (b) definieren wir die Menge $S^+(NSPEC,A):=S_T^+(NSPEC,A):=(S_t^+(NSPEC,A))_{t\in T}$.

(e) Sei $M\in MARK(NSPEC,A)$ eine Markierung.
Dann ist die Menge $S_t(NSPEC,M):=MULT_{fin}(VAL_t(NSPEC,M))$ der Multimengen von Auswertungen der freien Variablen einer Transition $t\in T$ unter der Markierung M in A ebenfalls ein $\mathbb{Z}$-Modul.

(f) Sei $M\in MARK(NSPEC,A)$ eine Markierung.
Das direkte Produkt $S(NSPEC,M):=S_T(NSPEC,A):=(S_t(NSPEC,M))_{t\in T}$ ist ebenfalls ein $\mathbb{Z}$-Modul, dessen Elemente wir *T-Vektoren des SNL-Systems SNL-SYS unter der Markierung M* nennen.

(g) Analog zu (e) definieren wir die Menge $S_t^+(NSPEC,M):=MULT_{fin}^+(VAL_t(NSPEC,M))$.

(h) Analog zu (f) definieren wir die Menge $S^+(NSPEC,M):=S_T^+(NSPEC,A):=(S_t^+(NSPEC,M))_{t\in T}$.

Es gelten offensichtlich die Teilmengenrelationen $S^+(NSPEC,M)\subseteq(NSPEC,M)\subseteq S(NSPEC,A)$ $\forall M\in MARK(NSPEC,A)$. Weiter haben wir $S^+(NSPEC,M)\subseteq S^+(NSPEC,A)\subseteq S(NSPEC,A)$.

Sei nun $M\in MARK(NSPEC,A)$. Dann stellen die T-Vektoren $\mu\in S^+(NSPEC,A)$ potentielle Kandidaten zur Beschreibung eines Schaltvorganges ausgehend von der Markierung M im SNL-System $SNL\text{-}SYS=(NSPEC,A)$ dar. Wenn wir die Einhaltung des Lokalitätsprinzips verlangen, müssen wir uns sogar auf T-Vektoren $\mu\in S^+(NSPEC,M)$ beschränken. Als weitere Voraussetzung für einen Schaltvorgang sollten aber auch die Transitionsformeln von allen beteiligten Auswertungen der freien Variablen der Transitionen erfüllt werden. Dies führt zu der folgenden Definition.

Def. 45 (T-Vektoren, die Transitionsformeln erfüllen)

Sei $SNL\text{-}SYS=(NSPEC,A)$ ein SNL-System und $M\in MARK(NSPEC,A)$. Weiter sei $\mu=\left(\sum_{i_t=1}^{k_t} d_{i_t}\grave{}\, v_{i_t}\right)_{t\in T}\in(NSPEC,A)$ ein beliebiger T-Vektor. Dann definieren wir:

Der *T-Vektor* μ *erfüllt die Transitionsformeln von* $SNL\text{-}SYS=(NSPEC,A)$ *unter der Markierung M* (kurz: μ *erfüllt NSPEC unter M*), falls gilt:

$$\forall t\in T,\ i_t=1,\ \ldots,\ k_t \text{ gilt: } M\models_{v_{i_t},\mathbf{R}} NSPEC$$

Wir schreiben dies als $M\models_\mu NSPEC$.

Wir müssen uns nun noch der Interpretation der Kantenbeschriftungen bei einem Schaltvorgang zuwenden, um die dynamische Semantik eines SNL-Systems vollständig beschreiben zu können. Hierzu bemerken wir zunächst, daß die Kantenbeschriftungen $A_F(f)$ gemäß der Definition einer SNL-Netzspezifikation formale Summen mit den Randbedingungen $typ(A_F(f))\subseteq sorts(p)$ und $fvars(A_F(f))\subseteq fvars(t)$ für $f=(p,t)$ bzw. (t,p) $t\in T$, $p\in P$ sind. Die Realisierung so einer Kantenbeschriftung liefert dann eine Abbildung $\mathbf{R}_{FSM}(A_F(f))=A_F(f)_\mathbf{R}: VAL_t(NSPEC,A)\rightarrow M_p(NSPEC,A)$. Die allgemeinen Regeln für die Semantik der Kantenbeschriftungen von Petri-Netzen legen nun nahe, daß $A_F(f)_\mathbf{R}(v)$ die Multimenge von Objekten ist, die beim Schalten von t mit der Variablenauswertung $v\in VAL_t(NSPEC,A)$ von Eingabestellen (d.h. $f=(p,t)$) abzuziehen bzw. auf Ausgabestellen (d.h. $f=(t,p)$) hinzuzufügen ist. Hierzu führen wir die folgende Definition ein.

Def. 46

Sei $SNL\text{-}SYS=(NSPEC,A)$ ein SNL-System. Die Abbildung $\Delta t_p: VAL_t(NSPEC,A)\rightarrow M(NSPEC,A)$ sei für alle $p\in P$ und $t\in T$ definiert durch

$$\Delta t_p = \begin{cases} 0\in M_p(NSPEC,A) & \text{falls } (t,p) \text{ und } (p,t)\notin F \\ -A_F(f)_\mathbf{R} & \text{falls } f=(p,t)\in F \text{ und } (t,p)\notin F \\ A_F(f)_\mathbf{R} & \text{falls } (p,t)\notin F \text{ und } f=(t,p)\in F \\ -A_F((p,t))_\mathbf{R}+A_F((t,p))_\mathbf{R} & \text{falls } (p,t)\in F \text{ und } (t,p)\in F \end{cases}$$

Δt sei definiert als die Familie $\Delta t := (\Delta t_p)_{p\in P}$.

Die Folgemarkierung von M, die entsteht, wenn die Transition t mit der Variablenauswertung v schaltet, sollte sich dann durch die Gleichung $M'=M+\Delta t(v)$ ergeben. Hierbei sind aber noch die Randbedingungen zu beachten,

daß unter der Markierung M die "richtigen" Objekte in genügender Anzahl auf den Eingabestellen vorhanden sein müssen, um beim Schaltvorgang abgezogen werden zu können, und weiter $M'=M+\Delta t(v)$ eine mögliche Markierung des SNL-Systems sein muß (d.h. M' muß den Kapazitäten und Formeln der Stellen genügen). Unter Berücksichtigung dieser Randbedingungen können wir nun definieren, wann eine Transition aktivierbar ist und was wir uns unter der Schaltregel für SNL-Systeme vorzustellen haben.

Def. 47 (Aktivierbarkeit von Transitionen und Schaltregel in SNL-Systemen)

Sei *SNL-SYS*=(*NSPEC*,*A*) ein SNL-System und $M \in MARK(NSPEC,A)$ eine mögliche Markierung von *SNL-SYS*.

(a) Eine Transition $t \in T$ *heißt im Mode* $v \in VAL_t(NSPEC,A)$ *unter* M *aktivierbar*, wenn die folgenden Bedingungen erfüllt sind.

(1) $v \in VAL_t(NSPEC,M)$ (Lokalitätsprinzip)

(2) $M \models_v eqn(t)$ (Transitionsformel erfüllt)

(3) $\forall\, p \in P$ mit $(p,t) \in F$ gilt: $A_F((p,t))_R(v) \leq m_p$
(die "richtigen" Objekte in genügender Anzahl auf den Eingabestellen vorhanden)

(4) $M'=M+\Delta t(v) \in MARK(NSPEC,A)$
(d.h. die Folgemarkierung ist eine mögliche Markierung des SNL-Systems)

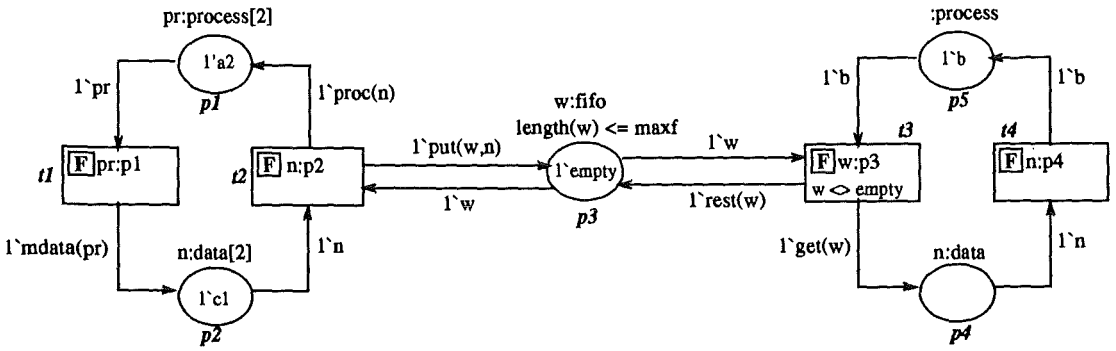
(b) Eine Transition $t \in T$ *heißt unter* M *aktivierbar*, falls ein $v \in VAL_t(NSPEC,A)$ existiert, so daß t im Mode v unter M aktivierbar ist.

(c) (Schaltregel)
Wenn eine Transition $t \in T$ in einem Mode $v \in VAL_t(NSPEC,A)$ unter einer Markierung M aktivierbar ist, so kann t im Mode v schalten, wobei das SNL-System in ein System mit der Folgemarkierung $M'=M+\Delta t(v)$ übergeht.

Wir haben nun definiert, wie wir uns einen dynamischen Zustandsübergang in einem SNL-System vorzustellen haben, wenn wir nur das Schalten jeweils einer Transition bzgl. einer Auswertung ihrer freien Variablen zu einer Zeit als atomare Operation auf den Stellen des Systems zulassen. Die Bedingung für die Aktivierbarkeit solcher Zustandsübergänge sind die Erfüllung des Lokalitätsprinzips (d.h. alle zur Auswertung v gehörigen Objekte sind auf den "richtigen" Umgebungsstellen der Transition vorhanden, also $v \in VAL_t(NSPEC,M)$), das Erfülltsein der Transitionsformel, das Vorhandensein einer ausreichenden Anzahl von den Objekten auf den Eingabestellen, die beim Schaltvorgang von diesen abzuziehen sind, sowie die Bedingung, daß die Folgemarkierung ebenfalls eine gültige Markierung des SNL-Systems ist. Wenn diese Bedingungen erfüllt sind, kann ein Zustandswechsel "Markierung M geht in die Markierung $M'=M+\Delta t(v)$ über" im System erfolgen. Wir wollen zum Schalten von Transitionen einmal ein Beispiel betrachten.

Beispiel 42 (Schalten einer Transition)

Betrachten wir die SNL-Netz-Spezifikation aus dem vorhergehenden Beispiel und realisieren sie mit einem geeigneten algebraischen System A, wobei wir annehmen, daß `mdata(a1)=c1` ist. Dann kann die Transition *t1* in dem Mode `pr` → `a1` schalten, da die Formel $A_T(t1)$ stets `true` ist und `1`a1` als Marke auf der Stelle *p1* vorhanden ist. Das Netz geht dann in die Markierung des folgenden Bildes über.



Wir führen mit der folgenden Definition noch einige Bezeichnungen ein.

Def. 48 (Schaltvorgang und Schaltfolge)

Sei *SNL-SYS*=(*NSPEC*,*A*) ein SNL-System.

(a) Unter einem *Schaltvorgang* verstehen wir ein Tripel $(M,t{:}v,M')$ mit M, $M' \in MARK(NSPEC,A)$, $t \in T$ und $v \in VAL_t(NSPEC,M)$, so daß t unter M im Mode v aktivierbar ist und $M'=M+\Delta t(v)$ die Folgemarkierung von M ist.

Um suggestiv darzustellen, daß M' aus M durch Schalten von t im Mode v entsteht, schreiben wir einen Schaltvorgang auch in der Form $M[t{:}v{>}M'$.

(b) Eine *Schaltfolge* ist eine Folge von Schaltvorgängen $(M_i,t_i{:}v_i,M_i')_{i=1,..,n}$ mit $M_i'=M_{i+1}$ für $i=1,..,n-1$.

Eine Schaltfolge schreiben wir suggestiv auch in der Form

$$M_1[t_{k_1}{:}v_{k_1}{>}M_2[t_{k_2}{:}v_{k_2}{>}.....M_n[t_{k_n}{:}v_{k_n}{>}M'_n = M_{n+1}$$

In der Regel interessieren uns nicht alle möglichen Markierungen eines SNL-Systems, sondern nur die, die von der Anfangsmarkierung aus durch Schalten von Transitionen erreichbar sind. Hierzu führen wir die folgenden Definitionen ein:

Def. 49 (Erreichbarkeitsrelation und Erreichbarkeitsmenge)

Sei *SNL-SYS*=(*NSPEC*,*A*) ein SNL-System und M_0 die zugehörige Anfangsmarkierung.

(a) Die *direkte Erreichbarkeitsrelation* => sei auf *MARK*(*NSPEC*,*A*) wie folgt definiert:

$\forall M, M' \in MARK(NSPEC,A)$ gilt M => M' genau dann, wenn eine Transition $t \in T$ und eine Auswertung $v \in VAL_t(NSPEC,A)$ existiert, so daß $(M,t{:}v,M')$ ein Schaltvorgang ist.

(b) Als *Erreichbarkeitsrelation* $\stackrel{*}{\Rightarrow}$ bezeichnen wir den transitiven und reflexiven Abschluß von =>, d.h. $\forall M, M' \in MARK(NSPEC,A)$ gilt $M \stackrel{*}{\Rightarrow} M'$ genau dann, wenn M=M' ist oder eine Schaltfolge $(M_i,t_i{:}v_i,M_i')_{i=1,..,n}$ existiert, für die $M=M$ und $M'_n=M'$ ist.

(c) $[M> := \{ M' \in MARK(NSPEC,A) \mid M \stackrel{*}{\Rightarrow} M' \}$

(d) $[M_0>$ heißt *Erreichbarkeitsmenge* des SNL-Systems *SNL-SYS*.

Die für uns relevanten Zustände eines SNL-Systems werden also durch die Erreichbarkeitsmenge $[M_0>$ beschrieben. Entsprechend interessieren uns nur Schaltvorgänge, die von Markierungen aus $[M_0>$ ausgehen können.

Def. 50

Sei *SNL-SYS*=(*NSPEC*,*A*) ein SNL-System und M_0 die zugehörige Anfangsmarkierung.

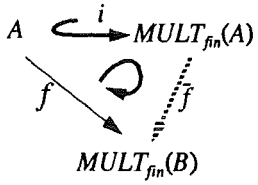
$TOCC(NSPEC,A) := \{ (M,t{:}v,M') \mid (M,t{:}v,M')$ ist ein Schaltvorgang und $M \in [M_0> \}$.

$[M_0>$ und *TOCC*(*NSPEC*,*A*) bilden die Knoten- und Kantenmengen eines Graphen, den wir *Erreichbarkeitsgraph* nennen wollen. Der Erreichbarkeitsgraph beschreibt die dynamische Struktur eines SNL-Systems bzgl. dem Schalten von Transitionen vollständig und wird in [Süß92] eingehend untersucht.

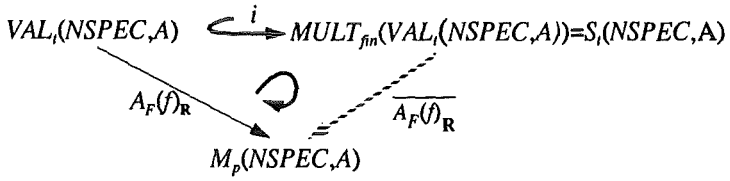
Wir haben bisher nur Zustandsübergänge in SNL-Systemen betrachtet, die als atomare Operationen nur das Schalten jeweils einer Transition in einem Mode zulassen. Wir haben aber schon mehrfach im Text zum Ausdruck gebracht, daß wir allgemeiner auch das nebenläufige Schalten einer Transition bzgl. mehrerer Schaltmodi und das nebenläufige Schalten mehrerer aktivierter Transitionen zulassen wollen. Hierzu wollen wir uns zunächst einmal ansehen, wie wir das nebenläufige Schalten einer Transition $t \in T$ in mehreren Modi beschreiben können. Hierzu müssen wir nicht mehr eine Auswertung $v \in VAL_t(NSPEC,A)$, sondern Multimengen von Auswertungen $\mu = \sum_{i=1}^{k} d_i \cdot v_i \in MULT^+_{fin}(VAL_t(NSPEC,A)) = S^+_t(NSPEC,A)$ als Schaltmodi betrachten.

Wir hatten aber eine Kantenbeschriftung $A_F(f)$ in der Realisation als eine Abbildung $A_F(f)_R{:}VAL_t(NSPEC,A) \to M_p(NSPEC,A)$ interpretiert, so daß $A_F(f)_R$ nicht so ohne weiteres auf eine Multisumme $\mu \in S^+_t(NSPEC,A)$ anwendbar ist. Zum Glück können wir für zwei Mengen A und B jede Abbildung

$f:A\to MULT_{fin}(B)$ auf genau eine eindeutige Art und Weise zu einem $\mathbb{Z}$-Modulhomomorphismus $\bar{f}:MULT_{fin}(A)\to MULT_{fin}(B)$ fortsetzen, so daß das folgende Diagramm kommutativ ist:



Hierzu definieren wir $\bar{f}(\sum_{k=1}^{n} d_k \cdot a_k) = \sum_{k=1}^{n} d_k \cdot f(a_k)$ für alle $n\in\mathbb{N}$, $n\geq 1$, $d_k\in\mathbb{Z}$ und $a_k\in A$ für $1\leq k\leq n$ und $\bar{f}(0)=0$. Es gibt damit zu $A_F(f)_R:VAL_t(NSPEC,A)\to M_p(NSPEC,A)$ genau einen $\mathbb{Z}$-Modulhomomorphismus $\overline{A_F(f)_R}:MULT^{+}_{fin}(VAL_t(NSPEC,A))=S^{\dagger}_{t}(NSPEC,A)\to M_p(NSPEC,A)$, der das Diagramm



kommutativ macht. Wenn wir Multimengen von Auswertungen der freien Variablen einer Transition betrachten, verbinden wir im folgenden mit der Realisation einer Kantenbeschriftung $A_F(f)$ stets diesen eindeutig bestimmten Modulhomomorphismus $\overline{A_F(f)_R}$. Analog zu Δt definieren wir nun $\Theta_t=(\Theta_{p,t})_{p\in P}:S^{\dagger}_{t}(NSPEC,A)\to M(NSPEC,A)$ durch

$$\Theta_{p,t}=\begin{cases} 0\in M_p(NSPEC,A) & \text{falls } (t,p) \text{ und } (p,t)\notin F \\ -\overline{A_F(f)_R} & \text{falls } f=(p,t)\in F \text{ und } (t,p)\notin F \\ \overline{A_F(f)_R} & \text{falls } (p,t)\notin F \text{ und } f=(t,p)\in F \\ -\overline{A_F((p,t))_R}+\overline{A_F((t,p))_R} & \text{falls } (p,t)\in F \text{ und } (t,p)\in F \end{cases}$$

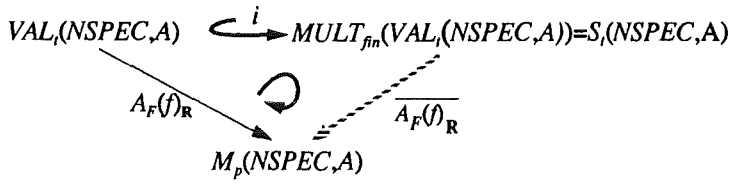
Wenn die Transition dann bzgl. der Multimenge $\mu_t\in S^{\dagger}_{t}(NSPEC,A)$ von Auswertungen ihrer freien Variablen schaltet, wird die Folgemarkierung analog zur Gleichung $M'=M+\Delta t(v)$ durch die Gleichung $M'=M+\Theta_t(\mu_t)$ beschrieben. Wenn wir nun noch das nebenläufige Schalten von Transitionen ins Kalkül ziehen, dann wird ein möglicher Zustandsübergang nicht mehr durch eine Multimenge, sondern durch eine Familie von Multimengen $\mu=(\mu_t)_{t\in T}$ mit $\mu_t\in S^{\dagger}_{t}(NSPEC,A)\ \forall t\in T$, d.h. durch einen T-Vektor $\mu\in S(NSPEC,A)$, beschrieben. Die Folgemarkierung ergibt sich in diesem allgemeinen Fall durch die Gleichung $M' = M + \sum_{t\in T}\Theta_t(\mu_t)$. Wenn wir nun $(\Theta_{p,t})_{p\in P,t\in T}$ als eine Matrix $\Theta\in[S_T(NSPEC,A),M_P(NSPEC,A)]=[S(NSPEC,A),M(NSPEC,A)]$ auffassen, da die $\Theta_{p,t}$ ja alle $\mathbb{Z}$-Modulhomomorphismen sind, entspricht der Summe $(\sum_{t\in T}\Theta_t(\mu_t) = \sum_{t\in T}(\Theta_{p,t})_{p\in P}(\mu_t))$ gerade die Anwendung der Matrix Θ auf den T-Vektor μ und wir können schreiben: $M'=M+\Theta\cdot\mu$. Wir führen daher den folgenden Begriff ein.

Def. 51 (Inzidenzmatrix eines SNL-Systems)

Sei $SNL\text{-}SYS=(NSPEC,A)$ ein SNL-System. $\forall p\in P$ und $\forall t\in T$ sei der $\mathbb{Z}$-Modulhomomorphismus $\Theta_{p,t}:S^{\dagger}_{t}(NSPEC,A)\to M_p(NSPEC,A)$ durch

$$\Theta_{p,t}=\begin{cases} 0\in M_p(NSPEC,A) & \text{falls } (t,p) \text{ und } (p,t)\notin F \\ -\overline{A_F(f)_R} & \text{falls } f=(p,t)\in F \text{ und } (t,p)\notin F \\ \overline{A_F(f)_R} & \text{falls } (p,t)\notin F \text{ und } f=(t,p)\in F \\ -\overline{A_F((p,t))_R}+\overline{A_F((t,p))_R} & \text{falls } (p,t)\in F \text{ und } (t,p)\in F \end{cases}$$

definiert, wobei $\overline{A_F(f)_R} : S_t(NSPEC,A) \to M_p(NSPEC,A)$ für alle $f=(p,t)$ *bzw.* $f=(t,p) \in F$ der eindeutig bestimmte $\mathbb{Z}$-Modulhomomorphismus ist, der das folgende Diagramm kommutativ macht:



Die Matrix $\Theta=(\Theta_{p,t})_{p\in P,t\in T} \in [S(NSPEC,A),M(NSPEC,A)]$ heißt dann *Inzidenzmatrix des SNL-Systems SNL-SYS.*

Wenn wir im folgenden von Schaltvorgängen bzgl. T-Vektoren eines SNL-Systems sprechen, wollen wir zur Unterscheidung zum Fall des Schaltens von Transitionen die Begriffe *Schalten von Schritten* und *Schrittvorgang* verwenden. Die Definition der Schaltregel für Schritte lautet wie folgt:

Def. 52 (Aktivierbarkeit von Schritten und Schrittregel)

Sei $SNL\text{-}SYS=(NSPEC,A)$ ein SNL-System und $M=(m_p)_{p\in P} \in MARK(NSPEC,A)$ eine Markierung.

(a) Ein T-Vektor $\mu=(\mu_t)_{t\in T} \in S(NSPEC,A)$ heißt *aktivierbar unter der Markierung M*, falls die folgenden Bedingungen erfüllt sind.

(1) $\mu \in S^+(NSPEC,M)$ (Lokalitätsprinzip).

(2) $M \models_\mu NSPEC$ (μ erfüllt die Transitionsformeln).

(3) $\forall p\in P$ gilt: $\sum_{t\in T,(p,t)\in F} \overline{A_F((p,t))}_R(\mu_t) \le m_p$ (d.h. die "richtigen" Objekte sind in genügender Anzahl auf den Eingabestellen vorhanden).

(4) $M'=M+\Theta \cdot \mu$ ist eine mögliche Markierung, d.h. $M+\Theta \cdot \mu \in MARK(NSPEC,A)$.

(b) (Schrittregel)
Sei der T-Vektor $\mu \in S(NSPEC,A)$ unter M aktivierbar. Das SNL-System kann dann im Mode μ schalten, wobei der Zustand des Systems von der Markierung M in die Markierung $M'=M+\Theta \cdot \mu$ übergeht.

Der Begriff der Aktivierbarkeit von T-Vektoren ergibt sich also analog zu dem Begriff der Aktivierbarkeit einer Transition t in einem Mode v. Die Aktivierbarkeit einer Transition $\bar{t}$ in einem Mode v können wir sogar als einen Spezialfall des Begriffs der Aktivierbarkeit von T-Vektoren auffassen, wenn wir als T-Vektoren $\mu=(\mu_t)_{t\in T}$ mit $\mu_t=1`v$ falls $\bar{t}=t$, $\mu_t=0$ sonst, verwenden.

Betrachten wir einmal ein Beispiel zum Begriff der Aktivierbarkeit von T-Vektoren und zur Schrittregel.

Beispiel 43 (Aktivierbarkeit von Schritten und Schrittregel)

Wir betrachten ein SNL-System `FARBE`, dessen Spezifikationsteil eine Sorte `farbe` mit zwei Konstanten `rot` und `grün` definiert, die wir trivialerweise über eine zweielementige Menge mit den Elementen *rot* und *grün* realisiert denken. Das Netz selbst habe die Gestalt, wie sie das folgende Bild verdeutlicht.

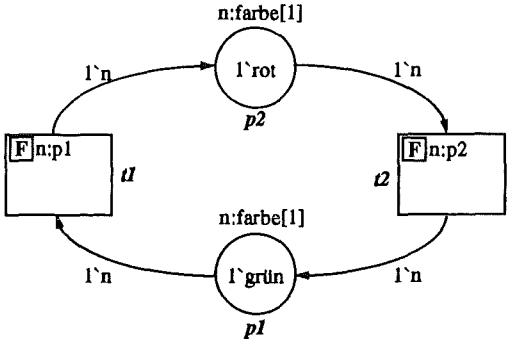


Abb. 1 SNL-System FARBE mit Anfangsmarkierung M_0

In diesem System kann keine Transition schalten, da für jede Transition die Nachbedingung nicht erfüllt ist (Die Kapazität 1 einer der Ausgabestellen wird überschritten). Der T-Vektor μ=(1`{n->grün},1`{n->rot})' ist aber aktivierbar, da beim Vertauschen der beiden Farbmarken während einer atomaren Operation die Kapazität der Stellen nicht überschritten wird. Nach dem Schalten dieses Schrittes haben wir dann das SNL-System mit der folgenden Markierung

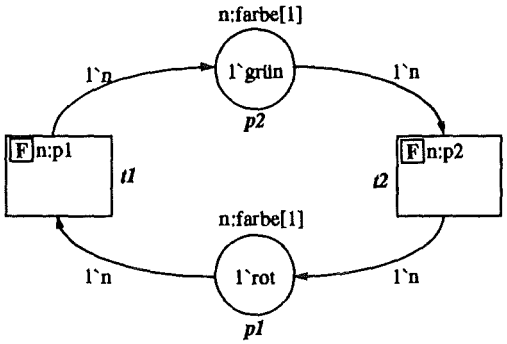


Abb. 2 SNL-System FARBE mit Markierung M_1

Wir wollen noch einige Begriffe einführen.

Def. 53 (Schrittvorgang, Schritt und Schrittfolge)

Sei *SNL-SYS*=(*NSPEC*,*A*) ein SNL-System und $M \in MARK(NSPEC,A)$ eine Markierung.

(a) Unter einem *Schrittvorgang* verstehen wir ein Tripel (M,μ,M') mit $M, M' \in MARK(NSPEC,A)$ und $\mu \in S(NSPEC,A)$, so daß μ unter M aktivierbar ist, und $M'=M+\Theta \cdot \mu$ die Folgemarkierung von M ist. μ nennen wir in diesem Fall den *Schritt*, der die Markierung M in die Markierung M' überführt.

Um suggestiv darzustellen, daß M' aus M durch Schalten des SNL-Systems im Mode μ entsteht, schreiben wir einen Schrittvorgang auch in der Form $M[\mu>M'$.

(b) Eine *Schrittfolge* ist eine Folge von Schrittvorgängen $(M_i,\mu_i,M_i')_{i=1,..,n}$ mit $M_i'=M_{i+1}$ für $i=1,..,n-1$.

Eine Schrittfolge schreiben wir suggestiv auch in der Form

$$M_1[\mu_1>M_2[\mu_2>.....M_n[\mu_n>M'_n = M_{n+1}$$

Wenn wir die Schrittsemantik eines SNL-Systems betrachten, interessieren uns in der Regel wiederum nicht alle möglichen Markierungen des SNL-Systems, sondern nur die, die von der Anfangsmarkierung aus durch Schalten von Schritten erreichbar sind. Hierzu führen wir die folgende Definition ein:

Def. 54 (Schritterreichbarkeitsrelation und Schritterreichbarkeitsmenge)

Sei *SNL-SYS*=(*NSPEC*,*A*) ein SNL-System und M_0 die zugehörige Anfangsmarkierung.

(a) Die *direkte Schritterreichbarkeitsrelation* $_S$=> sei auf $MARK(NSPEC,A)$ wie folgt definiert:

$\forall M, M' \in MARK(NSPEC,A)$ gilt $M\ {}_S{=>}\ M'$ genau dann, wenn ein T-Vektor $\mu \in S(NSPEC,A)$ existiert, so daß (M,μ,M') ein Schrittvorgang ist.

(b) Als *Schritterreichbarkeitsrelation* $^*_S$=> bezeichnen wir den transitiven und reflexiven Abschluß von $_S$=>, d.h. $\forall M, M' \in MARK(NSPEC,A)$ gilt $M\ {}^*_S{=>}\ M'$ genau dann, wenn M=M' ist oder eine Schrittfolge $(M_i,\mu_i,M_i')_{i=1,..,n}$ existiert, für die $M=M$ und $M'_n=M'$ ist.

(c) $[[M> := \{ M' \in MARK(NSPEC,A) \mid M\ {}^*_S{=>}\ M' \}$

(d) $[[M_0>$ heißt *Schritterreichbarkeitsmenge* des SNL-Systems *SNL-SYS*.

Die für uns relevanten Zustände eines SNL-Systems bzgl. der Schrittsemantik sind also die Elemente der Schritterreichbarkeitsmenge $[[M_0>$. Entsprechend interessieren uns nur Schrittvorgänge, die von Markierungen aus $[[M_0>$ ausgehen können.

Def. 55

Sei *SNL-SYS*=(*NSPEC*,*A*) ein SNL-System und M_0 die zugehörige Anfangsmarkierung.

$SOCC(NSPEC,A) := \{ (M,\mu,M') \mid (M,\mu,M')$ ist ein Schrittvorgang und $M \in [[M_0> \}$.

Die Mengen $[[M_0>$ und $SOCC(NSPEC,A)$ kann man wiederum als Knoten- und Kantenmenge eines Graphen, den wir *Schrittgraphen* nennen wollen, auffassen, der die dynamische Struktur eines SNL-Systems bzgl. der Schrittsemantik, die durch die Schrittregel gegeben ist, vollständig beschreibt. Der Schrittgraph wird ebenfalls genauer in [Süß92] beschrieben.

In SMARAGD erlauben wir es dem Modellierer, daß er SNL-Modelle sowohl mit der Schrittsemantik als auch mit der Semantik des Schaltens einzelner Transitionen untersuchen kann. Die Erreichbarkeitsanalyse innerhalb von SMARAGD, die auf den Begriffen Erreichbarkeitsgraph bzw. Schrittgraph aufbaut, ist der Gegenstand der Arbeit von W. Süß in [Süß92].

4.5 Modularisierung

Ein wesentlicher Nachteil der bislang vorgestellten Spezifikations- und Modellierungssprache SNL besteht darin, daß ein komplexes System als ein (flaches) Petri-Netz-Modell, das sämtliche Details in einer SNL-Netz-Spezifikation enthalten muß, modelliert werden muß. In der Praxis besteht aber so ein System in der Regel aus diversen Bausteinen, wobei ein Baustein durchaus auch wiederholt im Gesamtsystem auftaucht. Man wünscht sich daher für die Praxis Methoden, die es einem erlauben, gemäß der Bausteinphilosophie einzelne Komponenten eines Systems gesondert als SNL-Netz-Module (kurz Netz-Module) bereitzustellen, die man dann zu einer Gesamtspezifikation zusammensetzen kann. Dabei sollten Bausteine wiederverwendbar sein.

Jensen hat einige solcher Bausteinkonzepte für CP-Netze (Coloured Petri Nets) vorgestellt und bezeichnet die durch seine Modulkonzepte angereicherte Netzform Hierarchische CP-Netze (siehe z.B. [Huber89]). Wir werden einige von Jensen definierte Konzepte übernehmen und in einigen Punkten erweitern. Die von Jensen vorgestellten Konzepte lassen sich grob in drei Klassen einteilen: (a) Substitutionselemente, die als Platzhalter für Teilnetze stehen, (b) Aufruftransitionen, die die Netztopologie dynamisch verändern, und (c) ein vor allem graphisch zur Entflechtung von Netzen eingesetztes Hilfsmittel (sogenannte Fusionsmengen). Wir werden uns im folgenden auf die durch (a) gegebenen Modularisierungskonzepte beschränken.

Die Idee, die zur Einführung von Substitutionselementen führt, ist, ein System als eine hierarchisch angeordnete Menge von einzelnen Teilmodellen auf verschiedenen Abstraktionsstufen (die Ebenen der Hierarchie) anzusehen. Auf einer höheren Abstraktionsebene werden dabei die Details eines (nur in der Verwendung nicht aber in seinem Aufbau wichtigen) Teilsystems in der Weise abstrahiert, daß man dieses Teilsystem durch ein Substitutionselement beschreibt, wobei dieses Substitutionselement als Platzhalter für die exaktere Beschreibung des Teilsystems auf einer niedrigeren Abstraktionsebene steht.

Teilsysteme auf niedrigeren Ebenen können dabei wiederum Substitutionselemente enthalten, so daß eine hierarchische Struktur von Systemmodellen entsteht, deren unterste Ebene keine Substitutionselemente mehr und damit nur einfache Netze enthält. Die semantische Auswertung eines hierarchischen Modells erfolgt dann so, daß die Substitutionselemente nach definierten Regeln rekursiv durch die ihnen zugeordneten Teilmodelle ersetzt werden. Es sollte klar sein, daß man am Ende dieser Ersetzung dann ein flaches Netz im gewöhnlichen Sinne erhält, das man dann mit den bekannten Methoden auswerten kann.

Innerhalb der Netz-Modellierung bieten sich zwei Arten von Substitutionselementen an. Dies sind Substitutionsstellen und Substitutionstransitionen. Substitutionsstellen werden auf höherer Abstraktionsebene wie Stellen verwendet und dienen, wie wir gleich noch genauer darlegen werden, als Platzhalter für Netz-Objekte (Instanzen von Klassen im Sinne einer objektorientierten Sprache). Substitutionstransitionen werden wie Transitionen verwendet und stehen, wie wir gleich noch darlegen werden, für Netz-Module: die Teilsysteme des zu modellierenden Systems.

In den folgenden zwei Abschnitten wollen wir Netz-Objekte und ihre Instantiierung durch Substitutionsstellen sowie Netz-Module und ihre Verwendung durch Substitutionstransitionen genauer beschreiben.

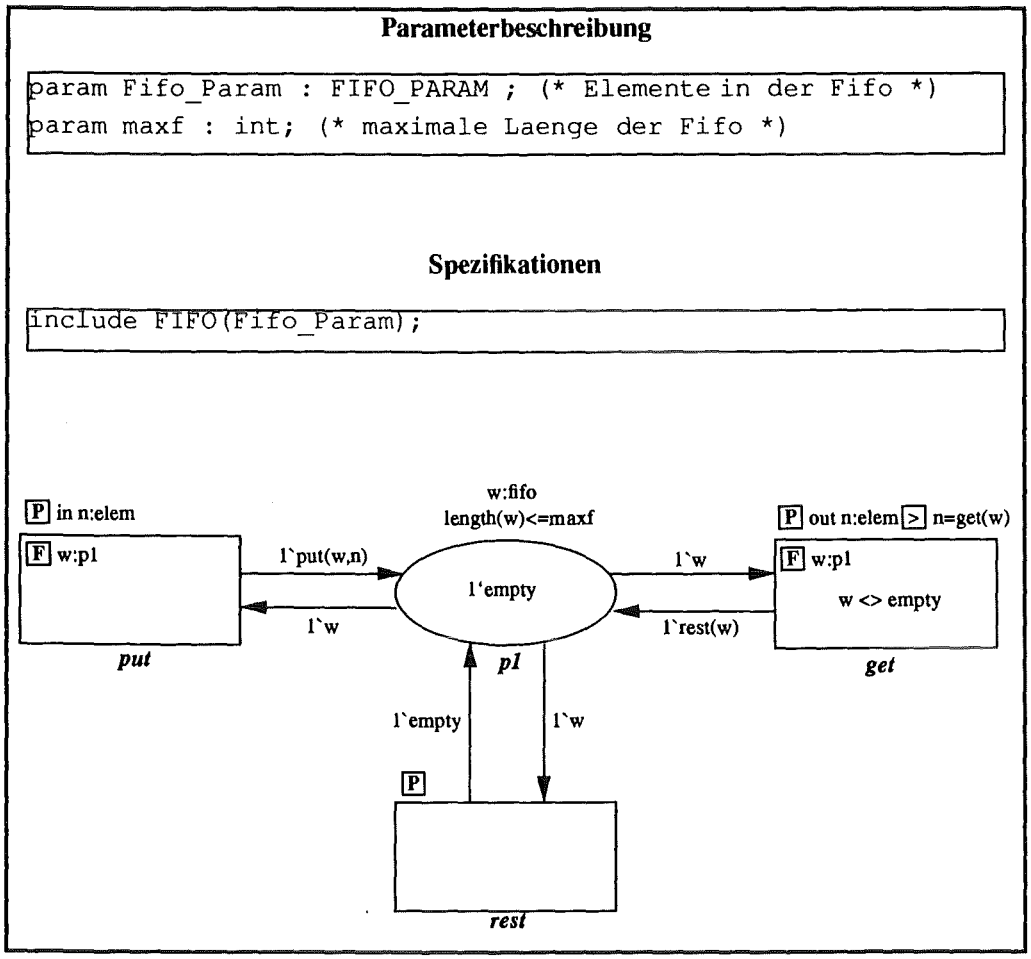
4.5.1 Netz-Klassen und Netz-Objekte

Eine SNL-Netz-Klasse (kurz Netz-Klasse) ist eine SNL-Netz-Spezifikation, in der zunächst einmal eine Menge von Transitionen, die sogenannten Port-Transitionen, ausgezeichnet sind. Jede Port-Transition stellt ein Interface bereit, das eine zugehörige Substitutionsstelle als Instantiierung eines zugehörigen Netz-Objektes zur Einbindung des Netz-Objektes in seine Umgebung nutzen kann. Weiter ist eine Netz-Klasse in der Regel eine parametrisierte SNL-Netz-Spezifikation, d.h. Teile der Signatur oder auch Markierungen von Stellen können innerhalb der Definition der Netz-Klasse durch formale Parameter beschrieben werden. Eine Instantiierung der Netz-Klasse durch eine Substitutionsstelle spezifiziert dann genau diese Parameter und legt damit eine Instanz der Netz-Klasse (ein Netz-Objekt) fest.

Betrachten wir einmal als Beispiel für eine Netz-Klasse ein Warteschlangenmodell, wie wir es schon in früheren Beispielen verwendet haben.

Beispiel 44 (Netz-Klasse FIFO)

Das folgende Bild zeigt eine mögliche Realisierung eines FIFO-Warteschlangenmodells als Netz-Klasse:



In der obigen Graphik sind alle für uns momentan wichtigen Aspekte der Netz-Klasse FIFO aufgeführt. Zunächst einmal hat `FIFO` zwei formale Parameter `Fifo_Param` und `maxf`. `Fifo_Param` ist als Typ eine Spezifikation `FIFO_PARAM` zugeordnet. `FIFO_PARAM` ist dabei als Spezifikation der Gestalt

```
specification FIFO_PARAM = spec
     sort elem;
end
```

in der globalen Deklarationsbox definiert und beschreibt die Elementsorte innerhalb der `FIFO`. `maxf` ist als Typ die Sorte `int` zugeordnet und gibt die maximale Anzahl von Elementen innerhalb der `FIFO` an. Zur Beschriftung der Netz-Klasse werden dann die Elemente der Spezifikation `FIFO(Fifo_Param)` verwendet, die sich durch Instantiierung des Funktors `FIFO` mit dem formalen Parameter `Fifo_Param` der Netz-Klasse ergibt.

Die Transitionen der Netz-Klasse sind weiter als Port-Transitionen gekennzeichnet, indem ihnen eine spezielle Beschriftung (ein kleines Quadrat mit Inschrift **P** und nachfolgender Text) zugeordnet ist. Port-Transitionen können Eingabe- und/oder Ausgabeparameter zugeordnet sein, die durch diese Beschriftung beschrieben werden. So hat die Port-Transition ***put*** einen Eingabeparameter `n`, der angibt,

welches Element in die Warteschlange einzutragen ist. Die Port-Transition ***get*** hat einen Ausgabeparameter `n`, dem beim Schalten von ***get*** das aus der Warteschlange ausgefügte Element zugeordnet wird. ***reset*** hat keinerlei Parameter. Die Art des Parameters (Ein- oder Ausgabe) wird durch ein vorangestelltes Schlüsselwort **in** oder **out** gekennzeichnet. Weiter muß der Typ des Parameters festgelegt werden.

Port-Transitionen mit Ausgabeparametern ist eine weitere Beschriftung (gekennzeichnet durch ein kleines Quadrat mit Inschrift >) zugeordnet, die die "Werte" der Ausgabeparameter festlegt. Im Fall der Port-Transition ***get*** hat diese Beschriftung die Gestalt `n=get(w)`.

Beim Schalten der Port-Transition ***put*** wird das Element der Sorte `elem`, das durch den Parameter `n` gegeben ist, in die Warteschlange auf der Stelle *p1* nach dem *first in first out* Prinzip eingetragen. Beim Schalten der Port-Transition ***get*** wird das erste Element aus der Warteschlange auf der Stelle *p1* ausgefügt und durch den Parameter `n` übergeben. Die Port-Transition ***reset*** löscht den gesamten Inhalt der Warteschlange auf *p1*.

In SNL hat die Netz-Klasse `FIFO` die folgende Gestalt:

```
netclass FIFO = class
     param        Fifo_Param : FIFO_PARAM;
                  maxf : int;
     include      FIFO(Fifo_Param);
     placedef     p1=place
                    var w:fifo;
                    eqn length(w)<=maxf;
                    marking 1`empty;
                  end;
     port         put=trans
                    in  n:elem
                    fvar w:p1;
                    arc p1 -> 1`w;
                        p1 <- 1`put(w,n);
                  end;
                  get=trans
                    out n:elem;
                    fvar w:p1;
                    arc p1 -> 1`w;
                        p1 <- 1`rest(w);
                    eqn w <> empty;
                    return n=get(w);
                  end;
                  reset=trans
                     fvar w:p1;
                     arc p1 -> 1`w;
                         p1 <- 1`empty;
                  end;
end
```

Wir wollen nun einmal die Syntax spezifizieren, mit der man Netz-Klassen in SNL definieren kann. Die prinzipielle Syntax zur Definition einer Netz-Klasse sieht folgendermaßen aus.

netclass_decl:	**netclass** *class_id* = *class*
class:	*class_id*
	class *class_body* **end**
class_id:	**id**
class_body:	{ *class_body_decls* }$_0^*$
class_body_decls:	*spec_sig_decls*
	spec_formula_decls
	net_decls
	net_param_decls
	class_port_decls

Der Rumpf einer Netz-Klassendefinition (*class_body*) besteht aus den für ein Netz (*net*) üblichen Deklarationsteilen und den zwei zusätzlichen Deklarationsteilen *net_param_decls* und *class_port_decls*, die die Parameter und Ports der Netz-Klasse beschreiben.

Der Parameterdeklarationsteil hat die folgende Gestalt:

net_param_decls:	**param** *net_param_decl* { ; *net_param_decl* }$_0^*$ [;]
net_param_decl:	*param_id* : *net_param_type*
net_param_type:	*spec*
	sort_name
	marking_type
param_id:	**id**

Der Typ eines Parameters kann eine Spezifikation (*spec*), eine Sorte (*sort_name*) oder ein Markierungstyp (*marking_type*) sein. Ein Markierungstyp gibt dabei an, welche Sorten innerhalb einer formalen Summe auftreten können, die zu so einem Markierungstyp gehören, und wie groß die maximale Anzahl von Elementen einer Sorte sein darf. Wir haben daher die folgende Syntax für einen Markierungstyp.

marking_type:	[**num**] ` *sort_name* { + [ **num** ] ` *sort_name* }$_0^*$

Dabei darf jede Sorte nur einmal innerhalb dieser formalen Summe von Sortennamen auftreten. Der optionale Faktor **num** gibt an, wieviel Individuen der nachfolgenden Sorte maximal in der formalen Summe vorkommen dürfen. Fehlt **num**, so wird als Defaultwert 1 angenommen.

Ports werden in Portdeklarationsteilen (*class_port_decls*) definiert. Diese haben die folgende Gestalt:

class_port_decls:	**port** *class_port_decl* { ; *class_port_decl* }$_0^*$ [;]
class_port_decl:	*port_id* = *class_port*
class_port:	*port_id*
	trans *class_port_body* **end**
class_port_body:	{ *class_port_body_decls* }$_0^*$
class_port_body_decls:	*trans_var_decls*
	spec_formula_decls
	arc_decls
	port_param_decls
	port_return_decls
port_id:	**id**

Ein Portdeklarationsteil beginnt mit dem Schlüsselwort **port**, dem eine Reihe von durch Semikolon getrennten Portdeklarationen folgt. Eine Portdeklaration ordnet gemäß unserer Syntax in üblicher Weise einem Portnamen eine Portdefinition (*class_port*) zu, die entweder über den Namen eines bereits definierten Ports oder über eine explizite Portdefinition innerhalb der Schlüsselworte **trans ... end** angegeben wird. Das Schlüsselwort **trans** weist dabei darauf hin, daß es sich hier um Port-Transitionen handelt. Der Rumpf einer Port-Transition besteht aus den Deklarationsteilen, die wir schon von Transitionsdefinitionen kennen, den *port_param_decls* Teilen, die die Parameter eines Ports definieren, und den *port_return_decls* Teilen, die die Werte von Ausgabeparametern definieren. Portparameterdeklarationsteile haben die folgende Syntax:

port_param_decls:	**in** *port_param_decl* { ; *port_param_decl* }$_0^*$ [;]
	out *port_param_decl* { ; *port_param_decl* }$_0^*$ [;]
port_param_decl:	*param_id* : *sort_name*

Eingabeparameter werden hinter dem Schlüsselwort **in** und Ausgabeparameter hinter dem Schlüsselwort **out** definiert. Eine Parameterdefinition (*port_param_decl*) ordnet dabei dem Parameternamen einen Sortennamen als Typ zu. Die *port_return_decls* ordnen Ausgabeparametern Terme als ihren Wert innerhalb von *port_return_decl* hinter dem Schlüsselwort **return** zu. Die genaue Syntax ist:

port_return_decls:	**return** *port_return_decl* { ; *port_return_decl* }$_0^*$ [;]
port_return_decl:	*param_name* = *term*

term ist dabei ein beliebiger Term bestehend aus den Signaturelementen der Spezifikation, die der Netz-Klasse zugeordnet sind, und den in der Transition definierten Variablen (ohne den Parameter selbst). Auf der Aufrufebene wird dann der Parameter durch den Term ersetzt.

Wir müssen noch unsere Definition der Markierung einer Stelle leicht modifizieren, da wir ja innerhalb einer Netz-Klasse erlauben wollen, daß so eine Markierung durch einen Parameter der Netz-Klasse beschrieben wird. Hierzu definieren wir die Syntax von *marking* wie folgt um:

marking: *formal_sum*
param_id

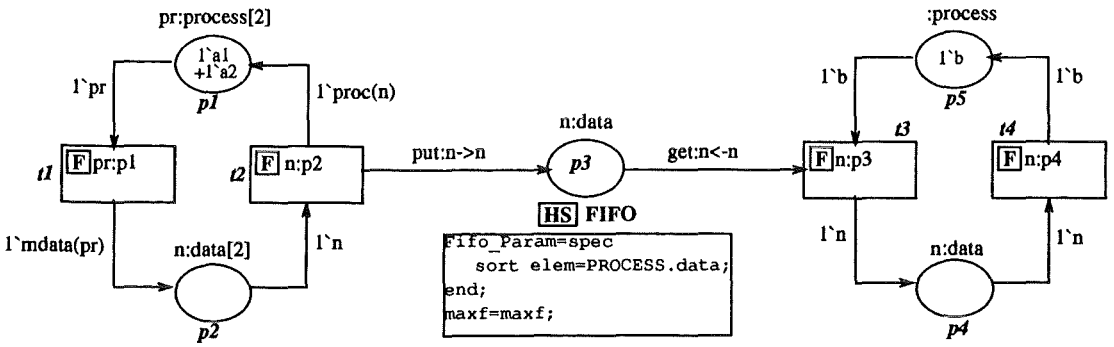
Wir können nun also anstatt einer formalen Summe auch einen Parameternamen als Markierung angeben. Natürlich muß der Parameter als Parameter der Netz-Klasse eines Typs *marking_type* definiert sein. Weiter muß der Typ des Parameters kompatibel mit dem Typ der Stelle sein.

Wie verwendet man nun eine Netz-Klasse? Zunächst einmal muß man eine Instanz einer Netz-Klasse erzeugen. Dies kann in Form der Definition einer Substitutionsstelle geschehen, die eine Instanz der betreffenden Klasse ist. Innerhalb der Substitutionsstellendefinition müssen dann die formalen Parameter der Netz-Klasse durch aktuelle Parameter spezifiziert werden.

Die Umgebung einer Stelle ist durch die Transitionen und die zugehörigen Kanten gegeben, mit denen die Stelle verbunden ist. Entsprechend hat eine Substitutionsstelle eine Umgebung bestehend aus Transitionen (den sogenannten Socket-Transitionen) und Kanten, mit denen die Substitutionsstelle verbunden ist. Diese Umgebung definiert nun in geeigneter Weise, wie das durch die Substitutionsstelle definierte Objekt in das Gesamtsystem einzufügen ist. Hierzu können die Kanten in der Umgebung der Substitutionstelle mit Aufrufen der Port-Transitionen beschriftet werden, die neben dem Namen einer Port-Transition die jeweiligen freien Parameter bestimmen, die dieser Port-Transition zugeordnet sind. Die Variablen, die dabei als Parameter einer Port-Transition angegeben werden, müssen dabei natürlich in der Umgebung der zur Kante gehörigen Transition definiert sein (als Variable einer Umgebungsstelle). Eine solche Kantenbeschriftung stellt dann eine Zuordnung einer Socket-Transition zu einer Port-Transition her, wobei weiter festgelegt ist, welche Beschriftungselemente der Socket-Transition auf Ein- bzw. Ausgabeparameter der zugehörigen Port-Transition abgebildet werden.

Beispiel 45 (Verwendung der Netz-Klasse FIFO)

Wir wollen einmal in unserem Erzeuger-Verbrauchersystem die Verwendung der Netz-Klasse `FIFO` zeigen. Dazu betrachten wir die folgende graphische Darstellung eines Netzes gemäß unserer Sprache SNL.



Die Stelle *p3* zeigt hier eine Instantiierung der Netzklasse `FIFO`, die graphisch durch ein Rechteck mit Inschrift HS (für Hierarchy Substitution), das dem Netz-Klassennamen `FIFO` vorangestellt ist, gekennzeichnet ist. Darunter sieht man ein Rechteck, das die Zuordnung der Parameter der Netz-Klasse `FIFO` zu ihren aktuellen Werten bei der Instantiierung enthält. Die Kantenbeschriftung der Kante von der Transition *t2* zur Substitutionsstelle *t3* besagt, daß der Wert der Variablen n der Transition *t3* dem Eingabeparameter n der Porttransition *put* zugeordnet wird. Die Kantenbeschriftung der Kante von der Substitutionsstelle *p3* zur Transition *t4* besagt, daß der Wert des Ausgabeparameters n der Porttransition *get* der Variablen n der Transition *t4* zugeordnet wird.

In unserer Spezifikationssprache SNL hat das obige Netz die folgende Gestalt:

```
petrinet ERZEUGER-VERBRAUCHER=net
      include     PROCESS;
      const       maxf : int;
      placedef    p1=place
                      var pr:process[2];
                      marking 1`a1+1`a2;
                  end;
                  p2=place
                      var n:data[2];
                  end;
                  p3=class FIFO with
                      param Fifo_Param=spec
                                  sort elem=PROCESS.data;
                            end;
                      param maxf=maxf;
                      var n:data;
                  end;
                  p4=place
                      var n:data;
                  end;
                  p5=place
                      var :process;
                      marking 1`b;
                  end;
      transition  t1=trans
                      fvar pr:p1;
                      arc p1 -> 1`pr;
                          p2 <- 1`mdata(pr);
                  end;
                  t2=trans
                      fvar n:p2;
                      arc p2 -> 1`n;
                          p1 <- 1`proc(n);
                          p3<- put:n->n;
                  end;
                  t3=trans
                      fvar n:p3;
                      arc p3 -> get:n<-n;
                          p4 <- 1`n;
                          p5 -> 1`b;
                  end;
                  t4=trans
                      fvar n:p4;
                      arc p4 -> 1`n;
                          p5 <- 1`b;
                  end;
end
```

Hier haben wir die Netz-Klasse `FIFO` als Substitutionsstelle *p3* mit einem Konstrukt der Gestalt *class* `FIFO` *with .... end* instantiiert. Die *class* `FIFO` *with ... end* Klausel kann man dabei lesen als: Klasse `FIFO` mit innerhalb von *with* und *end* definierten Ersetzungen von Parametern der Klasse `FIFO`.

Bei der Instantiierung wird also der Parameter `Fifo_Param` durch die Spezifikation `spec sort elem=PROCESS.data; end` und der Parameter `maxf` durch die Konstante `maxf` konkretisiert. In der `var n:elem` Deklaration wird ausgedrückt, daß der Typ der Parameter innerhalb der Port-Transitionen der Netz-Klasse `FIFO` nach der Instantiierung `data` ist, so daß die Variablen der Socket-Transitionen, die diesen Parametern zugeordnet werden, ebenfalls vom Typ `data` sein müssen.

Durch die Kanten von *t2* nach *p3* und *p3* nach *t3* und ihre Beschriftungen ist der Socket-Transition *t2* die Port-Transition *put* und der Socket-Transition *t3* die Port-Transition *get* zugeordnet worden. Der Eingabe- bzw. Ausgabeparameter der jeweiligen Port-Transition wurde dabei jeweils durch n spezifiziert.

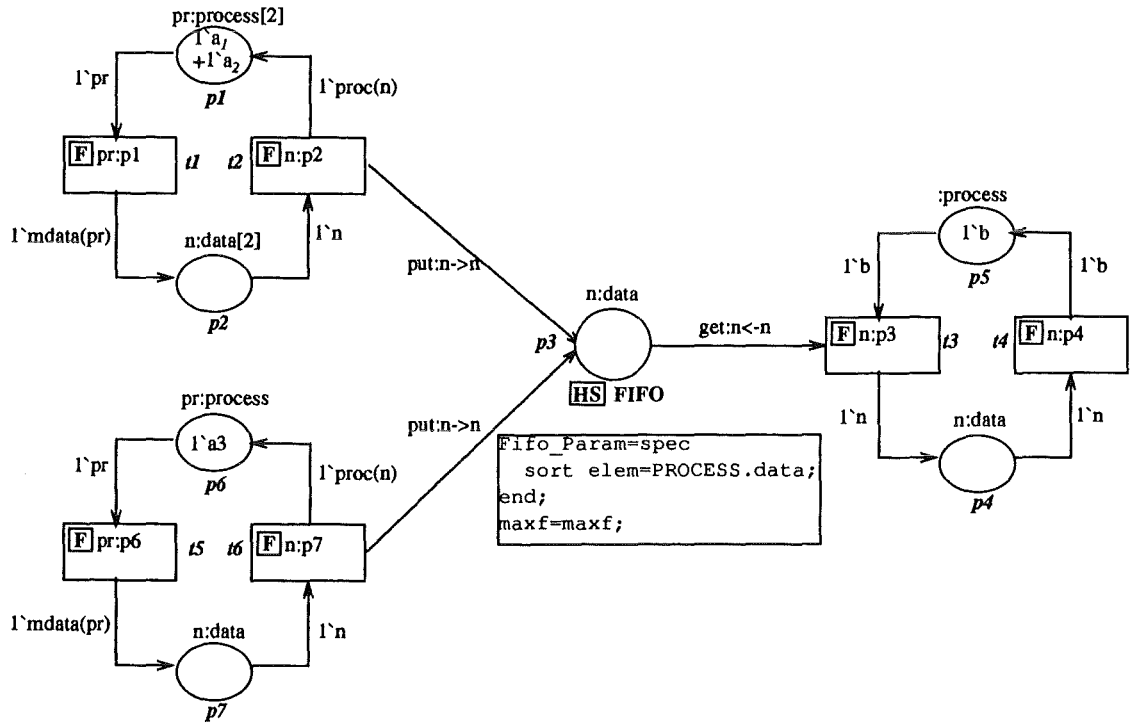
Wenn *t2* schaltet, wird offensichtlich ein Datenpaket, das durch n definiert ist, in die durch die Substitutionsstelle *p3* definierte Warteschlange eingetragen, sofern die Warteschlange nicht voll ist. Wenn *t3* schaltet, wird das erste Element aus der durch die Substitutionsstelle *p3* definierten Warteschlange ausgefügt und als Wert des Parameters n in der Umgebung von *t3* übergeben, sofern die Warteschlange nicht leer ist.

Jeder Socket-Transition ist also durch die Umgebungskanten einer Substitutionsstelle auf eindeutige Weise eine Port-Transition zugeordnet. Man beachte aber, daß die Umkehrung im allgemeinen nicht gilt. So ist in dem obigen Beispiel der Port-Transition *reset* keine Socket-Transition zugeordnet. Die Zuordnung braucht auch nicht injektiv zu sein (d.h. es kann durchaus verschiedenen Socket-Transitionen die gleiche Port-Transition zugeordnet werden).

Das folgende Bild zeigt ein Beispiel, in dem zwei Socket-Transitionen die gleiche Port-Transition benützen.

Beispiel 46 (Verdoppelung des Erzeugerzyklus)

Wir wollen einmal in unserem vorhergehenden Beispiel den Erzeugerzyklus verdoppeln. Wir erhalten dann das folgende Bild:



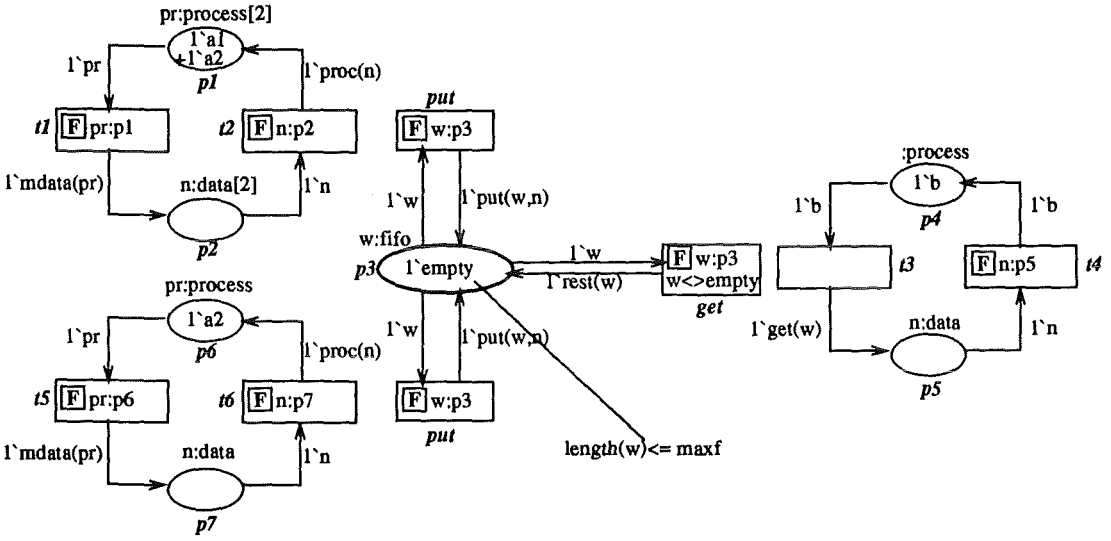
In diesem Bild sind sowohl die Socket-Transition *t2* als auch die Socket-Transition *t6* mit der Port-Transition *put* unserer Warteschlange verbunden, so daß beide Erzeugerzyklen ihre Daten jeweils in die Warteschlange, die durch *p3* gegeben ist, einfügen.

Die Semantik ist dabei, wie wir gleich noch näher erläutern werden, die, daß jede weitere Bindung einer Socket-Transition an eine bereits gebundene Port-Transition zu einer Duplizierung der Port-Transition mit den sie umgebenden Kanten führt. Sind also z.B. n Socket-Transitionen an die Port-Transition *put* gebunden, so gibt es n Instanzen der Port-Transition *put* in dem Netz, das zu der Instantiierung von FIFO durch *p3* gehört.

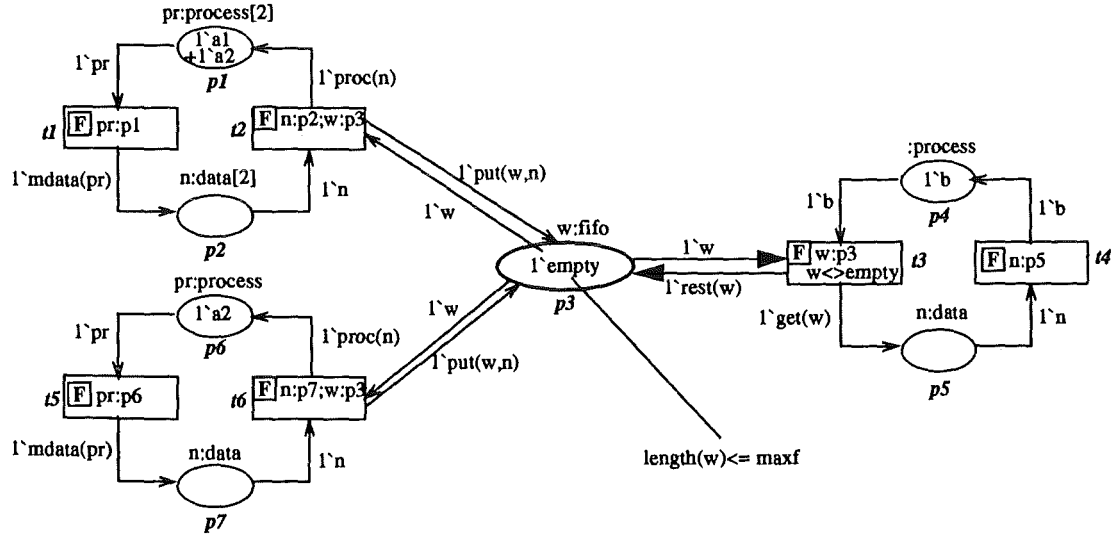
Die Semantik einer Substitutionsstelle ist so definiert, daß man die Substitutionsstelle mit ihren Kanten zunächst entfernt und dann das Netz der zugehörigen Netz-Klasse einsetzt, wobei formale Parameter durch aktuelle Parameter zu ersetzen sind, und die Port-Transitionen sooft zu duplizieren sind, wie Referenzen auf sie durch zugehörige Socket-Transitionen gegeben sind. Port-Transitionen, die nicht durch Socket-Transitionen referenziert

werden, sind mit ihren Kanten zu entfernen. Bei den Port-Transitionen werden die Eingabeparameter durch die jeweiligen Aufrufparameter der zugehörigen Socket-Transition ersetzt. In der Umgebung der Socket-Transitionen werden die Ausgabeparameter durch die in der Netz-Klasse definierten Werte ersetzt. In einem zweiten Schritt werden dann korrespondierende Socket- und Port-Transitionen aufeinandergeklebt, wobei eventuell auf einer Socket- und zugehörigen Port-Transition gegebene Formeln durch ein logisches Und verknüpft werden.

Das folgende Bild zeigt den Substitutionsschritt an unserem vorhergehenden Beispiel. In Bild (a) wurde zunächst die Substitutionsstelle und ihre umliegenden Kanten durch das zur Netz-Klasse gehörende Netz ersetzt und alle Parametertransformationen durchgeführt. Bild (b) zeigt dann das Netz nach dem Zusammenkleben korrespondierender Socket- und Port-Transitionen.



(a) Netz nach Ersetzen der Substitutionsstelle und Umsetzen der Parameter



(b) Netz nach dem Zusammenkleben der Socket- und Port-Transitionen

Wir wollen nun einmal die Aufrufsyntax von Netz-Klassen in SNL exakt definieren. Dazu müssen wir die Form einer Stellendeklaration und Kantenbeschriftung verallgemeinern.

Die Syntax der Stellendeklaration erweitern wir wie folgt:

place_decl:	*place_id* = *place*
	place_id = *class_inst*
class_inst:	**class** *class_id* **with** *class_inst_body* **end**
class_inst_body:	{ *class_inst_body_decls* }$_0^*$
class_inst_body_decls:	*inst_param_decls*
	inst_port_param_decls
inst_param_decls:	**param** *inst_param_decl* { ; *inst_param_decl* }$_0^*$ [;]

inst_param_decl:	*param_id* = *spec*
	param_id = *term*
	param_id = *formal_sum*
inst_port_param_decls:	**param** *inst_port_param_decl* { ; *inst_port_param_decl* }$_0^*$ [;]
inst_port_param_decl:	*var_id* : *sort_name*

Alternativ erlauben wir hier also, daß einem Stellennamen die Instantiierung einer Netz-Klasse zugewiesen wird. Diese Instantiierung erfolgt innerhalb eines **class ... with ... end** Sprachkonstruktes, dessen Rumpf *inst_body* die aktuellen Werte der Netz-Klassenparameter innerhalb von *inst_param_decls* festlegt. In *inst_port_param_decls* werden außerdem die Portparameter mit ihren aktuellen Typen als Variablen auf der Aufrufebene zur Verfügung gestellt. Gemäß den möglichen Typen der formalen Parameter einer Netz-Klasse sind die aktuellen Parameterwerte Spezifikationen, Terme einer gewissen Sorte oder formale Summen eines gewissen Markierungstyps.

Als Beschriftung von Kanten erlauben wir alternativ zu formalen Summen nun auch noch Konstrukte, die im Fall einer Kante zwischen einer Socket-Transition und einer Substitutionsstelle die zur Socket-Transition gehörende Port-Transition und die Instantiierung der Portparameter bestimmt. Die Syntax ist:

arg_decl:	*place_id* -> *arc_inscr*
	place_id <- *arc_inscr*
arc_inscr:	*formal_sum*
	port_socket_assignment
port_socket_assignment:	*port_id* : [*port_param_ass* { , *port_param_ass* }$_0^*$]
port_param_ass:	*term* -> *param_id*
	var_id <- *param_id*

Das neue Konstrukt ist hier der *port_socket_assignment* Teil. Dieser ordnet zunächst einer Socket-Transition eine Port-Transition (gegeben durch *port_id*) zu. Hinter dem Doppelpunkt werden dann die Zuordnungen der Socket parameter zu den Portparametern angegeben. Diese geschieht in der Form *term* -> *param_id*, wenn es sich um einen Eingabeparameter einer Port-Transition handelt. Im Fall eines Ausgabeparameters einer Port-Transition wird die Form *var_id* <- *param_id* verwendet. Die Typen der Größen auf beiden Seiten der Pfeile müssen dabei zusammenpassen. Im Fall, das eine Port-Transition keine Parameter hat, kann der Zuordnungsteil hinter dem Doppelpunkt auch fehlen.

In Bildern kennzeichnen wir Socket-Port-Zuordnungen, die sowohl Eingabe- als auch Ausgabeparameter besitzen durch einen Doppelpfeil (<->), Socket-Port-Zuordnungen, die nur Eingabeparameter einer Port-Transition spezifizieren durch einen Pfeil ->, und Socket-Port-Zuordnungen, die nur Ausgabeparameter einer Port-Transition besitzen, spezifizieren durch einen Pfeil <-.

Man beachte weiter, daß man einer Socket-Transition höchstens eine Port-Transition zuordnen darf (genau eine Kante ist hier erlaubt), denn es macht keinen Sinn, mehrere verschiedene Port-Transitionen mit einer Socket-Transition zu verbinden, denn dadurch würde man über die verschiedenen Port-Transitionen bei Ausführung der zugehörigen Socket-Transition eventuell auf gemeinsamen Stellen gleichzeitig nicht zueinander konsistente Markenbewegungen durchführen.

4.5.2 Netz-Module

Ein Netz-Modul ist wie eine Netz-Klasse eine parametrisierte Netz-Spezifikation, die aber im Gegensatz zu einer Netz-Klasse stellenberandet ist, d.h. die Ports eines Moduls sind Stellen und nicht Transitionen. Man kann sich dies so vorstellen, daß ein Netz-Modul über definierte Eingabekanten, die als Kanten zu den Eingabeportstellen definiert sind, Multisummen von Individuen als Eingabe holt, diese bearbeitet und dann als Ausgabe wiederum Multisummen von Individuen auf den Ausgabestellen abliefert. Bei der Verwendung eines Moduls werden vom Anwender Socket-Stellen auf korrespondierende Port-Stellen des Moduls abgebildet. Ein Netz-Modul kommuniziert dabei mit seiner Umgebung nur über diese Socket-Port-Stellenbeziehung, so daß die Wirkung eines Netz-Moduls nur durch die Zuordnung, die die Ausgabemultimengen in Abhängigkeit von den Eingabemultimengen eindeutig festlegt, beschrieben wird. Die internen Details werden bei der Benutzung für einen Anwender vollständig verdeckt.

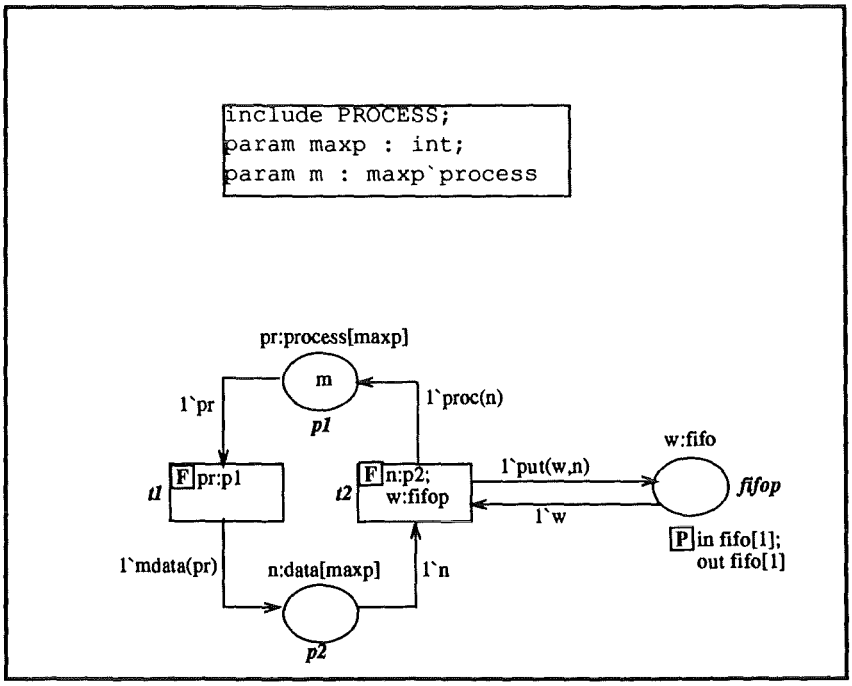
Wir wollen uns einmal an einem Beispiel einen typischen Netz-Modul ansehen, der einen Erzeuger in unserem Erzeuger-Verbraucher-System beschreibt.

Beispiel 47 (Erzeuger-Netz-Modul)

In unserer Netz-Sprache könnte ein typischer Erzeuger-Modul wie folgt aussehen:

```
netmodule ERZEUGER = module
      include     PROCESS;
      param       maxp : int;
                  m : maxp`process;
      include     FIFO(spec sort elem=PROCESS.data end);
      placedef    p1=place
                    var pr:process[maxp];
                    marking m;
                  end;
                  p2=place
                    var n:data[maxp];
                  end;
      port        fifop=place
                    var w:fifo;
                    out fifo[1];
                    in  fifo[1];
                  end;
      transition  t1=trans
                    fvar pr:p1;
                    arc p1 -> 1`pr;
                        p2 <- 1`mdata(pr);
                  end;
                  t2=trans
                    fvar n:p2 w:p3;
                    arc  p1 <- 1`proc(n);
                         p2 -> 1`n;
                         fifop <- 1`put(w,n);
                         fifop -> 1`w
                  end;
end
```

Wir wollen uns das ganze einmal in der folgenden bildlichen Darstellung verdeutlichen:



Der Netz-Modul `ERZEUGER` hat zwei Parameter. Der erste `maxp` bestimmt die Kapazität der Stellen *p1* und *p2*, die Prozesse und Daten aufnehmen. Der zweite Parameter `m` gibt die Anfangsmarkierung der Stelle *p1* an und bestimmt damit die Prozesse im Erzeuger-Modul. Weiter besitzt der Erzeuger-Modul eine Port-Stelle *fifop*. Diese stellt als Eingabestelle der Transition *t2* eine Marke vom Typ `fifo` bereit und bekommt als Ausgabestelle von *t2* eine Marke vom Typ `fifo` zurück, wobei das Datenelement, das durch die Variable `n` gegeben ist, an die Warteschlange angefügt wurde. Die Schnittstellenbeschreibung der Port-Stelle *fifo* muß angeben, von welchem Typ die Marken auf der Port-Stelle sind (also `fifo`) und wieviel Marken von so einem Typ maximal auf die Port-Stelle gelegt werden dürfen. Dies geschieht im Bild hinter dem Quadrat mit der Inschrift **P** in der Form
`in fifo[1]; out fifo[1].`

Wir wollen nun einmal die genaue Syntax für die Definition eines Netz-Moduls in SNL beschreiben. Analog zur Definition einer Netz-Klasse haben wir zunächst:

net_module:	**netmodule** *module_id* = *module*
module:	*module_id*
	module *module_body* **end**
module_id:	**id**
module_body:	{ *module_body_decls* }$_0^*$
module_body_decls:	*spec_sig_decls*
	spec_formula_decls
	net_decls
	net_param_decls
	module_port_decls

Ein Parameterdeklarationsteil (*net_param_decls*) hat die gleiche Syntax wie bei Netz-Klassen. Ein Portdeklarationsteil sieht zunächst ebenfalls wie bei einer Netz-Klasse aus:

module_port_decls:	**port** *module_port_decl* { *module_port_decl* }$_0^*$ [;]
module_port_decl:	*port_id* = *module_port*

Die Modul-Portdefinition unterscheidet sich aber von der Klassen-Portdefinition, da Ports von Modulen ja durch Stellen beschrieben werden.

module_port:	*port_id*
	place *module_port_body* **end**
module_port_body:	{ *module_port_body_decls* }$_0^*$
module_port_body_decls:	*sort_var_cap_decls*
	spec_formula_decls
	marking_decl
	port_interface_decls

Die *sort_var_cap_decls*, *spec_formula_decls* und *marking_decl* Teile beschreiben die Elemente einer Port-Stellenbeschriftung, die wir schon von einfachen Stellen kennen. Die *port_interface_decls* beschreiben die von außen sichtbaren Schnittstellen eines Netz-Modul-Ports. In ihnen muß festgelegt werden, welche Marken von außen auf den Port gelegt (im Fall eines Eingabeports) oder vom Port abgezogen (im Fall eines Ausgabeports) werden können. Dabei ist gegebenenfalls die maximale Anzahl der Marken von einem Typ anzugeben, die gleichzeitig auf den Port gelegt oder vom Port abgezogen werden können. Die *port_interface_decls* haben daher die Gestalt:

port_interface_decls:	**in** *port_interface_decl* { *_port_interface_decl* }$_0^*$ [;]
	out *port_interface_decl* { *port_interface_decl* }$_0^*$ [;]
port_interface_decl:	*sort_name* [[*token_num*]]
token_num:	**num**

Innerhalb einer *port_interface_decl* wird also eine Sorte und optional eine zugehörige Anzahl von Token dieser Sorte definiert, die als Eingabe (Schlüsselwort **in**) oder Ausgabe (Schlüsselwort **out**) der Port-Stelle nach außen bekannt gegeben wird. Eine Stelle kann dabei sowohl als Eingabe- und Ausgabestelle definiert werden.

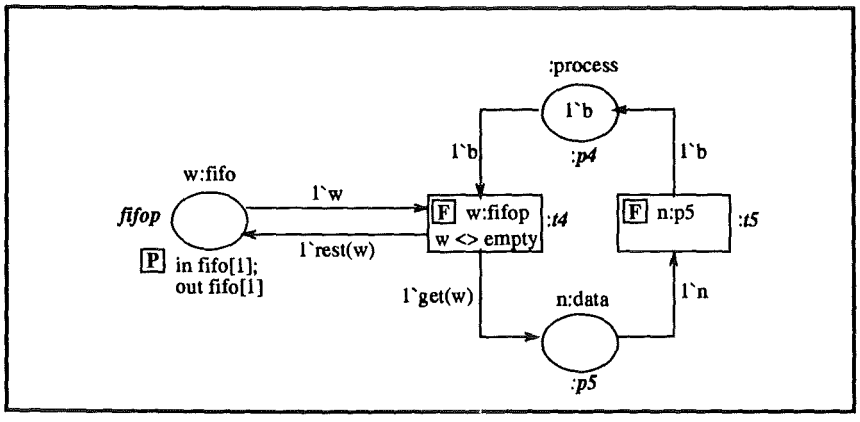
Wir wollen uns als ein weiteres Beispiel für einen Netz-Modul noch den Verbraucherteil unseres Erzeuger-Verbraucher-Systems ansehen.

Beispiel 48 (Verbraucher-Modul)

In unserer Netz-Sprache SNL würde der zu unserem Erzeuger-Verbraucher-Modell passende Verbraucher-Modul wie folgt aussehen.

```
netmodule VERBRAUCHER = module
    include     PROCESS;
                FIFO(spec sort elem=PROCESS.data end);
    placedef    p4=place
                   var :process;
                   marking 1`b; end;
                p5=place
                   var n:data; end;
    port        fifop=place (
                   var w:fifo;
                   in fifo[1];
                   out fifo[1]; end;
    transition  t4=trans
                   fvar w:fifop;
                   arc p4 -> 1`b;
                       p5 <- 1`get(w);
                       fifop <- 1`rest(w);
                       fifop -> 1`w; end;
                t5=trans
                   fvar n:p5;
                   arc p4 <- 1`b;
                       p5 -> 1`n; end;
end
```

Das folgende Bild verdeutlicht den Verbraucher-Modul. Der Verbraucher-Modul hat keine Parameter, aber ebenfalls eine Port-Stelle *fifop* wie der Erzeuger-Modul. Die Port-Spezifikation drückt aus, daß der Erzeuger-Modul am Port eine Marke vom Typ `fifo` entgegennimmt und wieder zurückgibt. Die Semantik ist dabei so, daß der Erzeuger das erste Element aus der Warteschlange entfernt und den Rest der Warteschlange dann zurückgibt.



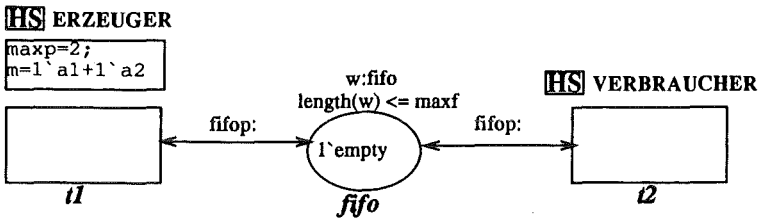
Wie sieht nun die Verwendung von Netz-Modulen aus? Wir wollen dies einmal an der Spezifikation unseres bekannten Erzeuger-Verbraucher-Systems zeigen, die nun die Netz-Module ERZEUGER und VERBRAUCHER verwendet.

Beispiel 49 (Erzeuger-Verbraucher Modell unter Verwendung von Modulen)

In unserer Netz-Sprache SNL läßt sich das Erzeuger-Verbraucher System nun wie folgt beschreiben:

```
net ERZEUGER-VERBRAUCHER = net
     include     PROCESS;
                 FIFO(spec sort elem=PROCESS.data end);
      const:     maxf : int;
     placedef    fifo=place
                     var w:fifo;
                     marking 1`empty;
                     eqn length(w) <= maxf;
                 end;
     transition  t1=module ERZEUGER with
                     param maxp=2;
                           m=1`a1+1`a2;
                     arc fifo <-> fifop;
                 end;
                 t2=module VERBRAUCHER with
                     arc fifo <-> fifop;
                 end;
end
```

Das folgende Bild verdeutlicht die Spezifikation:



Mit der Substitutionstransition *t1* wird hier zunächst ein Erzeuger-Modul mit zwei Prozessen `a1` und `a2` initialisiert. Mit der Substitutionstransition *t2* wird ein Verbraucher-Modul initialisiert. Die beiden Module kommunizieren über eine Stelle *fifo*, die eine Warteschlange vom Typ `fifo` initialisiert durch die leere Warteschlange aufnimmt. Der Port *fifop* des Verbrauchers bzw. Erzeugers ist mit dieser Stelle verbunden. Gemäß der Spezifikation erzeugen die beiden Prozesse im Erzeuger nun zyklisch Daten, die über die Port-Stelle *fifop* in die Warteschlange auf der Stelle *fifo* nach dem *fist-in-first-out* Prinzip eingehängt werden. Der Verbraucher wartet auf Daten und fügt diese aus der Warteschlange *fifo* aus.

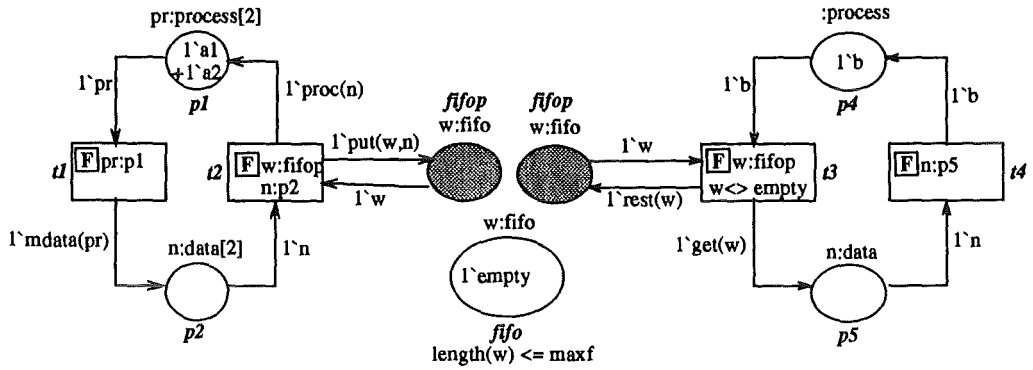
Wir müssen noch die Syntax zur Verwendung von Netz-Modulen angeben. Hierzu erweitern wir die Syntax von Transitionsangaben wie folgt:

trans_decl:	*trans_id* = *trans*
	trans_id = *module_inst*
module_inst:	**module** *module_id* **with** *module_inst_body* **end**
module_inst_body:	{ *module_inst_body_decls* }$_0^*$
module_inst_body_decls:	*inst_param_decls*
	module_arc_decls
module_arc_decls:	**arc** *module_arc_decl* { ; *module_arc_decl* }$_0^*$ [;]
module_arc_decl:	*arc_decl*

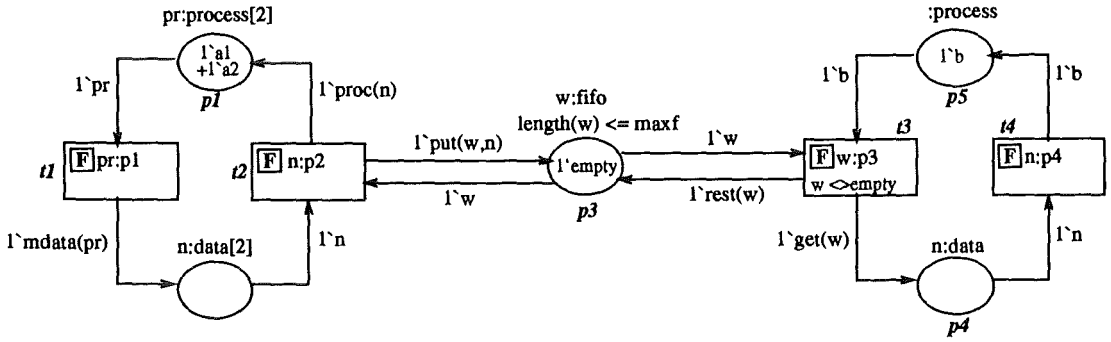
Die Instantiierung eines Moduls erfolgt also innerhalb eines **module ... with ... end** Konstruktes, in dessen Rumpf *module_inst_body* die aktuellen Parameter spezifiziert werden und die Zuordnung der Socket-Stellen zu Port-Stellen über spezielle Kantenangaben unter den *module_arc_decls* erfolgt. Die Kantenangaben erfolgen wie bei Netz-Klassen, allerdings haben wir hier stets den Fall, daß Port-Stellen keine Parameter haben und damit die Liste der Parameterzuordnungen in der Kantenbeschriftung leer ist.

Wir wollen jetzt einmal die Semantik des Modulkonzeptes beschreiben. Diese ergibt sich durch einige definierte Regeln, nach denen eine Substitutionstransition (eine Instantiierung eines Moduls) und die zugehörigen Kanten

durch das den Modul definierende Subnetz ersetzt werden, wobei die formalen Parameter entsprechend durch die aktuellen Parameter ersetzt werden. Diese Regeln lauten: Man entferne zunächst die Substitutionstransition und die zugehörigen Kanten. An ihre Stelle setzt man das zum Modul gehörende Teilnetz ein, wobei man formale Parameter durch ihre aktuellen Parameter ersetzt und für jede Socket-Stelle einer Port-Stelle eine Kopie der entsprechenden Port-Stelle bereitstellt. Im Anschluß daran verklebt man die zugehörigen Port-Stellen und Socket-Stellen. Die Menge der Variablen der so gewonnenen Stellen sind dabei die Vereinigungen der Variablen der Socket- und Port-Stellen. Die eventuell vorhandenen Gleichungen werden mit logischem Und verknüpft. Das folgende Bild zeigt den Vorgang der Ersetzung in Teil (a) und das Ergebnis des Zusammenklebens in Teil (b).



(a) Netz nach Ersetzen der Substitutionstransitionen



(b) Netz nach dem Zusammenkleben der Port- und Socket-Stellen

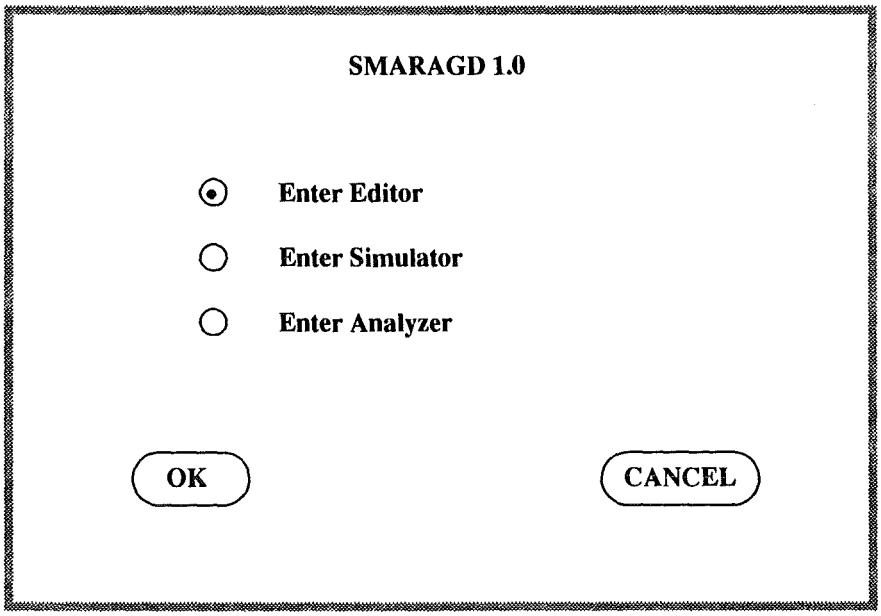
5. Ein graphischer Editor für SNL-Modelle

Der Editor in unserem Werkzeug SMARAGD basiert auf dem Design/OA* Toolkit, einem Werkzeug zur Programmierung von diagramm- und graphenorientierten Anwendungen. Als Vorlage zur Implementierung unseres Editors diente dabei der Source Code des Petri-Netz-Werkzeuges Design/CPN*, das seinerseits mit Design/OA implementiert ist. Das Benutzerinterface unseres Editors ist daher vom Aussehen und der Handhabung von Design/OA bestimmt, und unser Editor ähnelt stark dem Editor von Design/CPN (Man siehe hierzu [CPN90, OA90]).

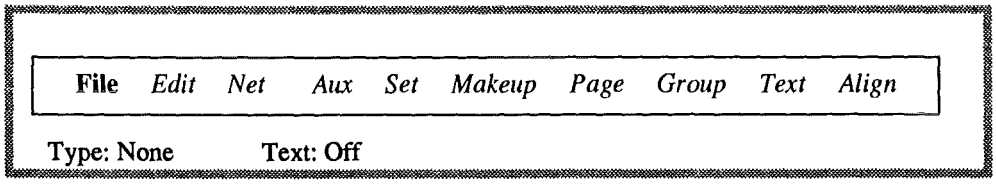
Im folgenden wollen wir das generelle Aussehen einer Design/OA Applikation (wie Design/CPN) am Beispiel unseres Editor beschreiben. Im weiteren werden wir dann auf die Elemente des graphischen Interfaces eingehen, die zur Realisierung von SNL-Modellen im Editor nötig sind.

5.1 Benutzerinterface des Editors

SMARAGD wird unter dem X-Windowsystem von einem Terminalemulator aus durch einen Aufruf des Kommandos *smaragd* gestartet. Das Werkzeug führt dann einige Initialisierungen durch und erscheint dann im X-Windowsystem mit einer zunächst generischen Menueleiste sowie einem Dialogfenster der folgenden Gestalt.



Über Radiobuttons kann der Benutzer durch diesen Dialog entscheiden, ob er den SMARAGD-Editor, den SMARAGD-Simulator oder die Analyseeinheit (Analyzer) aufrufen will. Nach dem Einstellen des gewünschten Modes wird dann der weitere Startup durch Drücken des OK Buttons eingeleitet. Wenn man den Editor-Mode gewählt hat, initialisiert dann SMARAGD seinen Editor. Danach sieht der Benutzer eine Menueleiste der folgenden Gestalt:



*Design/OA und Design/CPN sind Warenzeichen der Firma Metasoft Corporation, Cambridge

Eine Menueleiste dieser Gestalt dient als primäres Kommandointerface jeder mit Design/OA erstellten Applikation. Sie enthält die Namen einer Reihe von Pull-Down Menues, die man aktiviert, wenn man mit dem Mauszeiger auf den jeweiligen Namen weist und eine Maustaste drückt. Die einzelnen Menues und Menuepunkte beschreiben wir näher in 5.1.5.

Ein Teil der Editierkommandos, die man durch Auswahl aus einem Menue aufrufen kann, benötigt weitere Informationen vom Benutzer zur Ausführung ihrer Aufgabe oder besitzt weitere Optionen, die der Benutzer auswählen kann. Die hierzu erforderliche Benutzerabfrage wird in Design/OA-Applikationen typischer Weise durch temporär im X-Windowsystem erscheinende Dialogfenster realisiert, in denen der Anwender die erforderlichen Angaben durch spezielle Interaktionselemente leicht angeben kann. Dialogfenster und ihre Elemente beschreiben wir näher in 5.1.1.

Neben der Menueleiste und temporär auftretenden Dialogfenstern zeigt der Editor im X-Windowsystem in der Regel weitere Fenster, die Seiten des zu editierenden Netz-Diagrammes beinhalten. Auf diesen Seiten erzeugt der Anwender über die Editorkommandos in der Menueleiste und mit Hilfe der Maus graphische Objekte, die Elemente des gewünschten Netz-Modells darstellen oder auch nur zur Illustration und Dokumentation des Modells dienen. Seiten und die Design/OA-Objekte zum Aufbau von Diagrammen auf den Seiten erklären wir in 5.1.2. Die Design/OA-Objekte (in unserem Editor auch als Hilfsobjekte bezeichnet) stellen die graphischen Bauelemente der semantischen Objekte (wie Transition, Stelle, Kante und diverse Beschriftungsteile), die wir zur Realisierung eines SNL-Modells benötigen, dar. Semantische Objekte beinhalten über die rein graphische Darstellung durch geeignete Design/OA-Objekte hinaus weitere Restriktionen und Vorschriften bzgl. ihrer Verwendung und den textuellen Informationen, die sie enthalten. Die elementaren semantischen Objekte oder kurz elementaren Objekte des Editors, also die Objekte, die die elementaren semantischen Teile eines Netz-Modells darstellen, beschreiben wir in 5.1.3. Objekte, die die Hierarchisierung eines Netz-Modells unterstützen, beschreiben wir anschließend in 5.1.4.

5.1.1 Dialogfenster und ihre Elemente

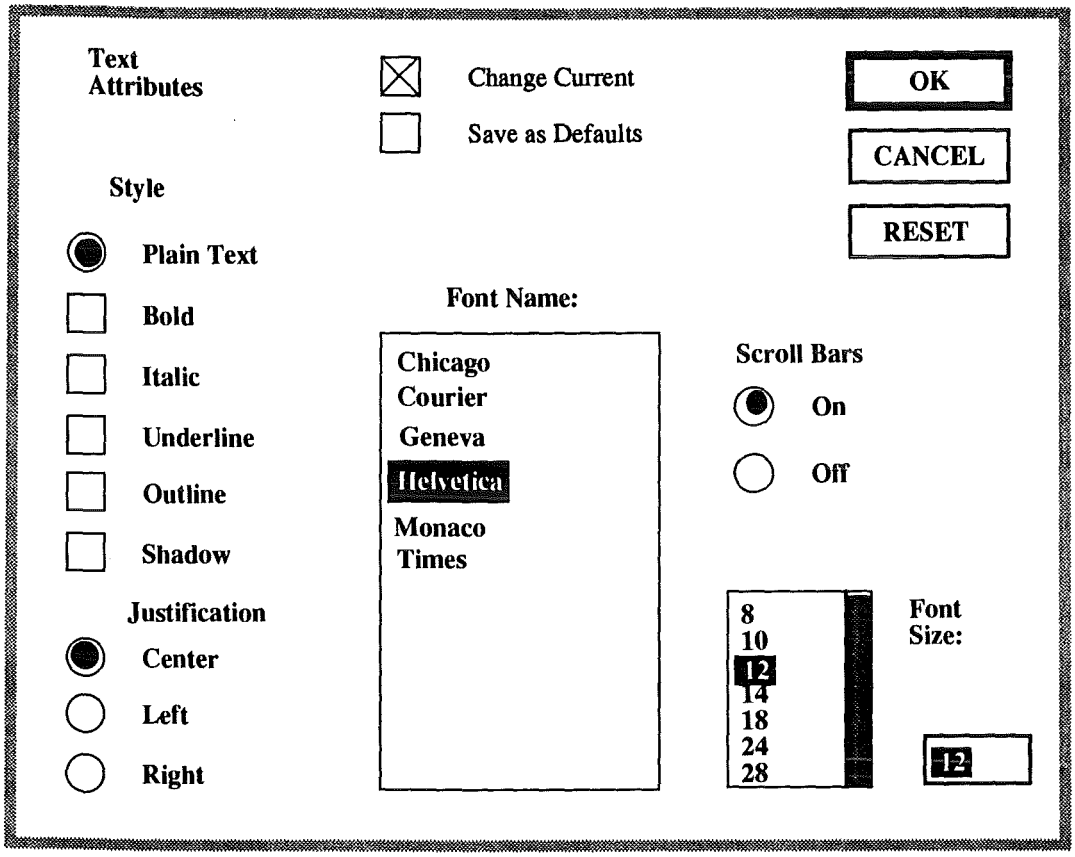
Dialogfenster dienen dem Editor als temporäres Interaktionsmedium mit dem Anwender, um Optionen und textuelle Eingaben für Kommandos abzufragen, Fehlermeldungen dem Benutzer mitzuteilen, oder erlauben dem Anwender diverse Objekt- und Systemparameter als Defaultwerte zu beeinflussen.

Ihre Bauelemente sind Druckknöpfe (Push Buttons oder kurz Buttons), um Eingaben zu bestätigen oder einen Dialog abzubrechen, Check Boxen, um nicht im Konflikt stehende Optionen zu aktivieren oder deaktivieren, Radioknöpfe (Radiobuttons), um in Konflikt zueinander stehende Optionen exklusiv auszuwählen, Nachrichtenfelder, die Mitteilungen des Systems an den Anwender darstellen, editierbare Textfelder oder Textboxen und rollbare Listenboxen.

Wir wollen die einzelnen Interaktionselemente einmal am Beispiel des Dialogfensters zur Einstellung der Textattribute von Objekten exemplarisch vorstellen.

Beispiel 50 Dialogfenster zur Einstellung von Textattributen

Das Dialogfenster zur Einstellung der Attribute von Textelementen hat die folgende Gestalt:



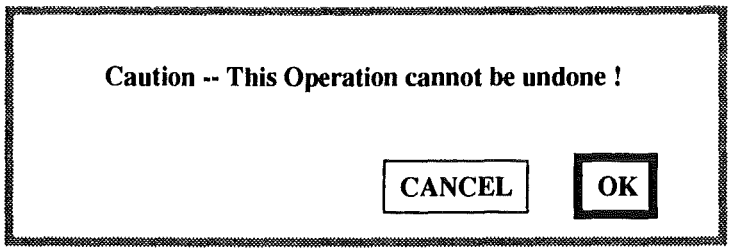
Über eine Listenbox kann man in diesem Dialog zunächst den Font für den Text auswählen. Der Helvetica Schrifttyp ist momentan selektiert. Auf der linken Seite lassen sich dann weitere Attribute der Schrift, wie Fettschrift (Bold), kursiv (Italics), unterstrichen (Underline), Outline oder schattiert (Shadow) über eine Gruppe von Check Boxen einstellen. Weiter kann hier über eine Gruppe von Radioknöpfen festgelegt werden, ob der Text zentriert, linksbündig oder rechtsbündig innerhalb eines Objektes erscheinen soll. Auf der rechten Seite kann man durch Selektion innerhalb einer Listenbox oder durch Textangabe innerhalb eines Textfeldes die Größe der Schrift einstellen. Weiter kann man über eine Gruppe von zwei Radioknöpfen festlegen, ob die Textbox Scrollbars haben soll oder nicht. In der Mitte oben kann man über Check Boxen angeben, ob die eingestellten Werte nur auf die aktuellen Objekte wirken oder als Defaultwerte für alle weiteren Textobjekte gespeichert werden sollen. Mit dem OK Button bestätigt man dann die angegebenen Einstellungen, mit RESET setzt man das Dialogfenster auf die Defaultwerte zurück und mit CANCEL beendet man den Dialog, ohne Änderungen an der aktuellen Texteinstellung durchzuführen.

Dialogfenster werden vom System nicht nur zur Einstellung von Optionen und Attributen von Kommandos und Operationen des Editors verwendet, sondern dienen auch zur Benachrichtigung des Benutzers im Fall von Fehlern, die beim Editieren oder bei kritischen Operationen, die der Benutzer durchführen will, auftreten.

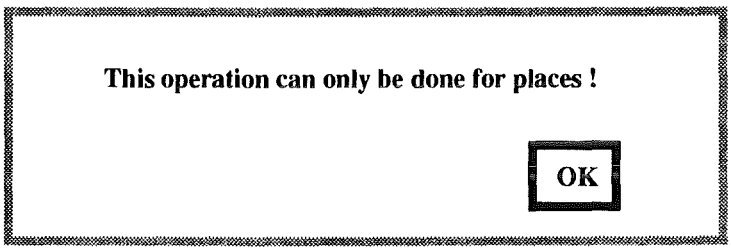
Wir wollen uns einmal die zwei Formen von Notizen ansehen, die der Editor zur Benachrichtigung von Benutzern in kritischen Situationen benützt.

Beispiel 51 (Nachrichten an den Benutzer über Alert Boxen)

Wenn ein Benutzer eine Aktion des Editors anstößt, die kritisch in dem Sinne ist, daß sie z.B. nicht mehr rückgängig gemacht werden kann, verlangt der Editor vom Benutzer eine Bestätigung der Aktion über ein Caution Dialogfenster der folgenden Gestalt:



Durch Drücken des OK Buttons kann der Benutzer dann die kritische Operation bestätigen oder durch CANCEL abbrechen.
Bei Fehlersituationen benachrichtigt das System den Benutzer über einen Dialog der folgenden Gestalt:



Hier kann der Benutzer die Kenntnisnahme der Fehlermeldung nur durch Drücken des OK Buttons bestätigen.

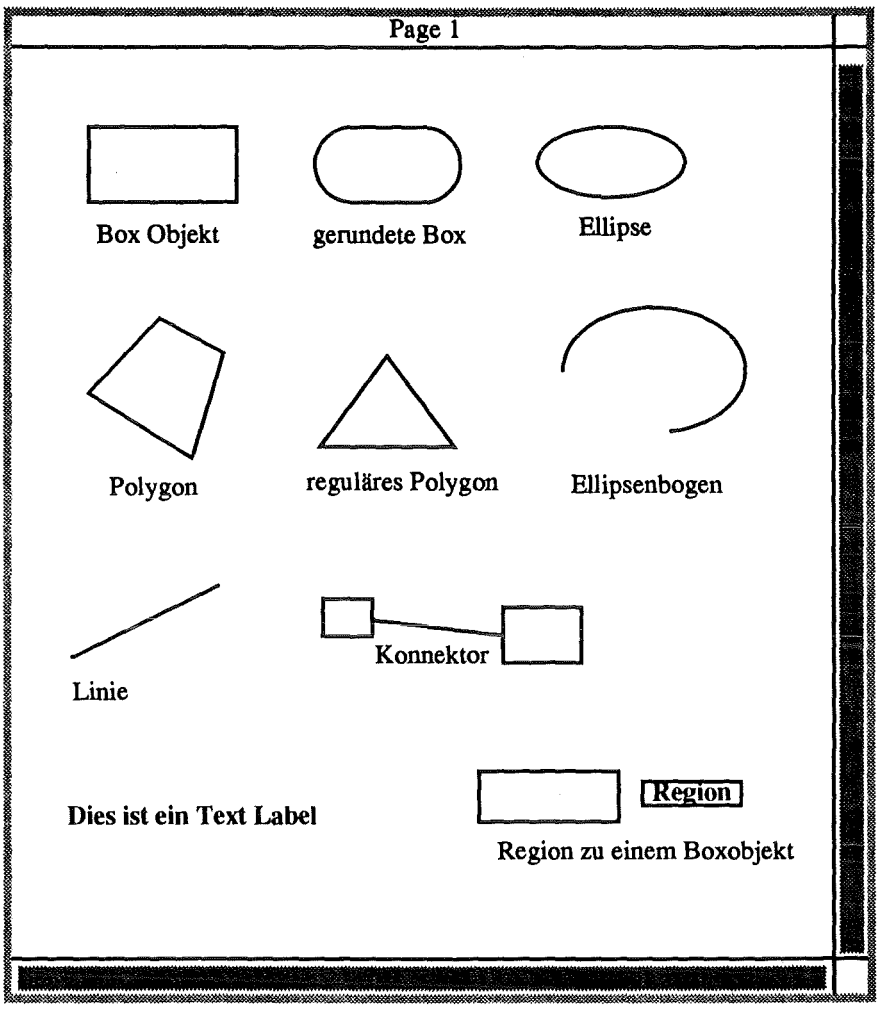
5.1.2 Seiten- und Diagramm-Objekte

Die Zeichenflächen für die Elemente eines SNL-Modells im Editor stellen die Seiten eines Diagrammes dar. Die erste Seite eines neuen Diagrammes erhält man, wenn man den Menuepunkt **New** im **File** Menue aktiviert. Weitere Diagrammseiten kann man dann mit dem Menuepunkt **New Page** des **Page** Menues einem Diagramm hinzufügen. Eine Diagrammseite ist ein zum Editor gehöriges Window im X-Windowsystem, das oben in einem Titelbalken den Seitennamen zeigt und unten und rechts durch Rollbalken begrenzt wird.

Das Window zeigt in der Regel nur einen Ausschnitt der Diagrammseite, die (abgesehen von Speicherbegrenzungen) beliebig groß sein kann. Die Scrollbars dienen dann dazu, den jeweils benötigten Ausschnitt der Diagrammseite im Window sichtbar zu machen.

Der Anwender erstellt auf den Diagrammseiten nun sein SNL-Modell, indem er verschiedene Diagrammobjekte gemäß der definierten Petri-Netz-Semantik zusammenfügt. Zusätzliche Hilfsobjekte erlauben ihm eine weitere Dokumentation und Illustration der Modelle. Sowohl die für die SNL-Modelle relevanten semantischen Objekte (wie Transition, Stelle und Kante) als auch die Hilfsobjekte basieren bzgl. ihrer graphischen Darstellung und Handhabung auf Objekten, die die Design/OA-Library vorgibt. Wir wollen im folgenden die Design/OA-Objekte, die wir als Hilfsobjekte in unserem Editor vorfinden, und ihre Handhabung vorstellen. Die semantischen Netz-Objekte, die von der Handhabung und ihrem Aussehen her von Design/OA-Objekten abgeleitet sind, werden wir in einem späteren Abschnitt beschreiben.

Design/OA-Objekte, die als Hilfsobjekte im Editor verfügbar sind, sind Boxen, gerundete Boxen, Ellipsen, Polygone, reguläre Polygone, Ellipsenbögen (Ellipsensegmente), Linien, Konnektoren, Textlabels und Regionen. Linien und Konnektoren sind dabei nichtgeschlossene Polygonzüge mit beliebig vielen Ecken, die man durch spezielle Graphikattribute auch als gerundete Kurvenzüge darstellen kann. Das folgende Bild zeigt eine Diagrammseite mit den verschiedenen Hilfsobjekten.



Wir wollen uns hier einmal zwei dieser Objekttypen (Konnektoren und Regionen), da sie von anderen graphischen Editoren her nicht so bekannt sind, genauer ansehen.

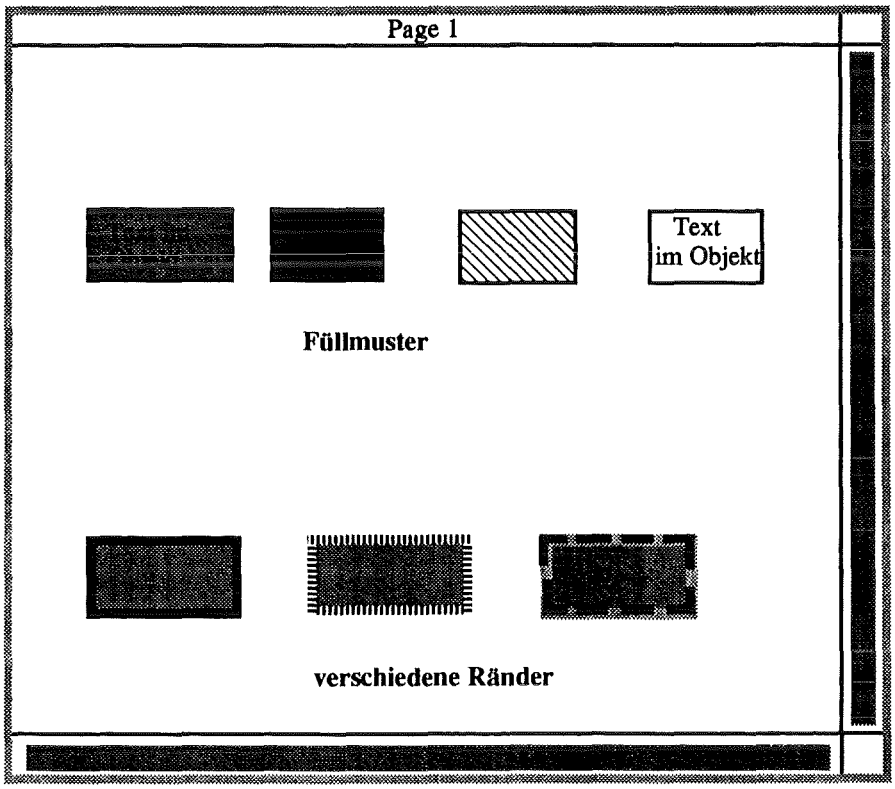
Ein Konnektor ist graphisch gesehen ein Polygonzug, der stets zwei andere Design/OA-Objekte miteinander verbindet. Bei graphischen Operationen auf durch Konnektoren verbundenen Objekten bleiben stets die Objekte automatisch verbunden, so daß z.B. nach dem Verschieben von einem der Objekte alle Konnektoren so angepaßt werden, daß wiederum die Ränder der verbundenen Objekte miteinander von Mittelpunkt zu Mittelpunkt verbunden bleiben.

Eine Region entspricht graphisch gesehen einem beliebigen anderen Design/OA-Objekt. Im Gegensatz zu einfachen Design/OA-Objekten ist ihr allerdings stets in eindeutiger Weise ein anderes Design/OA-Objekt als *Vater der Region* zugeordnet. Operationen auf dem Vaterobjekt von Regionen berücksichtigen automatisch die Regionen, so daß z.B. bei einer Verschiebung des Vaterobjektes die Regionen so mitbewegt werden, daß ihre relative Position zum Vater unverändert bleibt. Eine Operation auf einer Region selbst erfolgt dagegen stets ohne Auswirkung auf das Vaterobjekt oder eine andere Region. Aus diesem Grunde ist das Konzept der Regionen nicht vergleichbar mit dem Konzept der Gruppierung von graphischen Objekten, wie es aus anderen graphischen Editoren bekannt ist, und auch von Design/OA bereitgestellt wird. Auf Grund der Vater/Sohn-Beziehung eines Objektes zu seinen Regionen eignet sich das Konzept der Regionen hervorragend zur Angabe der semantischen Beschriftungselemente von SNL-Objekten, wie wir dies ein wenig später noch genauer sehen werden.

Geschlossenen Design/OA-Objekten, wie Box, gerundete Box, Ellipse, Polygon, Ellipsensegment und den Regionen, die durch diese Objekte dargestellt werden, kann man als ein Attribut einen Text zuordnen, der in ihrem Inneren dargestellt wird und eingegeben werden kann, wenn man einen speziellen Editiermode, den Text Mode, für das Objekt aktiviert. Ein Textfeld (Label) ist demgegenüber eine von einem anderen Objekt unabhängige Folge von Zeichen.

Eine Reihe von Textattributen erlauben es dem Anwender, den Text in verschiedenen Schriftarten (Times, Helvetica, Courier, etc), -formen (kursiv, fett, normal, unterstrichen) und Größen darzustellen. Ein anderes Textattribut kontrolliert, ob ein Text innerhalb eines anderen Objektes zentriert, links- oder rechtsbündig angeordnet wird.

Neben einem Text sind den geschlossenen Design/OA-Objekten weitere Attribute zugeordnet. Da haben wir zunächst einmal die graphischen Attribute, die z.B. festlegen, ob ein geschlossenes Objekt mit einem Muster oder schwarz ausgefüllt ist oder nicht. Die Graphikattribute eines geschlossenen Objektes legen weiter fest, mit welchem Muster der Rand des Objektes (z.B. durchgezogene, gepunktete oder gestrichelte Linie) dargestellt und wie dick der Rand gezeichnet wird. Weitere graphische Attribute bestimmen das Verhalten von Objekten, wenn sie übereinandergelegt werden. Man kann hier z.B. einstellen, ob ein Objekt durchsichtig ist oder nicht. Das folgende Bild zeigt geschlossene Objekte mit verschiedenen graphischen Attributen.



An Operationen auf Objekten stellt Design/OA und damit auch unser Editor maus- und menuegesteuerte Kommandos zum Erzeugen, Selektieren, Verschieben, Vergrößern und Verkleinern, zum Entfernen und zum Verändern der Form eines Objektes bereit. Es gibt weitere Kommandos, die die bekannten CUT, COPY und PASTE Operationen durchführen. Dabei können die meisten Menuekommandos auch über Tastenkombinationen angesprochen werden.

Eine Reihe von Objekten, auf denen gleiche Editoroperationen durchgeführt werden sollen, können zu einer Gruppe von Objekten zusammengefaßt werden, so daß die für diese Gruppe von Objekten aufgerufene Editoroperation auf sämtlichen Mitgliedern der Gruppe durchgeführt wird.

Nach dieser kurzen Übersicht über die Design/OA-Objekte wollen wir im folgenden die semantischen Objekte in unserem Editor genauer beschreiben.

5.1.3 Elementare Netzelemente im Editor

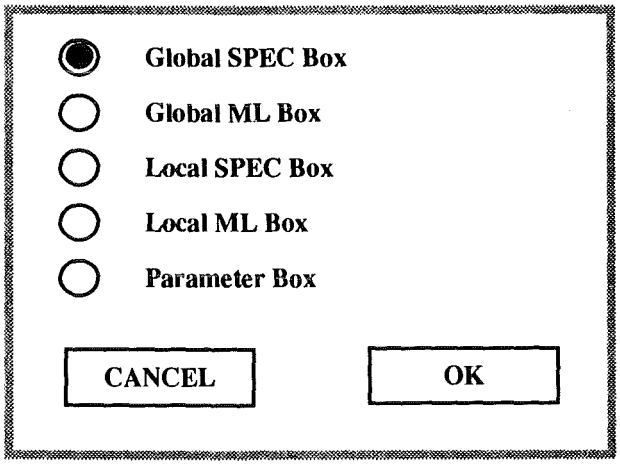
Als elementare Bauelemente zum Aufbau von SNL-Systemen stellt der Editor dem Anwender Deklarationsboxen, die größere textuelle Teile eines Modells aufnehmen, Transitionen, Stellen, Kanten und die diversen Beschriftungselemente von Transitionen, Stellen und Kanten bereit. Wir wollen diese elementaren semantischen Objekte nun im einzelnen vorstellen.

5.1.3.1 Deklarationsboxen

Deklarationsboxen enthalten die größeren textuellen Teile eines SNL-Modells. Sie sind graphisch gesehen Design/OA Box-Objekte. Dabei unterscheiden wir Deklarationsboxen in drei Typen. Spezifikationsdeklarationsboxen enthalten die abstrakten Datentypspezifikationen, die die Spezifikation der Beschriftungselemente des Netz-Modells bestimmen. Ihr Inhalt ist im wesentlichen im Sprachspezifikationsteil unter dem Abschnitt Spezifikation abstrakter Datentypen spezifiziert. ML-Deklarationsboxen enthalten dagegen ML-Deklarationen, die eine Realisierung der abstrakten Spezifikationselemente in der korrespondierenden Spezifikationsbox in der Sprache ML darstellen. Ihr Inhalt wird verwendet, um die abstrakten Beschriftungselemente des SNL-Modells während einer Simulation oder Erreichbarkeitsanalyse durch den ML-Interpreter zu interpretieren. Parameterdeklarationsboxen enthalten schließlich die Schnittstellenbeschreibung der formalen Parameter einer Netz-Klasse oder eines Netz-Moduls.

Wir führen noch eine weitere Unterscheidung von Deklarationsboxen in lokale und globale Spezifikations- bzw. ML-Deklarationsboxen durch. Zu jedem SNL-Modell gibt es genau eine globale Spezifikations- und ML-Box. Diese enthalten alle Spezifikationen, Signaturen und Funktordefinitionen bzw. die korrespondierenden ML-Strukturen, ML-Signaturen und ML-Funktoren, die global im gesamten SNL-Modell bekannt sind. Jeder Seite eines SNL-Modells (eine Seite entspricht bzgl. unseres SNL-Modells einer Ebene der Hierarchie bzgl. von Substitutionselementen) können weiter lokale Deklarationsboxen zugeordnet werden, deren Inhalt nur für die jeweilige Seite Gültigkeit hat. Lokale Deklarationsboxen enthalten im Gegensatz zu globalen Deklarationsboxen nur die inneren Teile einer SNL-Spezifikation oder einer ML-Struktur.

Eine Deklarationsbox erzeugt man durch Aktivierung des Menuepunktes **Declaration Node...** im **Net** Menue. Man erhält dann einen Dialog, der es einem erlaubt, den Typ der zu erzeugenden Deklarationsbox anzugeben. Das folgende Bild zeigt diesen Dialog:



Im folgenden wollen wir die einzelnen Deklarationsboxen und ihren Inhalt genauer vorstellen.

5.1.3.1.1 Globale Spezifikationsdeklarationsboxen

Globale Spezifikationsboxen dienen zur Angabe der im gesamten Netz verwendbaren abstrakten Datentypspezifikationen, Signaturen oder parametrisierten Spezifikationen, die bei Bedarf in lokalen Spezifikationsdeklarationsboxen referenziert oder eingebunden werden. Die Syntax zur Angabe einer einzelnen Spezifikations-, Signatur- oder Funktordeklaration haben wir bereits in der Sprachspezifikation spezifiziert. Eine globale Spezifikationsdeklarationsbox enthält nun einfach eine Folge solcher Spezifikations-, Signatur- und Funktordefinitionen, die jeweils durch ein Semikolon abgeschlossen sind, sowie möglicherweise Anweisungen, Definitionen aus externen Dateien einzubinden. Diese Dateien müssen dann wieder im Format einer globalen Spezifikationsdeklarationsbox aufgebaut sein. Das folgende Bild zeigt ein Beispiel einer globalen Deklarationsbox:

Global Declaration Page

```
(* Parameterspezifikation des FIFO Funktors *)
specification FIFO_PARAM = spec
     sort elem;
end;

(* Und nun der FIFO Funktor selbst *)

functor FIFO(FP:FIFO_PARAM) :
sig
    sort  elem;
    union fifo = empty | put of fifo * elem;
    ptl   get : fifo -> elem;
          rest : fifo -> fifo;
    opn   length : fifo -> int;
end
= spec
    sort elem=FP.elem;
    var  w:fifo; n:elem;
    eqn  def get(w) <=> w <> empty;
       if (w <> empty) then get(put(w,n))=get(w)
                       else get(put(w,n))=n;
       def rest(w) <=> w <> empty;
       if (w<>empty) then rest(put(w,n))=put(rest(w,n))
                       else rest(put(w,n))=empty;
       length(put(w,n))=length(w)+1;
       length(empty)=0;
 end;

specification PROCESS = spec
     union process = a1 | a2 | b;
     union data = c1 | c2;
     ptl mdata : (pr : process | pr <> b) -> data;
     opn proc : data -> process;
     var n : data; pr : process;
     eqn if (pr <> b) then proc(mdata(pr))=pr;
         mdata(proc(n))=n;
end;
```

Die Syntax des Textinhaltes einer globalen Deklarationsbox ist gegeben durch:

global_spec_box:	*{ global_decl ; }*$_0^*$
global_decl:	*adt_decl*
	extern_decl
adt_decl:	*spec_decl*
	sig_decl
	functor_decl
extern_decl:	**use " file_name "**

Die Extern-Deklaration besteht aus dem Schlüsselwort **use**, dem in Anführungszeichen (") eingeschlossen der absolute oder relative Pfadname der Datei folgt, die an der Stelle der Extern-Deklaration inkludiert werden soll. Man beachte, daß wir hier **file_name** einfach als ein Token auffassen, dessen Aussehen jedoch durch die Semantik eines UNIX Pfadnamens bestimmt ist. Dateien, die durch relative Pfadnamen bestimmt sind, werden dabei zunächst in Directories gesucht, die durch die Environment-Variable SMARAGDPATH als durch Doppelpunkt getrennte Liste von Directories gegeben sind. Wird hier die Datei nicht gefunden, so wird ein bei der Übersetzung von SMARAGD definiertes Defaultdirectory durchsucht.

5.1.3.1.2 Globale ML-Deklarationsboxen

Eine globale ML-Deklarationsbox dient zur Angabe des ML-Codes, der die Spezifikationen, Signaturen und Funktoren innerhalb der globalen Spezifikationsdeklarationsbox realisiert. Das folgende Bild zeigt eine ML-Deklarationsbox, die den Inhalt der globalen Spezifikationsbox aus dem vorherigen Bild realisiert

Global Declaration Page (ML-Teil)

```
(* Signatur des FIFO Funktors)
signature FIFO_PARAM = sig
    type elem;
end;
(* Und nun der FIFO Funktor selbst *)
functor FIFO(FP:FIFO_PARAM)= struct
   type elem=FP.elem
   datatype fifo = empty | put of fifo * elem
   exception get : unit and rest : unit
   fun get (empty) = raise get
      | get (put(empty,n))=n
      | get (put(w,n))=get(w)
   fun rest (empty) = raise rest
      | rest (put(empty,n)) = empty
      | rest (put(w,n)) = put(rest(w),n)
   fun length (empty) = 0
      | length (put(w,n)) = length(w)+1
end;

structure PROCESS = struct
   datatype process = a1 | a2 | b
   datatype data = c1 | c2
   exception mdata : unit
   fun mdata(a1)=c1 | mdata(a2)=c2
     | mdata(b) = raise mdata
   fun proc(c1)=a1 | proc(c2)=a2
end;
```

Eine ML-Deklarationsbox enthält damit ML-Struktur, ML-Signatur- und ML-Funktordefinitionen zu jeder korrespondierenden Spezifikations-, Signatur- und Funktordefinition innerhalb der globalen Spezifikationsdeklarationsbox. Ihre Syntax korrespondiert daher mit der Syntax einer globalen Spezifikationsdeklarationsbox:

global_ml_box:	$\{ ml\_decl ; \}_0^*$
ml_decl:	*module_decl*
	extern_decl
	other_decl
module_decl:	*ml_structure_decl*
	ml_sig_decl
	ml_functor_decl
extern_decl:	**use " file_name "**

Unter *module_decl* sind in der obigen Syntax die ML-Struktur, ML-Signatur- und ML-Funktordefinitionen zusammengefaßt, die mit den jeweiligen Konstrukten aus der globalen Spezifikationsdeklarationsbox korrespondieren. Die Syntax dieser ML-Modulkonzepte ist in [Harper88] beschrieben. Es gelten hier die Regeln zur Realisierung der Spezifikationskonstrukte, wie sie bereits als Regeln zur Realisierung der Spezifikationselemente im Sprachteil erörtert wurden.

Neben den ML-Moduldefinitionen, die Spezifikationskonstrukte realisieren, dürfen in der globalen ML-Deklarationsbox beliebige andere ML-Deklarationen und Ausdrücke stehen, die z.B. für ausführbare Codeteile (siehe etwas später bei der Beschriftung von Transitionen) während der Simulation des Netzes verwendet werden. Auf die genaue Syntax solcher anderer ML-Konstrukte, die unter dem Punkt *other_decl* zusammengefaßt sind, können wir an dieser Stelle nicht eingehen. Man findet sie z.B. in [Harper89, Harper90].

5.1.3.1.3 Lokale Spezifikationsboxen

Lokale Spezifikationsboxen enthalten im Gegensatz zu globalen Deklarationsboxen nicht eine Folge von Spezifikations-, Signatur- und Funktordefinitionen, sondern den Inhalt einer Spezifikationsdefinition. Die Syntax des Inhaltes einer lokalen Spezifikationsbox ist damit durch die Syntax eines Spezifikationsrumpfes gegeben.

local_spec_box: *spec_body*

Die Syntax für einen Spezifikationsrumpf haben wir bereits im Sprachteil beschrieben. Man beachte, daß damit eine lokale Spezifikationsbox keine Funktor- und Signaturdefinitionen, wohl aber Spezifikationsdefinitionen enthalten darf. Funktor-, Signatur- sowie auch Spezifikationsdefinitionen innerhalb der globalen Spezifikationsbox können jedoch wie in anderen Spezifikationsrümpfen auch verwendet werden. Das folgende Bild zeigt eine lokale Spezifikationsbox:

```
include PROCESS; (* PROCESS inkludieren *)
       FIFO(spec sort elem = PROCESS.data; end); (* FIFO *)

(* Konstante, die maximale Laenge von FIFO bestimmt *)
const maxf : int;
```

Beschriftungselemente der Netz-Objekte einer Seite des SNL-Modells, der eine gegebene lokale Spezifikationsbox zugeordnet ist, kann man ebenfalls als Elemente des Spezifikationsrumpfs der Spezifikation ansehen, die der Seite über die lokale Spezifikationsbox zugeordnet ist. Dies bedeutet, daß wir die innerhalb einer lokalen Spezifikationsbox definierten Namen für Sorten, Konstanten, Operatoren und Relationen für die Beschriftung verwenden können, ohne daß wir sie als Pfadnamen angeben müssen. Insbesondere ist auch die definierte Schreibweise (Infix oder Präfix oder Funktionsanwendung) wie innerhalb eines Spezifikationsrumpfes üblich voll auf die Beschriftungselemente übertragbar. Elemente von Unterspezifikationen bzw. inkludierten Spezifikationen oder Spezifikationen, die durch Funktoranwendungen eingebunden werden, müssen natürlich mit Pfadnamen referenziert werden, sofern man sie nicht durch eine Inkludedeklaration öffnet. Spezifikationen oder Funktoren, die innerhalb der globalen Spezifikationsbox definiert sind, wird man dabei in der Regel in eine lokale Deklarationsbox über eine Inklude- oder eine Unterspezifikationsdeklaration einbinden, wenn deren Elemente innerhalb der Beschriftung verwendet werden soll, anstatt sie direkt über Pfadnamen in der Beschriftung zu referenzieren.

5.1.3.1.4 Lokale ML-Boxen

Lokale ML-Boxen enthalten den ML-Code, der den Spezifikationsrumpf einer lokalen Spezifikationsbox realisiert. Das folgende Bild zeigt eine lokale ML-Box, die den Inhalt der lokalen Spezifikationsbox im vorherigen Bild realisiert.

```
open PROCESS (* PROCESS struct oeffnen *)
(* FIFO instantiieren und oeffnen *)
open FIFO(struct type elem= PROCESS.data end);

(* Konstante maxf initialisieren *)
val maxf = 10
```

Der Inhalt einer lokalen ML-Box stellt damit den Rumpf einer ML-Strukturdefinition dar:

local_ml_box: *ml_structure_body*

Auf die genaue Form von *ml_structure_body* können wir an dieser Stelle nicht näher eingehen. Man findet sie in der bereits vorgestellten ML-Literatur. Es gelten hier aber die gleichen Regeln, wie wir sie bereits im Rahmen der Realisierung von abstrakten Datentypspezifikationen in ML erläutert haben.

Wie auch bei lokalen Spezifikationsboxen gilt hier ebenfalls die Randbedingung, daß ML-Signatur- und ML-Funktordefinitionen nicht in lokalen ML-Boxen auftreten dürfen, da diese innerhalb einer ML-Strukturdefinition nicht vorkommen dürfen. Es dürfen aber wohl Strukturdefinitionen innerhalb einer lokalen ML-Box verwendet werden. Entsprechend kann man die Struktur-, Signatur- und Funktordefinitionen der globalen ML-Deklarationsbox innerhalb einer lokalen Deklarationsbox nutzen, als wären sie auf dem Top-Level des ML Interpreters auf gleicher Stufe mit der Struktur, die die lokale ML-Deklarationsbox enthält, deklariert. Innerhalb des SMARAGD-Werkzeuges ist dies bei der Interpretation eines Netzes genau der Fall.

Neben den Elementen, die zur Realisierung der korrespondierenden Objekte der lokalen Spezifikationsbox nötig sind, kann eine lokale ML-Box weitere ML-Codeteile enthalten, die z.B. von den Beschriftungselementen Code einer Transition während einer Simulation verwendet werden (Dies werden wir etwas später noch genauer erörtern).

5.1.3.1.5 Parameterdeklarationsboxen

Jeder Netz-Klasse bzw. jedem Netz-Modul (bzw. im SMARAGD-Editor der korrespondierenden Seite) ist genau eine Parameterdeklarationsbox zugeordnet, die die formalen Parameter der Netz-Klasse bzw. des Netz-Moduls beschreibt. Das folgende Beispiel zeigt die Parameterdeklarationsbox der FIFO Netz-Klasse.

```
param Fifo_Param : FIFO_PARAM; (* Elementsorte der FIFO *)
param maxf : int; (* maximale Laenge der FIFO *)
```

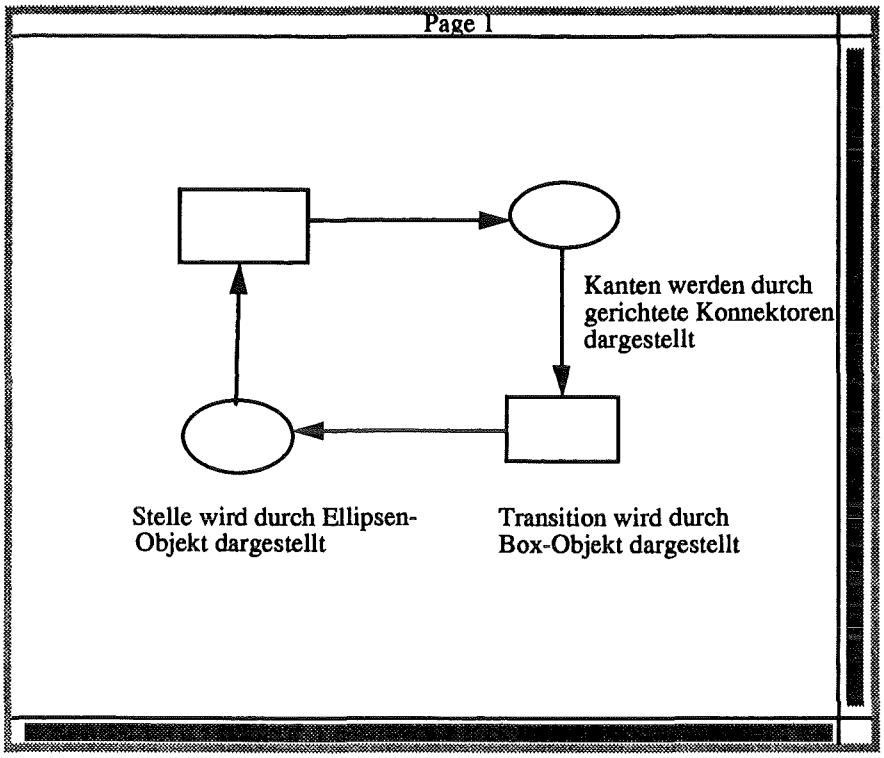
Die Syntax des Inhaltes einer Parameterdeklarationsbox ist:

param_box: *net_param_decls*

Dabei wurde die Syntax von *net_param_decls* bereits im Sprachteil definiert. In den Abschnitten, in denen wir die Hierarchisierungskonstrukte beschreiben, werden wir noch einige Beispiele für den Inhalt von Parameterdeklarationsboxen sehen.

5.1.3.2 Stellen, Transitionen und Kanten im Editor

Stellen werden graphisch als Design/OA-Ellipsen-Objekte (Default) im Editor dargestellt. Transitionen werden graphisch als Design/OA-Box-Objekte (Default) dargestellt, und Kanten werden im Editor durch Design/OA-Konnektoren repräsentiert. Da die Semantik von Petri-Netzen nur Kanten zwischen Stellen und Transitionen erlaubt, werden im Kantenerzeugungsmodus vom Editor nur Konnektoren als Kanten erzeugt, die jeweils eine Stelle und eine Transition miteinander verbinden. Weiter sind die Konnektoren, die für Kanten stehen, gemäß der Petri-Netz Semantik gerichtet, d.h. mit einem Pfeil vom ersten Objekt zum zweiten Objekt versehen. Das folgende Bild zeigt eine Diagrammseite mit Stellen-, Transitions- und Kanten-Objekten.

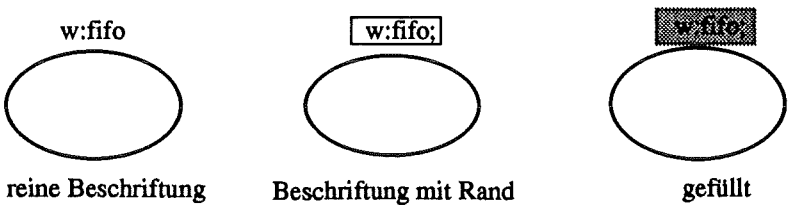


Die Stellen-, Transitionen- und Kanten-Objekte übernehmen dabei viele der Eigenschaften von Ellipsen-, Box- und Konnektor-Objekten, wie z.B. die Graphikattribute und die Textattribute, so daß ihr Aussehen entprechend veränderbar ist. Weiter ist die Zuordnung eines semantischen Objekts Stelle, Transition oder Kante zu den graphischen Design/OA-Objekten Ellipse, Box und Konnektor nur defaultmäßig vorgegeben und kann mit dem Menuepunkt **Change Shape** im **Makeup** Menue geändert werden. So kann man durchaus eine vielleicht besonders zu kennzeichnende Transition graphisch durch ein Dreieck (Design/OA-Objekt Polygon) repräsentieren.

5.1.3.3 Beschriftung von Stellen, Transitionen und Kanten

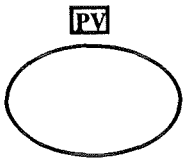
Beschriftungselemente werden anderen Netz-Objekten (wie Transition, Stelle oder Kante) als Design/OA-Regionen zugeordnet. Dabei kann der Anwender wählen, ob er Beschriftungselemente mit einer zusätzlichen *Tag-Region* (Schlüssel-Region) versieht.

Wenn ein Beschriftungselement keine Tag-Region besitzt, wird es durch ein Box-Objekt (größere textuelle Teile) oder Label-Objekt dargestellt, das die Beschriftung als Text enthält und dem zugehörigen Netz-Objekt als Region zugeordnet ist. Über die graphischen Attribute eines Box-Objektes kann man für Beschriftungen, die durch ein Box-Objekt dargestellt werden, festlegen, ob die zum Beschriftungselement gehörige Box mit einem Muster gefüllt oder nicht gefüllt, nur durch einen Rand begrenzt oder ganz unsichtbar dargestellt wird. Im letzteren Falle sieht man nur den reinen Beschriftungstext. Das folgende Bild zeigt mögliche Beschriftungsformen ohne Tag-Region für die Variablenbeschriftung einer Stelle.

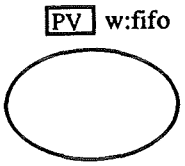


Wenn einem Beschriftungselement ein Tag zugeordnet ist, so erhält man eine Hierarchie von Regionen, in der ein Label oder Box-Objekt, das den eigentlichen Beschriftungstext aufnimmt, der Tag-Region als Kind-Objekt zugeordnet ist. Die Tag-Region selber wird als Kind-Objekt des Netz-Objektes angelegt, dem die Beschriftung zugeordnet werden soll. Eine Tag-Region wird stets als ein berandetes aber nicht gefülltes Box-Objekt dargestellt, das einen Text als Label gerade umfaßt. Dieser Text kann vom Benutzer vorgegeben werden und sollte eine prägnante Abkürzung zur Erkennung des Beschriftungstyps oder der Beschriftung sein.

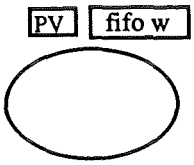
Tag-Regionen sind im Editor so realisiert, daß sie zwar bewegt, nicht aber von der Größe verändert werden können. Ein Doppelklick innerhalb der Tag-Region eines Beschriftungselementes, dessen Beschriftungstext sichtbar ist, bewirkt, daß das Box-Objekt, welches die Beschriftung als Text enthält, mitsamt dem Beschriftungstext unsichtbar wird. Man sieht anschließend nur noch die Tag-Region der Beschriftung als ein Zeichen für die unsichtbare Beschriftung. Ein erneuter Doppelklick innerhalb der Tag-Region läßt die untergeordnete Box-Region mit dem Beschriftungstext wieder zum Vorschein kommen. Das folgende Bild zeigt die Variablenbeschriftung einer Stelle in verschiedenen Formen mit Tag-Regionen:



Variablenbeschriftung wird durch Tag-Region beschrieben. Beschriftung selbst ist unsichtbar

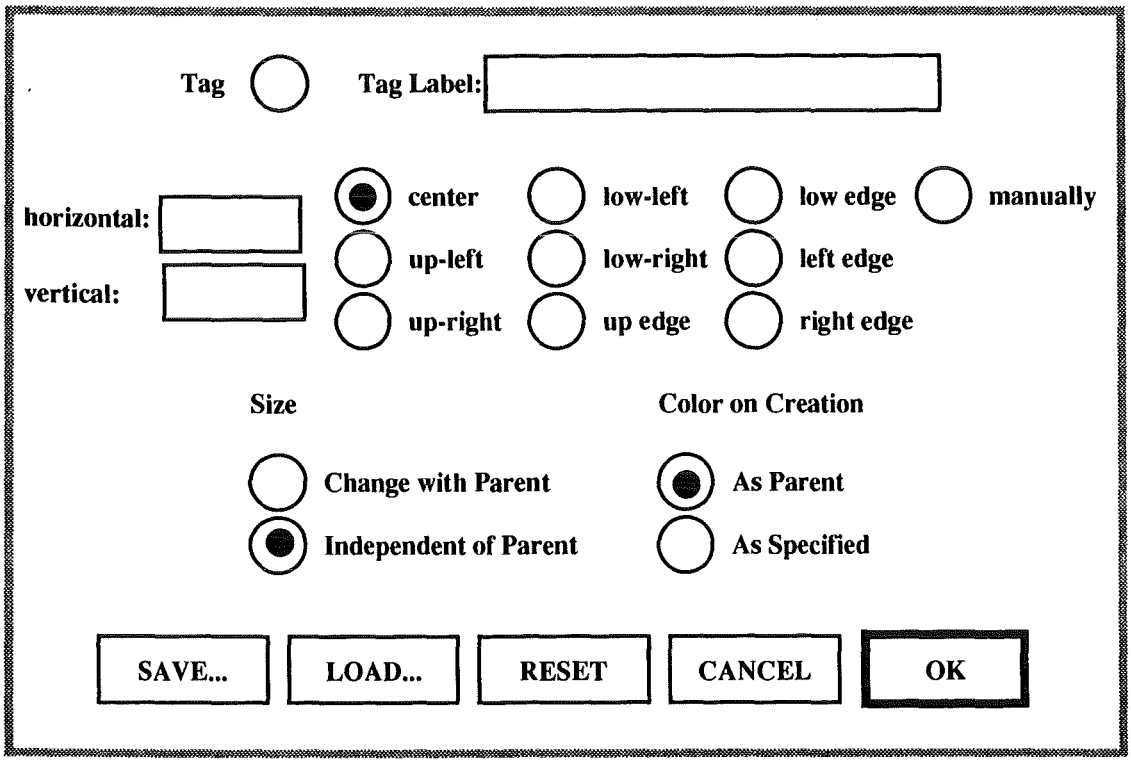


Tag und Beschriftung sind sichtbar.



Tag und Beschriftung sind sichtbar. Beschriftung hat Rand.

Die Zuordnung von Tag-Regionen zu Beschriftungselementen kann man über den **Region Attributes...** Menuepunkt im **Set** Menue ändern. Über diesen Dialog kann man auch eine Autopositionierung der Beschriftungselemente bzgl. ihrer Vaterobjekte festlegen. So kann der Benutzer vorgeben, ob ein Objekt automatisch bzgl. einer der Ecken, einer der Mittelpunkten der Kanten oder bzgl. des Mittelpunktes des Objektes selbst positioniert werden soll. Er kann auch eine Einstellung wählen, bei der er die Beschriftungselemente manuell beliebig positionieren kann. Bei Aktivierung des **Region Attributes...** Menuepunktes erscheint ein Dialogfenster der folgenden Gestalt:



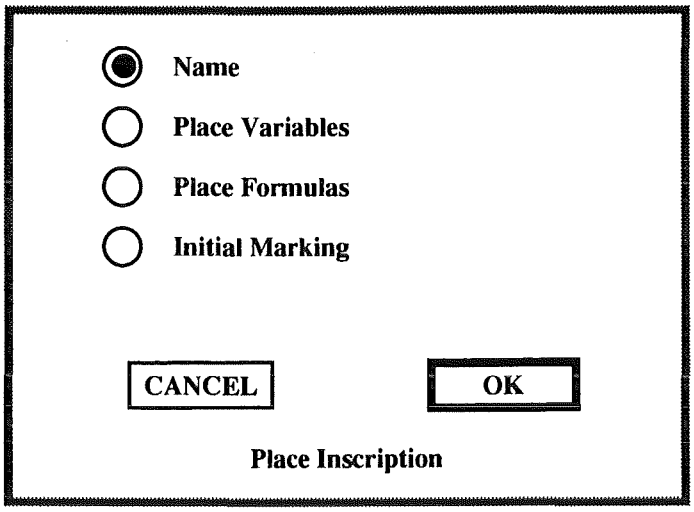
Wenn man die Tag-Option (Vorhandensein einer Tag-Region) durch Aktivieren des Radiobuttons selektiert, kann man im Textfeld das Label der gewünschten Tag-Region angeben. Über die Radiobuttongruppe **center**, **up-left**, **up-right**, **low-left**, **low-right**, **up edge**, **low edge**, **left edge**, **right edge** und **manually** kann der Benutzer einstellen, ob das Beschriftungselement automatisch bzgl. des Zentrums des Objektes, der oberen linken, der oberen rechten, der unteren linken oder der unteren rechten Ecke des Vaterobjektes, des Mittelpunktes der oberen, unteren, linken oder rechten Kanten des Objektes oder manuell positioniert werden. Die Texteditierfelder **horizontal** und **vertical** nehmen dabei Integerzahlen auf, die die relative Verschiebung des Beschriftungselementes vom gewählten Referenzpunkt des Vaterobjektes angeben. Zwei weitere Gruppen von Radiobuttons bestimmen, ob die Größe einer Region mit der Größe des Vaterobjektes verändert werden soll, und ob die Region die Farbe des Vaterobjektes erbt.

Defaultmäßig wird nur die Einstellung des aktuellen Beschriftungselementes und der folgenden geändert. Möchte man die neuen Optionseinstellungen als System- oder Diagrammdefaults sichern, so muß man die Check-Box **Save...** aktivieren. Durch Drücken des **OK** Buttons werden dann die neuen Einstellungen wirksam.

Wir wollen im folgenden die einzelnen Beschriftungselemente von Stellen, Transitionen und Kanten beschreiben, wobei wir eine Defaulteinstellung des Systems bzgl. der Graphik, Text- und Region-Attribute voraussetzen.

5.1.3.3.1 Beschriftungselemente von Stellen

Wenn man eine Stelle selektiert hat und den **Inscription...** Menuepunkt im **Net** Menue aktiviert, erhält man das folgende Dialogfenster, das die Auswahl eines speziellen Beschriftungselementes für die Stelle erlaubt.



Jeder Stelle kann man durch Auswahl der Beschriftungsoption **Name** zunächst einmal einen Namen zuordnen, wobei ein Name einfach ein Zeichenstring vom Tokentyp **id** ist.

Weiter ist jeder Stelle eine Variablenbeschriftung zugeordnet, die man bei Auswahl der Option **Place Variables** angeben kann. In der Variablenbeschriftung einer Stelle werden die Sorten, zugehörigen Kapazitäten und Variablen einer Stelle definiert. Die Syntax für den Texteintrag in einer **Place Variables** Region ist:

$$\textit{place\_var\_inscr}: \quad \textit{sort\_var\_cap\_decl}\ \{\ ;\ \textit{sort\_var\_cap\_decl}\ \}_0^*\ [\ ;\ ]$$

Die *sort_var_cap_decl* haben wir dabei bereits im Sprachteil definiert.

Über die **Place Formula** Option kann man einer Stelle Formeln zuordnen, die alle auf der Stelle liegenden Individuen erfüllen müssen. Die Syntax für diese Formeln, die den Textinhalt der zugehörigen Region bestimmen, ist dabei gegeben durch

$$\textit{place\_formula\_inscr}: \quad \textit{formula}\ \{\ ;\textit{formula}\ \}_0^*\ [\ ;\ ]$$

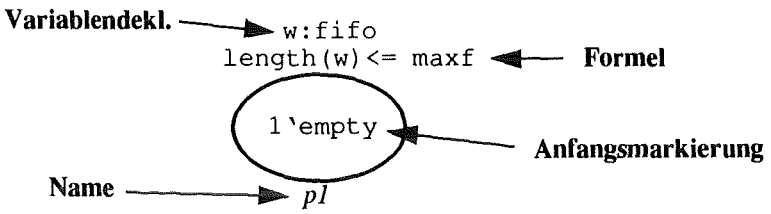
wobei *formula* wie in der Sprachspezifikation definiert ist. Man beachte, daß in der Formel nur Variablen verwendet werden dürfen, die im Stellenvariablendeklarationteil auch definiert sind. Weiter ist die Formel a priori geschlossen, d.h. freie Variablen werden automatisch als allquantifizierte Variablen aufgefaßt.

Über die **Initial Marking** Option kann man schließlich die Anfangsmarkierung der Stelle definieren. Sie hat die Syntax

marking_inscr: *formal_sum*

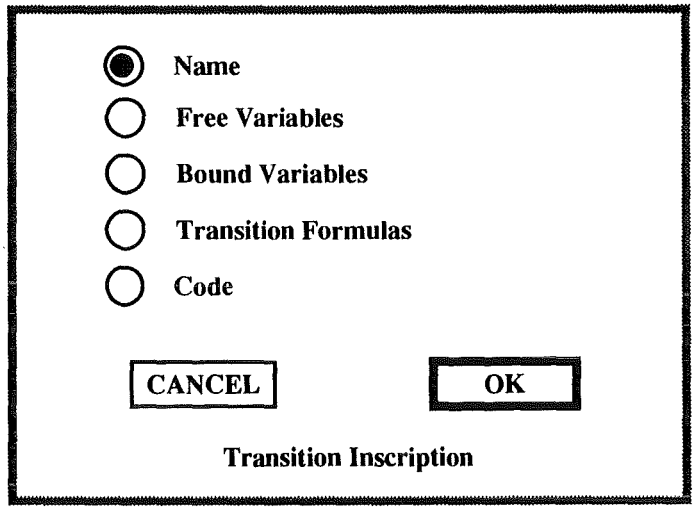
wobei *formal_sum* wie im Sprachspezifikationsteil definiert eine geschlossene formale Summe sein muß, deren Typ eine Teilmenge der Menge der Sortennamen, die der Stelle im Variablendeklarationsteil zugeordnet sind, sein muß.

Da man die Beschriftungselemente von Stellen auf Grund ihres textuellen Inhaltes leicht unterscheiden kann, sind sie defaultmäßig im System als Label-Regionen ohne Tag-Region eingestellt. Der Stellenname wird weiter defaultmäßig kursiv dargestellt. Das folgende Bild zeigt eine Stelle mit ihrer Beschriftung.



5.1.3.3.2 Beschriftungselemente von Transitionen

Wenn man eine Transition selektiert hat und den **Inscription...** Menuepunkt im **Net** Menue aktiviert, erhält man das folgende Dialogfenster, das die Angabe des speziellen Beschriftungselementes für die Transition erlaubt.



Mit der **Name** Option kann wie bei Stellen der Name der Transition als Token vom Typ **id** spezifiziert werden.

Über die **Free Variables** Option müssen die Variablen, die in der Transitionsformel, dem Codeteil der Transition und in den Beschriftungen der Umgebungskanten als freie Variablen verwendet werden, definiert werden, wobei jeder Variablen Umgebungsstellen als "Speicher" für mögliche Werte der Variablen zugeordnet sein müssen. Die Syntax hierfür ist:

trans_fvar_inscr: *trans_var_decl* { ; *trans_var_decl* }$_0^*$ [;]

Die Form von *trans_var_decl* wurde dabei bereits im Sprachteil erläutert.

Über die **Bound Variables** Option werden analog die gebundenen Variablen definiert, die nur als gebundene Variablen in der Transitionsformel verwendet werden dürfen. Ihre Definition erfolgt analog zur Definition der freien Variablen. Wir haben also:

trans_bvar_inscr: *trans_var_decl* { ; *trans_var_decl* }$_0^*$ [;]

Während gebundene Variablen nur als temporäre Größen in Quantorausdrücken innerhalb der Transitionsformel verwendet werden und damit keinen permanenten Wert haben, wird freien Variablen beim Schalten einer Transition ein Wert zugeordnet, der als Marke auf einer der Umgebungsstellen der Transition, an der die Variable über

die Transitionsvariablendeklarationen gebunden ist, vorhanden sein muß. Eine solche Zuordnung (Bindung) der freien Variablen zu Werten, die als Marken auf den Umgebungsstellen liegen, definiert den Mode, in dem eine Transition schaltet. Bei der Verwendung der freien Variablen kann man sich vorstellen, daß die Variable durch den bzgl. eines Schaltmodes gegebenen Wert ersetzt wird.

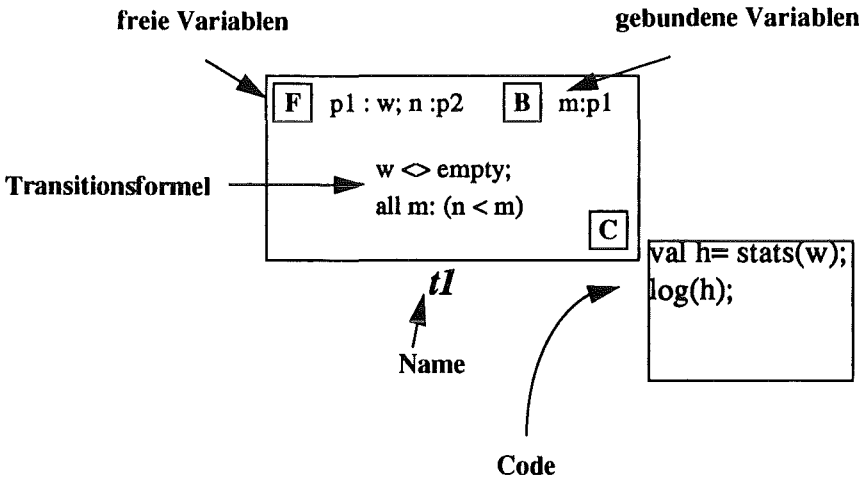
Über die **Transition Formulas** Option können prädikatenlogische Formeln erster Ordnung für die Transition angegeben werden, die bestimmen, ob die Transition in einem gegebenen Ausführungsmode schalten kann. Die Syntax für solche Formeln ist schon im Sprachdefinitionsteil als *formula* definiert worden. Wir haben also:

$$trans\_formula\_inscr: \quad formula\ \{\ ;formula\ \}_0^*\ [\ ;\ ]$$

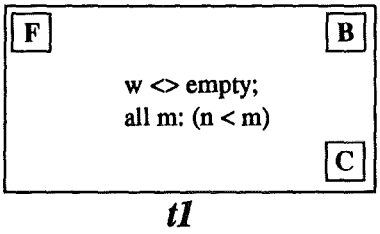
Dabei darf *formula* nur freie bzw. gebundene Variablen enthalten, die als freie bzw. gebundene Variablen der Transition definiert sind. Alle weiteren nicht im Sprachbericht vordefinierten Symbole der Formel müssen natürlich in der globalen bzw. lokalen Spezifikationsdeklarationsbox des Netzes definiert sein. Eine Ersetzung der freien Variablen der Transition an Werte, die in der Umgebung der Transition als Marken vorhanden sind, bestimmt einen potentiellen Schaltmode, der nur dann auftreten kann, wenn eine Transitionsformel bei Ersetzung der freien Variablen durch die ihnen zugeordneten Werte zu *true* ausgewertet werden kann. Im Fall mehrerer Formeln muß man sich diese durch ein logisches Und verknüpft denken.

Über die **Code** Option kann man einer Transition einen beliebigen ML-Ausdruck, dessen Komponenten entweder Standard ML Konstrukte, in der lokalen bzw. globalen ML-Deklarationsbox definierte Größen oder als freie Variablen der Transition definierte Namen sind, zuordnen. Das Code Segment wird bei der Simulation (nicht bei der Erreichbarkeitsanalyse) dann ausgeführt, wenn eine Transition in einem gewissen Mode schaltet, wobei die freien Variablen in ML durch Variablendeklarationen so gebunden sind, wie es der spezielle Schaltmode definiert.

Da sich die freien und gebundenen Variablendefinitionen in ihrer textuellen Form nicht unterscheiden, werden sie defaultmäßig im Editor mit Tag-Regionen mit Label **F** bzw. **B** versehen. Die **Code**-Region ist ebenfalls defaultmäßig mit einer Tag-Region mit Label C versehen. Das Beschriftungselement Name besitzt defaultmäßig keine Tag-Region, wird aber durch eine kursive Schrift (Roman Bold Italics) gekennzeichnet, während die Transitionsformel ebenfalls ohne Tag-Region in Normalschrift dargestellt wird. Das folgende Bild zeigt eine Transition mit ihren verschiedenen Beschriftungselementen.



Die Code-Region ist für die Semantik des Netzes selbst nicht von Belang, sondern dient in der Regel dazu, während einer Simulation z.B. statistische Daten zu sammeln. Der Übersichtlichkeit halber wird man die Text-Box der Code-Region daher oftmals unsichtbar halten. Da weiter die Definition der Variablen oftmals bereits aus den anderen Beschriftungen, in denen sie verwendet werden, ersichtlich ist, kann man auch deren textuellen Inhalt oftmals verdecken und nur durch die zugehörigen Tag-Regionen ihr Vorhandensein anzeigen. Die obige Transition mit ihren Beschriftungen wird man daher vielleicht in der folgenden Weise im Editor darstellen:



Je nachdem, aus welcher Sicht man das Netz betrachtet, kann man also bei Bedarf über das Tag Konzept unwichtige Informationen verdecken. Über die Graphik- und Textattribute sind andere Kennzeichnungsformen wichtiger Teile realisierbar.

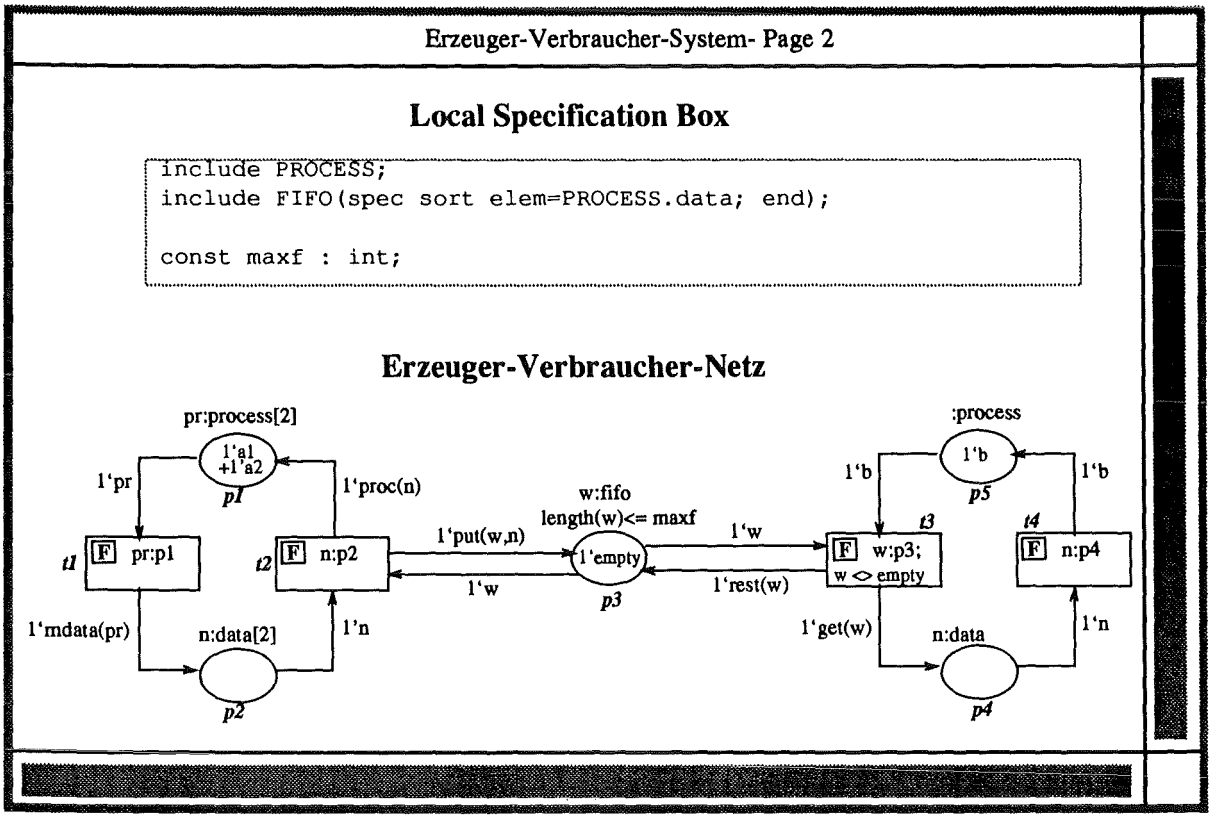
5.1.3.3.3 Beschriftung von Kanten

Kanten besitzen nur eine Beschriftung, die eine formale Summe über Termen der Spezifikationssprache darstellt. Wenn man den **Inscription...** Menuepunkt bei einer selektierten Kante aufruft, kann man daher gleich den Text der Beschriftung eingeben.

Die Syntax für die Kantenbeschriftung ist

arc_inscr: *formal_sum*

wobei *formal_sum* wie im Sprachspezifikationsteil definiert ist. Die Variablen innerhalb der formalen Summe müssen dabei als freie Variablen der zu der Kante gehörenden Transition definiert sein. Der Typ der formalen Summe muß eine Teilmenge der Sorten sein, die an die zugehörige Stelle gebunden sind. Kantenbeschriftungen werden defaultmäßig als Label-Regionen ohne Tag dargestellt. Damit haben wir alle elementaren semantischen Netz-Objekte vorgestellt. Das folgende Bild zeigt eine Seite eines typischen SNL-Modells, wie es mit dem Editor erstellt wurde.



5.1.4 Hierarchisierungselemente im Editor

Es gibt verschiedene Möglichkeiten, eine Hierarchisierung des zu entwickelnden Netz-Modells im Editor durchzuführen. In der Regel wird man eine benötigte Netz-Klasse oder einen benötigten Netz-Modul zunächst als gewöhnliches Netz auf einer neuen Diagrammseite unabhängig von den restlichen Seiten entwerfen. Wenn alle benötigten Stellen, Transitionen und Kanten und sich nicht auf die Hierarchisierung beziehenden Beschriftungen erzeugt sind, wird man einige der Transitionen oder Stellen als Port-Transitionen oder Port-Stellen der Netz-Klasse bzw. des Netz-Moduls auszeichnen. Dies geschieht über den Menuepunkt **Port Node...** im **Net** Menue. Weiter wird man bei Bedarf einige Beschriftungselemente des Netz-Moduls oder der Netz-Klasse als Parameter des Netz-Moduls oder der Netz-Klasse deklarieren. Dies erfolgt über die Erzeugung einer Parameterdeklarationsbox. Der Netz-Editor kennzeichnet die betreffende Seite bei der Erzeugung der Port-Transitionen bzw. Port-Stellen als Netz-Klasse bzw. Netz-Modul, so daß die Netz-Klasse bzw. der Netz-Modul auf anderen Seiten referenziert werden kann.

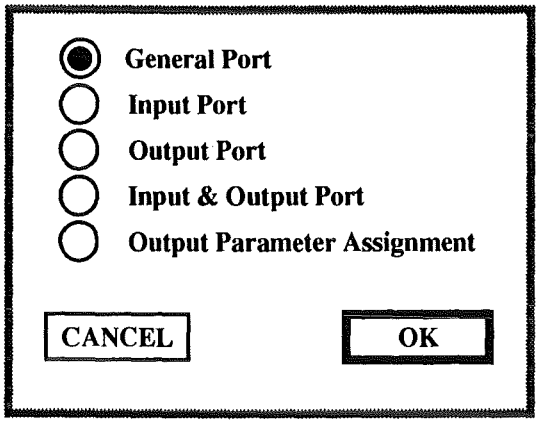
Die so definierte Netz-Klasse bzw. der so definierte Netz-Modul kann nun auf anderen Seiten referenziert werden. Hierzu konvertiert man auf so einer Seite eine Stelle (im Fall einer Netz-Klasse) bzw. eine Transition (im Fall des Netz-Moduls) über den Menuepunkt **Compound Node** des **Net** Menues in eine Substitutionsstelle bzw. eine Substitutionstransition. Im Verlauf dieser Konvertierung ordnet der Benutzer dem Substitutionsknoten die Diagrammseite zu, die die zugehörige Netz-Klasse oder den zugehörigen Netz-Modul beschreibt, und gibt die aktuellen Parameter zur Instantiierung der Netz-Klasse bzw. des Netz-Moduls an. Im Anschluß hieran muß nur noch die Zuordnung der Socket-Knoten zu den Port-Knoten erfolgen. Eine solche Zuordnung eines Socket-Knoten zu einem Port-Knoten erhält man durch Aktivierung des **Port Assignment...** Menuepunktes im **Net** Menue, wobei man zunächst einem selektierten Socket-Knoten einen Port-Knoten zuordnen und dann die Parameterzuordnung angeben muß.

Manchmal kommt es auch vor, daß man ein Netz-Modell editiert hat und dann feststellt, daß ein Teil des bereits erzeugten Netz-Modells besser als eine allgemeine Netz-Klasse oder ein allgemeiner Netz-Modul auf eine andere Seite ausgelagert werden und statt dessen im betrachteten Teil des Modells eine Instantiierung dieser Netz-Klasse bzw. dieses Netz-Moduls erfolgen sollte. Hierzu kann man mit dem Menuepunkt **Move to Subpage...** im **Net** Menue ein selektiertes Teilnetz auf eine neue Seite auslagern, wobei der Editor dieses Teilnetz automatisch auf der übergeordneten Seite durch eine Substitutionstransition oder -stelle ersetzt. Umgekehrt kann man durch den Menuepunkt **Replace by Subpage** im **Net** Menue eine selektierte Substitutionstransition bzw. -stelle automatisch durch das Teilnetz, das den zugehörigen Netz-Modul bzw. die zugehörige Netz-Klasse beschreibt, ersetzen.

Im folgenden wollen wir die Menuepunkte **Port Node**, **Compound Node**, **Port Assignment**, **Move to Subpage...** und **Replace by Subpage** des **Net** Menues genauer beschreiben.

5.1.4.1 Erzeugung von Portknoten

Um einen selektierten Netz-Knoten (Transition oder Stelle) in einen Port-Knoten zu verwandeln, benützt man den **Port Node...** Menuepunkt im **Net** Menue. Dieser konvertiert einen einfachen Knoten in einen Port von einem bestimmten Untertyp (Generic, Eingabe, Ausgabe oder Ein-/Ausgabe) und erlaubt es, dem selektierten Knoten weitere Beschriftungselemente (Schnittstellenbeschreibung und Parameterzuweisung) zuzuordnen. Im Fall einer Port-Transition definiert das eine Beschriftungselement (die Schnittstellenbeschreibung des Ports) die Ein- und/oder Ausgabeparameter der Port-Transition mit deren Typen. Das andere Beschriftungselement (die Parameterzuweisungen des Ports) enthält im Fall der Port-Transition Zuweisungen, die den "Wert" der Ausgabeparameter der Port-Transition festlegen. Im Fall einer Port-Stelle kann man dieser nur ein weiteres Beschriftungselement (die Schnittstellenbeschreibung) zuordnen. Dieses beschreibt, welche Multimengen von Objekten die Stelle als Eingabestelle erwartet bzw. als Ausgabestelle zurückgibt. Nach Aktivieren des **Port Node...** Menuepunktes erhält man ein Dialogfenster, über das der Editor abfragt, welches Beschriftungselement erzeugt werden soll und von welchem Untertyp der Port-Knoten sein soll. Dieses Dialogfenster hat die folgende Gestalt:



Bei Auswahl der Option **General Port** wird ein Portknoten vom Untertyp **Generic** erzeugt. Dies ist nur für Knoten vom Typ Transition erlaubt, und es wird in diesem Fall eine Port-Transition erzeugt, die keine Parameter hat. Bei Auswahl von **Input Port**, **Output Port** und **Input & Output Port** wird ein Portknoten vom Untertyp Eingabe, Ausgabe oder Ein-/Ausgabe erzeugt. Im Fall einer Transition kann diese als Port je nach Untertyp Eingabe-, Ausgabe- oder Ein- und Ausgabeparameter haben. Bei einer Stelle nimmt diese als Port je nach Untertyp Marken von der oberen Hierarchieebene entgegen (Eingabeport), gibt Marken nach oben zurück (Ausgabeport) oder erlaubt beides (Ein-/Ausgabeport).

Nach Auswahl einer der Optionen **General Port, Input Port, Output Port** oder **Input & Output Port** wird der selektierte Knoten (wenn die Syntaxprüfung es erlaubt) in einen Portknoten vom gewählten Untertyp konvertiert und vom System ein Beschriftungselement (Text Label) mit einer Tag-Region mit Label P für Port angelegt. Der Benutzer muß dann im Fall von Eingabe-, Ausgabe- oder Ein-/Ausgabeports die Schnittstellenbeschreibung des Ports in diesem neuen Beschriftungselement eingeben. Im Fall einer Port-Transition (Parameterdefinitionen) muß die Beschriftung der folgenden Syntax genügen:

$$\textit{trans\_port\_param\_inscr}\text{: } \{\ \textit{port\_param\_decls}\ \}_0^*$$

Die Syntax von *port_param_decls* haben wir dabei im Sprachteil schon ausgiebig erörtert. Im Fall einer Port-Stelle hat die Schnittstellenbeschreibung die Gestalt:

$$\textit{place\_port\_param\_inscr}\text{: } \{\ \textit{port\_interface\_decls}\ \}_0^*$$

Die Syntax von *port_interface_decls* haben wir ebenfalls schon im Sprachteil erörtert.

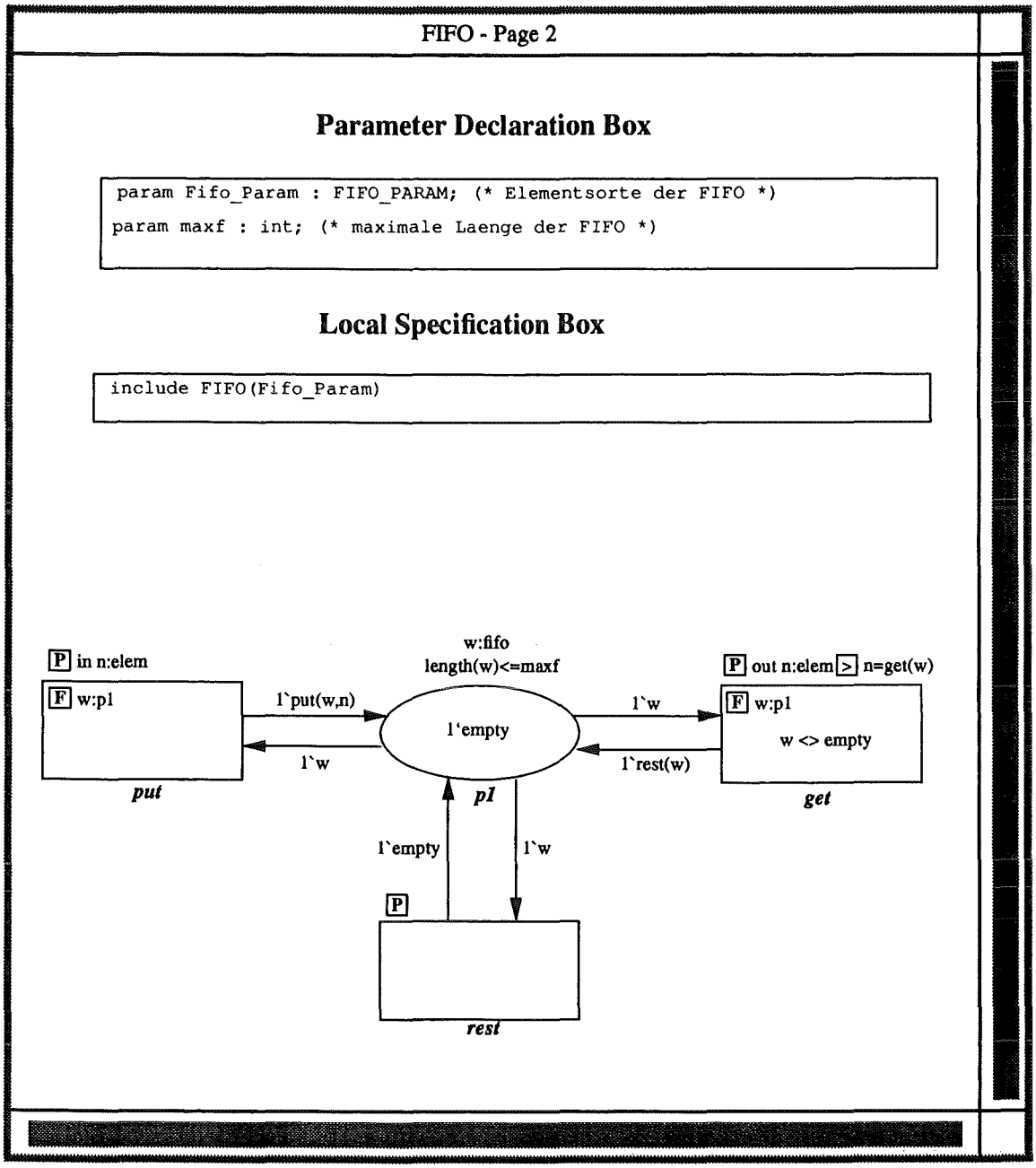
Wenn eine Porttransition vom Untertyp Ausgabe- oder Ein-/Ausgabe erzeugt wurde, so kann man dieser über einen erneuten Aufruf des **Port Node...** Menuepunktes und Auswahl der Option **Output Parameter Assignment** ein weiteres Beschriftungselement (Parameter Zuweisungen) zuordnen, daß die Werte der Ausgabeparameter bei einem Schalten der Port-Transition bestimmt. Dies Beschriftungselement wird defaultmäßig als Textlabel mit Tag > angelegt. Die textuelle Syntax zur Angabe der Werte der Ausgabeparameter ist:

$$\textit{trans\_port\_assign\_inscr}\text{: } \textit{port\_return\_decl}\ \{\ ;\ \textit{port\_return\_decl}\ \}_0^*\ [\ ;\ ]$$

Wir wollen uns zur Verwendung von Port-Transitionen einmal das Beispiel der Netz-Klasse FIFO ansehen.

Beispiel 52 (Die Netz-Klasse FIFO im SMARAGD-Editor)

Das folgende Bild zeigt unsere Netz-Klasse `FIFO` im SMARAGD-Editor.



Hier bilden die Transitionen ***put*** und ***get*** die Porttransitionen. ***put*** ist ein Eingabeport, der einen Eingabeparameter `n` verlangt, wobei `n` das in die `FIFO` auf der Stelle ***p1*** einzutragende Element bezeichnet. ***get*** ist ein Ausgabeport der `FIFO` Netz-Klasse. ***get*** besitzt einen Ausgabeparameter `n`, dem als "Wert" der Term `get(w)` zugeordnet ist. Der Ausgabeparameter gibt somit das erste Element innerhalb der `FIFO` an.

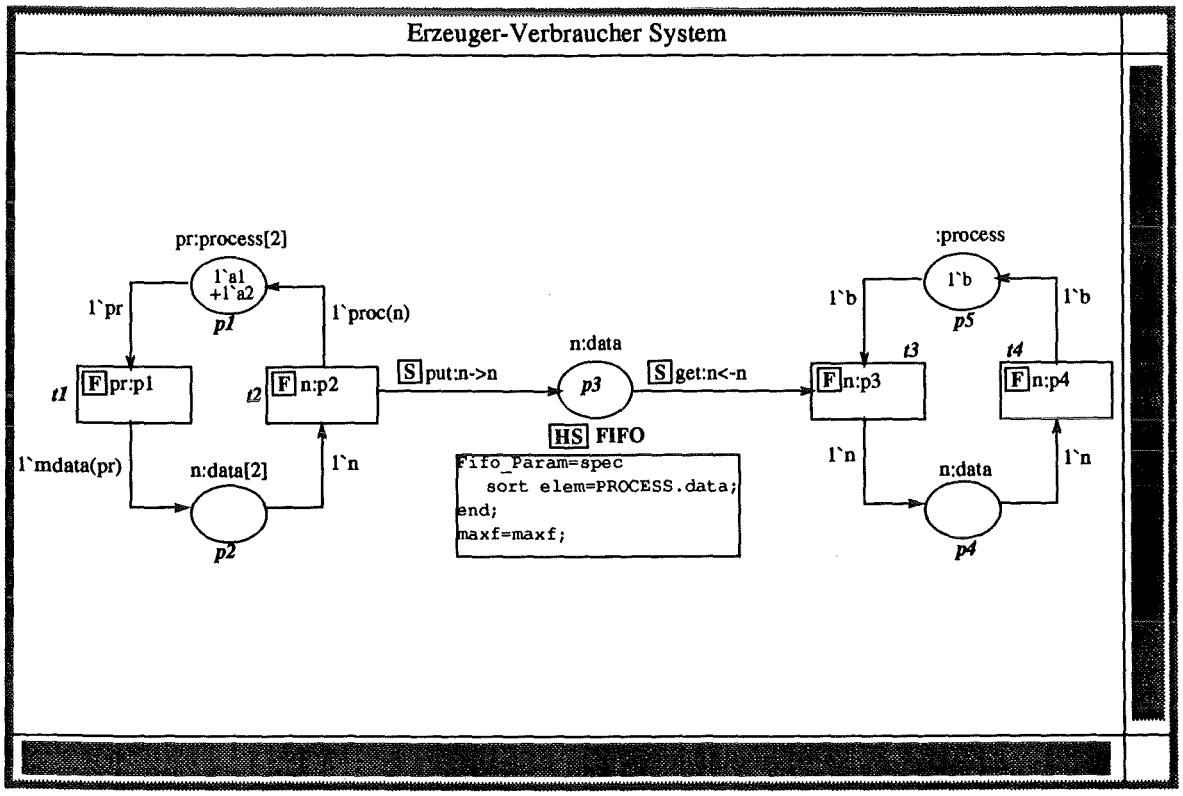
5.1.4.2 Erzeugung von Substitutionsknoten

Zur Erzeugung eines Substitutionsknotens selektiert man zunächst den betreffenden Knoten und ruft dann den Menuepunkt **Compound Node** im **Net** Menue auf. Das zugehörige Kommando öffnet dann automatisch die Hierarchieseite des SNL-Modells, die Knoten für alle Seiten des Modells enthält und die Beziehungen der Seiten untereinander durch Kanten und Beschriftungen anzeigt. Durch Anklicken des gewünschten Seitenknotens auf der Hierarchieseite selektiert der Benutzer nun die gewünschte Netz-Klasse oder den gewünschten Netz-Modul. Der Editor erlaubt dabei nur die Selektion der Seitenknoten, die ein zum Typ des Substitutionsknotens passendes Unternetz beinhalten.

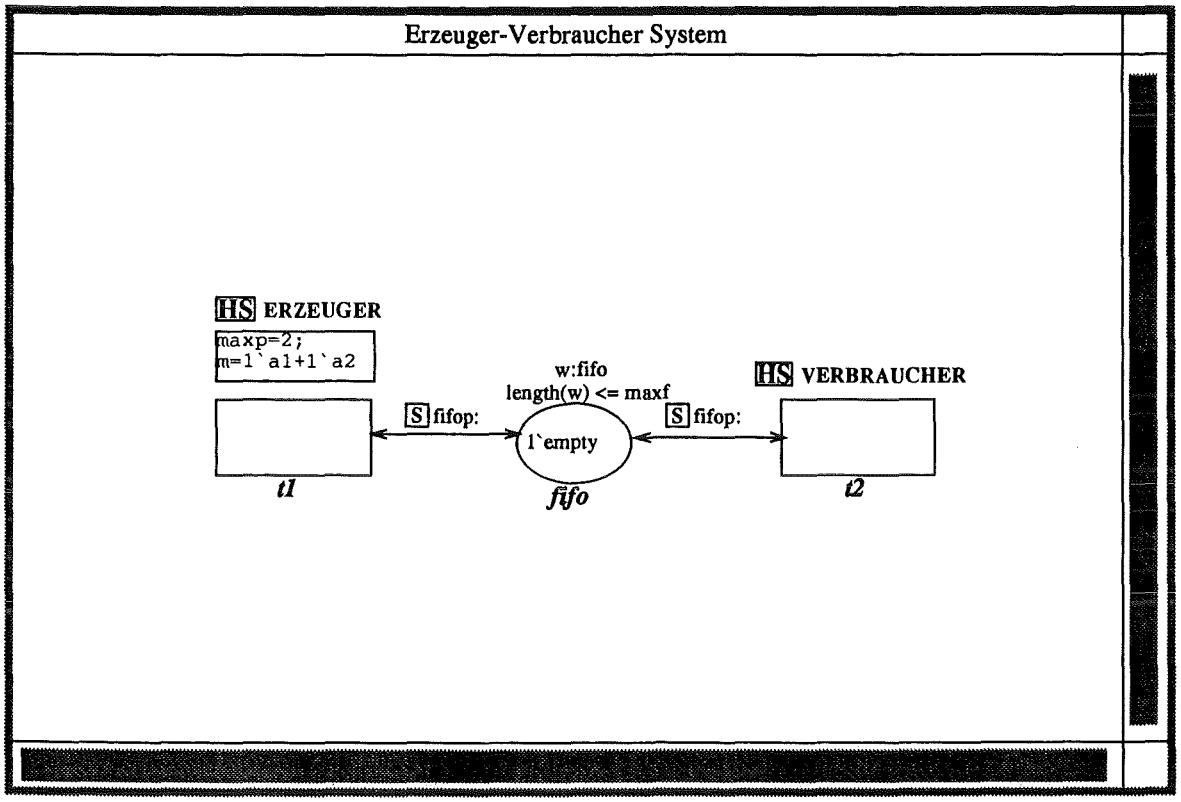
Bei richtiger Selektion konvertiert der Editor dann den selektierten Knoten in einen Substitutionsknoten und erzeugt für ihn defaultmäßig eine Tag-Region mit Inschrift HS, eine Label-Region und eine Box Region, die dieser Tag-Region zugeordnet sind. Die Label-Region enthält den Namen der Unterseite, die dem Substitutionsknoten zugeordnet worden ist, und wird automatisch vom System ausgefüllt. Die andere Box-Region dient zur Angabe der aktuellen Parameter der Netz-Klasse bzw. des Netz-Moduls und muß vom Benutzer ausgefüllt werden. Die Syntax zur Angabe der aktuellen Parameter wurde schon im Sprachteil definiert. Wir haben:

actual_param_inscr: { *inst_param_decl* }$_0^*$

Natürlich muß für jeden formalen Parameter der Netz-Klasse bzw. des Netz-Moduls hier genau ein aktueller Parameterwert angegeben werden. Das folgende Bild zeigt die Instantiierung der Netz-Klasse `FIFO` in unserem Erzeuger-Verbraucher Modell im Editor.



Die Instantiierung der Netz-Module Erzeuger und Verbraucher in einem Erzeuger-Verbraucher System im Editor zeigt das nächste Bild.



5.1.4.3 Zuweisung von Socketknoten zu Portknoten

Als letztes muß man nun die Socket/ Port-Zuordnungen vornehmen. Hierzu selektiert man eine Kante, die einen Socketknoten mit einem Substitutionsknoten verbindet und ruft den Menuepunkt **Port Assignment** im **Net** Menue auf. Der Editor öffnet dann die zu dem Substitutionsknoten gehörige Unterseite und erlaubt es dem Benutzer, einen Portknoten zu selektieren, der vom Typ her zu dem durch die selektierte Kante gegebenen Socketknoten paßt. Nach Anklicken so eines passenden Ports führt dann der Editor die Socket/Port-Zuordnung durch und erzeugt eine Port-Assignment Region, die er der selektierten Kante zuordnet. Diese wir defaultmäßig als Label-Region mit einer Tag-Region, die die Inschrift S hat, angelegt. Der Name des zugehörigen Portknotens wird automatisch in der Beschriftung dieser Assignment Region festgehalten. Die Zuordnung der aktuellen Portparameter im Fall von Socket-Transitionen, die Parameter besitzen, muß dann der Benutzer selbst durchführen, in dem er das Textlabel editiert. Die textuelle Syntax hierfür wurde schon im Sprachteil gegeben. Wir haben:

port_param_ass: *term* -> *param_id*
var_id <- *param_id*

Die letzten beiden Bilder mit den Instantiierungen der Netz-Klasse bzw. der Netz-Module zeigen auch das Aussehen der Socket/Port-Assignment Regionen, wie es als Default im Editor definiert ist.

5.1.4.4 Auslagern von Teilnetzen

Wir waren bislang davon ausgegangen, daß man bei der Modularisierung eines SNL-Modells zunächst die Netz-Module bzw. Netz-Klassen entwirft und diese im Anschluß daran instantiiert. Es kann jedoch auch der Fall sein, daß man mit einem Entwurf schon weit fortgeschritten ist und dann feststellt, daß man die ein oder anderen Teilnetze besser gesondert als allgemeingültige Netz-Module bzw. Netz-Klassen formuliert hätte. Für diese Situation stellen wir im Editor ein Kommando bereit, mit dem man selektierte Teilnetze auf eine gesonderte Diagrammseite auslagern und damit in eine Netz-Klasse bzw. in einen Netz-Modul konvertieren kann.

Hierzu muß der Anwender zunächst ein geeignetes Teilnetz selektieren. Der Editor verifiziert dann, ob die Berandung dieses Teilnetzes (bestimmt durch die selektierten Knoten, die Verbindungen mit nichtselektierten Knoten besitzen) alle vom gleichen Typ sind. Wenn dies der Fall ist, wird das selektierte Teilnetz auf eine neue

Diagrammseite ausgelagert, und je nach Typ der Berandungsknoten als Netz-Klasse bzw. Netz-Modul markiert, wobei die Randknoten in Portknoten vom entsprechenden Typ konvertiert werden. Man beachte dabei, daß die Randknoten auf oberer Ebene als Socketknoten verbleiben und auf der neuen Diagrammseite als Portknoten dupliziert werden, während innere Knoten und Kanten von der oberen Ebene auf die neue Diagrammseite mit der Netz-Klasse bzw. dem Netz-Modul verschoben werden. Anstatt des Teilnetzes wird auf oberster Ebene ein Substitutionsknoten vom entsprechenden Typ erzeugt, der mit den durch die Randknoten gegebenen Socketknoten durch entsprechende Socket-Port-Zuordnungskanten verbunden ist. Alle Beschriftungen, die in Konflikt mit der neuen Identität von Knoten und Kanten stehen werden dabei gelöscht. Dies sind die Beschriftungen der Kanten, die Socketknoten mit dem neuen Substitutionsknoten verbinden, da sie durch Socket-Port-Zuordnungen ersetzt werden müssen.

Der Anwender muß dann zum Abschluß noch von Hand die Portparameter und zugehörigen Socket-Port-Zuordnungen festlegen.

5.1.4.5 Ersetzen eines Substitutionsknotens durch ein Teilnetz

In Umkehrung zu der im obigen Abschnitt beschriebenen Editoroperation bietet dieser weiter eine Operation, mit der ein Substitutionsknoten durch das zu der zugehörigen Netz-Klasse bzw. dem zugehörigen Netz-Modul gehörige Teilnetz ersetzt werden kann.

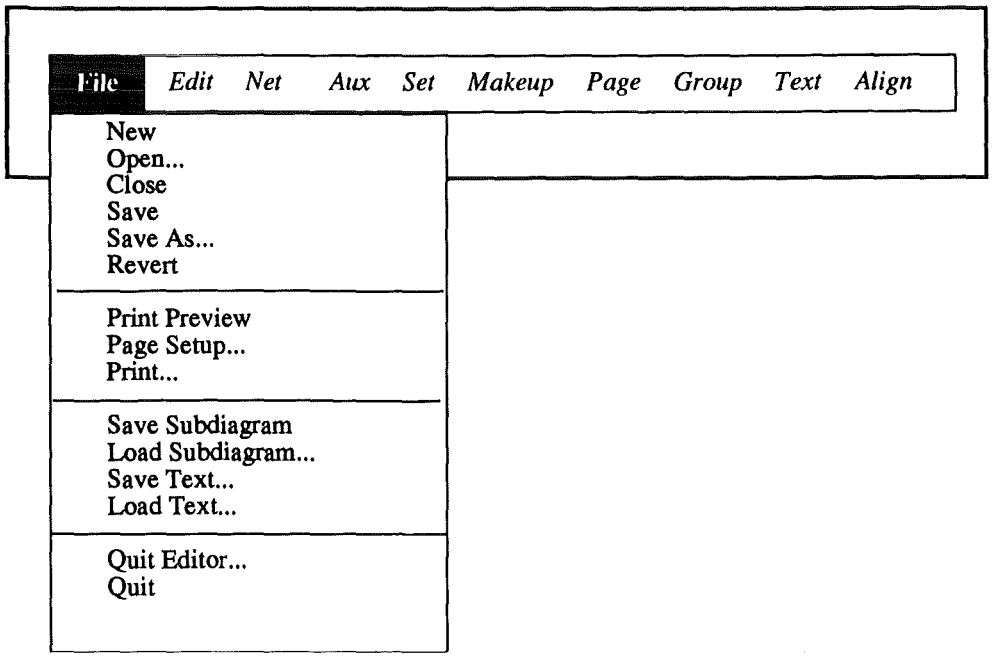
Hierzu selektiert ein Anwender zunächst einen Substitutionsknoten und wählt dann den zugehörigen Menuepunkt aus. Der Editor entfernt dann den Substitutionsknoten und setzt für diesen das zugehörige Teilnetz ein. Dabei werden die Socket-Port-Zuordnungen an den zugehörigen Kanten gelöscht und Portknoten auf die zugehörigen Socketknoten abgebildet.

Der Anwender muß dann im Anschluß hieran noch überprüfen, inwieweit die Variablen, die als Portparameter definiert waren, als Variablen der zugehörigen Socketknoten neu zu definieren sind.

Zum Abschluß dieses Kapitels über den Editor wollen wir noch eine kurze aber vollständige Übersicht über die Menues der Menueleiste und die einzelnen Menuepunkte geben.

5.1.5 Die Menueleiste des Editors

Wir wollen in diesem Unterkapitel einmal kurz die Menues und die in diesen enthaltenen Kommandos unseres Editors vorstellen. Wenn man das Pull-Down Menue **File** aktiviert, erhält man das folgende Menue:



Im **File** Menue sind alle Aktivitäten des Editors zusammengefaßt, die sich auf die Interaktion des Editors mit dem externen Betriebssystem beziehen. Das *New* Kommando initialisiert den Editor für ein neues SNL-Modell. Das *Open* Kommando ermöglicht es dem Benutzer, über einen File-System-Browser ein bereits erstelltes Diagramm in den Editor zu laden. Mit *Save* und *Save As...* speichert man ein editiertes SNL-Modell in einer Datei, wobei

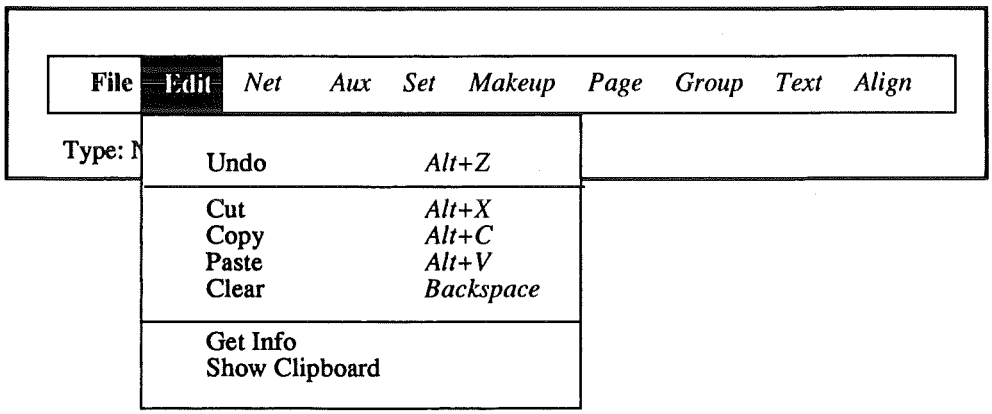
man *Save As...* dann verwendet, wenn noch kein Name für die Datei angegeben worden ist, oder dieser Name geändert werden soll. Mit *Revert* ersetzt man das Modell im Editor durch die in einer externen Datei gespeicherte Version des Modells.

Neben der Ausgabe eines Modells in einer Datei ist auch eine Ausgabe auf einem am System angeschlossenen Postscript-Drucker möglich. Mit dem *Print Preview* bzw. *Print* Kommando kann man ein im Editor befindliches SNL-Modell oder Teile davon auf einem Drucker ausgeben. Mit *Page Setup* kann man einige Voreinstellungen bzgl. der Druckerausgabe, wie das Seitenformat, ändern.

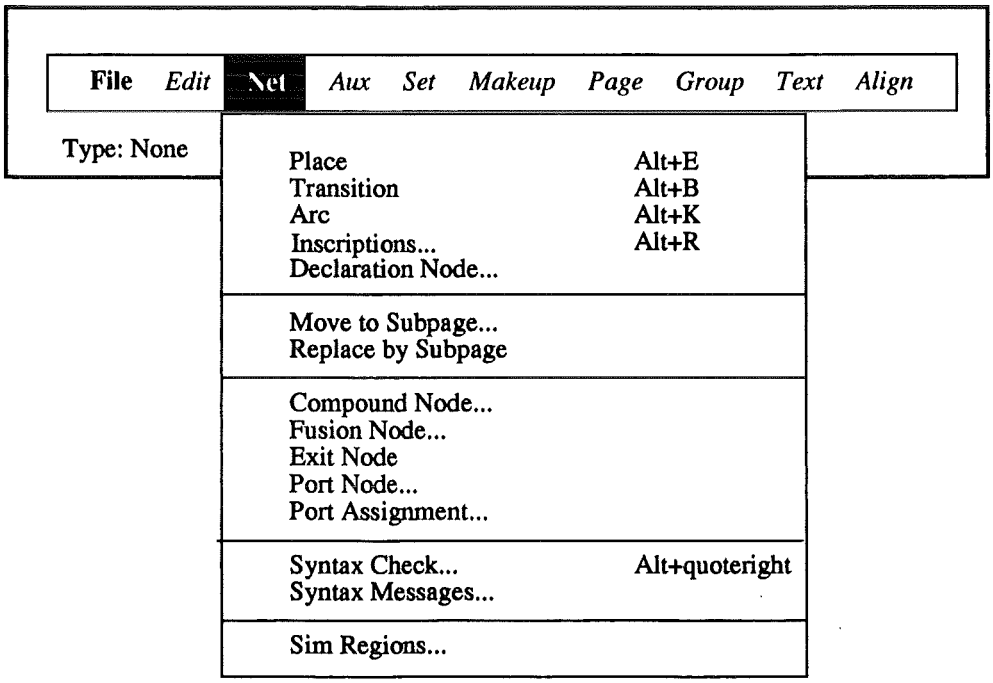
Die Kommandos *Save Subdiagram* und *Load Subdiagram* ermöglichen das Laden und Speichern von Teilen eines Diagramms, während *Save Text* und *Load Text* das Laden und Speichern textueller Teile eines Diagramms ermöglichen.

Mit *Quit Editor* kann man den Editor verlassen und in den Simulator oder in die Analyseeinheit wechseln. Mit *Quit* beendet man SMARAGD.

Das *Edit* Menue faßt wichtige Editieroperationen des Editors zusammen. Nach der Aktivierung erscheint das folgende Menue.



Über das Undo Kommando kann das letzte Editierkommando rückgängig gemacht werden. Mit *Cut* wird ein selektiertes Objekt, eine selektierte Gruppe von Objekten oder ein selektierter Text entfernt, wobei eine Kopie des Objektes bzw. der Objekte auf einem Clipboard gespeichert wird. Mit *Copy* kann ein Objekt oder eine Objektgruppe kopiert werden. *Paste* plaziert den Inhalt des Clipboards an die durch den Mauszeiger gegebene Stelle. Mit *Clear* kann ein selektiertes Objekt bzw. eine Objektgruppe entfernt werden. Mit *Show Clipboard* kann man sich den Inhalt des Clipboards ansehen. *Get Info* liefert Informationen über das selektierte Objekt oder die Objektgruppe.



Über das **Net** Menue werden die semantischen Elemente eines SNL-Modells erzeugt. Mit den meisten Menuepunkten dieses Menues haben wir uns schon im vorherigen Unterkapitel eingehend befaßt. Wir stellen sie daher hier nur noch in Form einer Auflistung zusammen.

Die Menuepunkte *Place, Transition* und *Arc* ermöglichen die Erzeugung von Stellen, Transitionen und Kanten des Netzes. Über den *Inscriptions...* Menuepunkt kann man Stellen, Transitionen oder Kanten Beschriftungselemente wie Namen, Variablen, Formeln und Formale Summen zuordnen. Über den Menuepunkt *Declaration Node...* kann man die verschiedenen Deklarationsboxen erzeugen, die textuelle Teile des SNL-Modells, wie die abstrakten Datentypdefinitionen und ML-Realisierungen dieser Spezifikationen aufnehmen.

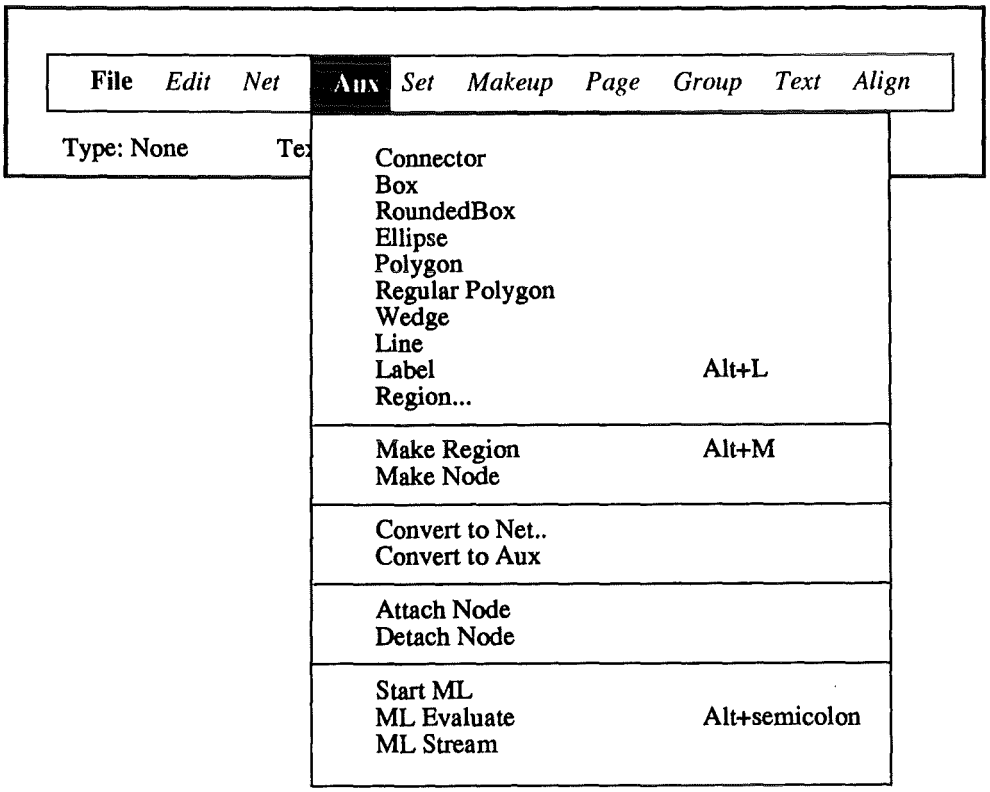
Mit *Move to Subpage* kann man ein selektiertes Teilnetz von einer Seite auf eine andere Seite verlagern, während der Menuepunkt *Replace by Subpage* umgekehrt einen Knoten, der als Repräsentant eines Netzes auf einer anderen Seite steht, durch das zugehörige Teilnetz ersetzt.

Durch den Menuepunkt *Compound Node* kann man einen Knoten (Transition oder Stelle) in eine Substitutionsstelle bzw. Substitutionstransition umwandeln, der damit eine weitere Diagrammseite als Modul bzw. Netz-Klasse zugeordnet ist. Mit dem Menuepunkt *Fusion Node* kann man eine Fusionsmenge von Knoten bilden. Das *Port Node* Kommando wandelt einen Knoten in eine Port-Stelle bzw. Port-Transition um, während *Port Assignment* eine Zuordnung von Socket- zu Port-Knoten bildet.

Syntax Check erlaubt es dem Benutzer, die Syntax der Beschriftung eines SNL-Modells zu überprüfen. *Syntax Messages* zeigt ein Fenster mit den Fehlermeldungen eines Syntaxchecks des aktuellen Diagrammes.

Mit *Sim Regions...* können Beschriftungsteile des Netzes in ML evaluiert werden.

Das **Aux** Menue stellt dem Anwender eine Reihe graphischer Hilfsobjekte zur Verfügung, die er zu Dokumentations- und Annotationszwecken einem SNL-Modell hinzufügen kann. Diese Objekte haben keinerlei semantische Bedeutung für das SNL-Modell. Außerdem enthält das **Aux** Menue einige Hilfskommandos. Das **Aux** Menue hat die folgende Gestalt:



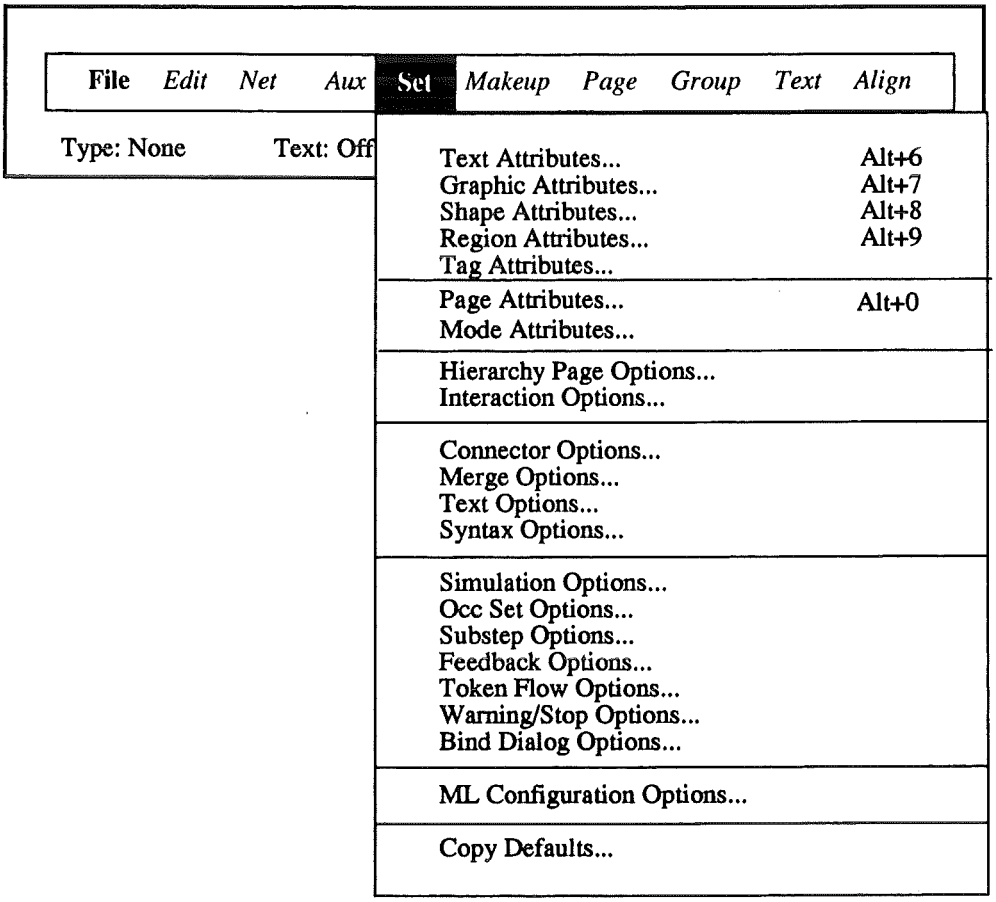
Über die Menuepunkte *Connector, Box, RoundedBox, Ellipse, Polygon, RegularPolygon, Wedge, Line, Label* und *Region...* kann der Anwender Konnektoren, Boxen, Boxen mit gerundeten Ecken, Ellipsen, Polygone, reguläre Polygone, Ellipsensegmente, Geraden, Textzeilen oder Regionen als Hilfsobjekte in ein SNL-Modell einfügen.

Das Kommando *Make Region* wandelt ein beliebiges Objekt (außer eine Region) in eine Region um. Das Kommando *Make Node* wandelt dagegen ein beliebiges Objekt in einen Knoten um.

Der Menuepunkt *Convert to Net* konvertiert ein Hilfsobjekt in ein semantisches Netz-Objekt (Stelle oder Transition). Auf diese Weise ist es möglich, eine Stelle oder Transition in einer beliebigen durch ein Hilfsobjekt definierten Form darzustellen. Durch *Convert to Aux* wird entsprechend ein Netz-Objekt in ein Hilfsobjekt umgewandelt.

Mit *Start ML* kann man den ML-Interpreter vom Editor aus starten. Mit *ML Evaluate* kann man einen beliebigen selektierten ML-Code im ML-Interpreter evaluieren. *ML Stream* liefert einem einen terminalorientierten Zugang zum ML-Interpreter.

Das **Set** Menue enthält Kommandos, mit denen man Defaultwerte der Applikation verändern kann. Es hat die folgende Gestalt:



Über das Kommando *Text Attributes...* kann man das Aussehen textueller Objekte des Systems beeinflussen, während der Menuepunkt *Graphic Attributes...* das Aussehen graphischer Objekte bestimmt. Mit *Shape Attributes* kann man die Defaultgröße von Objekten einstellen und über *Region Attributes...* die Attribute von Regionen beeinflussen. Über *Tag Attributes...* sind die Tag-Attribute von Beschriftungselementen einstellbar.

Über das Kommando *Page Attributes* lassen sich die Attribute der Diagrammseiten einstellen. Das Kommando *Mode Attributes* beeinflußt verschiedene semantische Attribute einer Seite als Modul bzw. Netz-Klasse während einer Simulation oder Erreichbarkeitsanalyse.

Die Attribute der *Hierarchy Page Options...* bestimmen das Layout der Hierarchieseite, die die Beziehungen der Seiten eines Netz-Modells untereinander anzeigt. *Interaction Options...* bestimmt die Interaktion des Systems mit dem Benutzer während einer Simulation des SNL-Modells.

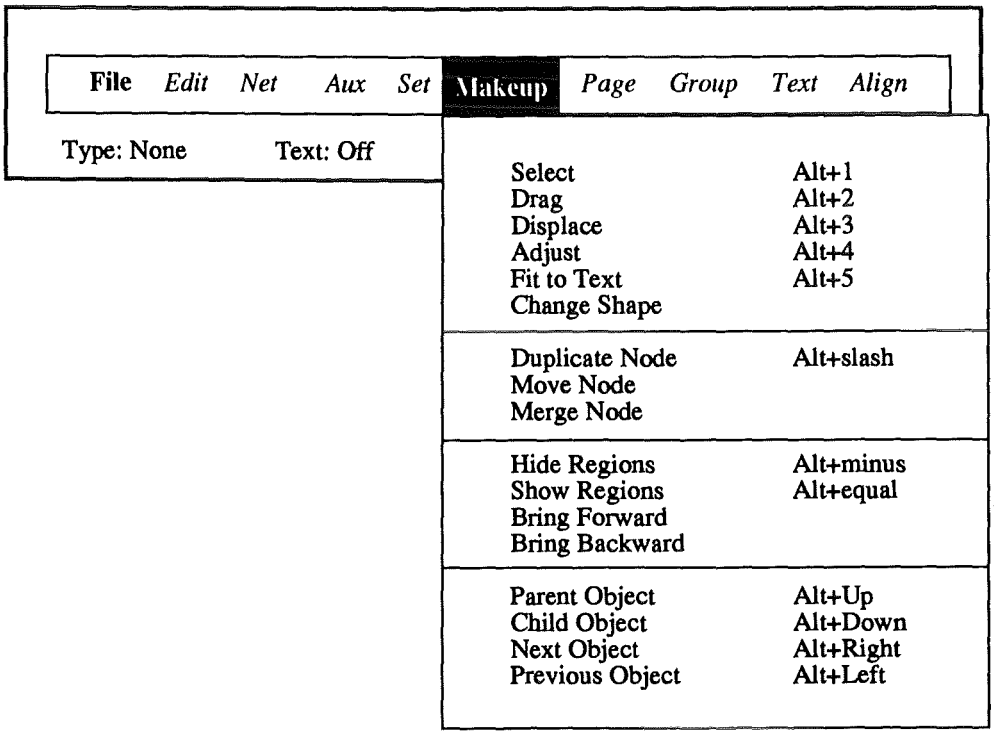
Eine weitere Gruppe von Menuepunkten im **Set** Menue beeinflußt Optionen während der Simulation und Erreichbarkeitsanalyse eines Petri-Netz-Modells. Die generellen Optionen für eine Simulation lassen sich über *Simulation Options...* einstellen. *Occ Set Options* bestimmt, welche Formen von Schaltvorgängen bei einer Simulation oder Erreichbarkeitsanalyse berücksichtigt werden. Hier kann man z.B. festlegen, ob nur das Schalten einzelner

Transitionen oder auch das Schalten von Schritten erlaubt sein soll. *Substep Options...* legen bei der Simulation fest, wann die Simulation anhält (Breakpoints) oder welche semantischen Objekte während einer Simulation gezeigt werden. *Feedback Options...* bestimmen das Feedback des Systems während der Simulation.

ML Configuration Options beschreiben die Informationen, die der Editor, der Simulator oder die Erreichbarkeitsanalyseeinheit benötigen, um den ML-Interpreter über das Netzwerk ansprechen zu können.

Über den Menuepunkt *Copy Defaults* kann man eingestellte Defaultwerte als system- oder modellweite Defaults speichern.

Das **Makeup** Menue enthält Kommandos, die verschiedene Operationen auf Objekten durchführen. Es enthält im einzelnen die folgenden Menuepunkte:



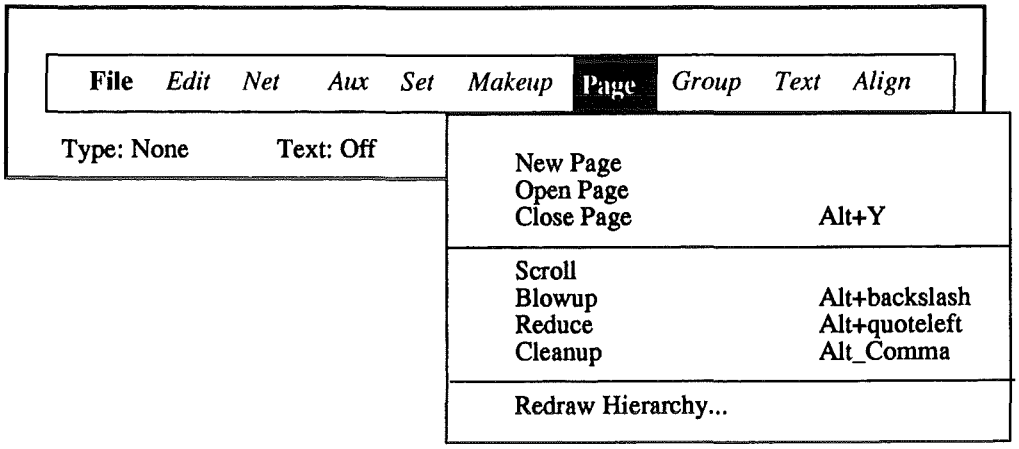
Der Menuepunkt *Select* ermöglicht die Selektion von Objekten. *Drag* erlaubt die Bewegung von Objekten, während *Displace* ein Objekt um eine voreingestellte Distanz bewegt. Mit *Adjust* kann man die Größe eines Objektes beliebig verändern, während *Fit to Text* ein Objekt genau so groß macht, das es einen darin enthaltenen Text umfaßt. Mit *Change Shape* kann man den graphischen Typ eines Objektes (Box, Ellipse, usw) verändern, wobei die semantische Funktion des Objektes erhalten bleibt. Damit kann man z.B. eine Stelle graphisch als Box darstellen.

Mit *Duplicat Node, Move Node* oder *Merge Node* kann man Objekte duplizieren, bewegen oder zusammenfassen.

Durch *Hide Regions* kann man eine Region unsichtbar machen und durch *Show Region* wieder zum Vorschein bringen. *Bring Forward* und *Bring Backward* verändern die Lage von Objekten in vertikaler Richtung. Das erste Kommando bringt ein selektiertes Objekt in den Vordergrund in Bezug auf Objekte, die dieses vorher teilweise verdecken, während das zweite Kommando das selektierte Objekt in den Hintergrund bringt.

Objekte werden im Editor in bestimmten Beziehungen zueinander gehalten. Regionen sind z.B. anderen Objekten (den Vätern) als Kinder in einer Hierarchie zugeordnet. Weiter sind Objekte gemäß der Reihenfolge, in der sie erzeugt werden, geordnet. Mit *Parent Object* selektiert man zu einem vorher selektierten Objekt ein mögliches Vaterobjekt. *Child Object* selektiert zu einem Vaterobjekt umgekehrt die Kinderobjekte. Mit *Next Object* selektiert man das auf das aktuelle Objekt folgende Objekt bzgl. der Erzeugungsreihenfolge, während man mit *Previous Object* das vorhergehende Objekt selektiert.

Das **Page** Menue faßt alle Aktivitäten in Bezug auf die Seiten eines SNL-Modells zusammen.

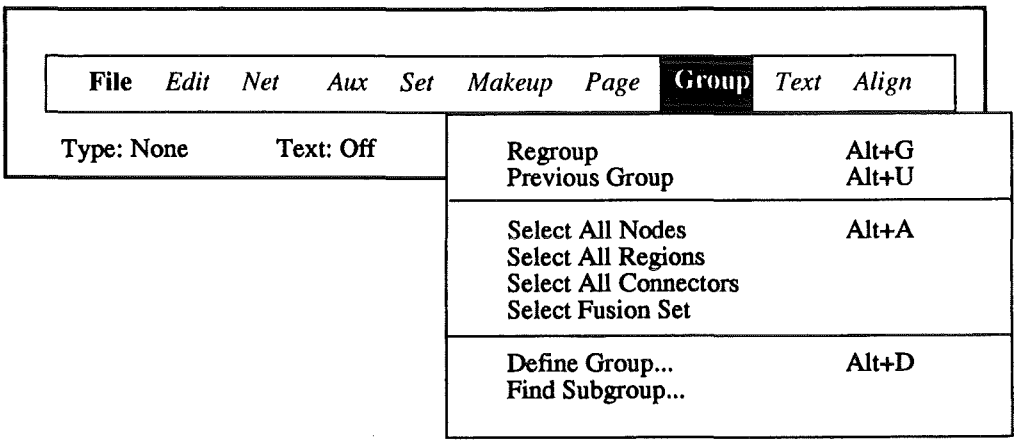


Über *New Page* öffnet man eine neue Diagrammseite, während man über *Open Page* eine bereits vorhandene geschlossene Seite öffnet. Mit *Close Page* schließt man entsprechend eine Seite.

Scroll erlaubt das Scrollen des Inhalts einer Seite, so daß prinzipiell der Inhalt einer Seite größer als das die Seite darstellende Fenster sein kann. Mit *Blowup* vergrößert man den Inhalt einer Seite bzw. mit *Reduce* verkleinert man den Inhalt einer Seite um einen vorgegebenen Faktor. *Cleanup* löscht den Inhalt einer Seite.

Das *Redraw Hierarchy* Kommando führt ein automatisches Layout der Hierarchie-Seite durch.

Das **Group** Menue faßt alle Kommandos zusammen, die sich auf die Behandlung von Gruppen von Objekten beziehen.

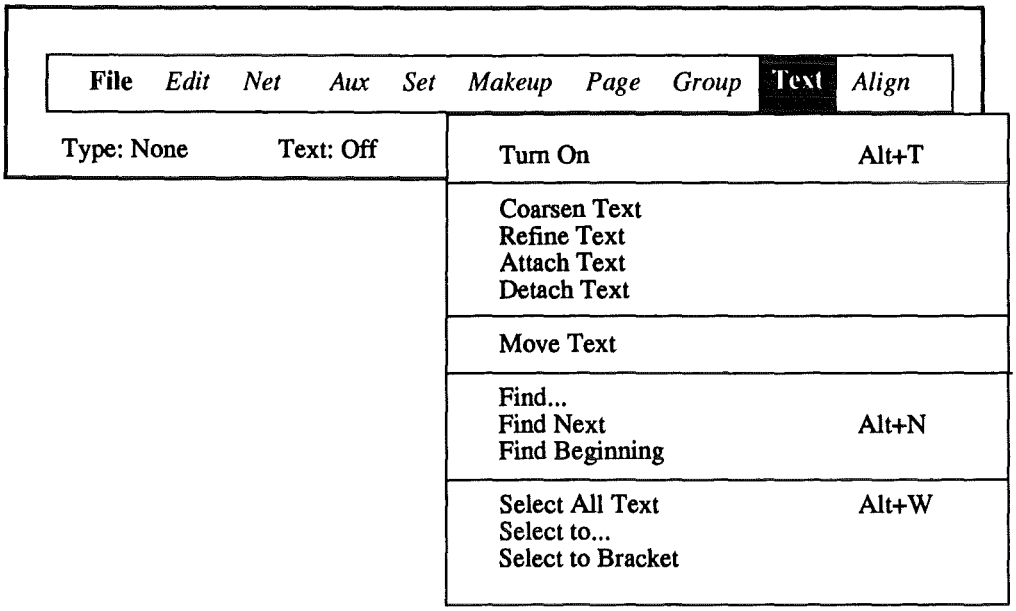


Regroup selektiert aufs neue die zuletzt existierende Gruppe von Objekten, während man mit *Previous Group* die vorhergehende Gruppe von Objekten selektiert.

Mit *Select All Nodes* selektiert man alle Knoten einer Seite, mit *Select All Regions* alle Regionen einer Seite, mit *Select All Connectors* alle Konnektoren einer Seite und mit *Select Fusion Set* alle Elemente einer Fusionsmenge.

Mit *Define Group* kann man interaktiv Objekte zu einer Gruppe zusammenfassen, während man mit *Find Subgroup* interaktiv Elemente aus einer Gruppe zu einer neuen Gruppe zusammenfassen kann.

Das **Text** Menue faßt alle textbezogenen Aktivitäten zusammen.

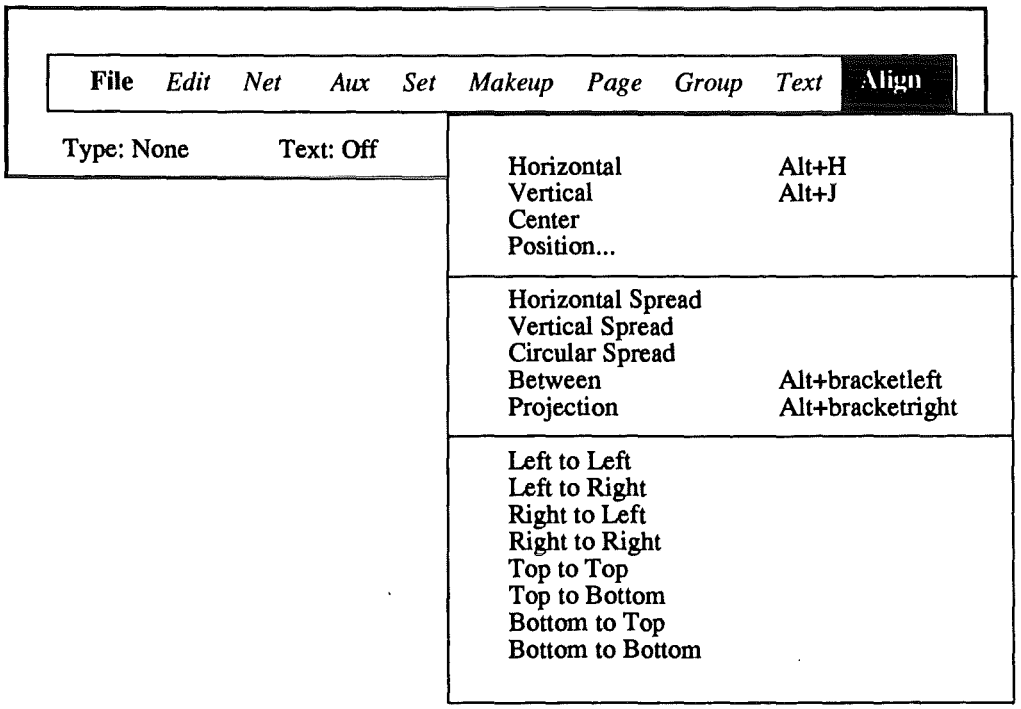


Über *Turn On* schaltet man den Textmode des Editors ein. In diesem können die Textobjekte eines SNL-Modells editiert werden. Der Menuepunkt *Turn On* wird dabei zum Menuepunkt *Turn Off* konvertiert, über den man den Textmode wieder ausschalten kann. Mit dem Menuepunkt *Move Text* kann man Text von einem Objekt zu einem anderen Objekt bewegen.

Find... erlaubt es dem Benutzer nach einem bestimmten Text zu suchen und diesen bei Bedarf durch einen neuen angegebenen Text zu ersetzen. Mit *Find Next* findet man das nächste Vorkommen eines selektierten Textes und mit *Find Beginning* den Anfang eines Textes.

Mit *Select All Text* selektiert man den gesamten Text eines Textobjektes, während man bei *Select to...* angeben kann, bis zu welcher Stelle ein Text selektiert werden soll. *Select to Bracket* selektiert schließlich den Text bis zur nächsten Klammer.

Das **Align** Menue bietet dem Benutzer vielfältige Möglichkeiten, Objekte gegeneinander auszurichten.



Horizontal richtet selekierte Objekte horizontal, *Vertical* richtet Objekte vertikal zueinander aus. Mit *Center* kann man Objekte auf einer Seite zentrieren, während man mit *Position* Objekte durch Eingabe von Koordinaten positionieren kann.

Weiter kann man Objekte durch Menuepunkte im **Align** Menue auf einer Seite verteilen. Mit *Horizontal Spread* verteilt man selektierte Objekte in horizontaler Richtung, mit *Vertical Spread* in vertikaler Richtung und mit *Circular Spread* kreisförmig um einen Punkt. Mit *Between* plaziert man ein Objekt zwischen anderen Objekten, während man mit *Projection* ein Objekt auf eine Gerade projiziert, die durch andere Objekte gegeben ist.

Mit *Left to Left* richtet man zwei Objekte so aus, daß ihre linken Seiten aneinanderliegen, mit *Left to Right* so, daß die linke Seite des ersten Objekts an der rechten Seite des zweiten Objekts liegt. Entsprechend positioniert *Right to Left* die Objekte so, daß die rechte Seite des ersten Objekts an der linken Seite des zweiten Objektes liegt, während bei *Right to Right* die rechten Seiten beider Objekte aneinandergelegt werden. Bei *Top to Top* werden die Objekte so positioniert, daß beide oberen Seiten aneinanderliegen, während bei *Top to Bottom* die obere Seite des ersten Objektes an die untere Seite des zweiten Objektes positioniert wird. Bei *Bottom to Top* wird schließlich die untere Seite des ersten Objekts an die obere Seite des zweiten Objekts positioniert, während bei *Bottom to Bottom* die unteren Seiten aneinandergelegt werden.

6. Vergleich mit anderen Petri-Netz-Modellen und -Werkzeugen

Unser SNL-Modell basiert in seiner Grundidee auf den Prädikat/Transitions-Netzen (PrT-Netze), wie sie von Genrich und Lautenbach erstmalig in [Genrich79] vorgestellt wurden. Neuere Beschreibungen dieses Höheren Petri-Netz-Modells findet man u.a. in [Genrich80e, Genrich81, Genrich83, Genrich86, Genrich89]. Die Definition eines Prädikat/Transitions-Netzes, wie sie in den angegebenen Artikeln vorgestellt wird, trennt zwar die Syntax eines Netzes von ihrer Semantik, da die Syntax des Netzes innerhalb eines prädikatenlogischen Kalküls beschrieben wird, andererseits wird die Semantik eines Netzes aber nicht ebenfalls formal in diesem prädikatenlogischen Kalkül beschrieben, sondern innerhalb des Netz-Modells schon durch eine Struktur (in unserer Sprache ein algebraisches System) angegeben, die die rein syntaktischen Elemente realisiert. Ein Nachteil dabei ist, daß die "statische" Semantik des Netzes zwar schon durch die angegebene Struktur festgelegt ist, aber die zugehörigen Beschriftungselemente, wie die Formeln auf den Transitionen erst noch bzgl. der Struktur ausgewertet werden müssen, um ihre Wirkung auf Schaltvorgänge z.B. in Werkzeugen berücksichtigen zu können.

Durch eine Interpretation der Beschriftungselemente eines PrT-Netzes bzgl. der zugehörigen Struktur, die unabhängig von der Netzmarkierung vor Ablauf des Netzes möglich ist (Genrich und Lautenbach bezeichnen dies als "statische" Semantik des Netzes), kann man ein PrT-Netz in ein weiteres Höheres Petri-Netz-Modell (ein sogenanntes Coloured Petri-Netz) transformieren, wie es parallel zu dem PrT-Netz-Modell von Jensen entwickelt wurde und u.a. in [Jensen81a, Jensen81b, Jensen86]) beschrieben wird. Das Coloured Petri-Netz-Modell ist dabei eine rein semantische Netz-Modellbeschreibung in dem Sinne, daß die Beschriftungen hier gleich über semantische Konstrukte, wie Mengen und Funktionen, definiert werden. Coloured Petri Netze haben daher den Vorteil, daß sie konkret sind und nicht die Trennung zwischen der syntaktischen Beschreibung und ihrer Realisation bzgl. eines algebraischen Systems aufweisen, wie dies bei PrT-Netzen der Fall ist, und von daher in vielen Fällen leichter zu handhaben sind. So gibt man in Coloured Petri-Netz-Werkzeugen, wie Design/CPN (siehe [CPN90]), die an einer Netzbeschriftung beteiligten Mengen und Funktionen in der Regel direkt als Typdefinitionen und Funktionsdefinitionen innerhalb einer interaktiven Computersprache, wie ML, an. Diese Beschreibung ist sehr konkret, hat aber gerade aus diesem Grund den Nachteil, daß ihre Semantik nicht in einem formalen Kalkül spezifiziert wird, sondern über die Implementierung ihrer Objekte, wie ML-Typen und ML-Funktionen, wobei die Wirkung dieser Objekte nur informell spezifiziert wird.

Eine weitere Klasse von Netzen, für die typische Vertreter die auf der objektorientierten Sprache Smalltalk basierenden Netz-Modelle, wie sie in [Dähler87] vorgestellt werden, sind, basieren ebenfalls ihrer Konzeption nach auf dem PrT-Modell, sind jedoch stark in ihrer Arbeitsweise an ihre auf Smalltalk-Klassen basierende Implementierung, wie z.B. durch das PACE Tool realisiert ist, ausgerichtet. Der Einsatz von Smalltalk liefert hier sehr mächtige, aber auch sehr von ihrer Implementierung in Smalltalk abhängige Netz-Modelle, deren theoretische Eigenschaften nur noch bedingt über eine Theorie, wie die der PrT-Netze beschrieben werden kann, was uns problematisch erscheint.

Die oben beschriebenen Vor- und Nachteile der Definition von PrT-Netzen und Coloured Petri-Netzen führte zu einer Reihe von Artikeln, in der Netz-Modelle vorgestellt wurden, die eine formale Beschreibung der Syntax und Semantik von Netz-Modellen in einem PrT-Netz-ähnlichen Kalkül unter Verwendung der abstrakten Spezifikation von Datentypen unterstützen, während die Realisation dieser Netz-Modelle in der Regel über die in der Praxis besser handhabbaren Coloured Petri Netze erfolgte. Auf diese Weise konnte man die Stärken sowohl des PrT-Netz-Modells als auch des Coloured Petri-Netz-Modells kombinieren. Hier seien insbesondere die Arbeiten von Memmi und Vautherin ([Memmi86, Memmi87, Vautherin86, Vautherin87]) erwähnt, in denen ein Netz zunächst rein syntaktisch und formal-semantisch in Form eines Σ-Schemas, bestehend aus einer zu Grunde liegenden algebraischen Spezifikation, einem Petri-Netz und einer auf Termen der algebraischen Spezifikation aufbauenden Beschriftung beschrieben und dann bzgl. einer Realisierung der Spezifikation durch ein algebraisches System als Coloured Petri-Netz interpretiert wird.

Während beim Ansatz von Vautherin und Memmi zur Realisation eines Σ-Schemas beliebige Modelle der zugehörigen Spezifikation zugelassen sind, so daß ein Σ-Schema in der Regel eine ganze Klasse möglicher Realisierungen als Coloured Petri-Netz besitzt, wobei diese Realisationen in der Regel durch eine manuelle Transformation der Spezifikation in ein zugehöriges Modell gewonnen werden, beschränken sich andere Ansätze, wie z.B. von Reisig in [Reisig91] oder das Konzept des PrE-Netz-Modells im Esprit Projekt GRASPIN (siehe z.B. [Krämer85, Schmidt89]) auf eine Realisation der Netz-Spezifikationen durch ein zu der algebraischen Spezifikation gehöriges initiales Modell. Dieser Ansatz hat den Vorteil, daß man prinzipiell in der Lage ist, das initiale Modell weitgehenst automatisch mit einem computergestützten System z.B. durch Term-Rewriting (siehe hier z.B. [Gerlach88, Fonio87]) in abstrakter Form realisieren zu können, während man im anderen Fall in der Regel die automatische Realisation von Spezifikationen durch Angabe von Konstruktionsregeln und darauf

aufbauenden Transformationsregeln, wie es auch für unsere SNL-Modelle geplant ist, unterstützen muß (Man siehe hier z.B. [CIP85, CIP87]). Der Ansatz über das initiale Modell hat aber auch den gewaltigen Nachteil, daß die Interpretation von Spezifikationen in rein symbolischer Form über Term-Rewriting Systeme, wie die Erfahrung im GRASPIN Projekt gezeigt hat (siehe [Schmidt89]), sehr zeitintensiv ist und in Petri-Netz-Werkzeugen zu Performanceproblemen führt. Wir folgen daher mit unserem Konzept der SNL-Modelle und ihrer Realisation im SMARAGD-System im wesentlichen dem Ansatz von Memmi und Vautherin, wobei wir allerdings einige wichtige Änderungen bzgl. der Interpretation der Semantik von Netz-Spezifikationen vorgenommen haben, die wir etwas später in diesem Abschnitt noch einmal zusammenfassend erörtern wollen.

Die einzelnen Varianten der PrT-Netz- und Coloured Petri-Netzformen unterscheiden sich (teilweise nur geringfügig) bzgl. der Form von Beschriftungen, die sie einschränkend bzw. ergänzend zu diesen zulassen. Das ursprüngliche PrT-Netz-Modell wurde z.B. in dem Sinne erweitert, daß Kantenbeschriftungen in der ursprünglichen Definition zunächst nur Multimengen von Tupeln von Variablen mit Koeffizienten 1, dann mit beliebigen Koeffizienten, dann mit Tupeln von Termen und schließlich mit durch boolesche Bedingungen darstellenden Koeffizienten zugeordnet waren. In anderen Ablegern der PrT-Netze, hierzu zählen auch die meisten auf abstrakte Datentypspezifikation basierenden Formen, sind die Kantenbeschriftungen nur Terme bzw. Multisummen von Termen, was äquivalent zur Verwendung von Tupeln von Termen ist, da Tupel ja als spezielle Datentypen spezifiziert werden können und insbesondere auch Multimengen als abstrakte Datentypdefinitionen definiert werden können.

Coloured Petri-Netze sind im Hinblick auf die Kantenbeschriftung in der Hinsicht eleganter, daß sie beliebige Funktionen zwischen Multimengen zulassen, so daß die recht gekünstelte Konstruktion von Multimengen von Termen mit booleschen Bedingungen als Koeffizienten bei ihnen überhaupt nicht nötig ist, um z.B. je nach Schaltmode eine unterschiedliche Anzahl von Marken über eine Kante fließen zu lassen. Dies gilt entprechend für Netze, die direkt auf einer (objektorientierten oder funktionalen) Computersprache aufsetzen. Es ist allerdings schwierig, solche Funktionen über Multimengen an den Kanten zuzulassen, wenn man z.B. im Sinne von Vautherin Realisierung und Spezifikation eines Netzes trennen will, da die Semantik solcher Funktionen nicht innerhalb eines logischen Kalküls erster Ordnung unabhängig von den speziell gewählten Multimengen beschrieben werden kann. Unser Netz-Modell liegt in Bezug auf die Kantenbeschriftung in Form von zunächst formalen Summen über Termen bei dem, was als Optimum hier für PrT-ähnliche Netze bislang vorgestellt wurde, weist aber leider nicht den Komfort auf, den man durch Funktionen, wie sie im Coloured Petri-Netz-Modell verwendet werden, erreichen kann (Man sehe sich hierzu auch die Bemerkungen zu diesem Thema im Ausblick an).

Stellen werden im ursprünglichen PrT-Netz-Modell mit "variablen" Prädikaten beschriftet, wobei der Begriff "variables" Prädikat hier unseres Erachtens ein bißchen redundant ist, da ein solches Prädikat in der Form, wie es bei Genrich und Lautenbach verwendet wird, lediglich eine informelle natürlichsprachliche Erläuterung der Bedeutung einer Stelle selbst ist, d.h. unserer Meinung nach ist eine Stelle selbst das formale Objekt, dem in der Interpretation des Netzes stets die "variable" Relation, beschrieben durch die aktuellen Marken auf der Stelle, zugewiesen wird. Die gleiche erklärende Funktion kann damit auch der Stellenname übernehmen, der üblicherweise einer Stelle zugeordnet wird. Sinnvoller ist es hier, wie dies bei fast allen Netz-Modellen, die auf Spezifikation aufbauen, üblich ist, Stellen Sorten, Tupel von Sorten oder dergleichen zuzuordnen, die bestimmen, welche Art von Objekten auf einer Stelle liegen können. Eine solche Erweiterung wird auch für neue Versionen des PrT-Netzbegriffes (z.B. in [Genrich89]) vorgestellt.

In der Realisation beschreiben solche Sortenangaben dann Mengen zugehöriger Objekte und entsprechen damit in der Realisierung den Colourmengen, wie sie bei Coloured Petri-Netz-ähnlichen Netz-Modellen üblich sind. In computergestützten Werkzeugen sind die Sorten dann in der Realisation als Datentypen mit zugehörigen Wertemengen codiert. Dies ist insbesondere bei den auf objektorientierten und funktionalen Sprachen basierenden Höheren Netz-Modellen per Definition schon der Fall. Wir haben gegenüber der üblichen reinen Sortenangabe die Angabe der Objekte, die auf Stellen liegen dürfen noch in der Weise erweitert, daß solche Objektmengen neben der Sortenangabe zusätzlich durch prädikatenlogische Formeln eingeschränkt werden können. Durch geeignete Formeln auf den Transitionen könnte man prinzipiell die gleiche einschränkende Wirkung erzielen. Man beachte hierbei jedoch, daß solche Formeln dann auf allen Transitionen, die Objekte auf ein und der gleichen Stelle ablegen, dupliziert werden müßten und weiter die Anfangsmarkierung nicht direkt darauf überprüft werden kann, ob sie ebenfalls dieser Einschränkung genügt, da man hierzu die Transitionsformeln in ihre Anteile bzgl. der Einschränkungsbedingungen von Stellen zerlegen müßte.

Transitionen werden in PrT-ähnlichen Netz-Modellen in der Regel mit Formeln einer prädikatenlogischen Sprache 1. Ordnung beschriftet, wobei man sich hier in spezifikationsorientierten Netz-Modellen von vorne herein auf gleichungsorientierte Formeln (also insbesondere Formeln ohne Existenzquantoren) beschränkt, da hier die Theorie algebraischer Spezifikationen einfacher ist, während das theoretische PrT-Netz-Modell hier beliebige

Formeln 1. Ordnung zuläßt. Reale Implementierungen von PrT-Netz-ähnlichen Werkzeugen beschränken sich aber in der Praxis ebenfalls auf gleichungsorientierte Formeln im Sinne boolescher Ausdrücke, da insbesondere Existenzquantoren für beliebige Datentypen nicht allgemein ausgewertet werden können. Coloured Petri-Netz-Modelle benötigen gemäß ihrer theoretischen Darstellung keine Formeln als Beschriftung von Transitionen, da man hier die möglichen Schaltmodi einer Transition direkt in Form einer Colourmenge angibt. In der praktischen Realisierung verwendet man aber in der Regel auch bei ihnen boolesche Formeln auf den Transitionen, da die Angabe einer Colourmenge für Transitionen in Form eines Datentyps vielfach zu unflexibel ist, um auch Teilmengen von Standarddatentypen als Colourmenge einer Transition zu deklarieren. Dies kann man gerade durch Einführen von booleschen Formeln. Auf Grund unserer neuartigen, dynamischen Interpretation der Semantik der Petri-Netzbeschriftungen sind wir in der Lage, sowohl theoretisch als auch praktisch beliebige prädikatenlogische Formeln 1. Ordnung als Beschriftung von Transitionen in SNL-Modellen zuzulassen.

Kommen wir zu dem Vergleich der Schaltregel. Mit Ausnahme unseres SNL-Modells, können die bei allen uns bekannten PrT-Netz-ähnlichen Netz-Modellen in einer Transitionsformel und den umliegenden Kantenbeschriftungen vorkommenden freien Variablen zunächst einmal durch beliebige Werte der ihnen (gegebenenfalls durch eine Realisation) zugeordneten Wertebereiche, die unabhängig vom Systemzustand sind, ersetzt werden, wobei die Transitionsformel als eine Vorbedingung für das Schalten einer Transition bzgl. einer solchen Variablenauswertung zu wahr ausgewertet werden muß. Hat man nun eine solche Variablenauswertung, die die Transitionsformel erfüllt, so werden durch Einsetzen dieser Variablenwerte in die Kantenbeschriftungen in der Umgebung der Transition die Multimengen von Objekten bestimmt, die von Eingabestellen abzuziehen bzw. auf Ausgabestellen hinzuzufügen sind. Als weitere Bedingung für einen Schaltvorgang müssen nun genügend solcher Objekte auf Eingabestellen vorhanden sein, und eine eventuell vorhandene Kapazitätsbeschränkung darf beim Drauflegen der Objekte auf Ausgabestellen nicht überschritten werden. Sind alle diese Bedingungen erfüllt, so kann eine Transition in dem Mode, der durch die Variablensubstitution gegeben ist, schalten, wobei die entsprechenden Objekte von den Eingabestellen abgezogen bzw. auf den Ausgabestellen hinzugefügt werden.

Diese Vorgehensweise führt in der Praxis sofort zu Problemen, wenn die beteiligten Wertebereiche nicht beschränkt sind. Dann würde man nämlich zunächst eine unbeschränkte Anzahl von Variablensubstitutionen darauf überprüfen müssen, ob für sie die Randbedingungen zum Schalten erfüllt sind, was in endlicher Zeit und mit endlichen Resourcen praktisch nicht möglich ist. Dies führte dazu, daß bei den praktischen Realisierungen solcher Modelle aus theoretischer Sicht recht willkürliche Einschränkungen der Beschriftung konkreter Netz-Modelle verlangt werden, die darauf hinauslaufen, daß man auf Grundlage z.B. der Objekte auf den Eingabestellen und spezieller Beschriftungen an den Kanten für jede Transition schon über die Randbedingung, daß genügend Objekte der richtigen Art auf den Eingabestellen vorhanden sein müssen, auf die möglichen Variablensubstitutionen schließen kann. Dies ist u.a. bei den in GRASPIN (siehe [Schmidt89) verwendeten PrE-Systemen der Fall. Andere Systeme unterstützen aus dem obigen Grund nur Simulation, da bei der Simulation scheinbar das Problem umgangen werden kann, da nicht alle möglichen Fälle auch simuliert werden müssen. Bei Coloured Petri Netzen treten die obigen Probleme aus theoretischer Sicht nicht auf, solange sich ein Anwender bei der Angabe der Colourmengen von Transitionen auf endliche Mengen mit möglichst wenigen Elementen beschränkt. In einem Werkzeug hat man jedoch auch hier ein Problem damit, alle möglichen Werte einer Colourmenge in Form einer geordneten Datenstruktur so anzuordnen, daß sie z.B. bei einem Analysealgorithmus sequentiell durchgegangen werden kann, wobei ja dieser Algorithmus für alle möglichen Arten von Colourmengenangaben funktionieren muß. Hier kann man sich natürlich von vorneherein auf bereits geordnete Mengen beschränken.

Unsere Interpretation der Netz-Beschriftungen einer SNL-Netz-Spezifikation und unsere Form der Schaltregel unterscheidet sich in einem wesentlichen Punkt von der oben beschriebenen Vorgehensweise. Diese Änderung bzgl. der Semantik von SNL-Modellen resultierte aus der Frage, wie sinnvoll in einem PrT-Netz-Modell die prädikatenlogischen Formeln auf den Transitionen und die Multisummen von Tupeln von Termen an den Kanten sind, wenn diese auf Grund ihrer "statischen" Struktur, d.h. auf Grund ihrer Unabhängigkeit vom Systemzustand und damit von der Markierung eines PrT-Systems, schon vor Ablauf des Netzes (also vor der Betrachtung der dynamischen Eigenschaften des Netzes) endgültig ausgewertet werden können. Diese einmalige Auswertung der "statischen" Netzbeschriftungen würde dann Transitionen Mengen von potentiell gültigen Auswertungen von Variablen (die, die die Formel der Transition erfüllen), im Sinne eines Coloured Petri-Netzes als Colourmengen, und Kanten Funktionen, die im Sinne einer Schrittsemantik über Multimengen solcher Transitionen zugeordneter Auswertungen von Variablen als Argumente haben und als Ergebnis Multimengen von Tupeln von Objekten liefern, die von einer zugehörigen Stelle abzuziehen bzw. auf diese Stelle zu legen sind, zuordnen. Führt man eine solche Realisation der "statischen" Beschriftungselemente eine PrT-Systems vor Ablauf des Systems durch, so ist man anschließend von dieser Interpretation der Beschriftungselemente befreit und kann sich bei der Betrachtung des dynamischen Ablaufs des Netzes ganz auf die dynamische Struktur konzentrieren. Dies ist wohl auch der Grund, warum Vautherin und Memmi ihre Σ-Schema direkt als Coloured Petri-Netz-Modelle realisieren.

Obwohl die oben beschriebene Sicht eines PrT-Systems ihrer Definition in den erwähnten Artikeln entspricht, glauben wir allerdings, daß die Intuition hinter dem PrT-Netz-Modell eigentlich eine etwas andere Interpretation der Semantik der Netz-Beschriftungen nahelegt. So führen Genrich und Lautenbach in ihren Artikeln neben den "statischen" Symbolen, die vor Ablauf des Netzes ausgewertet werden können, noch "dynamische" Prädikate ein, die den Stellen zugeordnet sind und den dynamischen Zustand eines Netzes beschreiben sollen. Die dynamischen Prädikate stehen dabei für Relationen, die in jedem Zustand des Systems genau die Markentupel einer zugehörigen Stelle beinhalten, die im aktuellen Markierungszustand auf dieser Stelle liegen und damit das "dynamische" Prädikat erfüllen.

Wir finden es nun naheliegend, auch den anderen Netzbeschriftungen eine solche dynamische Interpretation zu geben. Wenn wir der obigen Philosophie folgen, daß der Zustand eines PrT-Systems alleinig durch die durch die aktuelle Markierung gegebene Realisation der dynamischen Prädikate beschrieben wird, so sollte auch die Zustandsänderung eines solchen Systems von der durch die aktuelle Markierung gegebenen Realisation der dynamischen Prädikate dynamisch abhängen. Damit sollte also das "Ob" und das "Wie" eine Transition schaltet von der Interpretation der aktuellen Markierung abhängen. Dies ist weder im PrT-Netz-Modell noch im Coloured Petri-Netz-Modell oder einer ihrer Ableger der Fall. Wenn wir weiter im Sinne der Bedingungs/Ereignis-Netze noch fordern, daß ein möglicher Zustandsübergang, der durch Schalten einer Transition verursacht wird, nur auf Grund des aktuellen Zustandes des Systems in der lokalen Umgebung der Transition entscheidbar sein soll, so erhalten wir als Grundlage für die Schaltregel eine Art Lokalitätsprinzip, wie wir es formuliert haben. Auf Grundlage dieses Prinzips interpretieren wir Variablen nur in Bezug auf den Inhalt der Stellen in der Umgebung einer Transition, d.h. in Bezug auf den Wert der "dynamischen" Prädikate, die solchen Stellen zugeordnet sind. Hiermit vermeiden wir alle die Probleme, die wir oben in Bezug auf die Schaltregel in anderen Höheren Petri-Netzformen angesprochen haben. Um alle gemäß unserer Schaltregel möglichen Variablenauswertungen zu erhalten, brauchen wir nur die Mengen aller Objekte von der zu einer Variablen gehörigen Sorte zu bilden, die innerhalb der Umgebung einer Transition auf Stellen liegen, die dieser Variablen als Speicher für mögliche Werte zugeordnet sind. Diese Menge ist stets endlich, da die Stellen endliche Kapazitäten haben, und da die Markierungen über Multimengen beschrieben sind, die intern in einem Werkzeug bereits als Listen ohne Duplikate beschrieben sind, hat man automatisch eine sequentielle Anordnung dieser Werte, um allgemeingültige Algorithmen zur Analyse oder Simulation formulieren zu können.

Das sich unsere Interpretation der Beschriftungen unserer PrT-Netz-ähnlichen SNL-Netz-Spezifikationen wesentlich von der statischen Interpretation, wie sie in anderen Höheren Netzformen zu finden ist, unterscheidet, ersieht man aus den folgenden Bemerkungen. Während z.B. eine Hauptaussage bzgl. von PrT- und auch Coloured Petri-Netzen der Satz ist, daß man so ein Netz stets durch Entfalten in ein äquivalentes Bedingungs/Ereignis-Netz umwandeln kann, läßt sich diese Aussage nicht so einfach auf SNL-Modelle übertragen. Dies liegt daran, daß ja die Semantik der Transitionsformeln und Kantenbeschriftungen in unserer Netzform abhängig von der Markierung ist und somit ein Verfahren zur Entfaltung des Netzes die Zerlegung z.B. der Transitionen in Abhängigkeit von der Markierung durchführen müßte, was bei einer großen Anzahl von Markierungen schwierig ist. Unsere Interpretation der Netzbeschriftungen erlaubt es uns weiter, beliebige auch mit Existenz- und Allquantoren versehene Formeln als Schaltbedingungen für Transitionen auszuwerten. Interessant ist in Bezug auf unsere Schaltregel daher der Begriff einer Kontaktsituation. Hierunter versteht man im allgemeinen eine Situation im Netz, in der eine Transition zwar in Hinblick auf den Zustand ihrer Vorstellen schalten könnte, aber der Zustand einer Nachstelle dies verhindert. In unserem SNL-Modell kann nun die Auswertung der Variablen der Transitionsformel schon von Objekten abhängen, die auf Nachstellen liegen. Dadurch sind "Kontaktsituationen" möglich, die ein Schalten einer Transition aus dem Grund verhindern, daß Objekte auf den Nachstellen einer Transition die Auswertung der zugehörigen Transitionsformel zu wahr unmöglich machen.

Für die Modellierung großer Netz-Modelle sehen wir ein gutes Konzept zur Modularisierung von solchen Netz-Modellen als unbedingt notwendig an. In der Literatur finden sich hierzu einige Ansätze für eine solche Modularisierung, wobei Jensen's Ansatz für Hierarchische Coloured Petri-Netze (siehe z.B. in [Huber89]) am weitesten geht, da er sowohl Konzepte zur Substitution von Stellen durch transitionsberandete Unternetze, Substitution von Transitionen durch stellenberandete Unternetze und sogenannte Aufruftransitionen enthält. Wir haben daher unsere Konzepte Netz-Klasse und Netz-Module an den Konzepten der Substitutionsstellen (im Fall einer Netz-Klasse) bzw. Substitutionstransitionen (im Fall eines Netz-Moduls) von Jensen angelehnt, diese aber in einigen Schwachpunkten stark erweitert. Ein wesentliches Manko der Modulkonzepte, wie sie von Huber, Jensen und Shapiro in [Huber89] vorgestellt werden, ist das Fehlen einer Möglichkeit, Unternetze zu parametrisieren. Weiter gibt es für Hierarchische Coloured Petri-Netze, wie sie vom Design/CPN Werkzeug unterstützt werden, nur eingeschränkte Möglichkeiten der Parameterübergabe zwischen Socket- und Portknoten im Fall von Netz-Klassen (siehe hierzu [CPN90]). Eine Parametrisierung von Netz-Klassen bzw. Netz-Modulen, wie wir sie in dieser Arbeit vorgestellt haben, halten wir aber insbesondere in Bezug auf die Verwendung externer Bibliotheken all-

gemein verwendbarer Netz-Klassen bzw. Netz-Module für unbedingt erforderlich. Andernfalls sähen wir keine Möglichkeit, die Wiederverwendbarkeit solcher Netz-Module bzw. Netz-Klassen in verschiedenen Netz-Modellen zu ermöglichen.

Höhere Petri-Netz-Modelle werden zur Zeit in Werkzeugen überwiegend durch einen Editor und einen Simulator unterstützt, während in den meisten Fällen eine Analyseeinheit fehlt. Letzteres liegt daran, daß sich die Analyse als eine für viele solcher Modelle noch zu komplexe Aufgabe herausstellt, für die der Implementierungsaufwand und die benötigten Rechnerresourcen einfach zu groß sind. So gibt es auch für das Design/CPN-Werkzeug, das auf dem Coloured Petri-Netz-Modell basiert, zur Zeit nur die Möglichkeit, erstellte Modelle zu simulieren, während eine Analyseeinheit noch fehlt, aber angekündigt ist.

Wir halten dabei eine auf formale Analysemethoden basierende Auswertung der Eigenschaften eines Netz-Modells für unbedingt notwendig, da nur auf diese Weise gewisse Systemeigenschaften des Modells garantiert werden können. Wir sind zuversichtlich, daß auf Grund der neuartigen Form unserer Schaltregel die theoretischen Möglichkeiten einer Erreichbarkeitsanalyse auch für "sinnvolle" Modelle größerer Systeme praktisch in unserem Werkzeug umgesetzt werden können. Hierbei vertreten wir die Philosophie, daß insbesondere in Bezug auf Softwaresysteme in jedem praktisch zu realisierenden System die Anzahl der möglichen Marken im System innerhalb eines Zustandes relativ klein sein sollte, da sie in reellen Systemen stets durch die begrenzten Systemresourcen beschränkt werden.

Auf Grund dieser Annahme können wir dann davon ausgehen, daß in "sinnvollen" SNL-Modellen die Bestimmung der potentiellen Schaltmodi einer Transition auf Grund unserer Schaltregel weitgehenst deterministisch ablaufen wird, d.h. zu einem Zustand des Systems gibt es nur eine geringe Menge verschiedener Variablensubstitutionen. Gleichzeitig sollte bei einem in einem realen Projekt zu realisierenden System auch die Menge der Zustände und der Zustandsübergänge überhaupt, da sie die Komplexität des Systems bestimmt, nicht zu groß sein, denn in einem realen System muß die Implementierung des Systems ja jeden Zustand und Zustandsübergang deterministisch behandeln. Aus diesen Gründen sind wir zuversichtlich, daß Systeme, die nach den oben angedeuteten Prinzipien konzipiert sind, in annehmbarer Zeit durch Aufbau ihres Erreichbarkeitsgraphen analysiert werden können. Eine genaue Beschreibung der Analyseeinheit des SMARAGD-Systems und ein Vergleich der Analysemöglichkeiten des SMARAGD-Systems mit anderen Werkzeugen entnehme man bitte [Süß92]. Hier wird auch auf andere Analysemethoden, wie die Invariantentheorie, eingegangen.

7. Fazit und Ausblick

Die in den vorangegangenen Kapiteln beschriebene Funktionalität der Sprache SNL zur Spezifikation und Modellierung von verteilten Systemen ist in der ersten prototypischen Version des SMARAGD-Systems vollständig implementiert. Dieser erste Prototyp des Werkzeugs enthält neben dem in dieser Arbeit beschriebenen graphischen Editor für SNL-Modelle auch Möglichkeiten zur Simulation sowie zur Analyse von SNL-Modellen. Die Analysefähigkeiten des Systems werden ausführlich in [Süß92] beschrieben.

Auf Grund der im vorherigen Abschnitt beschriebenen Vorzüge des SNL-Netz-Modells sind wir davon überzeugt, daß sich die Konzeption des SMARAGD-Systems auch bei der Spezifikation und Modellierung größerer Systeme, wie wir sie durchführen möchten, bewähren wird. Das System ist allerdings in der momentanen prototypischen Realisierung nicht dazu gedacht, schon als Produktionswerkzeug zur Spezifikation und Modellierung von Systemen in realen Projekten eingesetzt zu werden, sondern soll vielmehr hauptsächlich die Machbarkeit des zu Grunde liegenden theoretischen und konzeptionellen Ansatzes zeigen und diese wurde auch voll bestätigt.

Als nächsten Schritt beabsichtigen wir daher eine Reimplementierung des SMARAGD-Systems mit dem Ziel, ein robusteres auch in realen Projekten einsetzbares System zu erhalten. Hierbei wollen wir die Erfahrungen, die wir durch die bisherige Benutzung des Prototyps gewonnen haben und noch durch einige größere Fallbeispiele vertiefen möchten, einfließen lassen.

Für die erweiterte Version des SMARAGD-Systems erscheint uns ein Redesign des bestehenden Prototyps aus einer Reihe von Gründen sinnvoll. Um zu einer schnellen prototypischen Implementierung für die Machbarkeitsanalyse der Konzeption zu gelangen, basiert der aktuelle Prototyp des SMARAGD-Systems auf dem proprietären Design/OA-Toolkit (siehe [OA90]) und dem Source Code des Design/CPN-Werkzeuges (siehe [CPN90]) der MetaSoftware Corporation. Auf Grund der damit verbundenen Lizensierungsprobleme ist es uns daher nicht möglich, unsere aktuelle SMARAGD-Implementierung an Interessierte frei weiterzuverteilen. Die nächste Version des SMARAGD-Systems soll daher auf Standards und frei verfügbaren Tools aufsetzen, um eine freie Verteilung des Systems zu ermöglichen. Hierzu muß im wesentlichen ein Redesign der Benutzeroberfläche, die auf Design/OA und dem Source Code von Design/CPN basiert, erfolgen, wobei einer der UNIX Quasistandards Motif oder OpenLook berücksichtigt werden soll. Dies wird es uns weiter erlauben, die aus dem Design/OA-Toolkit resultierende, in manchen Situationen umständliche Menuesteuerung des bisherigen Prototyps durch modernere Konzepte basierend auf einem Gemisch von Pull-Down-, Popup-Menues, Dialogboxen, etc, wie sie in Motif oder OpenLook (siehe z.B. [OpenLook90]) vorhanden sind, zu ersetzen.

Da ein großer Teil des bestehenden SMARAGD-Codes zur Handhabung interner Datenstrukturen, wie z.B. des Erreichbarkeitsgraphen, im aktuellen Prototyp bereits in der objektorientierten Sprache C++ geschrieben ist (Wir verwenden hier die LEDA Klassenbibliothek effizienter Datentypen der Universität Saarbrücken, siehe [Näher90]), liegt es nahe, als Implementierungssprache für die zweite Version des SMARAGD-Systems ganz auf C++ umzusteigen. Dies hat auch den Vorteil, daß der durch seinen Umfang schon jetzt recht unübersichtliche Code des aktuellen Prototyps basierend auf den objektorientierten Möglichkeiten von C++ besser strukturiert werden kann und damit die Robustheit der Implementierung zunehmen wird. Ein Umstieg auf C++ bietet weiter den Vorzug, daß der hieraus resultierende Code wesentlich einfacher weiterentwickelt werden kann, da durch das Klassenkonzept einzelne Teile des Codes besser getrennt und durch die Möglichkeiten des Vererbungskonzeptes einfacher erweitert werden kann.

Die im bisherigen Prototyp nur rudimentär vorhandenen Syntax- und Semantikprüfungen sollen in der zweiten Version des Tools so vollständig wie möglich implementiert werden. Während im bisherigen Tool nur die Syntax der Topologie eines SNL-Modells auf Konsistenz überprüft wird, wollen wir in der nächsten Version eine kontextsensitive, inkrementelle Syntaxanalyse der Beschriftungselemente, einschließlich der Deklarationsboxen, eines Netz-Modells im Sinne von syntaxgesteuerten Editoren einführen. Ziel wird es dabei sein, daß ein Netz-Modell gleichzeitig mit dem Abschluß der Editierarbeiten schon syntaktisch korrekt ist.

Eine weitere wichtige Komponente, die dem bisherigen Prototyp fehlt und in die nächste Version eingearbeitet werden soll, ist die schon an verschiedenen Stellen dieser Arbeit erwähnte Transformationseinheit, die den Benutzer bei der Realisation von SNL-Spezifikationen in ML durch eingebaute Transformationsregeln unterstützen soll. Für diese Komponente sind allerdings vorher noch theoretische Untersuchungen notwendig, welche die Einschränkungen der SNL-Semantik bzgl. der Angabe der Semantik von SNL-Spezifikationen bzw. die von einem Modellierer einzuhaltenden Konstruktionsregeln aufzeigen, die nötig sind, damit eine automatische Transformation von SNL-Spezifikationen unter den definierten Randbedingungen möglich ist.

Beim Redesign des SMARAGD-Systems wollen wir weiter einige theoretische und konzeptionelle Erweiterungen in Bezug auf die Sprache SNL einbeziehen, die zwar zur Zeit schon angedacht sind, aber noch einer genaue-

ren Analyse bedürfen. In Bezug auf algebraische Spezifikation möchten wir hier Höhere Spezifikationen in der Weise zulassen, daß Operatoren selbst als Elemente von "Höheren Sorten" aufgefaßt werden können, d.h. Operatoren sollten gleichberechtigt zu Objekten anderer Sorten in dem Sinne sein, das sie als Argumente anderer Operatoren oder Prädikate und als Ergebniswerte anderer Operatoren erscheinen können. Theoretische Konzepte für solche Spezifikationen höherer Ordnung gibt es schon in der Fachliteratur und werden u.a. in [Möller86, Poigne86] beschrieben. Sie werden weiter innerhalb der Spezifikationssprache SEGRAS des GRASPIN Projektes innerhalb eines Petri-Netz-Werkzeuges eingesetzt (siehe z.B. [Schmidt89]). Während uns die Integration solcher Höherer Konzepte zur Spezifikation innerhalb des Spezifikationsteils von SNL recht unproblematisch erscheint, haben wir die Auswirkungen solcher Konzepte bzgl. des SNL-Netz-Modells noch nicht eingehend untersucht, sehen hier aber prinzipiell keine großen Schwierigkeiten.

Im Rahmen der Erweiterung der Sprache SNL auf funktionale Konzepte ist auch interessant, inwieweit sich ein polymorphes Typkonzept in eine Theorie von Spezifikationen Höherer Ordnung integrieren läßt. Zur Behandlung von Operatoren als Datenobjekten muß ja die Theorie so erweitert werden, daß innerhalb der die Semantik beschreibenden Gleichungen Variablen erlaubt sind, die Operatoren von einem gewissen Typ beschreiben. In Bezug auf ein polymorphes Typkonzept müssen Variablen eingeführt werden, die Klassen von Sorten "von einem gewissen Typ" beschreiben. Uns ist allerdings bislang keine Erweiterung der Theorie der Spezifikation abstrakter Datentypen in Bezug auf ein polymorphes Konzept von Operatoren bekannt.

Bei SNL-Netz-Spezifikationen wollen wir untersuchen, inwieweit die Verwendung Höherer Sprachkonstrukte es uns insbesondere erlaubt, die Kantenbeschriftungen im dem Sinne zu erweitern, daß eine Kantenbeschriftung in der Realisierung in einem SNL-Netz-Modell durch Funktionen mit Multimengen als Ergebniswert beschrieben werden, wie dies in einem Coloured Petri-Netz der Fall ist. Weiter soll das Netz-Modell in Bezug auf die Verwendung von Netz-Modulen und Netz-Klassen erweitert werden. Der bisherige Prototyp erlaubt zwar in bereits sehr allgemeiner Form die Verwendung von Netz-Modulen bzw. Netz-Klassen, aber nur in der Weise, daß diese innerhalb eines Netz-Modells auf Unterseiten definiert sind. Im nächsten Prototyp wollen wir dagegen in der Lage sein, allgemein verwendbare Netz-Module bzw. Netz-Klassen einem Benutzer durch externe Bibliotheken bereitzustellen. Hiermit wollen wir erreichen, daß der Benutzer Netz-Module und Netz-Klassen im Sinne des Information-Hiding Prinzips als Black-Boxes verwenden kann, ohne daß er, wie das im Fall der Unterseiten im jetzigen Prototyp erfolgt, die Implementierungsdetails der verwendeten Netz-Klassen bzw. Netz-Module zu sehen bekommt. Wir können uns dabei gut vorstellen, daß es hier weiter sinnvoll ist, daß man zur besseren Visualisierung verwendeter Netz-Module bzw. Netz-Klassen, erlaubt, ihnen ein eindeutiges Icon zuzuordnen, daß dann anstatt eines einfachen Stellen- oder Transitionsymbols bei der Instantiierung als Symbol für den betreffenden Netz-Modul oder für die betreffende Netz-Klasse verwendet wird. Hierdurch kann man dann in SNL-Modellen, die nur auf der Instantiierung von Netz-Klassen bzw. Netz-Modulen basieren, die eigentliche Netz-Semantik vollständig verdecken. Der Benutzer sieht stattdessen je nach Applikation und verwendeten Bibliotheken z.B. ein Modell in Form eines Anlagenfließplanes für eine Maschinensteuerung mit Pumpen-, Ventilsymbolen, etc. oder das elektronische Schaltbild eines digitalen Systems mit verschiedenen Mikrochipsymbolen, Symbolen für Widerständen, usw.

Hierzu muß die bisherige Integration der Modulkonzepte innerhalb von SNL in der Weise verbessert werden, daß eine extern definierte Netz-Klasse bzw. ein extern definierter Netz-Modul nur auf Grundlage einer noch zu definierenden externen Schnittstellenbeschreibung ohne Kenntnis der Interna in ein SNL-Modell während der Editierphase eingebunden werden kann.

Wenn eine Simulation oder Analyse des betreffenden Modells erfolgen soll, müssen dann die internen Beschreibungen der extern definierten Netz-Module bzw. Netz-Klassen in das Modell einbezogen werden. Dies kann in einem ersten Ansatz so erfolgen, daß die beteiligten Netz-Module bzw. Netz-Klassen dann in Form ihrer SNL-Beschreibungen in das Modell geladen werden. Dies ist aber sicherlich nicht die effizienteste Methode. Insbesondere der ML-Code in den Deklarationsboxen der externen Netz-Klassen bzw. Netz-Module könnte bereits vorübersetzt sein, wie das von dem Standard ML System of New Jersey, daß wir zur Interpretation des ML-Codes benutzen, auch unterstützt wird. Inwieweit eine Vorübersetzung der Netz-Beschreibung selbst innerhalb von externen Bibliotheken sinnvoll ist, ist uns zur Zeit nicht klar. Zum einen fehlen hierzu leider theoretische Aussagen, die Mechanismen beschreiben, wie man eine Analyse des dynamischen Verhaltens eines Netz-Moduls bzw. einer Netz-Klasse unabhängig von ihrem Einsatz in der Art durchführen kann, daß die Ergebnisse dieser Analyse die Wirkungsweise der betreffenden Netz-Klasse bzw. des betreffenden Netz-Moduls bzgl. aller möglicher Instantiierungen in Modellen vollständig beschreibt. Hier wären auch theoretische Aussagen in der Hinsicht interessant, ob Netz-Klassen bzw. Netz-Module, die bestimmten Konstruktionsregeln genügen, ihnen zugeordnete Eigenschaften unabhängig von der Art ihrer Verwendung beibehalten.

An Hand von Beispielen aus der Praxis wollen wir Erfahrungen gewinnen, wie ausgewählte Klassen von verteilten Systemen durch Bibliotheken vorgefertigter Netz-Module, bzw. Netz-Klassen unterstützt werden können. So werden für verteilte Software-Applikationen Bibliotheken mit Bausteinen zur Kommunikation einzelner Prozesse der verteilten Applikation sinnvoll sein. Für Applikationen im Anlagenbau könnten Netz-Klassen oder Netz-Module innerhalb einer speziell für solche Anwendungen zugeschnittenen Bibliothek Bauteile wie Pumpen, Stellmotoren, Ventile, Meßfühler und dergleichen sein. Experimente mit dem Aufbau und der Anwendung solcher Bibliotheken werden dann schnell zu größeren, komplexeren Beispielnetzen führen, so daß wir uns hierdurch weitere Erfahrungen bzgl. der Verwendbarkeit des SMARAGD-Systems in Bezug auf umfangreiche Systeme erhoffen.

In Bezug auf das rein theoretische Konzept des SNL-Modells erscheint uns vor allem die Entwicklung eines geeigneten Morphismusbegriffs bzgl. der Klasse von SNL-Modellen interessant. Meine Vorstellungen gehen hierbei dahin, daß ein solcher Morphismus durch ein Paar von (linearen) Abbildungen zwischen korrespondierenden Modulen von P-Vektoren und T-Vektoren in der Weise beschrieben wird, daß sich zum einen die Inzidenzmatrix der Netze bzgl. dieser Abbildungen transformiert, und andererseits ein Schritt im einem System auf einen Schritt im anderen System abgebildet wird. Eine ähnliche Definition findet man für Coloured Petri Netze in [Dimitrovici90, Hummert89]. So eine Definition eines SNL-Modell-Homomorphismus liefert dann zum einen in natürlicher Weise einen zugehörigen Homomorphismus zwischen den zugehörigen Erreichbarkeitsgraphen der Netze, wobei gelten müßte: Sind zwei SNL-Modelle bzgl. des gewählten Homomorphismusbegriffs isomorph, so sind die zugehörigen Erreichbarkeitsgraphen isomorph (Man siehe für ähnliche Aussagen in Bezug auf Coloured Petri Netze wiederum [Dimitrovici90, Hummert89]). Zum weiteren sollte ein Morphismusbegriff in der obigen Art und Weise es erlauben, daß sowohl topologische Transformationen des Netzes, wie Teilen und Zusammenfügen von Stellen und/oder Transitionen, als auch Transformationen auf der Spezifikationsseite (d.h. Spezifikationsmorphismen und in der Realisierung Morphismen zwischen algebraischen Systemen) als SNL-Modell-Morphismen gedeutet werden können.

Aufbauend auf einen solchen Morphismusbegriff kann man dann die in dieser Arbeit nur informell beschriebene Semantik von Netz-Klassen bzw. Netz-Modulen, wie in der Theorie algebraischer Spezifikationen oder allgemeiner der Kategorientheorie üblich, über Push-In/Push-Out-Diagramme deuten ([Dimitrovici90, Hummert89]). Eine solche formale Basis der Semantik von hierarchischen Netzen bietet dann zunächst einmal die Grundlage, um weitergehende Fragen nach Eigenschaften von Netz-Modulen bzw. Netz-Klassen, die bei ihrer Instantiierung erhalten bleiben, überhaupt formal beschreiben zu können.

Neben der Formalisierung des Netz-Klassen- bzw. Netz-Modulkonzeptes, bietet einem ein Morphismusbegriff, wie er oben angedeutet wurde, weiter die Möglichkeit, verschiedene Transformationen von Netzen, formal zu beschreiben. Unter solchen Transformationen werden dann auf Grund der Morphismuseigenschaft eine Reihe von Eigenschaften von SNL-Modellen, wie z.B. S-Invarianten ebenfalls in natürlicher Weise transformiert, d.h. eine unter einem Morphismus transformierte S-Invariante ist wieder eine S-Invariante, allerdings eventuell eine triviale (Man siehe hierzu z.B. [Memmi87, Vautherin87]). Neben den rein topologischen Transformationen interessieren hierbei insbesondere auch solche Transformationen, die auf Grund einer anderen Realisierung der Spezifikationen eines Netz-Modells zustande kommen. Solche Morphismen sollten durch Morphismen zwischen solchen algebraischen Systemen induziert werden, die alle Realisationen ein und derselben Spezifikation (d.h. ein Modell einer vorgegebenen Spezifikation) sind. Hier gibt es stets einen eindeutig bestimmten Morphismus vom sogenannten initialen Modell der Spezifikation in ein beliebiges Modell. Die Eigenschaften eines SNL-Modells, die man dann vom initialen Modell auf beliebige andere Modelle über diesen eindeutig bestimmten Morphismus übertragen kann, stellen gewissermaßen die minimalen, in der Spezifikation festgelegten Eigenschaften des Modells dar. Andere mit dieser Methode nicht gewinnbare Eigenschaften sind dagegen von der aktuell gewählten Realisation und damit implementierungsabhängig. Eine weitere interessante Realisation für die Analyse eines SNL-Netzes ist das terminale Modell. Dieses kann man, da man hier gerade von der Individualität der Objekte im Netz abstrahiert, als ein Stellen/Transitionsnetz auffassen, daß einfach zu analysieren ist. Morphismen zwischen anderen Modellen und dem terminalen Modell erlauben es ebenfalls, einen Teil- der Analyseergebnisse bzgl. des terminalen Modells auf andere Modelle auszudehnen (Man siehe hierzu [Memmi87, Vautherin87]).

Mit diesem kurzen Ausblick auf mögliche weitere Entwicklungen des SMARAGD-Systems und zu Grunde liegenden SNL-Modells wollen wir diese Arbeit beschließen. Wir hoffen dabei, daß unsere bisherigen und zukünftigen Arbeiten am SMARAGD-System dazu beitragen werden, daß formale Methoden nicht nur von Theoretikern sondern vor allem auch von Praktikern in realen Projekten eingesetzt werden, um Designfehler bei großen verteilten Systemen schon innerhalb der Spezifikations- und Modellierungsphase weitgehenst zu vermeiden.

Literatur

Arbib75 — Arbib, M. A.; Manes, E. G.: *Arrows, Structures and Functors.* Academic Press, New York 1975

Battiston86 — Battiston, E.; DeCindio, F.; Mauri, G.: *OBJSA Nets: A Class of high-level Nets having Objects as Domains.* Interner Bericht der Universität Mailand, Mailand 1986

Berthomieu86 — Berthomieu, B.; et al: *Abstract Data Nets combining Petri Nets and Abstract Data Types for High Level Specifications of Distributed Systems.* In: Proceedings 7th European Workshop on Application and Theory of Petri Nets, Oxford, Juli 1986, pp. 25-48

Billington88 — Billington, J.; Wheeler, G. R.; Wilbur-Ham, M. C.: *PROTEAN: A High-level Petri Net Tool for the Specification and Verification of Communication Protocols.* In: IEEE Transactions on Software Engineering Vol 14, Nr. 3, 1988, pp. 301-316

Broy80 — Broy, M.; Wirsing, M.: *Initial versus terminal algebra semantics for partially defined abstract types.* TUM-I8018, Technische Universität München, München 1980

Broy82 — Broy, M.; Wirsing, M.: *Partial abstract data types.* Acta Informatika 18, 1982, pp. 47-64

Bruno85a — Bruno, G.; Marchetto, G.: *A Methodology based on High-Level Petri Nets for the Specification and the Design of Control Systems.* In: Proc. of the 3rd Workshop on SW-Specification and Design, London 1985, pp. 30-34

Bruno85b — Bruno, G.; Marchetto, G.: *An Integrated Software Production Environment for Automation Systems.* In: Proc. COMPSAC, Chicago 1985, pp. 350-357

Bruno86 — Bruno, G.; Marchetto, G.: *Process-translatable Petri Nets for Rapid Prototyping of Process-Control Systems.* In: IEEE Transactions on Software Engineering Vol. SE-12, Nr. 2, 1986, pp. 346-357

Burkhardt85 — Burkhardt, H. J.; Eckert, H.; Prinoth, R.: *Modellierung von OSI-Kommunikationsprotokollen und Protokollen mit Hilfe von Prädikat-Transitionsnetzen.* In: Zorn, W. (Ed.): Kommunikation in verteilten Systemen, Informatik-Fachberichte 95, Springer Verlag, pp. 610-645, Heidelberg 1985

CIP85 — CIP Language Group: Lecture Notes in Computer Science Vol. 183: *The Munich Project CIP, Volume1: The Wide Spectrum Language CIP-L.* Goos, G.; Hartmanis, H. (eds) -- Springer Verlag, Berlin, Heidelberg, New York, Tokyo, 1985

CIP87 — CIP Language Group: *The Munich Project CIP, Vol 2 : The transformation System CIP-S.* Springer LNCS 292, Berlin 1987

CPN90 — Meta Software Corporation: *Design/CPN User and Reference Manual.* Meta Software Corporation, Cambridge (MA) 1990

Dähler87 — Dähler, J.; et al: *A graphical tool for the design and prototyping of distributed systems.* In: ACM SIGSOFT Software Engineering Notes Vol 12, Nr. 3, pp. 25-36, 1987

Dimitrovici90 — Dimitrovici, C.; Hummert, U.; Petrucci, L.: *The Properties of Algebraic Nets Schemes in Some Semantics.* In: Proc. 11th International Conference on Application and Theory of Petri Nets, pp. 180-203, Paris 1990

Dimitrovici89a — Dimitrivici, C.; Hummert, U.: *Semantische Konstruktionen und Kategorien algebraischer Netzschemata.* Technischer Report 12, TU Berlin 1989

Dimitrovici89b — Dimitrovici, C.; Hummert, U.: *Kategorielle Konstruktionen für algebraische Petrinetze.* Technischer Report 23, TU Berlin 1989

Düpmeier90 — Düpmeier, C.; Korczynski, W.; Süß, W.: *Eine Einführung in die Grundlagen der Theorie der Höheren Petri-Netze.* KfK-Bericht 4636, Karlsruhe 1990

Dybjer83a — Dybjer, P.: *Algebraic models of functional languages.* Preprints of Proc. of Declarative Programming Workshop, University College London, London 1983

Dybjer83b — Dybjer, P.: *Category-theoretic logics and algebras of programs.* Ph. D. thesis, Chalmers Univerity of Technology, Chalmers 1983

Ehrich89 — Ehrich, H.-D.; Gogolla, M.; Lipeck, U. W.: *Algebraische Spezifikation abstrakter Datentypen.* Teubner Verlag, Stuttgart 1989.

Ehrig82 — Ehrig, H.; Kreowski, H.-J.; Mahr, B., Padawitz, P.: *Algebraic implementation of abstract data types.* In: TCS 20, pp. 209-263 (1982)

Ehrig85a — Ehrig, H.; Kreowski, H.-J.; Thatcher, J. W.; Wagner, E. G.; Wright, J. B.: *Parameterized data types in algebraic languages.* "Automata, Languages and Programming, Seventh Colloquium, Noordwijkerhout, the Netherlands", LNCS 85, Springer Verlag, Berlin 1985

Ehrig85b — Ehrig, H.; Mahr, B.: *Fundamentals of Algebraic Specification 1.* Springer Verlag, Berlin 1985.

Ehrig89 — Ehrig, H.; Heise, A.; Hummert, U.: *Algebraic High-Level Nets with Capacities.* Technischer Report, TU Berlin 1989

Ehrig90 — Ehrig, H.; Mahr, B.: *Fundamentals of Algebraic Specification 2.* Springer Verlag, Berlin 1990.

Fehling90 — Fehling, R.: *Hierarchische Petrinetze.* Forschungsbericht Nr. 344, Fachbereich Informatik der Universität Dortmund, Dortmund 1990

Feldbrugge86 — Feldbrugge, F.; Jensen, K.: *Petri Net Tool Overview 1986.* In: Brauer, W.; Reisig, W.; Rozenberg, G. (Eds.): Advanced Course on Petri Nets , Bad Honnef, Springer LNCS 254/255, Springer Verlag, Berlin 1986, pp. 20-61

Fonio87 — Fonio, H.: *Symbolic Execution of Nets Using Rewrite Systems.* GRASPIN TP GMD32, GMD, St. Augustin 1987

Genrich79 — Genrich, H. J.; Lautenbach, K.: *The Analysis of Distributed Systems by Means of Predicate/Transition-Nets.* In: Kahn, G. (Ed.): Semantics of Concurrent Computation, Springer LNCS 70, pp. 123-146, Springer Verlag, Berlin 1979

Genrich80a — Genrich, H. J.; Lautenbach, K.; Thiagarajan, P. S.: *Elements of General Net Theory.* In: Brauer, W. (Ed.): "Net Theory and Applications", LNCS 84, Springer Verlag, Berlin 1980, pp. 21-163

Genrich80b — Genrich, H. J.; Lautenbach, K.; Thiagarajan, P. S.: *Elements of General Net Theory - Condition/Event Systems.* In: Brauer, W. (Ed.): "Net Theory and Applications", LNCS 84, Springer Verlag, Berlin 1980, pp. 25-38

Genrich80c — Genrich, H. J.; Lautenbach, K.; Thiagarajan, P. S.: *Elements of General Net Theory - Nets and Logic.* In: Brauer, W. (Ed.): "Net Theory and Applications", LNCS 84, Springer Verlag, Berlin 1980, pp. 106-124

Genrich80d — Genrich, H. J.; Lautenbach, K.; Thiagarajan, P. S.: *Elements of General Net Theory - The Category of Nets.* In: Brauer, W. (Ed.): "Net Theory and Applications", LNCS 84, Springer Verlag, Berlin 1980, pp. 139-158

Genrich80e — Genrich, H. J.; Lautenbach, K.; Thiagarajan, P. S.: *Elements of General Net Theory - Predicate/Transition Systems.* In: Brauer, W. (Ed.): "Net Theory and Applications", LNCS 84, Springer Verlag, Berlin 1980, pp. 76-92

Genrich81 — Genrich, H. J.; Lautenbach, K.: *System Modelling with High-Level Petri Nets.* Theoretical Computer Science 13, pp. 109-136 (1981)

Genrich83 — Genrich, H. J.; Lautenbach, K.: *S-Invariance in Predicate/Transition-Nets.* In: Pagnoni, A.; Rozenberg, G. (Eds.): 3rd European Workshop on Application and Theory of Petri Nets, Informatik Fachberichte 66, Springer Verlag, Berlin 1983, pp. 98-111

Genrich86 — Genrich, H. J.: *Predicate/Transition Nets.* Lecture Notes in Computer Science Vol. 254: Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course, Bad Honnef, September 1986 / Brauer, W.; Reisig, W.; Rozenberg, G. (eds.) -- Springer Verlag, pp. 207-247 (1986)

Genrich89 — Genrich, H. J.: *Equivalence transformations on PrT-nets.* In:: Rozenberg, G. (Ed.): Advances in Petri Nets 1989, Lecture Notes in Computer Science Vol. 424, pp. 179-208, Springer Verlag, Berlin 1989

Gerlach88 — Gerlach, H.: *Rewriting applied to Pr/E-nets over algebraic specifications with constructors.* GRASPIN TP KAI29, GMD, St. Augustin 1988

Gogolla83 — Gogolla, M.: *Partially ordered sorts in algebraic specifications.* LNCS 154, Springer Verlag, Berlin 1983, pp. 139-153

Goguen75 — Goguen, J. A.; Thatcher, J. W.; Wagner, E. G.: *An Initial Algebra Approach To The Specification, Correctness And Implementation Of Abstract Data Types.* In: Yeh, R. (Ed.): "Current Trends in Programming Methodology IV: Data Structuring", Prentice Hall Verlag, New Jersey 1978, pp. 80-149

Goguen78 — Goguen, J. A.: *Abstract Errors for Abstract Data Types.* In: Neuhold, E. J. (Ed.): "Formal Descriptions of Programming Concepts", North-Holland Verlag, Amsterdam 1978, pp. 491-522

Goguen79 — Goguen, J. A.; Tardo, J. J.: *An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications.* In: "IEEE Proc. Specifications of Reliable Software", 1979, pp. 80-149

Grätzer79 — Grätzer, G.: *Universal Agebra.* Springer Verlag, Berlin 1979

Guttag78 — Guttag, J. V.; Horowitz, E.; Musser, D. R.: *The Design of Data Type Specifications.* In: "Current Trends in Programming Methodology", Vol. 4, Yeh, R. (Ed.), Prentice Hall Verlag, New Jersey 1978, pp. 60-79

Harper86 — Harper, R.; ET AL: *Standard ML.* LFCS Report (ECS-LFCS-86-2), Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh 1986

Harper88 — Harper, R.: *Modules and Persistence in Standard ML.* In: Atkinson, M. P.; Bunemann, P.; Morrison, R. (Eds.): Data Types and Persistence. Topics in Information Systems, Springer Verlag, Berlin 1988

Harper89 — Harper, R.: *Introduction to Standard ML.* Laboratory for Foundations of Computer Science Report Series, University of Edinburgh, Edinburgh 1989

Harper90 — Harper, R.; Milner, R.; Tofte, M.: *The Definition of Standard ML.* MIT Press, London 1990

Herrlich73 — Herrlich, H.; Strecker, G. E.: *Category Theory.* Allyn and Bacon, Boston 1973

Hickman80 — Hickman, J. L.: *A note on the concept of multiset.* In: Bulletin Of The Australian Mathematical Society Vol. 27, 1980

Huber89 — Huber, P.; Jensen, K.; Shapiro, R. M.: *Hierarchies in Coloured Petri Nets.* Proc. of the 10th Int. Conf. on Application and Theory of Petri Nets, pp. 192-209 (1989)

Hummert87 — Hummert, U.: *High-Level Netze.* Technischer Report 10, TU Berlin 1987

Hummert89 — Hummert, U.: *Algebraische Theorie von High-Level-Netzen.* Doktorarbeit, TU Berlin 1989

Jensen81a — Jensen, K.: *Coloured Petri Nets and the Invariant Method.* In: Theoretical Computer Science Vol. 14, pp. 317-336, North-Holland Verlag, 1981

Jensen81b — Jensen, K.: *How to Find Invariants for Coloured Petri Nets.* In: Mazurkiewicz, A. (Ed.): Mathematical Foundations of Computer Science, LNCS Vol. 118, pp. 327-338, Springer Verlag, Berlin 1981

Jensen86 — Jensen, K.: *Coloured Petri Nets.* Lecture Notes in Computer Science Vol. 254: Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course, Bad Honnef, September 1986 / Brauer, W.; Reisig, W.; Rozenberg, G. (eds.) -- Springer Verlag, pp. 248-299 (1987)

Klaeren83 — Klaeren, H. A.: *Algebraische Spezifikation, Eine Einführung.* Springer-Verlag, Berlin 1983.

Krämer85 — Krämer, B.: *Stepwise Construction of Non-Sequential Software Systems Using a Net Based Specification Language*. In: Advances in Petri Nets 84, LNCS 188, Springer Verlag, Berlin 1985, pp. 307-330

Krämer86a — Krämer, B.: *SEGRAS: The GRASPIN Specification Language* (Preliminary Reference Manual). GRASPIN TP GMD62, GMD, St. Augustin 1986, ISBN 3-88457-910-X

Krämer86b — Krämer, B.: *Interactive Graphical Specification using the syntax-directed SEGRAS-Lab*. In: Proc. 19th Ann. Hawaii Int. Conf. on System Sciences, Honolulu 1986, pp. 420-429

Krämer87 — Krämer, B.; Schmidt, H. W.: *Types and Modules for Net Specifications*. In: Voss, K.; Genrich, H. J.; Rozenberg, G. (Eds.): "Concurrency and Nets", Springer Verlag, Berlin 1987, pp. 270-286

Krämer89 — Krämer, B.: *Concepts, Syntax and Semantics of SEGRAS: A Specification Language for Distributed Systems*. GMD-Bericht Nr. 179, Oldenbourg Verlag, München 1989

Krämer90 — Krämer, B.; Schmidt, H.-W.: *Architecture and Functionality of a Specification Environment for Distributed Software*. In: Proc. of COMPSAC 90, 1990

Kreowsky84 — Kreowsky, H.-J.; Schmidt, H. W.: *Some Algebraic Concepts of the Specification Language SEGRAS and their Initial Semantics*. GMD-Studie Nr. 93, St. Augustin 1984

Lautenbach84a — Lautenbach, K.; Pagnoni, A.: *Liveness and Duality in marked-graph-like Predicate/Transition Nets*. In: Rozenberg, G. (Ed.): Advances in Petri Nets 1984, LNCS 188, pp. 331-352, Springer Verlag, Berlin 1984

Lautenbach84b — Lautenbach, K.; Pagnoni, A.: *On the various high-level Petri-Nets and their Invariants*. In: Newsletters Vol. 16, GMD 1984, pp. 42-58

Lautenbach86a — Lautenbach, K.: *Analysis Methods for Petri Nets Models*. In: Seminar 'Applicability of Petri Nets to Operations Research', Universita Commerciale Luigi Bocconi, Mailand 1986, pp. 14-43

Lautenbach86b — Lautenbach, K.: *Linear Algebraic Techniques for Place/Transition Nets*. In: Brauer, W.; Reisig, W.; Rozenberg, G. (Eds.): Advanced Course on Petri Nets, Bad Honnef, LNCS 254/255, Springer Verlag, Berlin 1986, pp. 142-167

Lugowski76 — Lugowski, H.: *Grundzüge der Universellen Algebra*. Teubner Verlag, Leipzig 1976

MacLane72 — MacLane, S.: *Categories for the working mathematician*. Springer Verlag, Berlin 1972

MacQueen85 — MacQueen, D. B.: *Modules in Standard ML*. In: Polymorphism II.2, October 1985

Manes76 — Manes, E. G.: *Algebraic Theories*. Springer Verlag, Berlin 1976

Memmi86 — Memmi, G.; Vautherin, J.: *Advanced Algebraic Techniques*. In: In: Brauer, W.; Reisig, W.; Rozenberg, G. (Eds.): Advanced Course on Petri Nets, Bad Honnef, LNCS 254/255, Springer Verlag, Berlin 1986

Memmi87 — Memmi, G.; Vautherin, J.: *Analysing Nets by the Invariant Method*. In: Rozenberg, G. (Ed.): Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I, pp. 300-336, Springer Verlag, Berlin 1987

Milner87 — Milner, R.: *Changes to the Standard ML Core Language*. LFCS Report Series, University of Edinburgh, Edinburgh 1987

Möller86 — Möller, B.: *Algebraic specifications with higher-order operations*. In: Meertens (Ed.): Proc. IFIP TC 2 Working Conf. Program Specification and Transformation, North Holland Verlag, Amsterdam 1986

Näher90 — Näher, S.: *LEDA User Manual*. Universität des Saarlandes, Saarbrücken 1990

OA90 — Meta Software Corporation: *Design/OA User and Reference Manual*. Meta Software Corporation, Cambridge (MA) 1990

OpenLook90 — Sun Microsystems: *Open Look Graphical User Interface Application Style Guidelines*. Addison-Wesley Publishing Company, Inc., New York 1990

Partsch90 — Partsch, H. A.: *Specification and Transformation of Programs, A Formal Approach To Software Development.* Springer Verlag, Berlin 1990

Peterson81 — Peterson, J. L.: *Petri Net Theory and the Modeling of Systems.* Prentice Hall Verlag, Englewood Cliffs 1981

Poigne86 — Poigne, A.: *On Specifications, Theories and Models with Higher Types.* In: Information and Control, Vol. 6, nos. 1-3, 1986

Reichel84 — Reichel, H.: *Structural Induction on Partial Algebras.* Mathematische Forschung (Mathematical Research) Vol 18. Akademie Verlag, Berlin 1984

Reisig85 — Reisig, W.: *Petri Nets with Individual Tokens.* In: Theoretical Computer Science Vol. 41, pp. 185-213, North-Holland Verlag, 1985

Reisig86 — Reisig, W.: *Petrinetze -- Eine Einführung (2. überarbeitete Auflage).* Springer Verlag, Berlin 1986

Reisig91 — Reisig, W.: *Petri nets and algebraic specifications.* In: Theoretical Computer Science Vol. 80, pp. 1-34, Elsevier Science Publishers, 1991

Richter78 — Richter, M. M.: *Logikkalküle.*, Teubner Studienbücher Informatik, Teubner Verlag, Stuttgart 1978

Sannella85 — Sannella, D. T.; Tarlecki, A.: *Program specification and development in Standard ML.* Proc. 12th ACM Symp. on Principles of Programming Languages, New Orleans 1985, pp. 67-77

Sannella86 — Sannella, D. T.; Tarlecki, A.: *Extended ML: an institution-independent framework for formal program development.* In: Proc. Workshop on Category Theory and Comp. Programming, Guildford. Springer LNCS 240, Berlin 1986, pp. 364-389

Schmidt89 — Schmidt, H. W.: *Specification and Correct Implementation of Non-Sequential Systems Combining Abstract Data Types and Petri Nets.* GMD-Bericht NR. 176, R. Oldenbourg Verlag, München/Wien 1989

Starke90 — Starke, P.: *Analyse von Petri-Netz-Modellen.* Teubner Verlag, Stuttgart 1990

Süß92 — Süß, W.: *Konzept und Entwicklung eines graphischen Analysewerkzeugs für Höhere Petri-Netze mit zustandsabhängiger Schaltregel.* KfK-Bericht 5246, Kernforschungszentrum Karlsruhe GmbH, Karlsruhe 1993

Thatcher82 — Thatcher, J. W.; Wagner, E. G.; Wright, J. B.: *Data Type Specification: Parameterization and the Power of Specification Techniques.* In: ACM Transactions on Programming Languages and Systems, Nr. 4, 1982, pp. 711-732

Vautherin86 — Vautherin, J.: *Parallel Systems Specifications with colored Petri nets and algebraic abstract data types.* In: Proc. 7-th European Workshop on Applications and Theory of Petri Nets, Oxford 1986, pp. 5-23

Vautherin87 — Vautherin, J.: *Parallel Systems Specifications with Coloured Petri Nets and Algebraic Specifications.* Lecture Notes in Computer Science Vol. 266: Advances in Petri Nets 1987 / Rozenberg, G. (ed.) -- Springer Verlag, pp. 293-308 (1987)

Wirsing83 — Wirsing, M.; Pepper, P.; Partsch, H.; Dosch, W.; Broy, M.: *On hierarchies of abstract data types.* Institut für Informatik TU München, TUM-I8007. Oder: Acta Informatika 20, 1983, pp. 1-33