Diplomarbeit

im Studiengang Informatik
an der Universität Hildesheim

Übersetzung von SDL-Spezifikationen in Petri-Netze

Institut für Informatik
Professor Dr. E. Best

Stefan Schwoon
Ostpreußenstraße 27
D-31139 Hildesheim

Betreuung: Bernd Grahlmann

August 1998

Ich versichere, die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel erstellt zu haben.

Hildesheim, 19. August 1998.

Stefan Schwoon

Zusammenfassung

Es wird aufgezeigt, wie in SDL erstellte formale Spezifikationen in Petri-Netze übersetzt werden können. Ein entsprechender Compiler wird implementiert und in das PEP-Projekt integriert. Um den Zweck der Verifikation von SDL-Spezifikationen zu erfüllen, wird der Compiler um eine Referenzkomponente erweitert und ein Formeltransformator erstellt.

Stichworte: SDL, Compiler, PEP, M-Netze, Referenzen, Verifikation

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Unter den Anforderungen, die an Hard- und Softwaresysteme gestellt werden, nehmen Zuverlässigkeit und Korrektheit einen zentralen Platz ein. Um gewünschte Eigenschaften von Progammen oder Schaltungen sicherzustellen, reichen menschliche Intuition und Erfahrung allein oftmals nicht aus. Dies gilt umso mehr, je größer ein System wird und je mehr parallele Abläufe es umfaßt. Parallelität jedoch ist ein wesentliches Merkmal vieler Anwendungen. Wünschenswert wäre es daher, Eigenschaften von Systemen automatisch überprüfen (verifizieren) zu können.

Um das Ziel automatischer Verifikation zu erreichen, sind mehrere Dinge notwendig:

- Das betrachtete System muß modelliert werden, und zwar mit möglichst großer Exaktheit. Insbesondere muß eindeutig festgelegt werden, wie die gleichzeitig stattfindenden Abläufe miteinander interagieren.
- Eigenschaften müssen formuliert werden, etwa in Gestalt von Formeln. Die Semantik dieser Formeln muß eindeutig bestimmt sein.
- Es muß festgestellt werden können, ob ein gegebenes Modell eine gegebene Formel erfüllt.

Für den ersten Punkt, die Modellierung paralleler Systeme, haben sich formale Beschreibungstechniken (Formal Description Techniques, FDTs) herausgebildet, zum Teil genormt durch internationale Gremien. Eine solche Spezifikationssprache ist SDL (Specification and Description Language, [CCI92]); diese und andere FDTs werden u.a. in [Hog89] und [Tur93] beschrieben.

Für den dritten Punkt stehen im Rahmen des PEP-Projekts [BG95] effiziente Modelchecker [Gra97b] zur Verfügung. Diese basieren auf Petri-Netzen [Rei85] und können Aussagen über temporallogische Eigenschaften wie die Erreichbarkeit von gegebenen Markierungen treffen.

In [FG98a] und [FG98b] haben Fleischhack und Grahlmann einen Ansatz vorgestellt, der eine Brücke zwischen diesen Punkten schlägt, und der Grundlage für diese Arbeit war. Es wird dargestellt, wie in SDL erstellte Spezifikationen in Petri-Netze übersetzt werden, genauer gesagt in M-Netze [BFF+95], eine besondere Klasse von High-Level-Petri-Netzen. Diese können mit bereits in PEP vorhandenen Werkzeugen in Low-Level-Netze transformiert und von Modelcheckern geprüft werden.

Zur Verifikation kann der Benutzer die zu überprüfenden Eigenschaften auf SDL-Ebene ausdrücken. Diese Formeln werden in Netz-Formeln übersetzt, von Modelcheckern geprüft, und die Ergebnisse auf die SDL-Ebene zurückübertragen. Hierzu wird ein Referenzschema gebraucht, das Einzelteile von SDL-Programm und zugehörigem Netz einander zuordnet.

Die Grafik in Abbildung 1.1 gibt diesen Ansatz schematisch wieder.

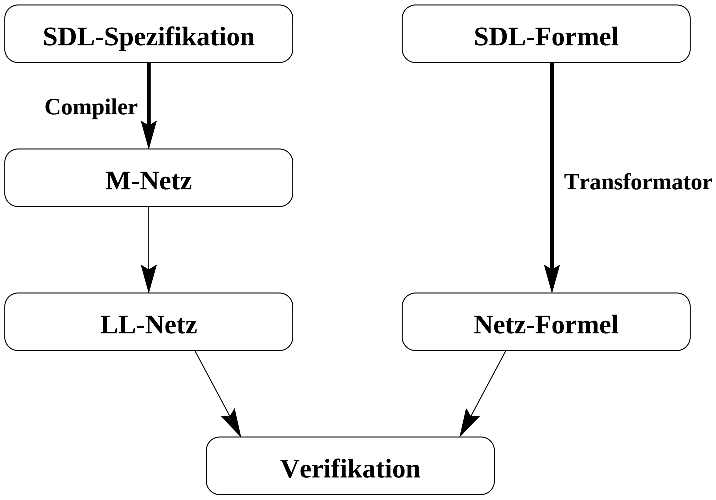

Abbildung 1.1: Der Ansatz im Überblick

Die Ausarbeitung gliedert sich wie folgt: In Kapitel 2 werden die Sprache SDL sowie ihre wichtigsten Konzepte vorgestellt und ein Beispiel gegeben, auf das in den nachfolgenden Kapiteln immer wieder zurückgegriffen wird.

In Kapitel 3 folgt eine kurze Beschreibung von M-Netzen und eine Zeichenerklärung; für eine formale Definition wird auf andere Werke verwiesen.

Kapitel 4 schließt mit der Definition der formalen Semantik an, wie sie vom vorliegenden Compiler implementiert wurde. Einzelne Punkte dieser Semantik werden in Kapitel 5 aufgegriffen und diskutiert, gleichzeitig werden alternative Möglichkeiten für Teile der Semantik vorgestellt.

Kapitel 6 folgt mit einer Beschreibung des Compilers, Kapitel 7 mit der des Formeltransformators und Kapitel 8 mit einer Beschreibung weiterer Tätigkeiten, die notwendig waren, um SDL in PEP zu integrieren.

Kapitel 9 beschließt die Arbeit mit einem Fazit und zeigt mögliche Weiterentwicklungen auf.

Kapitel 2

SDL

Die Natur vieler Anwendungen in der elektronischen Datenverarbeitung macht es sinnvoll oder gar notwendig, sie als verteilte Systeme zu modellieren. Dies gilt vor allem für Systeme, in denen für eine Menge von (gleichzeitigen) Teilnehmern eine bestimmte Dienstleistung erbracht werden soll. Beispiele hierfür sind etwa

- ein Datenbanksystem, in dem ein Server gleichzeitig Verbindungen mit einer Menge von Clients unterhält, Transaktionen durchführt, Anfragen beantwortet etc.
- ein Telefonnetz mit einer großen Menge von Teilnehmern, bei dem alle Teilnehmer jederzeit unabhängig voneinander Verbindungen zu beliebigen anderen Teilnehmern aufbauen können.

Grundlage jeder Programmentwicklung ist eine Spezifikation, die die Anforderungen an das in Frage stehende System wiedergibt. Diese Spezifikation kann je nach Lage der Dinge mehr oder weniger präzise und umfangreich sein (und existiert manchmal gar nur in den Gedanken der an der Entwicklung beteiligten Personen).

Verteilte Systeme erfordern spezielle Ansprüche an die Erstellung von Spezifikationen. Ihre Komplexität und die Tatsache, daß einzelne Komponenten eines Systems gelegentlich von unterschiedlichen Personen implementiert werden, lassen es wichtig erscheinen, das Verhalten dieser Komponenten (und damit des Systems als ganzem) genau festzulegen. Insbesondere sollen Verabredungen getroffen werden, wie sich Komponenten *zueinander* verhalten, wie sie miteinander kommunizieren und auf welchem Wege dies geschieht.

Damit diese Verabredungen ihren Zweck erfüllen, müssen sie präzise und eindeutig interpretierbar sein. Umgangssprachliche Beschreibungen sind dazu in der Regel nicht geeignet. Zweckmäßig erscheint vielmehr die Verwendung einer formalen Beschreibungssprache, deren Semantik klar festgelegt ist. Erst die Eindeutigkeit der Semantik ermöglicht nicht nur die präzise Spezifikation von Verhaltensregeln – sie läßt auch Aussagen über die *Korrektheit* einer solchen Spezifikation zu.

Um die Akzeptanz solcher Beschreibungssprachen zu erhöhen und ihre einheitliche Interpretation zu gewährleisten, haben internationale Normungsgremien Standards er-arbeitet, die die drei Spezifikationssprachen Estelle, LOTOS und SDL [CCI92, Hog89] definieren.

2.1 Entstehung und Anwendung von SDL

SDL entstand aus zunächst informellen zeichnerischen Beschreibungstechniken im Bereich der Telekommunikation. Ihre Normung wurde ab 1968 vom CCITT[1)] vorangetrieben, der erste offizielle Standard entstand jedoch erst im Jahre 1988. Dieser wurde 1992 erweitert und in [CCI92] beschrieben. In der Zwischenzeit hatten einzelne Firmen bereits eigene, für ihre speziellen Zwecke angepaßte Dialekte entwickelt, die bis heute bestehen und dazu beitragen, daß unter den Benutzern von SDL noch immer Uneinigkeit über einige nicht unwichtige Details der Sprache herrscht. Grundlage dieser Arbeit war der auch als SDL-92 bezeichnete Standard des CCITT.[2)]

SDLs Wurzeln in der Telekommunikation haben die Sprache unübersehbar geprägt:

- Um verteilte Systeme darstellen zu können, macht SDL Parallelität zu einem ihrer zentralen Konzepte.
- Die Möglichkeit der Spezifikation von Kommunikationsstrukturen und Signalen macht SDL besonders geeignet zum Entwurf von Protokollen.[3)]
- Dynamische Prozeßerzeugung erlaubt die Simulation von Systemen mit einer wechselnden Anzahl von Teilnehmern und von Client-Server-Systemen (man beachte, daß dies für die oben genannten Beispiele von Bedeutung ist).
- Die hierarchische Organisation eines Systems in Blöcke spiegelt reale Gegebenheiten wieder. Soll beispielsweise die Kommunikation zweier (physisch getrennter) Geräte spezifiziert werden, so können sie als zwei Blöcke dargestellt werden, die über einen Kanal verbunden sind.

Diese Arbeit behandelt nicht den vollen Sprachumfang von SDL, aus verschiedenen Gründen. Zum einen ist dieser Sprachumfang sehr groß (siehe [CCI92]), zum anderen sind nicht alle im SDL-Standard enthaltenen Konzepte für die Zielsetzung der Verifikation wirklich interessant, während andere sich äquivalent durch simplere Konstrukte ausdrücken lassen.

Eine Konzentration auf wesentlichere Konzepte von SDL schien daher möglich, ohne die Brauchbarkeit des Compilers für den angestrebten Verwendungszweck besonders zu beeinträchtigen. Der Schwerpunkt dieser Arbeit lag daher auf dem auch in [FG98a] behandelten Ausschnitt von SDL.

[1)] Comité Consultatif International Téléphonique et Télégraphique

[2)] Das CCITT wurde im Jahre 1993 durch den Telecommunication Standardization Sector der International Telecommunication Union (ITU) ersetzt.

[3)] Das bedeutendste Beispiel für die Anwendung von SDL dürfte die Spezifikation der ISDN-Protokolle sein.

Die in PEP integrierten Verifikationsalgorithmen basieren auf Low-Level-Netzen und können nur endliche Netze bearbeiten. Das vom Compiler erzeugte High-Level-Netz darf also nur Stellen mit endlichen Typen besitzen, andernfalls würde seine Entfaltung ein unendliches LL-Netz ergeben. Diese Tatsache mußte in die Überlegungen einfließen und führte zu der Entscheidung, als zulässige Wertemengen von Variablen nur endliche Teilmengen der ganzen Zahlen, Boolsche Werte und Prozeß-IDs zuzulassen.

Andere Einschränkungen beinhalten den Verzicht auf abstrakte Datentypen, die in SDL-92 eingeführten objektorientierten Erweiterungen sowie viele andere Details.

2.2 Grundlagen von SDL

In diesem Abschnitt wird eine kurze Übersicht der wichtigsten Konzepte von SDL gegeben. Das im nächsten Abschnitt vorgestellte Beispiel mag das Verständnis für die nachfolgenden Erläuterungen vertiefen.

Ein SDL-System besteht aus einer Menge von Prozessen. Prozesse laufen parallel und unabhängig voneinander ab, sie können jedoch auf festgelegten Wegen miteinander kommunizieren. Die Anzahl der Prozesse ist nicht statisch – zu Beginn der Ausführung des Systems wird eine Menge von Prozessen erzeugt, und während der Ausführung kann jeder Prozeß andere Prozesse erzeugen. Prozesse können nicht von außen oder von anderen Prozessen beendet werden, sondern terminieren ausschließlich von selbst.

Obgleich Prozesse wie bereits angedeutet in einer Hierarchie von Blöcken geordnet sind, stehen sie während der Ausführung vollkommen gleichberechtigt nebeneinander. Die Organisation in Blöcke gibt lediglich eine mögliche physische Anordnung wieder (s.o.).

Prozesse werden als erweiterte endliche Automaten verstanden. Wie gewöhnliche endliche Automaten sind sie gekennzeichnet durch eine (endliche) Menge von Zuständen. Ihre "Eingabe" besteht aus einer Folge von Signalen, die sie von anderen Prozessen erhalten, ihre "Ausgabe" aus Signalen, die sie an andere Prozesse oder an sich selbst verschicken. In Analogie zu endlichen Automaten wird aufgrund des momentanen Zustands und eines erhaltenen Signals über die nächste Aktion des Prozesses entschieden.

Eine Erweiterung gegenüber endlichen Automaten besteht darin, daß Prozesse neben ihrem Zustand Daten in Variablen speichern und manipulieren können. Darüberhinaus stehen Kontrollkonstrukte wie deterministische und nichtdeterministische Entscheidung, Schleifen (mittels Sprunganweisung) und (rekursive) Prozeduren zur Verfügung. Ferner können spontane Zustandsübergänge erfolgen (d.h. solche ohne Erhalt eines Signals).

Erwähnt werden muß an dieser Stelle die Unterscheidung zwischen Prozeß*typen* und Prozeß*instanzen*. Die Beschreibung von Prozessen in einer Spezifikation definiert Prozeßtypen, die während der Interpretation der Spezifikation instanziiert werden können. Für jeden Prozeßtyp sind die ursprüngliche und die maximal gleichzeitig erlaubte Anzahl von Instanzen festgesetzt; jede Instanz wird durch eine eindeutige

Identifikations-Nummer (ID) gekennzeichnet. Im folgenden wird zwischen Typ und Instanz eines Prozesses nur dann explizit unterschieden, wenn die gemeinte Bedeutung nicht aus dem Zusammenhang ersichtlich ist.

Zur Kommunikation stehen den Prozessen Kanäle und Signalrouten zur Verfügung. Die auf diesen Wegen übertragenen Informationen werden Signale genannt. Bei Kanälen und Signalrouten handelt es sich um zuverlässige Medien, d.h., bei der Übertragung gehen keine Daten verloren. Kanäle und Signalrouten können uni- oder bidirektional sein.

Kanäle verbinden Blöcke untereinander, Signalrouten dienen der Kommunikation von Prozessen innerhalb eines Blocks. Bei der Übertragung durch Kanäle entsteht eine (nichtdeterministische) Verzögerung, während Signalrouten Daten sofort weiterleiten. Auch bei Kanälen ist jedoch gewährleistet, daß die Reihenfolge der übertragenen Daten nicht verändert wird, d.h., ein Kanal verhält sich wie ein FIFO[4)]-Puffer.

Im Gegensatz zur dynamischen Natur der Prozesse sind die Kommunikationsstrukturen statisch, d.h., die vorhandenen Kanäle und Signalrouten werden bei der Spezifikation festgelegt. Während der Interpretation können keine neuen Kanäle oder Signalrouten erzeugt, sondern nur die bereits vorhandenen benutzt werden. Kanäle und Signalrouten sind quasi als konkrete physikalisch vorhandene Leitungen zu verstehen.

Erreicht ein Signal (nach Übertragung durch Kanäle und/oder Signalrouten) den Empfänger-Prozeß, so wird es zunächst in der Eingabewarteschlange (input queue) des Prozesses zwischengespeichert, bis der Prozeß seinen derzeitigen Zustandsübergang abgearbeitet hat. Erst dann wird das Signal gelesen und aus der Warteschlange entfernt. Ähnlich einem Kanal arbeitet eine Warteschlange als FIFO-Puffer.

2.3 Ein Beispiel

Eine der bemerkenswertesten Eigenschaften von SDL ist, daß die Sprache zwei syntaktische Formen besitzt: eine graphische und eine textuelle, die mit SDL/GR (graphical representation) bzw. SDL/PR (phrase representation) bezeichnet werden. Beide Formen sind gleichberechtigt, und zu jedem Sprachelement gibt es jeweils ein graphisches Symbol und eine semantisch äquivalente Textdarstellung. Einige Sprachelemente besitzen keine wirkliche graphische Darstellung; in diesem Falle wird auf die Textdarstellung zurückgegriffen.

Das folgende Beispiel zeigt beide Formen der Darstellung. Da der für diese Arbeit implementierte Compiler jedoch auf der textuellen Syntax basiert, setzt auch die Semantik in den späteren Kapiteln auf dieser auf.

Abbildung 2.1 zeigt das spezifizierte System `s` im Überblick. Zwei in verschiedenen Blöcken liegende Prozesse kommunizieren über ein Geflecht von Kanälen und Signalrouten miteinander. Im Textsegment auf der linken Seite wird ein Typ namens `set` mit der Wertemenge $\{0, 1, 2, 3\}$ definiert. Die Signale `sig` und `ack` sind mit einem

[4)] first in, first out

Argument dieses Typs behaftet.

Kanäle (hier: `c1`, `c2`) sind durch Pfeilspitzen in der Mitte der Leitung gekennzeichnet, Signalrouten (hier: `r1`, `r2`, `r1`) durch eine Pfeilspitze am Ende. Beide Arten von Leitungen sind mit der Liste der Signale versehen, die sie übermitteln können. Man beachte, daß die Signalroute `r1` im Block `foo` und die Signalroute `r1` im Block `bar` unterschiedliche, voneinander vollkommen unabhängige Objekte sind, die lediglich den gleichen Namen tragen. Der Übersetzer stellt auch in diesem Falle sicher, daß die Routen auseinandergehalten werden und ein korrektes Netz entsteht.

Die an den Ecken "abgefeilten" Kästchen stellen Prozesse dar, deren genauere Struktur in den nachfolgenden Abbildungen dargelegt wird. Hinter den Prozeßnamen (hier: `one` und `two`) sind die Anzahlen anfänglicher und maximal möglicher (gleichzeitig aktiver) Instanzen gegeben. Vom Prozeß `one` ist also anfänglich eine Instanz aktiv, und es kann maximal zwei Instanzen geben. Vom Prozeß `two` gibt es maximal eine Instanz, die auch von Beginn an aktiv ist.

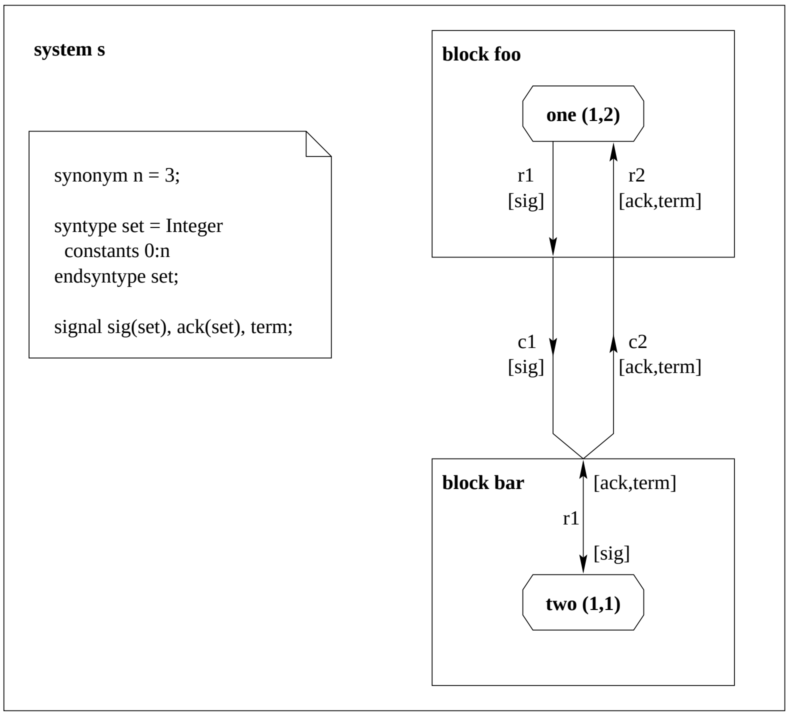

Abbildung 2.1: Ein SDL-System (Überblicksskizze)

Abbildungen 2.2 und 2.3 zeigen die Prozesse im Detail. Das System spezifiziert ein einfaches Protokoll, bei dem Instanzen des Prozesses `one` bei 0 beginnend in aufsteigender Reihenfolge Werte an den Prozeß `two` verschicken, die dieser nichtdeterministisch mit einer richtigen oder einer falschen Antwort quittiert. Bei einer falschen Antwort wird der zuletzt gesandte Wert solange wiederholt, bis er richtig quittiert wird. Bei Erreichen des Wertes 3 werden alle Prozesse terminiert, d.h. die Simulation des Systems beendet. Im folgenden werden die einzelnen Schritte detailliert erläutert.

Prozeß `one` besitzt einen formalen Parameter (`x`). Die implizite Variable **parent** liefert für jede Prozeßinstanz die ID der erzeugenden Prozeßinstanz. Für die anfängliche Instanz von `one` (bei der **parent** den speziellen Wert **Null** hat) ist der Wert von `x` undefiniert, er wird daher auf 0 gesetzt; für andere Instanzen wird der übergebene Wert beibehalten.

Nach dieser Initialisierung geht der Prozeß in den Zustand `send` über, den er spontan (ohne äußere Einwirkung, ausgedrückt durch Empfang des speziellen "Signals" **none**) verlassen kann, um die Prozedur `sendsig` aufzurufen und in den Zustand `wait` zu wechseln.

Die Prozedur `sendsig`, die einen "call by value"-Parameter hat, tut nichts weiter, als eine Instanz des Signals `sig` mit diesem Parameter als Argument zu versenden und sich anschließend zu beenden. Das versandte Signal kann nur einen möglichen Weg nehmen (siehe Abbildung 2.1), und zwar über die Signalroute `r1` im Block `foo`, den Kanal `c1` und die Signalroute `r1` im Block `bar` an die einzige Instanz des Prozesses `two`.

Der Prozeß befindet sich nun im Zustand `wait` und erwartet ein Signal (das hier nur vom Prozeß `two` kommen kann). Auf Erhalt des Signals `term` hin beendet sich der Prozeß. Erhält er das parametrisierte Signal `ack`, so wird dessen Argument in der Variablen `y` gespeichert und geprüft, ob dessen Wert um eins größer als der zuvor versandte Wert von `x` ist. Wenn nein, geht der Prozeß erneut in den Zustand `send` über (der Wert von `x` wird nochmals verschickt), ansonsten wird versucht, eine neue Instanz von `one` zu erzeugen. Dieser Instanz wird der Wert von `y` als formaler Parameter übergeben. Die implizite Variable **offspring** hält das Ergebnis dieses Versuchs fest: Ist ihr Wert **Null**, so schlug die Erzeugung fehl (weil bereits die maximale Anzahl von Instanzen existierte), und der Versuch wird wiederholt. Im Erfolgsfall enthält **offspring** die ID der erzeugten Instanz, und der Prozeß terminiert.

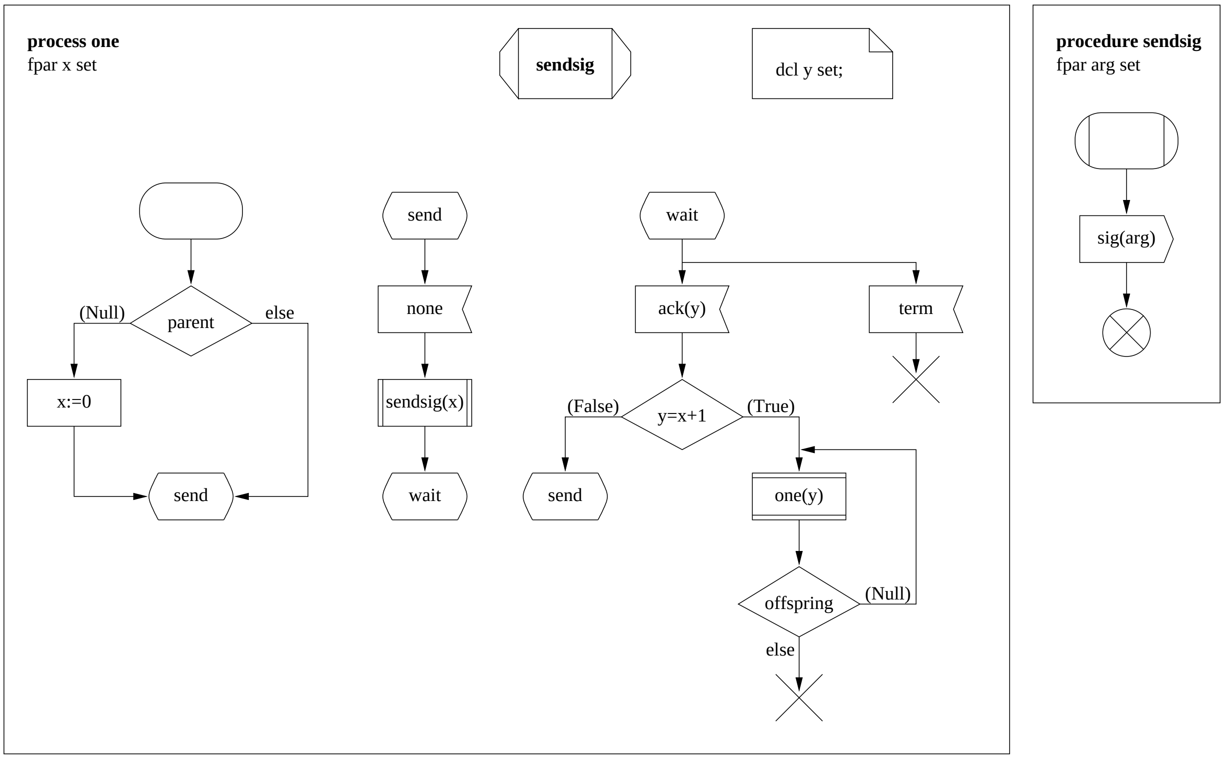

Abbildung 2.2: Detaillierte Darstellung des Prozesses `one`

Prozeß `two` geht vom Start weg in den Zustand `wait` über (der nichts mit dem gleichnamigen Zustand im Prozeß `one` zu tun hat) und wartet auf Eingaben. Erhält er das Signal `sig` mit einem Argumentwert von 3 ($n = 3$, siehe Abbildung 2.1), so antwortet er mit dem Signal `term` und beendet sich selbst. Die Angabe **to sender** beim Versand des Signals weist das System an, das Signal an die Prozeß-Instanz zu versenden, von der das Eingabe-Signal kam (die implizite Variable **sender** enthält die ID desjenigen Prozesses, dessen Signal zuletzt aus der Eingabewarteschlange gelesen wurde).

Bei einem Wert von weniger als 3 wird nichtdeterministisch (Schlüsselwort **any**) entschieden, ob die Prozedur **inc** aufgerufen wird. Diese ist mit einem "call by reference"-Parameter namens `val` versehen, den sie um eins erhöht. Nach der nichtdeterministischen Entscheidung wird der (möglicherweise um eins erhöhte) Argumentwert zurückgesandt. Anschließend geht der Prozeß wieder in den Zustand `wait` über (der Zustand "–" steht für den Zustand, in dem sich der Prozeß vor dem Zustandsübergang befunden hat).

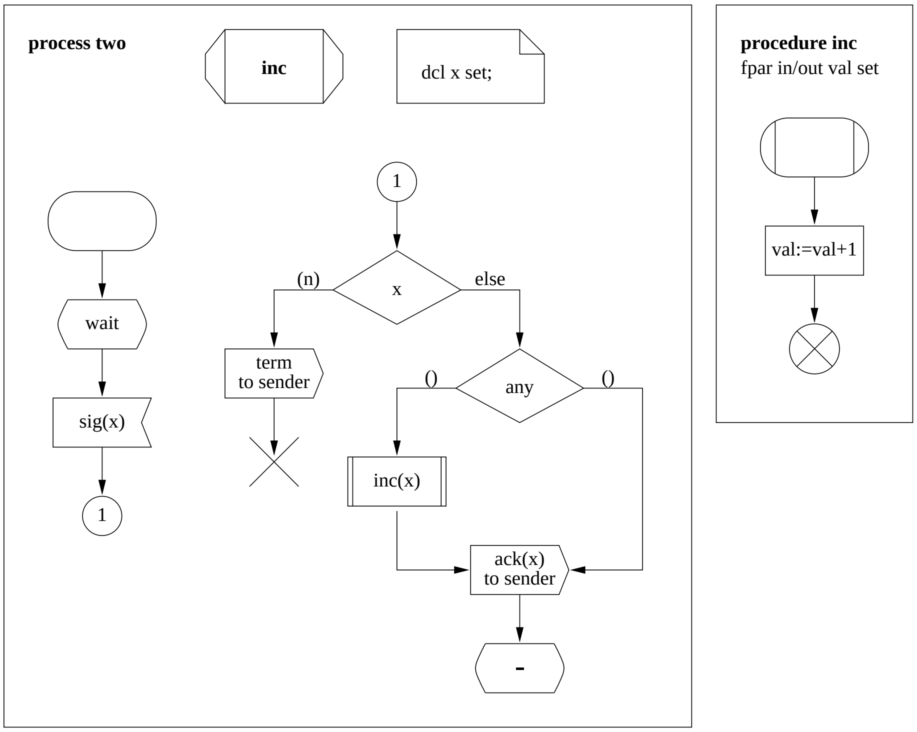

Abbildung 2.3: Detaillierte Darstellung des Prozesses `two`

Abbildung 2.4 zeigt die textuelle Repräsentation des Systems.

```
system s;

synonym n = 3;
syntype set = Integer
  constants 0:n
endsyntype;

signal sig(set), ack(set), term;

channel c1
  from foo to bar with sig;
endchannel;
channel c2
  from bar to foo with ack,term;
endchannel;
```

```
block foo;

  connect c1 and r1;
  connect c2 and r2;
  signalroute r1
     from one to env with sig;
  signalroute r2
     from env to one with ack,term;

  process one (1,2);
    fpar x set;
    dcl y set;

    procedure sendsig;
        fpar arg set;
      start;
        output sig(arg);
        return;
    endprocedure;

    start;
        decision parent;
          (Null): task x:=0;
          else:
        enddecision;
        nextstate send;

    state send;
      input none;
        call sendsig(x);
        nextstate wait;

    state wait;
      input ack(y);
        decision y=x+1;
        (False):nextstate send;
        (True): again:
                create one(y);
                decision offspring;
                 (Null):join again;
                 else:  stop;
                enddecision;
        enddecision;
```

```
      input term;
        stop;

  endprocess;

endblock;

block bar;

  connect c1,c2 and r1;
  signalroute r1
     from two to env with ack,term;
     from env to two with sig;

  process two;
    dcl x set;

    procedure inc;
        fpar in/out val set;
      start;
        task val:=val+1;
        return;
    endprocedure;

    start;
        nextstate wait;
    state wait;
      input sig(x);
        decision x;
          (n):  output term to sender;
                stop;
          else: decision any;
                  (): call inc(x);
                  ():
                enddecision;
                output ack(x) to sender;
                nextstate -;
        enddecision;
  endprocess;

endblock;

endsystem;
```

Abbildung 2.4: Das SDL-System in Textdarstellung

Kapitel 3

M-Netze

Das PEP[1)]-Projekt [BG95] macht Petri-Netze zum zentralen Werkzeug für die Modellierung und Analyse paralleler Algorithmen. Durch eine Vielzahl von Programmen und Algorithmen ist die Bearbeitung und Simulation von Netzen und die Verifikation von Netzeigenschaften möglich.

Um die Petri-Netz-Semantik paralleler Programmiersprachen festzulegen, eignen sich M-Netze, eine besondere Klasse von Petri-Netzen, wie sie in [BFF+95] beschrieben sind. M-Netze können mit bereits in PEP vorhandenen Werkzeugen in Low-Level-Petri-Netze (LL-Netze) entfaltet werden. Für diese wiederum steht eine Anzahl von Verifikations-Paketen zur Verfügung [Gra97b]. Die Angabe einer M-Netz-Semantik für SDL-Spezifikationen ist daher Grundlage, Korrektheitsnachweise für letztere zu führen. Um diese Nachweise möglichst komfortabel gestalten und Korrektheitseigenschaften auf SDL-Ebene spezifizieren zu können, muß die Semantik mit einem Referenzenschema versehen werden (siehe auch [Gra97c]). Dieses erlaubt die Zuordnung von Einzelteilen von SDL-Programm und zugehörigem Petri-Netz und muß bis auf LL-Ebene durchgereicht werden (siehe auch Abschnitt 4.1.3).

Auf eine Definition von M-Netzen und ihren Kompositionsoperatoren wird hier verzichtet, da eine solche bereits in [Gra98b] ausführlich und formal enthalten ist. Die dort gegebenen Definitionen wurden für diese Arbeit übernommen. Es wird daher nur eine Zeichenerklärung für die M-Netz-Darstellungen in den nachfolgenden Kapiteln geliefert.

3.1 Stellen

Stellen eines M-Netzes werden durch Kreise dargestellt. Ihre Attribute werden wie folgt gekennzeichnet:

- Der (eindeutige) Name wird serifenlos innerhalb des Kreises eingetragen.
- Eingangsstellen werden hellgrau, interne Stellen normal weiß und Ausgangsstellen dunkelgrau dargestellt.

[1)] Programming Enviroment based on Petri nets

- Der Typ wird in `Schreibmaschinenschrift` angegeben.

- Die Referenz, die einen Bezug zum Text der SDL-Spezifikation herstellt und die Funktion der Stelle im Rahmen der Übersetzung verdeutlicht, wird in **Fettdruck** dargestellt. Referenzen bestehen aus einer Nummer und einem Index. Die Nummer gibt an, zu welchem Quelltext-Objekt eine Stelle gehört, der Index gibt an, welche Funktion die Stelle für die Übersetzung dieses Quelltext-Objektes einnimmt. Referenzen werden in der Form **nummer:index** angegeben.

Die nachstehende Abbildung 3.1 zeigt eine interne Stelle mit Namen P1, dem Typ `{1..3} X {6..8}` und der Referenz **ref:stack** (wobei **ref** etwa die Referenz-Nummer eines SDL-Prozesses sei).

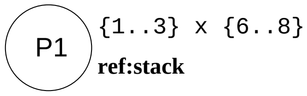

Abbildung 3.1: Beispiel für eine Stelle

3.2 Transitionen

Transitionen werden durch Quadrate mit folgenden Beschriftungselementen dargestellt:

- Der eindeutige Name wird serifenlos innerhalb des Quadrats verzeichnet.

- Die Schaltbedingung wird in `Schreibmaschinenschrift` eingetragen. Ist die Bedingung nicht explizit dargestellt, so ist sie `tt`, d.h. stets wahr.

- Mengen von Aktionstermen werden *kursiv* dargestellt. Nicht dargestellte Aktionsmengen können als leer angenommen werden. Die Mengenklammern werden i.a. weggelassen.

- Die Referenz, deren Bedeutung und Format analog zu der von Stellen ist, wird ebenso in **Fettdruck** dargestellt.

In Abbildung 3.2 ist eine Transition mit Namen T1, der Schaltbedingung `a>0`, einem Aktionsterm $A(c, d)$ und der Referenz **ref:atomic** zu sehen.

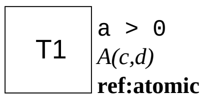

Abbildung 3.2: Beispiel für eine Transition

3.3 Kanten

Kantenbeschriftungen sind in `Schreibmaschinenschrift` eingetragen, wobei Mengenklammern weggelassen werden.

3.4 Sonderzeichen

Da der Netzeditor zur Repräsentation von Ausdrücken nur ASCII-Symbole benutzt, weicht die Darstellung einiger Zeichen zwischen dieser Ausarbeitung und den tatsächlich verwendeten Netzen voneinander ab. Tabelle 3.1 zeigt, welche Sonderzeichen durch besondere ASCII-Repräsentationen ersetzt wurden:

Sonderzeichen	ASCII-Darstellung
$\bullet$	`dot`
true	`tt`
false	`ff`
$\neg$	`!`
$\wedge$	`&`
$\vee$	`|`
$\neq$	`#`
$\leq$	`<=`
$\geq$	`>=`
mod	`%`
$\Rightarrow$	`=>`

Tabelle 3.1: Liste der verwendeten Sonderzeichen

Kapitel 4

Übersetzung

In diesem Kapitel wird eine M-Netz-Semantik für SDL-Spezifikationen gegeben, indem beliebigen SDL-Systemen S ein M-Netz $\zeta(S)$ zugeordnet wird. Die Semantik ist (weitgehend[1)]) kompositionell, d.h., die Semantik eines Systems ergibt sich durch die Zusammensetzung der Semantik seiner Einzelteile. Dies erleichtert nicht nur die Angabe der Semantik, sondern auch ihre Implementation.

Große Teile dieses Kapitels basieren auf der in [FG98a] vorgestellten M-Netz-Semantik für SDL-Systeme. Erweiterungen waren nötig für Anweisungen innerhalb von Prozeß- bzw. Prozedurrümpfen; auf weitere Unterschiede wird im nächsten Kapitel eingegangen.

Von einiger Wichtigkeit für die Anwendbarkeit von Verifikationsalgorithmen ist die Tatsache, daß die Semantik "sichere" M-Netze liefert, d.h. solche, in denen niemals zwei Marken gleichen Wertes zugleich auf derselben Stelle liegen können.

Bei der Übersetzung wurde die Maßgabe beachtet, größtmögliche Konformität zur Z.100 ([CCI92]) einzuhalten. Insbesondere werden einzelne Aktionen als atomare Anweisungen behandelt. Abweichungen zur Z.100 werden im nächsten Kapitel hervorgehoben.

4.1 Allgemeines

Um die detaillierteren Ausführungen zu einzelnen Sprachelementen verständlicher zu machen, seien einige allgemeine Erläuterungen vorangeschickt.

4.1.1 Kontrollflußmarken

Um festzuhalten, welche Anweisungen bei der Simulation eines SDL-Systems als nächste ausgeführt werden sollen, werden Kontrollflußmarken von einer Stelle zur nächsten weitergereicht. Im globalen Kontrollfluß reicht hierfür eine einzige Marke. Innerhalb von Prozessen hingegen muß die Möglichkeit gegeben sein, zwischen

[1)] siehe auch Abschnitt 5.1

unterschiedlichen Prozeßinstanzen zu unterscheiden, d.h., es müssen Marken mit unterschiedlichen Werten weitergegeben werden. Für jede Prozeßinstanz wird daher eine eigene Identifikations-Nummer (kurz ID) erzeugt, die im Kontrollfluß weitergereicht wird. Bei Prozeduren müssen neben Prozeßinstanzen noch Prozedurinstanzen unterschieden werden.

Um die Semantik von Aktionen innerhalb von Prozessen und Prozeduren einheitlich zu gestalten, wird folgende Vereinbarung getroffen:

- Im globalen Kontrollfluß wird die Marke $\bullet$ benutzt.
- Innerhalb von Prozessen wird das Tupel $(\bullet, \pi id)$ benutzt, wobei πid für Prozeß-ID steht.
- Innerhalb von Prozeduren wird das Tupel $(pid, \pi id)$ benutzt, wobei pid die ID der aktiven Prozedurinstanz und πid die Prozeßinstanz des aufrufenden Prozesses ist.

Prozesse benutzen also $\bullet$ anstelle einer Prozedur-ID.

Prozeß-IDs werden auch bei der Kommunikation zwischen Prozessen verwendet, um Sender und Empfänger eines Signals zu identifizieren. Aus diesem Grunde müssen Prozeß-IDs systemweit eindeutig sein. Um dies zu gewährleisten, wird bei der Vergabe von IDs ein globaler Zähler benutzt. Sollen wie im Beispiel von einem Prozeßtyp `one` zwei Instanzen möglich sein und von einem anderen Prozeßtyp `two` eine Instanz, so werden die IDs $\{1, 2\}$ für Instanzen von `one` und $\{3\}$ für die von `two` reserviert. Die speziellen IDs $\bullet$ und 0 werden für die vordefinierte SDL-Konstante **Null** bzw. für die Umgebung des SDL-Systems als Absender einer Nachricht verwendet.

Prozeduren werden hingegen nur innerhalb von Prozessen definiert und treten für die Umgebung eines Prozesses nicht in Erscheinung. Verschiedene Prozedurtypen müssen also nicht voneinander unterschieden werden. Daher bekommt jede Prozedur P die Prozedur-IDs $\{1, \ldots, MAX(P)\}$ zugewiesen, wobei $MAX(P)$ für die maximale Anzahl an Instanzen von P stehe.

4.1.2 Darstellung von Signalen

In SDL bekommen Signale einen benutzerdefinierten Namen (wie z.B. `sig` oder `ack`) und werden über diesen angesprochen. Die für PEP vorhandenen Editoren und andere Programme lassen als Marken jedoch nur ganze Zahlen und die speziellen Werte $\bullet$, True und False zu. Es ist also zu überlegen, welche Strategie zur Darstellung und Unterscheidung von Signalen in M-Netzen zu verwenden ist. Dabei spielen folgende Überlegungen eine Rolle:

Jeder Leitung (Kanal oder Signalroute) ist eine Menge von Signalen zugeordnet, die durch diese übermittelt werden können. Um ein Signal vom Sender- zum Empfänger-Prozeß zu befördern, muß es u.U. mehrere Leitungen durchlaufen, die neben diesem Signal unterschiedliche andere Signale transportieren können. Da die Schnittstellen zwischen Leitungen möglichst einfach sein und keine komplizierte Umrechnungen erfordern sollen, scheint es sinnvoll, Signalen global (d.h. systemweit) eindeutige Identifikatoren zuzuweisen, um sie im Netz zu repräsentieren.

Als Identifikatoren kommen aufgrund der Beschränkungen der PEP-Werkzeuge nur ganze Zahlen in Frage; wir beschränken uns dabei auf die natürlichen Zahlen.

Eine Komplikation ergibt sich dadurch, daß Signale mit einem oder mehreren Argumenten versehen sein können. Als Typen dieser Argumente kommen alle in SDL zugelassenen Mengen in Frage, d.h. innerhalb der von uns gewählten Beschränkungen die endlichen Teilmengen der ganzen Zahlen, Boolsche Werte und Prozeß-IDs. Da alle diese Mengen endlich sind, kann ein Signal mit Argumenten einfach durch eine Menge von Identifikatoren repräsentiert werden, wobei jeder Identifikator eine bestimmte Kombination von Argumenten bedeutet.[2)]

Eine Bijektion f_S zwischen Signalen eines SDL-Programms und natürlichen Zahlen wird dann wie folgt hergestellt: Ähnlich wie bei Prozeß-IDs wird ein globaler Zähler benutzt, um Signalen in der Reihenfolge ihres Auftretens in einer SDL-Spezifikation disjunkte Mengen von Zahlen, beginnend bei 0, zuzuordnen. Diese Mengen bestehen bei Signalen ohne Argument aus einer einzigen Zahl, bei Signalen mit Argumenten ist ihre Mächtigkeit gleich dem Produkt der Mächtigkeiten der einzelnen Argumenttypen.

Beispiel: Enthält eine Spezifikation das Fragment

```
signal foo(Boolean,{0,1}3)), bar;
```

so ist

$$f_S(\mathtt{foo}(\mathtt{False}, 0)) = 0,$$
$$f_S(\mathtt{foo}(\mathtt{True}, 0)) = 1,$$
$$f_S(\mathtt{foo}(\mathtt{False}, 1)) = 2,$$
$$f_S(\mathtt{foo}(\mathtt{True}, 1)) = 3 \text{ und}$$
$$f_S(\mathtt{bar}) = 4.$$

Die Gesamtheit aller einem Signal s zugeordneten Zahlen sei bezeichnet mit $f_S^*(s)$, d.h. im Beispiel $f_S^*(\mathtt{foo}) = \{0, 1, 2, 3\}$ und $f_S^*(\mathtt{bar}) = \{4\}$.

Der spezielle Wert $\bullet$ wird in einigen Fällen verwendet, um den Zustand "kein Signal vorhanden" auszudrücken.

4.1.3 Referenzen

Wie bereits erwähnt, bestehen Referenzen aus einer Nummer und einem Index, die hier verwendete Notation ist **nummer:index**. Die Nummer identifiziert eindeutig das Quelltext-Objekt, auf das sich die Referenz bezieht. Zu diesem Zweck werden bei der Übersetzung alle Kanäle, Signalrouten, Prozesse, Aktionen etc. bei 1 startend durchnumeriert; es besteht quasi eine Bijektion f_R zwischen Elementen des SDL-

[2)] Wären abzählbar unendliche Datentypen als Argumente zugelassen, so würde dieses Prinzip mit einigen Komplikationen noch immer funktionieren; bei überabzählbaren Datentypen jedoch nicht mehr.

[3)] `{0,1}` ist keine korrekte SDL-Syntax, wird hier aber der Einfachheit halber verwendet, da ein korrektes Beispiel komplizierter wäre.

Quelltextes und Referenznummern. (Dieses Prinzip wurde bereits in dem von Ackermann implementierten $\mathrm{B(PN)}^2$-Compiler verwendet, siehe [Gra97c] und [Ack97].)

Der Index einer Referenz gibt genauere Auskunft über die Funktion einer Stelle oder Transition in Bezug auf das referenzierte Objekt. Unterschiedliche Arten von Objekten werden mit unterschiedlichen Indizes referenziert. Indizes werden im folgenden mit symbolischen Namen wie **init**, **stack**, **atomic** usw. dargestellt, während im Übersetzer numerische Konstanten verwendet werden. Eine Zuordnung von symbolischen Namen zu numerischen Konstanten sowie eine Auflistung aller verwendeten Indizes findet sich in Abschnitt 6.2.2 im Rahmen der Formatbeschreibung von Referenzdateien.

4.1.4 Darstellung von Zuständen

Ähnlich Signalen werden die Zustände von Prozessen oder Prozeduren durch benutzerdefinierte Namen angesprochen und müssen bei der Übersetzung in Netze durch numerische Werte ersetzt werden. Dazu werden in jedem Prozeß und in jeder Prozedur Zustände in der Reihenfolge ihres ersten Auftretens bei 1 beginnend durchnumeriert. Diese Numerierung wird formal durch eine bijektive Funktion f_Q ausgedrückt. Die speziellen Werte $\bullet$ und 0 stehen für “Prozeß/Prozedur befindet sich in einem Zustandsübergang” bzw. “Prozeß/Prozedur terminiert”. Bei [FG98a] wird für letzteren Zustand auch das Zeichen † verwendet.

4.1.5 Erläuterungen zur Semantik

In den folgenden Abschnitten wird die formale Semantik eines SDL-Programms (d.h. eines Systems) festgelegt. Genauer gesagt wird, basierend auf der Grammatik von SDL, jedem Programmfragment P eine Semantik $\zeta(P)$ zugeordnet.

Neben ζ werden einige Hilfsfunktionen definiert:

- $\alpha(P)$ steht für eine Menge von Aktionssymbolen, die durch das Programmfragment P erzeugt werden, und über die das Netz auf höherer Ebene synchronisiert wird. Bei der Synchronisation über ein Aktionssymbol A werden (grob gesagt) Transitionen, die mit dem Symbol A versehen sind, mit solchen Transitionen vereinigt, die das Symbol $\overline{A}$ tragen. Bei jeder Vereinigung entsteht eine neue Transition, die dieses Symbol nicht mehr trägt. Eine formale Definition der Synchronisationsoperation wird in [Gra98b] gegeben.
  Wird ein Netz über eine Menge von Aktionssymbolen synchronisiert, so schließt daran gewöhnlich die sogenannte Restriktion an. Bei der Restriktion über ein Aktionssymbol A werden alle Transitionen entfernt, die mit A versehen sind. Auch hier ist eine formale Definition in [Gra98b] zu finden. Im folgenden wird der Begriff “Synchronisation” gleichbedeutend mit “Scoping” (d.h. Synchronisation mit anschließender Restriktion) verwendet, um sprachliche Probleme zu vermeiden.

- Entsprechend steht $\alpha_{global}(P)$ für die Menge der durch P erzeugten Aktionssymbole, bezüglich derer am Schluß der Übersetzung das gesamte Netz synchronisiert wird.
- $\alpha_{term}(P)$ sammelt die Aktionssymbole für die Terminierung aller Prozesse.
- κ_{init}, κ_{create} und κ_{term} stehen für Teile des globalen Kontrollflusses (siehe Erläuterung bei der System-Semantik in Abschnitt 4.2.1).

Zur Initialisierung und Terminierung von Kanälen, Prozessen u.a. werden einige einfache Netze benötigt. Abbildung 4.1 zeigt $M_P(X)$, wobei $P \in \{init, term\}$ ist. M_{leer} stehe für ein leeres Netz.

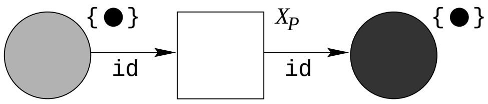

Abbildung 4.1: Das Netz $M_P(X)$

In den Regeln der Grammatik werden SDL-Schlüsselwörter **fett** und Nicht-Terminale normal gedruckt. Namen von Variablen o.ä. werden in `Schreibmaschinenschrift` dargestellt.

Häufiger Bestandteil der Grammatik sind Listen. Da die Regeln für Listen stets gleich sind, sei hiermit folgende "Meta-Regel" vereinbart:

element–list ::= *element* | *element* *element*–list

Im folgenden steht also (wenn nichts anderes angegeben wird) ein Nicht-Terminal der Form *element*–list für ein oder mehrere Vorkommen des Nicht-Terminals *element*. Eine ähnliche Regel wird für Komma-getrennte Listen aufgestellt:

element–klist ::= *element* | *element* **,** *element*–klist

4.2 Semantik

In diesem Abschnitt werden Syntax und M-Netz-Semantik von SDL (nach Sinnabschnitten geordnet) parallel zueinander entwickelt. Dabei wird "top-down" vorgegangen, d.h., es werden zunächst übergeordnete Syntaxregeln vorgestellt, die dann in späteren Abschnitten verfeinert werden.

4.2.1 System

Eine SDL-Spezifikation definiert ein System, bestehend aus Blöcken und Kanälen, die die Blöcke untereinander und mit der Umgebung des Systems verbinden.

system	::=	system–header system–decl–list system–trailer
system–header	::=	**system** `system–ident` **;**
system–decl	::=	block | channel | signaldef | constdef | typedef
system–trailer	::=	**endsystem** opt–id **;**
opt–id	::=	`ident` | ε

Der nach **endsystem** angegebene Name muß mit dem nach **system** genannten Namen übereinstimmen, wenn er angegeben wird; er kann jedoch auch weggelassen werden. Diese Regel gilt analog für alle anderen mit **end...** beginnenden Schlüsselwörter.

Die Semantik eines Systems (siehe die folgenden Gleichungen) erhält man, indem man zunächst die parallele Komposition aller aus Blöcken und Kanälen entstandenen M-Netze bildet und Synchronisation/Restriktion über die in α gesammelten Symbole durchführt. Anschließend wird der globale Kontrollfluß hinzugefügt und das Ergebnis bezüglich folgender Aktionen synchronisiert, die mit der Funktion α_{global} berechnet werden:

- Initialisierungs- und Terminierungsaktionen von Verbindungen (connects), Prozeduren, Kanälen und Signalrouten,
- Initialisierungs-, Terminierungs- und Erzeugungsaktionen von Prozessen.

Der globale Kontrollfluß besteht aus einer vierteiligen Sequenz von Netzen, deren Transitionen folgende Aufgaben erfüllen:

1. ein mit κ_{init} bezeichneter Teil, der nacheinander sämtliche Kanäle, Signalrouten, Verbindungen, Prozesse und Prozeduren initialisiert;
2. ein mit κ_{create} bezeichneter Teil, der nacheinander alle anfänglichen Prozeßinstanzen erzeugt;
3. eine Transition (siehe Abbildung 4.2), die alle Prozesse zugleich terminiert;
4. ein mit κ_{term} bezeichneter Teil, der nacheinander die Terminierung aller Kanäle, Signalrouten, Verbindungen und Prozeduren durchführt.

$$
\begin{aligned}
&\zeta(\text{system–header system–decl–list system–trailer})\\
&\qquad = \Big[\alpha_{global}(\text{system–decl–list}) :\\
&\qquad\qquad \zeta(\text{system–decl–list})\\
&\qquad\qquad\quad || \Big(\kappa_{init}(\text{system–decl–list});\ \kappa_{create}(\text{system–decl–list});\\
&\qquad\qquad\qquad M_{prcsterm}(\alpha_{term}(\text{system–decl–list}));\ \kappa_{term}(\text{system–decl–list})\Big)\Big]
\end{aligned}
$$

$$
\begin{aligned}
&\zeta(\text{system–decl system–decl–list})\\
&\qquad = [\alpha(\text{system–decl}) : \zeta(\text{system–decl}) \mid\mid \zeta(\text{system–decl–list})]
\end{aligned}
$$

$$
\begin{aligned}
&\alpha_P(\text{system–decl system–decl–list})\\
&\qquad = \alpha_P(\text{system–decl}) \cup \alpha_P(\text{system–decl–list}), \qquad P \in \{global, term\}
\end{aligned}
$$

$$
\begin{aligned}
&\kappa_P(\text{system–decl system–decl–list})\\
&\qquad = \kappa_P(\text{system–decl}); \kappa_P(\text{system–decl–list}), \qquad P \in \{init, create, term\}
\end{aligned}
$$

Die hier noch nicht aufgelösten Ausdrücke werden in den nachfolgenden Abschnitten erklärt.

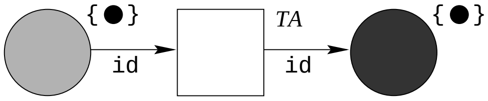

Abbildung 4.2: Das Netz $M_{prcsterm}(TA)$

4.2.2 Definitionen

Definitionen von Signalen, Konstanten oder Typen erzeugen keine Netze. Der Gültigkeitsbereich einer Definition ist das System, der Block, Prozeß etc., innerhalb dessen sie steht. Dieser Abschnitt gibt im wesentlichen die relevanten Grammatikregeln wieder.

Signale

Die zur Kommunikation zwischen Prozessen verwendeten Nachrichten werden in SDL Signale genannt. Sie dienen außerdem der Übermittlung von Daten und können zu diesem Zweck ein oder mehrere Argumente tragen.

signaldef	::=	**signal** sigdef–klist **;**
sigdef	::=	`signal–ident` | `signal–ident` **(** `type–ident`–klist **)**

Mit einem **signal**-Schlüsselwort können also mehrere Signale definiert werden. Der anstelle von `signal–ident` gegebene Name dient als Bezeichner für das Signal. Die in `type–ident`–klist aufgeführten Typen geben an, was für Argumente das Signal trägt. Argumente müssen endlichen Typs sein; es muß sich entweder um einen der vordefinierten Typen **Boolean** und **PId** oder um einen zuvor mit **syntype** definierten (endlichen) Typ handeln (siehe unten unter "Typdefinitionen").

$$\zeta(\text{signaldef}) = M_{leer}$$
$$\alpha(\text{signaldef}) = \alpha_{global}(\text{signaldef}) = \alpha_{term}(\text{signaldef}) = \emptyset$$
$$\kappa_P(\text{signal}) = M_{leer}, \qquad P \in \{init, create, term\}$$

Konstanten

Mit Konstanten können symbolische Namen für Werte vereinbart werden. Jedes weitere Auftreten des Namens wird dann durch den zugehörigen Wert ersetzt. Eine in SDL vordefinierte Konstante vom Typ **PId** ist **Null** und steht für "kein Prozeß".

constdef	::=	**synonym** synonymdef–klist **;**
synonymdef	::=	`const–ident` opt–id **=** const–expr

Durch Angabe von opt–id kann der Typ der Konstanten angegeben werden. opt–id muß einer der vordefinierten Typen **Natural** (natürliche Zahl), **Integer** (ganze Zahl), **Boolean** oder **PId** (Prozeß-ID) oder ein zuvor mit **syntype** definierter Typ sein.

Anstelle von einfachen Werten können vollständige Ausdrücke benutzt werden, in denen allerdings keine Variablen vorkommen dürfen. Die vollständige Definition von Ausdrücken und von const–expr ist in Abschnitt 4.2.13 zu finden.

$$\zeta(\text{constdef}) = M_{leer}$$
$$\alpha(\text{constdef}) = \alpha_{global}(\text{constdef}) = \alpha_{term}(\text{constdef}) = \emptyset$$
$$\kappa_P(\text{constdef}) = M_{leer}, \qquad P \in \{init, create, term\}$$

Typdefinitionen

Mit dem Schlüsselwort **syntype** können Bezeichner für Untertypen von bereits existierenden Datentypen vereinbart werden.

typedef	::=	**syntype** `subtype–ident = type–ident` opt–default opt–constants **endsyntype** opt–id **;**
opt–default	::=	**default** const–expr | ε
opt–constants	::=	**constants** range–klist | ε
range	::=	const–expr | const–expr **:** const–expr | operator const–expr
operator	::=	**=** | $\neq$ | $<$ | $\leq$ | $>$ | $\geq$

Die Wertemenge von `subtype–ident` ist die Schnittmenge der unter opt-constants aufgeführten Werte mit der Wertemenge von `type–ident` oder, falls opt-constants leer ist, die gesamte Wertemenge von `type–ident`. `type–ident` kann ein beliebiger vordefinierter oder zuvor ebenfalls mit **syntype** definierter Typ sein.

Ein durch opt-default angegebener Wert bestimmt, daß Variablen des vereinbarten Typs `subtype–ident` mit diesem initialisiert werden.

$$\zeta(\text{typedef}) = M_{leer}$$
$$\alpha(\text{typedef}) = \alpha_{global}(\text{typedef}) = \alpha_{term}(\text{typedef}) = \emptyset$$
$$\kappa_P(\text{typedef}) = M_{leer}, \qquad P \in \{init, create, term\}$$

4.2.3 Kanäle

Die Übermittlung von Signalen zwischen Prozessen findet in SDL durch Kanäle und Signalrouten statt. Kanäle modellieren dabei den Fall einer Verbindung zwischen physikalisch getrennten (und möglicherweise weit voneinander entfernten) Einheiten. Kanäle verbinden entweder zwei Blöcke miteinander oder einen Block mit der Umgebung des Systems. Die Übertragung eines Signals erfolgt mit zufälliger, nicht konstanter Verzögerung, wobei jedoch gewährleistet ist, daß Signale in der Reihenfolge ihres Eintreffens weitergeleitet werden (ein Kanal arbeitet also als FIFO-Puffer). Kanäle können uni- oder bidirektional sein; jeder Richtung ist eine Liste von Signalen zugeordnet, die auf diesem Weg übermittelt werden kann.

channel	::=	channel–header ch–path channel–trailer |
		channel–header ch–path ch–path channel–trailer
channel–header	::=	**channel** `channel–ident`
channel–trailer	::=	**endchannel** opt-id **;**
ch–path	::=	**from** ch-endpoint **to** ch-endpoint **with** `signal–ident`–klist **;**
ch–endpoint	::=	`block–ident` | **env**

In bidirektionalen Kanälen müssen die Endpunkte des zweiten Pfades die Richtung des ersten Pfades umkehren. Bidirektionale Kanäle können wie zwei unabhängige unidirektionale Kanäle betrachtet werden und werden auch so übersetzt. Die Endpunkte eines Pfades müssen unterschiedlich sein. Das Schlüsselwort **env** steht für die Umgebung des Systems.

Um die unterschiedlichen Richtungen bei bidirektionalen Kanälen auseinanderzuhalten, werden die Aktionssymbole mit Indizes 0 und 1 für Hin- bzw. Rückrichtung versehen (siehe unten).

$$\zeta(\textbf{channel}\ \texttt{channel–ident}\ \text{ch–path channel–trailer})$$
$$= M_{channel}(\texttt{channel–ident}, \text{ch–path}, 0, f_R(\texttt{channel–ident}))$$
$$\zeta(\textbf{channel}\ \texttt{channel–ident}\ \text{ch–path}_0\ \text{ch–path}_1\ \text{channel–trailer})$$
$$= M_{channel}(\texttt{channel–ident}, \text{ch–path}_0, 0, f_R(\texttt{channel–ident})) \\ \quad || \ M_{channel}(\texttt{channel–ident}, \text{ch–path}_1, 1, f_R(\texttt{channel–ident}))$$

$$\alpha(\textbf{channel}\ \texttt{channel–ident}\ \text{ch–path channel–trailer})$$
$$= \{\texttt{channel–ident}_0!\}$$
$$\alpha(\textbf{channel}\ \texttt{channel–ident}\ \text{ch–path ch–path channel–trailer})$$
$$= \{\texttt{channel–ident}_0!, \texttt{channel–ident}_1!\}$$
$$\alpha_{global}(\textbf{channel}\ \texttt{channel–ident}\ \text{ch–path channel–trailer})$$
$$= \{\texttt{channel–ident}_{0init}, \texttt{channel–ident}_{0term}\}$$
$$\alpha_{global}(\textbf{channel}\ \texttt{channel–ident}\ \text{ch–path ch–path channel–trailer})$$
$$= \{\texttt{channel–ident}_{0init}, \texttt{channel–ident}_{0term}, \\ \texttt{channel–ident}_{1init}, \texttt{channel–ident}_{1term}\}$$
$$\alpha_{term}(\text{channel}) = \emptyset$$

$$\kappa_P(\textbf{channel}\ \texttt{channel–ident}\ \text{ch–path channel–trailer})$$
$$= M_P(\texttt{channel–ident}_0), \qquad P \in \{init, term\}$$
$$\kappa_P(\textbf{channel}\ \texttt{channel–ident}\ \text{ch–path ch–path channel–trailer})$$
$$= M_P(\texttt{channel–ident}_0); M_P(\texttt{channel–ident}_1), \qquad P \in \{init, term\}$$

$$\kappa_{create}(\text{channel}) = M_{leer}$$

Abbildung 4.3 gibt das Netz $M_{channel}$ wieder. Das Netz ist parametrisiert durch folgende Größen:

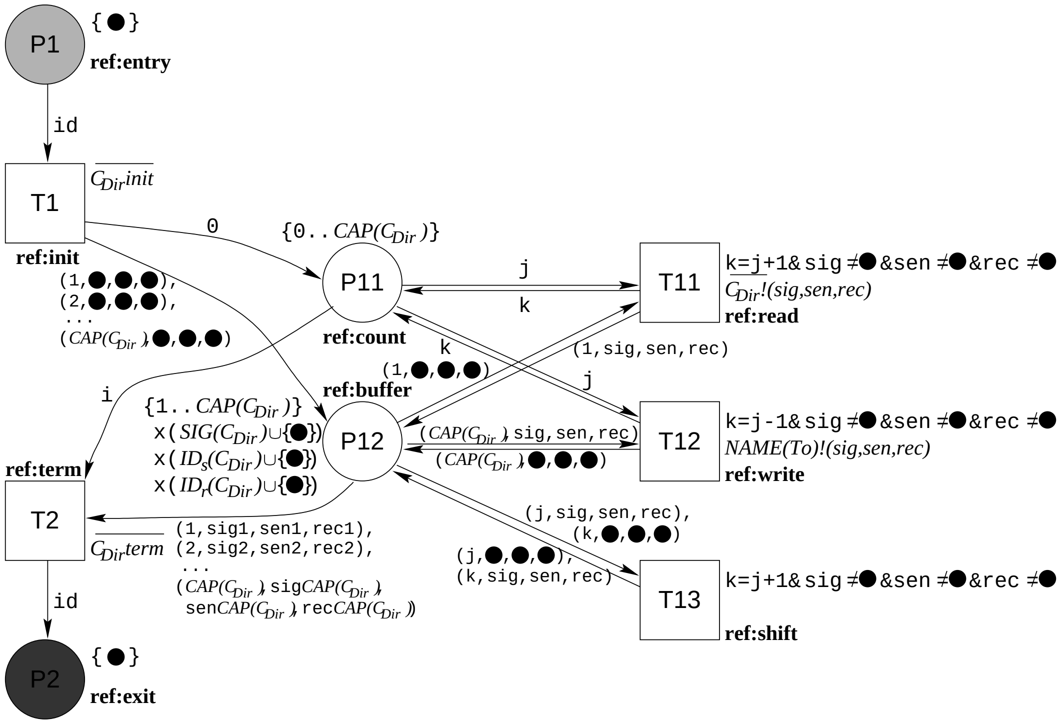

Abbildung 4.3: $M_{channel}(C, \textbf{from}\ From\ \textbf{to}\ To\ \textbf{with}\ SigList, Dir, \textbf{ref})$

- Den Namen des Kanals (C) und einen Richtungsindex Dir. Dabei steht 0 für die Hinrichtung, 1 für die Rückrichtung. Name und Index werden für viele Zwecke der Übersetzung zu einem Bezeichner C_{Dir} verschmolzen.

- Den Zielknoten des Kanals. In Abbildung 4.3 wird die Funktion $NAME(To)$ gebraucht, um den für das Ziel benutzen Aktionsnamen zu ermitteln. Ist der Endpunkt die Umgebung des Systems, so wird dieser aus dem Namen des Systems und einem fortlaufenden Zähler zusammengesetzt, um so einen eindeutigen Namen zu erhalten (auf diese Weise werden Schnittstellen zur Umgebung des Systems geschaffen). Andernfalls handelt es sich beim Zielknoten um einen Block. Dann wird festgestellt, zu welchem Verbindungsknoten des Blockes der Kanal führt, und *NAME* liefert den Namen dieser Verbindung (siehe Abschnitt 4.2.6). Der Name des Startknotens wird zur Erzeugung des Netzes *nicht* gebraucht.

- Die Kapazität $CAP(C)$ des Eingabe-Puffers. Diese ist nicht Teil der SDL-Syntax (in [CCI92] wird von unbeschränkten Kapazitäten ausgegangen). Die Möglichkeiten, $CAP(C)$ zu beeinflussen, sind in Kapitel 6 dargestellt.

- Die Menge der Signale, die mit C in Richtung Dir übertragen werden können. Die Zuordnung von Signalen zu natürlichen Zahlen ist in Abschnitt 4.1.2 erläutert worden. Die Wertestelle des Kanals muß alle die Werte enthalten, die den in `signal–ident`–klist vorkommenden Signalen zugeordnet sind, d.h., aus

$$\texttt{signal–ident}\text{–klist} = \texttt{signal–ident}_1, \ldots, \texttt{signal–ident}_n$$

folgt

$$SIG(C_{Dir}) = \bigcup_{i=1}^{n} f_S^*(\texttt{signal–ident}_i).$$

- Die Menge der Sender und Empfänger, d.h. der Prozesse, die Signale über diesen Kanal senden bzw. empfangen können. Die Mengen der zugehörigen Prozeß-IDs werden mit $ID_s(C_{Dir})$ bzw. $ID_r(C_{Dir})$ bezeichnet. Diese Mengen lassen sich nicht kompositionell, sondern nur bei Betrachtung des gesamten Systems ermitteln; ihre Berechnung wird in Abschnitt 6.3 dargelegt.

Eine alternative Semantik zu der hier gewählten und eine Diskussion der Unterschiede findet sich in Kapitel 5.

4.2.4 Blöcke

Blöcke enthalten einen oder mehrere Prozesse, die durch Signalrouten untereinander oder mit der Umgebung des Blocks verbunden sind. Schnittstellen zur Umgebung werden durch Verbindungsknoten (connects) geschaffen.

block	::=	block–header block–decl–list block–trailer
block–header	::=	**block** `block–ident` **;**
block–trailer	::=	**endblock** opt-id **;**
block–decl	::=	process | signalroute | connect |
		signaldef | constdef | typedef

Blöcke sind eine organisatorische Einheit; ihre Semantik ergibt sich durch die Semantik ihrer Einzelteile.

$$\zeta(\text{block–header block–decl–list block–trailer}) = \zeta(\text{block–decl–list})$$
$$\alpha(\text{block–header block–decl–list block–trailer}) = \emptyset$$
$$\alpha_P(\text{block–header block–decl–list block–trailer})$$
$$= \alpha_P(\text{block–decl–list}), \qquad P \in \{global, term\}$$
$$\kappa_P(\text{block–header block–decl–list block–trailer})$$
$$= \kappa_P(\text{block–decl–list}), \qquad P \in \{init, create, term\}$$

$$\zeta(\text{block–decl block–decl–list})$$
$$= [\alpha(\text{block–decl}) : \zeta(\text{block–decl}) \mid\mid \zeta(\text{block–decl–list})]$$
$$\alpha_P(\text{block–decl block–decl–list})$$
$$= \alpha_P(\text{block–decl}) \cup \alpha_P(\text{block–decl–list}), \qquad P \in \{global, term\}$$
$$\kappa_P(\text{block–decl block–decl–list})$$
$$= \kappa_P(\text{block–decl}); \kappa_P(\text{block–decl–list}), \qquad P \in \{init, create, term\}$$

4.2.5 Signalrouten

Ähnlich wie Kanäle dienen Signalrouten der Übermittlung von Signalen zwischen Prozessen. Im Unterschied zu diesen geschieht die Übermittlung jedoch ohne Verzögerung. Signalrouten verbinden zwei Prozesse oder einen Prozeß mit der Umgebung eines Blockes.

signalroute	::=	signalroute–header sr–path |
		signalroute–header sr–path sr–path
signalroute–header	::=	**signalroute** `route–ident`
sr–path	::=	**from** sr–endpoint **to** sr–endpoint **with** `signal–ident`–klist **;**
sr–endpoint	::=	`process–ident` | **env**

Die Semantik ist analog zu der von Kanälen. Auch hier wird ein Index zur Kennzeichnung von Hin- und Rückrichtung verwendet.

$$
\begin{aligned}
&\zeta(\textbf{signalroute}\ \texttt{route–ident}\ \text{sr–path})\\
&\qquad = M_{route}(\texttt{route–ident}, \text{sr–path}, 0, f_R(\texttt{route–ident}))\\
&\zeta(\textbf{signalroute}\ \texttt{route–ident}\ \text{sr–path}_0\ \text{sr–path}_1)\\
&\qquad = M_{route}(\texttt{route–ident}, \text{sr–path}_0, 0, f_R(\texttt{route–ident}))\\
&\qquad\qquad || \ M_{route}(\texttt{route–ident}, \text{sr–path}_1, 1, f_R(\texttt{route–ident}))
\end{aligned}
$$

$$
\begin{aligned}
&\alpha(\textbf{signalroute}\ \texttt{route–ident}\ \text{sr–path})\\
&\qquad = \{\texttt{route–ident}_0!\}\\
&\alpha(\textbf{signalroute}\ \texttt{route–ident}\ \text{sr–path sr–path})\\
&\qquad = \{\texttt{route–ident}_0!, \texttt{route–ident}_1!\}\\
&\alpha_{global}(\textbf{signalroute}\ \texttt{route–ident}\ \text{sr–path})\\
&\qquad = \{\texttt{route–ident}_{0init}, \texttt{route–ident}_{0term}\}\\
&\alpha_{global}(\textbf{signalroute}\ \texttt{route–ident}\ \text{sr–path sr–path})\\
&\qquad = \{\texttt{route–ident}_{0init}, \texttt{route–ident}_{0term},\\
&\qquad\qquad \texttt{route–ident}_{1init}, \texttt{route–ident}_{1term}\}\\
&\alpha_{term}(\text{signalroute}) = \emptyset
\end{aligned}
$$

$$
\begin{aligned}
&\kappa_P(\textbf{signalroute}\ \texttt{route–ident}\ \text{sr–path})\\
&\qquad = M_P(\texttt{route–ident}_0), \qquad P \in \{init, term\}\\
&\kappa_P(\textbf{signalroute}\ \texttt{route–ident}\ \text{sr–path sr–path})\\
&\qquad = M_P(\texttt{route–ident}_0); M_P(\texttt{route–ident}_1), \qquad P \in \{init, term\}\\
&\kappa_{create}(\text{signalroute}) = M_{leer}
\end{aligned}
$$

Abbildung 4.4 zeigt das Netz M_{route}. Die Tatsache, daß die Übertragung synchron (ohne Verzögerung) geschieht, vereinfacht das für Signalrouten gebrauchte Netz ganz erheblich. Es wird nur eine Transition zum gleichzeitigen Lesen und Schreiben gebraucht.

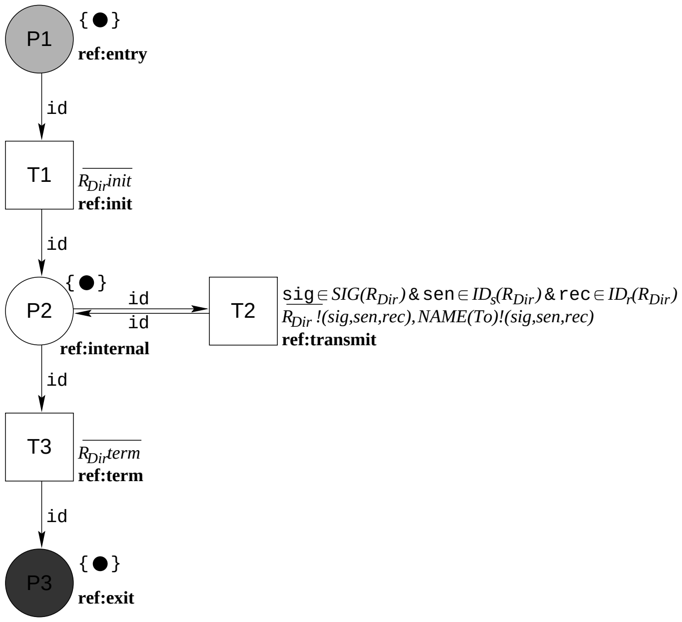

Abbildung 4.4: $M_{route}(R, \textbf{from}\ From\ \textbf{to}\ To\ \textbf{with}\ SigList, Dir, \textbf{ref})$

Die Parameter des Netzes sind ebenfalls ähnlich denen eines Kanal-Netzes mit folgenden Unterschieden:

- Der Name des Zielknotens kann die Umgebung des Blocks sein. In diesem Fall wird festgestellt, zu welchem Verbindungsknoten die Signalroute führt, und ein entsprechender Name wird erzeugt (siehe Abschnitt 4.2.6). Andernfalls handelt es sich beim Zielknoten um einen Prozess; dann ist $NAME$(`process–ident`) = `process–ident`.
- Da Signalrouten Nachrichten sofort weiterleiten und nicht zwischenspeichern, haben sie keine Kapazitätsbeschränkung.

4.2.6 Verbindungen

Verbindungen stellen eine Verknüpfung zwischen Signalrouten und Kanälen, die in einen Block hineingehen bzw. aus ihm herausführen, her. Für jeden Block B eines Systems muß gelten:

- Jeder Kanal, der B als einen seiner Endpunkte hat, muß in genau einer Verbindung von B auftauchen.
- Jede Signalroute in B, die die Umgebung des Blocks berührt, muß in genau einer Verbindung von B auftauchen.
- Alle in einer Verbindung auftauchenden Kanäle müssen B als Endpunkt haben, und alle auftretenden Signalrouten müssen die Umgebung von B berühren.
- Für jede Verbindung in B muß die Vereinigung der Signalmengen der Kanäle, die über diese Verbindung zu dem Block führen, gleich der Vereinigung der

Signalmengen der Signalrouten sein, die über diese Verbindung in den Block hineinführen; gleiches gilt analog für aus dem Block herausführende Leitungen.

connect ::= **connect** `channel`–`ident`–klist **and** `route`–`ident`–klist **;**

Die Semantik muß berücksichtigen, daß über eine Verbindung Signale in den Block hinein-, aber auch aus ihm herausfließen können. Daher besteht das Netz aus zwei nebeneinanderliegenden Teilen für je eine Richtung. Für jeden Kanal und jede Signalroute, der/die Signale durch diese Verbindung *empfangen* kann, wird eine Transition erzeugt.

$$
\begin{aligned}
&\zeta(\textbf{connect}\ \texttt{channel}\text{–}\texttt{ident}\text{–klist}\ \textbf{and}\ \texttt{route}\text{–}\texttt{ident}\text{–klist}\ ;)\\
&\quad = (1N^{in}_{cnstart} \cup 2N^{in}_{cnend} \cup 3N_{in}(\texttt{route}\text{–}\texttt{ident}\text{–klist}))\\
&\qquad\quad \oplus \otimes \{(1N^{in}_{cnstart})^{\bullet}, {}^{\bullet}(2N^{in}_{cnend}), {}^{\bullet}(3N_{in}(\texttt{route}\text{–}\texttt{ident}\text{–klist})),\\
&\qquad\qquad\qquad\qquad (3N_{in}(\texttt{route}\text{–}\texttt{ident}\text{–klist}))^{\bullet}\}\\
&\qquad || \ (1N^{out}_{cnstart} \cup 2N^{out}_{cnend} \cup 3N_{out}(\texttt{channel}\text{–}\texttt{ident}\text{–klist}))\\
&\qquad\quad \oplus \otimes \{(1N^{out}_{cnstart})^{\bullet}, {}^{\bullet}(2N^{out}_{cnend}), {}^{\bullet}(3N_{out}(\texttt{channel}\text{–}\texttt{ident}\text{–klist})),\\
&\qquad\qquad\qquad\qquad (3N_{out}(\texttt{channel}\text{–}\texttt{ident}\text{–klist}))^{\bullet}\}
\end{aligned}
$$

Abbildung 4.5 zeigt die Netze $N^{I}_{cnstart}$ und N^{I}_{cnend}, $I \in \{in, out\}$, die die Initialisierung und Terminierung der Verbindung bewerkstelligen. Das Netz $N^{I}_{cntransmit}$ wird benutzt, um die Netze N_{in} und N_{out} zu schaffen, indem für jeden Kanal und jede Signalroute, der/die Signale durch diese Verbindung empfängt, eine Transition an die Exit-Stelle von $N^{I}_{cnstart}$ gehängt wird. Abschnitt 5.5 im nächsten Kapitel liefert zusätzliche Erläuterungen und Beispiele zu diesem Thema, die Bedeutung von *NAME* wird weiter unten erklärt.

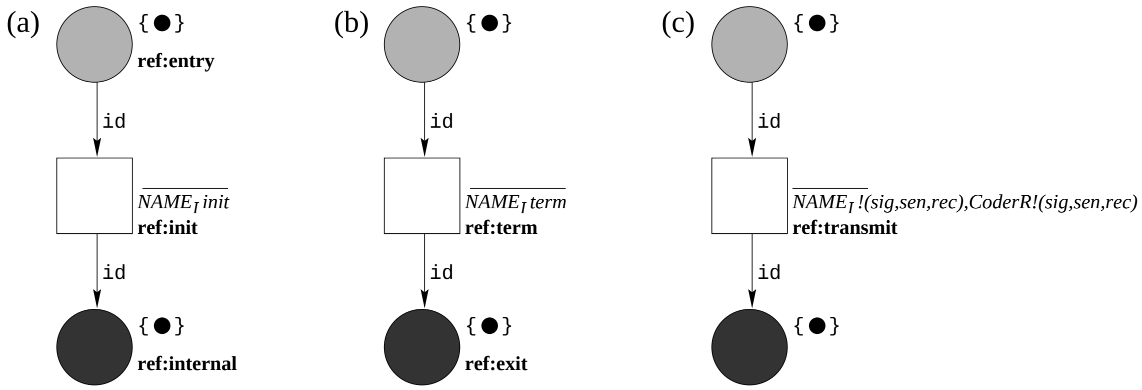

Abbildung 4.5: **(a)** $N^{I}_{cnstart}(\textbf{ref})$ **(b)** $N^{I}_{cnend}(\textbf{ref})$ **(c)** $N^{I}_{cntransmit}(CoderR, \textbf{ref})$

$$
N_{in}(\texttt{list}) = \mathop{[]}_{R \in IN(\texttt{list})} N^{in}_{cntransmit}(INDEXED(R), f_R(NAME))
$$
$$
N_{out}(\texttt{list}) = \mathop{[]}_{C \in OUT(\texttt{list})} N^{out}_{cntransmit}(INDEXED(C), f_R(NAME))
$$

Dabei sei $IN(\texttt{list})$ die Menge der Signalrouten in `list`, die von der Umgebung des Blockes zu einem Prozeß führen, d.h. solche, die Signale in den Block hineintransportieren. $INDEXED(R)$ liefere zu einer solchen Route den Namen r_{Dir} mit dem notwendigen Richtungsindex Dir, wie er bei der Semantik von Kanälen und Signalrouten eingeführt wurde. Analog zu IN sei OUT(`list`) die Menge der Kanäle in `list`, die Signale aus dem Block heraustransportieren. Abbildung 4.6 gibt das Zusammenwirken der Netze $N^I_{cnstart}$, N^I_{cnend} und $N^I_{cntransmit}$ schematisch wieder.

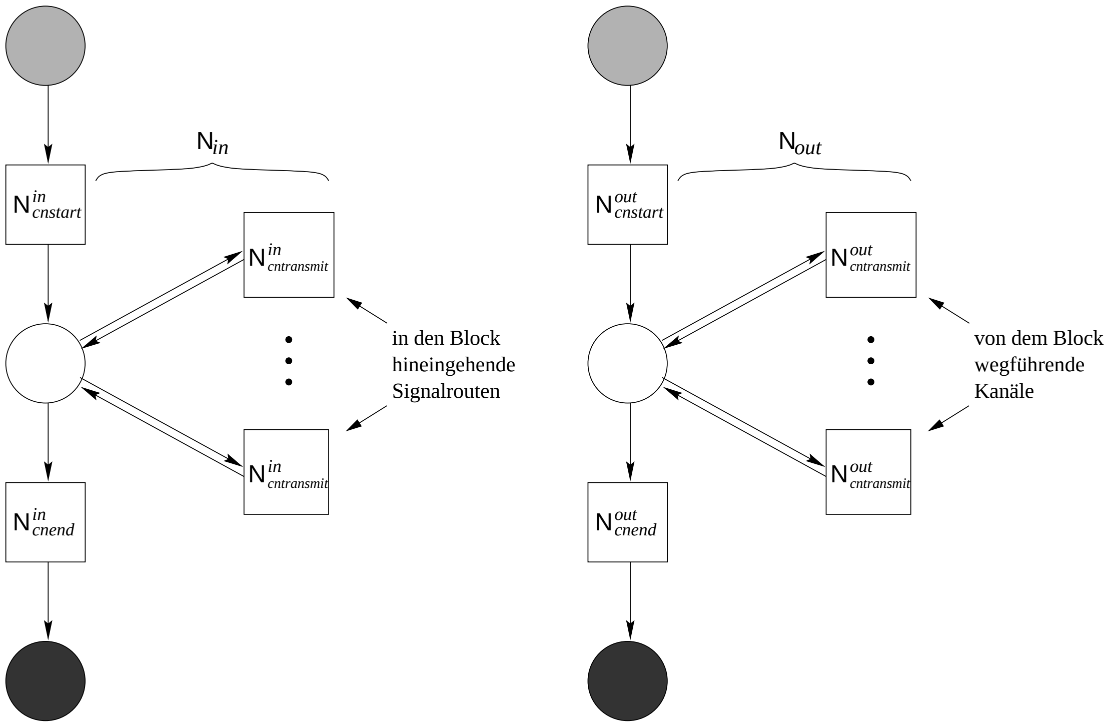

Abbildung 4.6: Schematische Darstellung der Semantik von Verbindungen

$$
\begin{aligned}
&\alpha(\text{connect}) = \{NAME_{out}!\} \\
&\alpha_{global}(\text{connect}) = \{NAME_{in}!\} \\
&\alpha_{term}(\text{connect}) = \emptyset \\
&\kappa_P(\text{connect}) = M_P(NAME_{in}); M_P(NAME_{out}), \qquad P \in \{init, term\} \\
&\kappa_{create}(\text{connect}) = M_{leer}
\end{aligned}
$$

Die Kommunikationsstrukturen einer SDL-Spezifikation werden im Petri-Netz dadurch abgebildet, daß die Lese-/Schreibtransitionen von Kanälen, Signalrouten, Prozessen und Verbindungen mit Aktionstermen versehen werden. Durch Synchronisation fallen dann die richtigen Transitionen zusammen. Verbindungen haben in SDL keinen Namen. Daher wird ein besonderes Schema benötigt, wie die für Verbindungen notwendigen Aktionssymbole genannt werden. Dazu wird eine eindeutige Kombination aus dem Namen des Blocks, zu dem die Verbindung gehört, und einem fortlaufenden Zähler benutzt. Diese Kombination wird in den obigen Definitionen durch $NAME$ repräsentiert.

Dieser Name wird u.a. benutzt, falls die Verbindung als Zielknoten eines Kanals oder einer Signalroute auftritt. Schickt ein Kanal Signale in einen Block, so wird

$NAME_{in}$ als Aktionsname für den Zielknoten benutzt, wobei $NAME$ entsprechend der benutzten Verbindung gesetzt wird. Entsprechend wird für Signalrouten, die Signale aus einem Block heraussenden, $NAME_{out}$ verwendet.

Wie aus den Definitionen von α und α_{global} hervorgeht, wird über das Aktionssymbol $NAME_{out}!$ auf Blockebene, über $NAME_{in}!$ jedoch erst auf Systemebene synchronisiert. Dies macht Sinn, da sich $NAME_{out}!$ mit Signalrouten innerhalb des Blocks verbindet, $NAME_{in}!$ jedoch mit Kanälen außerhalb des Blocks verschmilzt.

4.2.7 Prozesse

Prozesse sind die zentralen Bestandteile einer SDL-Spezifikation. Jedes Vorkommen eines Prozesses im Quelltext definiert einen Prozeßtyp, von dem eine Anzahl von Instanzen gebildet werden kann.

Jedem Prozeß ist eine anfängliche und eine maximale Anzahl von Instanzen zugeordnet; jede Instanz besitzt eine (systemweit eindeutige) Identifikationsnummer (ID). Prozesse können Signale entgegennehmen und selbst verschicken; eintreffende Signale werden zunächst in einer Eingabewarteschlange gespeichert.

Jeder Prozeßtyp besitzt eine Menge von Zuständen. Zustandsübergänge einer Instanz werden durch den Empfang eines Signals ausgelöst. Ein besonderer Zustandsübergang (ohne Erhalt eines Signals) findet bei der Erzeugung einer Instanz statt.

Prozesse können auf Variablen zugreifen und ihren Inhalt manipulieren. Jeder Prozeßinstanz ist dabei eine Instanz jeder Variablen zugeordnet. Zusätzlich gibt es für jede Instanz eine Reihe von vordefinierten Variablen:

- **self**: die Prozeß-ID der Instanz selber
- **parent**: die Prozeß-ID des erzeugenden Prozesses (= **Null**, falls es sich um eine anfängliche Instanz handelt)
- **offspring**: die Prozeß-ID des zuletzt von diesem Prozeß erzeugten Prozesses (= **Null**, falls die Erzeugung fehlschlug)
- **sender**: die Prozeß-ID desjenigen Prozesses, von dem zuletzt eine Nachricht aus der Eingabewarteschlange genommen wurde (= **self**, falls ein spontaner Zustandsübergang stattfand)

Darüber hinaus können Prozesse mit formalen Parametern versehen sein, die bei der Erzeugung einer Instanz mit Werten besetzt werden. Auf diese wird in SDL wie auf Variablen zugegriffen.

Als Datentypen für Variablen und formale Parameter stehen in der vorliegenden Implementation endliche Teilmengen der ganzen Zahlen, Boolsche Werte (vordefinierter Typ **Boolean**) oder die Menge der Prozeß-IDs innerhalb des Systems (vordefinierter Typ **PId**) zur Verfügung.

Prozesse können Prozeduren aufrufen, um wiederkehrende Vorgänge abzuarbeiten.

Einmal erzeugt, arbeitet eine Prozeßinstanz, bis sie durch eine **stop**-Anweisung terminiert wird. Sie kann nicht durch äußere Einwirkung beendet werden.

process	::=	process–header process–formalparms process–decl–list procedure–decls body process–trailer
process–header	::=	**process** `process–ident` **;** opt–instnum
opt–instnum	::=	ε | **(** opt–num **)** | **(** opt–num **,** opt–num **)**
opt–num	::=	const–expr | ε
process–formalparms	::=	**fpar** processparm–decl–klist **;** | ε
processparm–decl	::=	`parm–ident`–klist `type–ident`
process–decl	::=	variable | constdef | typedef
procedure–decls	::=	procedure procedure–decls | ε
process–trailer	::=	**endprocess** opt–id **;**

Wird bei opt–instnum nur eine Zahl angegeben, so handelt es sich um die Anzahl der anfänglichen Instanzen. Wird keine Anzahl anfänglicher Instanzen angegeben, so ist sie 1. Wird keine Anzahl maximaler Instanzen angegeben, so wird sie wie in Kapitel 6 erläutert ermittelt. Die Anzahlen anfänglicher und maximal möglicher Instanzen eines Prozesses Π werden im folgenden mit $INIT(\Pi)$ bzw. $MAX(\Pi)$ bezeichnet.

Die Semantik eines Prozesses erklärt sich wie folgt (siehe unten): Zunächst wird das Netz für den Prozeß-Rumpf (ζ(body)) erzeugt. Bei Erzeugung einer Prozeßinstanz müssen auch die Variableninstanzen initialisiert bzw. beim Verlassen terminiert werden. Die Initialisierungs- und Terminierungsnetze für Variablen (κ_{init} bzw. κ_{term}(process–decl–list) werden daher mit dem Rumpf sequentialisiert und die Variablennetze parallel dazu gelegt. Das Ergebnis (N_{body}(process)) wird unter Ersetzung der Transition R_Π in das Prozeßnetz $M_{process}$ (siehe Abbildung 4.8) eingesetzt, wobei die Ersetzung von Transitionen wie in [Gra98b] definiert ist. Danach wird die parallele Komposition mit den Prozedurnetzen gebildet und über eine Menge von Aktionssymbolen α_{proc}(process) synchronisiert. Diese Menge enthält u.a. Aktionssymbole für die impliziten Variablen des Prozesses sowie die speziellen Aktionssymbole q (Zugriff auf den Zustand des Prozesses), $dash$ (Zugriff auf den vorigen Zustand) und $in?$ (Lesezugriff auf die Eingabewarteschlange).

$$\begin{aligned}
\zeta(\text{process}) = \zeta(&\textbf{process}\ \texttt{process–ident}\ ;\ \text{process–formalparms} \\
&\quad \text{process–decl–list procedure–decls body process–trailer}) \\
= \Big[&\alpha_{proc}(\text{process}) : \\
&\quad M_{process}(\texttt{process–ident}, \text{process–formalparms}, f_R(\texttt{process–ident})) \\
&\qquad [\{\mathrm{R}_\Pi\} \leftarrow N_{body}(\text{process})] \\
&\quad \| \ \zeta(\text{procedure–decls})\Big]
\end{aligned}$$

$$\begin{aligned}
\alpha_{proc}(\text{process}) &= \alpha_{proc}(\textbf{process}\ \texttt{process–ident}\ \ldots) \\
&= \alpha_{global}(\text{process–decl–list}) \cup \alpha(\text{procedure–decls}) \cup \alpha(\text{process–formalparms}) \\
&\quad \cup \{\textbf{self}, \textbf{parent}, \textbf{offspring}, \textbf{offspring}_{get}, \textbf{offspring}_{put}, \textbf{sender}, \\
&\qquad q, dash, in?\}
\end{aligned}$$

$$
\begin{aligned}
N_{body}(\text{process}) &= N_{body}(\textbf{process}\ \texttt{process–ident}\ \ldots) \\
&= \Big[(\alpha(\text{process–decl–list}) \cup \alpha_{label}(\texttt{process–ident})) : \\
&\qquad \zeta(\text{process–decl–list}) \\
&\qquad\quad || \ (\kappa_{init}(\text{process–decl–list}); \zeta(\text{body}); \kappa_{term}(\text{process–decl–list}))\Big]
\end{aligned}
$$

Die Menge $\alpha_{label}(\Pi)$ sei die Menge aller Sprungmarken im Rumpf des Prozesses Π. Die Konstruktion dieser Menge ist recht einfach (siehe auch Abschnitt 4.2.11), wird im folgenden aber nicht kompositionell durchgeführt, um die Grammatik nicht übermäßig zu verkomplizieren.

Dem globalen Kontrollfluß werden die Initialisierungs- und Terminierungstransitionen für den Prozeß sowie für alle in ihm deklarierten Prozeduren hinzugefügt.

$$
\begin{aligned}
&\kappa_P(\textbf{process}\ \texttt{process–ident}\ \ldots) \\
&\quad = M_P(\texttt{process–ident}); \kappa_P(\text{procedure–decls}), \qquad P \in \{init, term\}
\end{aligned}
$$

Außerdem werden im globalen Kontrollfluß $INIT(\Pi)$ Instanzen des Prozesses erzeugt, indem eine Sequenz von mehreren Kopien des Netzes $M_{initial}$ (siehe Abbildung 4.7) gebildet wird.

$$
\begin{aligned}
&\kappa_{create}(\textbf{process}\ \texttt{process–ident}\ \ldots) \\
&\quad = \left.\begin{array}{c} M_{initial}(\texttt{process–ident}, \text{process–formalparms}); \\ \cdots \\ M_{initial}(\texttt{process–ident}, \text{process–formalparms}) \end{array}\right\} \begin{array}{c} INIT(\texttt{process–ident}) \\ \text{Male} \end{array}
\end{aligned}
$$

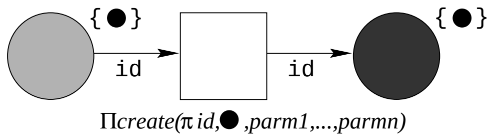

Abbildung 4.7: $M_{initial}(\Pi, \textbf{fpar}\ parm1\ set1, \ldots, parmn\ setn)$

$$
\begin{aligned}
&\alpha(\textbf{process}\ \texttt{process–ident}\ \ldots) = \{\texttt{process–ident!}\} \\
&\alpha_{global}(\textbf{process}\ \texttt{process–ident}\ \ldots) = \alpha_{global}(\text{procedure–decls}) \\
&\qquad \cup \{\texttt{process–ident}_{init}, \texttt{process–ident}_{term}, \texttt{process–ident}_{create}\} \\
&\alpha_{term}(\textbf{process}\ \texttt{process–ident}\ \ldots) = \{\texttt{process–ident}_{term}\} \\
&\alpha(\text{process–formalparms}) = \theta_{\textbf{in}}(\text{process–formalparms})
\end{aligned}
$$

$$
\begin{aligned}
&\text{fkt}(\text{procedure procedure–decls}) = \text{fkt}(\text{procedure})\ \text{op}\ \text{fkt}(\text{procedure–decls}), \\
&\qquad (\text{fkt}, \text{op}) \in \{(\zeta, ||), (\alpha, \cup), (\alpha_{global}, \cup), (\kappa_{init}, ;), (\kappa_{term}, ;)\} \\
&\text{fkt}(\text{process–decl process–decl–list}) = \text{fkt}(\text{process–decl})\ \text{op}\ \text{fkt}(\text{process–decl–list}), \\
&\qquad (\text{fkt}, \text{op}) \in \{(\zeta, ||), (\alpha, \cup), (\alpha_{global}, \cup), (\kappa_{init}, ;), (\kappa_{term}, ;)\}
\end{aligned}
$$

Das Prozedurnetz $M_{process}$ ist parametrisiert durch den Namen des Prozesses, die Liste der formalen Parameter sowie durch einige weiterer Größen, die in Form von Funktionen notiert werden. Die Abbildung 4.8 zeigt das Beispiel für einen Prozeß mit einem Parameter. Für jeden weiteren Parameter muß

- der Aktionsterm $\overline{\Pi_{create}}$ in den Transitionen T3 und T3'' um eine Kommunikationsvariable
- und das Netz um zwei Stellen und zwei Transitionen entsprechend P51, P51' und T51, T51' erweitert werden.

Der Ausdruck θ_x(list) liefere die Menge derjenigen Parameter in list, deren Übergabetyp x ist. (Alle Prozeß-Parameter sind vom Typ **in**.)

Die in $M_{process}$ verwendeten Funktionen haben folgende Bedeutung:

- $MAX(\Pi)$ steht wie bereits erwähnt für die maximale Anzahl von Instanzen von Π.
- $ID(\Pi)$ ist die Menge von Prozeß-IDs, die für Instanzen von Π verwendet wird. Die Prozeß-ID-Mengen aller Prozesse in einem System sind paarweise disjunkt, und es werden ausschließlich natürliche Zahlen (ohne Null) benutzt. Es gilt $ID(\Pi) = \{x+1, \ldots, x + MAX(\Pi)\}$ für ein $x \in \mathbb{N}$.
- $ID_p(\Pi)$ steht für die Menge aller IDs von Prozeß-Instanzen, die eine Instanz von Π erzeugen können. Entsprechend steht $ID_o(\Pi)$ für die IDs von Prozessen, die durch Π erzeugt werden können. Die Indizes p und o beziehen sich auf die Namen der impliziten Variablen **parent** bzw. **offspring**, in deren Wertemenge ID_p und ID_o Eingang finden. Diese Mengen können offensichtlich durch Durchforsten des Quelltextes nach **create**-Anweisungen gewonnen werden.
- $CAP(\Pi)$ ist die Kapazität der Eingabewarteschlange von Π. Die Möglichkeiten, diese Größe zu beeinflussen, sind in Kapitel 6 dargestellt.
- $ID_s(\Pi)$ ist die Menge der IDs derjenigen Prozesse, durch die Signale an P geschickt werden ($s \mathrel{\widehat{=}}$ **sender**).
- $SIG(\Pi)$ ist die Menge der Signale, die an Instanzen von Π verschickt werden können (genauer gesagt, die Menge der Werte, die diese Signale repräsentieren, siehe Abschnitt 4.1.2).
- $Q(\Pi)$ dient der Darstellung von Zuständen. Jedem im Rumpf von Π vorkommenden Zustand ist eine natürliche Zahl zugeordnet, die diesen Zustand im Netz repräsentiert (siehe Abschnitt 4.1.4). Genauer gesagt ist $Q(\Pi) = \{1, \ldots, n_\Pi\}$, wobei n_Π die Anzahl von Zuständen von Π sei.

Die Mengen $ID_s(\Pi)$ und $SIG(\Pi)$ lassen sich nur unter Betrachtung des gesamten Quelltextes ermitteln; der dazu verwendete Algorithmus ist in Kapitel 6 beschrieben.

In Kapitel 5 wird auf das Netz $M_{process}$ näher eingegangen; ferner werden alternative Prozeßnetze beschrieben und Vergleiche zwischen den Varianten angestellt.

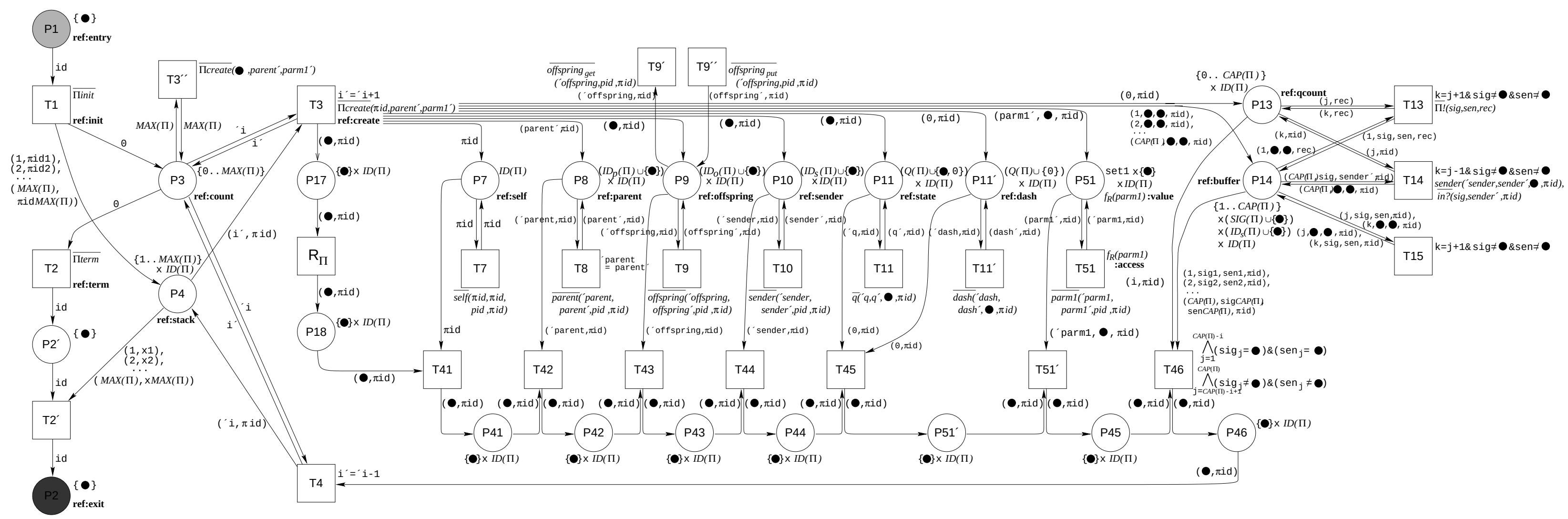

Abbildung 4.8: $M_{process}(\Pi, \textbf{fpar}\ parm1\ set1;, \textbf{ref})$

4.2.8 Prozeduren

Mit Prozeduren können wiederkehrende Befehlsfolgen zusammengefaßt werden. Prozeduren sind lokal zu einem Prozeßtyp; jeder Aufruf einer Prozedur durch eine Prozeßinstanz erzeugt eine Prozedurinstanz, die zu dieser Prozeßinstanz gehört und auf ihre Variableninstanzen (auch die der impliziten Variablen) zugreifen kann. Daneben können Prozeduren auch lokale Variablen halten. Prozeduren können verschachtelt werden und sich selbst rekursiv aufrufen.

procedure	::=	procedure–header procedure–formalparms procedure–decl–list procedure–decls body procedure–trailer
procedure–header	::=	**procedure** `procedure–ident` **;**
procedure–formalparms	::=	**fpar** procedureparm–decl–klist **;** $\mid \varepsilon$
procedureparm–decl	::=	parm–type `parm–ident`–klist `type–ident`
parm–type	::=	**in** $\mid$ **in/out** $\mid \varepsilon$
procedure–decl–list	::=	process–decl–list
procedure–trailer	::=	**endprocedure** opt–id **;**

Die Semantik einer Prozedur setzt sich ähnlich der eines Prozesses zusammen: Das Netz für den Rumpf wird in gleicher Weise mit den Variablennetzen zusammengesetzt, das Ergebnis jedoch stattdessen in das Netz $M_{procedure}$ eingesetzt. Hernach werden wie gehabt die lokalen Prozeduren parallel danebengelegt, und das Ergebnis wird synchronisiert.

$$\begin{aligned}
\zeta(\text{procedure}) &= \zeta(\textbf{procedure}\ \texttt{procedure–ident}\ ;\ \text{procedure–formalparms}\\
&\qquad\qquad \text{procedure–decl–list procedure–decls body procedure–trailer})\\
&= \Big[\alpha_{proc}(\text{procedure}) :\\
&\qquad M_{procedure}(\texttt{procedure–ident}, \text{procedure–formalparms},\\
&\qquad\qquad f_R(\texttt{procedure–ident}))[\{\mathrm{R}_P\} \leftarrow N_{body}(\text{procedure})]\\
&\qquad \parallel \zeta(\text{procedure–decls})\Big]
\end{aligned}$$

$$\begin{aligned}
\alpha_{proc}(\text{procedure}) &= \alpha_{proc}(\textbf{procedure}\ \texttt{procedure–ident}\ \ldots)\\
&= \alpha_{global}(\text{procedure–decl–list}) \cup \alpha(\text{procedure–decls})\\
&\quad \cup \alpha(\text{procedure–formalparms}) \cup \{q, dash\}
\end{aligned}$$

$$\begin{aligned}
N_{body}(\text{procedure}) &= N_{body}(\textbf{procedure}\ \texttt{procedure–ident}\ \ldots)\\
&= \Big[(\alpha(\text{procedure–decl–list}) \cup \alpha_{label}(\texttt{procedure–ident})) :\\
&\qquad \zeta(\text{procedure–decl–list})\\
&\qquad\quad \parallel \Big(\kappa_{init}(\text{procedure–decl–list});\\
&\qquad\qquad \Gamma^{\texttt{procedure}\text{–ident}}_{\theta_{\mathbf{in/out}}(\text{procedure–formalparms})}(\zeta(\text{body}));\\
&\qquad\qquad \kappa_{term}(\text{procedure–decl–list})\Big)\Big]
\end{aligned}$$

Die Aktionssymbole q und $dash$ innerhalb von α_{proc} ermöglichen den Zugriff auf den prozedur-internen Zustand. Die Reihenfolge der Synchronisierungen stellt sicher, daß bei der Benutzung dieser Aktionssymbole keine Konflikte mit den gleichnamigen Symbolen auf Prozeßebene entstehen.

θ_x ist wie bei Prozessen definiert. Die in [Gra98b] vorgestellte Relabeling-Funktion $\Gamma_A^P(N)$ indiziert im Netz N alle Vorkommen von Aktionssymbolen $a \in A$ mit dem Prozedurnamen P, d.h., a wird durch a_P ersetzt. Im vorliegenden Fall wird diese Umbenennung auf die **in/out**-Parameter der Prozedur angewendet.

Prozeduren können zwei Arten von Parametern haben, und zwar **in**-Parameter ("call by value") und **in/out**-Parameter ("call by reference"). Wird für einen Parameter das Feld parm–type leer gelassen, so gilt er als **in**-Parameter. Die beiden Arten von Parametern werden unterschiedlich behandelt (siehe auch Abschnitt 4.2.15):

- Für jeden **in**-Parameter werden je zwei Stellen und Transitionen entsprechend P51, P51' und T51, T51' in Abbildung 4.9 angelegt, und es wird der Aktionsterm P_{call} um eine Kommunikationsvariable erweitert. Was Synchronisation angeht, werden **in**-Parameter wie alle anderen Variablen behandelt.
- **in/out**-Parameter hingegen werden wie zuvor erwähnt umbenannt, indem sie mit dem Namen der Prozedur indiziert werden. Dies geschieht, da ihre Aktionssymbole auch beim Prozeduraufruf verwendet werden und daher über das Prozedurnetz hinaus eindeutig sein müssen (vgl. Abbildung 4.26). Synchronisiert wird über diese (umbenannten) Parameter auf höherer (i.a. auf Prozeß-) Ebene, um alle Aufrufe der Prozedur zu erfassen. Um dies formal zu darzulegen, wird die ursprünglich nur für Netze definierte Umbenennung durch Γ in den folgenden Gleichungen auch direkt auf Aktionssymbole angewendet.

$$
\begin{aligned}
&\alpha(\text{procedure–formalparms}) = \theta_{\mathbf{in}}(\text{procedure–formalparms})\\
&\alpha(\text{procedure}) = \alpha(\textbf{procedure}\ \texttt{procedure–ident}\ \ldots)\\
&\qquad = \ \{\texttt{procedure–ident}_{call}, \texttt{procedure–ident}_{return}\}\\
&\qquad\quad \cup\ \Gamma^{\texttt{procedure–ident}}_{\theta_{\mathbf{in/out}}(\text{procedure–formalparms})}(\theta_{\mathbf{in/out}}(\text{procedure–formalparms}))\\
&\alpha_{global}(\text{procedure}) = \alpha_{global}(\textbf{procedure}\ \texttt{procedure–ident}\ \ldots)\\
&\qquad = \ \{\texttt{procedure–ident}_{init}, \texttt{procedure–ident}_{term}\}\\
&\qquad\quad \cup\ \alpha_{global}(\text{procedure–decls})\\
&\kappa_P(\text{procedure}) = \kappa_P(\textbf{procedure}\ \texttt{procedure–ident}\ \ldots)\\
&\qquad = \ M_P(\texttt{procedure–ident}); \kappa_P(\text{procedure–decls}), \qquad P \in \{init, term\}
\end{aligned}
$$

Innerhalb einer Prozedur können (wie in einem Prozeß) lokale Variablen, Prozeduren, Konstanten und Typdefinitionen vereinbart werden, deren Gültigkeit auf den Rahmen der Prozedur beschränkt ist. Für procedure–decl–list gelten daher daher dieselben Regeln wie für process–decl–list.

Das Netz $M_{procedure}$ ist ähnlich wie das Prozeß-Netz parametrisiert durch den Namen der Prozedur und durch die Liste der formalen Parameter. Abbildung 4.9 zeigt das Beispiel einer Prozedur mit einem **in**-Parameter.

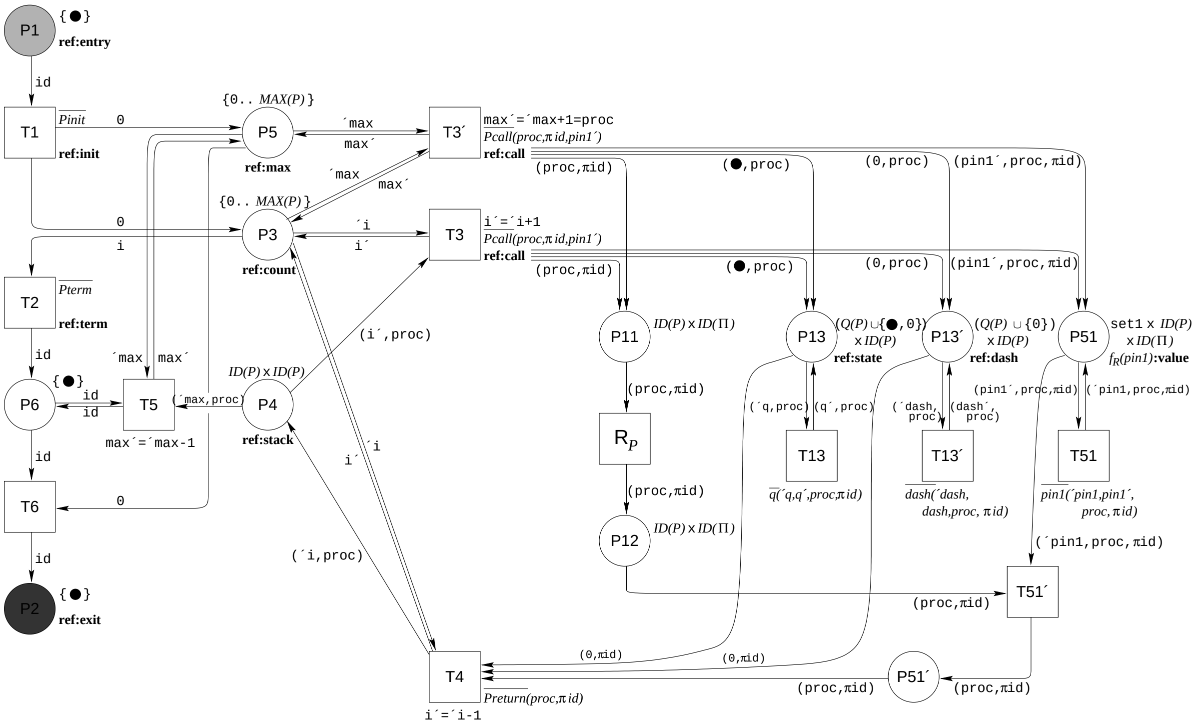

Abbildung 4.9: $M_{procedure}(P, \textbf{fpar in}\ pin1\ set1, \textbf{ref})$

Die Bedeutung der in Abbildung 4.9 verwendeten Hilfsfunktionen ist wie folgt:

- $MAX(P)$ ist die maximale Anzahl von Instanzen der Prozedur P. Möglichkeiten, wie diese Zahl beeinflußt werden kann, werden in Kapitel 6 beschrieben.
- $ID(P)$ ist die Menge der Prozedur-IDs, die für Instanzen von P zur Verfügung stehen. Wie in Abschnitt 4.1.1 erläutert, ist $ID(P) = \{1, \ldots, MAX(P)\}$.
- $ID(\Pi)$ ist die Menge der Prozeß-IDs desjenigen Prozesses Π, innerhalb dessen P definiert ist. Zur Verdeutlichung des Verhältnisses zwischen $ID(P)$ und $ID(\Pi)$ sei gesagt, daß jederzeit höchstens $MAX(P)$ Instanzen der Prozedur existieren können, und zwar insgesamt, nicht etwa je Prozeßinstanz! Jeder Prozeduraufruf einer Prozeßinstanz erzeugt eine Prozedurinstanz mit einem beliebigen freien Identifikator aus der Menge $ID(P)$.
- $Q(P)$ ist analog zu Prozessen eine Menge, die die Zustände von P repräsentiert.

4.2.9 Variablen

Variablen dienen der Speicherung von Werten. Variablen werden lokal zu einem Prozeß oder einer Prozedur vereinbart und ordnen jeder Instanz dieses Prozesses bzw. dieser Prozedur eine eigene Variableninstanz zu. Variablen können nicht global (d.h. systemweit) vereinbart werden.

Die Menge von Werten, die eine Variable speichern kann, wird durch ihren Typ bestimmt; dieser kann (in der vorliegenden Implementation) einer der vordefinierten

Typen **PId** oder **Boolean** oder ein mit **syntype** benutzerdefinierter Typ sein, siehe Abschnitt 4.2.2. Als benutzerdefinierte Typen kommen nur endliche Teilmengen der ganzen Zahlen in Frage.

variable	::=	**dcl** decl–klist **;**
decl	::=	`variable–ident`–klist `type–ident` |
		`variable–ident`–klist `type–ident` **:=** const–expr

Bei der zweiten Form der Deklaration wird ein Wert festgelegt, mit dem jede Instanz der Variable bei ihrer Erzeugung belegt wird. Bei der ersten Form wird der bei der Typdeklaration vereinbarte **default**-Wert zur Initialisierung verwendet.

Deklarationen mehrerer Variablen mit einer **dcl**-Anweisung können stets äquivalent in mehrere Deklarationen einzelner Variablen umgewandelt werden. Es genügt also die Angabe der Semantik einer solchen einzelnen Deklaration.

$$\zeta(\textbf{dcl}\ \texttt{variable–ident type–ident;})$$
$$= M_{variable}(\texttt{variable–ident}, \texttt{type–ident}, f_R(\texttt{variable–ident}))$$
$$\alpha(\textbf{dcl}\ \texttt{variable–ident type–ident;}) = \{\texttt{variable–ident}_{init}, \texttt{variable–ident}_{term}\}$$
$$\alpha_{global}(\textbf{dcl}\ \texttt{variable–ident type–ident;}) = \{\texttt{variable–ident}\}$$
$$\kappa_{init}(\textbf{dcl}\ \texttt{variable–ident type–ident;}) = M_{varinit}(\texttt{variable–ident})$$
$$\kappa_{term}(\textbf{dcl}\ \texttt{variable–ident type–ident;}) = M_{varterm}(\texttt{variable–ident})$$

Man beachte, daß aufgrund der in den obigen Abschnitten beschriebenen Semantik von Prozessen und Prozeduren die Netze κ_{init} und κ_{term} nicht im globalen Kontrollfluß, sondern in dem der zugehörigen Prozeß-/Prozedurinstanz stehen.

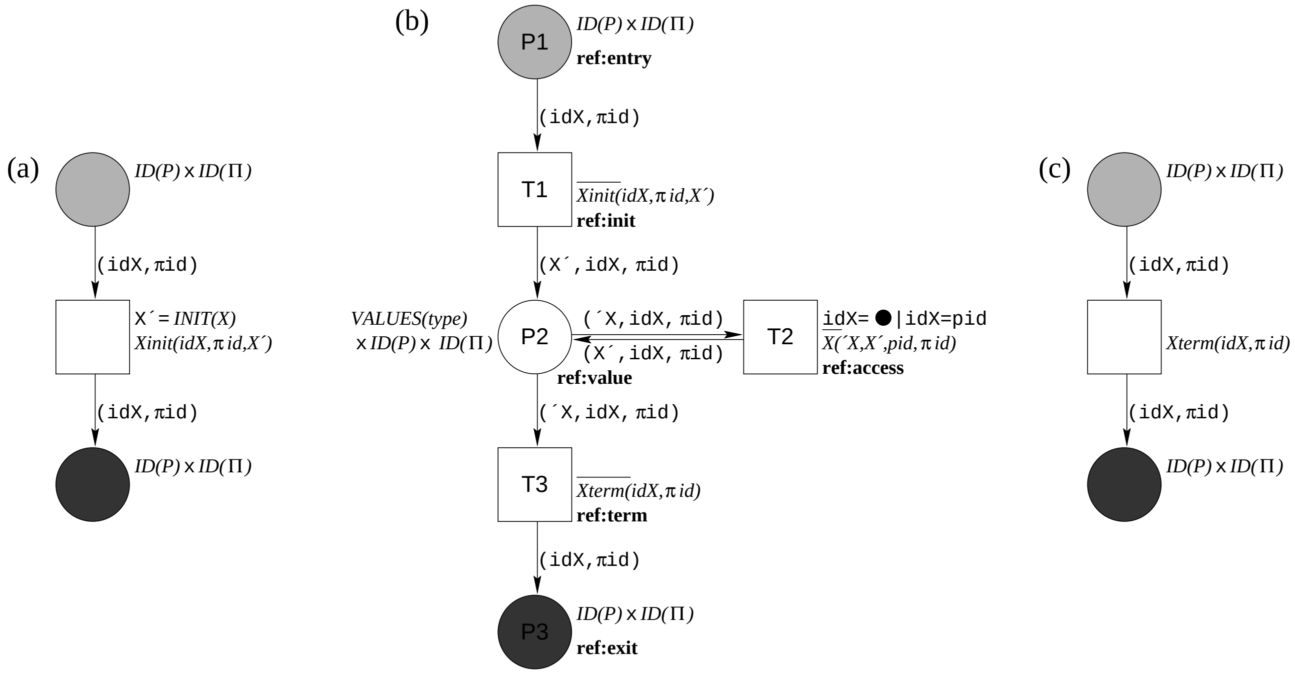

Abbildung 4.10: **(a)** $M_{varinit}(X)$ **(b)** $M_{variable}(X, type, \textbf{ref})$ **(c)** $M_{varterm}(X)$

Abbildung 4.10 zeigt die Netze $M_{varinit}$, $M_{variable}$ und $M_{varterm}$. Dabei steht

- $ID(P)$ für die Menge der Prozedur-IDs der Prozedur, innerhalb derer die Variable deklariert ist, und für $\{\bullet\}$, falls die Variable unmittelbar zu einem Prozeß gehört;
- $ID(\Pi)$ für die Menge der IDs des übergeordneten Prozesses, innerhalb dessen die Deklaration der Variable steht;
- $VALUES(type)$ für die Menge der Werte, die dem angegebenen Typ zugeordnet ist;
- $INIT(X)$ für den Wert, mit dem die Variable X initialisiert wird (s.o.).

4.2.10 Rumpf eines Prozesses bzw. einer Prozedur

Der Rumpf eines Prozesses oder einer Prozedur beschreibt eine Menge von Zustandsübergängen (auch (SDL-)Transitionen genannt). Ein besonderer Zustandsübergang, die sogenannte Start-Transition, muß vorhanden sein und wird zu Beginn der Ausführung des Prozesses bzw. der Prozedur abgearbeitet. Jeder Zustandsübergang kann entweder in einen beliebigen Zustand des Prozesses/der Prozedur führen oder den Prozeß terminieren (bzw. die Prozedur beenden).

body	::=	start states
states	::=	state states $\mid \varepsilon$

Die Semantik des Rumpfs wird durch ein Iterationskonstrukt ausgedrückt. Der Start-Teil der Iteration ist dabei die Übersetzung der Start-Transition, der iterierte Teil eine Auswahl aller anderen Zustandsübergänge, der End-Teil ein spezielles Netz, das genau dann ausgeführt wird, wenn sich der Rumpf beendet hat.

$$\zeta(\text{start states}) = [\,\zeta(\text{start}) * \zeta(\text{states}) * M_{exit}\,]$$
$$\zeta(\text{state states}) = \zeta(\text{state})\ []\ \zeta(\text{states})$$

Abbildung 4.11 zeigt das Netz M_{exit}. Wie alle folgenden Netze verwendet es die Mengen $ID(P)$ und $ID(\Pi)$, die die IDs der umschließenden Prozedur und des umschließenden Prozesses repräsentieren. Aus Gründen der Übersichtlichkeit werden diese nicht explizit in der Semantikfunktion weitergereicht (siehe auch Abschnitt 5.1). Gehört der Rumpf nicht zu einer Prozedur, so ist $ID(P) = \{\bullet\}$ anzunehmen.

In den Kantenbeschriftungen repräsentiert die Variable pid die Prozedur-ID, während πid für die Prozeß-ID steht (in der ASCII-Repräsentation der M-Netze wird πid durch `piid` ausgedrückt).

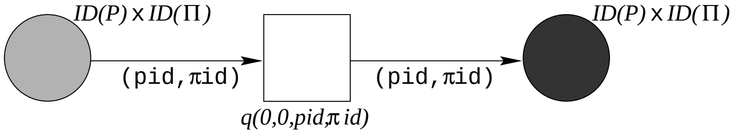

Abbildung 4.11: Das Netz M_{exit}

4.2.11 Zustandsübergänge

Zustandsübergänge bestehen aus zwei Teilen – einer auslösenden Bedingung und einer Folge von Anweisungen. Die auslösende Bedingung ist in der Regel eine Kombination von Zuständen und eintreffenden Signalen. Einzig die Start-Transition hat keine auslösende Bedingung.

start	::=	**start ;** transition

$$\zeta(\textbf{start;}\ \text{transition}) = \zeta(\text{transition})$$

Auslösende Bedingungen

Die auslösende Bedingung eines Zustandsübergangs legt fest, unter welchen Umständen die Aktionen in ihm ausgeführt werden. Sie legt fest:

- einen Zustand bzw. eine Menge von Zuständen, in denen sich der Prozeß/die Prozedur befinden muß;
- einen Stimulus, der den Zustandsübergang anstoßen kann, falls sich der Prozeß/die Prozedur in einem der zuvor genannten Zustände befindet.

In der vorliegenden Implementation werden zwei Arten von Stimuli unterstützt, und zwar das Eintreffen von Signalen sowie Spontaneität, d.h. solche Zustandsübergänge, die ohne äußeren Einfluß stattfinden.

state	::=	state–header state–body–list
state–header	::=	**state** `state–ident`–klist **;**
state–body	::=	stimulus transition
stimulus	::=	inputpart | spontrans

Der Angabe einer Zustandsmenge kann eine Reihe von Stimuli folgen, wobei jeder Stimulus einen Zustandsübergang definiert, dessen Aktionen in der nachfolgenden (SDL-)Transition aufgeführt sind. Die Menge der Zustände bezieht sich dann auf jeden einzelnen Zustandsübergang. Anders ausgedrückt:

$$\begin{aligned}&\zeta(\text{state–header state–body state–body–list})\\ &\qquad = \zeta(\text{state–header state–body})\ []\ \zeta(\text{state–header state–body–list})\end{aligned}$$

Die Angabe mehrerer Zustände ist als "oder" zu verstehen, d.h., die Zustandsübergänge können eintreten, falls einer der Zustände eintritt.

$$\zeta(\text{state–header stimulus transition}) = \zeta(\text{state–header stimulus}); \zeta(\text{transition})$$

$$\begin{aligned}&\zeta(\textbf{state}\ \texttt{state–ident}\ \texttt{state–ident}\text{–klist}\textbf{;}\ \text{stimulus})\\ &\qquad = \zeta(\textbf{state}\ \texttt{state–ident}\textbf{;}\ \text{stimulus})\ []\ \zeta(\textbf{state}\ \texttt{state–ident}\text{–klist}\textbf{;}\ \text{stimulus})\end{aligned}$$

Auslösen durch Erhalt eines Signals

Trifft ein Signal bei einem Prozeß ein, so wird es zunächst in dessen Eingabewarteschlange gespeichert. Da diese wie ein FIFO-Puffer arbeitet, "sieht" der Prozeß stets nur das vorderste Element dieser Schlange. Durch Verwendung eines inputparts (siehe unten) kann dieses Element betrachtet werden. Entspricht es einem der in signal–klist aufgeführten Signale, so wird es aus der Warteschlange entfernt und der zugehörige Zustandsübergang ausgelöst.

inputpart	::=	**input** signal–klist **;**
signal	::=	`signal–ident` | `signal–ident` **(** sigarg–klist **)**
sigarg	::=	const–expr | `variable–ident` | ε

Verwendete Signale müssen zuvor mit einer **signal**-Anweisung definiert worden sein (siehe Abschnitt 4.2.2). Anzahl und Typ der Argumente müssen mit der Signaldefinition übereinstimmen. Ein Zustandsübergang kann ausgelöst werden, wenn in der Eingabewarteschlange eines der in signal–klist aufgeführten Signale steht und dessen Argumente mit den in sigarg–klist genannten Parametern verträglich sind.

Eine Konstante als Argument in sigarg–klist fordert, daß das Signal in der Warteschlange ebenfalls diesen Wert als Argument besitzen soll. Wird eine Variable angegeben, so wird der Argument-Wert des Signals in der Warteschlange in dieser Variablen gespeichert. Wird eine Angabe für ein einzelnes Argument weggelassen, so wird der Wert des Signals an dieser Stelle ignoriert.

Die Angabe mehrerer Signale ist (ebenso wie bei Zuständen) als "oder" zu verstehen, d.h., der Zustandsübergang tritt ein, wenn eine der Kombinationen von Zustand und Signal zutrifft.

$$\begin{aligned}&\zeta(\text{state–header } \textbf{input} \text{ signal}\textbf{,} \text{ signal–klist}\textbf{;})\\ &\qquad = \zeta(\text{state–header } \textbf{input} \text{ signal}\textbf{;})\ []\ \zeta(\text{state–header } \textbf{input} \text{ signal–klist}\textbf{;})\end{aligned}$$

Jede Kombination von zulässigen Zuständen und Signalen erzeugt also einen Zweig der Auswahl. Wie gleich zu sehen ist, besteht jeder dieser Zweige nur aus einer Transition mit entsprechender Schaltbedingung.

Für Signale ohne Argumente sieht die Übersetzung verhältnismäßig einfach aus:

$$\begin{aligned}&\zeta(\textbf{state}\ \texttt{state–ident}\textbf{;}\ \textbf{input}\ \texttt{signal–ident}\textbf{;})\\ &\qquad = M_{inpnoarg}(\texttt{state–ident}, \texttt{signal–ident}, f_R(\textbf{input}\ \texttt{signal–ident}))\end{aligned}$$

Das Netz $M_{inpnoarg}$ ist in Abbildung 4.12 zu sehen. In ihm steht $f_Q(\texttt{state–ident})$ für die in Abschnitt 4.1.4 angesprochene Abbildung von Zuständen des SDL-Programms auf natürliche Zahlen.

Signale mit Argumenten erfordern eine aufwendigere Übersetzung:

$$\begin{aligned}&\zeta(\textbf{state}\ \texttt{state–ident}\textbf{;}\ \textbf{input}\ \texttt{signal–ident}\ \textbf{(}\text{sigarg–klist}\textbf{)}\textbf{;})\\ &\qquad = M_{inparg}(\texttt{state–ident}, \texttt{signal–ident}, \text{sigarg–klist},\\ &\qquad\qquad f_R(\textbf{input}\ \texttt{signal–ident}\ (\text{sigarg–klist})))\end{aligned}$$

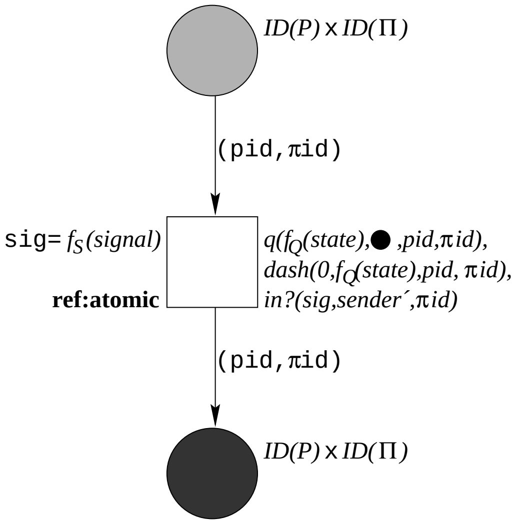

Abbildung 4.12: $M_{inpnoarg}(state, signal, \mathbf{ref})$

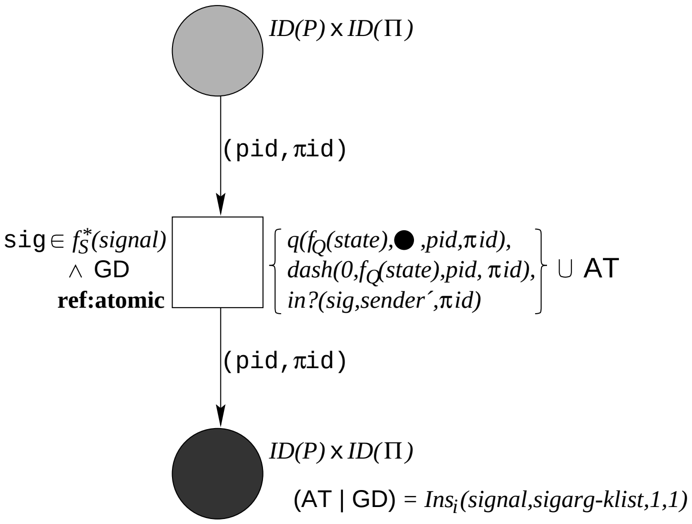

Abbildung 4.13: $M_{inparg}(state, signal, sigarg\text{-}klist, \mathbf{ref}))$

Das dazu verwendete Netz M_{inparg} ist in Abbildung 4.13 zu sehen.[4)] Zur Definition dieses Netzes wird die Hilfsfunktion Ins_i benutzt, in der die zusätzlichen Aktionsterme und Schaltbedingungen gesammelt werden, die für die Auswertung der Argumente notwendig sind. Wie bereits in Abschnitt 4.1.2 erwähnt ist jedem Signal s eine Menge $f_S^*(s)$ von natürlichen Zahlen zugeordnet. Beim Versand eines Signals s durch den Befehl **output** (siehe Abschnitt 4.2.12) werden dessen Argumente eineindeutig auf einen Wert aus $f_S^*(s)$ abgebildet und müssen bei der Interpretation durch **input** wieder zurückgewonnen werden. Das für die Kodierung verwendete Verfahren wird in Abschnitt 5.10 detailliert erläutert.

[4)] Ausnahmsweise werden hier (im Gegensatz zu anderen Netzen) Mengenklammern bei der Darstellung der Aktionsterme verwendet, damit der Vereinigungsoperator benutzt werden kann.

Jedes Argument des Signals trägt einen Teil zum Inhalt von Ins_i bei. Das dritte Argument von Ins_i (hier: m) hält fest, das wievielte Argument behandelt wird, im vierten Argument (hier: n) werden die Mächtigkeiten der Argumentmengen multipliziert (siehe Abschnitt 5.10). Der Ausdruck $ARG(\texttt{signal–ident}, m)$ liefere die Wertemenge des m-ten Arguments in der Definition von `signal–ident`.

$$
\begin{aligned}
&Ins_i(\texttt{signal–ident}, \text{sigarg sigarg–klist}, m, n) = (AT_1 \cup AT_2 \mid GD_1 \wedge GD_2),\\
&\quad \text{wobei } (AT_1 \mid GD_1) = Ins_i(\texttt{signal–ident}, \text{sigarg}, m, n)\\
&\quad \text{und } (AT_2 \mid GD_2) = Ins_i(\texttt{signal–ident}, \text{sigarg–klist},\\
&\qquad\qquad\qquad\qquad\qquad m+1,\ n * |ARG(\texttt{signal–ident}, m)|)
\end{aligned}
$$

Um die einzelnen Argumente aus Werten von $f_S^*(\texttt{signal–ident})$ herauszuholen, sind einige Umrechnungen erforderlich. Die Funktion E liefert einen Ausdruck, der den Anteil des m-ten Arguments aus dem Wert der Kommunikationsvariablen `sig` extrahiert:

$$
\begin{aligned}
E(\texttt{signal–ident}, m, n) &= \min(ARG(\texttt{signal–ident}, m))\\
&+ \ \Big(\big(\texttt{sig} - \min(f_S^*(\texttt{signal–ident}))\big)/n\Big) \bmod\ |ARG(\texttt{signal–ident}, m)|
\end{aligned}
$$

Für die Zwecke dieser Definition sei $\min(\mathbf{PId}) = -1$ und $\min(\mathbf{Boolean}) = 0$. Der Divisions-Operator “/” stellt die Integer-Division dar, d.h. Teilung ohne Nachkommastellen.

$$
\begin{aligned}
&Ins_i(\texttt{signal–ident}, \text{const–expr}, m, n)\\
&\quad = \begin{cases}
(\emptyset \mid E(\texttt{signal–ident}, m, n) = \text{const–expr}) & \text{falls const–expr} \in \mathbb{Z}\\
(\emptyset \mid E(\texttt{signal–ident}, m, n) = 0) & \text{falls const–expr} = \mathbf{False}\\
(\emptyset \mid E(\texttt{signal–ident}, m, n) = 1) & \text{falls const–expr} = \mathbf{True}\\
(\emptyset \mid E(\texttt{signal–ident}, m, n) = -1) & \text{falls const–expr} = \mathbf{Null}
\end{cases}
\end{aligned}
$$

Für Variablen x gilt

$$
\begin{aligned}
&Ins_i(\texttt{signal–ident}, x, m, n)\\
&\quad = \ (\{('x, x', pid, \pi id)\} \mid (E(\texttt{signal–ident}, m, n) = 0 \wedge x' = \mathbf{false})\\
&\qquad\qquad\qquad\qquad \vee\ (E(\texttt{signal–ident}, m, n) = 1 \wedge x' = \mathbf{true})),
\end{aligned}
$$

falls $ARG(\texttt{signal–ident}, m) = \mathbf{Boolean}$,

$$
\begin{aligned}
&Ins_i(\texttt{signal–ident}, x, m, n)\\
&\quad = \ (\{('x, x', pid, \pi id)\} \mid (E(\texttt{signal–ident}, m, n) = -1 \wedge x' = \bullet)\\
&\qquad\qquad\qquad\qquad \vee\ (E(\texttt{signal–ident}, m, n) = x' \wedge x' \geq 0)),
\end{aligned}
$$

falls $ARG(\texttt{signal–ident}, m) = \mathbf{PId}$, und

$$
\begin{aligned}
&Ins_i(\texttt{signal–ident}, x, m, n)\\
&\quad = \ (\{('x, x', pid, \pi id)\} \mid E(\texttt{signal–ident}, m, n) = x')
\end{aligned}
$$

sonst. Für weggelassene Argumente gilt

$$Ins_i(\texttt{signal–ident}, \varepsilon, m, n) = (\emptyset \mid \textbf{true})$$

Ein paar Beispiele mögen die komplizierte formale Definition ein wenig aufhellen:

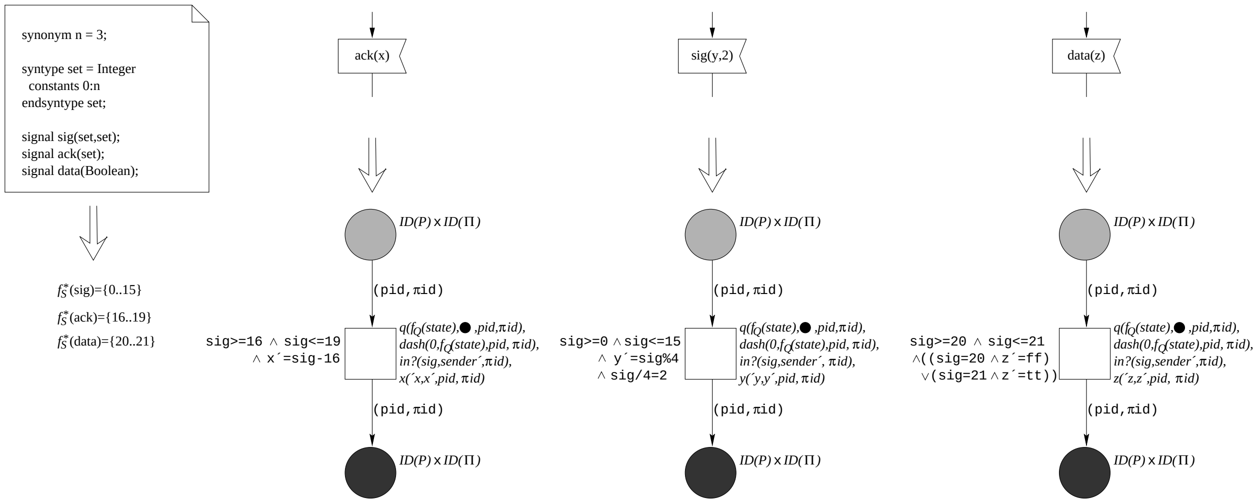

Abbildung 4.14: Beispiele für die Übersetzung von **input**

Spontaner Zustandsübergang

Spontane Zustandsübergänge können (nichtdeterministisch) stattfinden, ohne daß die betreffende Prozeßinstanz ein Signal erhalten hat. Spontane Zustandsübergänge und solche, die auf Erhalt eines Signals hin stattfinden, haben die gleiche Priorität, unabhängig vom Inhalt der Eingabewarteschlange. Wird ein spontaner Zustandsübergang ausgeführt, so wird **sender** := **self** gesetzt.

spontrans	::=	**input none ;**

Die Übersetzung eines solchen Zustandsüberganges gestaltet sich entsprechend einfach (siehe Abbildung 4.15):

$$\zeta(\textbf{state}\ \texttt{state–ident};\ \textbf{input none;}) = M_{spont}(\texttt{state–ident}, f_R(\textbf{input none}))$$

Transitionen

Transitionen bestehen in SDL aus einer Kette von Anweisungen. Diese legen das Verhalten von Prozessen/Prozeduren bei Ausführung eines Zustandsüberganges fest. Jede Anweisung kann mit einer Markierung (label) versehen sein, die als Sprungziel eines **join**-Befehls benutzt werden kann.

transition	::=	statement transition | ε
statement	::=	label action | action
label	::=	`label–ident :`
action	::=	output | task | decision | call | create | terminator
terminator	::=	nextstate | join | stop | return

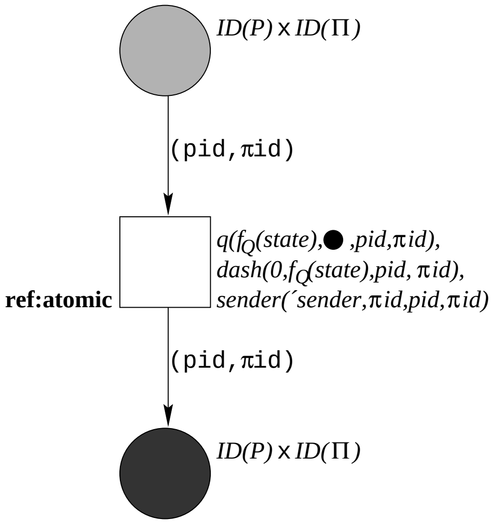

Abbildung 4.15: $M_{spont}(state, \mathbf{ref})$

Start-Transitionen oder solche, die zu einem Zustandsübergang gehören, müssen mit einem terminator enden oder mit einer terminierenden **decision**-Anweisung.

Die Anweisungen einer Transition werden nacheinander ausgeführt, die Übersetzung in M-Netze erfolgt daher, indem die Netze für die einzelnen Anweisungen gebildet und in einer Sequenz zusammengefaßt werden.

$$\zeta(\text{statement transition}) = \zeta(\text{statement}); \zeta(\text{transition})$$

Ist eine Anweisung mit einer Sprungmarke versehen, so wird das Netz aus Abbildung 4.16 vor das eigentliche Netz für die Anweisung gehängt. Ist diese Marke das Ziel eines **join**-Befehls (Abschnitt 4.2.18), so findet eine Synchronisation der Transition aus Abbildung 4.31 mit der Transition auf der linken Seite aus Abbildung 4.16 statt; auf diese Weise wechselt der Kontrollfluß von der **join**-Anweisung zur markierten Anweisung (vgl. [MR96]). Für jede Sprungmarke `label–ident` gilt dabei $\texttt{label–ident}_{join} \in \alpha_{label}(P)$, wobei P der Prozeß/die Prozedur ist, dessen Rumpf diese Sprungmarke enthält. Die Transition ohne Vorstelle aus Abbildung 4.16 verschwindet durch die an die Synchronisation anschließende Restriktion.

$$\zeta(\text{label action}) = \zeta(\text{label}); \zeta(\text{action})$$
$$\zeta(\texttt{label–ident:}) = M_{label}(\texttt{label–ident}, f_R(\texttt{label–ident}))$$

4.2.12 Übersetzung von output

Die **output**-Anweisung dient zum Versenden eines oder mehrerer Signale an eine Prozeßinstanz. Der Empfänger-Prozeß kann explizit adressiert (mit **to**) oder durch Angabe des Weges (mit **via**) indirekt bestimmt werden. Beide Angaben können auch weggelassen werden; in diesem Falle geht das Signal an irgendeinen Prozeß, mit dem eine Verbindung besteht, über die das Signal gesendet werden kann.

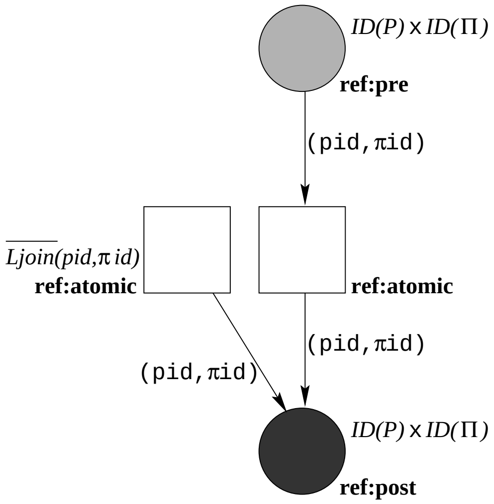

Abbildung 4.16: $M_{label}(L, \mathbf{ref})$

output	::=	**output** signal–klist output–to output–via **;**
output–to	::=	**to** `process-ident` | **to** `variable-ident` | ε
output–via	::=	**via** via–ident–klist | **via all** via–ident–klist | ε
via–ident	::=	`channel-ident` | `signalroute-ident`

Die Definition von signal–klist findet sich bei der Vorstellung der **input**-Anweisung in Abschnitt 4.2.11. Enthält signal–klist mehrere Elemente, so ist dies äquivalent zu mehreren **output**-Anweisungen mit je einem Signal.

$$\begin{aligned}&\zeta(\textbf{output}\ \text{signal}\textbf{,}\ \text{signal–klist output–to output–via}\textbf{;})\\ &\quad= \zeta(\textbf{output}\ \text{signal output–to output–via}\textbf{;})\,;\\ &\qquad\ \zeta(\textbf{output}\ \text{signal–klist output–to output–via}\textbf{;})\end{aligned}$$

Bei Benutzung von **via all** wird das Signal im "Multicast"-Modus versandt, d.h., über jede in `via-ident`–klist genannte Verbindung wird je eine Instanz des Signals geschickt.

$$\begin{aligned}&\zeta(\textbf{output}\ \text{signal–klist output–to}\ \textbf{via all}\ \text{via–ident}\textbf{,}\ \text{via–ident–klist}\textbf{;})\\ &\quad= \zeta(\textbf{output}\ \text{signal–klist output–to}\ \textbf{via}\ \text{via–ident}\textbf{;})\,;\\ &\qquad\ \zeta(\textbf{output}\ \text{signal–klist output–to}\ \textbf{via all}\ \text{via–ident–klist}\textbf{;})\end{aligned}$$

Um ein einzelnes Signal auszugeben, werden zunächst alle Pfade (bestehend aus Signalrouten und Kanälen) ermittelt, über die das Signal vom Sender zu einem Empfänger-Prozeß gelangen kann. Besteht output–to aus "**to** `process-ident`", so werden anschließend die möglichen Pfade auf diejenigen reduziert, die zu diesem Prozeß führen. Bei Angabe einer **via**-Liste findet eine weitere Beschränkung auf diejenigen Pfade statt, die die genannten Signalwege beinhalten. Für die vorliegende Semantik relevant sind dann nur noch die ersten Elemente auf diesen Pfaden, d.h. die vom Prozeß wegführenden Signalrouten. $PATH_{\Pi}(\text{signal}, \text{output–to}, \text{output–via})$ bezeichne im folgenden die Menge der Pfade (Signalrouten und Kanäle), über die das Signal signal vom Prozeß Π aus gemäß der **to-** und **via**-Klauseln weitergeleitet werden kann. $HEAD(p)$ bezeichne das erste Element auf einem solchen Pfad (d.h. eine von Π wegführende Signalroute), $TAIL(p)$ bezeichne den Empfänger-Prozeß am Ende des Pfades.

Einzelne **output**s werden mit Hilfe des Netzes aus Abbildung 4.17 übersetzt:

$$\zeta(\textbf{output}\ \text{signal output–to output–via}\textbf{;})$$
$$= \underset{p \in PATH_{\Pi}(\text{signal,output–to,output–via})}{[]} M_{output}(\text{signal}, \text{output–to}, p, f_R(\textbf{output}\ \text{signal}))$$

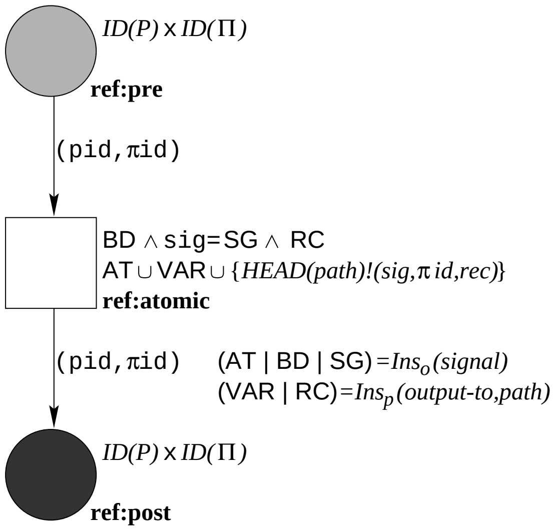

Abbildung 4.17: $M_{output}(signal, output\text{–}to, path, \mathbf{ref})$

Die Hilfsfunktionen Ins_o und Ins_p berechnen einige Hilfsgrößen. Ins_o(signal) berechnet u.a. die Abbildung der Argumentwerte auf eine natürliche Zahl (siehe Abschnitt 5.10 und vergleiche Abschnitt 4.2.11). Diese Funktion ist wie folgt definiert:

$$Ins_o(\texttt{signal–ident}) = (\emptyset \mid \mathbf{true} \mid f_S(\texttt{signal–ident}))$$

$$Ins_o(\texttt{signal–ident}(\text{sigarg–klist})) = (AT \mid BD \mid \min(f_S^*(\texttt{signal–ident})) + SG)$$
$$\text{mit } (AT \mid BD \mid SG) = Ins_o(\texttt{signal–ident}, \text{sigarg–klist}, 1, 1)$$
$$Ins_o(\texttt{signal–ident}, \text{sigarg sigarg–klist}, m, n) = (AT_1 \cup AT_2 \mid BD_1 \wedge BD_2 \mid SG_1 + SG_2)$$
$$\text{mit } (AT_1 \mid BD_1 \mid SG_1) = Ins_o(\texttt{signal–ident}, \text{sigarg}, m, n)$$
$$\text{und } (AT_2 \mid BD_2 \mid SG_2) = Ins_o(\texttt{signal–ident}, \text{sigarg–klist}, m+1, n * |ARG(\texttt{signal–ident}, m)|)$$

$$Ins_o(\texttt{signal–ident}, \text{const–expr}, m, n)$$
$$= \begin{cases} (\emptyset \mid \textbf{true} \mid \Big(\text{const–expr} - \min(ARG(\texttt{signal–ident}, m))\Big) \cdot n) & \\ \qquad\qquad\qquad\qquad\qquad \text{falls const–expr} \in \mathbb{Z} & \\ (\emptyset \mid \textbf{true} \mid 0) & \text{falls const–expr} = \textbf{False} \\ (\emptyset \mid \textbf{true} \mid n) & \text{falls const–expr} = \textbf{True} \\ (\emptyset \mid \textbf{true} \mid 0) & \text{falls const–expr} = \textbf{Null} \end{cases}$$

Für Variablen x gilt

$$Ins_o(\texttt{signal–ident}, x, m, n)$$
$$= \quad (\{(x, x, pid, \pi id)\} \mid c_m \in \{0, 1\} \wedge (c_m = 1 \iff x = \textbf{true}) \mid c_m \cdot n),$$

falls $ARG(\texttt{signal–ident}, m) = \textbf{Boolean}$,

$$Ins_o(\texttt{signal–ident}, x, m, n)$$
$$= \quad (\{(x, x, pid, \pi id)\} \mid c_m \in \{0, \ldots, |\textbf{PId}| - 1\} \wedge (x = \bullet \Rightarrow c_m = 0)$$
$$\wedge\ (x \neq \bullet \Rightarrow c_m = x + 1) \mid c_m \cdot n),$$

falls $ARG(\texttt{signal–ident}, m) = \textbf{PId}$, und

$$Ins_o(\texttt{signal–ident}, x, m, n)$$
$$= \quad (\{(x, x, pid, \pi id)\} \mid \textbf{true} \mid (x - \min ARG(\texttt{signal,m–ident})) \cdot n)$$

sonst. Für weggelassene Argumente gilt folgende Vorschrift:

$$Ins_o(\texttt{signal–ident}, \varepsilon, m, n) = (\emptyset \mid \textbf{true} \mid 0)$$

Die Funktion $Ins_p(\text{output–to}, p)$ definiert sich wie folgt:

$$Ins_p(\texttt{to process–ident}, p) = (\emptyset \mid rec \in ID(\texttt{process–ident}))$$
$$Ins_p(\texttt{to variable–ident}, p) = (\{\texttt{variable–ident}(rec, rec, pid, \pi id)\} \mid \textbf{true})$$
$$Ins_p(\varepsilon, p) = (\emptyset \mid rec \in ID(TAIL(p)))$$

Die zweite Regel für Ins_p sorgt dafür, daß die Kommunikationsvariable rec, die die ID des Empfängerprozesses hält, auf den Wert von `variable–ident` gesetzt wird, ohne daß sich der Wert von `variable–ident` selbst verändert. Bei der Variablen kann es sich auch um eine der impliziten Variablen des Prozesses handeln!

Auch hier mögen einige Beispiele die Semantik verdeutlichen helfen, siehe Abbildung 4.18.

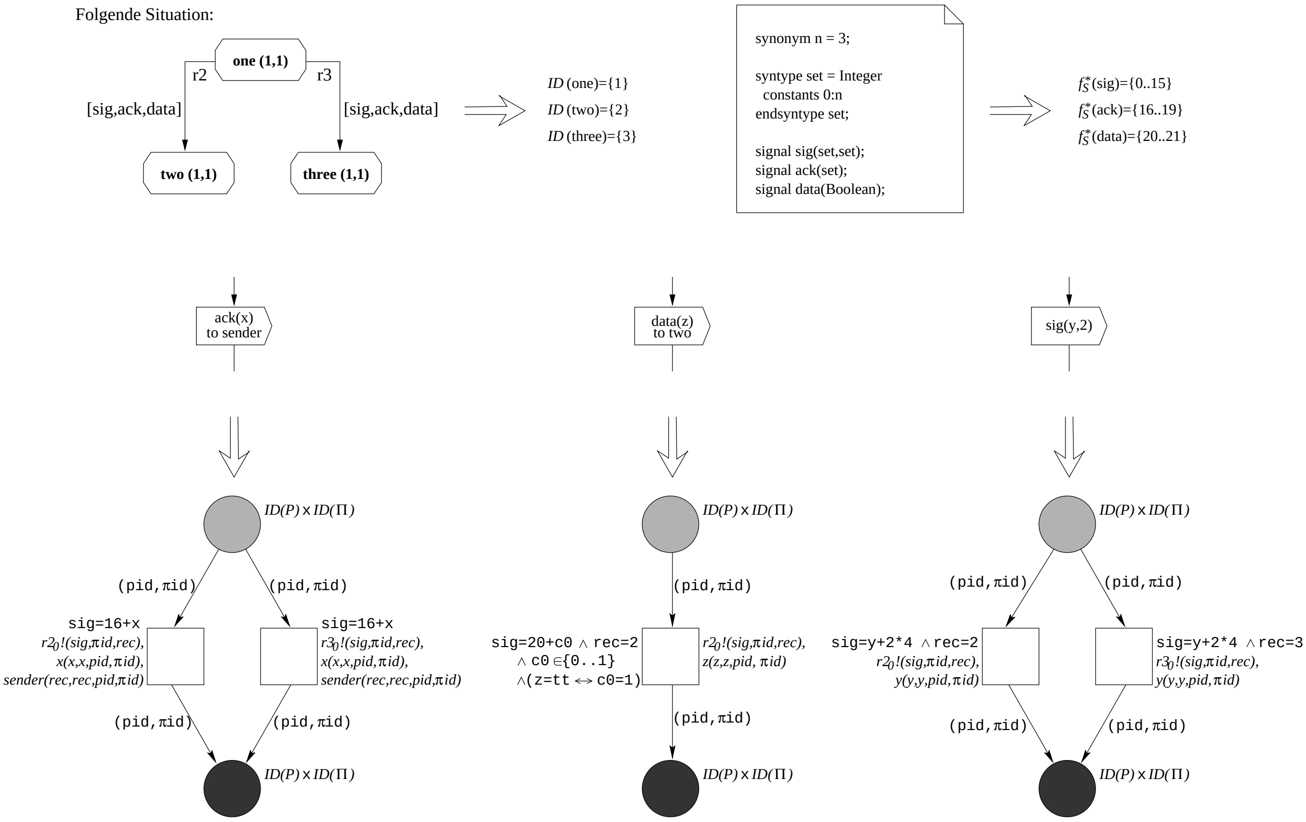

Abbildung 4.18: Beispiele für die Übersetzung von **output**

4.2.13 Übersetzung von task

task-Anweisungen dienen der Wertzuweisung an Variablen. Mit einer Anweisung können mehrere Wertzuweisungen erfolgen. An implizite Variablen eines Prozesses kann keine Zuweisung erfolgen.

task	::=	**task** assign–klist **;**
assign	::=	`variable–ident` **:=** expr
expr	::=	expr bin–op expr | un–op expr |
		(expr **)** | `variable–ident` | const
const–expr	::=	const–expr bin–op const–expr |
		un–op const–expr | **(** const–expr **)** | const
bin–op	::=	$\Rightarrow$ | **or** | **xor** | **and** | **=** | $\neq$ | > |
		$\geq$ | < | $\leq$ | **+** | − | ∗ | **/** | **mod**
un–op	::=	− | **not**
const	::=	**True** | **False** | **Null** | z

Für den Aufbau von Ausdrücken gelten die üblichen Prioritäten von Operatoren und Regeln über die Verträglichkeit von Operatoren und Operanden.

Mehrere Zuweisungen in einer Anweisung werden nacheinander ausgeführt:

$$\zeta(\textbf{task}\ \text{assign}\textbf{,}\ \text{assign–klist}\textbf{;}) = \zeta(\textbf{task}\ \text{assign}\textbf{;})\,;\zeta(\textbf{task}\ \text{assign–klist}\textbf{;})$$

Jede einzelne Anweisung wiederum wird durch ein einfaches Netz übersetzt, wobei sich die Beschriftung der Transition rekursiv aus dem verwendeten Wertausdruck ableitet (siehe Abbildung 4.19), ähnlich wie bei der Übersetzung atomarer Aktionen in $\mathrm{B(PN)}^2$ [FG97].

$$\begin{aligned}&\zeta(\textbf{task}\ \texttt{variable–ident} := \text{expr}\textbf{;})\\ &\qquad = \ M_{task}(\texttt{variable–ident}, \text{expr}, f_R(\texttt{variable–ident} := \text{expr}))\end{aligned}$$

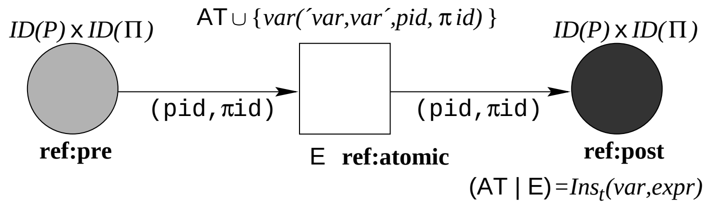

Abbildung 4.19: $M_{task}(var, expr, \mathbf{ref})$

$$\begin{aligned}&Ins_t(var, \text{expr}_1 \text{ bin–op } \text{expr}_2) = (AT_1 \cup AT_2 \mid E_1 \text{ bin–op } E_2)\\ &\qquad \text{mit } (AT_1 \mid E_1) = Ins_t(var, \text{expr}_1)\\ &\qquad \text{und } (AT_2 \mid E_2) = Ins_t(var, \text{expr}_2)\\ &Ins_t(var, \text{un–op expr}) = (AT \mid \text{un–op } E) \qquad \text{mit } (AT \mid E) = Ins_t(var, \text{expr})\\ &Ins_t(var, (\text{expr})) = (AT \mid (E)) \qquad \text{mit } (AT \mid E) = Ins_t(var, \text{expr})\\ &Ins_t(var, \text{const}) = (\emptyset \mid \text{const})\\ &Ins_t(var, var) = (\emptyset \mid \ 'var)\\ &Ins_t(var, x) = (x(x, x, pid, \pi id) \mid x) \qquad \text{falls } x \text{ eine Variable ist und } x \neq var\end{aligned}$$

Die beiden letzten Definitionen von Ins_t regeln die Behandlung der Vorwerte und Nachwerte von Variablen. Taucht die Variable, der ein Wert zugewiesen wird, auf der rechten Seite der Zuweisung auf, so muß ihr Vorwert benutzt werden. Kommen andere Variablen auf der rechten Seite vor, so wird durch die Verwendung gleicher Kommunikationsvariablen für Vorwert und Nachwert sichergestellt, daß sich der Wert der Variablen nicht ändert.

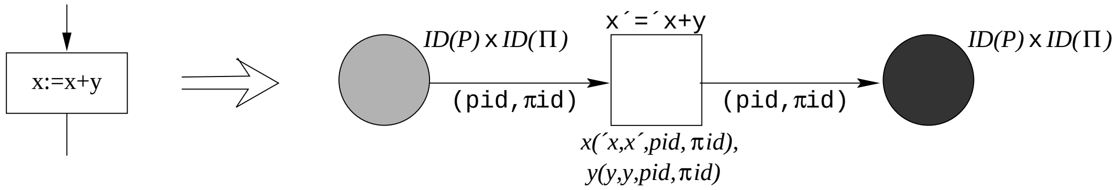

Abbildung 4.20: Übersetzungsbeispiel für **task**

4.2.14 Übersetzung von decision

Mit **decision** kann der Kontrollfluß eines Prozesses bzw. einer Prozedur verzweigt werden. Zu unterscheiden sind deterministische und nichtdeterministische Verzweigung.

decision	::=	d–decision | nd–decision

Eine **decision** gilt als terminator, falls alle ihre Zweige mit einem terminator enden. Eine solche terminierende **decision** darf am Ende eines Zustandsüberganges stehen, siehe Abschnitt 4.2.11.

Deterministische Verzweigung

Bei der deterministischen Verzweigung wird ein Ausdruck (die Frage) evaluiert. Jeder Zweig ist mit einer Antwort, d.h. einem Wertebereich, versehen. Der Kontrollfluß geht über in denjenigen Zweig, dessen Antwort auf die Frage paßt. Die Antworten aller Zweige müssen paarweise disjunkt sein. Trifft keine der Antworten zu, so wird der **else**-Zweig genommen. Ein **else**-Zweig muß zwingend vorhanden sein, falls die Antworten aller anderen Zweige nicht auf alle möglichen Werte der Frage passen.

d–decision	::=	d–decision–header d–answer–list opt–else decision–trailer
d–decision–header	::=	**decision** expr **;**
d–answer	::=	**(** range–klist **)** **:** transition
opt–else	::=	**else :** transition | ε
decision–trailer	::=	**enddecision ;**

Zur Definition von range siehe Abschnitt 4.2.2. Für die Darlegung der Semantik kann o.B.d.A. angenommen werden, daß ein **else**-Zweig fehlt, da ein solcher Fall äquivalent durch einen normalen Zweig ersetzt werden kann, dessen Antwort das Komplement der Vereinigung aller anderen Antworten ist. (Tatsächlich wird dies in der Implementation der Semantik auch so gehandhabt, siehe das Beispiel in Abbildung 4.22)

$$\begin{aligned}&\zeta(\text{d–decision–header d–answer d–answer–list decision–trailer})\\ &\qquad = \zeta(\text{d–decision–header d–answer decision–trailer})\\ &\qquad\quad\; [] \; \zeta(\text{d–decision–header d–answer–list decision–trailer})\end{aligned}$$

$$\begin{aligned}&\zeta(\textbf{decision}\text{ expr}\textbf{;}\ \textbf{(}\text{range–klist}\textbf{)}\textbf{:}\text{ transition decision–trailer})\\ &\qquad = M_{decision}(\text{expr}, \text{range–klist}, f_R(\text{expr}), f_R((\text{range–klist}\textbf{):})); \zeta(\text{transition})\end{aligned}$$

Abbildung 4.21 zeigt das Netz $M_{decision}$. Der in der Beschriftung der Transition verwendete Hilfsausdruck *VALUES*(range–klist) liefere die Menge der Werte, die durch range–klist spezifiziert wird (die exakte Definition von *VALUES* ließe sich sehr einfach aus der Grammatik herleiten, dürfte aber auch ohne explizite Angabe offensichtlich sein).

Die Hilfsfunktion Ins_d ist ähnlich wie bei der **task**-Anweisung rekursiv definiert, allerdings entfällt die für **task** bekannte Sonderbehandlung für die Variable auf der linken Seite der Gleichung.

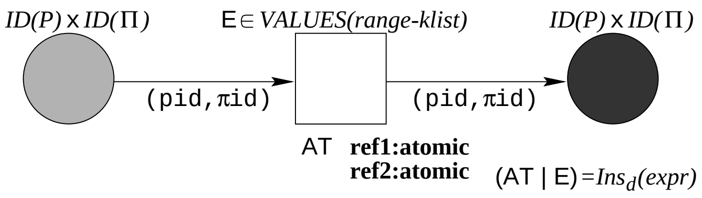

Abbildung 4.21: $M_{decision}(expr, range\text{-}klist, \mathbf{ref1}, \mathbf{ref2})$

$$
\begin{aligned}
&\mathit{Ins}_d(\mathrm{expr}_1 \text{ bin–op } \mathrm{expr}_2) = (AT_1 \cup AT_2 \mid E_1 \text{ bin–op } E_2)\\
&\qquad \text{mit } (AT_1 \mid E_1) = \mathit{Ins}_d(\mathrm{expr}_1)\\
&\qquad \text{und } (AT_2 \mid E_2) = \mathit{Ins}_d(\mathrm{expr}_2)\\
&\mathit{Ins}_d(\text{un–op expr}) = (AT \mid \text{un–op } E) \quad \text{mit } (AT \mid E) = \mathit{Ins}_d(\mathrm{expr})\\
&\mathit{Ins}_d((\mathrm{expr})) = (AT \mid (E)) \quad \text{mit } (AT \mid E) = \mathit{Ins}_d(\mathrm{expr})\\
&\mathit{Ins}_d(\mathrm{const}) = (\emptyset \mid \mathrm{const})\\
&\mathit{Ins}_d(x) = (x(x, x, pid, \pi id) \mid x), \quad \text{falls } x \text{ eine Variable ist}
\end{aligned}
$$

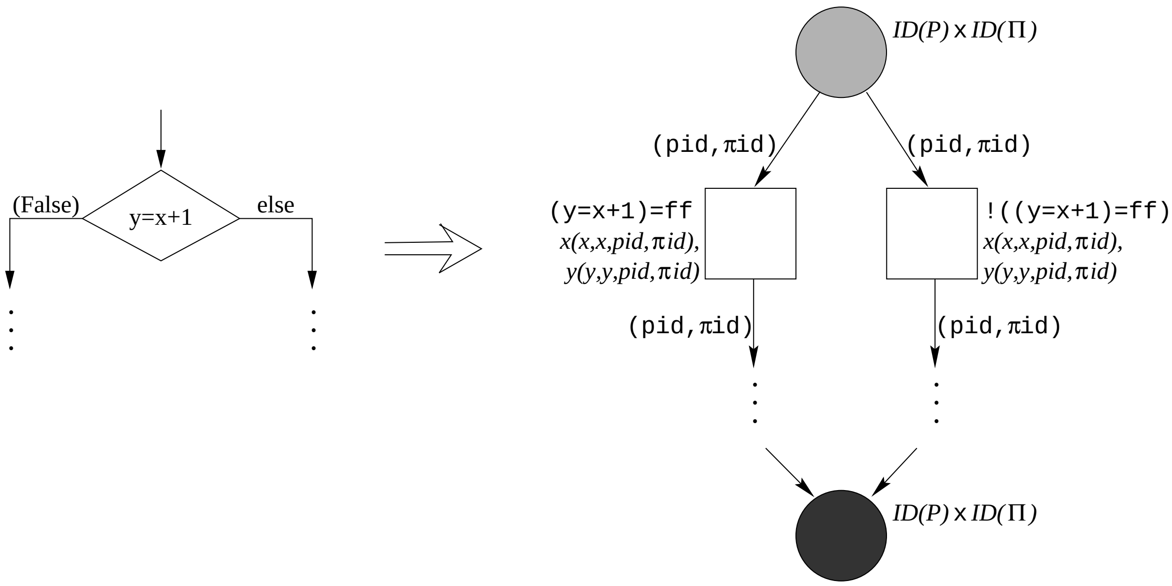

Abbildung 4.22: Beispiel für die Übersetzung von **decision** (deterministisch)

Nichtdeterministische Verzweigung

Die nichtdeterministische Verzweigung gestaltet sich erheblich einfacher. Sie wird durch das Schlüsselwort **any** im Frageteil der Anweisung eingeleitet. Die einzelnen Zweige sind nicht durch Antworten gekennzeichnet; einer von ihnen wird zufällig gewählt und ausgeführt.

nd–decision	::=	nd–decision–header nd–answer–list decision–trailer
nd–decision–header	::=	**decision any ;**
nd–answer	::=	**( ) :** transition

Dementsprechend einfacher gestaltet sich die Semantik:

$$
\begin{aligned}
&\zeta(\text{nd–decision–header nd–answer nd–answer–list decision–trailer})\\
&\qquad = \zeta(\text{nd–decision–header nd–answer decision–trailer})\\
&\qquad\quad\ [\,]\ \zeta(\text{nd–decision–header nd–answer–list decision–trailer})
\end{aligned}
$$

$$
\begin{aligned}
&\zeta(\text{nd–decision–header nd–answer decision–trailer})\\
&\qquad = \zeta(\textbf{decision any; ():}\text{transition decision–trailer})\\
&\qquad = M_{any}(f_R(\textbf{any}), f_R(\textbf{():})); \zeta(\text{transition})
\end{aligned}
$$

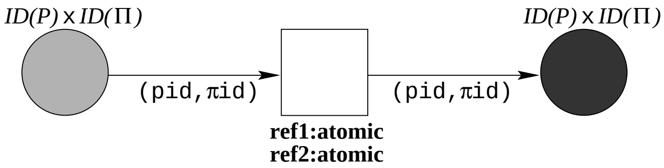

Abbildung 4.23: $M_{any}(\textbf{ref1}, \textbf{ref2})$

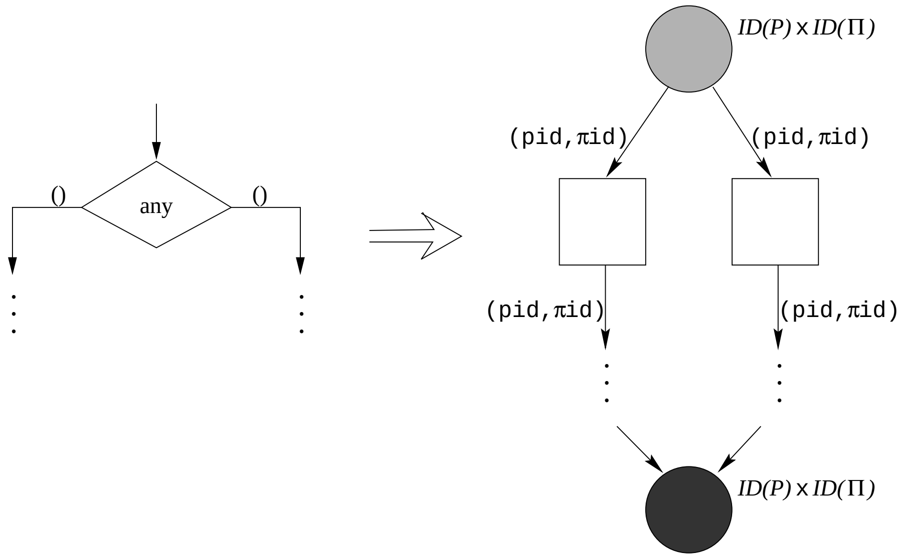

Abbildung 4.24: Beispiel für die Übersetzung von **decision** (nichtdeterministisch)

4.2.15 Übersetzung von call

Mit dem Befehl **call** wird eine Prozedur aufgerufen. Der Kontrollfluß des aufrufenden Prozesses/Prozedur hält solange an, wie die Prozedur läuft, und fährt dann mit dem auf **call** folgenden Befehl fort.

Beim Aufruf einer Prozedur können mehrere Parameter übergeben werden. SDL unterscheidet dabei zwei Arten von Parametern (vgl. Abschnitt 4.2.8):

- **in**-Parameter ("call by value"): Beim Aufruf kann ein beliebiger Ausdruck übergeben werden. Der Ausdruck wird einmal vor Beginn der Prozedur evaluiert, und der Parameter bekommt das Ergebnis dieser Auswertung zugewiesen.

- **in/out**-Parameter ("call by reference"): Hier muß eine Variable als Parameter übergeben werden. Jeder Zugriff auf den Parameter innerhalb einer Prozedur ist dann ein Zugriff auf die Variable, die beim Aufruf übergeben wurde.

call	::=	**call** `procedure–ident` **;** |
		call `procedure–ident` **(** callparm–klist **)** **;**
callparm	::=	expr | `variable–ident` | ε

Anzahl und Typ der Parameter beim Aufruf müssen mit den formalen Parametern der aufgerufenen Prozeduren übereinstimmen. Argumente für **in**-Parameter können weggelassen werden; ihr Wert innerhalb der Prozedur ist dann unbestimmt.

Die Übersetzung eines Aufrufes ohne Parameter gestaltet sich recht einfach (siehe Abbildung 4.25):

$$\zeta(\textbf{call}\ \texttt{procedure–ident;})$$
$$= M_{call}(\texttt{procedure–ident}, f_R(\textbf{call}\ \texttt{procedure–ident}))$$

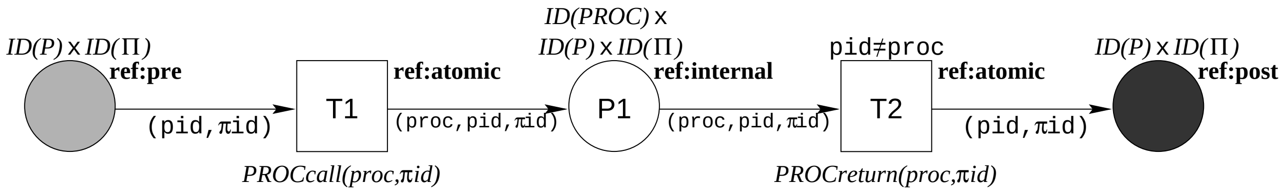

Abbildung 4.25: $M_{call}(PROC, \mathbf{ref})$

Für jeden übergebenen Parameter muß das Netz erweitert werden. Es sei AT die Menge der Aktionsterme und E die Menge der Konjunktionsglieder, um die T1 erweitert wird. Dann gilt

- für jeden **in**-Parameter: Das Aktionssymbol $\texttt{procedure–ident}_{call}$ wird um eine Kommunikationsvariable pin_m erweitert (falls es sich um den m-ten Parameter in der Argumentliste handelt). Falls für den Parameter tatsächlich ein Ausdruck expr angegeben ist (der Parameter also nicht leer gelassen wurde), werden AT um die Menge AT_1 und E um die Menge $pin_m = E_1$ erweitert. Dabei ist $(AT_1 \mid E_1) = Ins_d(\text{expr})$ wie bei **decision** bestimmt.

- für jeden **in/out**-Parameter: Das Netz wird um eine Referenzdurchreiche erweitert, d.h. um ein Stellen/Transitionspaar wie P11 und T11 aus Abbildung 4.26. Der Name des formalen Parameters aus der Prozedurdeklaration wird wie in Abschnitt 4.2.8 beschrieben umbenannt, indem er um den Namen der Prozedur erweitert wird. Die Wahl der Aktionsterme in T11 sorgt dafür, daß jeder Zugriff auf den Parameter (im Beispiel: val) auf die übergebene Variable (im Beispiel: x) umgeleitet wird.

Die Abbildung 4.26 zeigt Beispiele für den Aufruf von Prozeduren mit einem **in**- bzw. mit einem **in/out**-Parameter.

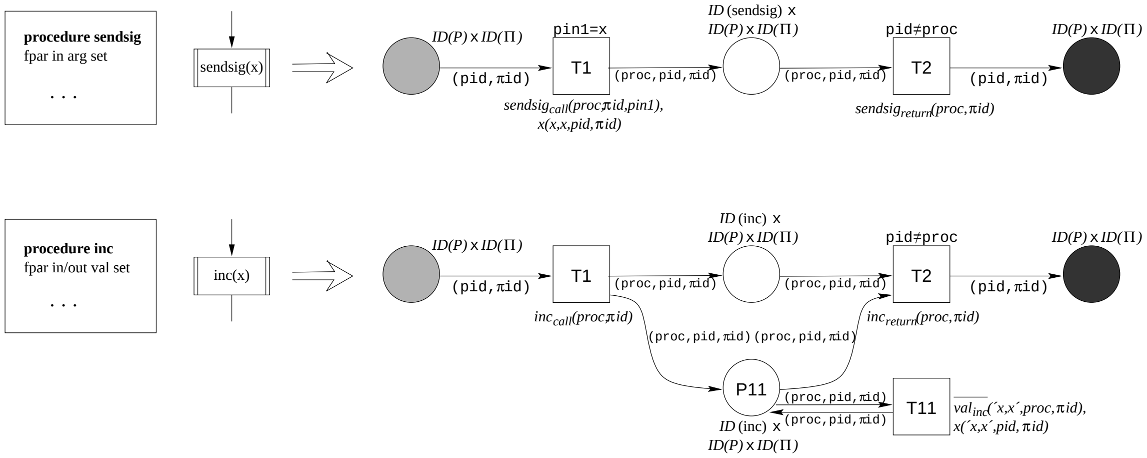

Abbildung 4.26: Beispiele für die Übersetzung von **call**

4.2.16 Übersetzung von create

Mit **create** wird eine neue Instanz eines Prozeßtyps erzeugt. Prozesse können nicht über Blockgrenzen hinweg erzeugt werden, d.h., der erzeugte Prozeß muß im selben Block liegen wie der erzeugende.

Als Resultat einer **create**-Anweisung wird (sofern dies möglich ist) eine neue Prozeßinstanz initiiert, deren implizite **parent**-Variable auf die ID des erzeugenden Prozesses gesetzt wird. Die Variable **offspring** im erzeugenden Prozeß bekommt die ID des erzeugten Prozesses zugewiesen. War eine Erzeugung nicht möglich, weil bereits die maximal mögliche Anzahl von Instanzen des gewünschten Prozeßtyps existierte, so wird kein Prozeß erzeugt, und die Variable **offspring** erhält den Wert **Null**.

Dem erzeugten Prozeß können Argumente übergeben werden, und zwar gemäß der Liste der formalen Parameter, wie sie bei der Deklaration des Prozesses angegeben wurde. Dabei sind nur "call by value"-Parameter möglich (vgl. die **in**-Parameter bei Prozeduren). Argumente können ausgelassen werden, die Werte der entsprechenden Parameter im erzeugten Prozeß sind dann undefiniert.

Die erzeugte Prozeßinstanz läuft so lange, bis sie auf eine **stop**-Anweisung stößt. Sie kann nicht "gewaltsam" von anderen Prozessen beendet werden, auch nicht vom erzeugenden Prozeß.

create	::=	**create** `process–ident` **;** |
		create `process–ident` **(** createparm–klist **)** **;**
createparm	::=	expr | ε

Auch hier sei zunächst wieder der einfache Fall ohne Parameter dargestellt:

$$\zeta(\textbf{create}\ \texttt{process–ident;}) \\ = M_{create}(\texttt{process–ident}, f_R(\textbf{create}\ \texttt{process–ident}))$$

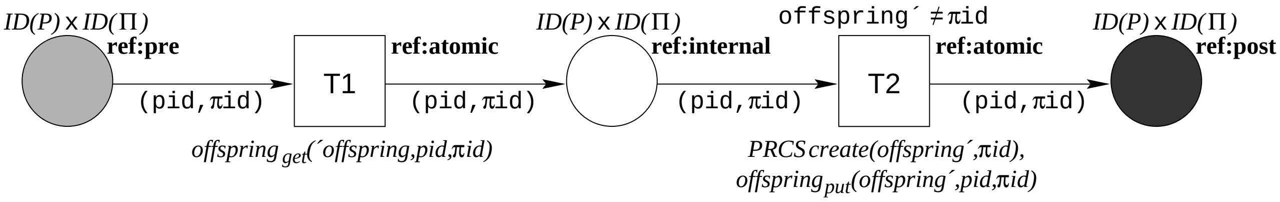

Abbildung 4.27: $M_{create}(PRCS, \mathbf{ref})$

Abbildung 4.27 zeigt das verwendete Netz M_{create}, welches zwei Transitionen besitzt. Die erste zieht die augenblickliche Marke von der Wertestelle für **offspring** ab, die zweite erzeugt den neuen Prozeß und speichert den neuen Wert von **offspring** (siehe auch Transitionen T9' und T9" in Abbildung 4.2.7). Die Implikationen dieser Aufteilung sind in Abschnitt 5.13 aufgeführt.

Für jeden Parameter wird dann die Transition T2 erweitert, und zwar analog zu der Weise, wie dies für **in**-Parameter im Abschnitt über **call** beschrieben ist, d.h., das Aktionssymbol $\texttt{process–ident}_{create}$ wird um eine Kommunikationsvariable erweitert, und zusätzliche Schaltbedingungen und Aktionssymbole für Variablen werden eingeführt (genauere Beschreibung in Abschnitt 4.2.7).

Abbildung 4.28 zeigt den Aufruf mit einem Parameter anhand eines Beispiels.

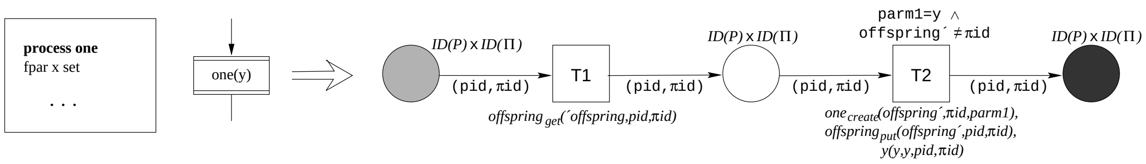

Abbildung 4.28: Beispiel für die Übersetzung von **create** mit einem Argument.

4.2.17 Übersetzung von nextstate

Die Anweisung **nextstate** beendet einen Zustandsübergang und legt einen neuen Zustand fest. Wie auch alle anderen terminierenden Anweisungen kann **nextstate** nur als letzte Anweisung in einer transition stehen.

nextstate ::= **nextstate** `state–ident` **;** |
nextstate – **;**

Mit der speziellen Form "**nextstate** –" wird ausgedrückt, daß der alte Zustand erhalten bleibt. Die in der Semantik verwendeten Netze sind in den Abbildungen 4.29 und 4.30 dargestellt.

$$\zeta(\textbf{nextstate}\ \texttt{state–ident;}) = M_{nextstate}(\texttt{state–ident}, f_R(\textbf{nextstate}\ \texttt{state–ident}))$$
$$\zeta(\textbf{nextstate} - \textbf{;}) = M_{nextstatedash}(f_R(\textbf{nextstate} -))$$

Dabei steht $f_Q(\texttt{state–ident})$ für die in Abschnitt 4.1.4 erläuterte Bijektion zwischen Zustandsbezeichnern des SDL-Quelltextes und natürlichen Zahlen.

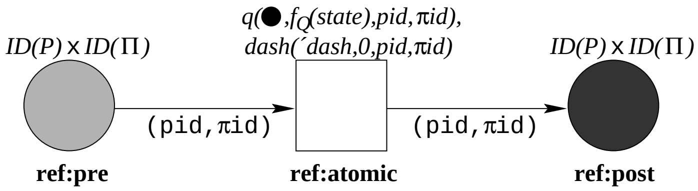

Abbildung 4.29: $M_{nextstate}(state, \mathbf{ref})$

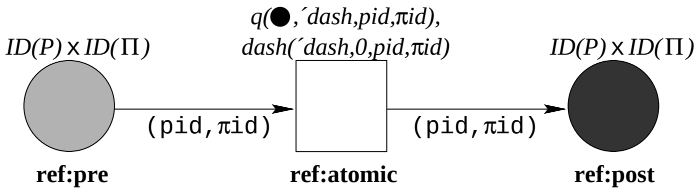

Abbildung 4.30: $M_{nextstatedash}(\mathbf{ref})$

4.2.18 Übersetzung von join

Mit **join** wird der Kontrollfluß zu einer anderen, durch eine Sprungmarke gekennzeichneten Stelle innerhalb desselben Prozeß-/Prozedurrumpfs umgeleitet (quasi ein "goto"-Befehl). Das verwendete Netz ist in Abbildung 4.31 zu sehen.

join ::= **join** `label-ident` **;**

$$\zeta(\textbf{join}\ \texttt{label-ident;}) = M_{join}(\texttt{label-ident}, f_R(\textbf{join}\ \texttt{label-ident}))$$

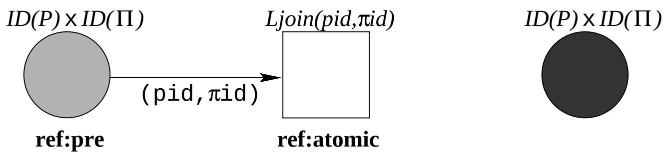

Abbildung 4.31: $M_{join}(L, \mathbf{ref})$

Da ein **join** nur am Ende eines Zustandsübergangs auftreten darf (entweder direkt oder als Schlußanweisung in einer terminierenden **decision**), verbindet sich die Exit-Stelle von M_{join} mit der zentralen Stelle des Iterationskonstrukts, welches den Kontrollfluß des/der zugehörigen Prozesses/Prozedur darstellt (vgl. Abschnitt 4.2.10). Die Exit-Stelle "verschwindet" also, und bleibt nicht als isolierte Stelle zurück.

4.2.19 Übersetzung von stop und return

Der Befehl **stop** beendet die Ausführung eines Prozesses. Die Instanz hört auf zu existieren, und ihre Prozeß-ID wird wieder freigegeben. Ein **stop** darf nur innerhalb eines Prozeßrumpfes auftreten (d.h., ein Prozeß kann niemals innerhalb einer Prozedur beendet werden).

stop ::= **stop ;**

Der Befehl **return** ist nur innerhalb einer Prozedur zulässig; er beendet die Ausführung der betreffenden Prozedurinstanz und führt dazu, daß die Ausführung des/der aufrufenden Prozesses/Prozedur fortgesetzt wird.

return ::= **return ;**

Für die Zwecke der Übersetzung sind beide Befehle gleich: Sie sorgen dafür, daß der Zustand des Prozesses bzw. der Prozedur auf 0 gesetzt wird, so daß der End-Teil der Rumpf-Iteration (siehe Abschnitt 4.2.10) ausgeführt wird.

$$\zeta(\text{stop}) = M_{stop}(f_R(\text{stop}))$$
$$\zeta(\text{return}) = M_{stop}(f_R(\text{return}))$$

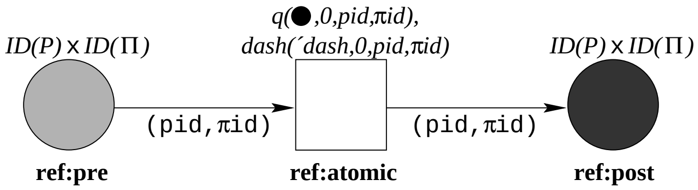

Abbildung 4.32: Das Netz $M_{stop}(\mathbf{ref})$

Kapitel 5

Diskussion

In diesem Kapitel soll näher auf die in Kapitel 4 vorgestellte Semantik von SDL-Systemen eingegangen werden. Einige Punkte der Semantik werden aufgegriffen und erläutert; insbesondere wird dargelegt, inwieweit die Semantik konform zu [CCI92] ist. Gegenüber der in [FG98a] vorgestellten Semantik, auf der diese Arbeit basiert, wurden einige Änderungen vorgenommen. Diese und andere Unterschiede werden aufzeigt und Vergleiche angestellt. In einigen Fällen werden Alternativen zur in Kapitel 4 gezeigten Semantik präsentiert.

5.1 Allgemeines

Um die M-Netz-Semantik einer SDL-Spezifikation formal anzugeben, wird eine Semantik-Funktion ζ definiert. Diese liefert zu einer SDL-Spezifikation S ein M-Netz $\zeta(S)$. Im wesentlichen geschieht dies kompositionell, d.h., S wird in seine Bestandteile zerlegt, und die Semantik des Gesamtsystems ergibt sich durch die Komposition der Einzelteile.

Einige Aspekte des Systems lassen sich nicht kompositionell darstellen. So ist die exakte Semantik eines Prozeduraufrufs abhängig von der Deklaration der formalen Parameter in der aufgerufenen Prozedur, die jedoch an einer ganz anderen Stelle im Quelltext steht. Andere Dinge, z.B. die Menge der Zustände eines Prozesses, lassen sich nicht mit sinnvollem Aufwand formal definieren. Um eine Konzentration auf die wesentlichen Aussagen zu ermöglichen, wurden in beiden Fällen Hilfsfunktionen angegeben, deren Bedeutung informal beschrieben wurde.

Im Unterschied zu [FG98a] wurde auch darauf verzichtet, die ID-Mengen für Prozeduren und Prozesse in der Semantikfunktion “durchzuziehen”; abgesehen davon, daß dies die rekursiven Definitionen für einzelne SDL-Anweisungen nicht gerade übersichtlicher gemacht hätte, ist die Zugehörigkeit von Anweisungen zu Prozessen bzw. Prozeduren durch die Lage im Quelltext eindeutig bestimmt, so daß diese Entscheidung die Klarheit der Semantik-Definition nicht beeinträchtigen sollte. Wäre ζ wie in [FG98a] so definiert worden, daß sie die ID-Mengen für Prozeduren und Prozesse als zusätzliche Argumente enthält, so wäre dies nur bei zwei Regeln von besonderem Interesse, und zwar bei der Gleichung von ζ(process) auf Seite 31 und von ζ(procedure) auf Seite 35. An diesen Stellen müßten die jeweils passenden ID-Mengen an N_{body}(process) bzw. N_{body}(procedure) und ζ(procedure–decls) übergeben werden; in allen anderen Regeln der Semantik würden die ID-Mengen für Prozesse und Prozeduren unverändert weitergereicht.

5.2 Bemerkungen zu Systemen

Ein SDL-System besteht aus einer Menge gleichzeitig arbeitender Prozesse, sowie einer Anzahl von Kanälen und Signalrouten, die ebenfalls gleichzeitig arbeiten und Signale zwischen den Prozessen transportieren. Die Semantik berücksichtigt diese Gleichzeitigkeit, indem die Netze für alle diese Bestandteile parallel komponiert werden. Ein globales Kontrollflußnetz sorgt für die korrekte Initialisierung und Terminierung der einzelnen Teile.

Im Kontrollfluß werden zunächst alle Bestandteile initialisiert; danach werden die anfänglichen Prozeß-Instanzen erzeugt. Die Simulation eines SDL-Systems dauert dann so lange, bis keine Prozeßinstanzen mehr existieren; dies wird durch die Prozeßterminierungs-Aktion (siehe Abbildung 4.2) sichergestellt, die alle Prozesse gleichzeitig terminiert und erst dann schalten kann, wenn alle Prozeßinstanzen beendet sind. Sequentielle Terminierung würde hier zu Fehlverhalten führen. Würde ein Prozeß terminiert, während andere Prozesse noch aktiv sind, so könnten nachfolgende **create**-Befehle für den terminierten Prozeß nicht mehr ausgeführt werden. Bei der Implementation der parallelen Prozeßterminierung wurde darauf geachtet, daß die LL-Netz-Entfaltung nicht unnötig viele Transitionen erzeugt, siehe Abschnitt 5.6.

Im Gegensatz zu der in [FG98a] vorgeschlagenen Lösung findet die Synchronisation über die Weiterleitung von Signalen an Prozesse und Signalrouten (etwa *process*! oder *route*!) nicht auf Systemebene, sondern auf Blockebene statt. Dies macht Sinn, da ein Prozeß Eingaben nur von solchen Signalrouten erhalten kann, die im selben Block wie der Prozeß liegen, und Signalrouten ihre Eingaben durch Verbindungen (connects) erhalten, die ebenfalls im selben Block liegen. Außerdem wird durch Synchronisation auf Blockebene gewährleistet, daß Schreiboperationen auch dann noch korrekt funktionieren, wenn gleichnamige Signalrouten in unterschiedlichen Blöcken existieren.

Die Synchronisation über Initialisierungs- und Terminierungsaktionen von Signalrouten findet dagegen auf Systemebene statt. Im Falle gleichnamiger Signalrouten kann es dann passieren, daß sich Transitionen des Kontrollflusses mit mehreren Signalrouten verbinden (siehe Abbildung 5.1). Dies beeinträchtigt aber nicht die Funktionsfähigkeit des Systems, da bei der Simulation jede Transition eine andere Signalroute initialisiert; nur die Reihenfolge der Initialisierungen kann anders sein als geplant.

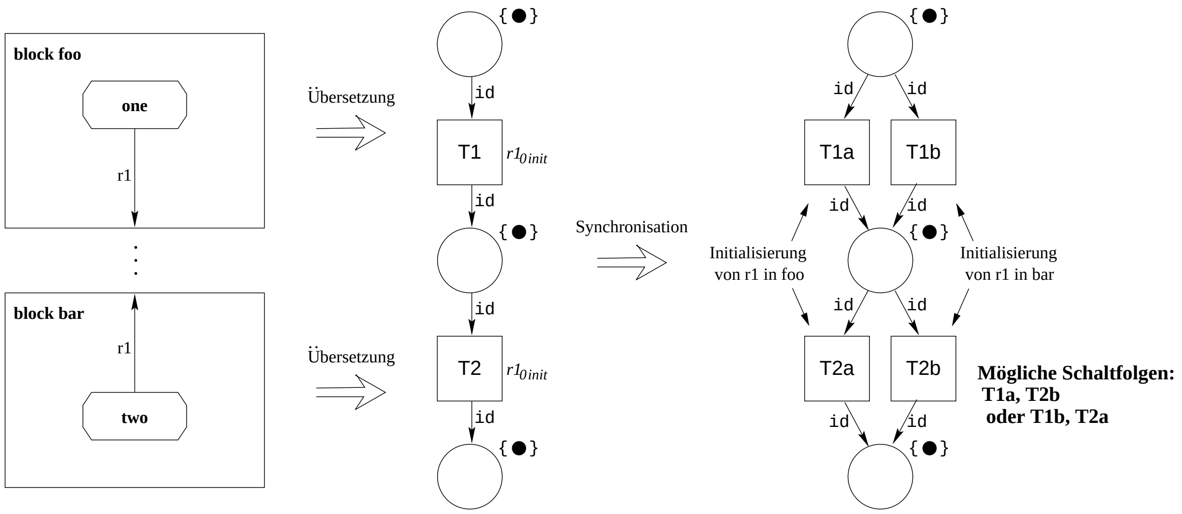

Abbildung 5.1: Globaler Kontrollfluß bei gleichnamigen Signalrouten

5.3 Bemerkungen zu Kanälen

Kanäle verbinden Blöcke und transportieren Signale mit Verzögerung. Das (standardmäßig) für Kanäle verwendete Netz ist in Abbildung 4.3 auf Seite 24 gezeigt.

Kanäle erhalten Signale durch **output**-Befehle in Prozessen (über Signalrouten und Verbindungen). Eingaben an einen Kanal kommen im Netz durch Transition T11 zustande. Die Ausgaben laufen über Transition T12; sie gehen über Verbindungen und Signalrouten an die Eingabewarteschlange der Empfängerprozesse. Die (nichtdeterministische) Verzögerung der Weiterleitung entsteht durch die Pause, die zwischen dem Schalten dieser Transitionen entsteht. In dieser Zwischenzeit werden Signale in der Stelle P12 gespeichert. Beachtenswert erscheint, daß durch die spezielle Konstruktion von Signalrouten und Verbindungen alle Schreibtransitionen auf dem Weg zwischen Kanal und Prozeß bei der Synchronisation zusammenfallen; Signale geraten also im synchronisierten Netz direkt (ohne weitere Zwischenspeicherung) aus dem Kontrollfluß eines Sender-Prozesses in den Kanal bzw. vom Kanal in die Eingabewarteschlange des Empfänger-Prozesses. Zusammen mit dem Signal selbst werden Informationen über den Sender und den Empfänger des Signals weitergeleitet.

Die Stelle P11 zählt, wieviele Signale sich zur Zeit im Kanal befinden. Eingaben an den Kanal werden stets an der ersten, "hintersten" Stelle im Puffer eingefügt, was nur möglich ist, wenn diese leer ist. Um mehrere Signale speichern zu können, werden Signale mit der "Shift"-Transition T13 innerhalb des Puffers nach vorne geschoben. Ausgegeben wird immer nur das vorderste Element in P12. Dadurch ist gewährleistet, daß ein Kanal als FIFO-Puffer arbeitet.

Abbildung 5.2 zeigt ein alternatives Kanalnetz, welches auf einem in [FG98a] gemachten Vorschlag basiert. Im Gegensatz zum Netz in 4.3 (welches vom Compiler standardmäßig verwendet wird) käme dieses Netz im Falle eines unendlich großen Puffers mit endlichen Kantenbeschriftungen aus.

Die Verwendung des in Abbildung 5.2 gezeigten Netzes kann im Compiler durch Angabe der Option `-BC1` erzwungen werden (siehe auch Kapitel 6).

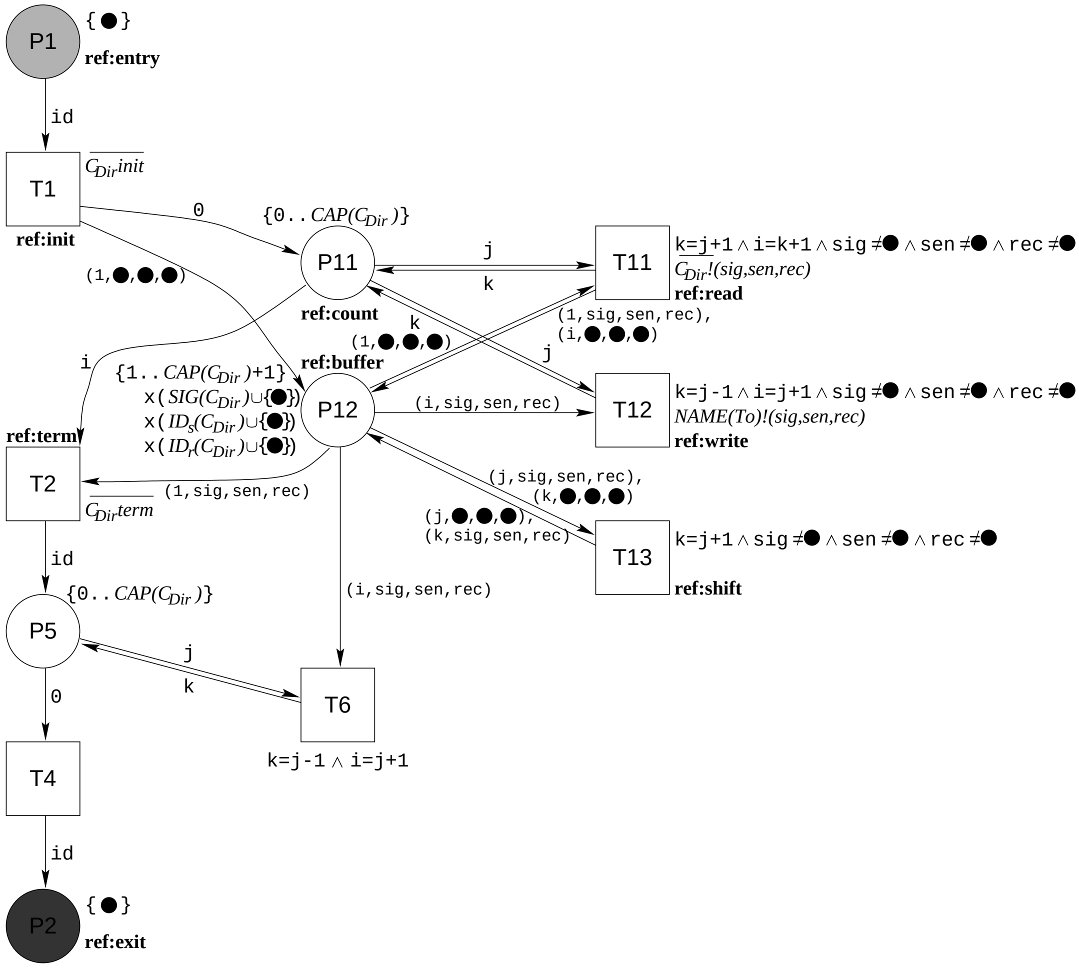

Abbildung 5.2: Alternatives Kanalnetz

Tabelle 5.1 stellt Vergleiche zwischen beiden Netzvarianten an. Dazu wurde das Beispiel aus Kapitel 2 mit unterschiedlichen Compileroptionen übersetzt. Die Größenangaben beziehen sich jeweils auf das gesamte Netz. Auf der linken Seite (unter der Überschrift `-DC1`) werden Ergebnisse für Kanäle mit Kapazität 1 aufgeführt, rechts Ergebnisse für Kapazität 2. Es zeigt sich, daß bei der standardmäßig verwendeten Variante aus Abbildung 4.3 das Netz mit steigender Kanalkapazität enorm anwächst, wobei viele Transitionen erzeugt werden, die niemals schalten können. Werden diese "toten" Transitionen entfernt, so liefern beide Varianten sehr ähnliche Ergebnisse. Auf die Effizienz des Modelcheckers wirken sich beide Varianten gleich aus, die Zahl der Cutoff-Ereignisse und Konfigurationen bleibt gleich.

Als Erweiterung gegenüber [FG98a] behandelt die vorliegende Implementation auch bidirektionale Kanäle. Diese werden wie zwei separate unidirektionale Kanäle übersetzt, deren Namen durch Indizes 0 und 1 unterschieden werden. In den Abschnitten über Verbindungen wird die Verwendung des Index im Namen implizit vorausgesetzt.

Beim Vergleich der vorliegenden Lösungen mit der Definition von Kanälen in der Z.100 [CCI92] fallen einige Unterschiede auf. So geht die Z.100 von Kanälen aus, deren Kapazität unbeschränkt ist. Während die Theorie der HL-Netze dies zulassen würde,

		-DC1		-DC2	
		-BC0	-BC1	-BC0	-BC1
LL-Netz:	Stellen	407	477	475	617
	Transitionen	775	861	7293	1249
Minimiertes LL-Netz:	Stellen	278	293	291	293
	Transitionen	190	203	201	203
Entfaltung:	Cutoff-Ereignisse	11	11	11	11
	Konfigurationen	12	12	12	12

Tabelle 5.1: Vergleich der verschiedenen Kanalnetze mit Kapazitäten 1 und 2.

sind die LL-Netz-basierten Verifikationsmethoden für endliche Netze ausgelegt. Da die Entfaltung des M-Netzes ein unendliches LL-Netz ergäbe, muß, entgegen dem Z.100-Standard, eine endliche Schranke für die Kapazität von Kanälen eingeführt werden. Die Möglichkeiten, wie diese Schranke vom Benutzer festgelegt werden kann, sind in Kapitel 6 dargelegt. Die maximal notwendige Kapazität eines Kanals (wenn diese endlich ist), kann mit Hilfe des Modelcheckers festgestellt werden, indem eine feste Kapazität angenommen und angefragt wird, ob der Puffer des Kanals jemals ganz gefüllt werden kann.

Ein Nachteil der zur Zeit verwendeten Lösung ist die Tatsache, daß Signale an tote Prozesse nicht wie in der Z.100 vorgesehen verfallen. Da ein solches Signal nicht mehr an den Prozeß ausgeliefert werden kann, verstopft es den Kanal, bis eine neue Prozeßinstanz erzeugt wird, die (zufällig) die ID des toten Prozesses wiederverwendet. Erst dann würde das Signal ausgeliefert, allerdings an eine Instanz, für die das Signal gar nicht bestimmt war! Um diesen unerwünschten Effekt zu verhindern, könnte man das Prozeßnetz so abändern, daß Eingaben (ohne Speicherung des Signals) an solche Prozeß-IDs möglich sind, die sich im Stack mit den verfügbaren IDs befinden. Es wäre von Vorteil, eine solche Lösung so zu implementieren, daß eine kleine Erweiterung des Formeltransformators (Kapitel 7) es erlauben würde, die Möglichkeit des Eintretens dieses Falles abzufragen.

5.4 Bemerkungen zu Signalrouten

Signalrouten transportieren Signale ohne Verzögerung. In dem Netz aus Abbildung 4.4 auf Seite 27 geschieht dies, indem die Empfangs- und die Sendetransition einer Signalroute zusammenfallen (T2) und das weitergeleitete Signal nicht zwischengespeichert wird. Die Empfangs- und Sendetransition einer Signalroute synchronisiert sich mit den entsprechenden Transitionen von Verbindungen, **output**-Befehlen und Eingabewarteschlangen von Prozessen, so daß sich die Existenz von Signalrouten im synchronisierten Netz kaum bemerkbar macht; die Aktionen, mit denen T2 beschriftet ist, dienen sozusagen als “Kitt” zwischen Prozessen und Verbindungen bzw. Kanälen.

Die Schaltbedingung von `T2` enthält Bedingungen wie “$sig \in SIG(R)$”. Der Compiler umschreibt diese und andere Bedingungen äquivalent mit “$sig = a_1 \vee \ldots \vee sig = a_n$”, falls $SIG(R) = \{a_1, \ldots, a_n\}$.

Wie auch bei Kanälen berücksichtigt der Compiler bidirektionale Signalrouten, die wie zwei undirektionale Routen mit Indizes 0 und 1 behandelt werden.

Die Ausgabe einer Signalroute geht entweder an einen Prozeß oder an die Umgebung eines Blocks, d.h. durch eine Verbindung (connection) an einen Kanal. Um das Ziel einer solchen Signalroute richtig zu adressieren, sind spezielle Vorkehrungen notwendig, die im nächsten Abschnitt dargelegt sind.

5.5 Bemerkungen zu Verbindungen

Die Behandlung von Verbindungen stellt eine Erweiterung der aus [FG98a] bekannten Semantik dar. Sie wird daher etwas ausführlicher als die anderer Sprachelemente diskutiert.

Interprozeßkommunikation findet in SDL über Kanäle und Signalrouten statt. Kanäle verbinden Blöcke untereinander, Signalrouten verbinden Prozesse untereinander oder mit der Umgebung des Blocks. Führen mehrere Kanäle an einen Block, innerhalb dessen sich mehrere Signalrouten befinden, so ist zunächst nicht ersichtlich, welchen Weg Signale nehmen sollen, die in einen Block hinein- oder aus ihm herausgehen. Diese Information wird durch Verbindungen (connections) gegeben. Sie stellen sozusagen Schnittstellen an den Rändern eines Blocks dar und legen fest, welche Kanäle mit welchen Signalrouten kommunizieren. Im Beispiel aus Kapitel 2 wird festgelegt, daß Signale, die von Prozeß `one` über die Signalroute `r1` hinausgehen, durch den Kanal `c1` weitertransportiert werden sollen. Entsprechend werden Signale, die über den Kanal `c2` in den Block gelangen, durch die Signalroute `r2` an `one` weitergereicht. Wie sich am Beispiel des Blocks `bar` zeigt, können Verbindungen Signale in beide Richtungen weiterleiten.

Die Semantik einer Verbindung muß also aus zwei Teilen bestehen: einem Teil, der Signale in den Block hineinleitet, und einem Teil, der Signale herausleitet. (Falls Signale nur in eine Richtung fließen, wird der nicht benötigte Teil vom Compiler weggelassen.) Verbindungen sollen die Übertragung von Signalen nicht verzögern; es macht also Sinn, sie ähnlich (bidirektionalen) Signalrouten zu übersetzen. Allerdings können Verbindungen ihre Ausgabe verzweigen, was für jeden Empfänger eine Sendetransition erfordert.[1)]

In Petri-Netzen werden Aktionssymbole benutzt, um die Kommunikation zwischen den Komponenten des Netzes zu steuern. Es stellt sich daher die Frage, welche Aktionssymbole für Verbindungen benutzt werden sollen, um die Kommunikation auf SDL-Ebene korrekt abzubilden. Die Aktionssymbole müssen die Verbindung eindeutig identifizieren, damit Signale nicht fehlgeleitet werden, und es muß eine Unterscheidung getroffen werden zwischen den Signalen, die in den Block eintreten, und denen,

[1)] Es wäre falsch, nur eine Sendetransition zu benutzen, die die Aktionsnamen aller Empfänger trägt, da dann eintreffende Signale dupliziert würden.

die ihn verlassen. Die Namen der Aktionssymbole setzen sich daher zusammen aus dem Namen des Blocks, einem Zähler, der verschiedene Verbindungen innerhalb eines Blocks auseinanderhalten hilft, und einem Index *out* (für herausgehende Signale) und *in* (für eintreffende Signale). Abbildung 5.3 zeigt, wie dies in unserem Beispiel aussähe.

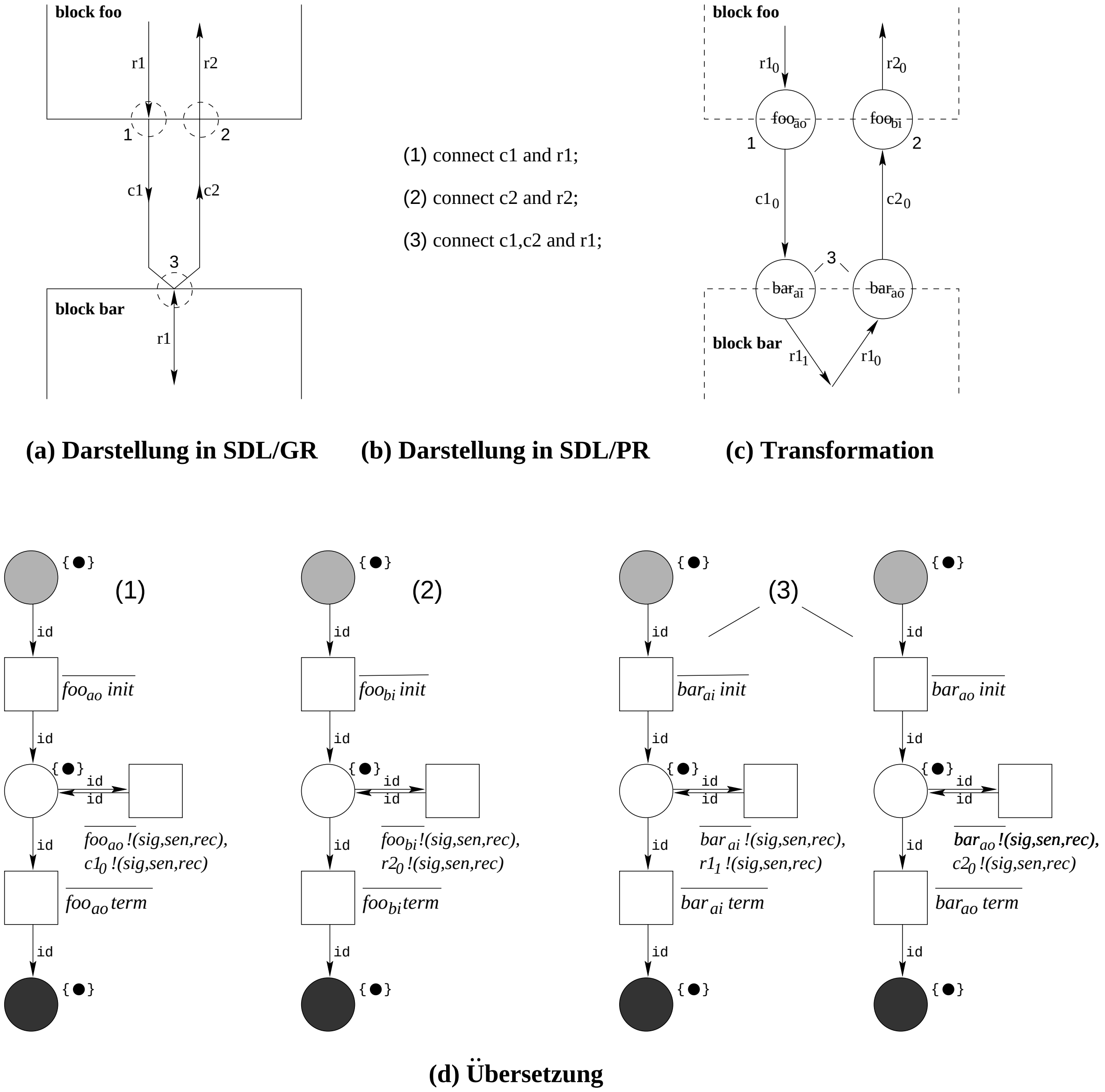

Abbildung 5.3: Übersetzung von **connect** anhand des Beispielprogramms

Signalrouten, die aus einem Block herausführen, schreiben also an den mit *out* indizierten Teil der Verbindung; Kanäle, die in den Block hineinführen, schreiben an den mit *in* indizierten Teil. Daher kann über das mit *out* indizierte Aktionssymbol bereits auf Blockebene synchronisiert werden, während dies für das *in*-Aktionssymbol erst auf Systemebene geschehen kann (vergleiche Abschnitt 4.2.6 im vorangegangenen Kapitel).

5.6 Bemerkungen zu Prozessen

Prozesse spielen als die “handelnden” Einheiten in SDL eine zentrale Rolle. Um sie korrekt in M-Netze zu übersetzen, gilt es vieles zu beachten. Das in Abbildung 4.8 auf Seite 34 gezeigte Netz ist dementsprechend komplex. Für eine ausführliche Beschreibung seiner Funktionsweise (und die der im folgenden vorgestellten Alternativen) wird auf [Gra98b] verwiesen; an dieser Stelle seien nur die wichtigsten Punkte erwähnt und die Änderungen erläutert, die sich gegenüber [Gra98b] durch die praktische Erprobung der Netze ergaben und zum Teil bereits in [FG98b] berücksichtigt wurden.

Im Netz $M_{process}$ aus Abbildung 4.8 lassen sich drei Abschnitte mit unterschiedlichen Funktionen ausmachen. Der Hauptteil mit den Transitionen T1 bis T4 und R_Π links, der Variablenteil mit den Transitionen T7 bis T11' in der Mitte und die Eingabewarteschlange rechts davon.

Hauptteil

Der Hauptteil dient der Erledigung von “Verwaltungsaufgaben”. Er enthält Transitionen zur Initialisierung (T1), zur Terminierung (T2, T2') und einen Stack zur Verwaltung der freien Prozeß-Identifikatoren (P3 und P4). Beim Erzeugen einer neuen Prozeßinstanz durch Schalten von T3 wird eine zuvor verfügbare ID aus dem Stack abgezogen, die beim Beenden der Instanz über T4 wieder zurückgegeben wird. Die Makro-Transition R_Π wird durch das Netz für den Rumpf des Prozesses ersetzt. Transition T3'' schaltet nur, falls die maximale Anzahl von Prozeßinstanzen bereits existiert und eine weitere (per **create**-Befehl) erzeugt werden soll. Gemäß Z.100 wird in solchen Fällen keine Instanz erzeugt und stattdessen der Wert **Null** geliefert, der hier mit $\bullet$ repräsentiert ist.

Im Vergleich zum Netz in [FG98a] fallen drei Unterschiede ins Auge:

- Der “Schwanz”, bestehend aus den Stellen P41 bis P46 und den Transitionen T41 bis T46. Jede der Transitionen zieht bei der Beendigung einer Prozeßinstanz Marken von den Stellen des Variablen- oder des Warteschlangenteils ab. Bei der parallelen Terminierung aus [FG98a] wanderten stattdessen alle Marken zur Transition T4.

- Eine spezielle Schaltbedingung bei T46, die die LL-Entfaltung dieser Transition verkleinert, indem die Zahl der möglichen Kombinationen der sig_i und sen_i verringert wird. Befinden sich bei Terminierung einer Prozeßinstanz noch k Signale in der Eingabewarteschlange, so sind genau k Signal-Einträge in P14 ungleich $\bullet$. Durch die Existenz der Shift-Transition T15 kann stets erreicht werden, daß diese Einträge sich in den letzten k Stellen der Schlange befinden, so daß sichergestellt ist, daß T46 nach einer eventuell notwendigen Umordnung des Puffers schalten kann.

- Die andersgeartete Terminierung des Prozeßnetzes. Bei der in [FG98a] vorgesehenen Variante werden die Markierungen von P3 *und* P4 durch T2 absorbiert. Bei der Entfaltung in LL-Netze sind allerdings sehr viele verschiedene Bindungen für die x_i möglich, und zwar n^n (falls $n = MAX(\Pi)$). Hinzu kommt, daß die Terminierungstransitionen aller Prozeßnetze im globalen Kontrollfluß zusammenfallen (siehe Abschnitt 4.2.1), d.h., die Anzahl der möglichen Bindungen multipliziert sich! Bei nur drei Prozessen mit je drei IDs ergibt die Entfaltung der Prozeßterminierungstransition dann $3^3 \cdot 3^3 \cdot 3^3 = 3^9 = 19683$ LL-Transitionen. Die hier gewählte Variante entschärft das Problem, indem die Marken von P3 und P4 in getrennten Schritten abgezogen werden; so ist immer noch gewährleistet, daß die parallele Prozeßterminierung erst dann stattfindet, wenn alle Prozeßinstanzen im System aufgehört haben zu existieren, aber die Anzahl der Bindungen addiert sich nunmehr, und zwar im zuvor genannten Beispiel zu $1 + 3^3 + 3^3 + 3^3 = 82$.

Tabelle 5.2 zeigt den Einfluß der drei Optimierungen auf die Größe der Übersetzung anhand des Protokollbeispiels aus Kapitel 2.

		Prozeßinstanzen wie im Beispiel		Jeweils drei Prozeßinstanzen	
		Standard	ohne Schwanz	Standard	ohne Schwanz
LL-Netz:	Stellen	407	369	907	871
	Transitionen	775	1617	3536	20294
Minimiertes LL-Netz:	Stellen	278	260	362	338
	Transitionen	190	172	280	268
Entfaltung:	Cutoff-Ereignisse	11	11	36	36
	Konfigurationen	12	12	40	40

		Kapazität 1		Kapazität 3	
		Standard	ohne Bed.	Standard	ohne Bed.
LL-Netz:	Stellen	407	407	491	491
	Transitionen	775	832	2251	28678
Minimiertes LL-Netz:	Stellen	278	278	306	306
	Transitionen	190	190	212	212
Entfaltung:	Cutoff-Ereignisse	11	11	11	11
	Konfigurationen	12	12	12	12

		Prozeßinstanzen wie im Beispiel		Jeweils drei Prozeßinstanzen	
		Standard	andere Term.	Standard	andere Term.
LL-Netz:	Stellen	407	405	907	905
	Transitionen	775	773	3536	4210
Minimiertes LL-Netz:	Stellen	278	276	362	360
	Transitionen	190	188	280	278
Entfaltung:	Cutoff-Ereignisse	11	11	36	31
	Konfigurationen	12	12	40	35

Tabelle 5.2: Auswirkungen der Änderungen am Prozeßnetz

In der ersten Tabelle sind die Ergebnisse bei Verwendung des Standard-Netzes (siehe Abbildung 4.8) den Ergebnissen gegenüberstellt, die man erhält, falls alle Marken direkt zu T4 wandern. Angegeben sind jeweils die Kenngrößen des Gesamtnetzes, und zwar einmal mit zwei Instanzen von `one` und einer Instanz von `two` (wie im Beispiel) und einmal mit je drei Prozeßinstanzen. Man erkennt, daß die Standard-Variante zwar etwas größere minimierte Netze liefert, dafür aber bei weitem nicht so große unminimierte Netze erzeugt. Die Größe des anderen Netzes (mehr als 20000 Transitionen!) stellt die Tools von PEP vor erhebliche Resourcenprobleme (Speicher, Zeit).

In der zweiten Tabelle sind die Ergebnisse mit und ohne die Schaltbedingung von T46 dargestellt, wobei die Kapazität der Eingabewarteschlange zwischen 1 und 3 variiert wurde. Da im verwendeten Beispiel die Warteschlangen zum Zeitpunkt der Terminierung eines Prozesses stets leer sind, wirkt sich das Fehlen der Bedingung (hier) nicht auf das minimierte Netz aus, allerdings treten ähnliche Resourcenprobleme wie oben auf.

In der dritten Tabelle schließlich wurde neben der Standard-Variante diejenige Variante erprobt, bei der (wie in [FG98a]) die Marken von P4 an T2 (anstelle von T2') wandern. Wiederum sorgt die Standard-Variante für eine (sehr kleine) Vergrößerung des minimierten Netzes, während ein kleineres unminimiertes Netz erzeugt wird, wenn auch der Unterschied nicht die Ausmaße hat, wie dies bei den beiden anderen Änderungen der Fall war.

Der Hauptteil des Prozeßnetzes kann (wie in [FG98a] vorgeschlagen) durch ein alternatives Teilnetz ersetzt werden, welches in Abbildung 5.4 dargestellt ist. Dieses Netz deckt den Fall unendlich vieler Prozeß-IDs ab, wofür es mit endlichen Kantenbeschriftungen auskommt (allerdings unterstützt der Compiler keine Prozesse mit unendlich vielen Instanzen, s.u.). Der Stack wird um eine Stelle (P5) erweitert, in der die maximale Anzahl bisher gleichzeitig aktiver Instanzen gezählt wird. Die ID-Tokens werden erst nach und nach bei Bedarf erzeugt, zu Beginn der Simulation ist die Stelle P4 leer. Die Transitionen T3 und T3' erzeugen neue Prozeßinstanzen. Dabei schaltet T3, falls P4 nicht leer ist; in dem Falle wird eine ID vom Stack genommen und die Marke auf P3 um eins erhöht, während sich der Inhalt von P5 nicht ändert. T3' schaltet, wenn mehr Instanzen als je zuvor während der Simulation erzeugt werden sollen. Dabei werden P3 *und* P5 erhöht und eine neue ID erzeugt, die nach Beendigung der Instanz nach P4 gelangt.

Auch hier wurde das ursprüngliche Netz um die sequentielle Terminierung der impliziten Variablen und die spezielle Schaltbedingung der Eingabewarteschlange erweitert, wohingegen die Terminierung des Prozeßnetzes bereits effizient genug war.

Die Verwendung des alternativen Netzes kann im Compiler durch den Schalter `-BP1` erreicht werden. Tabelle 5.3 stellt die Ergebnisse (bezogen auf die Übersetzung des Beispiels aus Kapitel 2) gegenüber. Es zeigt sich, daß die vorgestellte Alternative zwar deutlich kleinere Netze liefert, sobald man die erhaltenen Netze minimiert, daß sich aber die Entfaltung erheblich vergrößert. Weitere Tests müssen zeigen, welche Variante im allgemeinen besser ist.

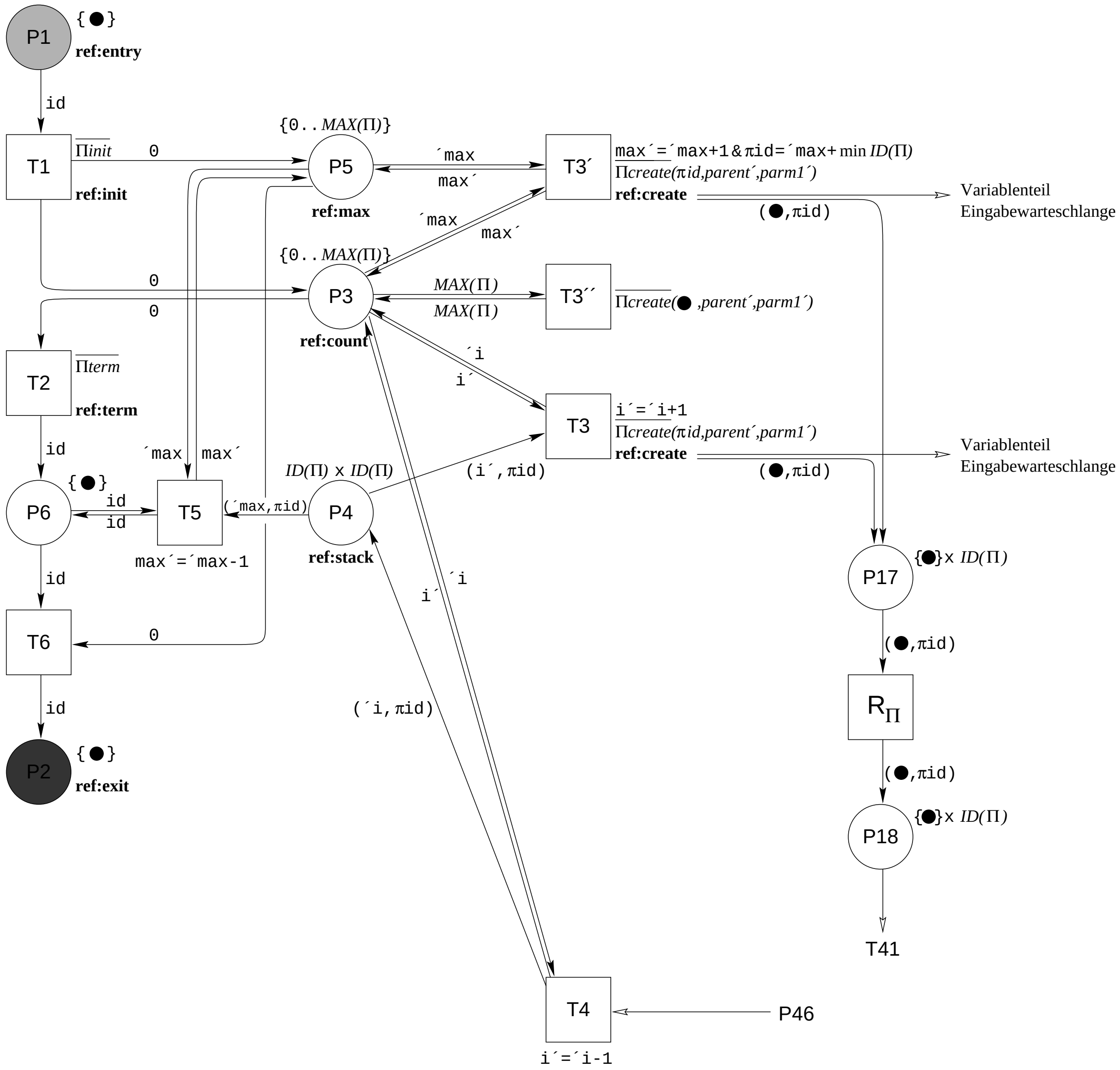

Abbildung 5.4: Alternative für den Hauptteil des Prozeßnetzes aus Abb. 4.8

Variablenteil

Der Variablenteil enthält Wertestellen und Zugriffstransitionen für

- die impliziten Variablen **self**, **sender**, **parent** und **offspring** (P7 bis P10);
- den aktuellen Zustand (P11), der während der Ausführung von Zustandsübergängen den Wert • annimmt; der Wert 0 wird benutzt um auszudrücken, daß der Prozeß terminieren wird;
- den Zustand, in dem sich der Prozeß zuletzt befunden hat (P11'), um die Anweisung **nextstate** – korrekt übersetzen zu können;
- jeden formalen Parameter des Prozesses, im Beispiel aus Abbildung 4.8 genau einen (P51).

		Prozeßinstanzen wie im Beispiel		Jeweils drei Prozeßinstanzen	
		`-BP0`	`-BP1`	`-BP0`	`-BP1`
LL-Netz:	Stellen	407	412	907	915
	Transitionen	775	781	3536	3489
Minimiertes LL-Netz:	Stellen	278	205	362	273
	Transitionen	190	109	280	171
Entfaltung:	Cutoff-Ereignisse	11	8	36	95
	Konfigurationen	12	14	40	190

Tabelle 5.3: Vergleich der Varianten für den Hauptteil

Bei der Erzeugung eines Prozesses durch Schalten von Transition T3 werden die Wertestellen initialisiert, wobei die impliziten Variablen mit den Inhalten belegt werden, die durch die Z.100 vorgesehen sind. Der anfängliche Zustand ist der Zwischenzustand $\bullet$, da zunächst die Start-Transition ausgeführt wird. Bei Beendigung eines Prozesses werden die Werte von diesen Stellen schrittweise abgezogen.

Auffällig ist, daß die Variable **offspring** nicht eine, sondern drei Zugriffstransitionen besitzt: Eine Lese-/Schreibtransition (T9), eine Nur-Lese-Transition (T9') und eine Nur-Schreib-Transition (T9''). Dies dient der besseren Übersetzung von **create**-Befehlen; siehe Abschnitt 5.13.

Eingabewarteschlange

Das Teilnetz für die Eingabewarteschlange (siehe Abbildung 4.8 auf Seite 34) ähnelt dem für Kanäle – in der Tat funktionieren beide Konstrukte ja auch als FIFO-Puffer. Stelle P13 hält für jede Instanz die Anzahl der in der Eingabe befindlichen Signale fest, die in P14 gespeichert sind. Der Typ von P14 ist ein 4-Tupel, jede Marke speichert die Stelle innerhalb des Puffers, das Signal selbst, die ID des Senders und für welche Instanz das Signal bestimmt ist. Eingaben in die Warteschlange erfolgen durch T13, das Auslesen durch **input**-Statements per T14; dabei wird automatisch der Inhalt von **sender** angepaßt. T15 ist die aus Kanälen bekannte Shift-Transition.

Auch für die Warteschlange ist ein alternative Lösung denkbar. Abbildung 5.5 zeigt ein Teilnetz, das anstelle des in Abbildung 4.8 für die Warteschlange benutzten Teils eingesetzt werden kann. Das neue Teilnetz ist über T3, T4 und P45 an den Hauptteil angebunden und kann (theoretisch) einen unendlich großen Eingabepuffer verwalten, wobei nur endliche Kantenbeschriftungen benutzt werden. P14 enthält einen Eintrag mehr als die gewünschte Kapazität, da stets ein leerer Eintrag (mit Signal = $\bullet$) gehalten wird, um das Vorrücken der Einträge im Puffer zu kontrollieren. Transition T16 schaltet dann, wenn sich eine Prozeßinstanz beendet hat, und ermöglicht so das schrittweise Leeren der Speicherstelle P14 durch T17. Durch diese Technik erübrigt sich das Einfügen einer speziellen Schaltbedingung, um die Anzahl der LL-Transitionen zu minimieren.

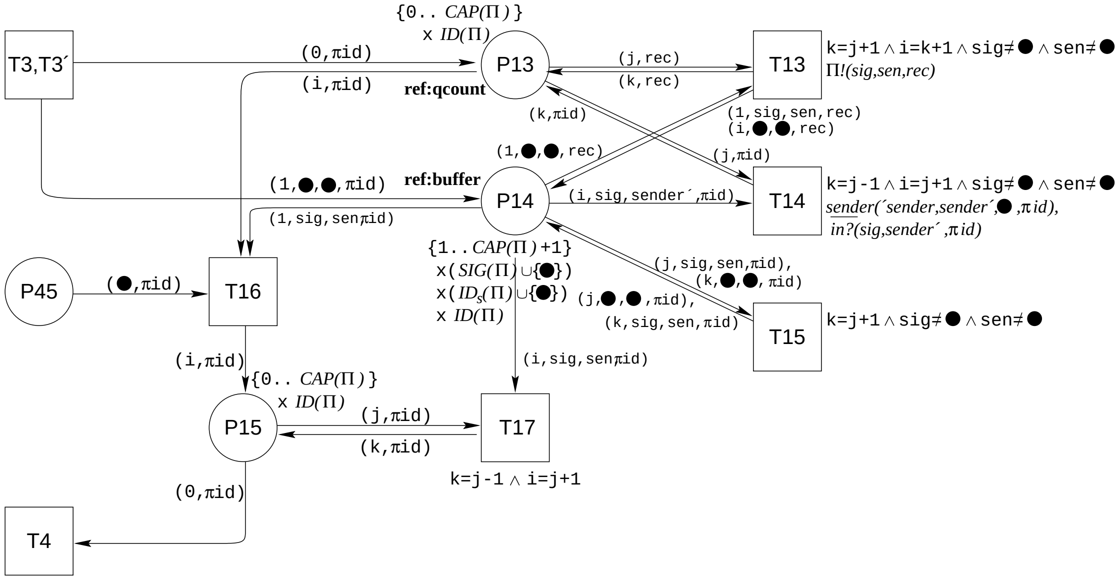

Abbildung 5.5: Alternative für die Warteschlange von Prozessen

Die Verwendung des alternativen Warteschlangen-Netzes läßt sich durch die Compiler-Option `-BP2` steuern; sollen die Alternativen für Hauptteil und Warteschlange gleichzeitig benutzt werden, so geschieht dies über die Option `-BP3`.

Vergleiche zwischen den Varianten sind in Tabelle 5.4 aufgelistet. Die Alternative aus Abbildung 5.5 liefert geringfügig größere Netze und gleichgroße Entfaltungen. Obwohl diese Alternative daher zunächst weniger interessant erscheint, könnte sie von Vorteil sein, falls irgendwann einmal Verifikationsmethoden für HL-Netze entwickelt werden.

		Prozeßinstanzen wie im Beispiel		Jeweils drei Prozeßinstanzen	
		`-BP0`	`-BP2`	`-BP0`	`-BP2`
LL-Netz:	Stellen	407	452	907	1051
	Transitionen	775	889	3536	3926
Minimiertes LL-Netz:	Stellen	278	295	362	388
	Transitionen	190	201	280	298
Entfaltung:	Cutoff-Ereignisse	11	11	36	36
	Konfigurationen	12	12	40	40

Tabelle 5.4: Vergleich der Varianten für die Eingabewarteschlange

Unterschiede zur Z.100

Einige Aspekte konnten bei der Modellierung von Prozessen nicht so realisiert werden, wie es in der Z.100 vorgesehen ist. So läßt die Z.100 unbegrenzte Anzahlen von Instanzen zu, und auch die Kapazität der Eingabewarteschlange von Prozessen ist unbegrenzt. Im Rahmen von PEP ist es jedoch nicht möglich, unendliche Objekte in M-Netzen zu repräsentieren (auch wenn die Theorie dies leistet), für die Anzahl von Prozeßinstanzen und die Kapazität der Warteschlangen muß also eine endliche Grenze gezogen werden. Die Möglichkeiten, die dabei verwendeten Größen zu bestimmen, sind in Kapitel 6 dargestellt. Gibt es eine endliche Grenze für die maximale Anzahl von Instanzen oder Signalen in der Warteschlange, so kann dies (analog zum Verfahren bei Kanälen) mit Hilfe des Modelcheckers festgestellt werden, indem bei einer gegebenen fixen Anzahl von Instanzen/Signalen angefragt wird, ob diese Anzahl jemals erreicht wird, siehe Kapitel 7.

Auf einen anderen Schwachpunkt der Implementierung, die ihre Ursache jedoch nicht direkt in der Modellierung der Prozesse hat, ist bereits im Zusammenhang mit Kanälen hingewiesen worden: Durch die Wiederverwendung von IDs können Prozeßinstanzen Signale erhalten, die nicht für sie bestimmt waren.

5.7 Bemerkungen zu Prozeduren

Ähnlich wie bei Prozessen muß die Semantik auch bei Prozeduren mehrere Instanzen ermöglichen und dafür sorgen, daß diese unterschieden werden können. Mehrere Instanzen einer Prozedur können gleichzeitig existieren, falls sich Prozeduren rekursiv aufrufen oder zwei Prozeßinstanzen zugleich dieselbe Prozedur benutzen.

Das Prozedur-Netz $M_{procedure}$ (siehe Abbildung 4.9 auf Seite 37) benutzt zur Verwaltung von Prozedur-IDs einen Stack, und zwar nach dem gleichen Prinzip wie die unbeschränkte Variante des Prozeßnetzes (Abbildung 5.4. Prozedur-IDs werden dabei keinesfalls getrennt nach Prozeßinstanzen verwaltet, vielmehr wird bei jedem Prozeduraufruf unabhängig von der Prozeß-ID eine beliebige gerade verfügbare Prozedur-ID vergeben. Prozedurinstanzen werden durch T3 oder T3′ erzeugt, wobei T3′ den Fall abdeckt, daß der Vorrat an bereits vergebenen IDs in P4 leer ist. Der Fall, daß keine IDs mehr zu vergeben sind, wird nicht extra abgefangen (s.u.). Durch das zweite Argument im Aktionssymbol P_{call} wird die ID der aufrufenden Prozeßinstanz an den Rumpf übergeben und so der Zugriff auf implizite Variablen sowie das Versenden von Signalen mit korrektem Absender ermöglicht. Ferner werden durch P_{call} Werte für alle **in**-Parameter übermittelt, mit denen deren Wertestellen sofort initialisiert werden. (Variablen werden im Rumpf initialisiert.) Der Rumpf der Prozedur wird anstelle der Makro-Transition R_P eingesetzt.

Gegenüber [FG98a] wurde das Netz um Stellen und Zugriffstransitionen für Zustände erweitert, d.h., Prozeduren haben eine eigene Zustandsmenge. Dies erhöht nicht nur die Konformität zu Z.100, sondern ist auch sinnvoll, weil Prozeduraufrufe sonst den Zustand (insbesondere den alten, in T13′ gespeicherten) Zustand des Prozesses durcheinander brächten.

Auch bei Prozeduren gab es Hindernisse, die einer völligen Kompatibilität zum Z.100-Standard im Weg standen. So war es notwendig, die maximale Anzahl gleichzeitig aktiver Prozedurinstanzen zu begrenzen (siehe Kapitel 6). Da es eine solche Beschränkung in der Z.100 nicht gibt, ist dort auch nicht vereinbart, was geschehen soll, wenn versucht wird, über die maximale Anzahl von Instanzen hinaus eine weitere zu erzeugen. In der vorliegenden Lösung gibt es daher kein Äquivalent zu T3″, wie es im Prozeßnetz vorhanden ist. Ist die maximale Anzahl von Prozedurinstanzen erreicht, so blockieren weitere Aufrufe so lange, bis eine der aktiven Instanzen beendet wird und seine ID freigibt. Wie bei Kanälen und Prozessen beschrieben kann eine endliche Grenze für die maximale Anzahl von Instanzen mit Hilfe des Modelcheckers gewonnen werden.

Prozeduren als Funktionen mit Rückgabewert wurden nicht implementiert. Funktionen können aber äquivalent durch normale Prozeduren mit einem zusätzlichen **in/out**-Parameter umschrieben werden, so daß durch diesen Mangel grundsätzlich keine in SDL vorhandene Funktionalität verlorengeht.

Prozeduren mit Unterprozeduren werden zwar unterstützt, aber da im Kontrollfluß nur die IDs der (Unter-)Prozedur und des Prozesses weitergereicht werden, ist es bei verschachtelten Prozeduren nicht möglich, korrekt auf Variablen der einschließenden Prozedur zuzugreifen. Ein ähnliches Problem würde auch im B(PN)^2-Compiler auftreten, wenn dort die Verschachtelung von Prozeduren erlaubt wäre.

5.8 Bemerkungen zu Variablen

Das Variablennetz aus Abbildung 4.10 (siehe Seite 38) ist recht einfach und besteht aus vier Teilen:

- der Initialisierung, die durch Schalten von T1 erfolgt;
- der Stelle P2, die den Wert der Variablen hält;
- der Zugriffstransition T2;
- und dem Terminierungsteil mit Transition T3.

T1 und T3 synchronisieren sich mit den entsprechenden Transitionen zu Beginn und am Ende des Rumpfes des jeweiligen Prozesses bzw. der Prozedur (vergleiche die Semantik von Prozessen in Abschnitt 4.2.7). Bei der Initialisierung wird P2 mit einem Wert belegt. Dieser Wert ist der Standardwert, der bei der Deklaration der Variablen oder der Deklaration ihres Typs vereinbart wurde. Ist in beiden kein Standardwert vereinbart, so wird das Minimum des Wertebereichs zur Initialisierung verwendet. Dies vermeidet eine nichtdeterministische Initialisierung und die damit verbundene erhöhte Anzahl von LL-Transitionen, kann aber dazu führen, daß die Verwendung uninitialisierter Variablen in einer SDL-Spezifikation unentdeckt bleibt. Nichtdeterministische Initialisierung könnte allerdings sehr einfach durch Weglassen der Schaltbedingung $X' = INIT(X)$ im Netz $M_{varinit}$ erreicht werden, wenn dies gewünscht ist.

Variablen gehören stets zu einer Prozeß- oder einer Prozedurinstanz, d.h., bei Erzeugung einer Prozeß- bzw. Prozedurinstanz muß jedesmal eine zugehörige Variableninstanz eingerichtet werden. Die Verwaltung unterschiedlicher Variableninstanzen wird durch die Erweiterung des Stellentyps um $ID(P) \times ID(\Pi)$ ermöglicht. Dabei ist $ID(P) = \{\bullet\}$, falls die Variable auf Prozeßebene deklariert ist. Die spezielle Schaltbedingung von T2 sorgt dafür, daß auch Prozeduren auf Variablen der sie aufrufenden Prozeßinstanz zugreifen können.

Zum besseren Verständnis der Funktionsweise sei noch angemerkt, daß die Initialisierung und Terminierung von Variablen *nicht* im globalen Kontrollfluß stattfindet, sondern im Kontrollfluß des Prozeß- respektive Prozedurrumpfs. Im Gegensatz zu den entsprechenden Transitionen von Kanälen, Prozessen, Prozeduren etc. schalten die Initialisierungs- und die Terminierungstransition also nicht bloß einmal während der Simulation, sondern *jedesmal*, wenn eine Variableninstanz erzeugt wird.

Die Synchronisation über Initialisierungs- und Terminierungsaktionen findet statt, sobald das Netz des zugehörigen Rumpfes zusammengesetzt wurde, da dort die Gegenstücke der Aktionen im Kontrollfluß zu finden sind. Synchronisation über Zugriffsaktionen findet erst statt, sobald das Netz um die Prozeduren/Unterprozeduren ergänzt wurde, damit auch (Unter-)Prozeduren korrekt auf Variablen zugreifen können, die auf Prozeß-Ebene deklariert wurden (vergleiche formale Semantik in Abschnitt 4.2.7). Gleichzeitig wird durch die Reihenfolge der Synchronisation(en) gewährleistet, daß lokale Variablendeklarationen gleichnamige Deklarationen auf höherer Ebene korrekt überdecken.

5.9 Bemerkungen zum Rumpf

Der Rumpf eines Prozesses bzw. einer Prozedur wird durch eine Iteration übersetzt. Dabei wird zunächst die Start-Transition ausgeführt. Im iterierten Teil wird einer der übrigen Zustandsübergänge ausgewählt; der Endteil wird erst dann ausgeführt, wenn der Prozeß den Zustand 0 erreicht, d.h. terminieren will.

Für den Compiler wurde das bei [Gra98b] definierte Iterationskonstrukt leicht abgewandelt, um eine etwas effizientere Übersetzung zu ermöglichen. Die Änderung besteht darin, das Netz N_{silent} wegzulassen, das vom Ende des iterierten Teils wieder an den Start des iterierten Teils bzw. des Endteils führt. Stattdessen werden die Exit-Stellen des iterierten Teils direkt mit den eigenen Entry-Stellen, den Exit-Stellen des Starts und den Entry-Stellen des Endteils verknüpft, d.h., die abgewandelte Form des Iterationskonstrukts, wie es im Compiler an dieser Stelle verwendet wird, lautet:

$$[N_1 * N_2 * N_3] = (1N_1 \cup 2N_2 \cup 3N_3) \oplus \otimes\{(1N_1)^\bullet, {}^\bullet(2N_2), (2N_2)^\bullet, {}^\bullet(3N_3)\}$$

Im Gegensatz zu allgemeinen Netzen ist es bei SDL möglich, N_{silent} wegzulassen, weil keines der Netze, auf die der veränderte Operator angewandt wird, Parallelismus enthält. Abbildung 5.6 zeigt das Zusammenwirken von N_1, N_2 und N_3 anhand einer schematischen Darstellung.

5.10 Bemerkungen zur Kodierung von Signalen

Das Prinzip, eine Abbildung von Signalen auf natürlichen Zahlen herzustellen, um erstere im Netz zu repräsentieren, wurde bereits in Abschnitt 4.1.2 im vorangegangenen Kapitel angedeutet. Signalen ohne Argumente wird je eine einzelne, eindeutige Zahl zugewiesen. Soll im SDL-Programm ein solches Signal verschickt werden, so

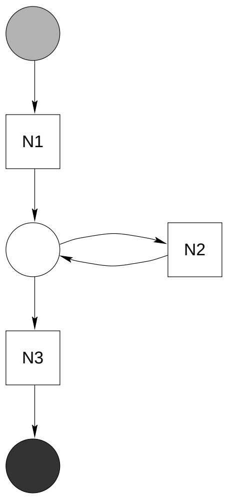

Abbildung 5.6: Verändertes Iterationskonstrukt $[N_1 * N_2 * N_3]$

wird im Petri-Netz diese Zahl in den Wertestellen von Kanälen und Warteschlangen gespeichert. Signalen mit Argumenten wird je eine Menge von Zahlen zugewiesen, die groß genug ist, jede Kombination von Argumenten durch eine eigene Zahl zu repräsentieren.

Wird also ein Signal mit Argumenten verschickt, so muß aus den verwendeten tatsächlichen Parametern die Zahl ermittelt werden, die deren Kombination im Netz repräsentiert. Umgekehrt müssen, wenn ein Signal per **input** eingelesen wird, aus der erhaltenen Zahl die Argumente zurückgewonnen werden. Beide Verfahren müssen selbstverständlich aufeinander abgestimmt sein, damit ein Signal auch korrekt dekodiert wird.

Für die Kodierung wird folgendes Verfahren benutzt: Die Zahlenmenge, die zur Repräsentation eines Signals s mit Argumenten verwendet wird, ist kontinuierlich, d.h., sie hat die Form $\{ i \in \mathbb{N} \mid i_0(s) \leq i \leq i_1(s) \}$ für bestimmte $i_0(s), i_1(s) \in \mathbb{N}$. Angenommen, das Signal s habe als Argumente Typen mit den Wertemengen $s_1, \ldots, s_n$, deren Mächtigkeiten $m_i = |s_i|,\ 1 \leq i \leq n$ sei. Es bezeichne

$$p_i(x_i) = m_i - \Big| \{ x \in s_i \mid x \geq x_i \} \Big|$$

die "Position" des Werts x_i in der Menge s_i. Ist dann $x_i \in s_i$ für alle $i = 1, \ldots, n$, so wird das Signal $s(x_1, \ldots, x_n)$ repräsentiert durch die Zahl

$$f_S(s(x_1, \ldots, x_n)) = i_0(s) + \sum_{i=1}^{n} \left(p_i(x_i) \cdot \prod_{j=1}^{i-1} m_j \right)$$

Anders ausgedrückt liefert $f_S(s(x_1, \ldots, x_n))$ die Zahl, die man erhält, wenn man $p_n(x_n) \ldots p_1(x_1)$ als Zahl in einem Zahlensystem auffaßt, dessen i-te Stelle jeweils mit m_i gewichtet ist, zuzüglich eines "Offsets" $i_0(s)$.

Die *Ins*-Funktionen in der formalen Semantik von **input** bzw. **output** kodieren/dekodieren Signale auf der Basis genau dieser Zuordnung. Für die speziellen Datentypen **Boolean** und **PId** wird für die Berechnung von p_i angenommen, daß **False** $<$ **True** und **Null** $< x,\ \ x \in \mathbb{N}$ sei.

5.11 Bemerkungen zu output

Die formale Semantik des **output**-Befehls, wie sie in Abschnitt 4.2.12 angegeben ist, beschreibt die Kodierung von Signalen, wobei als Argumente sowohl Konstanten als auch Variablen zugelassen sind. Für Variablen wird die Transition mit den benötigten Aktionssymbolen für Variablenzugriffe versehen.

Der vorliegende Compiler versucht zwar, die vollständige Syntax von **output** aus der Z.100 umzusetzen. Es zeigt sich jedoch, daß das Routing von Signalen in SDL mit der zur Zeit verwendeten Semantik nicht in allen Einzelheiten nachgebildet werden kann. Problematisch ist vor allem die Übersetzung der **via**-Klausel. Mit ihr kann ein Kanal oder eine Signalroute bestimmt werden, über die das Signal den Empfänger erreichen soll, was von Nutzen ist, falls die Angabe des Signals und/oder des Empfängers den Weg des Signals nicht eindeutig bestimmt.

Abbildung 5.7 zeigt als Beispiel einen Prozeß, der ein Signal `sig` über zwei unterschiedliche Signalrouten verschicken kann. Ohne Angabe einer **via**-Klausel kann das Signal über beide Routen geschickt werden; es wird nichtdeterministisch eine davon ausgewählt. Im rechten Bildteil ist der Weg des Signals eindeutig bestimmt.

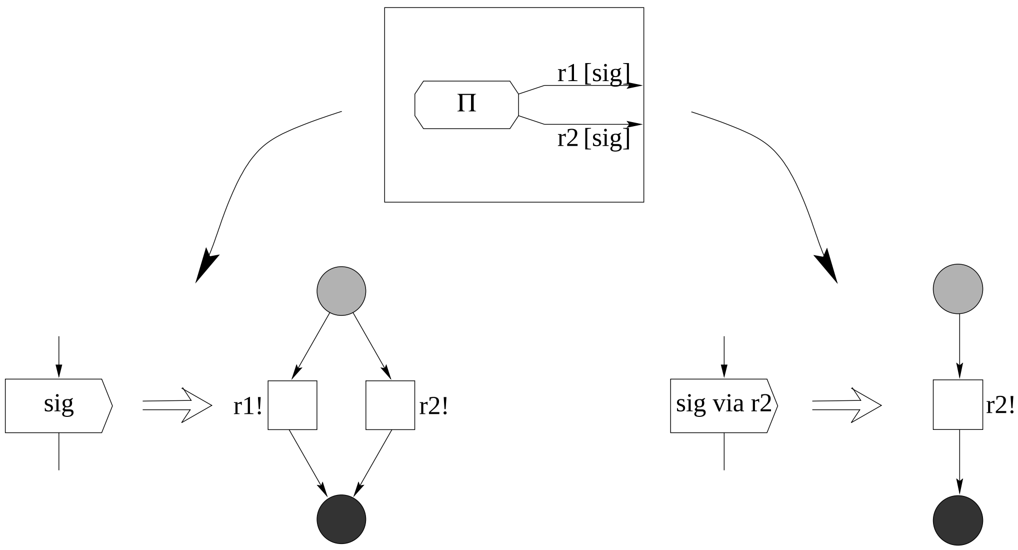

Abbildung 5.7: Korrekte Übersetzung der **via**-Klausel

Problematisch wird die korrekte Übersetzung in einem Fall, wie er in Abbildung 5.8 dargestellt ist. Hier stehen zum Transport eines Signals zwei Kanäle zur Verfügung. Ein Signal wird vom Prozeß über die einzig vorhandene Signalroute `r` abgesetzt, was durch die Übersetzung mit dem Aktionssymbol $r!$ im unteren Bildteil ausgedrückt wird. Erreicht es die Grenze des Blocks, so kann es von einem der beiden Kanäle weitertransportiert werden. Das Netz für die Verbindung zwischen der Signalroute und den Kanälen sorgt dafür, daß Signale, die von `r` kommen, an genau einen Kanal weitergereicht werden; die Auswahl erfolgt von Fall zu Fall, und zwar nichtdeterministisch. Dieses Verhalten ist konform zur Z.100, falls kein Kanal mittels **via** ausgewählt wurde (linker Bildteil). Fordert die SDL-Spezifikation in diesem Fall jedoch die Benutzung eines bestimmten Kanals, so kann dies durch die M-Netz-Semantik nicht gewährleistet werden, da ein Signal nur mit Informationen über die Identität von Sender und Empfänger, nicht aber mit Angaben über den zu verwendenden Weg versehen ist. Der Prozeß in der Abbildung kann Signale nur über die Signalroute `r` versenden; danach wählt das Signal selbständig einen der möglichen Wege zum vorgesehenen Empfänger.

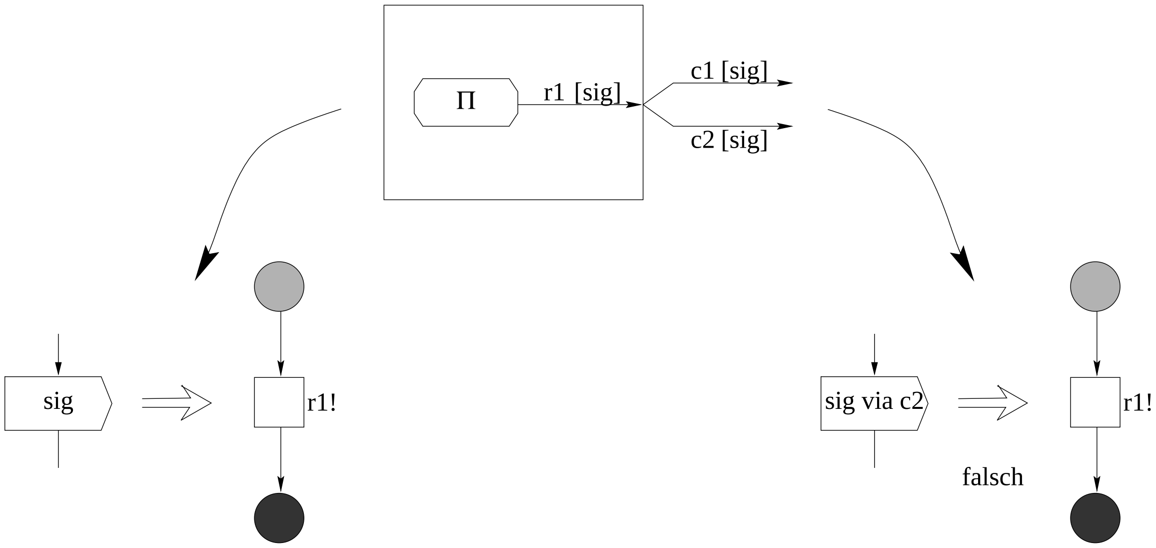

Abbildung 5.8: Links korrekte, rechts falsche Übersetzung von **via**

Dies hat zwei unerwünschte Auswirkungen:

- Signale nehmen möglicherweise nicht den vorgesehenen Weg zum Ziel.
- Werden zwei Signale nacheinander mit derselben **via**-Klausel verschickt, so werden sie möglicherweise über unterschiedliche Kanäle verschickt. Dadurch ist nicht gewährleistet, daß sie in der Reihenfolge am Ziel eintreffen, in der sie abgesandt wurden.

Somit ist der Compiler in seiner jetzigen Form nicht geeignet, Spezifikationen zu übersetzen, in denen Kommunikation mit Redundanz simuliert werden soll.

Ein weiteres Problem ergibt sich, falls mit "**to** `process–ident`" ein Prozeß(typ) als Empfänger eines Signals festgelegt wird. Laut Z.100 soll in diesem Fall eine aktive Instanz von `process–ident` zufällig als Empfänger gewählt werden. Im verwendeten Prozeßnetz gibt es jedoch keine Möglichkeit, von außerhalb des Prozesses festzustellen, wie viele und welche Instanzen gerade aktiv sind. Wie bereits in Abschnitt 5.3 angedeutet, kann das Versenden an eine nicht aktive Prozeß-ID zur Verstopfung des Kanals oder, falls nur Signalrouten im Spiel sind, zum Blockieren des **output**-Befehls führen. Der Compiler warnt daher den Benutzer vor der Verwendung dieses Konstrukts, falls von `process–ident` mehr als eine Instanz existieren kann.

Schließlich sei noch der Hinweis angebracht, daß bei Auslassen von Argumenten in einem **output**-Befehl $p_i(x_i) = 0$ angenommen wird. Dies führt zu einer Reduzierung des LL-Netzes, allerdings gibt dies die im SDL-Standard vorgesehene Undefiniertheit des Argumentwertes nicht wirklich wieder.

5.12 Bemerkungen zu call

Im Falle eines Prozeduraufrufs ohne Parameter wird das Netz M_{call} aus Abbildung 4.25 (Seite 54) verwendet. Es besteht aus drei Teilen: Transition T1 dient dem Prozeduraufruf. Die Ausführung des Kontrollflusses wird angehalten, indem die Kontrollflußmarke in P1 "geparkt" wird. T2 wird erst beim Beenden der Prozedur geschaltet. Die Bedingung $pid \neq proc$ in T2 verhindert im Falle rekursiver Prozeduraufrufe unnötig erzeugte LL-Transitionen – die aufrufende und aufgerufene Prozedurinstanz können schließlich nicht die gleiche ID benutzen.

Kommen **in**-Parameter hinzu, so wird das Aktionssymbol $PROC_{call}$ um je eine Kommunikationsvariable erweitert, so daß diese Parameter im Prozedur-Netz beim Aufruf initialisiert werden (siehe Abschnitt 4.2.8).

Für jeden **in/out**-Parameter kommt ein Stellen-/Transitionspaar als Referenzdurchreiche hinzu (siehe P11, T11 in Abbildung 4.26). P1 kann dann weggelassen werden, da ihre Funktion auch von P11 erfüllt wird. P11 hat außerdem die Funktion, Zugriffe auf den **in/out**-Parameter der Prozedur korrekt zuzuordnen. Dazu enthält die erste Dimension ihres Typs die ID der aufgerufenen Prozedurinstanz. P11 erfüllt *nicht* den Zweck, den Wert des Parameters zu halten – der Zugriff auf dessen Wertestelle wird durch die Beschriftung der Transition T11 geregelt. Da Typ und Inhalt aller zu Referenzdurchreichen gehörigen Stellen gleich sind, läge der Gedanke nahe, nur eine einzige Stelle zu erzeugen, die dann mit mehreren Durchreichetransitionen verbunden wäre. Dann aber wäre es nicht möglich, in einer Anweisung zugleich auf mehrere **in/out**-Parameter zuzugreifen, siehe [Ack97].

5.13 Bemerkungen zu create

Das vom Compiler verwendete Netz M_{create} (Abbildung 4.27) besteht aus zwei Teilen. Transition T1 zieht den vorher gültigen Wert von **offspring** ab, Transition T2 führt die Prozeßerzeugung durch und setzt den Wert von **offspring** neu.

Die Verteilung des Zugriffs auf **offspring** auf zwei Transitionen reduziert die Anzahl der resultierenden LL-Transitionen. Dies ist möglich, da der vorige Wert von **offspring** keinen Einfluß auf die Funktion von **create** hat, aber die Anzahl der möglichen Variablenbindungen erhöhen würde, falls der Zugriff in einem Schritt erfolgte.

Die Schaltbedingung $offspring' \neq \pi id$ reflektiert, daß die erzeugte Instanz nicht mit der erzeugenden identisch sein kann, und verringert ebenfalls die Zahl der LL-Transitionen.

5.14 Bemerkungen zu nextstate

In der in Kapitel 4 vorgestellten Semantik werden Vorrichtungen getroffen, die Anweisung "**nextstate –**" (kehre zurück in vorigen Zustand) zu übersetzen. Zu diesem Zweck wurden die Netze $M_{process}$ und $M_{procedure}$ um eine Wertestelle und eine Zugriffstransition für die "Variable" *dash* erweitert. Bei Beginn jedes Zustandsübergangs wird der alte Zustand von q nach *dash* gerettet, bei Ende jedes Zustandsübergangs zurück auf 0 gesetzt. Das Setzen und Rücksetzen vergrößert die LL-Entfaltung des Netzes, was unnötig ist, wenn das Wiederherstellen des alten Zustands nie benutzt wird. Enthält der Rumpf eines Prozesses bzw. einer Prozedur also nicht den Befehl **nextstate** –, so wird das Netz $M_{nextstate}$ aus Abbildung 4.29 (Seite 57) für den betreffenden Prozeß (bzw. die betreffende Prozedur) durch das einfachere Netz aus Abbildung 5.9 ersetzt. Außerdem wird das Aktionssymbol *dash* aus den Beschriftungen der Netze $M_{inpnoarg}$ und M_{inparg} (Abbildungen 4.12 und 4.13) entfernt.

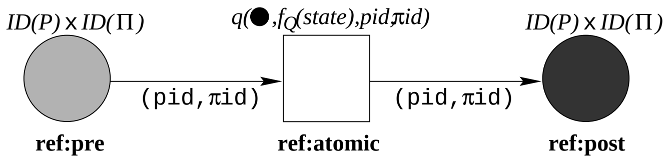

Abbildung 5.9: Alternatives Netz für **nextstate**

Kapitel 6

Der Compiler

Um die in Kapitel 4 vorgestellte M-Netz-Semantik für SDL-Spezifikationen zu implementieren, wurde im Rahmen dieser Arbeit ein Compiler (*s2m*) entwickelt. Bei der Planung des Compilers traten folgende Aspekte in den Vordergrund:

- Da der Sprachumfang von SDL sehr groß ist und daher die Implementierung eines Parsers keine triviale Aufgabe darstellt, sollten moderne Erkenntnisse über Compilerbau genutzt werden, um den Übersetzungsvorgang effizient zu gestalten und keinen unnötigen Aufwand für seine Programmierung zu betreiben. Aus diesem Grunde orientiert sich *s2m* an dem aus [ASU88] bekannten Phasenmodell und teilt die Übersetzung in zwei große Abschnitte, Frontend und Backend genannt, auf.
  Im Frontend wird der Quelltext eingelesen, und der zugehörige Syntaxbaum wird erzeugt. Scanner und Parser des Frontends wurden mit lex/yacc-Dateien realisiert, wobei die semantische Analyse durch die Aktionen der Grammatik-Regeln erfolgt.
  Im Backend wird daraufhin aus dem Syntaxbaum das zugehörige Netz gewonnen. Innerhalb des Backends sind zwei Phasen zu unterscheiden: die Untersuchung der Kommunikationsstrukturen, wie sie weiter unten (Abschnitt 6.3) dargestellt wird, und die eigentliche Netzerzeugung.

- Der Compiler soll die vorgestellte Semantik möglichst exakt nachbilden. Für diesen Zweck ist es von Vorteil, wenn die Arbeitsweise des Compilers dem Aufbau der Semantik entspricht. Da die Semantik kompositionell ist, baut der Compiler das Gesamtnetz aus den Übersetzungen der Einzelteile zusammen, d.h., das Backend arbeitet den vom Frontend erzeugten Syntaxbaum ab. Ein One-Pass-Compiler wäre für SDL ohnehin nicht möglich, da für die Übersetzung bestimmte Aspekte eine Rolle spielen, die sich nur bei Betrachtung des gesamten Netzes analysieren lassen (siehe Abschnitt 6.3).

- Um den Compiler flexibel zu halten, werden wie in [Ack97] Makro-Netze verwendet, d.h., viele Bestandteile der Semantik werden nicht fest im Programm kodiert, sondern als parametrisierte M-Netze in Konfigurationsdateien abgelegt. Dadurch ist es sehr einfach möglich, alternative Semantiken auszuprobieren, siehe die Vorschläge in Kapitel 5.

- Für andere Compiler des PEP-Projekts gab es bereits allgemein gehaltene und erprobte Routinen zum Laden und Speichern von Netzen, Behandlung von Makro-Netzen, für Kompositionsoperatoren u.a. Anstatt das Rad neu zu erfinden, wurden diese Routinen aus dem $B(PN)^2$-Compiler [Ack97] übernommen, d.h., *s2m* ist aus *pn2mnet* hervorgegangen, und zwar wie folgt:
  - Das $B(PN)^2$-Frontend wurde durch ein SDL-Frontend ersetzt.
  - Der $B(PN)^2$-spezifische Teil des Backends wurde durch einen SDL-spezifischen Teil ersetzt.
  - Das übrige Backend wurde übernommen und in einigen Punkten erweitert. Die vorgenommenen Änderungen können wiederum späteren Versionen von *pn2mnet* zugute kommen.

  Die in [Ack97] aufgestellte Vermutung, ein SDL-Compiler lasse sich einfach durch Erstellen eines SDL-Frontends und einiger Konfigurationsdateien aus *pn2mnet* ableiten, unterschätzt jedoch deutlich den Aufwand, den die Anpassung des sprachspezifischen Teils des Backends an SDL mit sich bringt. Dies liegt daran, daß SDL erheblich komplexer als $B(PN)^2$ ist, sich nicht vollkommen kompositionell übersetzen läßt, und vor allem daran, daß die Sprachelemente von SDL keine direkten Entsprechungen in Netzoperatoren haben, wie dies bei $B(PN)^2$ der Fall ist.

6.1 Aufruf und Optionen

s2m verarbeitet Textdateien, die Spezifikationen in SDL/PR enthalten, und erzeugt (gemäß der in Kapitel 4 vorgestellten Semantik) daraus High-Level-Petri-Netze für den Gebrauch mit PEP (Formatbeschreibung siehe [Gra96]). Daneben werden noch drei weitere Ein- bzw. Ausgabedateien benutzt:

- Eine Datei, in der solche Eigenschaften der SDL-Spezifikation abgelegt werden, die nicht Teil des Sprachumfangs von SDL sind (siehe Abschnitt 6.2.1).
- Eine Referenzdatei, in der Bezüge zwischen Teilen des SDL-Programms und den Stellen und Transitionen des Netzes hergestellt werden. Diese Informationen sind für den in Kapitel 7 beschriebenen Formeltransformator von zentraler Bedeutung. Eine nähere Beschreibung ist in Abschnitt 6.2.2 gegeben.
- Eine Datei mit weiteren für den Formeltransformator wichtigen Informationen wie z.B. Angaben über die in den Abschnitten 4.1.2 und 4.1.4 beschriebenen Darstellungen von Signalen und Zuständen durch natürliche Zahlen, siehe auch Abschnitt 6.2.3.

Der Kommandozeilenaufruf von *s2m* ist wie folgt:

```
s2m [optionen] dateiname.sdl [dateioptionen]
```

Dabei ist `optionen` eine beliebige Kombination folgender Schalter:

- `-S` Die Benutzung von `-S` unterdrückt die Durchführung jeglicher Synchronisationsoperationen während der Übersetzung. Dies kann nützlich sein, falls unterschiedliche Programiersprachen kombiniert werden sollen ([Gra98a]).

- `-L` Laut [CCI92] können Prozeduren nicht nur innerhalb von Prozessen, sondern auch auf System- oder Blockebene definiert werden (globale Prozeduren). Wird eine solche Prozedur innerhalb eines Prozesses verwendet, so soll eine Kopie der Prozedur für diesen Prozeß angelegt werden. *s2m* verfährt daher so, daß für jede Verwendung einer globalen Prozedur innerhalb eines Prozesses eine neue Kopie des Prozedurnetzes angelegt wird.
  Bei Benutzung von `-L` wird stattdessen für jede globale Prozedur nur ein Netz angelegt, als dessen Prozeß-IDs die Menge $\{\bullet\}$ angenommen wird. Das Netz für den Aufruf einer solchen Prozedur wird mit den dafür notwendigen Änderungen versehen.
  Die Verwendung von `-L` führt bei globalen Prozeduren zu kleineren Netzen, der Normalfall (ohne `-L`) orientiert sich jedoch stärker am Z.100-Standard.

- `-BC`n Steuert die Verwendung einer Semantik-Variante für Kanäle:
  Falls $n = 0$: Benutze Netz aus Abbildung 4.3.
  Falls $n = 1$: Benutze Netz aus Abbildung 5.2.

- `-BP`n Steuert die Verwendung einer Semantik-Variante für Prozesse:
  Falls $n = 0$: Benutze Netz aus Abbildung 4.8.
  Falls $n = 1$: Hauptteil wie in Abb. 5.4, Warteschlange wie in Abb. 4.8.
  Falls $n = 2$: Hauptteil wie in Abb. 4.8, Warteschlange wie in Abb. 5.5.
  Falls $n = 3$: Hauptteil wie in Abb. 5.4, Warteschlange wie in Abb. 5.5.

- `-DC`n Setze Default-Kapazität von Kanälen auf n (Standardwert ist 1). Die Kapazität einzelner Kanäle kann auch individuell festgelegt werden, siehe Abschnitt 6.2.1.

- `-DP`n Setze Default-Anzahl maximaler Instanzen eines Prozesses, die verwendet wird, falls in der Prozeßdefinition keine maximale Anzahl festgelegt wird. Standardwert ist 1.

- `-DQ`n Setze Default-Kapazität der Eingabewarteschlangen von Prozessen (Standardwert ist 1). Die Kapazität kann auch für jeden Prozeß individuell festgelegt werden (siehe Abschnitt 6.2.1).

- `-DZ`n Setze die standardmäßig verwendete maximale Anzahl von Prozedurinstanzen. Ohne diese Option wird pro Instanz des umgebenden Prozesses eine Prozedurinstanz zugelassen. Auch diese Zahlen können wie in 6.2.1 beschrieben für einzelne Prozeduren verändert werden.

Mit `dateioptionen` können folgende Einstellungen getätigt werden:

- `-o netzdatei:` Legt den Namen der Datei fest, in der das erzeugte Netz abgespeichert wird. Wird diese Option weggelassen, so wird `dateiname.hl_net` angenommen.

`-r referenzdatei:` Legt den Namen der erzeugten Referenzdatei fest, Standard ist `dateiname.sdl_hl_ref`.

`-i infodatei:` Mit dieser Option wird der Name der Datei bestimmt, in die zusätzliche Informationen (siehe Abschnitt 6.2.3) geschrieben werden sollen. Wird diese Option nicht verwendet, so wird `dateiname.sdl_info` erzeugt.

`-p konfigurationsdatei:` Wird diese Option verwendet, so werden zusätzliche Eigenschaften der Spezifikation (siehe 6.2.1) aus `konfigurationsdatei` gelesen, andernfalls aus `dateiname.sdl_prop` (falls vorhanden).

Eine bequemere Benutzeroberfläche zu *s2m* bietet das PEP-Tool (siehe Kapitel 8).

6.2 Dateiformate

Im folgenden werden kurz die Formate der zuvor erwähnten zusätzlichen Dateien beschrieben.

6.2.1 Konfigurationsdatei

Konfigurationsdateien enthalten Informationen über einzelne Objekte von Quelltexten, die für den Übersetzer wichtig, jedoch nicht Teil der Syntax von SDL sind. Jedem SDL-Programm kann eine individuelle Konfigurationsdatei zugeordnet werden.

Konfigurationsdateien bestehen aus zwei Teilen, einem Kopf und einer Liste von Einstellungen. Der Kopf besteht aus 3 Zeilen mit folgendem Inhalt:

```
PEP
SDL_Properties
FORMAT_N
```

Die Einstellungen bestehen aus einer beliebigen Anzahl von Zeilen, deren Format wie folgt ist:

`objekt-ident .` parameter = *wert*

Dabei kann `objekt-ident` der Bezeichner eines Kanals, eines Prozesses oder einer Prozedur sein. Die erlaubten Parameter sind abhängig vom Typ des Objekts. Die Einstellungen in einer Konfigurationsdatei haben stets Priorität gegenüber den Einstellungen, die mit den Kommandozeilenparametern `-BC`, `-BP`, `-DC`, `-DP`, `-DQ` und `-DZ` getätigt werden.

Für einen Kanal können folgende Parameter gesetzt werden:

- `capacity`: Setzt das maximale Fassungsvermögen des Kanals.
- `version`: Bestimmt die Semantikvariante, die für den Kanal verwendet wird (*wert* = 0: benutze Variante aus Abbildung 4.3, *wert* = 1: benutze Variante aus Abbildung 5.2).

Prozesse haben folgende Parameter:

- `capacity`: Das Fassungsvermögen der Eingabewarteschlange.
- `version`: Die zu verwendende Semantikvariante, wobei $wert \in \{0, \ldots, 3\}$ wie bei der Kommandozeilenoption `-BP` interpretiert wird.
- `instances`: Maximale Anzahl von Instanzen eines Prozesses. Diese Einstellung hat Priorität gegenüber eventuell vorhandenen Angaben im Quelltext!

Für Prozeduren wird nur ein Parameter unterstützt:

- `instances`: Die Zahl der maximal gleichzeitig aktiven Prozedurinstanzen.

Eine mögliche Konfigurationsdatei für das Beispiel aus Kapitel 2 wäre etwa:

```
PEP
SDL_Properties
FORMAT_N
one.version=3
c2.capacity=2
```

Durch diese Einstellungen würde festgelegt, daß für den Prozeß `one` die Semantikvarianten aus den Abbildungen 5.4 und 5.5 verwendet werden und daß der Kanal `c2` zwei Signale gleichzeitig zwischenspeichern kann.

6.2.2 Referenzdatei

Referenzdateien stellen einen Bezug zwischen dem SDL-Quelltext und dem Ergebnis der Übersetzung her, d.h., es wird angegeben, welche Funktion die einzelnen Stellen und Transitionen eines Netzes haben. Ein entsprechendes Verfahren für $\mathrm{B(PN)}^2$ wurde von Grahlmann in [Gra97c] und [Gra98b] beschrieben. Der hier für SDL benutzte Ansatz funktioniert analog dazu, und auch das Dateiformat orientiert sich an dem für $\mathrm{B(PN)}^2 \rightarrow$HL-Referenzen verwendeten Format [Gra97a].

SDL→HL-Referenzdateien bestehen aus vier Teilen:

- einem Kopf
- einem Definitionsteil
- einem Teil für die Stellen
- einem Teil für die Transitionen

Jede Referenzdatei bezieht sich auf ein bestimmtes Netz.

Der Kopf

Der Kopf besteht aus den folgenden drei Zeilen:

```
PEP
References_SDL_HL
FORMAT_N
```

Der Definitionsteil
Der Definitionsteil beginnt mit der Zeile

```
DEF
```

und enthält daran anschließend je eine Zeile für jedes Objekt des Quelltextes, das von einer der Stellen oder Transitionen referenziert wird. Objekte werden durchnumeriert, vgl. die Beschreibung der Funktion f_R in Abschnitt 4.1.3. Einige Objekte, namentlich Kanäle, Signalrouten und Verbindungen, erhalten bei Bedarf zwei Referenznummern, d.h., ein bidirektionaler Kanal wird im Rahmen der Referenzdatei wie zwei unabhängige unidirektionale Kanäle behandelt.

Jede Zeile beginnt mit der Nummer des Objekts, gefolgt von einem Hinweis auf die Lage des Objekts im Quelltext und einem Kennzeichen für seinen Typ. Der darauffolgende Rest der Zeile ist von Objekttyp zu Objekttyp unterschiedlich, d.h., eine Zeile hat das folgende Format:

refnr `R"(`*zeile,spalte;zeile,spalte*`)"` `b"`*name*`"` *kennzeichen* + Rest

Dabei sind die kursiv gedruckten Ausdrücke durch entsprechende Zahlen zu ersetzen mit Ausnahme von *kennzeichen*, welches ein einzelner Buchstabe ist, und *name*, welches eine Zeichenkette ist und den im Quelltext verwendeten Namen des Objekts enthält. In der untenstehenden Beschreibung des typabhängigen Teils der Zeile steht *str* für eine in Anführungszeichen eingeschlossene Zeichenkette und *num* für eine ebenfalls in Anführungszeichen stehende Zahl. Die angegebenen Beispiele sind der Referenzdatei für das Beispiel aus Kapitel 2 entnommen.

Objekttyp	**Kennzeichen + Rest**
System	`K`
	Beispiel: `1 R"(1,1;1,9)"Kb"s"`
Block	`E`
	Beispiel: `7 R"(17,1;17,10)"Eb"foo"`
Kanal	`C` *str* `v` *num* `c` *num*
	Es enthält *str* die Namen der Signale, die der Kanal übermitteln kann, eingeschlossen in geschweifte Klammern. Handelt es sich um einen bidirektionalen Kanal, so ist zuvor noch ein Kennzeichen für die Richtung angegeben, und zwar “>” für die Hin- und “<” für die Rückrichtung.
	Die Zahl hinter `v` gibt die benutzte Sematik-Variante an (entsprechend der mit `-DC`*n* oder der durch die Konfigurationsdatei erfolgten Festgelegung). Die Zahl nach `c` gibt die Kapazität des Kanals an.
	Beispiel: `5 R"(10,1;12,11)"C"sig"v"0"c"1"b"c1"`
Signalroute	`R` *str*
	Das Format von *str* wie bei Kanälen.
	Beispiel: `12 R"(21,3;22,30)"R"sig"b"r1"`
Signal	`S`
	Beispiel: `2 R"(8,8;8,15)"Sb"sig"`

Objekttyp	**Kennzeichen + Rest**
Verbindung	`X` *str* *str* enthält genau ein Zeichen, und zwar “>”, falls über den referenzierten Teil der Verbindung Signale in den umschließenden Block hineinfließen, und “<”, falls die umgekehrte Richtung gemeint ist. Beispiel: `8 R"(19,3;19,20)"X">"b"foo@a"`
Variable	`V` *str* `u` *num* *str* beschreibt den Wertebereich der Variable, und *num* ergibt die Referenznummer des Prozesses bzw. der Prozedur, zu dem die Variable gehört. Beispiel: `16 R"(28,9;28,10)"V"0..3"u"14"b"y"`
In-Parameter	`I` *str* `u` *num* Bedeutung von *str* und *num* wie bei Variablen. Beispiel: `15 R"(27,10;27,11)"I"0..3"u"14"b"x"`
In/Out-Parameter	`O` *str* `u` *num* Bedeutung von *str* und *num* wie bei Variablen. Beispiel: `51 R"(78,14;78,17)"O"0..3"u"50"b"val"`
Prozeß	`P` *str* `v` *num* `c` *num* *str* gibt die Menge der dem Prozeß zugewiesenen Prozeß-IDs wieder. `v` gibt die Nummer der verwendeten Semantik-Variante und `c` die Kapazität der Eingabewarteschlange an. Beispiel: `14 R"(26,3;26,20)"P"1..2"v"0"c"1"b"one"`
Prozedur	`U` *str* Hier stehen in *str* die verwendeten Prozedur-IDs. Beispiel: `17 R"(30,5;30,22)"U"1..2"b"sendsig"`
Aktion	`A u` *num* Mit diesem Typ werden alle innerhalb von Prozeß- bzw. Prozedurrümpfen auftauchenden Arten von Befehlen beschrieben. Die Zahl *num* verweist auf die Referenznummer des Prozesses/der Prozedur. Beispiel: `19 R"(33,2;33,17)"Au"17"b"output sig(arg);"`
Antwort	`W` Eine Referenz dieser Art verweist auf einen Antwort-Teil einer **decision**. Im oben erwähnten Namensfeld (`b"`*name*`"`) steht die Antwort, wie sie im Quelltext formuliert wurde. Beispiel: `22 R"(39,4;39,10)"Wb"(Null):"`
Sprungmarke	`L u` *num* Die Verwendung von *num* ist gleich der bei Aktionen. Beispiel: `31 R"(53,10;53,15)"Lu"14"b"again"`
Stimulus	`T` Verweist auf **input**-Anweisungen. Das Namensfeld enthält den Quelltext des **input**-Statements. Beispiel: `28 R"(45,7;45,17)"Tb"input none;"`

Stellenreferenzen

Dieser Teil ordnet – unter Bezugnahme auf den Definitionsteil – den Stellen des Netzes Referenzen zu. Er beginnt mit der Zeile

`PL`

gefolgt von Zeilen der Form

Stellenname `T"`*str*`"`

Dabei enthält *str* eine oder mehrere Referenzen der Form *refnr*:*index*, wobei *refnr* ein Objekt aus dem Definitionsteil bestimmt, und *index* eine Zahl ist, die die Bedeutung der Stelle für dieses Objekt näher erläutert. Eine Auflistung der möglichen Indizes ist in der Tabelle unten zu finden. Hat eine Stelle mehrere Referenzen, so sind sie in *str* durch einen Stern (*) getrennt.

In den Kapiteln 4 und 5 wurden Referenzindizes mit symbolischen Namen versehen. Die folgende Tabelle zeigt auf, welche Indizes für welchen Objekttyp definiert werden, welcher numerische Wert zur Darstellung eines Indizes benutzt wird, und welche Bedeutung er hat:

Objekttyp	**Name**	**Wert**	**Bedeutung**
Kanal	buffer	4	Inhalt des Puffers
	entry	8	Eingangsstelle
	exit	16	Ausgangsstelle
	count	8192	Füllgrad des Puffers
Signalroute	internal	0	Zwischenstelle
	entry	8	Eingangsstelle
	exit	16	Ausgangsstelle
Verbindung	internal	0	Zwischenstelle
	entry	8	Eingangsstelle
	exit	16	Ausgangsstelle
Prozeß	count	1	Anzahl aktiver Instanzen
	stack	4	ID-Stack
	entry	8	Eingangsstelle
	exit	16	Ausgangsstelle
	self	64	Wertestelle für **self**
	parent	128	Wertestelle für **parent**
	offspring	256	Wertestelle für **offspring**
	sender	512	Wertestelle für **sender**
	state	1024	Zustand
	dash	2048	zuletzt benutzer Zustand
	qcount	8192	Füllgrad der Warteschlange
	buffer	16384	Inhalt der Warteschlange
Prozedur	count	1	Anzahl aktiver Instanzen
	stack	4	ID-Stack
	entry	8	Eingangsstelle
	exit	16	Ausgangsstelle
	max	32	Anzahl maximal aktiver Instanzen
	state	1024	Zustand

Objekttyp	Name	Wert	Bedeutung
	dash	2048	zuletzt benutzer Zustand
Variable	value	4	Wertestelle
	entry	8	Eingangsstelle
	exit	16	Ausgangsstelle
Label	pre	1	vorher
	post	2	nachher
Aktion	internal	0	Befehl in Ausführung begriffen
	pre	1	vorher
	post	2	nachher

Transitionenreferenzen

Der Aufbau des Teils für die Transitionen entspricht dem für Stellen, er beginnt jedoch mit der Zeile

```
TR
```

Die für Transitionen benutzten Indizes sind wie folgt:

Objekttyp	Name	Wert	Bedeutung
Kanal	init	32	Initialisierung
	term	64	Terminierung
	read	256	Lesen (Kanal nimmt ein Signal auf)
	write	512	Schreiben (Kanal gibt Signal ab)
	shift	4096	Shift-Transition
Signalroute	init	32	Initialisierung
	term	64	Terminierung
	transmit	128	Weiterleitung eines Signals
Verbindung	init	32	Initialisierung
	term	64	Terminierung
	transmit	128	Weiterleitung eines Signals
Prozeß	init	32	Initialisierung
	term	64	Terminierung
	create	8192	Prozeßerzeugung
Prozedur	init	32	Initialisierung
	term	64	Terminierung
	call	8192	Prozeduraufruf
	return	16384	Prozedurende
Variable	init	32	Initialisierung
	term	64	Terminierung
	access	128	Zugriff
Label	atomic	0	Sprungmarke
Aktion	atomic	0	Ausführung

6.2.3 Informationsdatei

Wie in Kapitel 4 beschrieben stellt der Compiler Zustände und Signale im Netz durch natürliche Zahlen dar; auch für Prozeß-IDs werden natürliche Zahlen benutzt, um Prozeßinstanzen im Kontrollfluß wie auch als Absender und Empfänger von Signalen zu identifizieren. Um die vom Compiler erzeugten Netze sinnvoll interpretieren und Rückschlüsse aus den Ergebnissen seiner Simulation ziehen zu können, ist es notwendig, Näheres über diese Darstellungen zu wissen. Informationsdateien enthalten die dazu notwendigen Daten.

Informationsdateien bestehen aus vier Teilen:

- einem Kopf
- einem Teil für Zustände
- einem Teil für Prozeß-IDs
- einem Teil für Signale

Der Kopf

Der Kopf besteht aus den folgenden drei Zeilen:

```
PEP
SDL_Information
FORMAT_N
```

Der Teil für die Zustände

Dieser beginnt mit der Zeile

```
STATE
```

und enthält für jeden in einem Prozeßrumpf benutzten Zustand eine Zeile der Form

`process-ident.state-ident =` f_Q(`state-ident`)

bzw. für jeden in einem Prozedurrumpf erscheinenden Zustand die Zeile

`procedure-ident.state-ident =` f_Q(`state-ident`)

Der Teil für die Prozeß-IDs

Dieser Teil wird durch die Zeile

```
PID
```

eingeleitet und beinhaltet für jeden Prozeß Π genau $|ID(\Pi)|$ Zeilen der Form

`process-ident[`i`] =` $\min(ID(\Pi)) - 1 + i$

für $i \in \{1, \ldots, |ID(\Pi)|\}$.

Der Teil für die Signale

Der Start dieses Abschnitts wird markiert durch

```
SIGNAL
```

Daran anschließend folgt je eine Zeile für jedes in der Spezifikation benutzte Signal und für jede mögliche Kombination seiner Argumente. Kann ein Signal s beispielsweise mit den Argumenten $x_1, \ldots, x_n$ versehen sein, so wird diese Zeile erzeugt:

$s(x_1,\ldots,x_n)$ = $f_S(s(x_1,\ldots,x_n))$

6.3 Betrachtung der Kommunikationsstrukturen

Wie bereits erwähnt läßt sich die Übersetzung von SDL-Spezifikationen nicht vollkommen kompositionell gestalten. Dies wäre jedoch wünschenswert, um den Compiler so zu konstruieren, daß er die vorgestellte Semantik möglichst exakt implementiert. Aus diesem Grund wurde das Backend in zwei Phasen geteilt: Die erste Phase berechnet unter Betrachtung des gesamten Systems einige Größen, die dann von der zweiten Phase (der kompositionellen Netzerzeugung) verwendet werden.

Die von der ersten Phase berechneten Größen beinhalten:

- für jeden Kanal und für jede Signalroute die (IDs der) Prozesse, die auf diesem Kanal/dieser Signalroute als Sender bzw. Empfänger eines Signals in Erscheinung treten können;
- für jeden Prozeß die Menge der Signale, die er empfangen kann, und die Menge der (IDs der) Prozesse, von denen er Signale empfangen kann;
- Informationen über die Wege, die ein Signal nehmen kann, um von einem Prozeß zum anderen zu gelangen; dies wird für die Übersetzung von **output** gebraucht.

Bis auf den letzten Punkt ließen sich diese Informationen auch durch großzügige Abschätzungen ersetzen; bei größeren Systemen würde dies jedoch rasch zu sehr großen LL-Netzen führen. Im folgenden wird die Berechnung der in den ersten beiden Punkten genannten Mengen dargestellt; der dabei erzeugte Graph wird dann in der zweiten Phase bei der Übersetzung von **output** genutzt.

Der Algorithmus konstruiert einen Graphen, dessen Knoten die Prozesse und Verbindungen eines Systems repräsentieren, und dessen Kanten die Kanäle und Signalrouten darstellen. Knoten und Kanten werden mit einer Beschriftung versehen, die sich im Verlaufe des Algorithmus ändert und mit Hilfe derer die genannten Größen berechnet werden. So enthält die Beschriftung von Prozeß-Knoten (wie oben angedeutet) die Menge der Signale, die der zugehörige Prozeß erhalten kann, sowie die Menge der Prozeß-IDs, von denen diese Signale kommen können. Die Beschriftung von Verbindungs-Knoten ist etwas komplizierter: Sie enthält die Menge der Prozeß-IDs, an die Signale durch diese Verbindung gesandt werden (*RECV*), und die Menge der Prozeß-IDs, die Signale durch diese Verbindung senden (*SEND*). In den Mengen *SENT* und *RECVD* merkt sich der Algorithmus, welche von diesen IDs er bereits behandelt hat (siehe unten). Kanten schließlich sind beschriftet mit den Signalen, die der zugehörige Kanal/die zugehörige Signalroute transportieren kann, und mit den sendenden und empfangenden Prozessen (ebenfalls wie oben angedeutet).

Der Algorithmus besteht aus drei Abschnitten:

Initialisierung

Zu einem SDL-System S wird ein gerichteter Graph $N_S = (P \cup X, A)$ mit einer Beschriftungsfunktion ι konstruiert. Ein Knoten $p \in P$ besitzt die Beschriftung $\iota(p) = (SIG_p \mid SEN_p)$. Für jeden Prozeß Π des Systems enthält das Netz genau einen Knoten $p_\Pi \in P$, der anfänglich mit der Beschriftung $\iota(p_\Pi) = (\emptyset \mid ID(\Pi))$ versehen ist.

Die Beschriftung eines Knotens $x \in X$ ist ein 4-Tupel, d.h. $\iota(x) = (SEND_x \mid SENT_x \mid RECV_x \mid RECVD_x)$. Für jede Verbindung Ξ des Systems enthält das Netz zwei Knoten, $x_\Xi \in X$ für die in den zugehörigen Block hineingehenden Signale und $y_\Xi \in X$ für die herausgehenden Signale. Die anfängliche Beschriftung der Knoten ist $\iota(x_\Xi) = \iota(y_\Xi) = (\emptyset \mid \emptyset \mid \emptyset \mid \emptyset)$.

Eine Kante $a \in A$ wird mit dem Tripel $\iota(a) = (SIG_a \mid SEN_a \mid REC_a)$ beschriftet. Für jeden Kanal C existiert eine gerichtete Kante, deren Endpunkte diejenigen Knoten sind, die die Endpunkte des Kanals repräsentieren. Diese Kante a_C hat zu Beginn die Beschriftung $\iota(a_C) = (SIG(C) \mid \emptyset \mid \emptyset)$. Für bidirektionale Kanäle werden zwei Kanten angelegt; Signalrouten werden für die Zwecke dieses Algorithmus wie Kanäle behandelt.

Außer den genannten Knoten und Kanten enthält der Graph keine weiteren Knoten oder Kanten. Für das Beispiel aus Kapitel 2 zeigt Abbildung 6.1 den Zustand des Graphen nach der Initialisierung.

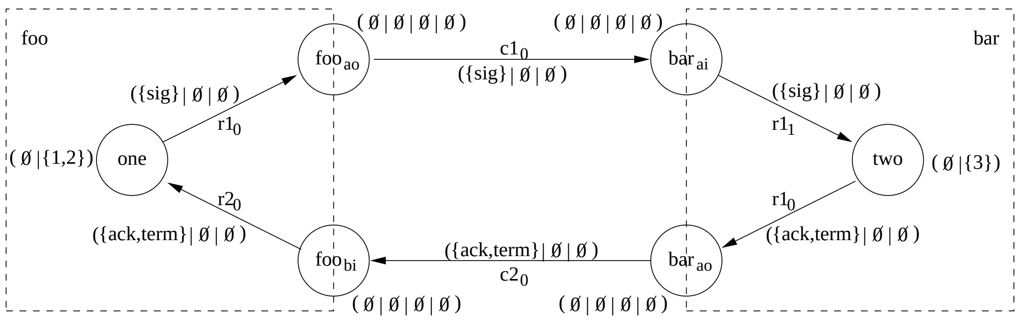

Abbildung 6.1: Zustand des Netzes nach der Initialisierung

Durch das so erzeugte Netz werden nun schrittweise Prozeß-IDs "verschickt", wodurch sich die Beschriftungen von Knoten und Kanten ändern. Signale werden stets in zwei Richtungen verfolgt, und zwar entlang und entgegen der Richtung, in der sie fließen. Dies ist notwendig, um sowohl Sender- als auch Empfängermengen zu ermitteln.

Start

Nach der Initialisierung werden für jeden Prozeß Π folgende Schritte ausgeführt:

für jede Kante $a = (p_\Pi, p_{\Pi'})$
erweitere SEN_a um $ID(\Pi)$
erweitere $SEN_{p_{\Pi'}}$ um $ID(\Pi)$

für jede Kante $a = (p_{\Pi'}, p_\Pi)$
erweitere REC_a um $ID(\Pi)$
erweitere SIG_{p_Π} um SIG_a

für jede Kante $a = (p_\Pi, x)$, $x \in X$
erweitere SEN_a um $ID(\Pi)$
erweitere $SEND_x$ um $ID(\Pi) \setminus SENT_x$

für jede Kante $a = (x, p_\Pi)$, $x \in X$
erweitere REC_a um $ID(\Pi)$
erweitere $RECV_x$ um $ID(\Pi) \setminus RECVD_x$
erweitere SIG_{p_Π} um SIG_a

Iteration

Die Iteration wird wiederholt, bis $SEND_x = RECV_x = \emptyset$ für alle Knoten $x \in X$ gilt.

Gibt es ein $x \in X$, so daß $SEND_x \neq \emptyset$ ist, so geschieht folgendes:

für jede Kante $a = (x, p_\Pi)$, $p_\Pi \in P$
erweitere SEN_a um $SEND_x$
erweitere SEN_{p_Π} um $SEND_x$

für jede Kante $a = (x, x')$, $x' \in X$
erweitere SEN_a um $SEND_x$
erweitere $SEND_{x'}$ um $SEND_x \setminus SENT_{x'}$

anschließend
erweitere $SENT_x$ um $SEND_x$
setze $SEND_x = \emptyset$

Für jedes $x \in X$ mit $RECV_x \neq \emptyset$ wird dies getan:

für jede Kante $a = (p_\Pi, x)$, $p_\Pi \in P$
erweitere REC_a um $RECV_x$

für jede Kante $a = (x', x)$, $x' \in X$
erweitere REC_a um $RECV_x$
erweitere $RECV_{x'}$ um $RECV_x \setminus RECVD_{x'}$

anschließend
erweitere $RECVD_x$ um $RECV_x$
setze $RECV_x = \emptyset$

Am Ende dieser Iteration gilt für jeden Prozeß Π: $SIG(\Pi) = SIG_{p_\Pi}$ und $ID_s(\Pi) = SEN_{p_\Pi}$. Für jeden Kanal (bzw. jede Signalroute) C gilt $ID_s(C) = SEN_{a_C}$ und $ID_r(C) = REC_{a_C}$. Abbildung 6.2 zeigt das Endergebnis des Algorithmus für das Beispiel.

Bemerkung: $ID_s(\Pi)$ enthält $ID(\Pi)$, da **sender** den Wert **self** annimmt, wenn ein spontaner Zustandsübergang stattfindet.

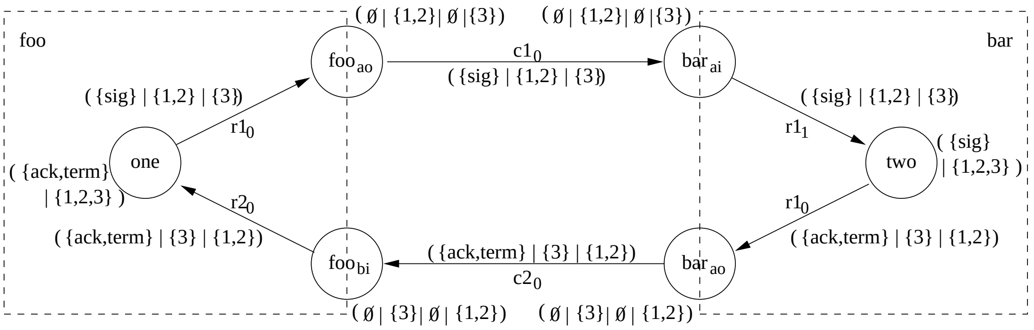

Abbildung 6.2: Zustand des Netzes am Ende des Algorithmus

Kapitel 7

Der Formeltransformator

Wie eingangs erwähnt stand neben der Simulation die automatische Verifizierung von SDL-Spezifikationen als Motivation für diese Arbeit Pate. Das PEP-Projekt integriert auf Basis von Petri-Netzen eine Reihe von Verifikationsalgorithmen [Gra97b]. Mit diesen Algorithmen können Aussagen über Eigenschaften von Netzen gewonnen werden, etwa "Es ist möglich, daß Transition T schaltet oder die Markierung M erreicht wird." Um wirklich aussagekräftige Erkenntnisse über SDL-Spezifikationen zu erhalten, sollte aber die Möglichkeit gegeben sein, Aussagen nicht erst über Petri-Netze (d.h. über das Ergebnis der Übersetzung), sondern direkt über SDL-Spezifikationen zu treffen, etwa "Es ist möglich, daß die Variable `x` im Prozeß `two` irgendwann einmal den Wert 3 annimmt." Um diese Möglichkeit zu schaffen, wurde entschieden, den von Stefan Römer entwickelten Formeltransformator für SDL-Zwecke zu erweitern.

Der Transformator erzeugt Netz-Formeln, deren Erfülltheit mit einem auf McMillan-Entfaltungen [McM92, ERV96] basierenden Modelchecker ([Esp93, Gra97d]) überprüft wird. Formeln machen Aussagen über die Erreichbarkeit von Markierungen, d.h., es können Erkenntnisse über das dynamische Verhalten eines Netzes gewonnen werden. Formeln werden stets ausgewertet unter Bezugnahme auf ein gegebenes LL-Netz und eine Anfangsmarkierung (meist die Startmarkierung des Netzes). Sind beispielsweise ein Netz N und eine Markierung M (wobei M eine Markierung von N sei) gegeben, so bedeuten die Formeln

- $\mathsf{P1} \wedge \mathsf{P2}$: Die Stellen $\mathsf{P1}$ und $\mathsf{P2}$ sind in M markiert.
- $\Diamond(\mathsf{P1} \wedge \mathsf{P2})$: Wird ausgehend von der Markierung M das Netz N simuliert, so kann eine Markierung M' erreicht werden, bei der $\mathsf{P1}$ und $\mathsf{P2}$ markiert sind ($\Diamond$ = "irgendwann").
- $\Box(\mathsf{P1} \wedge \mathsf{P2})$: Jede Markierung, die von M aus erreicht werden kann, enthält $\mathsf{P1}$ und $\mathsf{P2}$ ($\Box$ = "immer").
- $\Box\Diamond(\mathsf{P1} \wedge \mathsf{P2})$: Aus jeder Markierung, die von M aus erreicht werden kann, kann wiederum eine Markierung erreicht werden, die $\mathsf{P1}$ und $\mathsf{P2}$ enthält.

Der Formeltransformator kann als Makro-Expander bzw. als Präprozessor für den Modelchecker verstanden werden: Er versteht eine größere Menge von Formeln als der

Modelchecker. Erhält er eine Formel als Eingabe, so ersetzt er solche Teilausdrücke, die der Modelchecker nicht versteht, durch einfachere Ausdrücke, die dieser auswerten kann. Im bisherigen Formeltransformator wurde etwa der Ausdruck "Aktion a" ersetzt durch den Namen der Transition, die im zugehörigen Netz die Übersetzung von a bildet. Wird dann gefragt, ob der Vorbereich der Transition markiert sein kann, so ist dies gleichbedeutend mit der Frage nach der Ausführbarkeit von a. Auch allgemeine Netz-Eigenschaften wie etwa "Netz terminiert" konnten ausgedrückt werden und wurden durch entsprechend kompliziertere Ausdrücke ersetzt.

Die Erweiterung des Formeltransformators für die Zwecke dieser Arbeit bestand darin, ihm die Übersetzung SDL-spezifischer Eigenschaften in Netz-Formeln beizubringen. So können jetzt beispielsweise die Inhalte von Variablen, der Zustand, in dem sich eine Prozeßinstanz befindet, der Inhalt und Füllgrad von Kanälen und andere Dinge abgefragt werden. Dazu bedient sich der Transformator der vom Compiler erzeugten Referenz- und Informationsdateien (siehe Abschnitte 6.2.2 und 6.2.3).

7.1 Programmaufruf und Optionen

Der Formeltransformator besteht aus dem Programm *TransForm*. Er erzeugt aus einem Low-Level-Netz und einer Eingabeformel ϕ unter Zuhilfenahme geeigneter Referenzdateien eine Ausgabeformel $T(\phi)$, die an den Modelchecker weitergereicht werden kann. Der Aufruf ist wie folgt:

```
TransForm [option [option [...]  ]  ]
```

Als Optionen können dabei (in beliebiger Reihenfolge) auftreten:

`-n=netzdatei:` Legt die Datei fest, aus der das Netz gelesen wird. Es muß sich dabei um ein Low-Level-Netz handeln (der Modelchecker arbeitet nur mit solchen Netzen).

`-i=eingabedatei:` Aus dieser Datei wird die Eingabeformel gelesen.

`-o=ausgabedatei:` In diese Datei wird die transformierte Formel geschrieben.

`-s=referenzdatei:` Gibt eine Datei an, aus der SDL→LL-Referenzen genommen werden, die eine Verbindung zwischen dem ursprünglichen SDL-Programm und dem Netz herstellen. Nur bei Benutzung dieser Option schaltet *TransForm* in den "SDL-Modus"; ohne `-s` arbeitet *TransForm* ohne die in dieser Arbeit gemachten Erweiterungen.
SDL→LL-Referenzen werden bei der Entfaltung von High-Level- in Low-Level-Netze durch das Programm *hl2ll* (Abschnitt 8.2) erzeugt; Grundlage dafür sind die durch den SDL-Compiler geschaffenen SDL→HL-Referenzen.

`-x=infodatei:` Aus dieser Datei werden die in Abschnitt 6.2.3 beschriebenen Zusatzinformationen gelesen. Wird `-s`, aber nicht `-x` angegeben, so versucht *TransForm*, die Informationen aus einer Datei zu lesen, die denselben Basisnamen wie die Referenzdatei, aber die Endung `.sdl_info` hat.

7.2 Erweiterungen

Folgende Erweiterungen wurden am Formeltransformator vorgenommen:

Quantoren/Variablen: Der Allquantor und der Existenzquantor können benutzt werden, um eine (Teil-)Formel über mehrere Werte eines Parameters zu variieren. Dazu können (gebundene) Variablen in einer Formel vorkommen. So akzeptiert der Transformator Formeln der Art

$$\forall a \in set \colon \phi$$

$$\exists a \in set \colon \phi$$

wobei ϕ eine Formel sein muß, die die Variable a enthalten kann. Die Formel wird so transformiert, daß (die Transformation der Formel) ϕ für jeden Wert der Menge *set* einmal wiederholt wird, wobei a durch den jeweiligen Wert aus *set* ersetzt wird. Besteht *set* aus den Werten $i_1, \ldots, i_n$, so erzeugt das Programm aus den obigen Formeln

$$T(\phi[a \leftarrow i_1]) \wedge \ldots \wedge T(\phi[a \leftarrow i_n])$$

bzw.

$$T(\phi[a \leftarrow i_1]) \vee \ldots \vee T(\phi[a \leftarrow i_n])$$

Mengen können entweder durch explizite Angabe von Werten konstruiert werden ($\{\mathtt{i_1}, \mathtt{i_2}, \ldots, \mathtt{i_n}\}$), oder es können folgende Ausdrücke verwendet werden:

- *IDSET*(Π), d.h. die Menge der IDs eines Prozesses Π;
- *IDSET*(P), d.h. die Menge der IDs einer Prozedur P;
- *PID*, d.h. die Menge aller Prozeß-IDs einschließlich **Null** und 0;
- *VALUE*(v), d.h. die Wertemenge einer Variablen v;
- *QUEUEPOS*(Π), d.h. die Menge, mit der Einträge in der Warteschlange eines Prozesses Π indiziert werden ($= \{1, \ldots, CAP(\Pi)\}$);
- *QUEUEPOS*(C), d.h. die Menge, mit der Einträge in der Pufferstelle eines Kanals C indiziert werden ($= \{1, \ldots, CAP(C)\}$).

Zugriff auf Elemente der Spezifikation: Indem die SDL$\rightarrow$LL-Referenzen ausgewertet werden, können Zugriffe auf folgende Bestandteile einer SDL-Spezifikation transformiert werden:

- die Anzahl aktiver Instanzen eines Prozesses;
- die Anzahl aktiver Instanzen einer Prozedur;
- die Werte der impliziten Variablen einer Prozeßinstanz (**parent**, **offspring**, **sender**);
- die Werte der Variablen und Parameter von Prozessen und Prozeduren;

- den Zustand einer Prozeß- bzw. Prozedurinstanz
- die Anzahl von Signalen in der Warteschlange von Prozessen;
- den Inhalt einer Warteschlange (Signal, Sender);
- die Anzahl von Signalen in der Pufferstelle eines Kanals;
- den Inhalt eines Kanals (Signal, Sender, Empfänger).
- die Aufrufkette von Prozeduren

7.3 Syntax von Formeln

Dieser Abschnitt faßt die Syntaxregeln für die vom Transformator verstandenen Formeln zusammen, soweit sie die Verifikation auf SDL-Ebene betreffen. (Eine Beschreibung der gesamten Syntax mit Regeln für allgemeine Petri-Netze würde hier zu weit führen und hätte auch nichts mit dem zu tun, was im Rahmen dieser Arbeit getan wurde.)

Die oben beschriebenen Erweiterungen für SDL sind in den Regeln für quant, property und condition zusammengefaßt. Für die Verwendung verschiedener Zeichensätze gelten dieselben Regeln wie in Kapitel 4.

Der Aufbau einer Formel ist wie folgt:

formula ::= $\Diamond$ formula | $\Box$ formula |
formula $\wedge$ formula | formula $\vee$ formula |
formula $\Rightarrow$ formula | formula $\Leftrightarrow$ formula |
$\neg$ formula | **(** formula **)** | **true** | **false** |
macro | quant | property | condition

Unter dem Nichtterminal macro sind einige allgemeine Netzeigenschaften zusammengefaßt, die auch für solche Netze interessant sind, die aus SDL-Programmen hervorgegangen sind. Der Ausdruck **deadlockfree** ist genau dann wahr, wenn keine Verklemmung auftreten kann, d.h. stets gewährleistet ist, daß mindestens eine Transition schalten kann; **termination** ist genau dann wahr, wenn das Netz (d.h. auf SDL-Ebene übertragen das System) terminiert, d.h. die Endmarkierung (alle Exit-Stellen sind markiert) erreicht werden kann.

macro ::= **deadlockfree** | **termination**

Die Syntax für den Einsatz des All- bzw. des Existenzquantors ist wie folgt:

quant ::= $\forall$ `var–ident` quant–set **:** formula |
$\exists$ `var–ident` quant–set **:** formula
quant–set ::= **{** explicit–set **}** |
IDSET (process–type **)** | **IDSET (** procedure–type **)** |
PID | **VALUE (** variable–type **)** |
QUEUEPOS (process–type **)** |
QUEUEPOS (`channel–ident` **)**

explicit–set	::=	set–elem | set–elem **,** explicit–set
set–elem	::=	z | z_1 **..** z_2
process–type	::=	`process–ident`
procedure–type	::=	`procedure–ident`
variable–type	::=	process–type **.** `variable–ident` |
		procedure–type **.** `variable–ident`

Jeder der folgenden Ausdrücke wird in eine oder-verknüpfte Liste von Stellen transformiert, die genau dann markiert sind, wenn der Ausdruck wahr wird. Um diese Stellen zu finden, werden die SDL→LL-Referenzen zu Hilfe genommen. Beispiel: Soll (im Programm aus Kapitel 2) der Ausdruck

```
one.noinstances > 0
```

transformiert werden, so werden die LL-Stellen mit der Referenz f_R(`one`):count gesucht, die bei der Entfaltung von High-Level- in Low-Level-Netz für Werte von mehr als 0 erzeugt wurden (zum Format von SDL→LL-Referenzen siehe Abschnitt 8.2). Wird keine passende Stelle gefunden, so wird der Ausdruck in **false** transformiert.

property	::=	process–instance **. noqueue** arith–op value |
		process–type **. noinstances** arith–op value |
		process–instance **. parent** class–op process–value |
		process–instance **. offspring** class–op process–value |
		process–instance **. sender** class–op process–value |
		variable–instance arith–op value |
		variable–instance class–op process–value |
		procedure–type **. noinstances** arith–op value |
		`channel–ident` **. noqueue** arith–op value |
		process–instance **. state** class–op state–name |
		procedure–instance **. state** class–op state–name |
		process–instance **. queue** [value] **. signal** class–op signal–name |
		process–instance **. queue** [value] **. sender** class–op process–value |
		`channel–ident` **. queue** [value] **. signal** class–op signal–name |
		`channel–ident` **. queue** [value] **. sender** class–op process–value |
		`channel–ident` **. queue** [value] **. receiver** class–op process–value |
		procedure–instance **. caller** class–op process–instance |
		procedure–instance **. caller** class–op procedure–instance

Um Prozeß-/Prozedurinstanzen zu unterscheiden, muß der Name i.a. durch eine ID gefolgt werden. Die ID kann weggelassen werden, falls nur eine einzige Instanz möglich ist, und muß angegeben werden, wenn mehrere Instanzen möglich sind.

process–instance	::=	process–type |
		process–type [z] |
		process–type [`var–ident`]
procedure–instance	::=	procedure–type |
		procedure–type [z] |
		procedure–type [`var–ident`]
variable–instance	::=	process–instance **.** `variable–ident` |
		procedure–instance **.** `variable–ident`

Je nach Ausdruck sind unterschiedliche Operatoren und Operanden zugelassen:

arith–op	::=	$= \mid \neq \mid < \mid \leq \mid > \mid \geq$
class–op	::=	$= \mid \neq$
value	::=	z | `var–ident` | **True** | **False**
process–value	::=	process–instance | **Null** | **System**

Dabei steht **System** für die Umgebung des Systems als Absender bzw. Empfänger eines Signals (Prozeß-ID 0). Die zulässigen Werte für state–name und signal–name werden der auf der Kommandozeile angegebenen Informationsdatei entnommen. Der spezielle Zustandsname **Internal** steht für "Zustandsübergang wird abgearbeitet".

Mit condition kann eine zusätzliche Bedingung an eine der gebundenen Variablen gestellt werden, indem ein Vergleich mit einer Konstanten oder mit einer anderen gebundenen Variablen angestellt wird. Diese Bedingungen werden in einen Wahrheitswert (**true** oder **false**) transformiert.

condition ::= value arith–op value

In der oben dargestellten Grammatik wurden einige Sonderzeichen verwendet, die im Transformator durch Folgen von ASCII-Symbolen dargestellt werden. Sonderzeichen und ASCII-Darstellung sind in der untenstehenden Tabelle gegenübergestellt.

Sonderzeichen	ASCII-Darstellung
$\Diamond$	`^`
$\Box$	`#`
$\wedge$	`*`
$\vee$	`+`
$\Rightarrow$	`=>`
$\Leftrightarrow$	`<=>`
$\neg$	`-`
false	`0`
true	`1`
$\forall$	`FORALL`
$\exists$	`EXISTS`
$\neq$	`#`
$\leq$	`<=`
$\geq$	`>=`

7.4 Beispiele

Anhand der Spezifikation aus Kapitel 2 werden ein paar Anwendungsbeispiele gezeigt:

Kann sich jemals mehr als ein Signal in der Warteschlange eines Prozesses oder eines Kanals befinden?

$$\Diamond((\exists\, a\ IDSET(one):\ one[a].noqueue > 1)$$
$$\vee\ c1.noqueue > 1 \vee c2.noqueue > 1 \vee two.noqueue > 1)$$

Antwort (des Modelcheckers): Nein
Dieser Test macht selbstverständlich nur Sinn, wenn der Compiler angewiesen wurde,

Kanäle und Eingabewarteschlangen mit einer Kapazität vom mehr als einem Signal zu übersetzen (z.B. Optionen `-DC2`, `-DQ2`)!

Die Variable `x` des Prozesses `two` kann ihren Wert niemals vermindern:

$$\forall\, a\; VALUE(two.x):$$
$$\Box(two.x = a \Rightarrow (\forall\, b\; VALUE(two.x):\ \Box\, b < a \Rightarrow two.x \neq b))$$

Antwort: Ja

Derselbe Sachverhalt, negativ formuliert:

$$\exists\, a\; VALUE(two.x):$$
$$\Diamond(two.x = a \wedge (\exists\, b\; VALUE(two.x):\ \Diamond\, b < a \wedge two.x = b))$$

Antwort: Nein

Das System terminiert:

termination

Antwort: Ja

Im Verlauf der Simulation erreicht die Variable x den Wert 3:

$$\Diamond\, two.x = 3$$

Antwort: Ja

Das Signal `term` wird irgendwann über den Kanal `c2` versandt:

$$\exists\, p\; QUEUEPOS(c2):\ \Diamond\, c2.queue[p].signal = term$$

Antwort: Ja

Die Eingabewarteschlangen von Instanzen des Prozesses `one` können nur dann Signale enthalten, wenn sich die betreffende Instanz im Wartezustand befindet oder gerade einen Zustandsübergang abarbeitet.

$$\forall\, a\, hfIDSET(one):$$
$$\Box\,(one[a].noqueue > 0 \Rightarrow (one[a].state = wait \vee one[a].state = INTERNAL)$$

Antwort: Ja

Oder auch:

$$\forall\, p\; QUEUEPOS(c2){:}\;\; \forall\, a\; IDSET(one){:}$$
$$\square\,(c2.queue[p].receiver = one[a]$$
$$\Rightarrow (one[a].state = wait \vee one[a].state = INTERNAL))$$

Antwort: Ja

Es wird nie ein Signal an `two` gesandt, wenn dieser Prozeß nicht existiert:

$$\square\,(c1.noqueue > 0 \Rightarrow two.noinstances = 1)$$

Antwort: Ja

Aus den letzten beiden Formeln ergibt sich, daß niemals Signale an tote Prozeßinstanzen versandt werden.

Kapitel 8

Integration in PEP

Um den Compiler und den Formeltransformator vollständig in PEP zu integrieren, mußten noch einige kleinere Arbeiten verrichtet werden, die dieses Kapitel beschreibt: Das PEP-Tool unterstützt nun SDL-Projekte, und einige Hilfsprogramme (*hl2ll*, *slist* und *simplify*) wurden für SDL-Zwecke erweitert.

8.1 Das PEP-Tool

Die Erweiterungen am PEP-Tool wurden von Stephan Ptak durchgeführt und waren nicht Teil dieser Arbeit. Sie werden hier dennoch beschrieben, um insbesondere neuen Benutzern von PEP den Einstieg in die Benutzung von Compiler und Modelchecker zu erleichtern. Eine ausführlichere Beschreibung des PEP-Tools findet sich in [BG95].

Bei Aufruf des PEP-Tools erscheint die in Abbildung 8.1 gezeigte Dialogbox. Durch Wahl von "New Project" kann ein Projekt angelegt werden, bereits existierende Projekte können durch "Open Project" geladen werden.

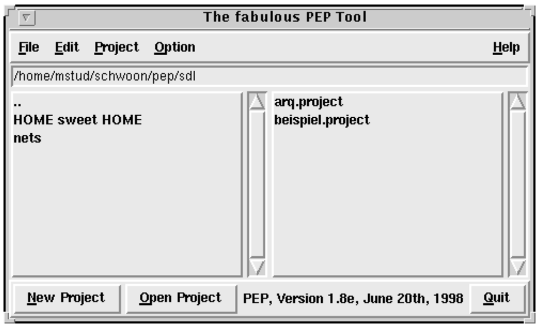

Abbildung 8.1: Das Startfenster von PEP

Ein "Projekt" besteht aus einer Menge von Objekten (bzw. den Dateien, in denen diese Objekte abgespeichert sind) – im Falle eines SDL-Projekts sind dies u.a. ein Quelltext, das durch den Compiler erzeugte HL-Netz, das daraus (durch *hl2ll*) erzeugte LL-Netz, der daraus wiederum gewonnene endliche Präfix und eine Formeldatei. Abbildung 8.2 zeigt das Projektfenster, welches nach Auswahl eines Projekts erscheint. Das exakte Erscheinungsbild dieses Fensters (die Anordnung der Icons) kann vom Benutzer geändert werden.

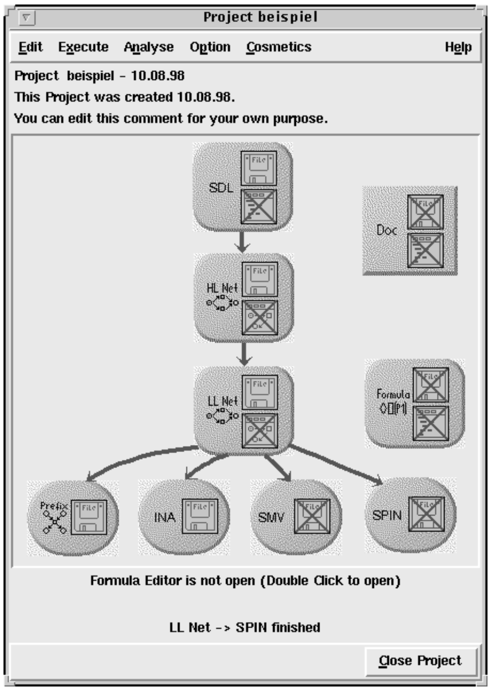

Abbildung 8.2: Fenster für ein Projekt unter PEP

Durch Doppelklick (mit der linken Maustaste) auf die Icons für den SDL-Quelltext, die Netze und die Formel kann der zugehörige Editor aufgerufen werden. Abbildung 8.3 zeigt den Quelltext-Editor. Schlüsselwörter innerhalb des Quelltexts werden fett dargestellt.

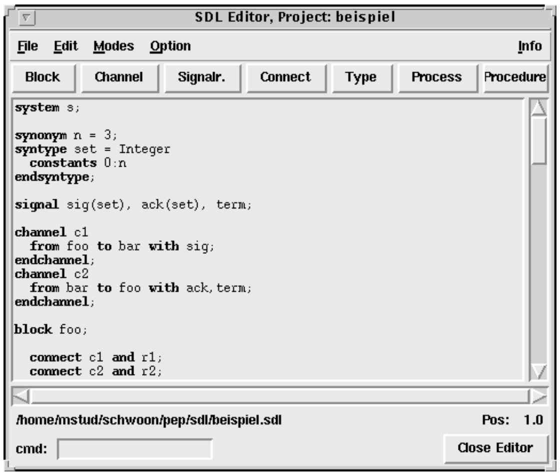

Abbildung 8.3: Der SDL-Quelltexteditor

Durch Wahl von "Edit SDL Properties" im Menü "File" wird ein Editor für die zum Quelltext gehörige Konfigurationsdatei (siehe Abschnitt 6.2.1) aufgerufen, der in Abbildung 8.4 zu sehen ist.

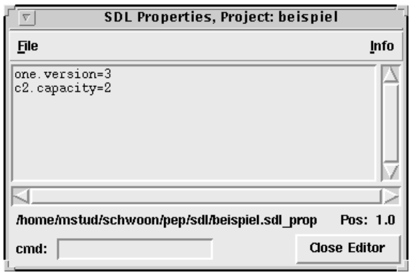

Abbildung 8.4: Editor für Konfigurationsdateien

Durch Doppelklick auf einen der Pfeile wird der zugehörige Übersetzer aufgerufen. Z.B. wird durch Doppelklick auf den Pfeil zwischen SDL-Quelltext und HL-Editor der in Kapitel 6 beschriebene Compiler aufgerufen, d.h., aus dem Quelltext wird das zugehörige HL-Netz erzeugt. Durch Druck auf die rechte Maustaste können Einstellungen für jeden Übersetzer getätigt werden. Abbildung 8.5 zeigt die Dialogbox für den SDL-Compiler.

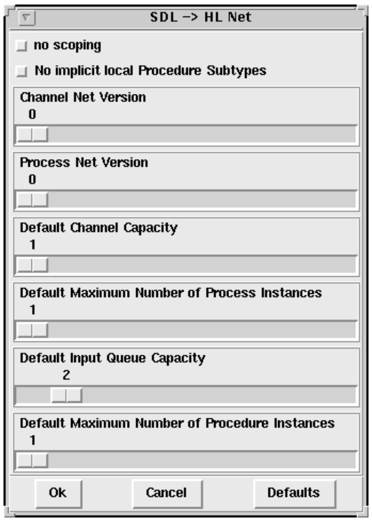

Abbildung 8.5: Einstellung der Compiler-Optionen

In den Netz-Editoren können die jeweiligen Petri-Netze simuliert werden. Abbildung 8.6 zeigt den HL-Editor.

Die vom Compiler erzeugten Referenzen können im Editor auf zweierlei Art genutzt werden. Sind der Quelltext-Editor und einer der Netz-Editoren gleichzeitig offen, so wird der Netz-Editor durch Wahl des "Ref"-Knopfs (rechts oben im Bild) in einen "Referenz-Modus" geschaltet. Wird daraufhin eine Stelle oder eine Transition angeklickt, so werden die zugehörigen Quelltext-Fragmente im SDL-Editor farbig hervorgehoben. Ferner bewirkt die Aktivierung von "Highlight Execution" im Menü "Option", daß bei anschließender Simulation des Netzes die Zuordnung der schaltenden Transitionen zum Quelltext gezeigt wird. Auf diese Weise kann die Simulation des Netzes gleichzeitig als Simulation des SDL-Systems angesehen werden.

Durch Doppelklick auf das Formel-Icon öffnet sich der Formeleditor (Abbildung 8.7), durch Doppelklick auf das Prefix-Icon ein Fenster für den Modelchecker (Abbil-

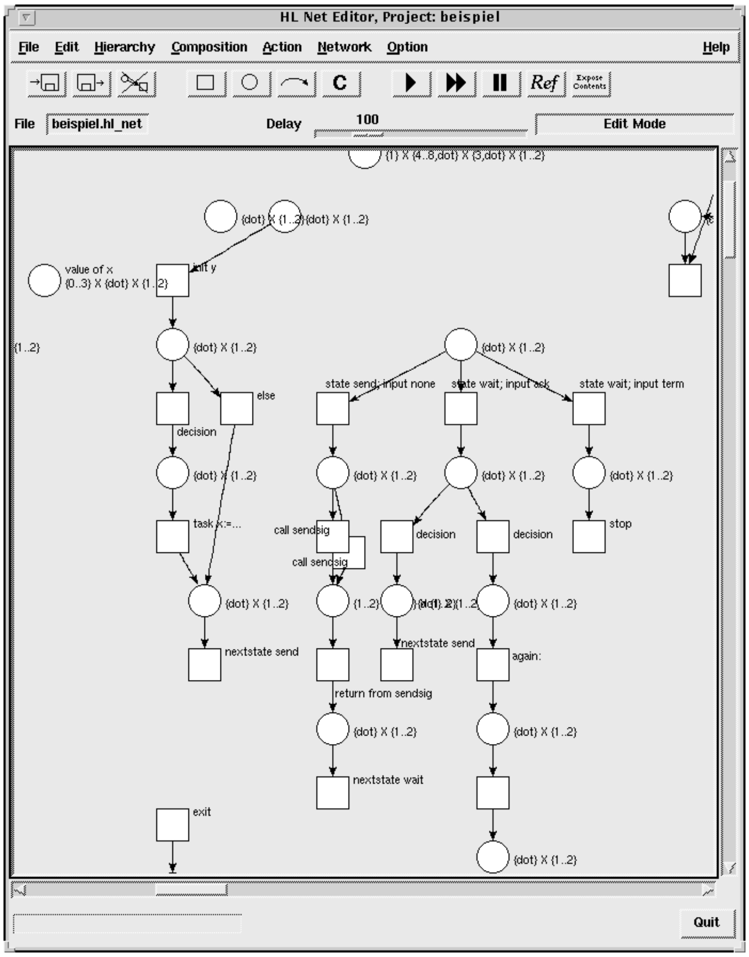

Abbildung 8.6: Der HL-Netz-Editor

dung 8.8). Eine Unterstützung für die anderen in Abbildung 8.2 gezeigten Verifikationspakete ist zu diesem Zeitpunkt noch nicht implementiert.

Wählt man im Modelchecker-Fenster den Knopf "Ok", so wird die zuvor im Formeleditor abgespeicherte Formel durch den (in Kapitel 7 beschriebenen) Transformator in eine Netz-Formel umgewandelt und geprüft. Das Ergebnis, sowie (Gegen-)Beispiele werden im Modelchecker-Fenster gezeigt.

8.2 Erweiterungen an hl2ll

Die Entfaltung von High-Level- in Low-Level-Petri-Netze geschieht durch das Programm *hl2ll*, dessen Erstimplementierung durch Stefan Roemer erfolgte. Ist ein HL-Netz aus einer SDL-Spezifikation hervorgegangen, so ist bei der Entfaltung darauf zu achten, daß der Zusammenhang zwischen SDL-Quelltext und LL-Netz nicht verlorengeht, d.h., bereits existierende Referenzen zwischen SDL-Quelltext und HL-Netz müssen auf LL-Ebene durchgezogen werden. Die Erweiterungen an *hl2ll* implementieren SDL→LL-Referenzen wie folgt:

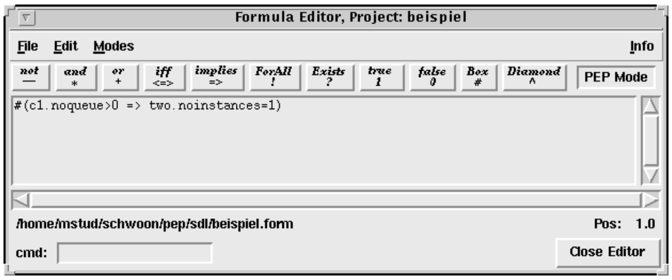

Abbildung 8.7: Der Formeleditor

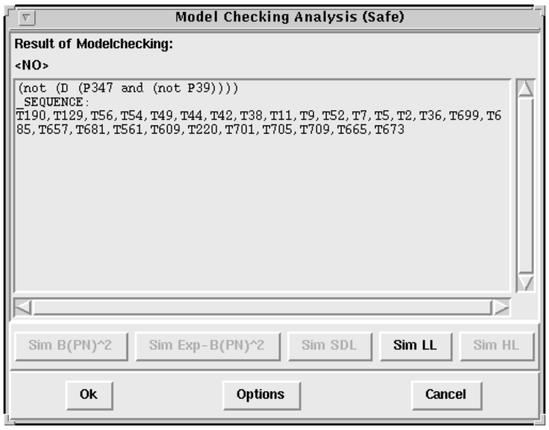

Abbildung 8.8: Fenster für die Ergebnisse des Modelcheckers

- Der Definitionsteil der SDL→HL-Referenzen wird unverändert übernommen.
- Die Entfaltung einer HL-Stelle p, deren Typ α_p ist, führt (wie in [BFF+95] beschrieben) zu einer Menge $\{\, p_a \mid a \in \alpha_p \,\}$ von LL-Stellen, d.h., für jeden Wert in α_p besitzt die Entfaltung eine LL-Stelle. Ist ϱ_p die (SDL→HL-)Referenz von p, so wird $\varrho_p; a$ die (SDL→LL-)Referenz von p_a.
- Alle Transitionen des LL-Netz erhalten die Referenz der HL-Transition, aus der sie hervorgegangen sind.

Das Dateiformat für SDL→LL-Referenzen entspricht dem in Abschnitt 6.2.2 beschriebenen Format von SDL→HL-Referenzen, lediglich die Typbezeichnung im Kopf wird zu `References_SDL_LL` geändert. Die Referenzen werden vom Formeltransformator und von *slist* (siehe nächster Abschnitt) ausgenutzt.

8.3 Erweiterungen an slist

Das Programm *slist*, welches Bestandteil der PEP-Distribution ist, erzeugt zu einem LL-Netz Listen von Stellen, die jeweils eine Pseudo-S-Invariante bilden. Unter einer (Pseudo-) S-Invariante versteht man eine Menge von Stellen, von denen bei Simulation des Netzes stets genau (höchstens) eine Stelle markiert ist [Gra98b]. Diese Informationen erleichtern die Arbeit des Prefix-Builders [ERV96] und anderer Verifikationstools.

Bisher geschah die Erzeugung von S-Listen für $B(PN)^2$-Projekte und basierte auf $B(PN)^2 \rightarrow$LL-Referenzen. Es lag nahe, *slist* für SDL-Projekte zu erweitern. Der Aufruf von *slist* zu diesem Zweck lautet wie folgt:

```
slist -sdl datei.ll_net [-r datei.sdl_ll_ref] [-o datei.slist]
```

Dadurch wird *slist* angewiesen, Invarianten für das Netz `datei.ll_net` zu identifizieren und in die Datei `datei.slist` zu schreiben. Die Feststellung von Invarianten geschieht dabei ausschließlich unter Zuhilfenahme der Referenz-Information aus `datei.sdl_ll_ref`); das tatsächliche Schaltverhalten des Netzes an sich wird nicht weiter untersucht! Jedem Objekt des Quelltexts ist eine Anzahl von Invarianten zugeordnet, z.B. bilden die Stellen des globalen Kontrollflusses eine Invariante, ebenso wie die Stellen im Kontrollfluß eines Prozesses (eine Invariante für jede Instanz), die Wertestellen einer Variablen(instanz) usw. Angewendet auf das Beispiel aus Kapitel 2 kann *slist* 92% der LL-Stellen einer Invariante zuordnen.

Der Aufruf von *slist* geschieht bei Benutzung des PEP-Tools automatisch, d.h., sein Wirken vollzieht sich im wesentlichen unsichtbar für den Benutzer.

8.4 Erweiterungen an simplify

Bei der Berechnung des endlichen Präfix eines LL-Netzes wird eine Liste "toter" Transitionen, d.h. solcher, die niemals schalten können, ausgegeben. Diese Information kann genutzt werden, um das LL-Netz (und gegebenenfalls das zugehörige HL-Netz) zu verkleinern – alle toten Transitionen werden gelöscht, ebenso die daraufhin isolierten Stellen. Dies ist die Aufgabe des Hilfsprogramms *simplify*. Um die korrekte Funktion aller anderen Programme zu gewährleisten, muß *simplify* auch verschiedene Referenzdateien von Einträgen toter Transitionen und Stellen befreien. Die an *simplify* durchgeführten Änderungen betrafen daher die Behandlung von SDL$\rightarrow$HL- und SDL$\rightarrow$LL-Referenzen.

Kapitel 9

Fazit

Dieses Kapitel gibt eine kurze Zusammenfassung der geleisteten Arbeit sowie einen Ausblick auf mögliche Weiterentwicklungen.

In der vorliegenden Arbeit wurde ein Compiler entwickelt, der SDL-Spezifikationen in M-Netze übersetzt. Gleichzeitig wurde die Funktion dieses Compilers, d.h. die verwendete M-Netz-Semantik von SDL, formal beschrieben. Grundlage hierzu war die in [FG98a] geleistete theoretische Vorarbeit; die dort beschriebene M-Netz-Semantik wurde durch diese Arbeit vollständig umgesetzt sowie erweitert und präzisiert. Ziel bei der Entwicklung des Compilers war es, größtmögliche Konformität zum Z.100-Standard [CCI92] herzustellen.

Die genannten Erweiterungen betrafen die Anweisungen im Rumpf von Prozessen bzw. Prozeduren, deren Semantik zuvor nur anhand von Beispielen gegeben war, sowie eine Reihe von Details wie z.B. bidirektionale Kanäle und Signalrouten. Von Grund auf definiert und implementiert wurde die Semantik von Blöcken, Verbindungen, Typ- und Signaldefinitionen. Um Signale mit allen möglichen Argument-Kombinationen korrekt zu behandeln, wurde ein Schema erdacht, wie Signale und ihre Argumente kodiert und in Petri-Netzen repräsentiert werden. Die Angabe der Semantik erfolgte weitestgehend nicht an Beispielen, sondern wurde formal und allgemeingültig gehalten.

Der Compiler übersetzt die meisten wichtigen Sprachkonstrukte von SDL. Der volle Sprachumfang wurde nicht umgesetzt, was angesichts der Größe desselben allerdings auch eine immense Aufgabe wäre. Einige der ausgelassenen Bereiche könnten Bestandteil weiterführender Arbeiten sein (etwa die Implementierung von Services, des Save-Konstrukts, von Timern), andere lassen sich äquivalent durch bereits vorhandene Konstrukte ausdrücken oder sind für das übergeordnete Ziel der Verifikation nicht so sehr interessant (z.B. abstrakte Datentypen).

Die in PEP vorhandenen Algorithmen und Programme bringen einige Einschränkungen mit sich – Netze (auch LL-Netze) müssen eine endliche Größe haben, was den Verzicht auf unendliche Datentypen erzwingt, und auch bei endlichen Netzen stößt man schnell an praktische Grenzen, insbesondere wird die Verifikation von Netzeigenschaften durch den 'state explosion' genannten Effekt behindert. In [FG98a] werden kompositionelle und interaktive Verifikation als Ansätze zur Lösung dieses Problems skizziert; die Unterstützung solcher Verfahren durch den Compiler könnte ebenfalls Gegenstand weiterer Betrachtungen sein.

Indem der Compiler SDL-Programme unterstützt, die mit der Außenwelt des spezifizierten Systems kommunizieren, ist die Möglichkeit gegeben, den Compiler für hybride Programmierung mit Petri-Netzen einzusetzen. Beispielsweise könnten zwei Netze, die aus einer SDL-Spezifikation bzw. aus einem $B(PN)^2$-Programm hervorgegangen sind, parallel nebeneinander gelegt werden und miteinander kommunizieren, wenn sich Kanäle des $B(PN)^2$-Programms mit denen des SDL-Programms synchronisieren. In [Gra98a] wurde bereits vorgeschlagen, SDL und $B(PN)^2$ so zu kombinieren, daß Zugriffe auf die Variablen der jeweils anderen Sprache möglich sind, oder $B(PN)^2$ um die Kommunikationsmechanismen von SDL erweitert wird.

Ein weiterer Teil der Arbeit befaßte sich mit dem Formeltransformator, der eine bequeme Benutzerschnittstelle zu dem von Esparza entwickelten Modelchecker [Esp93, Gra95] bietet. Die Unterstützung anderer Verifikationspakete wie INA [Sta92], SMV [CMCH96, McM93] oder SPIN [Hol90, Hol97] durch den Transformator wäre ebenfalls wünschenswert und kann Inhalt zukünftiger Arbeiten sein. Dazu müßte die Grammatik des Formeltransformators so erweitert werden, daß sie Formeln aus deren Logik verstehen und umsetzen kann.

Erstrebenswert scheint auch die Entwicklung eines graphischen SDL-Editors und dessen Integration in PEP. Der Benutzer könnte dann anstelle des bisherigen textbasierten Editors diesen Editor benutzen, sofern er eine für den Compiler verständliche Textdarstellung liefert. Um Editor und Compiler aufeinander abzustimmen, werden zu gegebener Zeit geeignete Änderungen notwendig sein.

Anhang A

Liste der Syntaxregeln

Hier wird der vom Compiler unterstützte Teil des Sprachumfangs von SDL aufgeführt, indem die Syntaxregeln aus Kapitel 4 nochmals wiederholt werden.

System:

system	::=	system–header system–decl–list system–trailer
system–header	::=	**system** `system–ident` **;**
system–decl	::=	block | channel | signaldef | constdef | typedef
system–trailer	::=	**endsystem** opt–id **;**
opt–id	::=	`ident` | ε

Signale:

signaldef	::=	**signal** sigdef–klist **;**
sigdef	::=	`signal–ident` | `signal–ident` **(** `type–ident`–klist **)**

Konstanten:

constdef	::=	**synonym** synonymdef–klist **;**
synonymdef	::=	`const–ident` opt–id **=** const–expr

Typdefinitionen:

typedef	::=	**syntype** `subtype–ident` **=** `type–ident` opt–default opt–constants **endsyntype** opt–id **;**
opt–default	::=	**default** const–expr | ε
opt–constants	::=	**constants** range–klist | ε
range	::=	const–expr | const–expr **:** const–expr | operator const–expr
operator	::=	**=** | $\neq$ | $<$ | $\leq$ | $>$ | $\geq$

Kanäle:

channel	::=	channel–header ch–path channel–trailer | channel–header ch–path ch–path channel–trailer
channel–header	::=	**channel** `channel–ident`
channel–trailer	::=	**endchannel** opt-id **;**
ch–path	::=	**from** ch-endpoint **to** ch-endpoint **with** `signal–ident`–klist ;
ch–endpoint	::=	`block–ident` | **env**

Blöcke:

block	::=	block–header block–decl–list block–trailer
block–header	::=	**block** `block–ident` **;**
block–trailer	::=	**endblock** opt-id **;**
block–decl	::=	process | signalroute | connect | signaldef | constdef | typedef

Signalrouten:

signalroute	::=	signalroute–header sr–path | signalroute–header sr–path sr–path
signalroute–header	::=	**signalroute** `route–ident`
sr–path	::=	**from** sr–endpoint **to** sr–endpoint **with** `signal–ident`–klist ;
sr–endpoint	::=	`process–ident` | **env**

Verbindungen:

connect	::=	**connect** `channel–ident`–klist **and** `route–ident`–klist **;**

Prozesse:

process	::=	process–header process–formalparms process–decl–list procedure–decls body process–trailer
process–header	::=	**process** `process–ident` **;** opt–instnum
opt–instnum	::=	ε | **(** opt–num **)** | **(** opt–num **,** opt–num **)**
opt–num	::=	const–expr | ε
process–formalparms	::=	**fpar** processparm–decl–klist **;** | ε
processparm–decl	::=	`parm–ident`–klist `type–ident`
process–decl	::=	variable | constdef | typedef
procedure–decls	::=	procedure procedure–decls | ε
process–trailer	::=	**endprocess** opt–id **;**

Prozeduren:

procedure	::=	procedure–header procedure–formalparms procedure–decl–list procedure–decls body procedure–trailer
procedure–header	::=	**procedure** `procedure–ident` **;**
procedure–formalparms	::=	**fpar** procedureparm–decl–klist **;** | ε
procedureparm–decl	::=	parm–type `parm–ident`–klist `type–ident`
parm–type	::=	**in** | **in/out** | ε
procedure–decl–list	::=	process–decl–list
procedure–trailer	::=	**endprocedure** opt–id **;**

Variablen:

variable	::=	**dcl** decl–klist **;**
decl	::=	`variable–ident`–klist `type–ident` | `variable–ident`–klist `type–ident` **:=** const–expr

Rumpf:

body	::=	start states
states	::=	state states | ε
start	::=	**start ;** transition
state	::=	state–header state–body–list
state–header	::=	**state** `state–ident`–klist **;**
state–body	::=	stimulus transition

input:

stimulus	::=	inputpart | spontrans
inputpart	::=	**input** signal–klist **;**
signal	::=	`signal–ident` | `signal–ident` **(** sigarg–klist **)**
sigarg	::=	const–expr | `variable–ident` | ε
spontrans	::=	**input none ;**

Transitionen:

transition	::=	statement transition | ε
statement	::=	label action | action
label	::=	`label–ident` **:**
action	::=	output | task | decision | call | create | terminator
terminator	::=	nextstate | join | stop | return

output:

output	::=	**output** signal–klist output–to output–via **;**
output–to	::=	**to** `process–ident` | **to** `variable–ident` | ε
output–via	::=	**via** `via–ident`–klist | **via all** `via–ident`–klist | ε
`via–ident`	::=	`channel–ident` | `signalroute–ident`

task:

task ::= **task** assign–klist **;**
assign ::= `variable–ident` **:=** expr
expr ::= expr bin–op expr | un–op expr |
(expr) | `variable–ident` | const
const–expr ::= const–expr bin–op const–expr |
un–op const–expr | (const–expr) | const
bin–op ::= $\Rightarrow$ | **or** | **xor** | **and** | **=** | $\neq$ | $>$ |
$\geq$ | $<$ | $\leq$ | **+** | $-$ | ***** | **/** | **mod**
un–op ::= $-$ | **not**
const ::= **True** | **False** | **Null** | z

decision:

decision ::= d–decision | nd–decision
d–decision ::= d–decision–header d–answer–list
opt–else decision–trailer
d–decision–header ::= **decision** expr **;**
d–answer ::= (range–klist) **:** transition
opt–else ::= **else :** transition | ε
decision–trailer ::= **enddecision ;**
nd–decision ::= nd–decision–header nd–answer–list decision–trailer
nd–decision–header ::= **decision any ;**
nd–answer ::= ( ) **:** transition

call:

call ::= **call** `procedure–ident` **;** |
call `procedure–ident` (callparm–klist) **;**
callparm ::= expr | `variable–ident` | ε

create:

create ::= **create** `process–ident` **;** |
create `process–ident` (createparm–klist) **;**
createparm ::= expr | ε

Terminatoren:

nextstate ::= **nextstate** `state–ident` **;** |
nextstate – **;**
join ::= **join** `label–ident` **;**
stop ::= **stop ;**
return ::= **return ;**

Anhang B

Quelltext des Parsers

```
/*****************************************************************************/
/* File sdl.y                                                                */
/* Bison source for SDL parser                                               */
/* 1998 by Stefan Schwoon                                                    */
/*****************************************************************************/

%{
#include "function.h"           /* memory allocation functions */
#include "main.h"               /* important definitions */
#include "symtab.h"             /* functions for handling the symbol table */
#include "node.h"
#include "sortrange.h"          /* predefined types */
#include "actions.h"
#include "ref.h"                /* the reference component */
#include "info.h"               /* for sdl_info files */
#include "property.h"           /* for sdl_prop files */
#include "strbuf.h"             /* dynamically growing string buffer */

/****************************************************************************/

int num_pi_ids;               /* total number of process ids in system */
int yylex();

/* these are used by the rules for processes and procedures */
T_list create_list;
T_list called_procs;
T_list created_procs;

ref_instance tmpref;

/****************************************************************************/

/* indicate parse error */
int yyerror(char *msg)
{
        Msg(ERR+64,msg); /* +64 prevents double incrementation of yynerrs */
        return 1;
}
```

```
/* check if name after END<keyword> matches the name after <keyword> */
void check_end(T_symbol sym, T_ident ident, char *keyword)
{
        if (!ident || ident->sym == sym) return;
        Msg(ERR,"named END%s does not match start",keyword);
}


void copy_ref_start(T_ref dst, T_ref src)
{
        dst->start_l = src->start_l;
        dst->start_c = src->start_c;
}


void copy_ref_end(T_ref dst, T_ref src)
{
        dst->end_l = src->end_l;
        dst->end_c = src->end_c;
}


/* access sdl_info database to retrieve information
   about a certain channel                       */
void find_channel_props(T_relay relay, char *name)
{
        initbuf(); addbuf(name); addbuf(".capacity");
        relay->capacity = prop_find_key(strbuf,sdl_defcap);

        initbuf(); addbuf(name); addbuf(".version");
        relay->version = prop_find_key(strbuf,sdl_chanver);
}

%}

%union {
        T_symbol symbol;        /* entry in the symbol table */
        T_ident idptr;
        T_list list;            /* generic list type */
        T_transition trans;     /* list of actions */
        T_action action;
        T_syntype type;         /* user-defined type */
        T_node node;            /* used for expressions */
        intset set;             /* set of integers */
        void* ptr;              /* generic pointer */
        ident_instance ident;
        ref_instance ref;       /* reference */
        int value;
}

/* most tokens are annotated with a reference to their
   occurence in the source code                        */
%token <ref> SYSTEM ENDSYSTEM
%token <ref> BLOCK ENDBLOCK
%token <ref> CHANNEL ENDCHANNEL
```

```
%token <ref> FROM TO WITH
%token <ref> SIGNAL
%token <ref> SIGNALROUTE
%token <ref> CONNECT
%token <ref> PROCESS ENDPROCESS
%token <ref> START STATE INPUT
%token <ref> OUTPUT
%token <ref> NEXTSTATE STOP
%token <ref> VIA ALL
%token <ref> SYNONYM SYNTYPE ENDSYNTYPE
%token <ref> DEFAULT CONSTANTS
%token <ref> DCL
%token <ref> TRUE FALSE NULL_PID
%token <ref> TASK ASSIGN
%token <ref> IMPLIES NE GE LE
%token <ref> OR XOR AND MOD NOT
%token <ref> DECISION ANY ELSE ENDDECISION
%token <ref> PROCEDURE ENDPROCEDURE CALL RETURN
%token <ref> FPAR IN OUT
%token <ref> CREATE
%token <ref> INPUT_NONE
%token <ref> JOIN

%token <value> CONST
%token <ident> IDENT
%token <ref> '(' ';' ')' ',' ':'


/* priorities of operators in expressions */
%left IMPLIES
%left OR XOR
%left AND
%nonassoc '=' NE '>' GE '<' LE
%left '+' '-'
%left '*' '/' MOD
%right NOT unaryminus


%type <symbol> system syshead
%type <list> sysbodylist sysrefresolvelist
%type <symbol> sysbodyentity sysrefresolveentity

%type <symbol> block blkhead
%type <list> blkbodylist
%type <symbol> blkbodyentity

%type <symbol> channel
%type <symbol> signalroute
%type <ptr> path

%type <symbol> sigdef singlesigdef
%type <list> sigdeflist sortlist

%type <symbol> connect relayreferident relaydeclident
%type <list> relayreferlist relaydecllist
```

```
%type <idptr> optname
%type <list> siglist
%type <symbol> sigident

%type <symbol> data syndef syntypedef
%type <symbol> syndeflist singlesyndef
%type <value> optdefault const range
%type <value> negconst sigconst
%type <list> optconstants rangelist noteq
%type <type> osyntype syntype

%type <symbol> dcl varident
%type <list> dcllist singledcl varlist

%type <symbol> process prcshead body
%type <value> oinstno onum
%type <list> prcsdcllist
%type <symbol> prcsdclentity

%type <symbol> procedure prcdhead
%type <list> prcddcllist
%type <symbol> prcddclentity
%type <list> prcdfmlparms prcdparmlist singleparmdcl
%type <value> parmtype

%type <trans> start trigger inputpart
%type <list> statespeclist statespec statelist
%type <symbol> stateident
%type <list> triggerlist
%type <list> term_transition transition tlist

%type <list> usiglist sigarglist
%type <ptr> singleusig singlesigarg

%type <action> stmt action
%type <action> output task
%type <action> decision call create
%type <action> terminator nextstate
%type <action> stop stoporreturn join
%type <symbol> label

%type <symbol> toclause
%type <ptr> viaclause
%type <list> vialist
%type <symbol> viaident

%type <list> assignlist
%type <ptr> assignment
%type <value> constexpr
%type <node> expr

%type <node> question
%type <list> answerlist answer
%type <list> optrangelist

%type <list> optparms actparmlist
```

```
%start system

%%

/****************************************************************************/
/* rules start here                                                         */
/****************************************************************************/

/* General remarks:
   Handling the symbol table entries is a good deal more complicated than
   in BPN2 since entries can be referenced before they are actually defined
   or even declared.
   The parser uses these routines (defined in symtab.c) to access information
   in the symbol table:

        define_ident: Treat this occurence of a symbol as the defining
                      occurence. A new symbol is created, carrying a given
                      identifier and type. The current level of the
                      specification may not contain another definition of
                      the same identifier, or a declaration of this identifier
                      with another type.

        decl_ident:   Treat this occurence as a declaration. If no declaration
                      or definition of this identifier is present on the
                      current level, a new symbol is created. Otherwise,
                      the previous declaration/definition must refer to an
                      identifier of the same type as in this declaration.
                      All declarations must be resolved by a definition
                      on the same level of the specification.

        refer_ident:  Treat this occurence as a reference to a symbol that
                      must already be declared (with correct type) in either
                      the current or any enclosing structure.
*/

/**********/
/* SYSTEM */
/**********/

system  : syshead
          sysbodylist
          ENDSYSTEM optname ';'
          {
                T_symbol sym;
                T_structure this = (STree = $$ = $1)->e.struc;
                check_end($$,$4,"SYSTEM");
                revert_list($2);
                this->subtype = TYPE_SYSTEM;
                this->entities = $2;
                this->symtab = dec_blknum();
                this->num_connects = 0;
          }

syshead : SYSTEM IDENT ';'
          {
```

```
                $$ = define_ident(&$2,TYPE_STRUCTURE,alloc_structure());
                inc_blknum();

                /* create ‘env’ as a pseudo-entry in the symbol table
                   so that channels can acces it */
                s_lookup("env",&$2);

                define_ident(&$2,TYPE_STRUCTURE,alloc_structure());
                init_default_types();
                num_pi_ids = 2;       /* NULL and 0 */
                 create_list = alloc_list();;

                /* enable global procedures to access implicit variables
                   if local procedure copies are activated (default)    */
                if (!(sdlflag & SDLF_NO_PRCD_COPIES)) init_implicit_vars();

                $$->ref = add_ref($$,REF_SYSTEM,&$1,&$3);
          }

sysbodylist
        : sysbodyentity sysbodylist
          {       list_add($$ = $2, $1);      }
        | {       $$ = alloc_list();          }

sysbodyentity
        : block
        | channel
        | sigdef
        | data
        | procedure

/*********/
/* BLOCK */
/*********/

block   : blkhead
          blkbodylist
          ENDBLOCK optname ';'
          {
                T_structure this = ($$ = $1)->e.struc;
                check_end($$,$4,"BLOCK");
                this->subtype = TYPE_BLOCK;
                revert_list(this->entities = $2);
                this->symtab = dec_blknum();
                /* number of connections inside this block */
                this->num_connects = 0;
          }

blkhead : BLOCK IDENT ';'
          {
                $$ = define_ident(&$2,TYPE_STRUCTURE,alloc_structure());
                inc_blknum();

                /* create ‘env’ as a pseudo-entry in the symbol table
                   so that signalroutes can acces it */
                s_lookup("env",&$2);
```

```
                define_ident(&$2,TYPE_STRUCTURE,alloc_structure());
                $$->ref = add_ref($$,REF_BLOCK,&$1,&$3);
          }

blkbodylist
        : blkbodyentity blkbodylist
          {       list_add($$ = $2, $1);        }
        | {       $$ = alloc_list();       }

blkbodyentity
        : process
        | connect
        | signalroute
        | sigdef
        | data
        | procedure

/***********/
/* CHANNEL */
/***********/

channel : CHANNEL IDENT                /* unidirectional */
          path ENDCHANNEL optname ';'
          {
                T_relay this = (T_relay) $3;
                $$ = define_ident(&$2,TYPE_RELAY,this);
                this->bidirectional = False;
                this->is_channel = True;

                find_channel_props(this,$$->name);

                check_end($$,$5,"CHANNEL");
                $$->ref = add_ref($$,REF_CHANNEL,&$1,&$6);
          }
        | CHANNEL IDENT                /* bidirectional */
          path path
          ENDCHANNEL optname ';'
          {
                T_relay this = (T_relay) $3;
                $$ = define_ident(&$2,TYPE_RELAY,this);
                this->bidirectional = True;
                this->is_channel = True;

                find_channel_props(this,$$->name);

                if (this->ep1 != ((T_relay) $4)->ep2 ||
                    this->ep2 != ((T_relay) $4)->ep1)
                        Msg(ERR,"Endpoints of bidirectional channel "
                                "must be the same in both ways");
                this->siglist2 = ((T_relay) $4)->siglist1;
                free($4);
                check_end($$,$6,"CHANNEL");
                $$->ref = add_ref($$,REF_CHANNEL,&$1,&$7);
                add_ref($$,REF_CHANNEL,&$1,&$7);
          }
```

```
/***************/
/* SIGNALROUTE */
/***************/

/* Signalroutes are treated almost like channels, but they
   have is_channel set to False                         */

signalroute
        : SIGNALROUTE IDENT path            /* unidirectional */
          {
                T_relay this = (T_relay) $3;
                $$ = define_ident(&$2,TYPE_RELAY,this);
                this->bidirectional = False;
                this->is_channel = False;
                $$->ref = add_ref($$,REF_SIGNALROUTE,&$1,&tmpref);
          }
        | SIGNALROUTE IDENT path path       /* bidirectional */
          {
                T_relay this = (T_relay) $3;
                $$ = define_ident(&$2,TYPE_RELAY,this);
                this->bidirectional = True;
                this->is_channel = False;
                if (this->ep1 != ((T_relay) $4)->ep2 ||
                    this->ep2 != ((T_relay) $4)->ep1)
                        Msg(ERR,"Endpoints of bidirectional signalroute "
                                "must be the same in both ways");
                this->siglist2 = ((T_relay) $4)->siglist1;
                free($4);
                $$->ref = add_ref($$,REF_SIGNALROUTE,&$1,&tmpref);
                add_ref($$,REF_SIGNALROUTE,&$1,&tmpref);
          }

/**********/
/* SIGNAL */
/**********/

sigdef  : SIGNAL sigdeflist ';'
          {
                /* Only one entry is stored in the entity list of the
                   surrounding structure (system, block).
                   This entry contains a placeholder name as its identifier. */
                T_ident dummy = s_dummy("$$Siglist");
                $$ = define_ident(dummy,TYPE_SIGDEF,$2);
                revert_list($2);
          }

sigdeflist:
          singlesigdef
          {       list_add($$ = alloc_list(), $1);        }
        | singlesigdef ',' sigdeflist
          {       list_add($$ = $3,$1);                   }

singlesigdef : IDENT
          {
                T_signal sig = alloc_signal();
```

```
                sig->args = NULL;
                $$ = define_ident(&$1,TYPE_SIGNAL,sig);
                $$->ref = add_ref($$,REF_SIGNAL,&$1.ref,&$1.ref);
          }
        | IDENT '(' sortlist ')'
          {
                T_syntype st;
                T_signal sig = alloc_signal();

                /* Are there any unbounded argument types like
                   Integer or Natural?                        */
                for_list ($3,st,T_syntype)
                        if (!st->set) Msg(FAT,"cannot use unlimited sort");

                sig->args = $3;
                $$ = define_ident(&$1,TYPE_SIGNAL,sig);
                $$->ref = add_ref($$,REF_SIGNAL,&$1.ref,&$4);
          }

sortlist: syntype
          { list_add($$ = alloc_list(), $1);        }
        | syntype ',' sortlist
          {       list_add($$ = $3,$1);             }

/***********/
/* CONNECT */
/***********/

/* Since connect instructions can occur before the definition of the
   signalroutes they reference, we need to handle signalroutes differently,
   i.e., they will be defined whereas channels are referenced.
*/

connect : CONNECT relayreferlist AND relaydecllist ';'
          {
                T_ident dummy = s_dummy("$$Connect");
                T_connect this = alloc_connect();
                $$ = define_ident(dummy,TYPE_CONNECT,this);
                this->outlist = $2;
                this->inlist = $4;
                $$->ref = add_ref($$,REF_CONNECT,&$1,&$5);
                add_ref($$,REF_CONNECT,&$1,&$5);
          }

relayreferlist: relayreferident
          {       list_add($$ = alloc_list(), $1);         }
        | relayreferident ',' relayreferlist
          {       list_add($$ = $3,$1);                    }

relayreferident: IDENT
          {       $$ = refer_ident(&$1,TYPE_RELAY);        }

relaydecllist: relaydeclident
          {       list_add($$ = alloc_list(), $1);         }
        | relaydeclident ',' relaydecllist
          {       list_add($$ = $3,$1);                    }
```

```
relaydeclident: IDENT
          {       $$ = decl_ident(&$1,TYPE_RELAY);         }

/***********************/
/* general definitions */
/***********************/

optname : IDENT
          {
                $$ = alloc_ident();
                memcpy($$,&$1,sizeof(ident_instance));
          }
        | {       $$ = NULL;        }

path    : FROM IDENT TO IDENT WITH siglist ';'
          {
                if ($2.sym == $4.sym)
                        Msg(ERR, "endpoints of channel/signalroute "
                                 "must be different");
                $$ = alloc_relay();
                ((T_relay) $$)->ep1 = decl_ident(&$2,TYPE_STRUCTURE);
                ((T_relay) $$)->ep2 = decl_ident(&$4,TYPE_STRUCTURE);
                ((T_relay) $$)->siglist1 = $6;
                copy_ref_end(&tmpref,&$7);
          }

siglist : sigident
          {       list_add($$ = alloc_list(), $1);        }
        | sigident ',' siglist
          {       list_add($$ = $3,$1);                   }

sigident: IDENT
          {       $$ = refer_ident(&$1,TYPE_SIGNAL);      }

/********/
/* DATA */
/********/

data    : syndef
        | syntypedef;

syndef  : SYNONYM syndeflist ';'
          {
                T_ident dummy = s_dummy("$$Synonym");
                $$ = define_ident(dummy,TYPE_CONST,(void*)1);
          }

syndeflist:
          singlesyndef
        | singlesyndef ',' syndeflist

singlesyndef: IDENT osyntype '=' constexpr
          {
                $$ = define_ident(&$1,TYPE_CONST,(void*)$4);
                check_sort($2,$4);
```

```
          }

syntypedef
        : SYNTYPE IDENT '=' syntype
          optdefault optconstants
          ENDSYNTYPE optname ';'
          {
                T_syntype syn = alloc_syntype();

                    $$ = define_ident(&$2,TYPE_SYNTYPE,syn);
                syn->def_init = $4->def_init;
                syn->set = $4->set;
                syn->range = $4->range;

                /* intersect optconstants with range of parent type */
                if ($6)
                {
                        syn->range = intersect_range($6,syn->range,True);
                        free_list($6);
                        if (range_isfinite(syn->range))
                                syn->set = create_set(syn->range);
                }

                if ($5 != NONE) syn->def_init = $5;
                if (syn->def_init != NONE) check_sort(syn,syn->def_init);
                check_end($$,$8,"SYNTYPE");
          }

optdefault
        : DEFAULT constexpr /* oend */
          {       $$ = $2;          }
        | {       $$ = NONE;        }

optconstants
        : CONSTANTS rangelist
          {       $$ = $2;          }
        | {       $$ = NULL;        }

rangelist: range
          {       list_add($$ = alloc_list(), $1);        }
        | range ',' rangelist
          {       list_add($$ = $3,$1);                   }
        | noteq
        | noteq ',' rangelist
          {
                int r;
                for_list ($1,r,int) list_add($3,r);
                free_list($1);
                $$ = $3;
          }

range   : constexpr
          {       $$ = create_range($1,$1);                }
        | constexpr ':' constexpr
          {       $$ = create_range($1,$3);                }
        | '=' constexpr
```

```
          {       $$ = create_range($2,$2);                }
        | '>' constexpr
          {       $$ = create_range($2+1,MAX_POS);         }
        | GE constexpr
          {       $$ = create_range($2,MAX_POS);           }
        | '<' constexpr
          {       $$ = create_range(MIN_NEG,$2-1);         }
        | LE constexpr
          {       $$ = create_range(MIN_NEG,$2);           }

noteq   : NE constexpr
          {
                $$ = alloc_list();
                list_add($$,create_range(MIN_NEG,$2-1));
                list_add($$,create_range($2+1,MAX_POS));
          }

osyntype: syntype
        | {       $$ = NULL;        }

/* As an extension of native SDL syntax, one could allow the
   specification of types {0,1} here.                       */

syntype : IDENT
          {       $$ = refer_ident(&$1,TYPE_SYNTYPE)->e.syntype;        }

const   : CONST
        | TRUE
          {       $$ = TT;        }
        | FALSE
          {       $$ = FF;        }
        | NULL_PID
          {       $$ = DOT;       }

negconst: '-' CONST
          {
                if (($$ = -$2) < MIN_NEG) Msg(FAT,"constant too low - "
                                                  "you trying to break me?");
          }

/*******/
/* DCL */
/*******/

dcl     : DCL dcllist ';'
          {
                T_ident dummy = s_dummy("$$Dcllist");
                $$ = define_ident(dummy,TYPE_VARDEF,$2);
                revert_list($2);
          }

dcllist : singledcl
        | singledcl ',' dcllist
          {
                T_symbol sym;
                for_list ($1,sym,T_symbol) list_add($3,sym);
```

```
            free_list($1);
            $$ = $3;
        }

singledcl
        : varlist syntype
        {
            T_symbol sym;
            if (!$2->set) Msg(FAT,"cannot use unlimited sort");
            for_list ($1,sym,T_symbol) sym->e.syntype = $2;
            $$ = $1;
        }
        | varlist syntype ASSIGN constexpr
        {
            T_symbol sym;
            T_syntype syn = alloc_syntype();

            if (!$2->set) Msg(FAT,"cannot use unlimited sort");
            syn->set = $2->set;
            syn->def_init = $4;
            syn->range = $2->range;
            check_sort(syn,syn->def_init);

            for_list ($1,sym,T_symbol) sym->e.syntype = syn;
            $$ = $1;
        }

varlist : varident
        {       list_add($$ = alloc_list(), $1);        }
        | varident ',' varlist
        {       list_add($$ = $3,$1);                   }

varident: IDENT
        {
            $$ = define_ident(&$1,TYPE_VARIABLE,NULL);
            $$->ref = add_ref($$,REF_VARIABLE,&$1.ref,&$1.ref);
            $$->ref->end_c++;       /* for text editor */
        }

/***********/
/* PROCESS */
/***********/

/* process uses prcdfmlparms (from the procedure rules) to avoid duplicating
   the definitions. in/out-Parameters are thrown out later.
*/

process : prcshead
          prcdfmlparms
          prcsdcllist
          body
          ENDPROCESS optname ';'
        {
            T_structure this = ($$ = $1)->e.struc;
            T_prcsinf prcsinf = alloc_prcsinf();
            T_fmlparm parm;
```

```
T_symbol sym;

check_end($$,$6,"PROCESS");
prcsinf->initial = lower_bound(this->num_connects);
prcsinf->maximum = upper_bound(this->num_connects);
prcsinf->num_states = make_states();
prcsinf->called_procs = called_procs;
prcsinf->fmlparms = $2;

/* check sdl_info file for information */
initbuf(); addbuf($$->name); addbuf(".capacity");
prcsinf->capacity = prop_find_key(strbuf,sdl_inqcap);

initbuf(); addbuf($$->name); addbuf(".version");
prcsinf->version = prop_find_key(strbuf,sdl_procver);

for_list ($2,parm,T_fmlparm) if (parm->type)
        Msg(ERR,"process cannot have in/out parameters");
this->num_connects = 0;
this->entities = $3;
list_add(this->entities,$4);
this->prcsinf = prcsinf;
revert_list(this->entities);
this->symtab = dec_blknum();

/* increase total number of process ids */
num_pi_ids += prcsinf->maximum;

add_state_info($$);

/* create requests are complicated because you can refer
   to processes that have not yet been defined. created_procs
   collects a list of (temporary) symbol table entries
   containing one such entry for every create request being
   made in the current procedure. At this point, this list
   is evaluated, and all processes are re-declared on the
   block level.                                            */

for_list (created_procs,sym,T_symbol)
{
        T_createreq cr = alloc_createreq();
        T_ident id = alloc_ident();

        cr->parent = $$;
        cr->child = sym;
        if (sym->type == TYPE_UNDECLARED)
        {
                /* undo name change made in create rule */
                *(sym->name) &= ~0x80;

                s_lookup(sym->name,id);
                cr->child = decl_ident(id,TYPE_STRUCTURE);
                ((T_call) sym->e.data)->sym = cr->child;
        }
        list_add(create_list,cr);
}
```

```
                free_list(created_procs);
          }

prcshead: PROCESS IDENT oinstno ';'
          /* end oprcsfmlprms oinpset oprcsdcllst */
          {
                int initial = lower_bound($3);
                int maximum = upper_bound($3);

                initbuf();
                addbuf($2.sym->name);
                addbuf(".instances");
                maximum = prop_find_key(strbuf,
                        (maximum == NONE)? sdl_prcsinst : maximum);
                $3 = create_range(initial,maximum);

                $$ = define_ident(&$2,TYPE_STRUCTURE,alloc_structure());
                $$->e.struc->num_connects = $3;

                $$->e.struc->subtype = TYPE_PROCESS;
                called_procs = alloc_list();

                /* initialize list of create requests in this process */
                created_procs = alloc_list();

                inc_blknum();

                /* create implicit variables in the symbol table
                   unless they were already created by the system rule */
                if (sdlflag & SDLF_NO_PRCD_COPIES) init_implicit_vars();

                if (initial > maximum)
                        Msg(ERR,"initial number larger than maximum");
                if (initial < 0 || maximum <= 0)
                        Msg(ERR,"number of instances is broken");

                $$->ref = add_ref($$,REF_PROCESS,&$1,&$4);
          }


/* initial and maximum number of instances are hidden
   as upper and lower bounds of a range type                */

oinstno : '(' onum ')'
          {
                $$ = create_range(($2 == NONE)? 1 : $2, NONE);
          }
        | '(' onum ',' onum ')'
          {
                $$ = create_range(($2 == NONE)? 1 : $2, $4);
          }
        | {
                $$ = create_range(1,NONE);
          }

onum    : constexpr
```

```
        | {       $$ = NONE;       }

prcsdcllist
        : prcsdclentity prcsdcllist
          {       list_add($$ = $2, $1);        }
        | {       $$ = alloc_list();            }

prcsdclentity
        : data
        | dcl
        | procedure

/*************/
/* PROCEDURE */
/*************/

procedure:prcdhead
          prcddcllist
          body
          ENDPROCEDURE optname ';'
          {
                T_structure this = ($$ = $1)->e.struc;

                check_end($$,$5,"PROCEDURE");
                this->prcsinf->num_states = make_states();
                this->prcsinf->initial = 0;

                initbuf();
                addbuf($$->name);
                addbuf(".instances");
                this->prcsinf->maximum = prop_find_key(strbuf,sdl_prcdinst);

                this->entities = $2;
                list_add(this->entities,$3);
                revert_list(this->entities);
                this->symtab = dec_blknum();
                add_state_info($$);
          }

prcdhead: PROCEDURE IDENT ';'
          {
                $$ = define_ident(&$2,TYPE_PROCEDURE,alloc_structure());
                $$->ref = add_ref($$,REF_PROCEDURE,&$1,&$3);
                $<symbol>1 = $$;
                inc_blknum();
          }
          prcdfmlparms
          {
                revert_list($5);
                $$ = $<symbol>1;
                $$->e.struc->prcsinf = alloc_prcsinf();
                $$->e.struc->prcsinf->fmlparms = $5;
          }

prcddcllist
        : prcddclentity prcddcllist
```

```
          {       list_add($$ = $2, $1);       }
        | {       $$ = alloc_list();           }

prcddclentity
        : data
        | dcl
        | procedure

prcdfmlparms
        : FPAR prcdparmlist ';'
          {       $$ = $2;                 }
        | {       $$ = alloc_list();       }

prcdparmlist
        : singleparmdcl
        | singleparmdcl ',' prcdparmlist
          {
                T_fmlparm parm;
                for_list ($1,parm,T_fmlparm) list_add($3,parm);
                free_list($1);
                $$ = $3;
          }

singleparmdcl
        : parmtype varlist syntype
          {
                T_symbol sym;
                if (!$3->set) Msg(FAT,"cannot use unlimited sort");
                $$ = alloc_list();
                for_list ($2,sym,T_symbol)
                {
                        T_fmlparm parm = alloc_fmlparm();
                        parm->type = $1;
                        parm->sym = sym;
                        sym->e.syntype = $3;
                        sym->ref->type = $1? REF_IN_OUT_PARM : REF_IN_PARM;
                        list_add($$,parm);
                }
                free_list($2);
          }

parmtype: IN
          {       $$ = False;       }
        | IN '/' OUT
          {       $$ = True;        }
        | {       $$ = False;       }

/****************/
/* process body */
/****************/

body    : start statespeclist
          {
                /* create a dummy entry in the entity list of the process
                   or procedure (annotated with the list of transitions) */
                T_ident dummy = s_dummy("$$PrcsBody");
```

```
                $$ = define_ident(dummy,TYPE_BODY,$2);
                list_add($2,$1);
                revert_list($2);
          }

start   : START ';' term_transition
          {
                $$ = alloc_transition();
                $$->states = NULL;
                $$->signals = NULL;
                $$->actions = $3;
          }

statespeclist
        : statespec statespeclist
          {
                T_transition trans;
                for_list ($1,trans,T_transition) list_add($2,trans);
                free_list($1);
                $$ = $2;
          }
        | {       $$ = alloc_list();       }

statespec:
          STATE statelist ';' triggerlist
          {
                T_transition trans;
                for_list ($4,trans,T_transition) trans->states = $2;
                $$ = $4;
          }

statelist: stateident
          {       list_add($$ = alloc_list(), $1);        }
        | stateident ',' statelist
          {       list_add($$ = $3,$1);                   }

stateident: IDENT
          {       $$ = decl_ident(&$1,TYPE_STATE);        }

triggerlist
        : trigger triggerlist
          {       list_add($$ = $2, $1);        }
        | {       $$ = alloc_list();            }

trigger : inputpart

/*********/
/* INPUT */
/*********/

inputpart: INPUT usiglist ';' term_transition
          {
                $$ = alloc_transition();
                $$->signals = $2;
                $$->actions = $4;
                $$->ref = add_ref($$,REF_TRIGGER,&$1,&$3);
```

```
          }

usiglist: singleusig
          {       list_add($$ = alloc_list(), $1);        }
        | singleusig ',' usiglist
          {       list_add($$ = $3,$1);                   }

singleusig
        : IDENT                /* signal without arguments */
          {
                T_signaluse sig = alloc_signaluse();
                sig->sym = refer_ident(&$1,TYPE_SIGNAL);
                if (sig->sym->e.signal->args)
                        Msg(ERR,"incorrect number of arguments to signal");
                sig->parm = NULL;
                $$ = sig;
          }
        | IDENT '(' sigarglist ')'
          {
                T_signaluse sig = alloc_signaluse();
                T_parameter parm;
                T_list args;
                int i = 0;

                /* check number of arguments */
                sig->sym = refer_ident(&$1,TYPE_SIGNAL);
                if (!(args = sig->sym->e.signal->args)
                                || args->count != $3->count)
                        Msg(ERR,"incorrect number of arguments to signal");
                sig->parm = MYmalloc(args->count * sizeof(parameter_instance));

                /* check type of arguments */
                for_list ($3,parm,T_parameter)
                {
                        T_syntype reqtype = list_elem(args,i,T_syntype);
                        if (parm->flag == PARM_VARIABLE)
                                check_subset(reqtype->set,
                                                parm->var->e.syntype->set);
                        else if (parm->flag == PARM_CONST)
                                check_sort(reqtype,parm->value);
                        memcpy(sig->parm+(i++),parm,sizeof(parameter_instance));
                        free(parm);
                }
                free_list($3);
                $$ = sig;
          }
        | INPUT_NONE        /* spontaneous transition */
          {
                $$ = NULL;
          }

sigarglist
        : singlesigarg
          {       list_add($$ = alloc_list(), $1);          }
        | singlesigarg ',' sigarglist
          {       list_add($$ = $3,$1);                     }
```

```
singlesigarg
        : sigconst        /* In order to avoid r/r conflicts sigconst is used.
                           Normally, this should be constexpr.              */
          {
                T_parameter parm = alloc_parameter();
                parm->flag = PARM_CONST;
                parm->value = $1;
                $$ = parm;
          }
        | IDENT
          {
                T_parameter parm = alloc_parameter();
                parm->flag = PARM_VARIABLE;
                parm->var = refer_ident(&$1,TYPE_VARIABLE);
                $$ = parm;
          }
        | {
                T_parameter parm = alloc_parameter();
                parm->flag = PARM_UNUSED;
                $$ = parm;
          }

sigconst: const
        | negconst;

/**************/
/* TRANSITION */
/**************/

/* term_transition is a transition with exactly one terminating action
   at the end of its list of actions.
*/

term_transition
        : transition
          {
                T_list l = $$ = $1;
                if (!l->count || !list_elem(l,l->count-1,T_action)->terminating)
                        Msg(ERR,"transition is not terminated");
          }

transition: tlist
          {
                T_action act;
                Boolean term = False;

                revert_list($$ = $1);
                for_list ($$,act,T_action)
                {
                        if (term) Msg(ERR,"terminating statement before "
                                          "end of transition");
                        term = act->terminating;
                }
          }
```

```
tlist   : stmt tlist
          {       list_add($$ = $2, $1);            }
        | {       $$ = alloc_list();                }

stmt    : label action
          {       ($$ = $2)->label = $1;          }
        | action
          {       ($$ = $1)->label = NULL;        }

action  : output
        | task
        | decision
        | call
        | create
        | terminator

label   : IDENT ':'
          {
                $$ = define_ident(&$1,TYPE_LABEL,(void*)1);
                $$->ref = add_ref($$,REF_LABEL,&$1.ref,&$2);
          }

/**********/
/* OUTPUT */
/**********/

output  : OUTPUT usiglist toclause viaclause ';'
          {
                T_output out;
                $$ = alloc_action();
                $$->type = ACTION_OUTPUT;
                $$->terminating = False;
                out = $$->e.output = (T_output) $4;
                out->signals = $2;
                out->dst = $3;
                $$->ref = add_ref($$,REF_ACTION,&$1,&$5);
          }

toclause: TO IDENT
          {
                if ($2.sym->type == TYPE_VARIABLE
                                && $2.sym->e.syntype->set == SORT_PID)
                        $$ = refer_ident(&$2,TYPE_VARIABLE);
                else
                        $$ = refer_offscope(&$2,TYPE_STRUCTURE);
          }
        | {       $$ = NULL;        }

viaclause
        : VIA vialist
          {
                T_output out = alloc_output();
                out->multicast = False;
                out->via = $2;
                $$ = out;
          }
```

```
        | VIA ALL vialist
          {
                T_output out = alloc_output();
                out->multicast = True;
                out->via = $3;
                $$ = out;
          }
        | {
                T_output out = alloc_output();
                out->multicast = False;
                out->via = alloc_list();
                $$ = out;
          }

vialist : viaident
          {       list_add($$ = alloc_list(), $1);          }
        | viaident ',' vialist
          {       list_add($$ = $3,$1);                      }

viaident: IDENT
          {       $$ = refer_offscope(&$1,TYPE_RELAY);       }

/********/
/* TASK */
/********/

task    : TASK assignlist ';'
          {
                $$ = alloc_action();
                $$->type = ACTION_TASK;
                $$->terminating = False;
                $$->e.list = $2;
                $$->ref = NULL;
                copy_ref_end(list_elem($2,$2->count-1,T_assignment)->ref,&$3);
                revert_list($2);
                copy_ref_start(list_elem($2,0,T_assignment)->ref,&$1);
          }

assignlist
        : assignment
          {
                list_add($$ = alloc_list(), $1);
          }
        | assignment ',' assignlist
          {
                list_add($$ = $3,$1);
                copy_ref_end(((T_assignment)$1)->ref,&$2);
          }

assignment
        : IDENT ASSIGN expr
          {
                T_assignment assign = alloc_assignment();
                intset type;

                assign->var = refer_ident(&$1,TYPE_VARIABLE);
```

```
                type = assign->var->e.syntype->set;
                assign->expr = $3;
                $$ = assign;

                /* check types of IDENT and expr */
                assign->ref = add_ref($$,REF_ACTION,&$1.ref,&$2);
                if ($3->sort == ETYPE_INTEGER
                                && type != SORT_PID && type != boolset) break;
                if ($3->sort == ETYPE_BOOLEAN && type == boolset) break;
                if ($3->sort == ETYPE_PID && type == SORT_PID) break;
                Msg(ERR,"type mismatch in assignment");
          }

/********/
/* EXPR */
/********/

constexpr: expr
          {
                if (!$1->constant) Msg(ERR,"expression is not constant");
                $$ = eval_node($1);
                free_node($1);
          }

expr    : expr IMPLIES expr
          { $$ = create_node(NODE_IMPLIES,$1,$3,ETYPE_BOOLEAN,ETYPE_BOOLEAN); }
        | expr OR expr
          { $$ = create_node(NODE_OR,$1,$3,ETYPE_BOOLEAN,ETYPE_BOOLEAN); }
        | expr XOR expr
          { $$ = create_node(NODE_XOR,$1,$3,ETYPE_BOOLEAN,ETYPE_BOOLEAN); }
        | expr AND expr
          { $$ = create_node(NODE_AND,$1,$3,ETYPE_BOOLEAN,ETYPE_BOOLEAN); }
        | expr '=' expr
          { $$ = create_node(NODE_EQ,$1,$3,ETYPE_ANY,ETYPE_BOOLEAN); }
        | expr NE expr
          { $$ = create_node(NODE_NE,$1,$3,ETYPE_ANY,ETYPE_BOOLEAN); }
        | expr '>' expr
          { $$ = create_node(NODE_GT,$1,$3,ETYPE_INTEGER,ETYPE_BOOLEAN); }
        | expr GE expr
          { $$ = create_node(NODE_GE,$1,$3,ETYPE_INTEGER,ETYPE_BOOLEAN); }
        | expr '<' expr
          { $$ = create_node(NODE_LT,$1,$3,ETYPE_INTEGER,ETYPE_BOOLEAN); }
        | expr LE expr
          { $$ = create_node(NODE_LE,$1,$3,ETYPE_INTEGER,ETYPE_BOOLEAN); }
        | expr '+' expr
          { $$ = create_node(NODE_PLUS,$1,$3,ETYPE_INTEGER,ETYPE_INTEGER); }
        | expr '-' expr
          { $$ = create_node(NODE_MINUS,$1,$3,ETYPE_INTEGER,ETYPE_INTEGER); }
        | expr '*' expr
          { $$ = create_node(NODE_MULT,$1,$3,ETYPE_INTEGER,ETYPE_INTEGER); }
        | expr '/' expr
          { $$ = create_node(NODE_DIV,$1,$3,ETYPE_INTEGER,ETYPE_INTEGER); }
        | expr MOD expr
          { $$ = create_node(NODE_MOD,$1,$3,ETYPE_INTEGER,ETYPE_INTEGER); }
        | '-' expr %prec unaryminus
          { $$ = create_node(NODE_NEG,$2,NULL,ETYPE_INTEGER,ETYPE_INTEGER); }
```

```
        | NOT expr
          { $$ = create_node(NODE_NOT,$2,NULL,ETYPE_BOOLEAN,ETYPE_BOOLEAN); }
        | '(' expr ')'
          { $$ = create_node(NODE_IMPLIES,$2,NULL,ETYPE_ANY,ETYPE_ANY); }
        | IDENT
          {
                intset set;
                $$ = alloc_node();
                $$->type = NODE_VAR;
                $$->constant = False;
                $$->op1 = (T_node) refer_ident(&$1,TYPE_VARIABLE);

                set = ((T_symbol) $$->op1)->e.syntype->set;
                if (set == boolset) $$->sort = ETYPE_BOOLEAN;
                else if (set == SORT_PID) $$->sort = ETYPE_PID;
                else $$->sort = ETYPE_INTEGER;
          }
        | const
          {
                $$ = alloc_node();
                $$->type = NODE_CONST;
                $$->constant = True;
                $$->op1 = (T_node) $1;
                if ($1 == TT || $1 == FF) $$->sort = ETYPE_BOOLEAN;
                else if ($1 == DOT) $$->sort = ETYPE_PID;
                else $$->sort = ETYPE_INTEGER;
          }

/************/
/* DECISION */
/************/

decision: DECISION question ';'
          answerlist
          ENDDECISION ';'
          {
                T_list tr, l1;
                T_decision dec = alloc_decision();
                Boolean term = True, nonterm = True;
                int i = 0;

                $$ = alloc_action();
                $$->type = ACTION_DECISION;
                $$->e.decision = dec;
                dec->question = $2;
                dec->answers = alloc_list();
                dec->transitions = alloc_list();
                dec->refs = alloc_list();
                revert_list($4);

                $$->ref = add_ref($$,REF_ACTION,&$1,&$3);

                if ($4->count < 6)
                        Msg(ERR,"need at least two decision branches");

                /* answerlist ($4) returns one big list containing three
```

```
           entries for each answer. Here, these entries are separated
           and to form three distinct lists.
        */
        for_list ($4,tr,T_list)
                switch (i++ % 3)
                {
                    case 0: list_add(dec->refs,tr); break;
                    case 1: list_add(dec->transitions,tr); break;
                    case 2: list_add(dec->answers,tr); break;
                }

        free_list($4);

        /* check for terminating decision */
        for_list (dec->transitions,tr,T_list)
        {
                Boolean thisterm = tr->count
                     && list_elem(tr,tr->count-1,T_action)->terminating;
                term &= thisterm;
                nonterm &= !thisterm;
        }

        if (!(term || nonterm))
                Msg(ERR,"all or none of the decision branches "
                        "must be terminating");
        $$->terminating = term;

        /* check if answers are disjunct and if else is used properly */
        if (dec->question)
        {
                T_list irange, allrange = alloc_list();
                Boolean elsepart = False;
                int r;

                for_list (dec->answers,l1,T_list)
                {
                        if (elsepart) Msg(ERR,"else branch must be "
                                              "last in decision");
                        if ((elsepart = !l1)) continue;
                        irange = intersect_range(allrange,l1,False);
                        if (irange->count) Msg(ERR,"answers must be "
                                                  "mutually exclusive");
                        free_list(irange);
                        for_list (l1,r,int) list_add(allrange,r);
                }
                free_list(allrange);
        }
        else
                for_list (dec->answers,l1,T_list)
                        if (l1) Msg(ERR,"decision on ANY requires "
                                        "empty answers");
  }

question: expr
        | ANY
          {       $$ = NULL;       }
```

```
answerlist
        : answer
        | answer answerlist
          {
                $$ = $2;
                list_add($$,list_elem($1,0,void*));
                list_add($$,list_elem($1,1,void*));
                list_add($$,list_elem($1,2,void*));
                free_list($1);
          }

answer  : '(' optrangelist ')' ':' transition
          {
                T_ref ref = add_ref($2,REF_ANSWER,&$1,&$4);
                $$ = alloc_list();
                list_add($$,$2);
                list_add($$,$5);
                list_add($$,ref);
          }
        | ELSE ':' transition
          {
                T_ref ref = add_ref(NULL,REF_ANSWER,&$1,&$2);
                $$ = alloc_list();
                list_add($$,NULL);
                list_add($$,$3);
                list_add($$,ref);
          }

optrangelist
        : rangelist
        | { $$ = NULL; }

/********/
/* CALL */
/********/

call   : CALL IDENT optparms ';'
         {
                T_list plist;
                T_fmlparm parm;
                int i = 0;

                $$ = alloc_action();
                $$->type = ACTION_CALL;
                $$->terminating = False;
                $$->e.call = alloc_call();
                $$->e.call->sym = refer_ident(&$2,TYPE_PROCEDURE);
                $$->e.call->actparms = $3;
                $$->ref = add_ref($$,REF_ACTION,&$1,&$4);
                revert_list($3);
                list_add(called_procs,$$->e.call->sym);

                $$->e.call->fmlparms = plist
                        = $$->e.call->sym->e.struc->prcsinf->fmlparms;
```

```
                if (plist->count != $3->count)
                        Msg(ERR,"procedure call with wrong number of args");

                /* check type of arguments */
                for_list (plist,parm,T_fmlparm)
                {
                        T_node node = list_elem($3,i++,T_node);
                        intset type = parm->sym->e.syntype->set;

                        if (parm->type && (!node || node->type != NODE_VAR))
                                Msg(ERR,"need variable for in/out parameter");
                        if (!node) continue;
                        if ((node->sort == ETYPE_INTEGER
                                    && type != SORT_PID && type != boolset)
                           || (node->sort == ETYPE_BOOLEAN && type == boolset)
                           || (node->sort == ETYPE_PID && type == SORT_PID))
                                continue;
                        Msg(ERR,"argument type mismatch");
                }
          }

optparms: '(' actparmlist ')'
          {       $$ = $2;                  }
        | {       $$ = alloc_list();        }

actparmlist
        : expr
          {       list_add($$ = alloc_list(), $1);          }
        | expr ',' actparmlist
          {       list_add($$ = $3,$1);                     }
        | ',' actparmlist
          {       list_add($$ = $2,NULL);                   }
        | {       list_add($$ = alloc_list(), NULL);        }

/**********/
/* CREATE */
/**********/

create  : CREATE IDENT optparms ';'
          {
                T_symbol proc;

                $$ = alloc_action();
                $$->type = ACTION_CREATE;
                $$->terminating = False;
                $$->e.call = alloc_call();
                $$->ref = add_ref($$,REF_ACTION,&$1,&$4);

                /* Since forward references are possible, this rule creates
                   a temporary symbol list entry. This entry is appended
                   to created_procs and evaluated in the process rule.     */

                $$->e.call->sym = proc = decl_ident(&$2,TYPE_STRUCTURE);
                if (!proc->e.data)
                {
                        proc->type = TYPE_UNDECLARED;
```

```
                        proc->e.data = $$->e.call;

                        /* change the name information in the symbol table
                           so that other create requests of the same process
                           can get their own temporary entries in the table */
                        *(proc->name) |= 0x80;
                }
                list_add(created_procs,proc);

                $$->e.call->actparms = $3;
                revert_list($3);
          }

/**************/
/* TERMINATOR */
/**************/

terminator
        : nextstate
        | stop
        | join


nextstate
        : NEXTSTATE IDENT ';'
          {
                $$ = alloc_action();
                $$->type = ACTION_NEXTSTATE;
                $$->terminating = True;
                $$->e.state = decl_ident(&$2,TYPE_STATE);
                $$->ref = add_ref($$,REF_ACTION,&$1,&$3);
          }
        | NEXTSTATE '-' ';'
          {
                T_ident dummy = s_dummy("-");
                $$ = alloc_action();
                $$->type = ACTION_NEXTSTATE;
                $$->terminating = True;
                $$->e.state = decl_ident(dummy,TYPE_STATE);
                $$->ref = add_ref($$,REF_ACTION,&$1,&$3);
          }

stop    : stoporreturn
          {
                $$ = $1;
                $$->type = ACTION_NEXTSTATE;
                $$->terminating = True;
                $$->e.state = NULL;
          }

stoporreturn
        : STOP ';'
          {
                ($$ = alloc_action())->ref = add_ref($$,REF_ACTION,&$1,&$2);
          }
        | RETURN ';'
```

```
          {
                ($$ = alloc_action())->ref = add_ref($$,REF_ACTION,&$1,&$2);
          }

join    : JOIN IDENT ';'
          {
                $$ = alloc_action();
                $$->type = ACTION_JOIN;
                $$->terminating = True;
                $$->e.goal = decl_ident(&$2,TYPE_LABEL);
                $$->ref = add_ref($$,REF_ACTION,&$1,&$3);
          }

%%

#include "scanner.c"
```

Literaturverzeichnis

[Ack97] Martin Ackermann. Konzeption eines Compilers für eine parallele Sprache mit Prozeduren. Diplomarbeit, Universität Hildesheim, Mai 1997.

[ASU88] A. Aho, R. Sethi, and J.D. Ullman. *Compilerbau*. Addison-Wesley, 1988.

[BFF+95] E. Best, H. Fleischhack, W. Frączak, R. P. Hopkins, H. Klaudel und E. Pelz. A Class of Composable High Level Petri Nets. In G. De Michelis und M. Diaz, Herausgeber, *Proceedings of ATPN'95 (Application and Theory of Petri Nets), Torino*, *Lecture Notes in Computer Science*, Band 935, Seiten 103–118. Springer, Juni 1995.

[BG95] Eike Best und Bernd Grahlmann. PEP: Documentation and User Guide. Universität Hildesheim, November 1995. Erhältlich zusammen mit dem Tool via: http://www.informatik.uni-hildesheim.de/~pep.

[CCI92] CCITT. *Specification and Description Language, CCITT Z.100, Geneva.* International Consultative Committee on Telegraphy and Telephony, 1992.

[CMCH96] E. Clarke, K. McMillan, S. Campos und V. Hartonas-Garmhausen. Symbolic Model Checking. In R. Alur und T. A. Henzinger, Herausgeber, *Proceedings of CAV'96 (Computer Aided Verification), New Brunswick*, *Lecture Notes in Computer Science*, Band 1102, Seiten 419–422. Springer-Verlag, Juni 1996.

[ERV96] Javier Esparza, Stefan Römer und Walter Vogler. An Improvement of McMillan's Unfolding Algorithm. In Tiziana Margaria und Bernhard Steffen, Herausgeber, *TACAS'96 (Tools and Algorithms for the Construction and Analysis of Systems)*, *Lecture Notes in Computer Science*, Band 1055, Seiten 87–106. Springer-Verlag, 1996.

[Esp93] Javier Esparza. *A Partial Order Approach to Model Checking*. Habilitation, Universität Hildesheim, Februar 1993. Eine zusammengefaßte Version ist unter [Esp94] erschienen.

[Esp94] Javier Esparza. *Model Checking Using Net Unfoldings*, Seiten 151–195. Nummer 23 in Science of Computer Programming. ELSEVIER, 1994.

[FG97] Hans Fleischhack und Bernd Grahlmann. A Petri Net Semantics for $B(PN)^2$ with Procedures. In *Proceedings of PDSE'97 (Parallel and Distributed Software Engineering), Boston MA*, Seiten 15–27. IEEE Computer Society, Mai 1997.

[FG98a] Hans Fleischhack und Bernd Grahlmann. A Compositional Petri Net Semantics for SDL. In *Proceedings of ATPN'98 (Application and Theory of Petri Nets)*, Lecture Notes in Computer Science. Springer-Verlag, Juni 1998.

[FG98b] Hans Fleischhack und Bernd Grahlmann. Towards Compositional Verification of SDL Systems. In *Proceedings of 31st HICSS (Hawaii International Conference on System Science) – Software Technology Track*, Seiten 404–414. IEEE Computer Society, Januar 1998.

[Gra96] Bernd Grahlmann. Petri Net File Formats. In J. Desel, E. Kindler und A. Oberweis, Herausgeber, *Proceedings of AWPN'96 (3. Workshop Algorithmen und Werkzeuge für Petrinetze)*, Nummer 341 in Forschungsberichte, Seiten 21–26. Institut für Angewandte Informatik Universität Karlsruhe, Oktober 1996.

[Gra97a] Bernd Grahlmann. Format Descriptions. Universität Hildesheim. http://www.informatik.uni-hildesheim.de/~pep, März 1997.

[Gra97b] Bernd Grahlmann. The PEP Tool. In Orna Grumberg, Herausgeber, *Proceedings of CAV'97 (Computer Aided Verification), Lecture Notes in Computer Science*, Band 1254, Seiten 440–443. Springer-Verlag, Juni 1997.

[Gra97c] Bernd Grahlmann. The Reference Component of PEP. In Ed Brinksma, Herausgeber, *Proceedings of TACAS'97 (Tools and Algorithms for the Construction and Analysis of Systems), Lecture Notes in Computer Science*, Band 1217, Seiten 65–80. Springer-Verlag, April 1997.

[Gra98a] Bernd Grahlmann. Combining Finite Automata, Parallel Programs and SDL Using Petri Nets. In Bernhard Steffen, Herausgeber, *Proceedings of TACAS'98 (Tools and Algorithms for the Construction and Analysis of Systems), Lecture Notes in Computer Science*, Band 1384, Seiten 102–117. Springer-Verlag, März 1998.

[Gra98b] Bernd Grahlmann. *Parallel Programs as Petri Nets.* Doktorarbeit, Universität Hildesheim, 1998.

[Gra95] Burkhard Graves. Ein Modelchecker für eine Linear-Time-Logik. Diplomarbeit, Universität Hildesheim, Januar 1995.

[Gra97d] Burkhard Graves. Computing Reachability Properties Hidden in Finite Net Unfoldings. In S. Ramesh und G. Sivakumar, Herausgeber, *Proceedings of FST&TCS'97 (Foundations of Software Technology and Theoretical Computer Science), Lecture Notes in Computer Science*, Band 1346. Springer-Verlag, 1997.

[Hog89] Dieter Hogrefe. *Estelle, Lotos und SDL. Standard-Spezifikationssprachen für verteilte Systeme.* Springer, 1989.

[Hol90] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1990.

[Hol97] G.J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(05):279–295, 1997.

[McM92] K. L. McMillan. Automatic Verification Based on Occurence Nets. Technical report, 1992.

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publisher, 1993.

[MR96] S. Melzer und S. Römer. Synchronisierende Automaten in PEP. In J. Desel, E. Kindler und A. Oberweis, Herausgeber, *Proceedings of AWPN'96 (3. Workshop Algorithmen und Werkzeuge für Petrinetze)*, Nummer 341 in Bericht, Seiten 52–59. Institut für Angewandte Informatik Universität Karlsruhe, Oktober 1996.

[Rei85] Wolfgang Reisig. *Petri Nets. An Introduction*, *EATCS Monographs on Theoretical Computer Science*, Band 4. Springer Verlag, 1985.

[Sta92] Peter H. Starke. INA: Integrated Net Analyzer. Handbuch, 1992.

[Tur93] Kenneth J. Turner. *Using Formal Description Techniques. An Introduction to ESTELLE, LOTOS and SDL.* Wiley & Sons, 1993.