PEARL Rundschau

Inhalt

September 1981 Band 2 Nr. 3

PEARL Rundschau

September 1981
Band 2
Nr. 3

Der PEARL-Verein e.V. (PEARL-Association) hat das Ziel, die Verbreitung der Realzeitprogrammiersprache PEARL (Process and Experiment Automation Realtime Language) und ihre Anwendung sowie die Einheitlichkeit von PEARL-Programmiersystemen zu fördern.
Sitz des Vereins ist Düsseldorf. Seine Geschäftsstelle befindet sich im VDI-Haus, Graf-Recke-Straße 84, 4000 Düsseldorf 1.

Herstellung durch die VDI-Verlag GmbH, Düsseldorf

Die PEARL-Rundschau ist das Mitteilungsblatt des PEARL-Vereins e.V. Neben Vereinsangelegenheiten werden für alle PEARL-Interessenten Informationen über Erfahrungen, Anwendungsmöglichkeiten und Produkte gegeben.
Preis des Einzelheftes für Nichtmitglieder: DM 15,–, für Studenten DM 5,–. Jahresabonnement DM 75,–.
Zur Veröffentlichung werden sowohl fachliche Abhandlungen über Realzeit-Anwendungen, als auch Kurznachrichten zu Themen des Prozessrechnereinsatzes und der Verwendung höherer Sprachen angenommen, soweit sie für PEARL-Interessenten von Bedeutung sein können.

Es obliegt dem Autor, die Rechte zur Veröffentlichung seines Beitrags in der PEARL-Rundschau sicherzustellen. Der PEARL-Verein erhebt keine Einwände gegen die Kopie einzelner Beiträge bei Angabe der Quelle.

Beiträge können jederzeit, auch unaufgefordert, an ein Mitglied des Redaktionskollegiums geschickt werden. Autoren werden gebeten, bei der Schriftleitung ein Info-Blatt zur Manuskriptgestaltung anzufordern.

Mit dem Namen des Autors gekennzeichnete Beiträge geben nicht unbedingt die Meinung des Schriftleiters, des PEARL-Vereins oder dessen Vorstand wieder.
Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. berechtigt auch ohne besondere Kennzeichnung nicht zur Annahme, daß solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürfen.

Vorwort der Schriftleitung

Dies ist also das seit langem angekündigte Heft über die Normungstechnik von PEARL. Wir freuen uns besonders darüber, daß es uns gelungen ist, einen dafür besonders qualifizierten Fachmann als "guest-editor" zu gewinnen. Herr Prof. Dr. H. Rzehak, der den Lehrstuhl für Echtzeitrechensysteme an der Hochschule der Bundeswehr in München innehat, ist seit Jahren Vorsitzender des UA 5.8 "Programmiersprachen zur Steuerung technischer Prozesse" im Normenausschuß Informationsverarbeitung des DIN und somit sowohl mit den Problemen der Anwendungsgebiete als auch der Normung von PEARL bestens vertraut. Wir möchten ihm daher im Namen der Schriftleitung und des gesamten PEARL-Vereins herzlich für seine Mühe danken, die er aufgewendet hat, dieses Heft zu gestalten.

Die Artikel in diesem Heft sollen ein sowohl von Anwendern als auch Compilerbauern seit langem artikuliertes Bedürfnis erfüllen: eine leidlich lesbare und verständliche Einführung in die zur Normung von PEARL entwickelten und angewandten Methoden.

Diese Techniken haben ja wegen ihres "Härtegrades" eine ganze Weile zu einer erheblichen Polarisierung innerhalb der mit PEARL befaßten Gruppen geführt und der Unterzeichnete muß gestehen, daß er selbst auch nicht gerade zu den Befürwortern gehörte und das auch äußerte. Nachdem ihm aber gerade die Petrinetze eine wertvolle Denkhilfe gerade im Frühstadium von Entwürfen wurden, schien ihm seine eigene Kritik nicht mehr so ganz berechtigt. Inzwischen scheint sich auch herauszukristallisieren, daß eine "Primitivversion" der attributierten Bäume zu der Softwarebeschreibungstechnik der Zukunft werden kann. Sollte sich dies bestätigen, so wäre die am Beispiel von PEARL geleistete Arbeit von noch größerer Bedeutung für die Fachwelt, als sie sowieso schon zu sein scheint. Immerhin ist, nach glaubwürdiger Versicherung der damit befaßten Spezialisten, PEARL die bisher erste und einzige Realzeitprogrammiersprache, die wirklich formal sauber und eindeutig beschrieben ist.

Eine formale Beschreibungsweise, die komplexe Zusammenhänge erfassen will, muß aber ihrerseits eine Menge "eingebauten Vorwissens" enthalten, das in ihren Definitionen und Methoden steckt. Dieses Vorwissen fehlt aber zwangsläufig dem mehr anwendungsorientierten Techniker, da er sich wiederum mit Problemen zu befassen hat, von denen sich der Beschreibungsspezialist nichts träumen läßt. Daraus ergibt sich aber in weiterer Konsequenz ein Widerspruch zum Charakter traditioneller Normen, die unter anderem für Hersteller und Anwender eine gemeinsame Verständigungsbasis über die Eigenschaften von Produkten bilden sollten. Dies führt weiter zu Mißtrauen gegenüber der Norm, was ihren Einsatzwert schwächt. Also muß dieses Mißtrauen durch Aufbau von besserem Verständnis abgebaut werden. Wir hoffen, daß dieses Heft einen Beitrag zu diesem Prozeß liefert und wären an der Meinung unserer Leser dazu höchst interessiert.

Eine weitere Erläuterung soll zu dem Artikel von Herrn Prof. Dr. Frevert über ein einfaches Werkzeug für Entwurf und Dokumentation von PEARL-Programmen gegeben werden. Er ist als Vorgriff auf ein im nächsten Jahr geplantes Heft über die Bedeutung über die Bedeutung von PEARL im Entwicklungs- und Lebensdauerzyklus von Software gedacht. Interessant erscheint er vor allem dadurch, daß er zeigt, wie man auch mit einfachen Mitteln bereits deutliche Fortschritte erzielen kann.

Zum Abschluß noch eine kurze Vorschau auf weitere Hefte:

Zur Tagung "PEARL in der Ausbildung" und zur Jahrestagung werden Tagungsbände mit den Vorträgen erscheinen. Das Heft mit englischen Übersetzungen von ausgewählten Artikeln der PEARL-Rundschau ist in Arbeit. Weiterhin ist es gelungen, Herrn Dr. Ehrenberger vom Institut für Reaktorsicherung in Garching als "guest-editor" für ein Heft über Programmverifikation bei höheren Programmiersprachen zu gewinnen.

Für die Schriftleitung
Mit freundlichen Grüßen
Dr. P. Elzer

Zu diesem Heft

Von den Herausgebern der PEARL-Rundschau wurde ich gebeten, als Gast-Editor ein Heft herauszugeben, das Lesehilfen zum Studium der Norm von Full PEARL enthält. Dieser Bitte komme ich mit dem vorliegenden Heft nach. Ich habe mich bemüht, didaktisch geschickte Autoren zu finden. Sehr am Herzen lag mir dabei ein Beitrag, der den Grundgedanken der attributierten Grammatik erläutert und der ohne spezielle Vorkenntnisse verständlich ist.

Auch mit dieser Hilfe wird das Einarbeiten in den Normentwurf nicht ganz einfach sein. Für den Nicht-Spezialisten treffen zwei Erschwernisfaktoren zusammen: Eine reichhaltige, komplexe Sprache wie PEARL und die ungewohnte Darstellungsmethode. Es sollte niemand den Versuch unternehmen wollen, an Hand des Normentwurfes und der Beiträge in diesem Heft PEARL zu erlernen. Erreichbar erscheint es jedoch, daß der Leser in Zweifelsfällen Fragen zur Zulässigkeit oder Wirkung bestimmter Sprachkonstrukte selbst beantworten kann. Doch auch hier ist vor den Erfolg der Schweiß gesetzt: Die Artikel eignen sich nicht gerade zur Bettlektüre.

Wer mit dem Begriff "attributierte Grammatik" nichts anzufangen weiß, möge zuerst den Artikel von Heilbrunner und Schmitz lesen. In Beispielen, die sich an PEARL und den Normentwurf anlehen, wird der grundsätzliche Aufbau einer attributierten Grammatik erläutert und dargestellt, welche besonderen Möglichkeiten diese Definitionsmethode bietet. Einige Leser werden vielleicht bereits mit diesem allgemeinen Einblick in die Darstellungsmethode zufrieden sein. In den beiden Beiträgen von Kastens und Karbe geht es jedoch ganz tief ins Detail. Der Leser sollte hier vor allem die Beispiele selbständig nachvollziehen und daran anschließend selbst gestellte Fragen zu beantworten versuchen.

Zur Abrundung ist noch ein Artikel über die Entwicklung der Norm für PEARL aufgenommen worden. Er soll auch Außenstehenden einen Einblick in den Entscheidungsprozess vermitteln. Es bleibt zu hoffen, daß dieses Heft dem Anspruch gerecht wird, die Norm für PEARL verständlich zu machen und daß es hilft, die bestehenden Akzeptanzprobleme abzubauen.

Helmut Rzehak

Helmut Rzehak

Die Entwicklung der Norm für die Programmiersprache PEARL

Prof. Dr. H. Rzehak, München

1. Die Entwicklung des Sprachkonzeptes

Die Entwicklung der Programmiersprache PEARL entstand aus dem Bedürfnis, Echtzeitprogramme vollständig in einer höheren Programmiersprache formulieren zu können. Die Verwendung von Assemblerprogrammen sollte für den Anwendungsprogrammierer damit überflüssig werden. Da in erster Linie an einen Einsatz zur Steuerung technischer Prozesse gedacht war, mußte eine genügend anwendungsnahe Behandlung der Prozessperipherie in das Sprachkonzept aufgenommen werden. Letztlich mußte der Versuch unternommen werden typische Betriebssystemfunktionen in der Programmiersprache einheitlich zu definieren. Die Programme sollen auf der Ebene der Programmiersprache portabel sein; Kompilierer und Laufzeitsystem müssen eine entsprechende Anpassung an das verwendete Rechensystem, insbesondere auch an das Betriebssystem liefern.

Mit dem Sprachkonzept mußte in weiten Bereichen Neuland betreten werden. Es ist daher nicht verwunderlich, wenn der erste veröffentlichte Entwurf vom April 1973 bis zu der heute vorliegenden Norm bzw. dem Normentwurf vielfache Änderung erfahren hat. Meilenstein auf diesem Wege war insbesondere die Herausgabe von KFK-PDV 120 "Basic PEARL Language Description" und KFK-PDV 130 "Full PEARL Language Description" im Jahre 1977. Die dort beschriebene Sprache entspricht bereits im wesentlichen der derzeit genormten Sprache. Es schmälert nicht das Verdienst der Autoren und Mitarbeiter, wenn festgestellt wird, daß die zitierten Arbeiten hinsichtlich der Darstellungsmethode noch Mängel aufweisen. In einer so jungen und noch stark in Entwicklung befindlichen Wissenschaft wie der Informatik kann man nicht an Arbeiten aus dem Jahre 1977 die Maßstäbe des Jahres 1981 legen.

PEARL wird - wie verschiedene andere Programmiersprachen auch - gelegentlich als "Komiteesprache" bezeichnet. Gemeint ist damit, daß verschiedene Gruppen mit zum Teil stark divergierenden Auffassungen und Interessen an der Sprachentwicklung beteiligt waren, zwischen denen Kompromisse geschlossen werden mußten. Da parallel zur Sprachentwicklung bereits Pilotimplementierungen durchgeführt wurden, mag die Einigung in Einzelfällen auch davon beeinflußt worden sein, was eine starke Interessengruppe bereits implementiert hat. Von großem Vorteil war es jedoch, daß durch die heterogene Zusammensetzung der PEARL-Entwickler Gesichtspunkte aus sehr vielen Bereichen berücksichtigt wurden.

Bei der Entwicklung von PEARL mußte in einigen Teilbereichen Neuland beschritten werden, auch ist die Erprobung einzelner Sprachkonzepte in Pilotimplementierungen zeitraubend. Rückwirkend betrachtet ist es außerordentlich bedauerlich, daß zwischen der ersten Veröffentlichung im Jahre 1973 und den konsolidierten Entwürfen im Jahre 1977 eine Zeitspanne von 4 Jahren verging. Mögliche Implementierer und Anwender waren stark verunsichert über das endgültige Aussehen von PEARL. Die rasche Verbreitung wurde entscheidend gehemmt.

2. Die Ausarbeitung der Normvorlage

Normen geben den Stand der Technik wieder und müssen in gewissen Abständen an den sich ändernden Stand der Technik angepaßt werden. Im DIN strebt man eine Überarbei-

tung nach 5 bis höchstens 7 Jahren an. Der Anwender einer Norm muß sich darauf verlassen können, daß der Inhalt einen gewissen Grad der Reife erlangt hat. Da im gesamten Bereich der Informationsverarbeitung der Stand der Technik noch stark im Wandel begriffen ist, gibt es hier einen permanenten Zielkonflikt. Einerseits ist Normung erforderlich, um überhaupt die Zusammenarbeit über die Grenzen von Organisationen hinweg zu ermöglichen und wirtschaftliche Lösungen verwirklichen zu können. Andererseits können durch voreilige Normung Festlegungen getroffen werden, die sich vergleichsweise rasch als hinderlich erweisen oder gar mit großem Aufwand für die Wirtschaft geändert werden müssen.

Nach Erscheinen von KFK-PDV 120 und 130 war man sich grundsätzlich darüber einig, daß PEARL den erforderlichen Grad der Reife erreicht hat, um Gegenstand einer Norm werden zu können. Man sah jedoch noch Mängel in der Darstellung, die durch eine entsprechende Überarbeitung beseitigt werden müßten. In diesen Berichten wird die Sprache wie vielfach üblich durch eine formal definierte kontextfreie Syntax mit zusätzlichen nicht formalisierten Erläuterungen über weitere Einschränkungen und einer verbalen Beschreibung der Wirkung der einzelnen Anweisung beschrieben. Diese Beschreibung ist nicht genügend präzise, so daß Unklarheiten über die Zulässigkeit bzw. Wirkung einzelner Anweisungsfolgen bestehen. Des weiteren ist in den zitierten Berichten nicht klar genug herausgestellt, welche Einzelheiten der Sprache implementationsabhängig sind. Abgesehen davon kann der Kompilerbauer nur verhältnismäßig mühsam die Kontextabhängigkeiten aus der Sprachbeschreibung entnehmen, was häufig zur Folge hat, daß der Kompilierer nicht alle fehlerhaften Programme erkennt und zurückweist.

Der Normenausschuß Informationsverarbeitung (NI) im DIN hat sich als das in der Bundesrepublik Deutschland für die Normung von PEARL zuständige Organ seit den Jahren 76/77 mit dieser Sprache beschäftigt. Für die erforderliche Sacharbeit wurde der Unterausschuß 5.8 "Programmiersprachen zur Steuerung technischer Prozesse" gegründet. Um möglichst schnell für Implementierer und Anwender verfügbare Unterlagen zu erstellen, mußten Kompromisse bezüglich der wünschenswerten Präzision der Beschreibungsmethode geschlossen werden. Der Arbeitsplan bestand

a) in der Ausarbeitung der Norm für Basic PEARL basierend auf KFK-PDV 120. Um das Dokument schnell fertigzustellen, sollte die verbale Beschreibung beibehalten werden mit Ausnahme der Bereiche Realzeitobjekte und Realzeitoperationen, für deren Beschreibung Petri-Netze herangezogen wurden.

b) in der anschließenden Ausarbeitung der Norm für Full PEARL basierend auf KFK-PDV 130. Es war beabsichtigt, hierfür eine möglichst voll formalisierte Beschreibung zu verwenden. Die Auswahl einer geeigneten Methode sollte erst zu Beginn der Arbeiten festgelegt werden.

Die Arbeiten führten zunächst zur Veröffentlichung des Normentwurfes DIN 66 253 "Die Programmiersprache PEARL; Teil 1 Basic PEARL" im Jahre 1978. Entsprechend dem Arbeitsplan wurden nun die Arbeiten am Normentwurf für Full PEARL begonnen. Zur Auswahl einer geeigneten Definitionsmethode muß gesagt werden, daß eine erprobte Methode zur formalen Definition einer Realzeit-Programmiersprache nicht zur Verfügung stand; es mußte also Neuland beschritten werden. Zum einen sind die Kontextabhängigkeiten und die implementierungsabhängigen Details zu beschreiben. Man faßt dies unter dem Begriff der statischen Semantik zusammen. Ferner muß die Wirkung der Anweisungen zur Laufzeit des Programmes beschrieben werden (dynamische Semantik). Beim Stand der Technik im Jahre 1978 konnte nicht davon ausgegangen werden, daß eine voll formalisierte Beschreibung der dynamischen Semantik für alle Sprachelemente und auf beliebigem Detaillierungsgrad erreichbar ist. Da sich Petri-Netze zur Beschreibung von Realzeitobjekten und -operationen in Basic PEARL durchaus bewährt hatten, sollten zur Beschreibung der dynamischen

Semantik in Full PEARL ebenfalls Petri-Netze verwendet werden. Zur Beschreibung der statischen Semantik standen zwei Methoden zur Auswahl:

- eine Zweischichten-Grammatik in der besonderen Form einer Konjugationsgrammatik
- eine attributierte Grammatik.

Man ging davon aus, daß eine voll formalisierte Verbindung zu den Netzdarstellungen in beiden Fällen nicht mehr rechtzeitig entwickelt werden kann. Die Entscheidung fiel zugunsten einer attributierten Grammatik aus. Hauptgründe waren:

- Die Darstellung der Implementierungsabhängigkeiten erschien leichter möglich; ebenso die Formulierung einer Teilmengen-Bedingung für Basic PEARL.

- Die Einarbeitung in die Darstellung einer attributierten Grammatik erschien auf der Basis einer ingenieurmäßigen Ausbildung leichter möglich.

Das Problem der Akzeptanz einer formalen Sprachdefinition wurde damals klar gesehen. Allen Mitarbeitern in den Normungsgremien war aber klar, daß verläßlichere Unterlagen nur mit einer stärkeren Formalisierung erstellt werden können. Eine Sprachnorm wird zunächst für die Implementierer der Sprache sowie für die Autoren von Ausbildungsunterlagen herausgegeben. Dieser Personenkreis besteht aus Fachleuten, die mit formalen Definitionsmethoden grundsätzlich vertraut sind, und denen das Einarbeiten in eine spezielle Darstellungsweise zugemutet werden kann. Eine Sprachnorm kann jedoch kein geeignetes Hilfsmittel sein, an Hand dessen ein Programmierer die Sprache erlernen kann. Hierzu sind vielmehr entsprechende Lehrbücher erforderlich. Natürlich trägt es zur Verbreitung einer Programmiersprache bei, wenn die offizielle Sprachnorm für einen möglichst großen Personenkreis verständlich ist. Die Definition mit Hilfe einer attributierten Grammatik bietet noch am besten die Möglichkeit, daß sich ein interessierter Anwender mit einer naturwissenschaftlich-technischen Ausbildung und guten Programmierkenntnissen an Hand gezielter Anleitung soweit mit der Norm vertraut macht, daß er in der Lage ist Zweifelsfragen an Hand der Norm selbst zu beantworten.

Ein anderes grundsätzliches Problem entstand durch die Tatsache, daß für Full PEARL eine gänzlich andere Beschreibungsmethode verwendet werden sollte als für Basic PEARL. Trotz dieser verschiedenen Art der Beschreibung mußte sichergestellt werden, daß Basic PEARL eine Teilsprache von Full PEARL ist. Letztlich konnte das Problem nur dadurch gelöst werden, daß in der Definition von Full PEARL eine entsprechende Teilmengenbedingung aufgenommen wurde. DIN 66 253 Teil 2 enthält daher eine integrierte Beschreibung von Full PEARL und Basic PEARL. Durch umfangreiche Prüfungen mußte sichergestellt werden, daß diejenige Sprache, die in DIN 66 253 Teil 2 durch die Teilmengenbedingung beschrieben wird, nicht den vorher getroffenen Festlegungen für Basic PEARL widerspricht. In wenigen Punkten mußten jedoch mit Rücksicht auf die Konsistenz von Full PEARL Änderungen vorgenommen werden.

Der Begriff "Teilsprache" wurde schließlich in der folgenden Weise verwendet

- Jedes gültige Full PEARL-Programm, das von einem Kompilierer für Basic PEARL akzeptiert wird, ist gleichzeitig ein gültiges Basic PEARL-Programm.

- Jedes gültige Basic PEARL-Programm ist ein gültiges Full PEARL-Programm. Die Ergebnisse dürfen nicht davon abhängen, ob es mit einem Kompilierer für Basic PEARL oder für Full PEARL übersetzt wurde.

Da DIN 66 253 Teil 2 eine integrierte Beschreibung von Full PEARL und Basic PEARL enthält, hätte man prinzipiell auf Teil 1 verzichten können. Man wollte jedoch mit der Herausgabe einer gültigen Norm für Basic PEARL nicht so lange warten und entschloß sich daher parallel zur Arbeit an Teil 2 den Teil 1 in einer berichtigten

Fassung als Vornorm herauszugeben. Diese liegt seit Juni 81 vor und ist hinsichtlich der Seitenzahl geschrumpft, da der Datenträger mit einem engeren Zeichen- und Zeilenabstand als Druckvorlage ausgegeben wurde. Die Numerierung der Kapitel ist jedoch unverändert.

Der Normentwurf für DIN 66 253 Teil 2 liegt seit Dezember 1980 vor. Mit Herausgabe der gültigen Norm kann im Frühjahr 1982 gerechnet werden. Die gewählte Beschreibungsmethode hat sich grundsätzlich bewährt, wenn auch erwartungsgemäß noch Schwierigkeiten in der Darstellung des Zusammenhangs zwischen den Netzen und der attributierten Grammatik bestehen. Vorschläge zur Behebung dieses Mangels sind in der Zwischenzeit veröffentlicht worden. Wenn diese auch sicherlich bereits 1978 in den Köpfen der Autoren existierten, so fehlten konkrete Vorschläge für die Anwendung. Eine Beseitigung des bestehenden Mangels wird also noch zukünftigen Überarbeitungen der Norm für PEARL vorbehalten bleiben.

3. Ausblick

Mit Rücksicht auf eine internationale Verbreitung von PEARL wurde die Norm ebenso wie die vorausgehenden Sprachbeschreibungen in Englisch abgefaßt. Leider ist es bisher nicht gelungen, die internationale Normung durch die ISO entscheidend voranzubringen. Die internationale Normung steht im Brennpunkt vielschichtiger Interessen und man darf daher durchaus feststellen, daß das Schicksal von PEARL kaum durch technisch-wissenschaftliche Faktoren bestimmt war. Es besteht jedoch durchaus die Hoffnung, daß der Durchbruch unter etwas günstigeren Umständen noch gelingt.

Erheblich bedeutender für den Anwender ist es, daß sich durch den PEARL-Verein eine Organisation gebildet hat, die sich der Pflege von PEARL im Vorfeld der Normung angenommen hat, so daß die Sprache zukünftigen Erfordernissen angepaßt wird. Sowohl die Rechtsgrundlage als auch die finanzielle Ausstattung erlaubt es dem DIN nicht, Kompilierer auf Einhaltung der Norm zu überprüfen. Auch hier besteht eine zukünftige Aufgabe, die nur durch Organisationen außerhalb des DIN wahrgenommen werden kann. Weiterhin besteht ein Bedürfnis zur Vereinheitlichung der Programmierumgebung, z.B. einheitliche Testhilfen. Die weite Verbreitung einer Programmiersprache hängt letztlich auch von der Verfügbarkeit dieser Hilfsmittel ab.

Literatur:

T. Martin: The Development of PEARL.
Bericht KFK-PDV 129, Gesellschaft für Kernforschung m.b.H. Karlsruhe 1977

Zur attributierten Grammatik von PEARL

Von Prof. Dr. S. Heilbrunner und Dipl. -Inform. L. Schmitz, München

1. Einleitung

Der Normenentwurf zur Programmiersprache PEARL [1] enthält eine umfangreiche attributierte Grammatik. Darin eingebettet sind Petri-Netze zur Definition asynchroner Abläufe sowie die Beschreibung der Bedeutung von PEARL-Sprachkonstrukten in "technischem Englisch". Der vorliegende Artikel gibt eine Einführung in attributierte Grammatiken und in die besondere Weise ihrer Verwendung in der PEARL-Norm.

Bei der Definition von Programmiersprachen wird oft eine in Backus-Naur-Form (BNF) niedergeschriebene kontextfreie Grammatik benutzt. Einschränkungen, die sich in BNF nicht formulieren lassen - sogenannte Kontextbedingungen - , werden in natürlicher Sprache angegeben. Attributierte Grammatiken sind Erweiterungen von kontextfreien Grammatiken, die unter anderem die Formulierung von Kontextbedingungen im Rahmen der Grammatik zulassen.

Als Anwendungsbeispiel für diese Einführung dient ein ganz kleiner Ausschnitt aus PEARL, der in Abschnitt 2 mit Hilfe einer kontextfreien Grammatik und zusätzlichen Kontextbedingungen beschrieben wird. In Abschnitt 3 folgt die schrittweise Attributierung dieser kontextfreien Grammatik, wobei informelle Schreibweisen benutzt werden. Der vierte Abschnitt bringt die gleiche Grammatik nochmals, benutzt aber die Schreibweisen des PEARL-Normentwurfs. [1)]

Attributierte Grammatiken wurden eingeführt von D. Knuth [2] und haben seither einen festen Platz unter den Methoden zur formalen Definition von Programmiersprachen. Marcotty, Ledgard und Bochmann [3] geben dazu eine vergleichende Übersicht. Ein weiteres Anwendungsgebiet für attributierte Grammatiken sind übersetzererzeugende Systeme. Eine kurze Einführung in diese Anwendungen gibt Wilhelm [4]. In [3] und [4] findet man auch weitere Literaturhinweise.

2. Ein PEARL-Ausschnitt

Die Produktionsregeln R1-R34 in T a f e l 1 beschreiben einen PEARL-Ausschnitt PA, dessen Programme aus einem Vereinbarungsteil und einem darauf folgenden Zuweisungsteil bestehen. Vereinbart werden einfache Variablen der Typen *fixed* und *float*. Bei Zuweisungen sind beide Seiten einfache Variablen. Die folgende Zeichenreihe ist ein korrektes PA-Programm.

Tafel 1. Die Syntax von PA. Terminale Symbole sind kursiv geschrieben. Die Abkürzungen bedeuten decl(aration), seq(uence), ass(ignment) und var(iable).

R1 :	pa-program	::=	*begin* decl-seq ; ass-seq *end*
R2 :	decl-seq	::=	decl ; decl-seq
R3 :	decl-seq	::=	decl
R4 :	ass-seq	::=	ass ; ass-seq
R5 :	ass-seq	::=	ass
R6 :	decl	::=	*dcl* var *fixed*
R7 :	decl	::=	*dcl* var *float*
R8 :	ass	::=	var := var
R9 :	var	::=	*a*
R10:	var	::=	*b*
. . . . .			
R34:	var	::=	*z*

1) Um Mißverständnissen vorzubeugen, sei darauf hingewiesen, daß unsere Grammatik keine Teilgrammatik des Normentwurfs ist.

```
begin
  dcl a fixed; dcl b float; dcl c fixed;
  b := a; c := a; b := c
end
```

Dieses PA-Programm wird uns als laufendes Beispiel durch den Rest des Artikels begleiten.

PA-Programme unterliegen folgenden *Kontextbedingungen:*

KB1: Keine Variable wird mehrmals vereinbart

KB2: Jede benutzte Variable wird vereinbart

KB3: Ist die rechte Seite einer Zuweisung vom Typ *float*, dann auch die linke Seite.

Zur Erläuterung der Kontextbedingungen betrachten wir folgende Zeichenreihe, die den Regeln R1-R34 genügt.

```
begin
  dcl a fixed; dcl a fixed; dcl c float;
  b := a; a := c;
end
```

Die Variable *a* wird zweimal vereinbart im Widerspruch zu KB1. Die Zuweisung *b := a* widerspricht KB2, die Zuweisung *a := c* widerspricht KB3.

Neben Kontextbedingungen gibt es noch implementierungsabhängige Einschränkungen von Übersetzern und Rechensystemen oder auch Einschränkungen der Sprache (PEARL) auf eine Teilmenge (BASIC-PEARL). Wir versehen deshalb PA mit der *Implementierungsbedingung*

IB: Es dürfen höchstens anzvar Variablen vereinbart werden, wobei anzvar eine implementierungsabhängige Konstante ist.

Außerdem benutzen wir die (etwas künstliche) *Teilmengenbedingung*

TB: In BASIC-PA hat die linke Seite jeder Zuweisung den gleichen Typ wie die rechte Seite.

Unser laufendes Beispiel verletzt TB und erfüllt IB, falls anzvar ≥ 3 gilt.

3. Attributierte Grammatiken

Wir wollen zeigen, wie sich die Bedingungen aus Abschnitt 2 in die Grammatik für PA einbringen lassen, das heißt, mit den Regeln R1-R34 verknüpfen lassen. Zunächst ordnen wir jeder Zeichenfolge, die sich mit der kontextfreien Grammatik herstellen läßt, ihren *Strukturbaum* zu. B i l d 1 zeigt den Baum für das laufende Beispiel.

In den nächsten Abschnitten versehen wir die Knoten des so gewonnenen Strukturbaumes mit gewissen Werten - sogenannten *Attributwerten*, kurz: *Attribute* - und prüfen die Bedingungen KB1-KB3, IB und TB anhand dieser Werte nach.

3.1. Attributierung des Beispiels

Um die Kontextbedingungen KB2 und KB3 nachprüfen zu können, muß man bei jeder Zuweisung wissen, welche Variablen vereinbart wurden und welche Typen sie haben. Wir versehen deshalb alle ass-Knoten des Strukturbaumes mit der Liste der vereinbarten Variablen und deren Typen. Im Beispiel ergibt das die Liste

$$(\langle a, fixed\rangle, \langle b, float\rangle, \langle c, fixed\rangle)$$

für die Knoten 10,22 und 35. Weiter versehen wir die var-Knoten mit den daraus abgeleiteten Variablenbezeichnern als Attributwerten. Für die Knoten 19 und 21 erhalten wir *b*, bzw. *a*.

Für den bei ass_{10} beginnenden Teilbaum ergibt die Attributierung B i l d 2.

Nach dieser Attributierung können wir eine neue Fassung von KB2 angeben. Sie wird an die Produktionsregel

R8: ass ::= var := var

angefügt und lautet:

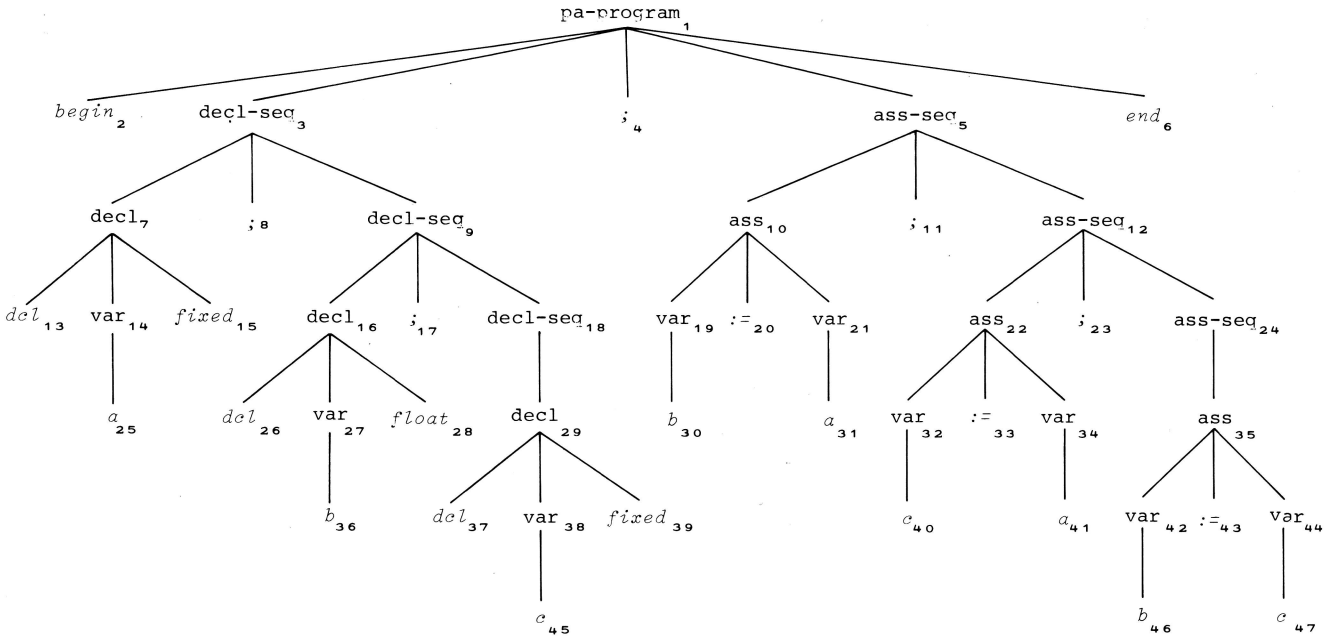


Bild 1. Strukturbaum zum laufenden Beispiel. Die Indizes numerieren die Knoten zur besseren Unterscheidung

KB2': Die Attributwerte des linken und des rechten var-Knotens kommen im Attributwert des ass-Knotens vor.

Man prüft KB2' für B i l d 2 sofort nach. KB2' muß im attributierten Strukturbaum überall dort gelten, wo R8 verwendet wurde. Die Formulierung von KB2' ist noch etwas umständlich. Das liegt vor allem daran, daß wir die Attribute nicht benennen können. Wir führen deshalb *Attributnamen* ein. Die Attributwerte der ass-Knoten nennen wir env-Attributwerte (nach engl. environment: Umgebung) und schreiben ass.env für den env-Attributwert der ass-Knoten. Man sieht, Attribute können als Komponenten der Knoten des Strukturbaumes verstanden werden.

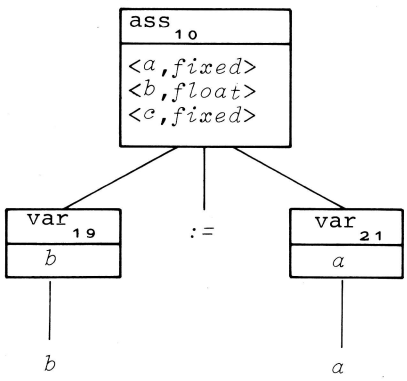


Bild 2. Attributierter Teilbaum für ass_{10}.

Entsprechend verfahren wir mit den var-Knoten. Ihre Attribute bezeichnen wir mit var.denot (für engl. denotation: Bezeichnung). In der Produktionsregel R8 kommt var zweimal vor. Um Eindeutigkeit zu erreichen, numerieren wir gleiche syntaktische Variablen in den Produktionsregeln von links nach rechts und erhalten damit

var[1].denot bzw. var[2].denot

als Bezeichnung für die denot-Attribute des ersten bzw. zweiten Auftretens von var in R8.

Für KB2 ergibt sich damit in Zusammenhang mit der Regel

R8: ass ::= var := var

schließlich die Formulierung

KB2": ass.env enthält Paare <var[1].denot,...> und <var[2].denot,...>.

Wir benutzten jetzt auch in KB3 Attributwerte und fügen an R8 noch an:

KB3': Enthält ass.env das Paar <var[2].denot,*float*> , dann auch das Paar <var[1].denot,*float*>

Man prüft leicht nach, daß an allen drei Stellen, wo R8 im Strukturbaum des laufen-

den Beispiels benutzt wird, die Bedingungen KB2" und KB3' erfüllt sind. Man betrachte hierzu die entsprechenden Ausschnitte aus B i l d 1 . Schließlich fassen wir noch die Teilmengenbedingung neu und fügen sie an R8 an als

TB': Kommt <var[2].denot,*fixed*> in ass.env vor, dann auch <var[1].denot,*fixed*> (weiter ist wegen KB3' nichts zu fordern).

Bis jetzt wurde gezeigt, wie Attributwerte bei kontextfreien Regeln benannt werden, und wie sie zum Abfassen von Bedingungen benutzt werden können. Wie aber berechnet man Attributwerte? Dazu werden den Produktionsregeln neben den Bedingungen noch *Attributauswertungsregeln* beigefügt, die an jeder Stelle des Strukturbaums anzuwenden sind, an der die Produktionsregel benutzt wird. Attributauswertungsregeln haben die Form von Zuweisungen an Attributnamen.

Im Beispiel wird der Produktionsregel

R9: var ::= *a*

die Attributauswertungsregel

var.denot := *a*

mitgegeben. Die Regeln R10-R34 erhalten analog dazu die Attributauswertungsregeln

var.denot := *b* ..... var.denot := *z*

Damit können wir alle var-Knoten attributieren.

Schwieriger ist die Bestimmung der env-Attribute. Wir nehmen an, der Knoten $ass\text{-}seq_5$ habe bereits sein env-Attribut. Dann lassen sich die Attributwerte der Söhne von $ass\text{-}seq_5$ berechnen, indem wir an die Regel

R4: ass-seq ::= ass ; ass-seq

die Attributauswertungsregeln

ass.env := ass-seq[1].env ,
ass-seq[2].env := ass-seq[1].env

anfügen und an

R5: ass-seq ::= ass

die Attributauswertungsregel

ass.env := ass-seq.env

Mit den bisherigen Attributauswertungsregeln können wir den Zuweisungsteil von PA-Programmen vollständig attributieren. Wir erhalten den Teilbaum aus B i l d 3 .

Wir attributieren jetzt den Vereinbarungsteil und betrachten dazu B i l d 4 , das einen Ausschnitt aus dem Strukturbaum zeigt. Um mehrfache Vereinbarungen von Variablen sofort erkennen zu können, müssen wir für jeden decl-Knoten die Liste der links davon vereinbarten Variablen kennen. Zur Aufbewahrung dieser Liste geben wir den decl-Knoten das Attribut pre-env. Die zu einem decl-Knoten gehörige Vereinbarung verlängert dieses Attribut um einen Eintrag und erzeugt das Attribut des nächsten decl-Knotens. Zur Aufbewahrung der verlängerten Listen geben wir jedem decl-Knoten noch ein zweites Attribut, das post-env-Attribut. Für die decl-Knoten aus B i l d 4 erhalten wir damit folgende Attributierungen, wobei () die leere Liste bezeichnet.

$decl_7$	
pre-env:	post-*env*:
()	<*a*,*fixed*>

$decl_{16}$	
pre-env:	post-env:
<*a*,*fixed*>	<*a*,*fixed*> <*b*,*float*>

$decl_{29}$	
pre-env:	post-env:
<*a*,*fixed*> <*b*,*float*>	<*a*,*fixed*> <*b*,*float*> <*c*,*fixed*>

Wie lauten die zugehörigen Attributauswertungsregeln? Im Beispiel (vgl. B i l d 4) können Attribute von $decl_{16}$ nicht unmittelbar an $decl_{29}$ übergeben werden, da es keine Produktionsregel gibt, deren Anwendung $decl_{16}$ und $decl_{29}$ gleichzeitig betrifft.

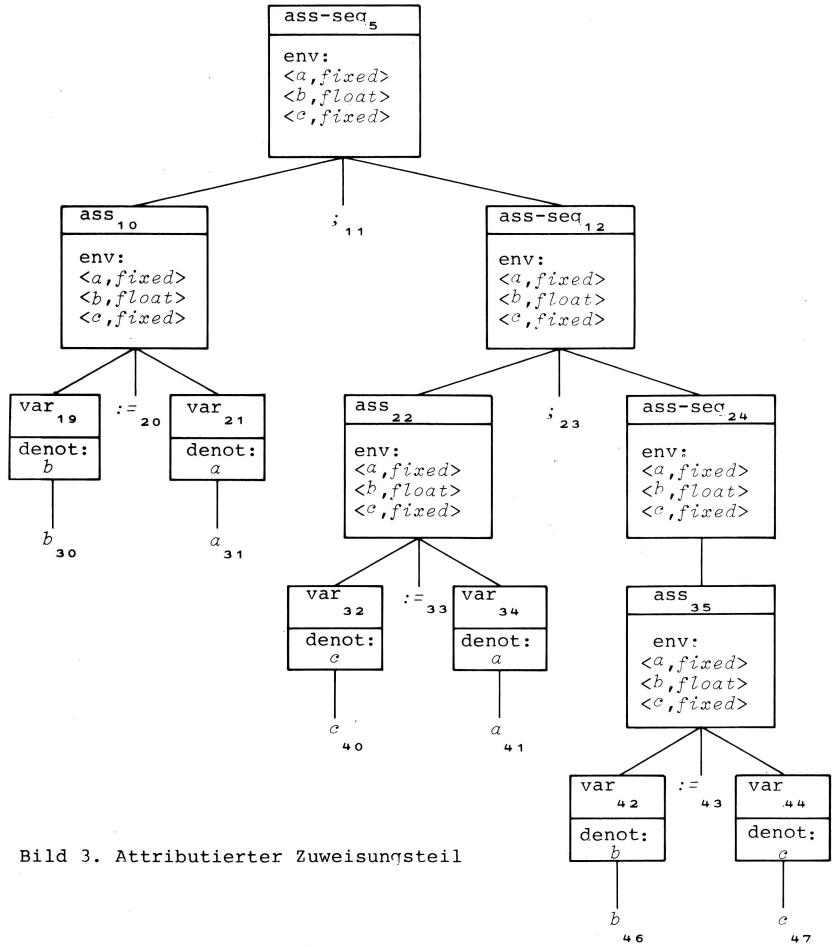


Bild 3. Attributierter Zuweisungsteil

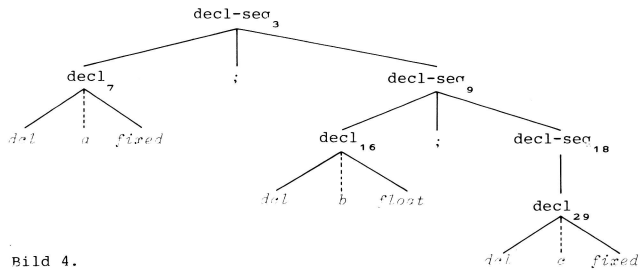


Bild 4.

Vielmehr müssen Attribute von $decl_{16}$ über die Knoten $decl\text{-}seq_9$ und $decl\text{-}seq_{18}$ zu $decl_{29}$ transportiert werden. Zur Abwicklung dieses Transports versehen wir auch die decl-seq-Knoten mit den Attributen pre-env und post-env. Das ergibt B i l d 5.

Aus diesem Bild lassen sich die Attributauswertungsregeln sofort ablesen. Für

R2: decl-seq ::= decl ; decl-seq

ergeben sich

decl.pre-env := decl-seq[1].pre-env ,
decl-seq[2].pre-env := decl.post-env ,
decl-seq[1].post-env := decl-seq[2].post-env.

Für

R3: decl-seq ::= decl

ergeben sich

decl.pre-env := decl-seq.pre-env ,
decl-seq.post-env := decl.post-env .

Wir erinnern daran, daß die var-Knoten bereits das var.denot-Attribut besitzen und versehen

R6: decl ::= *dcl* var *fixed*

mit

decl.post-env := decl.pre-env
+<var.denot,*fixed*>

und

R7: decl ::= *dcl* var *float*

mit

decl.post-env := decl.pre-env
+<var.denot,*float*>

In den letzten zwei Attributauswertungs-

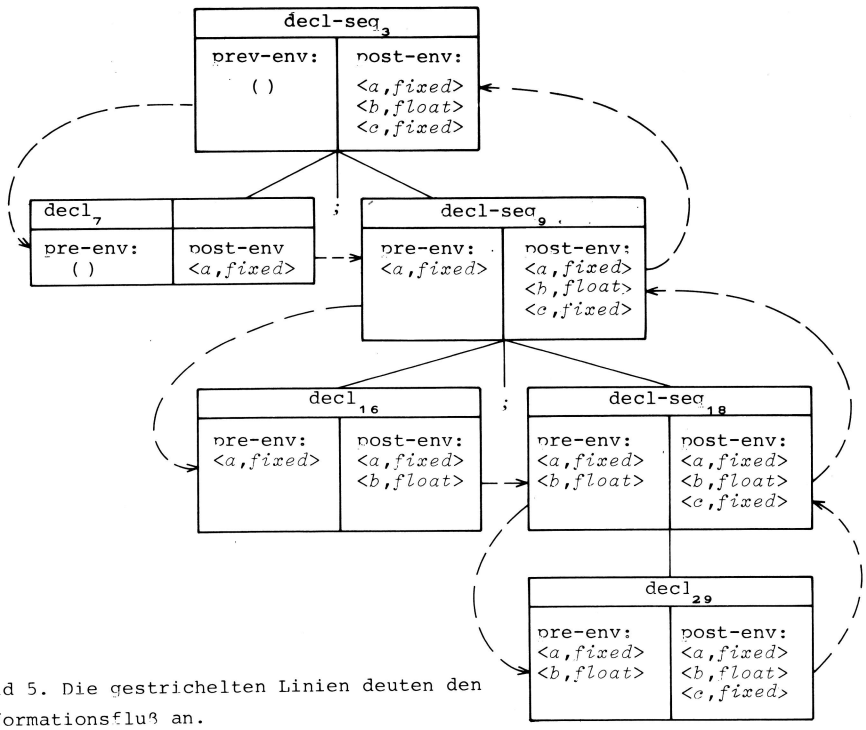


Bild 5. Die gestrichelten Linien deuten den Informationsfluß an.

regeln ist "+" der Konkatenationsoperator für Listen.

Es fehlen noch zwei Attributierungsregeln. Die erste führt die ursprünglich leere Umgebung ein. Die zweite reicht die im Vereinbarungsteil aufgesammelte Liste von Variablen an den Zuweisungsteil weiter. Das ergibt für

R1: pa-programm ::= *begin* decl-seq ; ass-seq *end*

die Attributierungsregeln

decl-seq.pre-env := () ,
ass-seq.env := decl-seq.post-env

Insgesamt erhalten wir für den Vereinbarungsteil den attributierten Baum aus Bild 6.

An R1 schließt sich noch die Implementierungsbedingung IB an als

IB': Die Liste decl-seq.post-env enthält höchstens anzvar Einträge.

3.2. Begriffliches

Bei der Attributierung des Beispiels im letzten Abschnitt sind eine Reihe von Fragen offen geblieben, die wir nun diskutieren wollen.

Was bedeutet es, wenn eine Bedingung nicht erfüllt ist? Wenn bei der Attributierung des Strukturbaumes zu einer vorgelegten Zeichenreihe Z die Kontextbedingungen überall dort erfüllt sind, wo die zugehörigen kontextfreien Regeln zum Aufbau des Strukturbaumes verwendet wurden, dann und nur dann ist Z ein korrektes PA-Programm. Sind außerdem alle Teilmengenbedingungen erfüllt, dann ist Z ein korrektes BASIC-PA-Programm. Verletzt ein korrektes (BASIC-) PA-Programm eine Implementierungsbedingung, dann genügt Z nicht den Einschränkungen der betreffenden Implementierung.

In welcher Reihenfolge werden Attribute ausgewertet? Grundsätzlich wird jeder Attributwert genau einmal bestimmt. Die Reihenfolge der Attributauswertungen ist beliebig, unterliegt aber der Einschränkung,

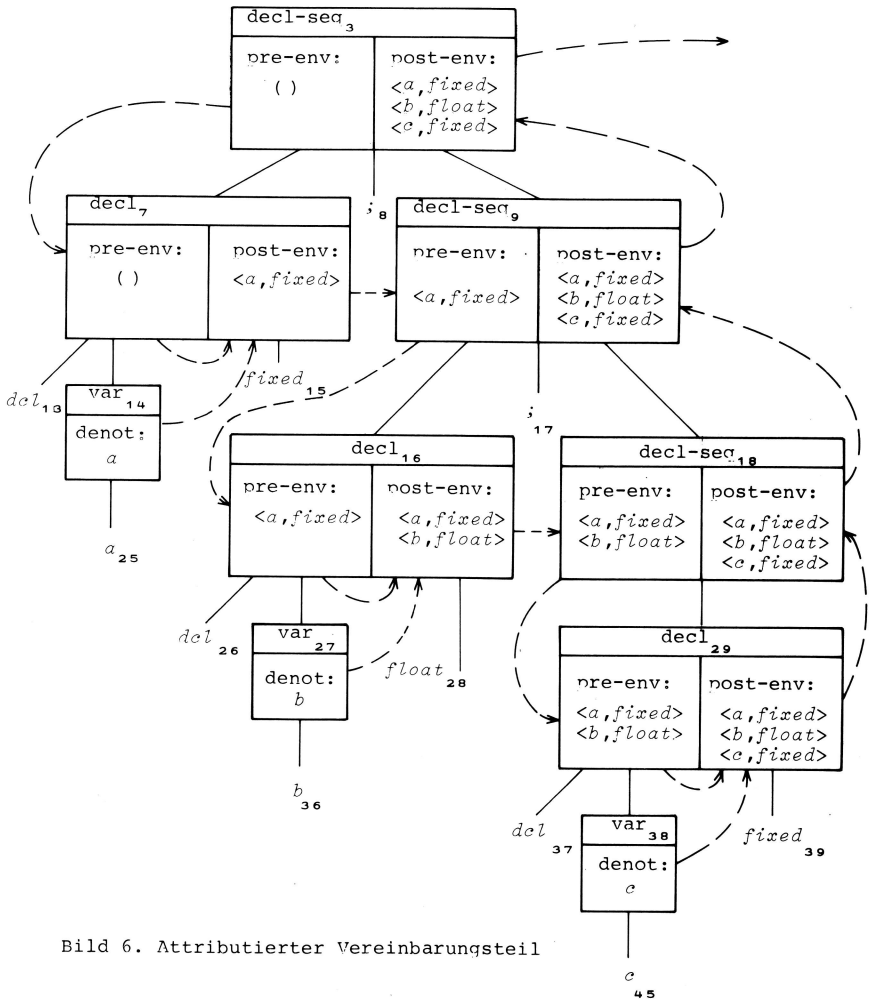


Bild 6. Attributierter Vereinbarungsteil

daß eine Attributauswertungsregel erst dann anwendbar ist, wenn die auf der rechten Seite vorkommenden Attributwerte bereits berechnet sind. Für PA-Programme können die Attribute in einem Durchlauf des Strukturbaumes von links nach rechts ausgewertet werden. Bei praktischen Programmiersprachen ist das in der Regel nicht der Fall, und zwar aus dem gleichen Grund, aus dem diese Programmiersprachen Mehr-Pass-Übersetzer erfordern.
Im allgemeinen gilt deshalb für die Attributierung folgende Vorschrift:

V: Solange noch nicht alle Attributwerte eines Strukturbaumes bestimmt sind, suche eine anwendbare Attributauswertungsregel und wende sie an.

Diese Vorschrift führt nur dann zu einer vollständigen und eindeutigen Attributierung von Strukturbäumen, wenn die attributierte Grammatik wohldefiniert [2] ist.

Hätten wir z.B. im letzten Abschnitt eine der benötigten Attributauswertungsregeln vergessen, dann wäre die vollständige Attributierung des Beispiels offenbar nicht möglich gewesen. Hätten wir dagegen zu viele Attributauswertungsregeln angegeben, wären u.U. für den gleichen Strukturbaum verschiedene Attributierungen möglich gewesen. Eine andere Fehlersituation hängt mit den Abhängigkeiten zwischen Attributwerten zusammen, die wir uns wie im B i l d 6 durch gestrichelte Pfeile gegeben denken: Bilden solche gestrichelte Pfeile in einem Strukturbaum einen geschlossenen *Zyklus*, dann läßt sich keines der auf dem Zyklus liegenden Attribute auswerten, da man zur Auswertung eines jeden Attributs den Wert eines anderen, ebenfalls auf dem Zyklus gelegenen Attributs benötigt.

Gibt es zu der Produktionsregel A ::= ...B... eine Attributauswertungsregel A.x := ..., dann heißt das Attribut x von A *synthetisiert berechnet*. Gibt es zu der Produktionsregel eine Attributauswertungsregel B.y := ..., dann heißt das Attribut y von B *ererbt berechnet*.

Mit Hilfe dieser Begriffe können wir Bedingungen dafür angeben, wann eine attributierte Grammatik wohldefiniert [2] ist:

(i) Alle Knoten zur gleichen syntaktischen Variablen haben Attribute gleichen Namens. [1])

(ii) Ist x Name eines Attributs einer syntaktischen Variablen A, so wird A.x stets entweder synthetisiert oder ererbt berechnet und ist damit entweder ein *synthetisiertes* oder ein *ererbtes* Attribut von A (T a f e l 2). Das Startsymbol der Grammatik hat keine ererbten Attribute.

Tafel 2. Attribute von PA

syntaktische Variable	Attribute	
	ererbt	synthetisiert
decl-seq	pre-env	post-env
decl	pre-env	post-env
ass-seq	env	
ass	env	
var		denot

(iii) In jeder Produktionsregel gibt es zu jedem synthetisierten Attribut der linken Seite und zu jedem ererbten Attribut der rechten Seite genau eine Attributauswertungsregel, d.h. die Menge der Attributauswertungsregeln ist vollständig.

(iv) In keinem Strukturbaum hängen Attributwerte zyklisch voneinander ab.

Im allgemeinen ist Zyklenfreiheit nur mit erheblichem Aufwand feststellbar. In besonderen Fällen, wie zum Beispiel bei PEARL, läßt sie sich jedoch mit vertretbarem Aufwand beweisen.

4. ALADIN

Will man eine umfangreiche attributierte Grammatik übersichtlich und vollständig niederschreiben, so erweist sich dabei ein strenges Schema als zweckmäßig. Die in der PEARL-Beschreibung [1] verwendete Sprache ALADIN bietet ein solches Schema an.

Die attributierte PEARL-Grammatik gliedert sich in mehrere, ähnlich aufgebaute Teilgrammatiken. Im Anhang bringen wir die attributierte Grammatik zu PA im Stile einer solchen Teilgrammatik (vgl. z.B.[1], S. 125-162). Wir erläutern nun die darin vorkommenden ALADIN-Sprachelemente.

Im letzten Abschnitt haben wir *Attributtypen* -- das sind die Wertebereiche, zu denen Attributwerte gehören -- informell eingeführt, z.B. als "Listen der vereinbarten Variablen und deren Typen". ALADIN verlangt die Vereinbarung von Attributtypen in Anlehnung an die Vereinbarung von Datentypen in höheren Programmiersprachen. Die ALADIN-Beschreibung des Typs tp_env von env lautet

```
TYPE tp_env : LISTOF tp_pair
```

Der Attributtyp tp_pair ist dabei zu definieren durch

```
TYPE tp_pair:  STRUCT(s_denot: tp_denot,
                      s_kind: tp_kind)
```

Das heißt, Werte des Attributtyps tp_pair sind zusammengesetzt ("STRUCT") aus Werten vom Typ tp_denot und tp_kind. Als Selektoren für die Komponenten werden dabei s_denot und s_kind verwendet. Ist etwa p

[1]) Der PEARL-Normentwurf verwendet eine etwas schwächere Bedingung, die allerdings schwerer zu formulieren ist.

vom Typ tp_pair, so ist p.s_denot seine erste Komponente.

Die Attributtypen tp_denot und tp_kind sind skalar und werden durch Aufzählung ihrer Elemente definiert.

```
TYPE tp_denot: (sc_A,sc_B,...,sc_Z) ,
TYPE tp_kind: (sc_FIXED,sc_FLOAT) .
```

Weitere Möglichkeiten zur Bildung von Attributtypen finden sich in [1], S.15 f.

Den Namen sind Präfixe vorangestellt, die als Lesehilfe gedacht sind. "tp_" kennzeichnet Namen von Attributtypen, "s_" wird bei Selektoren verwendet, "at_" bei Attributnamen und "sx_" bei syntaktischen Variablen (vgl. [1], S. 14).

Konstantendefinitionen führen Namen für Werte ein. Das dient vor allem der Abkürzung und der Lesbarkeit. Außerdem treten dadurch implementierungsabhängige Werte nur bei den Konstantendefinitionen auf und können deshalb für jede Implementierung leicht ergänzt werden. Wir benutzen die Konstanten c_imp_anzvar und c_non_BASIC_PA in diesem Sinn.

Jede *syntaktische Variable* muß vereinbart werden. Dabei sind die Namen und Typen ihrer Attribute anzugeben. Ererbte Attribute sind durch den Zusatz "INH" (für engl. inherited) gekennzeichnet. Die synthetisierten Attribute werden nicht besonders gekennzeichnet.

ALADIN-Regeln enthalten neben einer Produktionsregel die zugehörigen Bedingungen und Attributauswertungsregeln. Im folgenden Schema dürfen die nicht benötigten Teile weggelassen und die CONDITION-Teile wiederholt werden (vgl.[1], z.B. S. 44 und 134 f).

```
RULE    Produktionsregel
   SUBSET
      CONDITION  Teilmengenbedingung
   STATIC
      Attributauswertungsregeln
      CONDITION Kontext- oder Implementierungsbedingung
   DYNAMIC
      Beschreibung der Wirkung von
      Sprachelementen in "technical
      English"
END
```

ALADIN erlaubt die Definition von *Funktionen* und ihre Verwendung in Ausdrücken. Ihre Definition lehnt sich an die Vereinbarung von Funktionsprozeduren in höheren Programmiersprachen an (vgl.[1], S. 23). ALADIN stellt eine Reihe vordefinierter Funktionen zur Verfügung (vgl.[1],S.23), die für unseren PEARL-Ausschnitt ausreichen.

Die env-Attribute sind Listen von Paaren, die jeweils aus einem Variablennamen und dem zugehörigen Typ bestehen. Da sich die Paare durch die Variablennamen unterscheiden, erklären wir Variablennamen zu *Schlüsseln* ("KEY"), indem wir "KEY s_denot" bei der Definition des Attributtyps tp_env anfügen. Diese Konvention spielt bei der Verwendung der vordefinierten Funktionen eine Rolle.

Wir benötigen folgende vordefinierte Funktionen:

LENGTH(p_list) = Länge der Liste p_list.

KEY_IN_LIST(p_key,p_list) = "wahr", falls p_list ein Element mit dem Schlüssel p_key enthält, und "falsch" sonst.

SELECT_BY_KEY(p_key,p_list) = das Glied von p_list mit dem Schlüssel p_key, falls es ein solches gibt und es eindeutig bestimmt ist, undefiniert sonst.

Der Anhang enthält die in ALADIN formulierte Grammatik zu PA.

Literatur

[1] DIN 66 253, Teil 2: Programmiersprache PEARL, FULL PEARL. Berlin, 1980.

[2] K n u t h, D.E.: Semantics of context-free languages. Math. System Theory 2 (1968) S. 127-145 *und* Math. Systems Theory 5 (1971) S. 95.

[3] M a r c o t t y, M., L e d g a r d, H.F., B o c h m a n n, G.V.: A Sampler of Formal Definitions. Computing Surveys Vol. 8, No. 2 (1976) S. 191-276.

[4] W i l h e l m, R.: Attributierte Grammatiken. Informatik-Spektrum Bd. 2, No. 3 (1979) S. 123-130.

ANHANG

Attribute Types

```
TYPE tp_env:    LISTOF tp_pair KEY s_denot;
TYPE tp_pair:   STRUCT(s_denot: tp_denot, s_kind: tp_kind);
TYPE tp_denot:  (sc_A,sc_B,...,sc_Y,sc_Z);
TYPE tp_kind:   (sc_FIXED,sc_FLOAT);
```

Constants

```
CONST c_imp_anzvar: 10;
    % Kommentarzeilen, wie diese beginnen mit dem Prozentzeichen. Anstelle von 10
    % hätte man auch jede andere Zahl wählen können, z.B. 255.
CONST c_non_BASIC_PA: TRUE;
    % Mit c_non_BASIC_PA = TRUE  ist die Teilmengenbedingung immer wahr. Ersetzt man
    % TRUE durch FALSE, so ist die Teilmengenbedingung nur dann wahr, wenn die
    % Zeichenfolge ein BASIC-PA-Programm ist.
```

Symbols

```
NONTERM sx_PA-program: ;
NONTERM sx_decl_seq: at_pre_env: tp_env INH, at_post_env: tp_env;
NONTERM sx_ass_seq:  at_env: tp_env INH;
NONTERM sx_decl:     at_pre_env: tp_env INH, at_post_env: tp_env;
NONTERM sx_ass:      at_env: tp_env INH;
NONTERM sx_var:      at_denot: tp_denot;
```

Rules

```
RULE sx_PA_program ::=  'BEGIN' sx_decl_seq ';' sx_ass_seq 'END'
   % Terminale Zeichen sind in Apostrophe einzuschließen.
   STATIC sx_ass_seq.at_env := sx_decl_seq.at_post_env;
          sx_decl_seq.at_pre_env := tp_env()
           % leere Liste vom Typ tp_env.
   CONDITION LENGTH(sx_ass_seq.at_env)≤·c_imp_anzvar
           % Implementierungsbedingung
END;
```

```
RULE sx_decl_seq ::= sx_decl ';' sx_decl_seq
   STATIC sx_decl_seq[1].at_post_env := sx_decl_seq[2].at_post_env;
          sx_decl.at_pre_env := sx_decl_seq[1].at_pre_env;
          sx_decl_seq[2].at_pre_env := sx_decl.at_post_env;
END;

RULE sx_decl_seq ::= sx_decl
   STATIC sx_decl_seq.at_post_env := sx_decl.at_post_env;
          sx_decl.at_pre_env := sx_decl_seq.pre_env
END;

RULE sx_ass_seq ::= sx_ass ';' sx_ass_seq
   STATIC sx_ass.at_env := sx_ass_seq[1].at_env;
          sx_ass_seq[2].at_env := sx_ass_seq[1].at_env
END;

RULE sx_ass_seq ::= sx_ass
   STATIC sx_ass.at_env := sx_ass_seq.at_env
END;

RULE sx_decl ::= 'DCL' sx_var 'FIXED'
   STATIC sx_decl.at_post-env := sx_decl.at_pre_env +
                                 tp_env(tp_pair(sx_var.at_denot,sc_FIXED))
          % Der Operator + verkettet zwei Listen gleichen Typs -- hier des Typs tp_env.
          % tp_pair(...,...) setzt die beiden Argumente zu einem Paar zusammen und
          % tp_env(...) macht daraus eine einelementige Liste.
   CONDITION NOT KEY_IN_LIST(sx_var.at_denot,sx_decl.at-pre_env)
          % Kontextbedingung KB1
END;

RULE sx_decl ::= 'DCL' sx_var 'FLOAT'
     wie oben aber mit FLOAT statt FIXED
END;

RULE sx_ass ::= sx_var ':=' sx_var
   SUBSET CONDITION c_non_BASIC_PA OR
             SELECT_BY_KEY(sx_var[1].at_denot,sx_ass.at_env).s_kind =
             SELECT_BY_KEY(sx_var[2].at_denot,sx_ass.at_env).s_kind
             % Teilmengenbedingung. Beachte die logische Äquivalenz von
             % A V B  und  ¬A→B
   STATIC CONDITION SELECT_BY_KEY(sx_var[2].at_denot,sx_ass.at_env).s_kind = sc_FIXED
               OR   SELECT_BY-KEY(sx_var[1].at_denot,sx_ass.at_env).s_kind = sc_FLOAT
             % Kontextbedingung KB3
      CONDITION KEY_IN_LIST(sx_var[1].at_denot,sx_ass.at_env) AND
                KEY_IN_LIST(sx_var[2].at_denot,sx_ass.at_env)
                % Kontextbedingung KB2
END;

RULE sx_var ::= 'A'
   STATIC  sx_var.at_denot := sc_A
END;
Weitere Regeln sind zu ergänzen für B,C,...,Y,Z.
```

Erläuterungen zur attributierten Grammatik für PEARL

Dr. U. Kastens, Karlsruhe

1. Überblick

Die Definition der Programmiersprache PEARL und ihrer Teilsprache Basic PEARL ist im Normentwurf DIN 66253 Teil 2 standardisiert worden [1]. Er soll sowohl Implementierern als auch Anwendern als Referenzdokument dienen, das unterschiedliche Interpretationen soweit wie möglich ausschließt. Deshalb ist der größte Teil der Spracheigenschaften mit formalen Methoden beschrieben: die syntaktische Struktur und die Kontextabhängigkeiten durch eine attributierte Grammatik (AG) und die Bedeutung - soweit sie die asynchrone Ausführung von Programmen betrifft - durch Petri-Netze.

Anders als Lehrbücher oder Einführungen zu Programmiersprachen stellen formale Sprachdefinitionen naturgemäß höhere Anforderungen an die Vorkenntnisse der Leser. Klare Beschreibungskonzepte können und sollen zwar das Verständnis erleichtern, aber die Notwendigkeit zur Einarbeitung in das Dokument und seine Beschreibungsmethoden vermeiden sie nicht vollständig.

Wir wollen in diesem Papier Verständnishilfen geben, die den Zugang zu der formalen Definition der statischen Spracheigenschaften durch die AG erleichtern. Wir wenden uns dabei insbesondere an die Leser, die den Normentwurf heranziehen wollen, um Fragen zu Spracheigenschaften zu beantworten. Sprachimplementierer, die den Normentwurf als Teil des Pflichtenheftes oder der Übersetzerspezifikation benutzen, werden sich über den Rahmen dieser Erläuterungen hinaus Detailkenntnisse erarbeiten müssen.

Im zweiten Abschnitt werden die Grundprinzipien der Beschreibungsmethode "Attributierte Grammatiken" erläutert. Der dritte Abschnitt führt in das Arbeiten mit dem Normdokument ein und gibt am konkreten Beispiel Hinweise, wie man die Gliederung und verschiedene Arten von Querverweisen nutzt, um Fragen zu Spracheigenschaften zu beantworten. In Abschnitt 4 erläutern wir einige Beschreibungstechniken, die zur Definition zentraler Spracheigenschaften, wie Gültigkeit von Definitionen, Datentypen und die Teilspracheigenschaft angewandt werden. Ein Überblick über diese Techniken ist wichtig, weil sich diese Spracheigenschaften auf viele Elemente der Sprache auswirken. Die konkrete Notation der AG in der Beschreibungssprache ALADIN wird in Abschnitt 5 einführend erklärt.

2 Attributierte Grammatiken als Sprachdefinition

Eine AG definiert sowohl die syntaktische Struktur von Programmen einer Sprache als auch kontextabhängige Eigenschaften und Bedingungen. Anhand einer solchen Sprachdefinition kann deshalb entschieden werden, ob ein gegebenes Programm statisch korrekt ist, d.h. ob ein Sprachübersetzer es akzeptieren muß. Die Frage, ob ein Programm zusammen mit einem Eingabe-Datensatz auch ausführbar ist, wird im allgemeinen nicht durch die AG beantwortet, sondern durch die Definition der dynamischen Semantik - der Bedeutung der Sprachelemente.

Aus vielen informellen oder teilweise formalisierten Sprachdefinitionen ist der folgende Beschreibungsstil bekannt: Die syntaktische Struktur von Programmen wird durch eine kontext-freie Syntax, z.B. in

Backus-Naur-Form angegeben. Hinzu kommen verbal formulierte Aussagen, z.B. über die Gültigkeitsbereiche von Definitionen und die Typen von Ausdrücken, die nicht kontext-frei erfaßbar sind. (Die dem Normentwurf vorangehenden PEARL-Beschreibungen (z.B. [2], [3]) folgen diesem Stil.) Eine AG umfaßt auch diese kontextabhängigen Spracheigenschaften. Da das "Skelett" einer AG eine kontext-freie Grammatik in gewohnter Notation ist, kommt die Beschreibungsmethode dem gewohnten Denken entgegen, das zwischen syntaktischer Struktur und kontextabhängigen Regeln unterscheidet.

Ebenso wie verbale Definitionen Strukturelementen der Sprache Eigenschaften zuordnen (z.B. "der Typ eines Ausdrucks"), ordnet eine AG den Symbolen der kontext-freien Grammatik Attribute zu (z.B. Ausdruck.Typ). Die Analogie läßt sich fortsetzen: Eine verbale Aussage wie "Der Typ einer Vergleichsoperation ist BOOL" entspricht einer Attributregel

Vergleich.Typ := BOOL,

die der Produktion

Vergleich ::= Formel '=' Formel

zugeordnet ist. Restriktionen wie "Die linke und die rechte Seite einer Zuweisung müssen gleichen Typ haben" werden als Kontextbedingungen über Attributen formuliert, z.B. zur Produktion

Zuweisung ::= Variable ':=' Ausdruck

die Bedingung

CONDITION Variable.Typ = Ausdruck.Typ

Die Beschreibungsmethode "attributierte Grammatik" beruht also auf folgenden Prinzipien:

1. Eine AG basiert auf einer kontext-freien Grammatik.

2. Den Terminalen und Nichtterminalen der kontext-freien Grammatik werden Attribute zugeordnet, welche Eigenschaften dieser Sprachelemente beschreiben.

3. Den Produktionen der kontext-freien Grammatik werden Attributregeln zugeordnet, welche beschreiben, wie Attribute (Eigenschaften) zu bestimmen sind. Kontextabhängigkeiten werden durch Bezugnahme auf Attribute aus dem Kontext ausgedrückt.

4. Durch Kontextbedingungen zu Produktionen wird die Menge der syntaktisch korrekten Programme eingeschränkt auf die Menge der statisch korrekten (übersetzbaren) Programme.

Wir verzichten hier auf ein Beispiel und verweisen auf [4]. Eine exakte Definition attributierter Grammatiken und weitere Beispiele findet man z.B. in [6].

Will man anhand einer AG feststellen, ob ein gegebenes Programm korrekt ist, so geht man wie folgt vor:

1. Man bestimmt die syntaktische Struktur des Programms. Dazu leitet man das Programm unter Anwendung von Produktionen der kontext-freien Grammatik aus dem Startsymbol (hier: sx_PEARL_Programm) ab und stellt die Ableitungsschritte als Strukturbaum dar. Das Startsymbol ist dessen Wurzel, die Knoten sind Terminale und Nichtterminale der Grammatik, die Symbole des Programms sind Blätter des Baums. Zu jeder angewandten Produktion verbinden Kanten die Symbole der rechten Seite mit dem Nichtterminal der linken Seite der Produktion. Eine Reihe von Symbolen kann man aus dem Strukturbaum weglassen, da sie ausschließlich syntaktische Bedeutung haben (z.B. BEGIN, END, ;). Man spricht dann auch von einem abstrakten Strukturbaum.

2. Der Strukturbaum wird durch Attribute zum attributierten Strukturbaum erweitert. Jedem Knoten ordnet man die Attribute zu, die zum entsprechenden Symbol der Grammatik gehören.

3. Man bestimmt die Werte der Attribute von Baumknoten (und dadurch die kontextabhängigen Eigenschaften des Programms). Dazu wendet man diejenigen Attributregeln an, die den Produktionen

zugeordnet sind, welche zur Herstellung des Baums benutzt wurden. Häufig werden in einer Attributregel Attribute benutzt, die nicht durch Attributregeln zur gerade betrachteten Produktion bestimmt werden. Ihre Attributregeln findet man bei den in der unmittelbaren Umgebung angewandten Produktionen. Um den durch Attribute beschriebenen Informationsfluß nachzuvollziehen (z.B. von Objektdeklarationen zu Anwendungen) muß man im Strukturbaum entlang der Kanten auf- und absteigen.

4. Um die Korrektheit eines Programms zu ermitteln, wertet man wie in (3) die Kontextbedingungen zu den angewandten Produktionen aus.

Ist eine AG wohldefiniert [8], so können nach diesem Verfahren in jedem Strukturbaum die Werte aller Attribute eindeutig bestimmt werden. Die Wohldefiniertheit der PEARL-AG ist mit Hilfe des Übersetzer-erzeugenden Systems GAG [7] nachgewiesen worden.

Zur Beantwortung spezieller Fragen ist es meist nicht nötig, die Attributierung eines Programm vollständig nachzuvollziehen. Man kann sich auf Teile eines Strukturbaums (z.B. eine Zuweisung mit ihrer unmittelbaren Umgebung) und auf wenige Attribute (z.B. die Typ-Attribute) beschränken. Zusammenhänge über einen größeren Kontext lassen sich meist aus der Kenntnis einiger zentraler Beschreibungskonzepte erschließen (siehe Abschnitt 4).

3 Gliederung der attributierten Grammatik und Querverweise

In diesem Abschnitt stellen wir die Gliederung der AG im Normentwurf vor. Anhand eines Beispiels zeigen wir, wie man Antworten auf Fragen nach PEARL-Eigenschaften in der AG findet.

Die AG für PEARL ist in fünf Kapitel (Parts) gegliedert, welche jeweils die Definitionen eines Teilbereichs der Sprache zusammenfassen. Jedes Kapitel beginnt mit der Definition der dort angewandten Attributtypen. Es folgen Abschnitte, in denen jeweils zuerst die Attributzuordnungen für Symbole, dann die Regeln für diese Symbole und schließlich die darin angewandten Funktionen angegeben sind. Die Typ- und Symboldefinitionen sind alphabetisch, die Regeln und Funktionen sind im Sinne einer sukzessiven Verfeinerung angeordnet (in Part Six und Part Seven auch alphabetisch). Diese Gliederung ist hilfreich für den ersten Einstieg in die AG, bei dem man zwar sein Problem einem Themenkreis (z.B. "Tasking") zuordnen kann, aber noch keine Begriffe und Namen aus der AG kennt.

Die gesamte AG ist mit Zeilennummern versehen, deren führende Stellen den Abschnitt kennzeichnen, z.B. 15.0274. (Im folgenden verweisen wir auf die AG im Normentwurf mit diesen Zeilennummern: {15.0274}.) Die Anhänge B und C stellen mit Hilfe der Zeilennummern Querverweise in die AG her: Anhang B enthält die kontext-freien Produktionen - alphabetisch sortiert nach dem Namen des Symbols auf der linken Seite - zusammen mit einem Verweis auf die Stelle der zugehörigen AG-Regel. Anhang C gibt zu jedem Namen aus der AG die Definitionsstelle(n) an.

Mit Hilfe der Anhänge beantwortet man leicht folgende Fragentypen:

(a) Wie wird das Symbol sx_exp kontext-frei abgeleitet? Im Anhang B die Produktionen anhand der alphabetischen Ordnung verfolgen!

(b) Welche Attributregeln gibt es zur Produktion sx_glob_id ::= sx_id? Aus Anhang B dem Verweis in die AG folgen!

(c) Welche Attribute hat sx_exp? Aus Anhang C dem Verweis auf die Symboldefinition folgen!

(d) Wie ist die Funktion f_coerce (der Typ tp_type,...) definiert? Mit Anhang C die Definition aufsuchen!

Als Beispiel wollen wir die Beantwortung folgender Frage vorführen: Welche Typen sind in Basic PEARL auf der linken Seite einer Zuweisung zulässig?

Den Einstieg finden wir z.B., indem wir im Anhang B die Ableitung von sx_statement nach sx_assignment verfolgen und in der AG die Regeln in {16.0959} und {16.0994} aufsuchen. (Die Zuweisung gehört zum Abschnitt "Expressions", da sie in PEARL auch an Ausdrucksposition stehen kann.)

Wegen der Teilsprachbedingung in {16.0998} brauchen wir die zweite Alternative nicht zu betrachten (siehe Abschnitt 4.1). In {16.0964} finden wir eine Bedingung zu einem Attribut der linken Seite der Zuweisung

```
CONDITION
f_assignable(sx_destination.at_attr.s_type)
AND
f_no_INV_component(sx_destination.at_attr)
```

Hinzu kommt eine Teilsprachbedingung über dasselbe Attribut zur zweiten Regel für sx_destination in {16.1061}:

```
sx_destination.at_attr.s_type
   IS tp_basic_type
```

In {16.0185} ist der Typ von at_attr mit tp_attr angegeben. Mit Anhang C finden wir die Typdefinition dazu in {15.0147}: tp_attr beschreibt Zugriffsfunktionen durch Angabe der Invariabilität und des Typs des Bezugsobjekts. In {15.0274} ist die Abstraktion der PEARL-Typen (tp_type) als Vereinigung von tp_basic_type mit einigen anderen Typen definiert. Aus der Definition für tp_basic_type in {15.0156} entnimmt man, daß die Teilsprachbedingung in {16.1061} nur für die PEARL-Typen FIXED, FLOAT, BIT, CHAR, CLOCK oder DUR erfüllt ist. Anhand der Definition von f_assignable in {16.3004} stellt man fest, daß sie für diese Typen das Ergebnis TRUE hat. Der erste Teil der Bedingung in {16.0964} ist für Basic PEARL also immer wahr. Der zweite Teil ist erfüllt, wenn es sich um eine Zugriffsfunktion für variable Objekte handelt. Diese Aussage erhält man durch Untersuchen der Funktion f_no_INV_component in {16.2960} für den Fall tp_attr. Die Antwort auf unsere Frage lautet also: In Basic PEARL dürfen auf der linken Seite von Zuweisungen variable Objekte der Typen FIXED, FLOAT, BIT, CHAR, CLOCK oder DUR stehen.

4 Beschreibungstechniken

Wie in allen höheren Programmiersprachen gibt es auch in PEARL einige Spracheigenschaften, die sich auf Sprachelemente in allen Teilbereichen der Sprache auswirken, z.B. die Gültigkeitsregeln für Definitionen, der Typbegriff für Objekte, die Typbestimmung in Ausdrücken. Im Normentwurf gehören auch die Teilsprachbedingungen für Basic PEARL zu solchen übergreifenden Eigenschaften. In diesem Abschnitt wollen wir dem Leser zumindest eine grobe Vorstellung von den Techniken vermitteln, mit denen diese Eigenschaften beschrieben werden, um die Beantwortung spezieller Fragen zu PEARL zu erleichtern.

4.1 Teilsprachbedingungen

Die Definition von Basic PEARL - eine echte Teilsprache von PEARL - ist in die PEARL-Definition integriert. Kontext-freien Produktionen, die in Basic PEARL nicht anwendbar sind, ist eine SUBSET-Bedingung der Form

```
SUBSET CONDITION c_non_Basic
```

zugeordnet, z.B. in {16.0997}. Produktionen, die in Basic PEARL nur anwendbar sind, wenn eine einschränkende Bedingung B erfüllt ist, sind durch

```
SUBSET CONDITION c_non_Basic OR B
```

gekennzeichnet (z.B. in {16.1057}). Für die Definition von Basic PEARL nimmt man an, daß c_non_Basic der Wert FALSE zugeordnet ist. Dann ist von obigen Bedingungen nur die zweite erfüllbar. Für die PEARL-Definition sind alle SUBSET-Bedingungen erfüllt, da in diesem Fall c_non_Basic den Wert TRUE hat. Produktionen, die in Basic PEARL nicht ableitbare Symbole weiter ableiten, sind im allgemeinen keine SUBSET-Bedingungen zugeordnet:

```
sx_definition ::= sx_PRECEDENCE_def
```

wird durch SUBSET-Bedingung für Basic PEARL ausgeschlossen, die deshalb nicht erreichbare Produktion

```
sx_PRECEDENCE_def ::= ...
```

jedoch nicht.

4.2 Gültigkeitsbereiche - Bezeichneridentifikation

In höheren Programmiersprachen definieren die Gültigkeitsbereichsregeln, wie zu einem angewandten Auftreten eines Bezeichners die zugehörige Definition bestimmt wird. In der AG beschreiben Attribute und Attributregeln diesen "Informationsfluß" von den Definitionen zu den Anwendungsstellen. Wir wollen ihn hier entgegen seiner Richtung verfolgen: Das typische angewandte Auftreten eines Bezeichners repräsentiert die Produktion in {16.1110}:

RULE sx_access_id ::= sx_id

Die zugehörige Definition wird mit Hilfe der rekursiven Funktion f_identify_id {12.0433} bestimmt. Sie liefert das erste Element aus einer Liste von Definitionen (2. Parameter), dessen Bezeichner-Komponenten mit dem gesuchten Bezeichner (1. Parameter) übereinstimmt. Falls es kein solches Element gibt, ist die Kontextbedingung in {12.0459} verletzt. Diese Liste von Definitionen ist der Wert des Attributs at_id_defs zum kleinsten die Anwendungsstelle umfassenden sx_PROBLEM_division, sx_block_Body, sx_detach_block_Body oder sx_loop_scope. (Dies ist die Bedeutung des INCLUDING-Konstrukts in {16.1117}, vergl. Abschnitt 5.3.2.)

Die vier Symbole repräsentieren die Gültigkeitsbereiche in PEARL-Programmen. Als Beispiel für die Berechnung eines Attributs at_id_defs betrachten wir die Attributregel in {12.0257}: Sie bestimmt den Wert von

sx_block_body.at_id_defs

durch Konkatenation von vier Teillisten: den lokalen Definitionen, den Markendefinitionen aus dem Anweisungsteil, den Parameterdefinitionen, falls der Block ein Prozedurrumpf ist, und den im kleinsten umfassenden Gültigkeitsbereich geltenden Definitionen. Da die zum Block globalen Definitionen in der Liste am Ende stehen, werden sie bei der Bezeichneridentifikation (f_identify_id) nur dann gefunden, wenn sie nicht durch eine gleichbezeichnete lokale Definition verdeckt sind.

Von dieser Identifikationstechnik wird in folgenden Fällen abgewichen: Globale Bezeichner in Spezifikationen werden anhand des Paares (Bezeichner, Modul-Qualifikation) in der Liste aller über Modulgrenzen hinweg gültigen Definitionen, die sx_PEARL_program zugeordnet ist, identifiziert {13.0050}.

Operatorbezeichner können "überladen" sein, d.h. es können in einem Block mehrere gleich bezeichnete Operatordefinitionen gültig sein, die mit Hilfe der Operandentypen unterschieden werden {16.0592}.

Die Selektoren von Verbund- oder DATION-Komponenten werden in den Komponentenlisten des Verbund- oder DATION-Typs identifiziert {16.1195}.

Für Typbezeichner wird die oben beschriebene Identifikationstechnik nicht angewandt, da sie z.B. bei rekursiv definierten Typen wie in

TYPE t STRUCT [k REF t, ...]

zyklische Attributabhängigkeiten verursachen würde. Stattdessen wird jede Typdefinition "eindeutig umbenannt" {15.0481}. Beim angewandten Auftreten eines Typbezeichners (sx_type_indicant, {15.0973}) wird zunächst nur diese neue eindeutige Benennung identifiziert. Dazu werden anstelle der Attribute at_id_defs die Attribute at_pre_defs benutzt. Erst bei "späterer" Analyse des Typs wird der neuen Benennug die Typdefinition zugeordnet.

4.3 Beschreibung von PEARL-Typen

Aussagen über Typen werden in PEARL zu zahlreichen Sprachelementen gemacht, z.B. zu Ausdrücken oder zu Typangaben in Definitionen. Attribute, welche solche Eigenschaften beschreiben, haben einen Wertebereich, der eine Abstraktion der statisch relevanten Eigenschaften von PEARL-Typen ist. Der Wertebereich wird in {15.0274} durch den Attributtyp tp_type als Vereinigung einiger Untertypen definiert. Der PEARL-Typ FIXED(5) wird z.B. durch tp_FIXED_type(5) beschrieben -

ein Verbundwert vom Typ tp_FIXED_type, dessen einzige Komponente den Wert 5 hat (vgl. Abschnitt 5.2). Untertypen, die keine weiteren unterscheidenden Merkmale haben, wie tp_SIGNAL_type in {15.0257}, sind als einwertige Aufzählungstypen definiert. Zusammengesetzte Typen, z.B. tp_PROC_type in {15.0231}, werden durch Verbunde definiert, deren Komponenten (z.B. s_params, s_RESULT) die Typeigenschaften beschreiben. Dazu wird häufig wieder auf den Wertebereich von tp_type bezug genommen, z.B. s_RESULT (siehe Abschnitt 5.2).

Der Begriff der "Zugriffsfunktion" ist in diesem Zusammenhang durch

```
TYPE tp_attr: STRUCT (s_INV : BOOL,
                      s_type: tp_type)
```

in {15.0147} abstrahiert. Der PEARL-Typ REF FIXED(5) wird also durch

```
tp_attr (FALSE, tp_FIXED_type(5))
```

abstrahiert und beschreibt "ganzzahlige Variable". In PEARL gehört zum Konzept "Zugriffsfunktion" der Begriff der "Variabilität": Mit Zugriffsfunktionen, deren Typ durch INV gekennzeichnet ist, darf der Wert des Bezugsobjekts nicht verändert werden, z.B. REF INV FIXED(5). Dies wird ausgedrückt durch die erste Komponente s_INV von tp_attr:

```
tp_attr (TRUE, tp_FIXED_type(5)).
```

PEARL-Deklarationen und -Spezifikationen, z.B.

```
DCL i FIXED(5)
```

führen Zugriffsfunktionen ein. Deshalb enthält die Beschreibung von Definitionen (tp_access_def in {11.0056}) eine Komponente vom Typ tp_attr, die in diesem Beispiel den Wert tp_attr(FALSE, tp_FIXED(5)) hat.

Der Wertebereich von tp_type umfaßt außer den Abstraktionen der PEARL-Typen auch solche, die in keinem syntaktisch korrekten Programm auftreten können, z.B.

```
tp_attr(FALSE,
        tp_attr(FALSE, tp_PROC_type(...))),
```

was REF REF PROC entspräche, und solche, die nur in bestimmtem Kontext zulässig sind (z.B. darf ein Interrupt nicht Prozedurergebnis sein). Die Einschränkung des Wertebereichs auf die im jeweiligen Kontext zulässigen PEARL-Typen definieren die Funktionen f_type_consistent und f_type_consistency_check in {15.1578}. Den Kontext charakterisiert der Parameter p_restriction (z.B. sc_is_RESULT_tpye).

4.4 Typbestimmung in Ausdrücken

In PEARL werden wie in allen typisierten Programmiersprachen Aussagen über Typen von Ausdrücken gemacht. Da es in PEARL implizit angewandte Anpassungsoperationen (z.B. Typausweitung, Dereferenzieren, Aufruf ohne Parameter) gibt, unterscheidet man zwischen den Typen vor und nach Anwendung impliziter Anpassungen. Sie werden beschrieben durch die Attribute at_pre_type und at_post_type zu Symbolen der Ausdrucksgrammatik.

Das abgeleitete Attribut at_pre_type (siehe Abschnitt 5.1) wird z.B. für Bezeichner durch Bezeichneridentifikation {16.1110}, für ein- und zweistellige Formeln durch Operatoridentifikation {16.0542}, {16.0674} und für bedingte Ausdrücke durch Typabgleich {16.0172} bestimmt.

Das erworbene Attribut at_post_type (siehe Abschnitt 5.1) wird im Kontext bestimmt, in den der Ausdruck eingebettet ist. Wir unterscheiden hier zwei Situationen:

- Der Kontext bestimmt den Zieltyp vollständig und unabhängig von at_pre_type. Dies gilt z.B. für die rechte Seite von Zuweisungen {16.0968}.
- Der Kontext bestimmt nur die "Typklasse" für at_post_type, in CONT sx_Basic_exp z.B. eine nicht näher spezifizierte Zugriffsfunktion {16.1032}. Abhängig von at_pre_type wird daraus der vollständige Typ berechnet. Zur Beschreibung solcher Typklassen dient der Attributtyp tp_type_pattern {15.0348}, dessen Wertebereich eine Vergröberung des Wertebereichs von tp_type ist.

Für jeden Ausdruck muß gelten: at_pre_type ist an at_post_type anpaßbar, wie es Attributbedingungen der Form

```
CONDITION
   f_coercible (sx_exp.at_pre_type,
                sx_exp.at_post_type)
```

z.B. in {16.0646} verlangen. Die Anpaßbarkeit ist in {15.2669} wie folgt erklärt: t1 = at_pre_type muß durch Anwenden einer (möglicherweise leeren) Sequenz von Anpassungsoperationen in einen Typ t2 transformierbar sein, so daß t2 mit t3 = at_post_type verträglich ist. Die Relation "verträglich" wird durch f_types_compatible (t3,t2) in {15.2001} definiert. Sie ist nicht symmetrisch: Der Typ t3 = at_post_type kann z.B. in Bezug auf die Zugriffsrechte (INV) restriktiver sein als t2: Für

at_pre_type = t1 = t2 = REF FIXED(5)

und

at_post_type = t3 = REF INV FIXED(5)

gilt

f_types_compatible (t3,t2) und
f_coercible (t1,t3)

Die umgekehrte Richtung ist unzulässig, da die Zugriffsrechte erweitert würden. Die Verträglichkeitsbedingungen werden in Bezug auf die Zugriffsrechte mit "steigender Referenzstufe" verschärft (Parameter p_kind der Funktion f_type_compatibility_check).

5 Erläuterungen zu ALADIN

Die AG für PEARL ist in der Beschreibungssprache ALADIN (a Language for attributed definitions) formuliert. ALADIN verbindet Eigenschaften, die typisch sind für AG-Definitionen (Attributzuordnung, Attributregeln, Produktionen), mit Sprachelementen und Notationen aus höheren Programmiersprachen (Datentypen, Ausdrücke, Funktionen). In diesem Abschnitt geben wir eine kurze Charakterisierung der Sprache und erläutern anhand zahlreicher Beispiele aus dem Normentwurf die Bedeutung und Anwendung der Sprachelemente, von denen wir annehmen, daß sie nicht aus Programmiersprachen bekannt sind. Der Normentwurf selbst enthält eine Beschreibung aller in der PEARL-AG verwendeten Sprachelemente ([1], S. 13 ff); eine vollständige ALADIN-Definition ist in [5] enthalten.

ALADIN ist (wie PEARL, ALGOL68 oder PASCAL) eine streng typisierte Sprache.

Der Wertebereich jedes Attributs ist durch seinen Typ exakt definiert. Daraus und aus seinem Namen kann man häufig schon ohne Kenntnis der Attributregeln ersehen, welche Eigenschaft das Attribut beschreibt. Formeln und Ausdrücke müssen den Typregeln von ALADIN genügen. Der Zwang zur Einhaltung dieser Regeln vermeidet Beschreibungsfehler.

ALADIN ist (wie LISP) eine streng applikative, variablenfreie Sprache. Da Attribute statische Eigenschaften von Sprachelementen beschreiben, sind ihre Werte unveränderlich. Es widerspräche dem Grundprinzip attributierter Grammatiken, Attribute als Variable anzusehen. Als Konsequenz gibt es in ALADIN keine Elemente zur Ablaufsteuerung (z.B. Schleifen oder bedingte Anweisungen), sondern nur Ausdrücke, die z.B. durch Fallunterscheidungen strukturiert sind. Die Reihenfolge, in der Attributregeln angegeben sind, ist bedeutungslos und nicht als Ausführungsreihenfolge zu verstehen.

ALADIN sieht vor, daß die Objekte der AG (Attribute, Symbole, Typen, Funktionen, usw.) einen Namen haben, der in einer Definition explizit eingeführt wird. Durch suggestive Wahl der Namen wird die Lesbarkeit der AG wesentlich erhöht. Einige Konventionen zur Namensgebung werden in [1], S. 14 gegeben.

Der Kern einer AG in ALADIN ist eine Menge von Regeln, die jeweils eine kontext-freie Produktion mit den zugeordneten Attributregeln und Attributbedingungen zusammenfassen. Hinzu kommen Symboldefinitionen, die den Symbolen Attribute zuordnen, Typdefinitionen zur Beschreibung der Attribut-Wertebereiche, Definitionen von Funktionen und von Werten, die in den Attributregeln angewandt werden.

5.1 Attributzuordnung

Jedes Nichtterminal der kontext-freien Grammatik wird in einer Symboldefinition eingeführt, welche ihm Attribute bestimmter Typen zuordnet, z.B. in {16.0201}.

```
NONTERM sx_exp:
        at_pre_type   : tp_type,
        at_post_type  : tp_type INH,
        at_value      : tp_value,
        at_PRECEDENCE : tp_precedence;
```

In Attributregeln werden die Attribute in der Form sx_exp.at_pre_type notiert. Der Zusatz INH zum Attributtyp gibt an, daß es sich um ein erworbenes (engl.: inherited) Attribut handelt, mit dem ein Informationsfluß im Strukturbaum von oben (Wurzel) nach unten beschrieben wird – in diesem Beispiel der Typ des Ausdrucks, der vom umfassenden Kontext (z.B. rechte Seite einer Zuweisung) verlangt wird. Die nicht so gekennzeichneten Attribute sind abgeleitet (engl.: synthesized) und beschreiben einen Informationsfluß von unten nach oben im Strukturbaum.

Auch einige Terminale werden durch Symboldefinitionen eingeführt, um ihnen Attribute zuzuordnen, z.B. in {16.0300}

```
TERM sx_FIXED_const_denot :
        at_integer_value : INT
```

Die Attribute dieser Terminale nehmen eine Sonderstellung ein: Sie beschreiben eine Eigenschaft des Symbols, die sich unmittelbar aus seiner Aufschreibung ergibt, z.B. den Zahlwert 134 der Zahlkonstanten "0134". Es gibt keine Attributregeln für diese Attribute, ihre Werte werden als bekannt vorausgesetzt. (In einem PEARL-Übersetzer werden sie durch die lexikalische Analyse der Symbole bestimmt.)

5.2 Attributtypen

Durch den Typ eines Attributs wird sein Wertebereich festgelegt. Er kann als Abstraktion der Eigenschaft verstanden werden, die das Attribut beschreibt. ALADIN hat neben einigen elementaren Typen mächtige Typkonstruktoren, die es erlauben, auch komplexe Eigenschaften (z.B. die Definitionen eines Blocks) strukturiert zu beschreiben. Mit Hilfe von Typkonstruktoren werden in Typdefinitionen neue, für die Anwendung passende Typen eingeführt (vergleichbar mit den MODE-Deklarationen in ALGOL68 oder den Typdefinitionen in PASCAL). Im folgenden erläutern wir die elementaren Typen und die Typkonstruktoren zusammen mit den wichtigsten der jeweils anwendbaren Operationen. (Wertvergleiche sind für alle Typen definiert und werden hier nicht mehr erwähnt.)

Im Normentwurf werden die elementaren Typen INT, BOOL und SYMB benutzt. Der Wertebereich des Typs SYMB umfaßt die terminalen Symbole der Sprache. Typische Anwendungen von SYMB sind Attribute von Terminalen, welche die Identität des Symbols beschreiben, z.B. sx_id.at_id für Bezeichner {13.0108} oder sx_op_token.at_op_token für Operatoren {16.0305}. Für Werte vom Typ SYMB können Fallunterscheidungen angewandt werden, z.B. zur Identifikation von Standard-Operatoren in {16.2275}.

Neben den Werten der Grundtypen werden neue elementare Werte (wie in PASCAL) durch Aufzählung ihrer Namen eingeführt, z.B. in {11.0068}:

```
TYPE tp_definition_kind:
        (sc_length_or_prec_def,
           ...
         sc_TASK_del)
```

(Werte dieses Typs klassifizieren PEARL-Definitionen, um für Basic PEARL die Reihenfolgebedingung zu prüfen.) Außer Fallunterscheidungen sind Ordnungsrelationen möglich, z.B. in {12.0369}. Die Ordnung der Werte wird durch ihre Reihenfolge in der Typdefinition festgelegt.

Aufzählungstypen können zur Mengenbildung verwendet werden, z.B. in {17.0116} mit {17.0136}:

```
TYPE tp_from_to : (sc_from, sc_to)
TYPE tp_dir_set : SETOF tp_from_to .
```

(Werte dieses Mengentyps geben an, in welchen Richtungen eine Datenstation betrieben werden kann.) Es sind die

Mengenoperationen (+, -, *, <=, >=, IN) definiert. Mengenwerte werden durch Aufzählung der Elemente mit vorangestelltem Typnamen angegeben, z.B. tp_dir_set (sc_to); tp_dir_set() gibt die leere Menge an.

Die Wertebereiche von INT und von Aufzählungstypen können auf Ausschnitte eingeschränkt werden, z.B. in {15.0228}

```
TYPE tp_posint: [c_zero : MAXINT]
```

Der Typkonstruktor STRUCT bildet (wie in ALGOL68) zusammengesetzte Wertebereiche für Verbunde, z.B. in {11.0056}.

```
TYPE tp_access_def:
     STRUCT (s_id     : SYMB,
             s_global: SYMB,
             s_value : tp_value,
             s_attr  : tp_attr),
```

mit dem PEARL-Definitionen (einer bestimmten Klasse) abstrakt beschrieben werden. Die Komponentennamen (s_id, s_global, usw.) werden in Selektionsoperationen angewandt, z.B. in {12.0338} p_access.s_id. Verbundwerte werden durch die Werte der Komponenten in der Reihenfolge, welche die Typdefinition vorschreibt, mit vorangestelltem Typnamen angegeben, z.B. in {13.1215}.

```
tp_access_def (HEAD (p_ids),
               p_global,
               HEAD (p_value),
               p_attr)
```

Der Typkonstruktor UNION vereinigt (wie in ALGOL68) mehrere Wertebereiche zu einem neuen, z.B. in {11.0103}.

```
TYPE tp_id_def:
     UNION (tp_access_def,
            tp_OPERATOR_def,
            tp_TYPE_def)
```

(Hier werden Beschreibungen verschiedener, bezeichneter PEARL-Definitionen zusammengefaßt.) Werte dieses Typs werden durch implizite Anpassung aus Werten der Untertypen gebildet. Mit Fallunterscheidungen kann festgestellt werden, ob ein Wert des vereinigten Wertebereichs zu einem der Teilbereiche gehört, z.B. in {12.0532}.

```
CASE HEAD (p_l) OF
IS tp_access_def:
    IF THIS.s_global /= c_not_global
    ...
OUT ...
ESAC
```

Der Standardname THIS ist eine Abkürzung für den untersuchten Ausdruck zwischen CASE und OF. Häufig wird stattdessen auch ein frei gewählter Name explizit eingeführt. Das obige Beispiel würde dann lauten

```
CASE ce_acc: HEAD (p_l) OF
IS tp_access_def:
   IF ce_acc.s_global /= c_not_global
    ...
OUT ...
ESAC
```

Die Prüfung der Zugehörigkeit zu einem Untertyp kann ebenso durch einen Typtest (IS-Operator) formuliert werden:

```
IF HEAD (p_l) IS tp_access_def
THEN IF HEAD (p_l) QUA tp_access_def
        .s_global /= c_not_global
            ...
ELSE ...
```

Der QUA-Operator gibt an, daß der davor stehende Ausdruck als Wert des Untertyps tp_access_def benutzt wird. In den oben angegebenen Fallunterscheidungen wird er an entsprechender Stelle impliziert.

Der Typkonstruktor LISTOF definiert als Wertebereich Sequenzen von Werten eines Grundtyps (lineare Listen), z.B. in {11.0108}.

```
TYPE tp_id_def_list : LISTOF tp_id_def
```

Diesen Typ hat z.B. das Attribut sx_block.at_id_defs, das alle gültigen benannten Definitionen beschreibt. Auf Listenwerte sind vordefinierte Funktionen, wie HEAD, TAIL, FRONT, LAST, LENGTH und EMPTY (siehe [1], S. 23) und die Konkatenation (+) anwendbar, z.B. in {12.0257}.

```
sx_block_body.at_id_defs :=
    sx_definitions_ety.at_id_defs
  + sx_statements_ety.at_label_defs
    ...
```

Das Durchlaufen der Elemente einer Liste (z.B. zur Suche eines Elements in {12.0433}) wird durch rekursive Funktionen beschrieben.

Listenwerte gibt man wie Verbundwerte durch Aufzählung ihrer Elemente mit vorangestelltem Typnamen an, z.B.

```
tp_id_def_list(a,b,c).
```

Die leere Liste wird entsprechend notiert: tp_id_def_list(). Einige Listentypdefinitionen enthalten eine KEY-Angebe, die nur im Zusammenhang mit der vordefinierten ALADIN-Funktion SELECT_BY_KEY Bedeutung hat (siehe [1], S. 22/23).

Die Typbildungsregeln für ALADIN sind im wesentlichen aus ALGOL68, PASCAL und LISP übernommen. Ein prinzipieller Unterschied ergibt sich jedoch aus dem streng applikativen Charakter von ALADIN: In höheren Programmiersprachen sind rekursiv definierte Typen nur unter Verwendung von Referenzen zulässig, z.B. ein Binärbaum in ALGOL68:

```
MODE Knoten=UNION (Blatt,Zweig);
MODE Zweig =STRUCT (REF Knoten links,
                                rechts,
                     INT opr);
MODE Blatt =STRUCT (INT wert);
```

Einerseits enthält ALADIN kein Referenzkonzept - es würde nicht zur Variablenfreiheit passen. Andererseits besteht keine Veranlassung, in einer Beschreibungssprache rekursiv definierte Typen auszuschließen, solange ihr Wertebereich endliche Werte enthält. Das obige Beispiel lautet deshalb in ALADIN:

```
TYPE Knoten: UNION (Blatt,Zweig);
TYPE Zweig : STRUCT (links, rechts: Knoten,
                     opr: INT);
TYPE Blatt : STRUCT (Wert: INT);
```

(Für die Implementierung solcher Strukturen wendet man natürlich das Referenzkonzept an.)

Ein Typ wie

```
TYPE seq: STRUCT (Wert:INT, nachf:seq)
```

ist auch in ALADIN unzulässig.

Typische Anwendungen rekursiver Typdefinitionen sind die Wertebereiche, welche die PEARL-Typen beschreiben (tp_type in {15.0274}).

5.3 Regeln

Der Kern einer AG besteht aus den Regeln, die eine kontext-freie Produktion mit den zugehörigen Attributregeln und -bedingungen zusammenfaßt.

Die Gliederung der Regeln sei am Beispiel in {16.0994} erläutert:

```
RULE sx_assignment ::=        % RULE 2
     sx_destination ':=' sx_assignment
SUBSET CONDITION c_non_Basic
STATIC
   CONDITION ...
   sx_assignment[2].at_post_type := ...
   sx_assignment[1].at_pre_type  := ...
   CONDITION ...
   sx_assignment[1].at_value := ...
DYNAMIC
   % ...
END
```

Im ersten Abschnitt steht die kontext-freie Produktion in erweiterter Backus-Naur-Form notiert. (Die Erweiterungen erläutern wir unten.) Der Kommentar % RULE 2 gibt an, daß es sich um die zweite alternative Produktion mit gleicher linker Seite (sx_assignment) handelt. Der zweite Abschnitt (SUBSET) enthält eine Bedingung, welche in diesem Fall die Anwendung der Produktion für Basic PEARL verbietet (siehe Abschnitt 4.1).

Der dritte Abschnitt (STATIC) enthält Attributbedingungen (eingeleitet durch CONDITION) und Attributregeln, welche den Wert des links vom Zuweisungszeichen angegebenen Attributs bestimmen. Die rechte Seite einer Attributregel ist ein Ausdruck, der von anderen Attributen abhängen kann. Alle in der Regel vorkommenden Attribute beziehen sich auf Symbole der kontext-freien Produktion (Ausnahme siehe unten). Ein Symbol, das mehrfach in der Produktion vorkommt, wird in Attributbenennungen indiziert angegeben. Die Menge der Attributregeln ist im folgenden Sinne immer vollständig und eindeutig: Es gibt

genau eine Attributregel zu jedem abgeleiteten Attribut des Symbols auf der linken Seite der Produktion (sx_assignment[1].at_pre_type und sx_assignment[1].at_value) und zu jedem erworbenen Attribut jedes Symbols auf der rechten Seite der Produktion (sx_assignment[2].at_post_type).

Der DYNAMIC-Abschnitt enthält Kommentare mit Angaben zur dynamischen Semantik (meist "Laufzeitbedingungen" oder Verweise auf Abbildungen mit Netzen).

Außer der kontext-freien Produktion kann jeder der Abschnitte fehlen.

Im folgenden erläutern wir zunächst die Erweiterungen der BNF zusammen mit den Konsequenzen für die Attributregeln und geben dann die Bedeutung einiger abkürzender Schreibweisen in Attributregeln an.

5.3.1 Schreibweise der Produktionen - Auswirkungen auf Attributregeln

Symbolfolgen, die in der Anwendung einer Produktion optional sind, werden durch [,] geklammert, z.B. in {14.0449}.

```
RULE sx_cond_statement ::=
      'IF' sx_BIT1_exp
      'THEN' sx_statements_ety
     ['ELSE' sx_statements_ety]
      'FIN'
      ...
```

Wird in Ausdrücken auf ein Attribut eines solchen Symbols Bezug genommen, so muß sichergestellt sein, daß der Wert des Ausdrucks auch dann definiert ist, wenn das Symbol fehlt. Dies geschieht durch einen bedingten Ausdruck, in dem das optionale Symbol anstelle der Bedingung steht, z.B. in {14.0463}:

```
 IF sx_statements_ety[2]
 THEN at_label_defs
 ELSE tp_id_def_list()
 FI
```

Im THEN-Teil wird der Attributname ohne das optionale Symbol angegeben. Der ELSE-Teil gibt den Wert des bedingten Ausdrucks an für den Fall, daß das Symbol fehlt.

Logische Ausdrücke der Form Symbolname IS THERE geben an, ob in einer Anwendung der Produktion das optionale Symbol tatsächlich vorhanden ist. In {15.0873}

```
 RULE sx_REF_type ::=
      'REF' [sx_INV] sx_type
              ...
 STATIC
      sx_REF_type.at_type ::=
         tp_attr (sx_INV IS THERE,
                  sx_type.at_type)
 END
```

entscheidet das Vorhandensein des INV-Symbols über die Variabilitätseigenschaft des Referenztyps.

Die übrigen Erweiterungen der BNF beschreiben verschiedene Formen der syntaktischen Wiederholung: Der * in {14.0093}

```
RULE sx_statement ::=
     sx_label_dcl* sx_unlabelled_statement
```

bedeutet, daß eine Anweisung mit beliebig vielen (auch keiner) Markendefinition beginnen kann. Ein + anstelle des * würde mindestens eine Markendefinition verlangen. Soll eine Sequenz von mehreren Symbolen wiederholt werden, so sind diese in (...)+ bzw. (...)* eingeschlossen, z.B. in {13.0678}

```
RULE sx_PROC_dcl ::=
     (sx_id ':')+ 'PROCEDURE' ...
```

Eine korrekte Ableitung wäre z.B.

P1 : P2 : P3 : PROCEDURE ...

In {13.1038}

```
 RULE sx_ids ::=
      '(' (sx_id // ',' ) ')'
```

wird eine nicht leere Sequenz von Bezeichnern beschrieben, die durch ',' getrennt sind. Korrekte Ableitungen wären z.B. (A) oder (A,B) oder (A,B,C) ...

Nimmt man auf Attribute von Symbolen Bezug, die in einer der beschriebenen Weisen als "wiederholbar" gekennzeichnet sind, so meint man nicht den Wert eines einzelnen Attributs, sondern die Liste der

Attributwerte aller Symbole, die aus der Wiederholung abgeleitet wurden. Wird z.B. sx_ids nach obiger Produktion zu (A,B,C) abgeleitet, so hat der Ausdruck sx_id.at_id in {13.1042} den Wert

tp_symb_list('A', 'B', 'C').

5.3.2 Abkürzende Schreibweisen in Attributregeln

Attributregeln, die einen "Wert-Transport" mit gleich benannten Attributen beschreiben, werden durch TRANSFER abgekürzt. Z.B. in der "Kettenproduktion" {16.0927}

```
RULE sx_prim_exp ::= sx_basic_exp
STATIC TRANSFER
END
```

steht TRANSFER für die drei Attributregeln

```
sx_prim_exp.at_pre_type :=
   sx_basic_exp.at_pre_type;
sx_prim_ex.at_value :=
   sx_basic_exp.at_value;
sx_basic_exp.at_post_type :=
   sx_prim_exp.at_post_type;
```

Jeweils eines der beiden Attribute gehört zum Symbol auf der linken Seite der Produktion. Die "Transportrichtung" wird durch die Attributklasse bestimmt: at_pre_type und at_value sind abgeleitet - "Transport nach oben im Baum", at_post_type ist erworben - "Transport nach unten im Baum".

In der Form TRANSFER at_value, z.B. in {16.0947}, wird die Abkürzung auch für einzelne Attributregeln neben ausgeschriebenen Attributregeln benutzt.

Um einen "Attributwert-Transport" über viele syntaktische Zwischenstufen zu beschreiben, wird mit dem INCLUDING-Konstrukt auf ein Attribut eines Symbols Bezug genommen, das nicht in der Produktion vorkommt. Es ist ein Symbol, aus dem die linke Seite der Produktion (direkt oder indirekt) abgeleitet ist, z.B. in {13.0538} zur Regel in {13.0527}:

```
RULE sx_GLOBAL_attr ::=
 'GLOBAL' ['(' sx_GLOBAL_qual ')']
          ...
STATIC
 sx_GLOBAL_attr.at_GLOBAL_qual :=
  IF sx_GLOBAL_qual
  THEN at_GLOBAL_qual
  ELSE INCLUDING sx_MODULE.at_GLOBAL_qual
  FI
END
```

Im folgenden Beispiel werden zwei Alternativen für die Herkunft des Attributwerts angegeben {13.1123}

```
RULE sx_FIXED_precision_def ::=
     'LENGTH' 'FIXED'
     '('sx_FIXED_const_denot')'
SUBSET
  CONDITION c_non_basic OR
  ((INCLUDING
      (sx_PROBLEM_division.at_context,
       sx_block_body.at_context))
    = sc_PROBLEM_division)
   ...
```

Es wird dann das im Strukturbaum auf dem Wege nach oben nächste der beiden Symbole gewählt. Ist das z.B. ein sx_block_body, dessen Attribut at_context immer den Wert sc_block_body hat, so ist diese SUBSET-Bedingung verletzt (d.h. LENGTH-Definitionen dürfen in Basic PEARL nur auf der Ebene der PROBLEM-Division stehen).

Auf Attribute aus Teilbäumen, die aus der rechten Seite der Produktion abgeleitet sind, wird mit dem CONSTITUENT-Konstrukt Bezug genommen. Da es in der PEARL-AG nur an wenigen Stellen vorkommt, sei hier auf die Beschreibung in [1], S.21 verwiesen.

In den Beispielen haben wir bisher nur solche Attributbedingungen vorgestellt, die innerhalb von Regeln gleichrangig neben Attributregeln stehen. In einigen Attributregeln und Funktionsrümpfen stehen Ausdrücke der Form (a CONDITION b), welche den Wert a haben und gleichzeitig die Kontextbedingung b spezifizieren, z.B. in {15.2944}.

```
FUNCTION f_deproceduring
  (p_PROC_type : tp_PROC_type) tp_type :
   f_expand_type
      ((p_PROC_type.s_RESULT
        CONDITION
        EMPTY(p_PROC_type.s_params)))
```

Die Anpassungsoperation "Deprozedurieren" liefert den Ergebnistyp der Prozedur unter der Voraussetzung, daß keine formalen Parameter existieren. Im ELSE-Teil der Attributregel in {12.0266}

```
sx_block_body.at_FIXED_precision :=
  IF ...
  THEN ...
  ELSE HEAD
     ((f_select_FIXED_precision
       (sx_definitions_ety.at_pre_defs)
        CONDITION LENGTH (IT) = 1))
  FI
```

wird die FIXED-Precision eines Blocks gemäß der ersten FIXED-Precision-Definition im Definitionsteil bestimmt und gleichzeitig gefordert, daß es keine weitere gibt. IT ist eine Abkürzung des Ausdrucks, der vor CONDITION steht. In einigen Fallunterscheidungen, z.B. in {13.0670} findet man als Wert eines Falls Konstrukte der Form (a CONDITION FALSE). Sie drücken aus, daß in PEARL-Programmen dieser Fall unzulässig ist.

Literatur

[1] DIN 66253 Teil 2, Informationsverarbeitung, Programmiersprache PEARL, Full PEARL, Beuth Verlag Berlin 1980

[2] DIN 66253 Teil 1, Informationsverarbeitung, Programmiersprache PEARL, Basic PEARL, Beuth Verlag Berlin 1978

[3] Full PEARL Language Description, Gesellschaft für Kernforschung mbH, Karlsruhe, PDV-Bericht KfK-PDV 130, 1977

[4] S. Heilbrunner, M. Schmitz: Zur attributierten Grammatik von PEARL (in dieser PEARL-Rundschau)

[5] U. Kastens: ALADIN - Eine Definitionssprache für attributierte Grammatiken. Fak. f. Informatik, Universität Karlsruhe, Bericht 7/79

[6] U. Kastens: Ordered Attributed Grammars. Acta Informatica 13, 1980, 229-256

[7] U. Kastens, E. Zimmermann: GAG - A Generator Based on Attributed Grammars. Fak. f. Informatik, Universität Karlsruhe, Bericht 16/81

[8] D.E. Knuth: Semantics of context-free languages. Math. Syst. Th. 2 (1968), 127-145. Korrekturen in: Math. Syst. Th. 5 (1971), 95 ff

Transformationstechniken bei Condition/Event-Netzen in der Sprachdefinition von Full-PEARL

Dipl.-Inform. Bernhard Karbe, München

0. Einleitung

Zur Erläuterung der Semantik der Programmiersprache PEARL, wie sie durch das Deutsche Institut für Normung in der DIN 66253 Teil 2 (Full PEARL) festgelegt ist, werden Petri-Netze in der Interpretation von Condition/Event-Netzen herangezogen. Dieser Beitrag ist als Lesehilfe gedacht, und soll die dabei verwendeten Petri-Netz-Transformationen erläutern.

Es wird vorausgesetzt, daß der Leser mit Petri-Netzen vertraut ist. Eine für Ingenieure geeignete Einführung in Petri-Netze stellt die Arbeit [5] dar.

1. Definitionen und Regeln

1.1. Stellen und Transitionen

Ein Petri-Netz besteht aus S-Knoten, T-Knoten und Pfeilen (Kanten). Jeder S-Knoten, im folgenden auch Stelle oder Platz genannt, wird als Kreis, jeder T-Knoten, auch Transition genannt, als Kästchen dargestellt. Jede Kante kann nur von einer Stelle (Kreis) zu einer Transition (Kästchen) oder von einer Transition zu einer Stelle gehen. Führt ein Pfeil von einem Knoten A zu einem Knoten B und ein anderer von B nach A, so wird als Abkürzung ein Pfeil mit zwei Köpfen (jeder an einem Ende), hier Doppelpfeil genannt, benutzt.

In Transitionsnetzen können Stellen eine variable Anzahl von sog. Markierungen (Tokens) besitzen. Die Transitionen beschreiben die möglichen elementaren Veränderungen der Verteilung der Markierungen in dem Netz. In einem Condition/Event-Netz kann jede Stelle höchstens eine Markierung besitzen. Markierungen werden graphisch durch Punkte in den Stellen repräsentiert. Jedes Netz besitzt eine initiale Markierung.

1.2. Fortschaltungsregeln

Im folgenden werden nur noch elementare Condition/Event-Netze und deren in dem DIN-Entwurf verwendete Abstraktionen betrachtet.

Führt ein Pfeil von einer Stelle zu einer Transition t, so wird die Stelle Eingabeplatz der Transition t genannt. Führt ein Pfeil von einer Transition t zu einer Stelle, heißt die Stelle Ausgabeplatz der Transition t.

Die Verteilung der Markierungen in einem Transitionsnetz ändert sich genau dann, wenn (mindestens) eine Transition "zündet", "schaltet" oder "feuert". Eine Transition t kann nur dann zünden, wenn alle Eingabeplätze von t markiert und alle Ausgabeplätze unmarkiert sind. Zünden bedeutet: Von jedem Eingabeplatz der Transition wird eine Markierung entfernt und in jedem Ausgabeplatz der Transition wird eine Markierung hinzugefügt. Transitionen zünden zufällig. Eine Aussage darüber, innerhalb welcher Zeit nach Eintreten der Voraussetzungen zum Zünden die Transition tatsächlich schaltet, kann nicht gemacht werden. Zwei Transitionen können genau dann gleichzeitig feuern, wenn sie unabhängig voneinander sind, d.h. keine gemeinsamen Eingabe- oder Ausgabeplätze besitzen.

Man kann also folgende zwei Situationen unterscheiden:
Eine Transition hat (prinzipiell) die Möglichkeit zu zünden, wenn alle Eingabeplätze von ihr markiert und alle Ausgabeplätze von ihr unmarkiert sind. Es sei darauf hingewiesen, daß sie zünden kann, aber nicht muß (siehe Bild 1 und Bild 2).
Zwei Transitionen befinden sich im Konflikt, wenn sie mindestens einen Eingabe- oder einen Ausgabeplatz gemeinsam haben und beide die Möglichkeit haben zu zünden. (Bild 3 und Bild 4). Zündet nämlich eine von beiden, so kann die andere nicht mehr zünden. Wie der Konflikt gelöst wird, wird nicht mehr durch das Transitionsnetz selbst beschrieben.

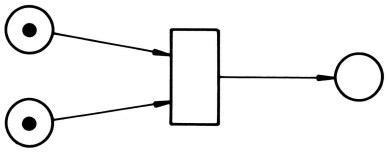


Bild 1: Transition kann zünden

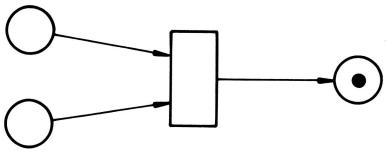


Bild 2: Transition hat gezündet

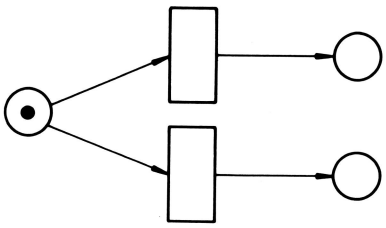


Bild 3: Transitionen haben Eingabeplatz gemeinsam

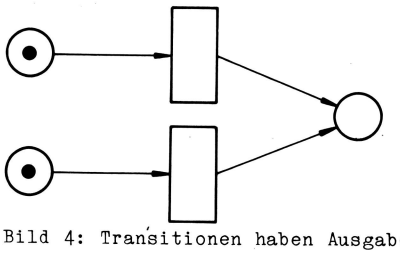
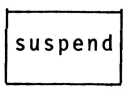


Bild 4: Transitionen haben Ausgabeplatz gemeinsam

Es gibt zahlreiche Beispiele für Konfliktsituationen in technischen Systemen und aus dem täglichen Leben. Man denke z.B. daran, daß der Rufknopf für einen einzelnen Aufzug in verschiedenen Stockwerken gleichzeitig gedrückt wird, oder daran, daß derselbe Teilnehmer in einem Telefonnetz von verschiedenen anderen Teilnehmern aus gleichzeitig angewählt wird. In diesen Fällen ist klar, daß nur genau eine der möglichen Folgesituationen eintreten kann. Häufig ist jedoch nicht vorhersagbar, welche dies ist. Im Telefonnetz hängt dies z.B. davon ab, welcher Ruf früher bei der Teilnehmerschaltung des angerufenen Teilnehmers ankommt, und dies ist letzlich zufällig. Eine korrekte Darstellung einer solchen Situation kann folglich nur darin bestehen, die Konfliktlösung offen zu lassen. Schließlich kann von außen nicht entschieden werden, welche Folgesituation eintritt. Anders ausgedrückt: Überall dort, wo in der Beschreibung der Sprache PEARL Konflikte in den Transitionsnetzen auftreten, ist es dem Implementierer der Sprache anheimgestellt, wie er den Konflikt löst. Er ist frei in der Art der Lösung. Andererseits darf ein Anwender von PEARL seine Programmsysteme nicht so entwerfen, daß sie nur dann bestimmungsgemäß funktionieren, wenn eine ganz bestimmte Art der Konfliktlösung unterstellt wird. Es wird hierdurch nicht nur die Übertragbarkeit der Programme gefährdet, sondern es ist auch durchaus denkbar, daß der Sprachimplementierer bei einer Revision des Übersetzers eine andere Konfliktlösung wählt.

2. Notationale Konventionen

Im beschreibenden Text wird auf Stellen mit den runden Klammern (...) und auf Transitionen mit den eckigen Klammern [...] Bezug genommen. In den Klammern steht der Knotenname, also z.B. [suspend] für



Die Knoten (P) und [T] sind elementare Knoten des Condition/Event-Netzes und lassen sich formal als Stellen der Kapazität 1 und Transitionen eines Condition/Event-Netzes interpretieren. Alle anderen Knoten sind Abstraktionen. Sie sind Platzhalter für einsetzbare Teilnetze (s. Abschnitt 3) oder Konnektoren (s. Abschnitt 4).

Prinzipiell lassen sich Netze mit solchen nicht elementaren Knoten auf Netze mit ausschließlich elementaren Knoten mittels der unten zu beschreibenden Transformationstechniken abbilden. Es sei jedoch davor gewarnt anzunehmen, man könne aus einem vorgegebenen Full-PEARL-Programm, der im DIN-Entwurf angegebenen Grammatik und insbesondere deren DYNAMICS-Klauseln auf mechanische Art ein rein elementares Condition/Event-Netz herleiten, das vollständig die Semantik des dazugehörenden Programms beschreibt. Eine Beschreibung sequentieller Abläufe mittels Petri-Netzen würde nämlich bewirken, daß in dem DIN-Entwurf wesentlich mehr Details vorkommen würden, als dies zur Spezifikation der Sprache PEARL notwendig und zweckmäßig wäre. Für den Sprach-Implementierer würden Dinge festgelegt, die nicht mehr Eigenschaften der Sprache definieren, sondern eine bestimmte Implementierung vorschreiben. Deshalb ist eine Spezifikation solcher Abläufe im DIN-Entwurf unterblieben [4]. Vielmehr werden durch die angegebenen Netze die nichtsequentiellen PEARL-Eigenschaften modelliert und damit auch die nichtsequentiellen Eigenschaften eines PEARL-Programms. Um jedoch den Zusammenhang der verschiedenen Eigenschaften bzw. den sie modellierenden Netzen beschreiben zu können, sind die unten erklärten Transformationsregeln ausgesprochen nützlich.

Die Inschrift der Knoten gibt deren Bedeutung bezüglich PEARL wider, u.U. gefolgt von einem Kommentar, dem % vorangestellt ist. Für die formale Interpretation als Petri-Netz sind die Inschriften bis auf die der Knoten (P), (P INITIAL) und [T] bedeutungslos. Verschiedene Knoten in einem Netz können durchaus dieselbe Inschrift besitzen. Knoten, die lediglich einen Kommentar enthalten, aber keinen Knotennamen, sind Platzhalter für einzusetzende Teilnetze, die (in dem DIN-Entwurf) nicht weiter ausgeführt werden und im wesentlichen Implementierungsabhängigkeiten beschreiben. Knoten, die Knotennamen enthalten, kennzeichnen Abstraktionen, die im DIN-Entwurf verfeinert werden. Entweder beschreiben sie Konnektoren, oder es gibt für sie einsetzbare Teilnetze. I.a. besitzen sie (im DIN-Entwurf) einen Kommentar, der eine Referenz auf eine Figur enthält, in der die Verfeinerung beschrieben ist.

Transitionsnetze besitzen eine initiale Markierung. Diese wird durch die Inschrift "INITIAL" in gewissen Knoten (P) und Vergröberungen, die (P) enthalten, beschrieben.

In den Beispielen werden oft Netze aus dem o.g. DIN-Entwurf verwendet. Dabei werden nicht in allen Fällen die dortigen Inschriften vollständig übernommen. Sie müssen aus Platzgründen abgekürzt werden. I.a. entfällt der Kommentar.

3. Einsetzung von Teilnetzen

Üblicherweise wird die Syntax einer Programmiersprache durch ein Produktionssystem beschrieben, durch das alle syntaktisch korrekten Programme hergeleitet werden können. Damit ist aber über die Semantik eines Programmes noch nichts ausgesagt. Eine entsprechende Aussage kann erreicht werden, wenn man mit der syntaktischen Ableitung die Herleitung eines Condition/Event-Netzes kombiniert.

3.1. Substitution von Knoten durch Teilnetze

Die Anwendung von syntaktischen Ableitungsregeln bedeutet eine Ersetzung einer nichtterminalen Zeichenreihe Z1 durch eine andere Zeichenreihe, die terminale und/oder nichtterminale Symbole enthält. Entsprechend bedeutet die Substitutionsregel für Condition/Event-Netze die Ersetzung eines Knotens K im Netz durch das zu K gehörende Teilnetz T_k. Gegebenenfalls, jedoch nicht in allen Fällen, kann die syntaktische Ableitung und die Transformation des Transitionsnetzes simultan bzw. kombiniert ablaufen, indem durch eine syntaktische Regel ein Teil-Transitionsnetz festgelegt ist.

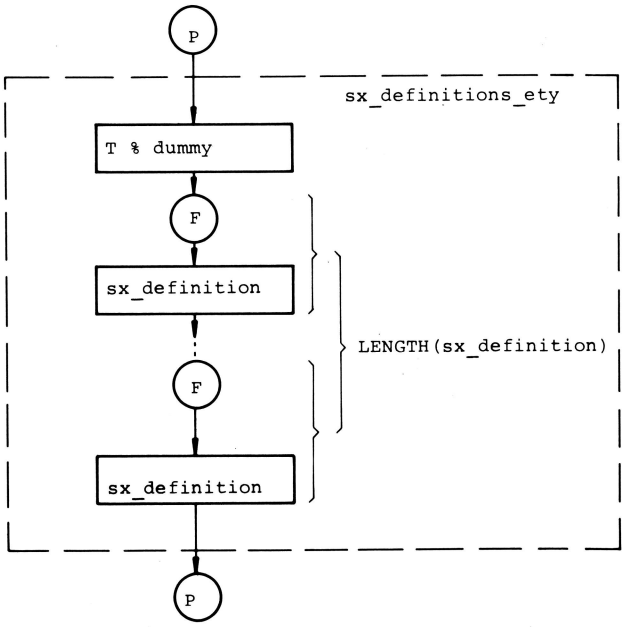


Bild 5: Petri-Netz zur Grammatik-Regel "sx_definitions_ety ::= (sx_definition ';')*" (Beispiel 3.1-0)

Beispiel 3.1-0

Betrachte die PEARL-Syntax-Regel in Abschnitt 13.2 "RULE sx_definitions_ety::= (sx_definition';')*" und das dazugehörige Petri-Netz in Figur #13 im DIN-Entwurf (Bild 5).

Wird bei einer syntaktischen, Ableitung diese Regel angewendet, so wird in dem (ja hoffentlich auch vorliegenden) Netz der Knoten [sx_definitions_ety] durch das Teilnetz in Bild 5 ersetzt. In dem Teil netz treten so viele Transitionen [sx_de finition] auf, wie bei der Ersetzung des Nonterminals "sx__definitions__ety" der Term "sx_definition ';'" auftritt. Dies wiederum hängt von dem vorliegenden Programm ab. Das terminale Symbol Semikolon ";" wird hier im Petri-Netz durch (F) modelliert, das jedoch vor der Transition [sx__definition] steht, im Gegensatz zur Syntax, wo das Semikolon dem Nonterminal "sx_definition" folgt.

Einsetzbare Teilnetze werden graphisch derart dargestellt, daß der zu verfeinernde Knoten als gestrichelte Kontur (Kreis oder Rechteck) gezeichnet wird, die mit seinem Namen versehen ist. Darin befindet sich das (normal dargestellte) Teilnetz.

In dem Ausgangsnetz N, in dem der Knoten K substituiert werden soll, ist K durch Kanten a_i mit dem Restnetz von N verknüpft. Die a_i müssen nun durch die Kanten a'_j, mit denen das Teilnetz T_k mit dem Restnetz von N verknüpft werden soll, ersetzt werden. Die a'_j sind gerade die Kanten in der Darstellung von T_k, die die gestrichelte Kontur von K kreuzen.

Beispiel 3.1-1 [3]:

Sei das Netz in Bild 6 gegeben.

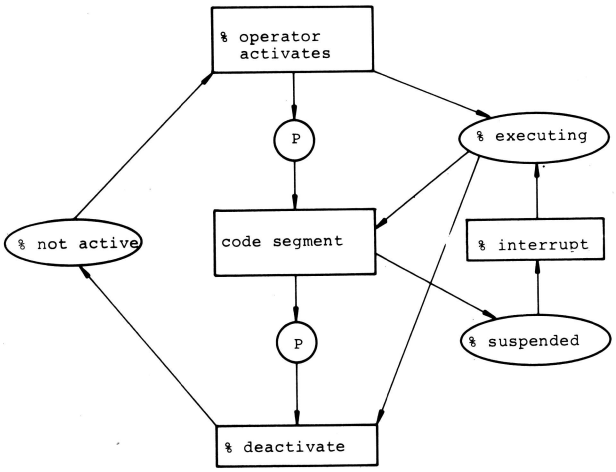


Bild 6: Ausgangsnetz, in dem der Knoten [code segment] ersetzt werden soll (Beispiel 3.1-1)

Es gebe in der Grammatik folgende Produktion:

<code segment> ::= <resume statement>
<code segment>|
<resume statement>

Aufgrund der syntaktischen Struktur eines vorliegenden Programmes soll die erste Alternative der Produktionsregel gewählt werden. Für diese gibt es das (einzusetzende) Teilnetz T_k aus Bild 7.

Dann hat das Gesamtnetz nach der Ersetzung des Knotens code segment durch das Teilnetz T_k die Struktur, wie sie in Bild 8 angegeben ist.

Bevor der Ersetzungsmechanismus weiter erläutert wird, sollen noch kurz zwei Begriffe eingeführt werden:
Jeder Knoten K', der mit dem Knoten K in einem vorgegebenen Netz durch eine Kante

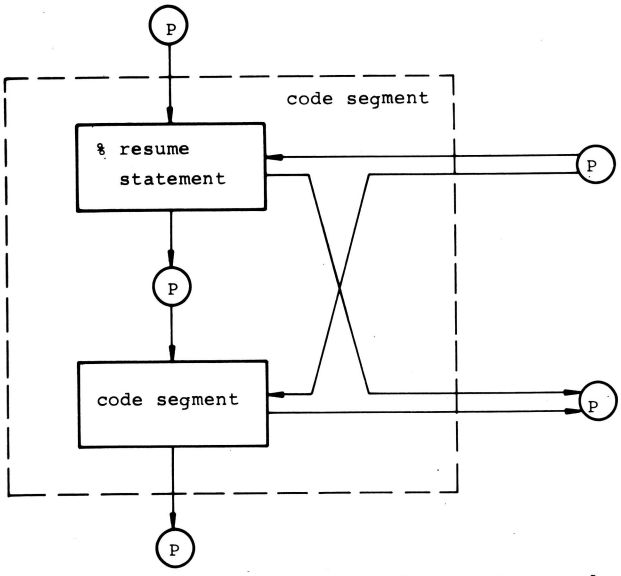


Bild 7: Einsetzbares Teilnetz für den Knoten [code segment] (Beispiel 3.1-1)

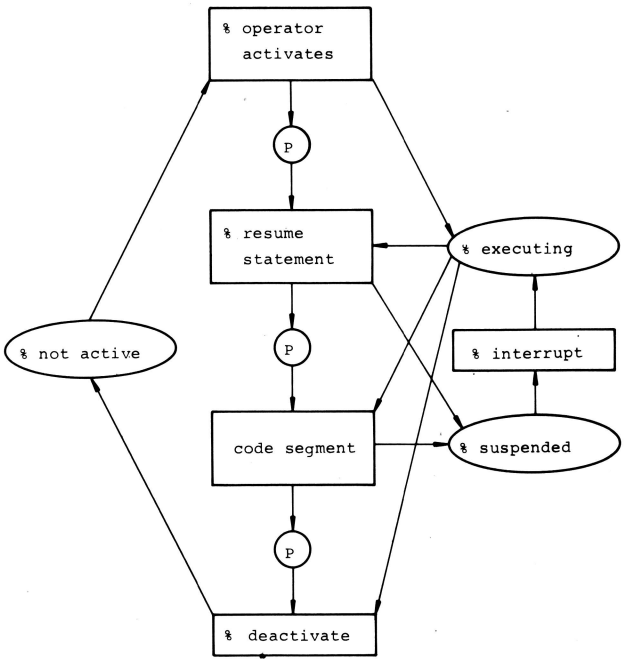


Bild 8: Ergebnisnetz nach einer Ersetzung des Knotens [code segment] durch sein Teilnetz (Beispiel 3.1-1)

verbunden ist, heißt <u>Nachbarknoten</u> von K. Die Nachbarknoten eines Teilnetzes T_k, durch das der Knoten K verfeinert wird, sind die Knoten, die außerhalb der K darstellenden gestrichelten Kontur liegen.
Jeder Knoten in dem Teilnetz T_k, der durch eine Kante mit mindestens einem der Nachbarknoten von T_k verbunden ist, heißt <u>Randknoten</u> von T_k.
Ein Teilnetz T_k, das eine Stelle (bzw. eine Transition) K verfeinert, muß also Stellen (bzw. Transitionen) als Randknoten und Transitionen (bzw. Stellen) als Nachbarknoten besitzen.

Nimmt man das Ausgangsnetz N, das den Knoten K enthält, und die Beschreibung des K verfeinernden Teilnetzes T_k her, so müssen weder die Anzahl der Nachbarknoten von K und von T_k, noch die Anzahl der Kanten übereinstimmen, über die K bzw. T_k mit einzelnen Nachbarknoten verbunden ist. Eine einzelne Kante a_i in N kann durchaus durch mehrere Kanten a'_j substituiert werden. Ein Nachbarknoten von T_k kann mit mehreren von K assoziiert werden. Gegebenenfalls muß ersterer genügend oft incl. der ihn mit T_k verbindenden Kanten dupliziert werden, bis eine 1:1 Abbildung möglich wird. Dann können die Nachbarknoten von K und T_k identifiziert werden. Umgekehrt gilt nicht, daß mehrere Kanten a_i zu einer Kante a'_j zusammenfallen können bzw. mehrere Nachbarknoten von T_k zu einem von K.

Es können demnach die Nachbarknoten von K mit denen von T_k und die Kanten a_i mit den a'_j nicht einfach mechanisch identifiziert werden. Diese Identifizierung soll in den folgenden fünf Punkten an Hand von Beispielen beschrieben werden:

1) Man betrachte die Darstellung des Teilnetzes T_k, in der K durch die gestrichelte äußerste Kontur gekennzeichnet ist. Wenn nun alle Kanten, die zu K führen, zu einem einzigen Randknoten K' in T_k führen und alle Kanten, die von K wegführen, von einem (gegebenenfalls mit K' identischen) Knoten K" von T_k wegführen, werden die Kanten gemäß ihrer Richtung identifiziert. Gemäß der Identifizierung von den Kanten werden die Nachbarknoten von K und T_k identifiziert. Die Namen der Nachbarknoten bei der Darstellung von T_k sind belanglos und können weggelassen werden.

Beispiel 3.1-2:

Betrachte die Figuren #25 und #02 aus dem DIN-Entwurf. Figur #02 aus Bild 10 soll in die Figur #25 aus Bild 9 eingesetzt werden. Es wird der Knoten (F) ersetzt.

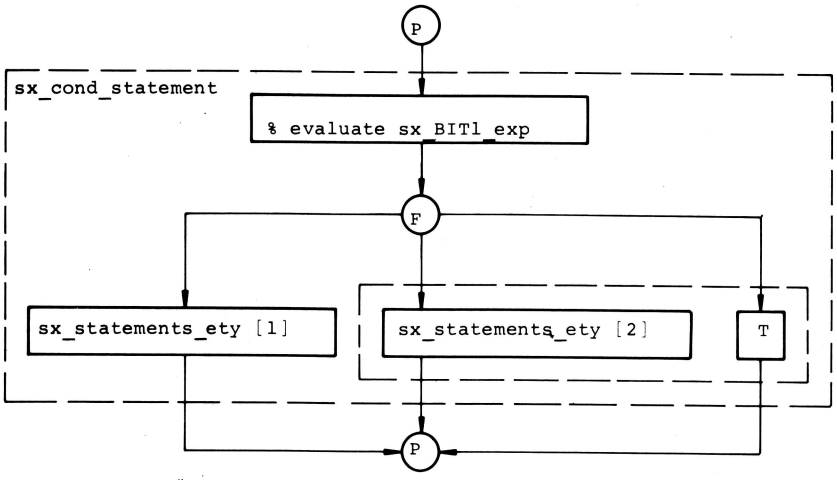


Bild 9: Figur #25 aus dem DIN-Entwurf

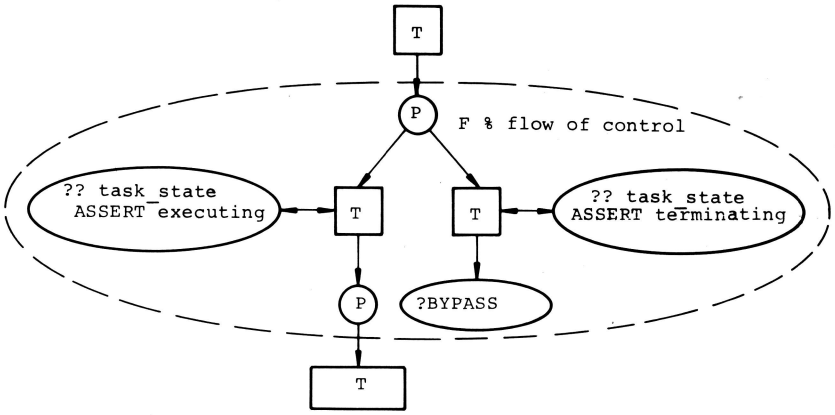


Bild 10: Figur #02 aus dem DIN-Entwurf

Die Einsetzung von #02 in #25 vollzieht sich in mehreren Schritten. Als erstes wird bei der Betrachtung von Bild 10 festgestellt, daß der hier betrachtete Fall vorliegt. In Bild 9 wird festgestellt, wieviele Kanten in den Knoten (F) hinein (hier eine) und wieviele herausführen (hier drei). Entsprechend wird in Bild 10 der untere Nachbarknoten [T] verdreifacht. Es gibt also drei Knoten $[T]_1$, $[T]_2$ und $[T]_3$. Zu jedem dieser Knoten führt eine Kante vom unteren Randknoten (P) in Bild 10. Die Identifizierung dieser Knoten mit den Knoten [sx_statements_ety [1]], [sx_statements_ety [2]] und [T] in Bild 9 ist beliebig. Der obere Nachbarknoten [T] in Bild 10 wird mit dem Knoten [% evaluate sx_BIT1_exp] in Bild 9 identifiziert. Da die Identifizierung der Nachbarknoten nun klar ist, kann man den Knoten (F) in Bild 9 streichen (incl. der zu ihm führenden Kanten) und durch das Teilnetz in Bild 10 ersetzen. Das erzeugte Netz ist in Bild 11 dargestellt.

2) Ein ebenso einfacher Fall liegt vor, wenn alle Kanten, die zu T_k führen, von einem einzigen Nachbarknoten kommen und alle Kanten, die von T_k wegführen, zu einem einzigen Nachbarknoten hinführen.

Beispiel 3.1-3

Betrachte Figur #06 in Bild 13 und das linke obere Teilnetz in Figur #04 aus dem DIN-Entwurf in Bild 12. Der Knoten [ASSIGN true TO varname] in Bild 12 soll ersetzt werden.
Die Einfügung von Bild 13 in Bild 12 geschieht hier ohne die Übernahme der Konnektoren (Dies erfolgt in Beispiel 4-2). Diese Einsetzung benötigt etwas Fingerspitzengefühl. Klar ist, daß der untere Nachbarknoten (P) in Bild 13

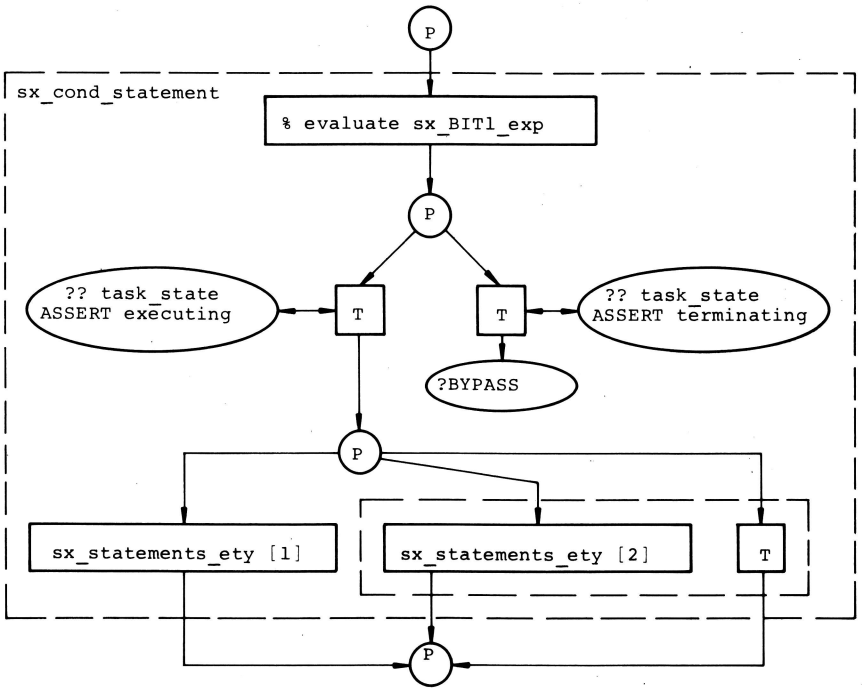


Bild 11: Figur #25 nach der Ersetzung des Knotens (F). durch Figur #02 aus dem DIN-Entwurf

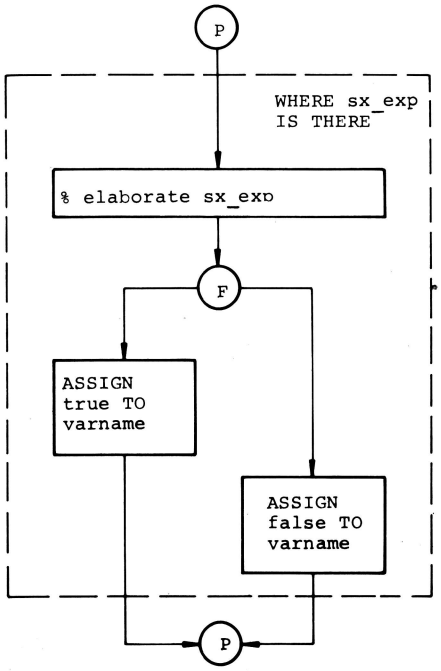


Bild 12: Linkes oberes Teilnetz aus Figur #04 aus dem DIN-Entwurf

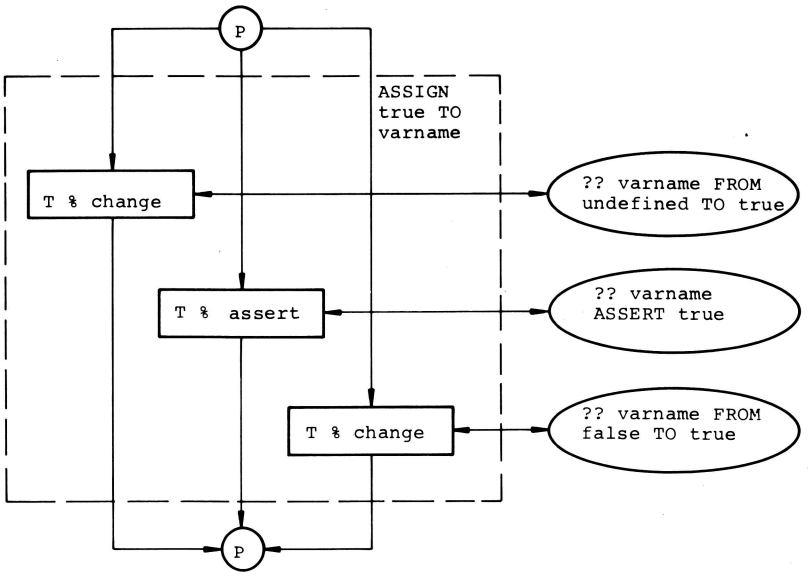


Bild 13: Figur #06 aus dem DIN-Entwurf

mit dem unteren Nachbarknoten (P) in Bild 12 identifiziert werden muß. Man erinnert sich nun, daß der Knoten (F) in Bild 12 als unteren Randknoten einen Knoten (P), siehe Bild 10, besitzt. Infolgedessen ist der obere Nachbarknoten (P) in Bild 13 mit dem Knoten (F) in Bild 12 zu identifizieren. Das Ergebnis der Einsetzung kann nun auf einfache Art erzeugt werden und ist in Bild 14 dargestellt.

3) Die Verteilung der Kanten kann auch durch die Unterscheidung von einfachen und Doppelpfeilen für T_k abgeleitet werden. Insbesondere können Doppelpfeile zu K durch mehrere Einfachpfeile bei T_k aufgelöst werden.

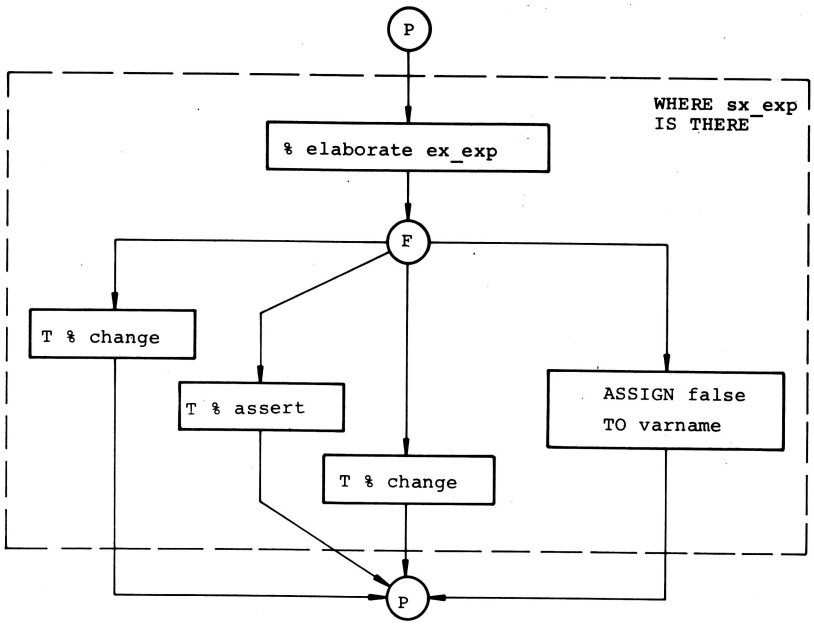


Bild 14: Einsetzung von Figur #06 in Figur #04, oberes linkes Teilnetz, ohne Übernahme der Konnektoren

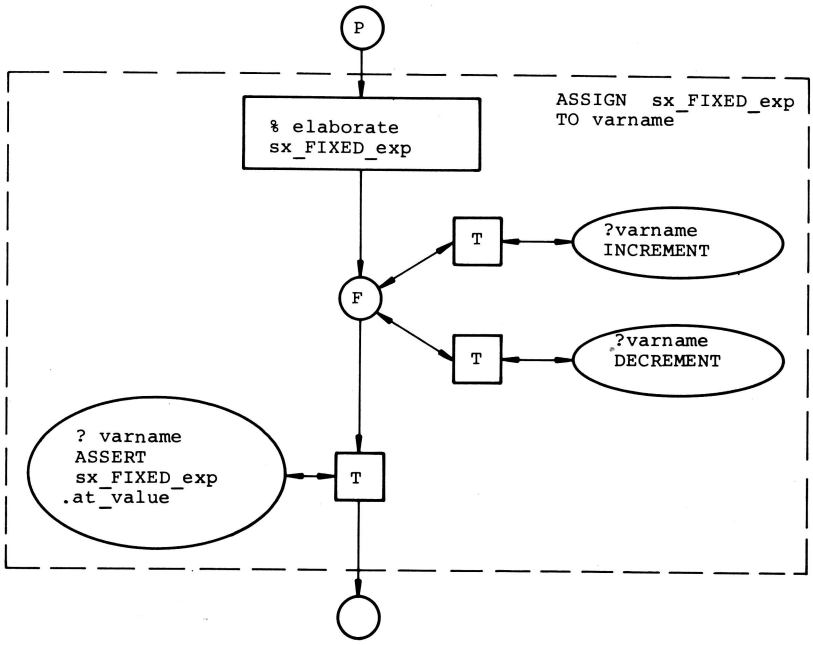


Bild 15: Figur #08, linkes Teilnetz, aus dem DIN-Entwurf

Beispiel 3.1-4

Ersetzung von Doppelpfeilen im Ausgangsnetz durch Einfachpfeile: Betrachte die Figuren #02 und #08, linkes Teilnetz, aus dem DIN-Entwurf in den Bildern 10 und 15. Geht man von Bild 15 aus, so müssen als erstes die beiden Doppelpfeile, die nach (F) führen, durch je zwei Pfeile entgegengesetzter Richtung ersetzt werden. Alle Pfeile, die zum Knoten (F) führen, müssen gemäß der in Beispiel 3.1-2 beschriebenen Regel zum oberen (P)-Randknoten des zu ersetzenden Knoten (F) in Bild 10 hinführen, und all die Kanten, die von (F) wegführen, müssen vom unteren (P)-Randknoten wegführen. Der Rest der Ersetzung ist recht einfach und braucht daher nicht noch einmal erklärt zu werden. Das Ergebnis der Ersetzung findet man in Bild 16.

4) Sollten diese Regeln noch nicht zu einer

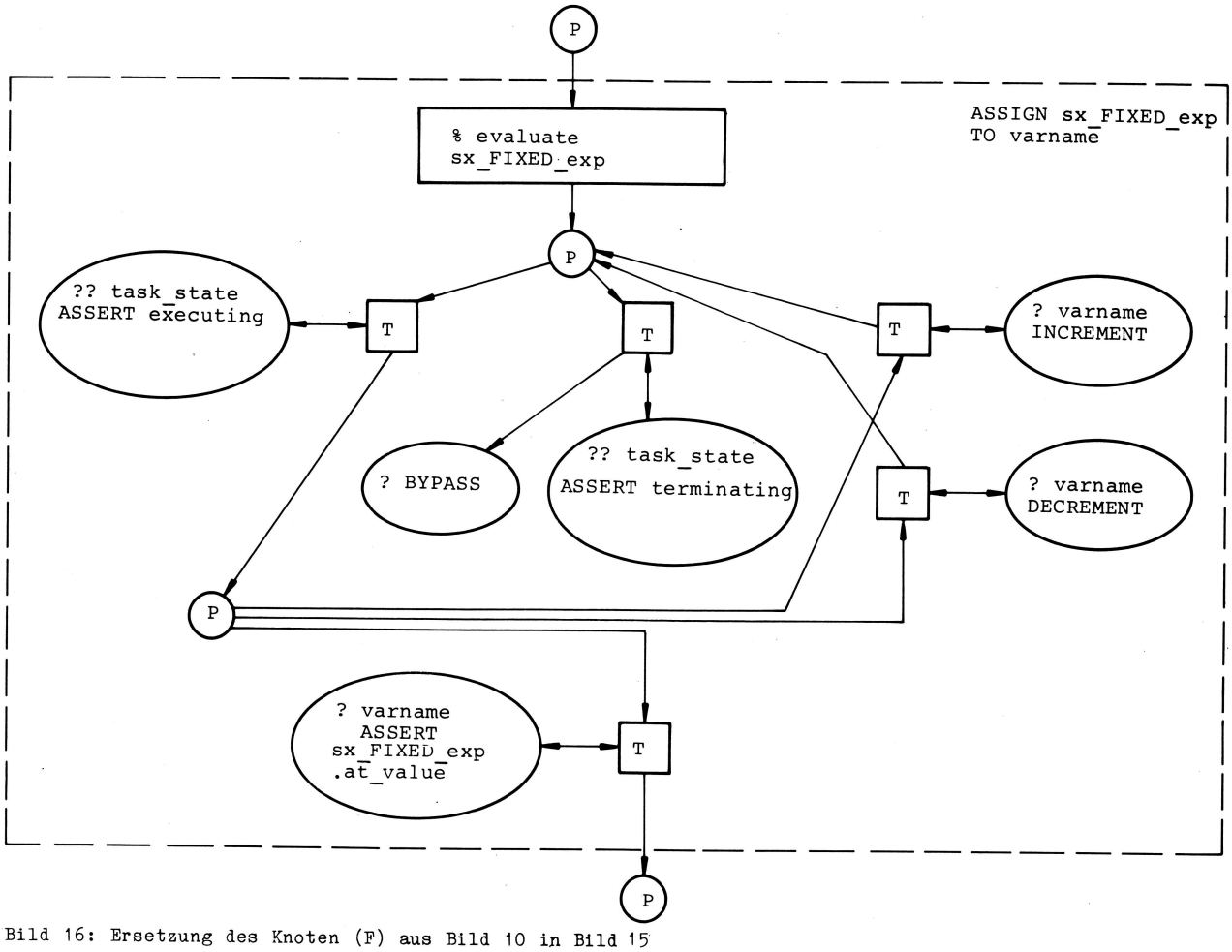


Bild 16: Ersetzung des Knoten (F) aus Bild 10 in Bild 15

eindeutigen Ersetzung bei unbenannten Nachbarknoten führen, ist die relative Lage der Kanten im Netz, in der sie K bzw. T_k berühren, ausschlaggebend. Neben dem obigen Beispiel 3.1-0 sei folgendes angegeben:

Beispiel 3.1-5

In Bild 17 findet man die Figur #27, unterer Teil, in der der Knoten (F) aus Bild 10 schon eingesetzt ist. Es geht jetzt darum, den Knoten [sx_loop_scope] durch das Teilnetz in Bild 18 zu ersetzen. Das dort abgebildete Teilnetz hat zwei verschiedene Nach barknoten, zu denen Pfeile hinführen. Wie sind diese Knoten zu identifizieren? Entsprechend der Lage: Der linke (P)-Nachbarknoten in Bild 18 wird mit dem linken in Bild 17 identifiziert. Entsprechend wird mit dem rechten verfahren. Das Ergebnis der Ersetzung findet man in Bild 19.

5) Die Identifizierung der Nachbarknoten von K und T_k kann auch über die Knotennamen geschehen. Knoten mit denselben Inschriften werden identifiziert. Hiermit ist es bei komplizierteren Verknüpfungen insbesondere möglich, einen Nachbarknoten von T_k mit mehreren von K zu assoziieren.

Beispiel 3.1-6

Betrachtet man die Figuren #40 und #41 aus dem DIN-Entwurf unter der Annahme, daß die Bedingung "WHERE at_task_op = sc_ACTIVATE" gilt, d.h. daß man den obersten Knoten (frame for a schedule ...) in Bild 20 und keinen der darunterliegenden ersetzen will, dann ist die Figur #41 aus dem DIN-Entwurf, wie in Bild 21 dargestellt, zu verwenden. Das dort dargestellte Teilnetz ersetzt den Knoten (frame for a schedule sc_ACTIVATE) in Bild 20. Die Verknüpfung des Teilnetzes in Bild

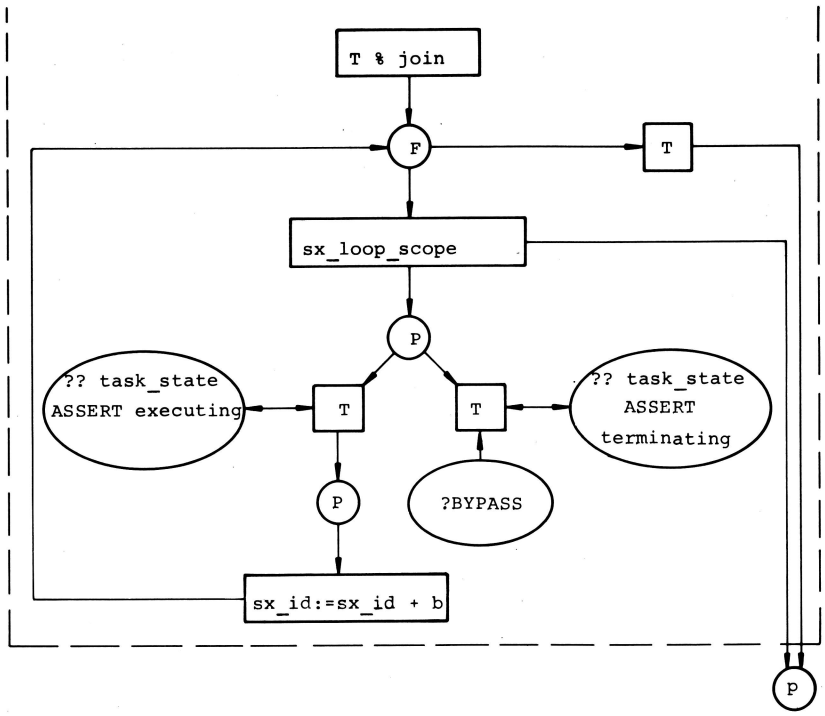


Bild 17: Figur #27, unteres Teilnetz, aus dem DIN-Entwurf

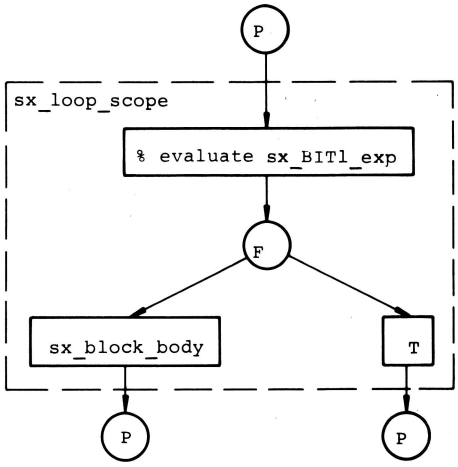


Bild 18: Figur #28 aus dem DIN-Entwurf

21 mit dem Knoten [? no activation pending] in Bild 20 muß vorgenommen werden. Die Doppelpfeile zu dem ersetzten Knoten werden z.T. durch mehrere Einfachpfeile ersetzt. Die weitere Identifizierung der Knoten ist so einfach, das das Ergebnis-Netz nicht noch einmal dargestellt werden soll.

3.2 Bedingte Ableitungen

Die Verfeinerung eines Netzes hängt generell vom einzelnen Programm, das zu erzeugen ist, also von der Wahl der Alternativen in den syntaktischen Produktionsregeln und von der möglichen Implementierung ab. Die Netze sind im DIN-Entwurf gewissen Ab-

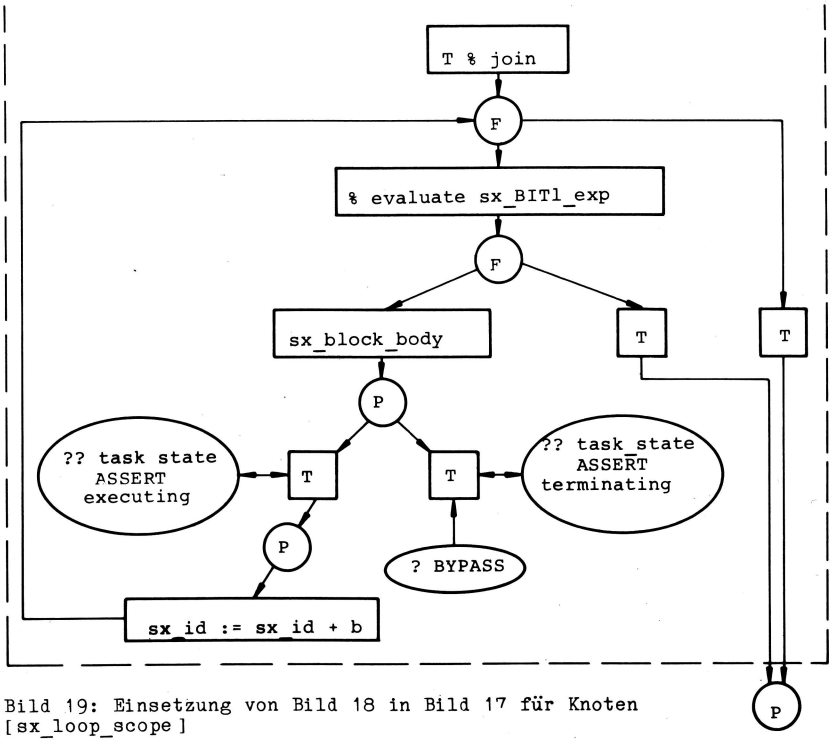


Bild 19: Einsetzung von Bild 18 in Bild 17 für Knoten [sx_loop_scope]

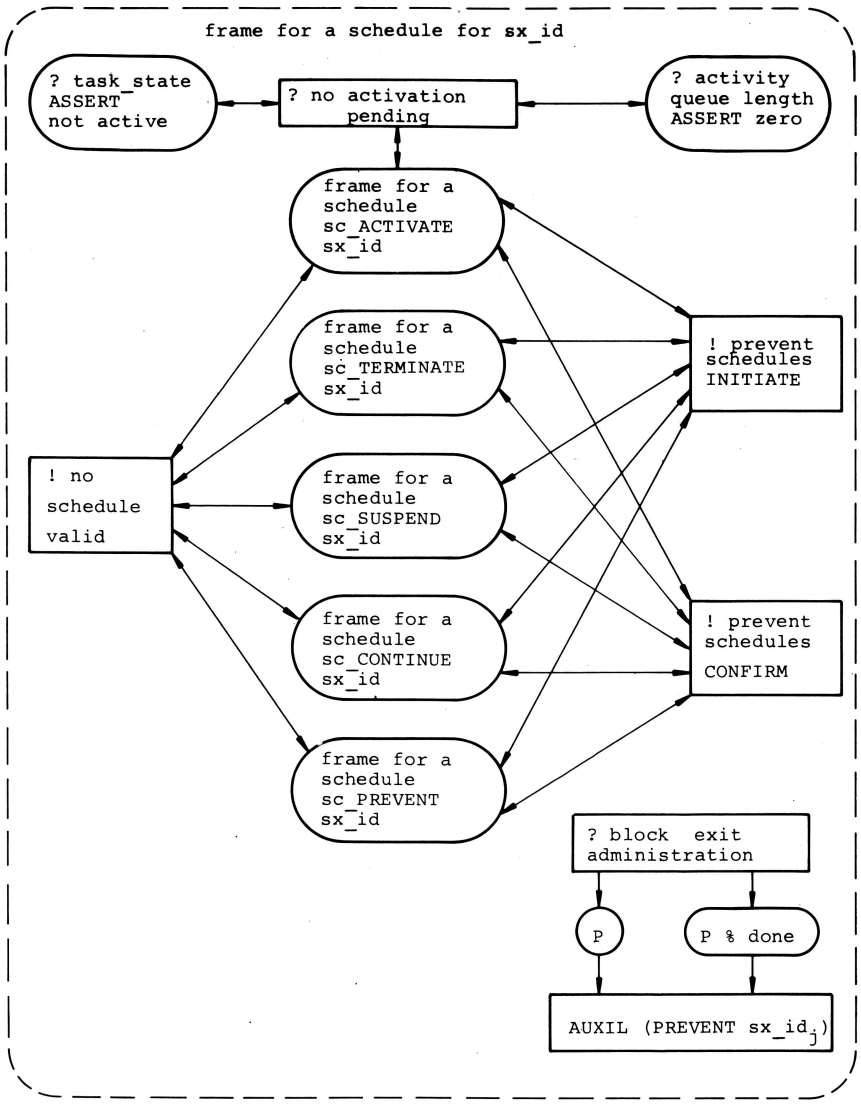


Bild 20: Figur #40 aus dem DIN-Entwurf

leitungen zugeordnet, deren Semantik sie beschreiben. Die vorgegebenen Netze bilden abstrakte Darstellungen der Konzepte und Implementationen gewisser syntaktischer Konstrukte.

Die durch die attributierte Grammatik beschriebenen Kontextbedingungen sind in den Knoten bzw. Teilnetzen als WHERE-Konstrukte beschrieben. Der Gültigkeitsbereich einer Kontextbedingung wird durch eine gestrichelte Kontur, interpretierbar als inneres Teilnetz, gekennzeichnet. Ein Knoten oder Teilnetz wird genau dann erzeugt, wenn diese Kontextbedingungen zutreffen. Ansonsten wird der Knoten (bzw. das Teilnetz) inclusive aller ihn berührenden Kanten aus dem Netz entfernt. Die Bedingung kann die bekannten Boole'schen Operatoren (AND, OR, ...) und Prädikate aus der attributierten Grammatik der Sprache Full PEARL, z.B. "IS THERE", enthalten (s. auch Beispiel 3.1-3).

Beispiel 3.2-1

Man betrachte die Figur #04 aus dem DIN-Entwurf im Bild 22. Es sei angenommen, daß die Bedingung "UNLESS sx_exp IS THERE" nicht zutrifft. Dann entfällt das rechte gestrichelt umrandete Teilnetz in Bild 22, wie in Bild 23 dargestellt. Alle zu den [T]-Knoten hin- und von ihnen wegführenden Kanten entfallen.

Im allgemeinen können Daten und Deklarationen als Stellen und Operationen durch Transitionen dargestellt werden. Die Verknüpfung einer Deklaration (Stelle) einer Ope-

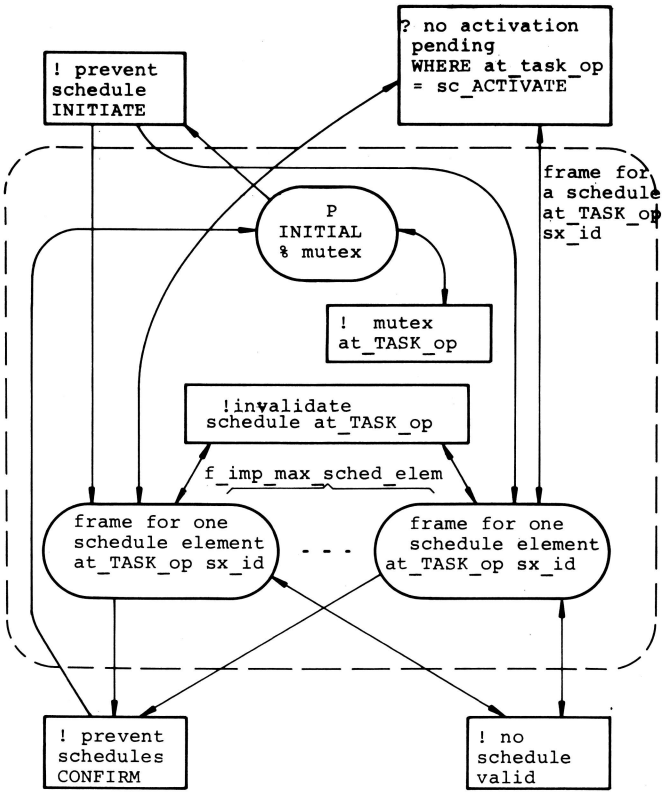


Bild 21: Figur #41 aus dem DIN-Entwurf

ration mit ihrem Aufruf (Transition) geschieht durch die im nächsten Abschnitt zu beschreibenden Konnektoren.

4. Konnektoren

Konnektoren beschreiben Kontextbeziehungen und semantische Eigenschaften eines Programms. Ihr Zweck ist in dem DIN-Entwurf dargestellt. Hier soll nur ihre Behandlung, d.h. insbesondere ihre Eliminierung, erörtert werden.

Konnektoren sind dadurch gekennzeichnet, daß ihrem Knotennamen ein Ausrufezeichen (!) oder Fragezeichen (? oder ??) voranstehen. Konnektoren mit Ausrufezeichen heißen Konnektordefinitionen; Konnektoren mit Fragezeichen heißen Konnektoranwendungen. Um eine u.U. entstehende Unsicherheit zu beseitigen, sei an dieser Stelle bemerkt, daß die Konnektoren mit einem vorangestellten Fragezeigen Kontextbeziehungen beschreiben, die schon zur Übersetzungszeit geklärt wer-

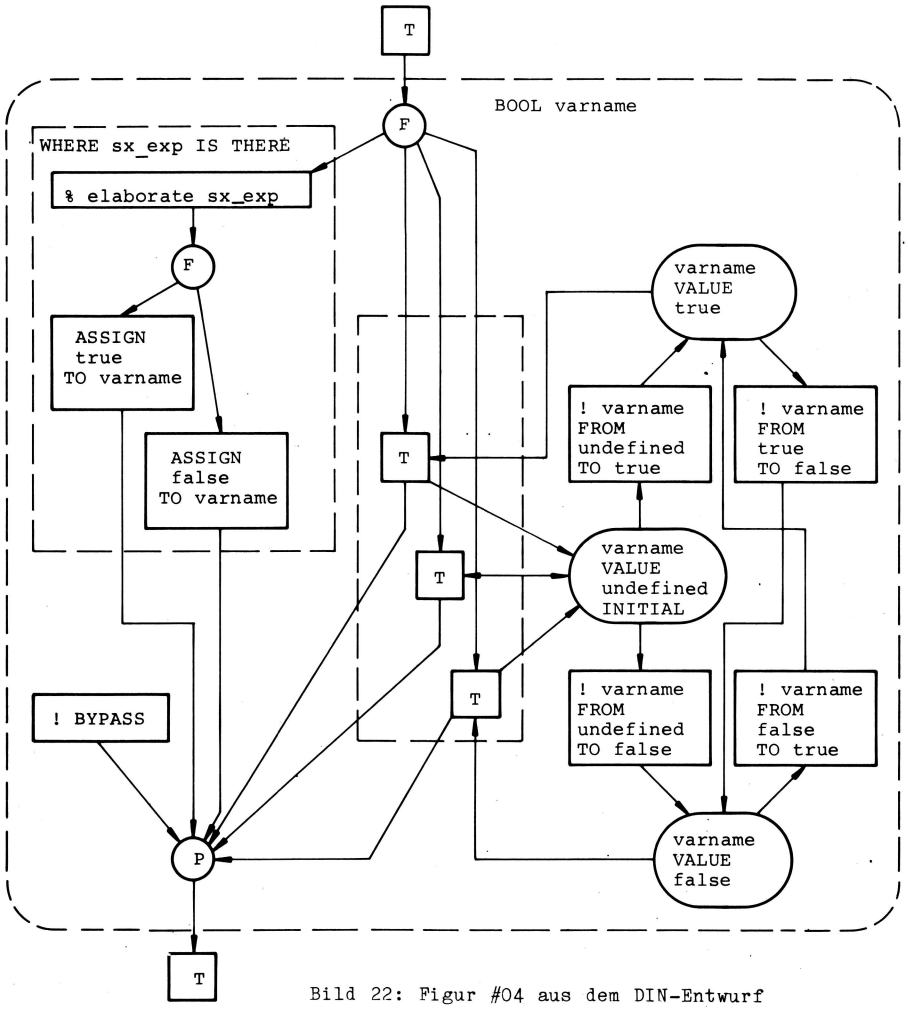


Bild 22: Figur #04 aus dem DIN-Entwurf

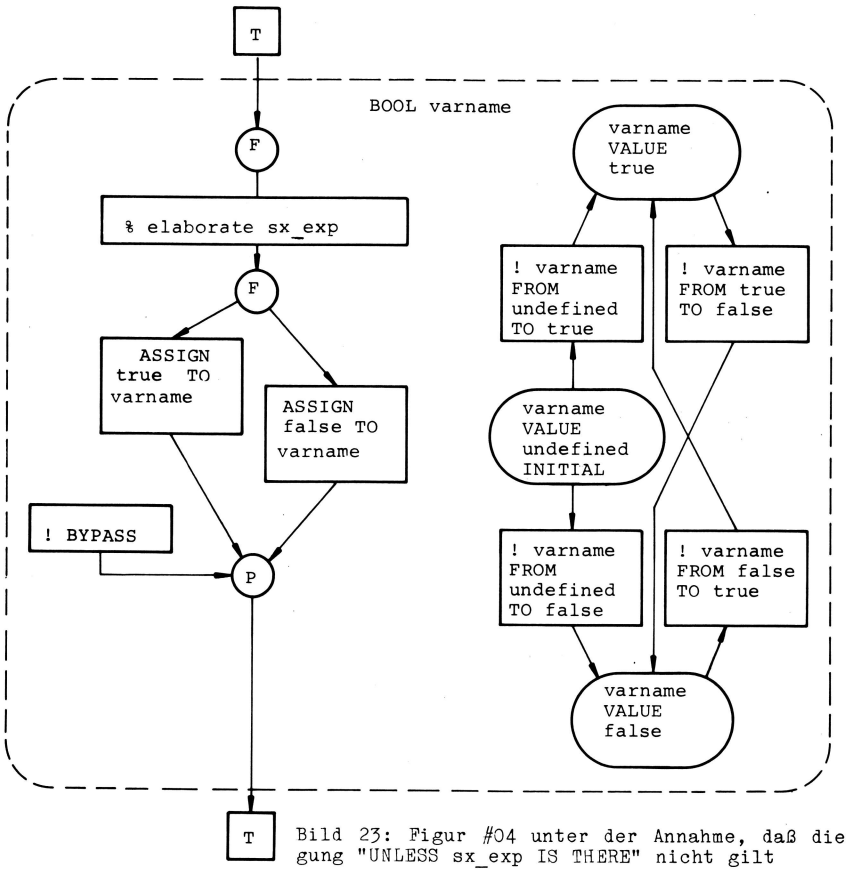


Bild 23: Figur #04 unter der Annahme, daß die Bedingung "UNLESS sx_exp IS THERE" nicht gilt

den können (statisches Binden), und die mit zwei vorangestellten Fragezeichen solche, die erst zur Laufzeit festliegen (dynamisches Binden). Das dynamische Binden wird in dem (statischen) Petri-Netz derart dargestellt, daß alle möglichen Abläufe modelliert werden. Die Tatsache, daß ein Konnektor ein oder zwei Fragezeichen besitzt, ist für die formalen Eliminationsregeln insofern relevant, als auf unterschiedliche Art von der Konnektoranwendung aus die dazugehörige Konnektordefinition zu suchen ist.

Will man ein Condition/Event-Netz mit ausschließlich elementaren Knoten herleiten, sind zuerst alle Ersetzungen von Knoten durch Teilnetze auszuführen. Ist man in diesem Produktions-Prozeß an dem Punkt angekommen, daß kein Knoten im Netz mehr ersetzt werden kann, müssen noch die Konnektoren eleminiert werden, um zu einem reinen Condition/Event-Netz gelangen zu können. Die bei der Substitiution von Knoten durch Teilnetze auftretenden Konnektoren dürfen also erst eliminiert werden, wenn das Netz restlos verfeinert ist. Es muß sichergestellt sein, daß nicht in einem späteren Verfeinerungsschritt Konnektoranwendungen auftreten, für die es keine Konnektordefinitionen mehr gibt, da diese schon eliminiert worden sind. Es ist also zweierlei zu beachten:

1. Die Substitutionsschritte müssen rekonstruierbar sein, insbesondere die eingesetzten Teilnetze.
2. Die Konnektoren dürfen erst eliminiert werden, wenn das Netz nicht weiter verfeinert werden kann.

Die <u>Eliminierung von Konnektoren</u> wird in drei Schritten durchgeführt.

1) Für jede Konnektoranwendung finde eine einzelne, eindeutige Konnektordefinition. Die Inschriften, insbesondere die Knotennamen, der Knoten müssen identisch sein bis auf die Konnektorkennzeichnungen (!, ? oder ??). Der Knoten der Konnektordefinition hat dabei eine andere Kontur als der bzw. die Knoten der Konnektoranwendung. Entweder sind alle Anwendungen S-Knoten und die Definition ein T-Knoten oder umgekehrt.

2) Ersetze jede Kante zu einer Konnektoranwendung durch eine Kante zu jedem der Knoten, zu dem es von der Konnektordefinition aus eine Kante gibt. Tue entsprechend dasselbe mit den Kanten, die von einer Konnektoranwendung wegführen. Dabei müssen Doppelpfeile gegebenenfalls in je einen Hin- und einen Rückpfeil aufgespalten werden.

3) Entferne alle Konnektoranwendungen, deren Pfeile im Schritt 2 ersetzt worden sind und die jetzt isolierte Knoten darstellen. Entferne dann alle Konnektordefinitionen, auch wenn es zu ihnen keine Konnektoranwendungen gegeben hat, incl. aller zu oder von ihnen wegführenden Kanten.

Der einfachste Fall ist die Verknüpfung zweier Netze über Konnektoren. Dieser soll als erster erläutert werden.

Beispiel 4-1

Man betrachte die Figur #03 aus dem DIN-Entwurf mit ihren beiden Teilnetzen, wie sie in Bild 24 dargestellt sind. Das obere Teilnetz besitzt zwei verschiedene Konnektoranwendungen, die zwei gleichbenannten Konnektordefinitionen des unteren Teilnetzes gegenüberstehen. Die Doppelpfeile zu den Konnektoranwendungen spalte

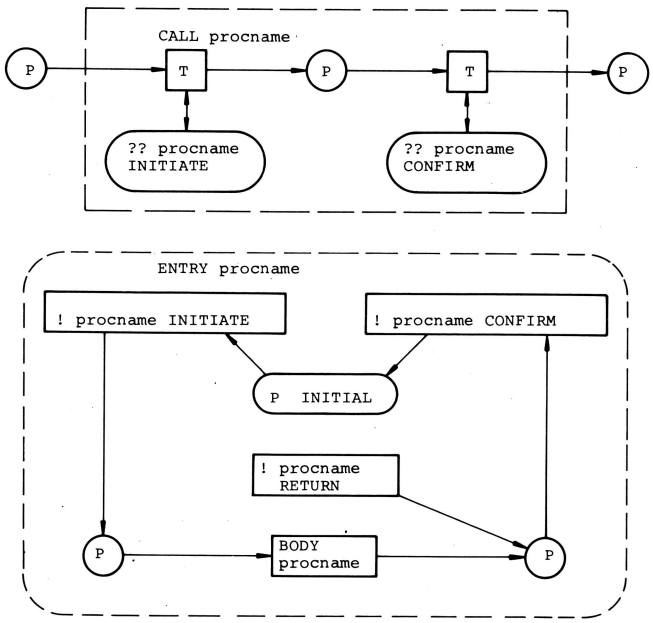


Bild 24: Figur #03 aus dem DIN-Entwurf

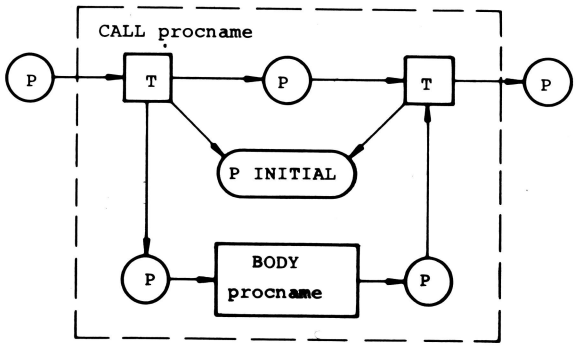


Bild 25: Verknüpfung der beiden Teilnetze aus Bild 24 durch Eliminierung der Konnektoren

in je zwei Einfachpfeile entgegengesetzter Orientierung auf. Identifiziere die Konnektoranwendungen mit ihren Konnektor definitionen und ersetze die Kanten. Die Konnektordefinition [!procname RETURN] findet keine -anwendung und wird gestrichen incl. der von ihr ausgehenden Kante. Das Ergebnis der Konnektoreliminierung findet man in Bild 25.

Dieses Beispiel kennzeichnet jedoch nur einen Spezialfall davon, daß dynamisches Binden kennzeichnende Konnektoren eliminiert werden sollen (also solche, deren Konnektoranwendungen durch "??" gekennzeichnet sind). Die wesentlichen (praktischen) Schwierigkeiten bei der Eliminierung von Konnektoren liegen bei der Suche zusammengehöriger Konnektoranwendungen einerseits und der dazugehörigen Konnektordefinition andererseits. Und bezüglich dieser Suche unterscheiden sich Konnektoranwendungen, die statisches Binden (also solche mit "?"), und die, die dynamisches Binden kennzeichnen, erheblich. Konnektordefinitionen gleichen Namens können mehrfach auftreten in einem Netz (z.B. [!BYPASS] in Beispiel 4-3) und die -anwendungen müssen in korrekter Weise den -definitionen zugeordnet werden. Deshalb besitzen Suche und Zuordnung von Konnektordefinitionen eine besondere Bedeutung. Beschäftigen wir uns als erstes mit Konnektoren, die dynamisches Binden kennzeichnen.

Betrachtet man die Verfeinerung eines Netzes N, so werden für einige Knoten K_i in N Teilnetze T_{ki} eingesetzt. In diesen Teilnetzen T_{ki} werden wiederum gewisse Knoten

K_{ij} durch Teilnetze T_{kij} ersetzt usw. Die Verfeinerungsstruktur eines Netzes läßt sich also als Baumstruktur auffassen, wobei das Ausgangsnetz N, das verfeinert wird, die Wurzel des Baumes ist und die Knoten K_i, die durch Teilnetze T_{ki} ersetzt werden, Unterbäume beschreiben. T_{ki} ist in dem Unterbaum i der Wurzelknoten, der wiederum Unterbäume (i,j) hat für all die Knoten K_{ij}, die durch Teilnetze T_{kij} ersetzt werden, usw. Die Wurzel N bilde die Ebene 0, die Knoten K_i die Ebene 1, die Knoten K_{ij} die Ebene 2 usw. Bei der Suche der Konnektordefinition zu einer gegebenen Konnektoranwendung K, die durch "??" gekennzeichnet ist, gehe, wie folgt, vor: Angenommen K tritt auf der Ebene n in dem Verfeinerungsbaum auf, und zwar in dem Teilnetz zum Knoten $K_{i1,i2,...,i(n-1),in}$. Prüfe nun, ob in demselben Teilnetz eine passende Konnektordefinition liegt oder in einem Teilnetz, das bezüglich der Verfeinerungsstruktur in dem Baum unterhalb des Knotens $K_{i1,i2,...,in}$ liegt; suche also alle Teilnetze des Unterbaums (i1,i2,...,in) nach einer passenden Konnektordefinition ab. Führt diese Suche nicht zum Erfolg, gehe bzgl. der Verfeinerungsstruktur einen Knoten in Richtung Wurzel zurück, also zum Knoten $K_{i1,i2,...,i(n-1)}$; untersuche das dazugehörende Teilnetz sowie alle Teilnetze des Baums (i1,i2,...,i(n-1)) in Richtung zu den Blättern. Die Suche erfolgt beim Absteigen zu den Blättern Ebene für Ebene.

Findet man auch in diesem Unterbaum keine passende Konnektordefinition, so gehe noch einen Knoten zurück im Gesamtverfeinerungsbaum des Netzes N, also zum Knoten $K_{i1,i2,...,i(n-2)}$ und untersuche dessen Unterbaum (i1,i2,...,i(n-2)). Wiederhole diesen Prozeß, bis eine erste passende Konnektordefinition im Unterbaum (i1,i2,...,ik) gefunden ist. Nimm dann die Konnektordefinition K', die als erste passende auftritt und die in dem Unterbaum (i1,i2,...,ik) die kleinste Ebenennummer besitzt. Zeichnet man den Verfeinerungsbaum von oben nach unten, d.h. derart, daß die Wurzel oben liegt und die Blätter des Baums unten, so nimm die Konnektordefinition, die am weitesten oben in dem betrachteten Unterbaum liegt.

Beispiel 4-2

Als erstes soll die Figur #06 (siehe Bild 13) aus dem DIN-Entwurf in das Netz aus Beispiel 3.2-1 in Bild 23 eingesetzt werden. Im Bild 26 ist das Ergebnis der Einsetzung abgebildet. Damit das Netz auf normalem Papierformat noch darstellbar ist, soll die Eliminierung der Konnektoren [!varname FROM undef/true/false TO true/false] vorgezogen werden. Es wird deshalb der Einfachheit halber angenommen, daß weitere Verfeinerungen keine weiteren Konnektoranwendungen für die Konnektordefinitionen im rechten Teilnetz produzieren. Die Konnektordefinition [!BYPASS] darf nicht gestrichen werden, da sie die Konnektoranwendungen aus den (F)-Knoten zu befriedigen hat. Die Konnektoranwendungen (?? varname ASSERT true/false) bleiben vorerst stehen. Sie werden ja sowieso nicht gestrichen nach dem oben besprochenen Eliminierungsverfahren. Die Doppelpfeile, die zu den zu eliminierenden Konnektoranwendungen führen, müssen durch zwei Einfachpfeile entgegengesetzter Orientierung ersetzt werden. Führt man dann die oben genannten drei Eliminierungsschritte aus, erhält man das Netz in Bild 27. Bei den Konnektoranwendungen (?? varname FROM undefined/true/false TO true/false) lagen die dazugehörigen Konnektordefinitionen in demselben Teilnetz. Bei der Suche der Konnektordefinitionen [! varname ASSERT true/false] müssen die Verfeinerungen der Knoten (varname VALUE true/false) untersucht werden. Eines der beiden (im wesentlichen gleichen) Teilnetze ist in Bild 28 dargestellt (siehe Figur #05 im DIN-Entwurf). Dort treten die gesuchten Konnektordefinitionen auf. Wir setzen also diese Teilnetze in das Bild 27 ein und eliminieren gleich die Konnektoren. Das Ergebnis ist in Bild 29 dargestellt.

Es bleibt noch der Fall zu besprechen, daß Konnektoren, die statisches Binden kennzeichnen, eliminiert werden sollen, also solche, bei denen die Konnektoranwendungen durch "?" gekennzeichnet sind. Die Frage ist wieder, wie findet man zu einer gegebe-

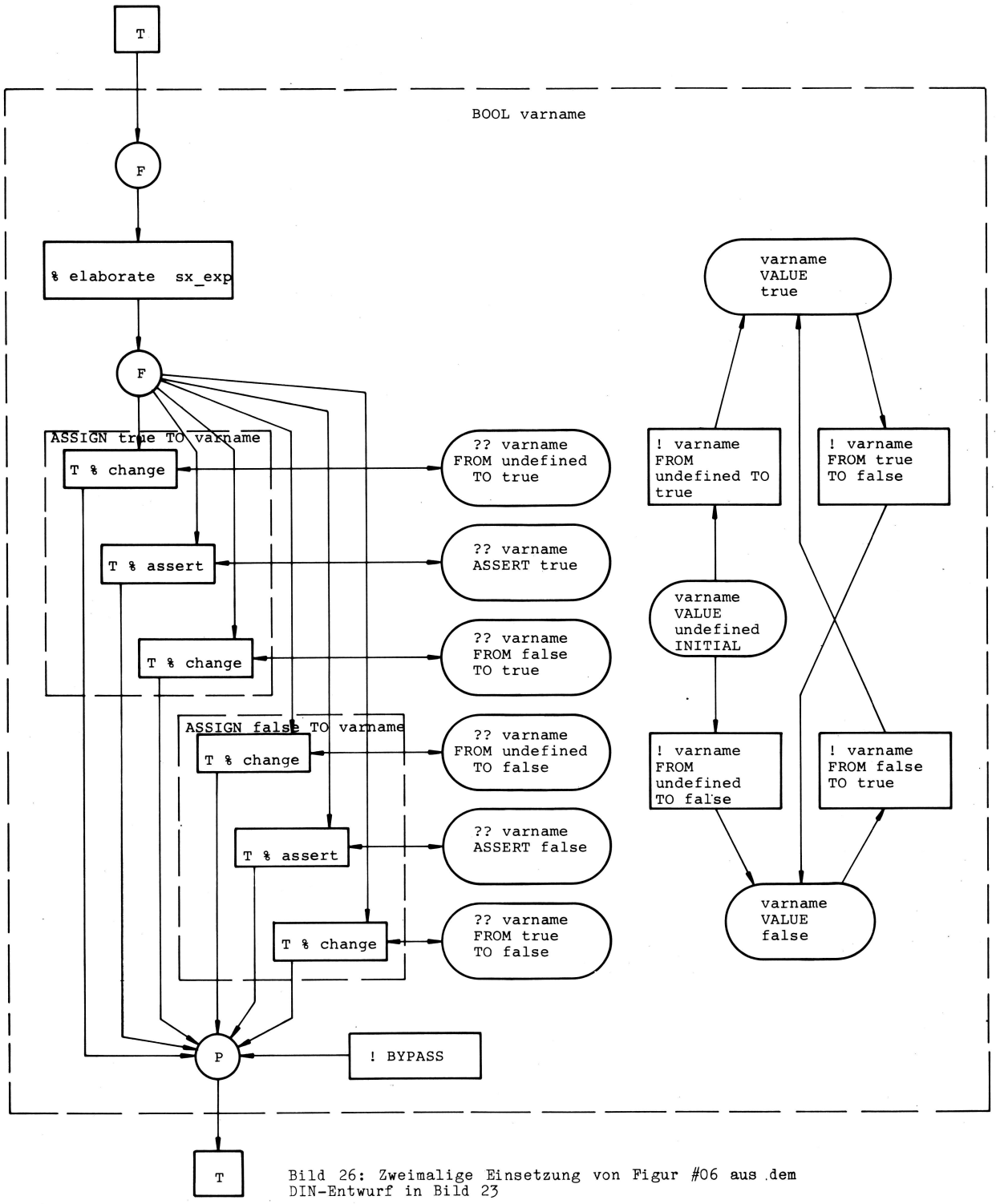


Bild 26: Zweimalige Einsetzung von Figur #06 aus dem DIN-Entwurf in Bild 23

nen Konnektoranwendung K die dazugehörige -definition. Sei wieder ein Netz N mit seinem Verfeinerungsbaum, wie oben besprochen, gegeben. Eine Konnektoranwendung ist dann mit der in der Verfeinerungsstruktur in Richtung des Baumknotens nächstliegenden Konnektordefinition zu identifizieren. Anders ausgedrückt: Wenn eine Konnektoranwendung K auf der Ebene n in dem Baum in dem Teilnetz zum Knoten $K_{i1,i2,\ldots,in}$ auftritt, dann sucht sie ihre Konnektordefinition erst in ihrem eigenen Teilnetz, dann in dem zum Knoten $K_{i1,i2,\ldots,i(n-1)}$, dann in dem zu $K_{i1,i2,\ldots,i(n-2)}$ usw., bis sie die erste passende Konnektordefinition K' gefunden hat, und bricht ab mit der Suche. Diese Konnektordefinition K' wird mit K assoziiert.

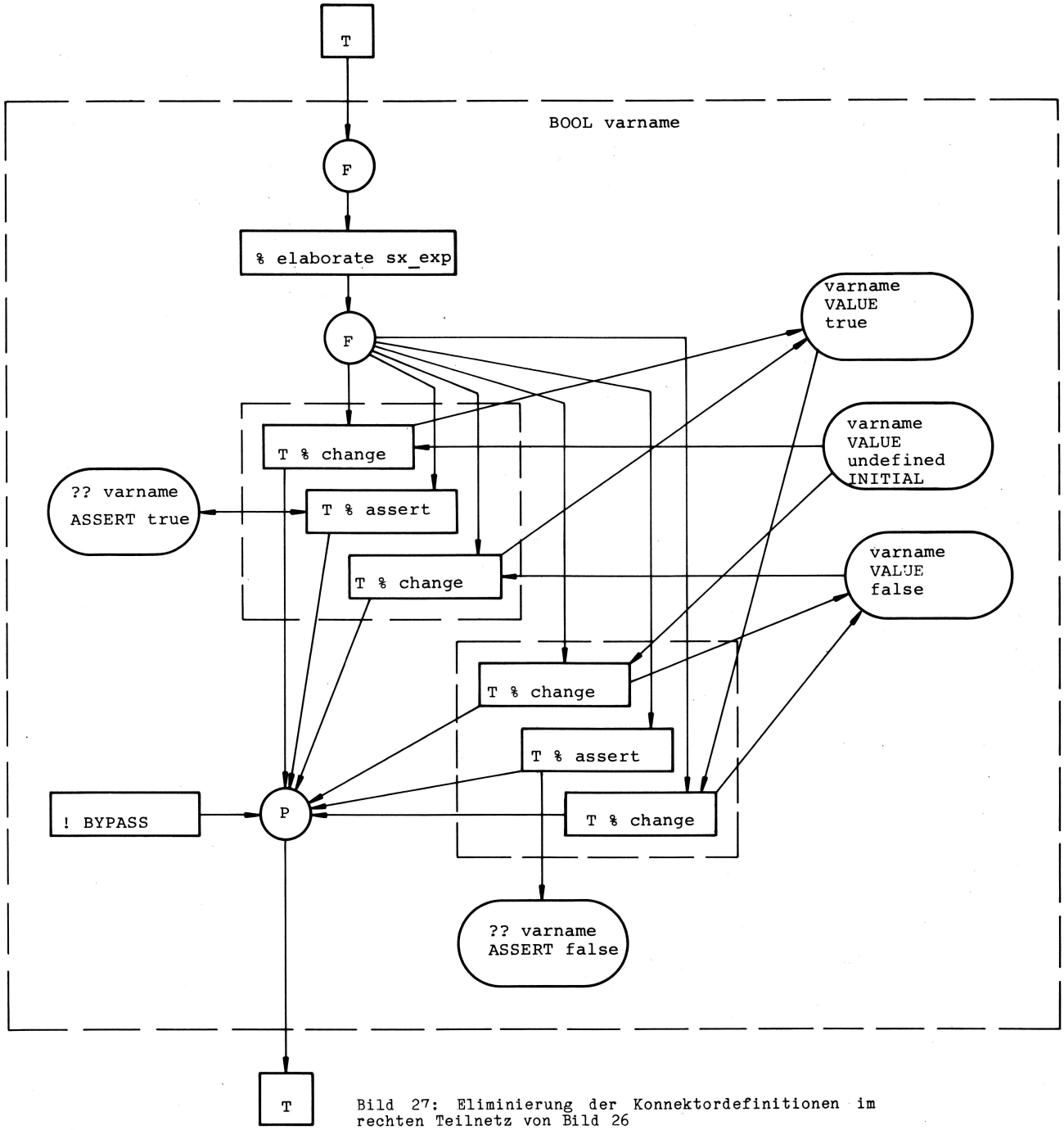


Bild 27: Eliminierung der Konnektordefinitionen im rechten Teilnetz von Bild 26

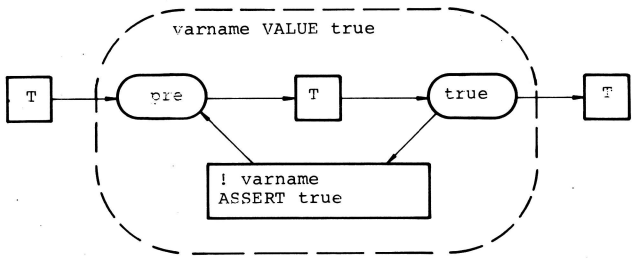


Bild 28: Figur #05 aus dem DIN-Entwurf für "valname = true"

Beispiel 4-3

Das Auftreten mehrerer Konnektordefinitionen sei an der Figur #38 aus dem DIN-Entwurf in Bild 30 demonstriert, in die die Figuren #11 in Bild 31 und #02 in Bild 10 eingesetzt werden sollen. Wieder sollen die Einschränkungen über die Verfeinerungen aus dem vorigen Beispiel gel-

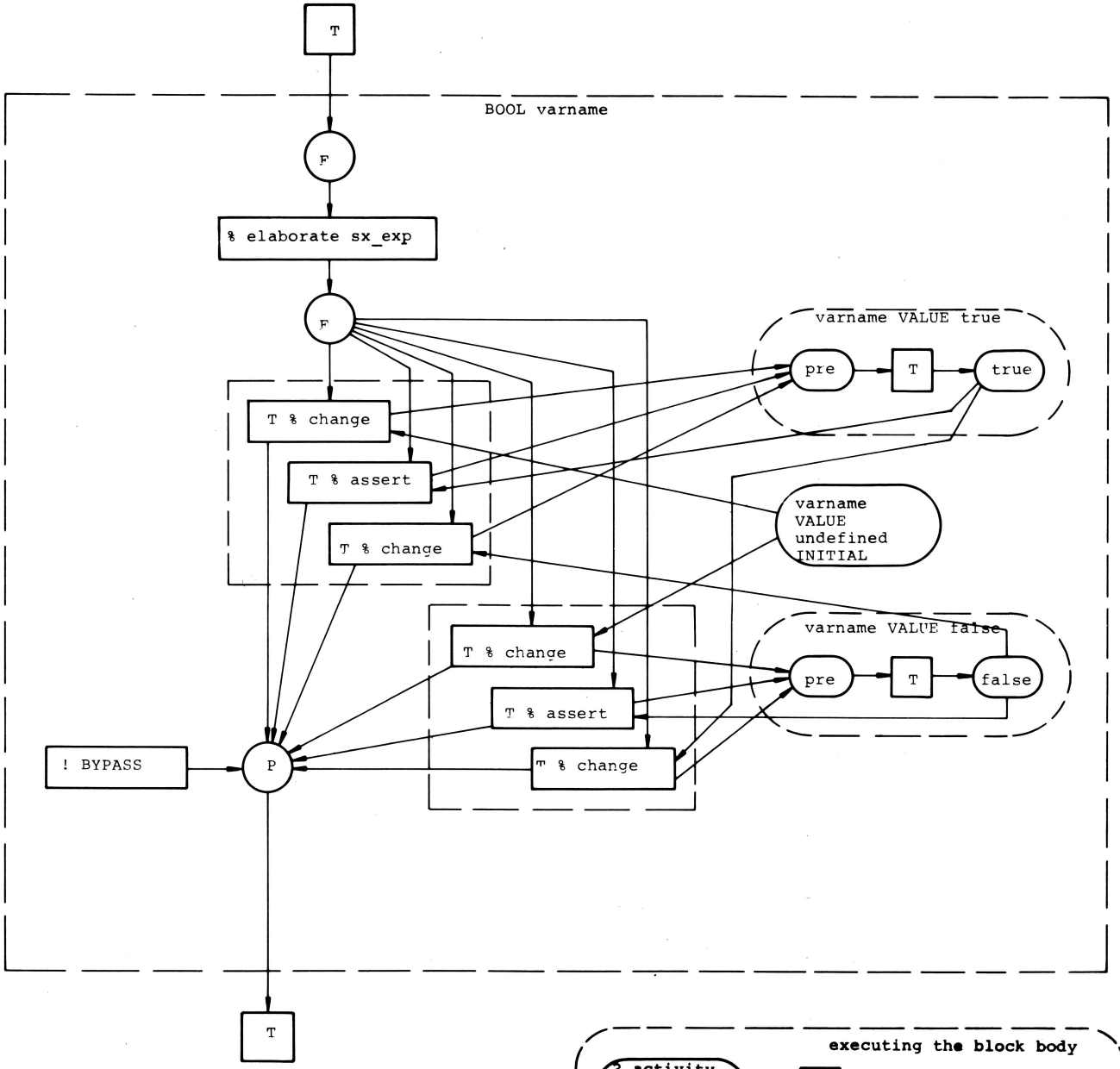


Bild 29: Bild 27 nach Einsetzung des Bildes 28 und Eliminierung der Konnektoren

ten. Es sei angenommen, daß der Knoten (task_state) keine Konnektoren (?BYPASS) mehr erzeugt. Nach der Einsetzung hat das Netz die Struktur wie in Bild 32, wenn im Teilnetz [sx_block_body] die Knoten (F) auch wiederum durch das Teilnetz in Bild 10 ersetzt worden sind. Die Konnektoranwendungen (?BYPASS) im Teilnetz [sx_block_body] werden mit der Konnektordefinition [!BYPASS] in diesem Teilnetz identifiziert. Würde man das Netz noch weiter verfeinern, müßten die Knoten (?BYPASS), die durch die Verfeinerung der Knoten [sx_

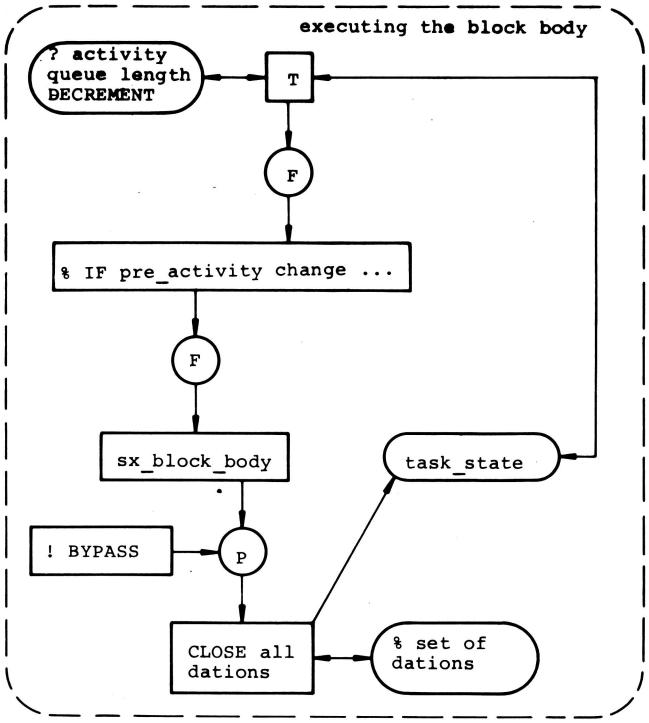


Bild 30: Figur #38 aus dem DIN-Entwurf

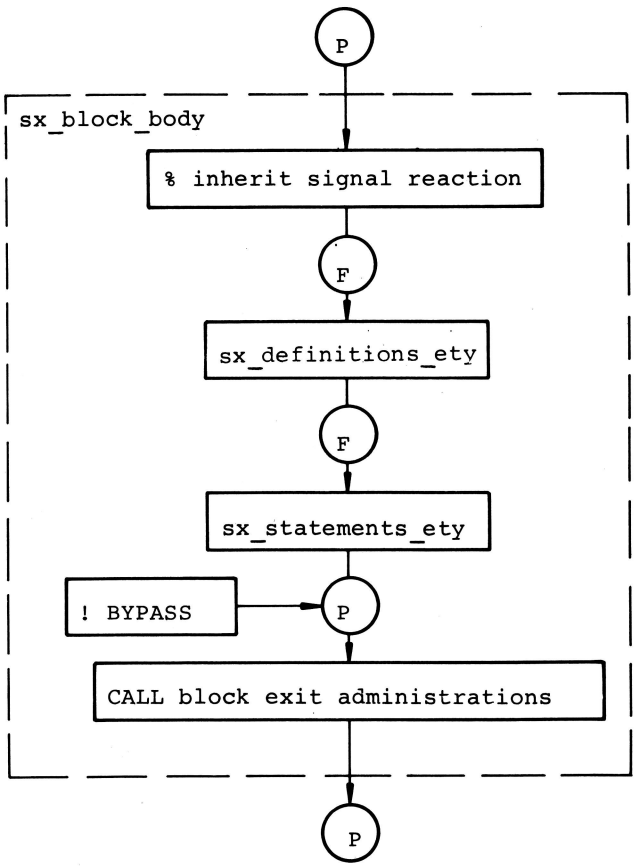


Bild 31: Figur #11 aus dem DIN-Entwurf

definitions_ety] und [sx_statements_ety] produziert werden, mit dem [!BYPASS]-Knoten im Teilnetz [sx_block_body] identifiziert werden. Die beiden übrig bleibenden Konnektoranwendungen (?BYPASS) ganz oben werden mit der Konnektordefinition [!BYPASS] ganz unten in Bild 32 assoziiert. Diese Konnektoren liegen in einem äußeren Verfeinerungsblock bezüglich des Teilnetzes [sx_block_body]. Das Ergebnis der Konnektoreliminierung findet man in Bild 33.

Zum Abschluß möchte ich noch Prof. Dr. H. Rzehak, Hochschule der Bundeswehr, und Dr. E. Wegner, GMD, für die Durchsicht des Manuskriptes und die vielen Anregungen danken.

Literatur

[1] DIN 66 253 Teil 2 (Full PEARL), DIN-Entwurf, Nov. 1980, Beuth Verlag

[2] E. Wegner: Transforming nets along the syntactic production of programs; First European Workshop on the Application and Theory of Petri-Nets, Strasbourg, Sept. 1980, (revised paper)

[3] E. Wegner: Semantics of a Language for Describing Systems and Processes; IST Report 36 revised, manuscript Jan. 1978

[4] E. Wegner: private communication

[5] K. Zuse: Petri-Netze aus der Sicht des Ingenieurs; Vieweg + Sohn Verlag, Braunschweig, 1980

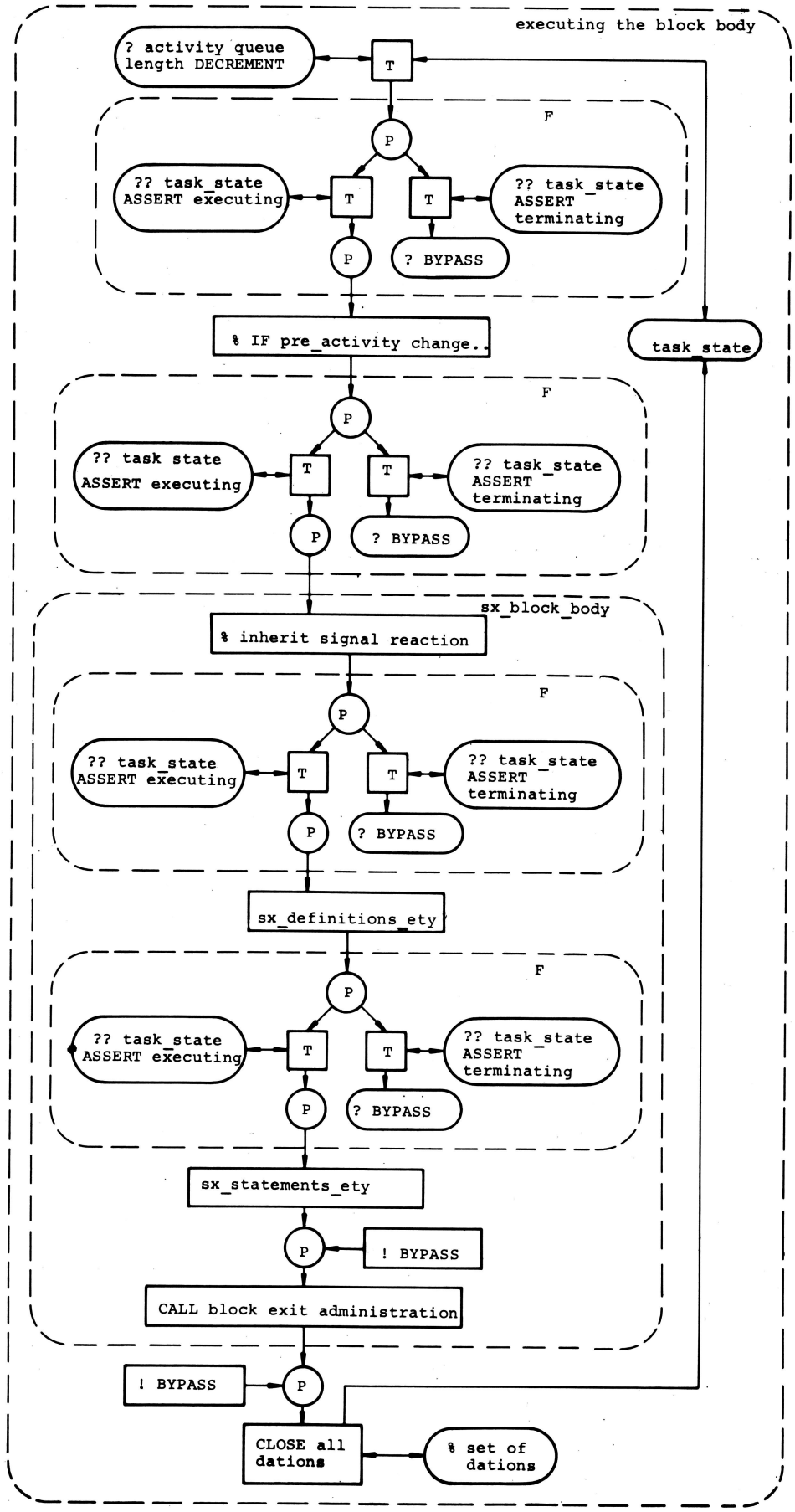


Bild 32: Bild 31 nach Einsetzung der Teilnetze

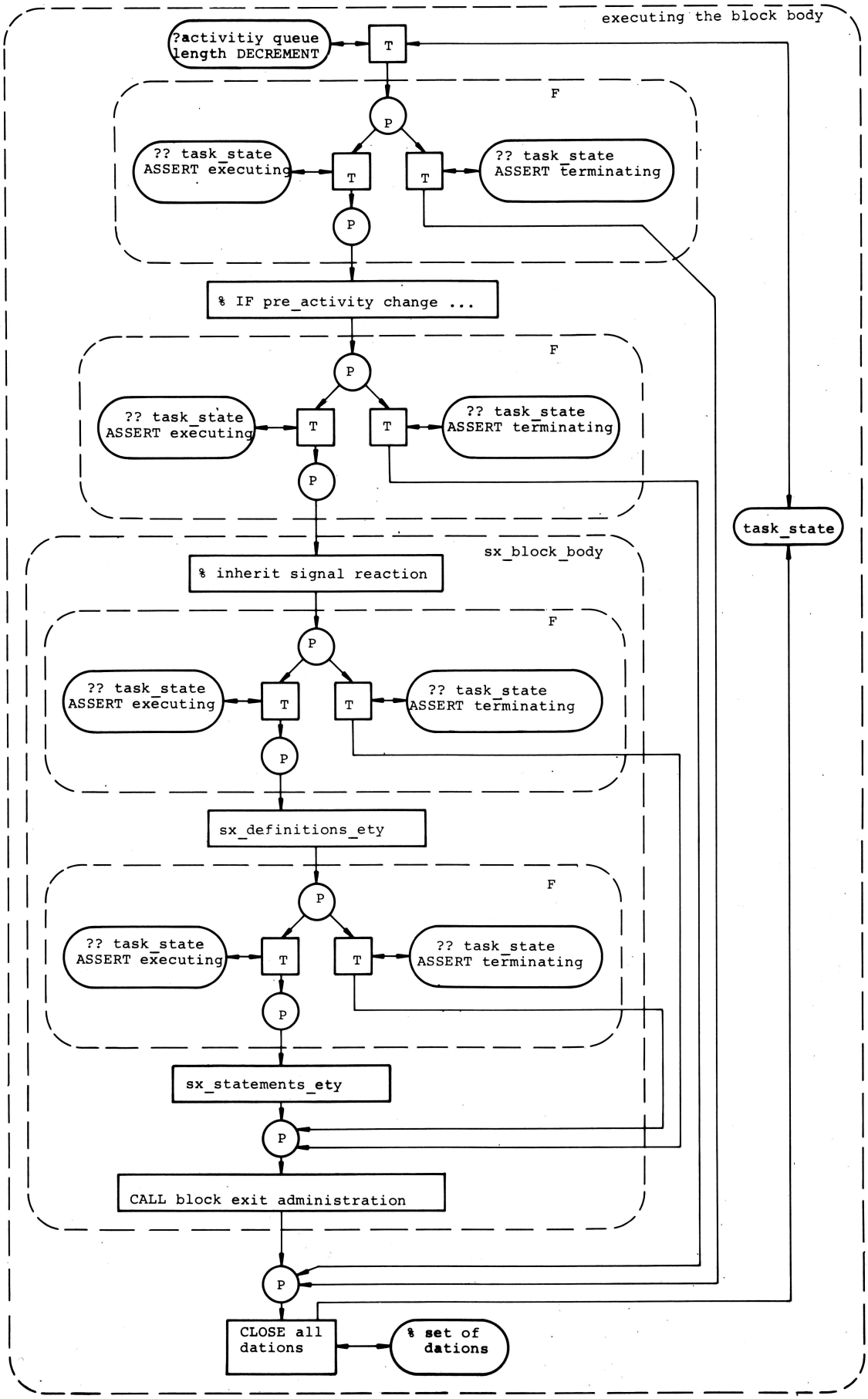


Bild 33: Bild 32 nach Eliminierung der Konnektoren (?BYPASS) und [!BYPASS]

Eine Methode zur schnelleren Entwicklung und übersichtlichen Dokumentation von PEARL-Programmen.

von Prof. Dr. L. Frevert, Bad Salzuflen

Zusammenfassung

Es wird eine Methode beschrieben, mit der aus einem Grobentwurf durch Verfeinerung in kleinen Schritten ein Feinentwurf entwickelt wird, aus dem sich durch Anwendung eines Preprozessors ein kompilierbares Programm ergibt. Die dabei entstehende Dokumentation des Entwurfs-Prozesses ist gleichzeitig eine übersichtliche Dokumentation des Programmes.

Abgrenzung gegen andere Verfahren

Die hier beschriebene Methode wurde aufgrund von sehr guten Erfahrungen entwickelt, die mit der Systemsprache META-S /1/ gemacht worden waren. META-S und die Schulsprache ELAN /2/ enthalten Sprachelemente für die Top-down-Entwicklung von Algorithmen und sind daher sehr geeignet für die schnelle Entwicklung schwieriger Algorithmen. Ursprünglich war nur beabsichtigt, dieses Prinzip für die Entwicklung von PEARL-Algorithmen nutzbar zu machen, wobei ein Nachteil der vorerwähnten Sprachen - hohe Schreibarbeit und schlechte Lesbarkeit längerer Programme - vermieden werden sollte. Es stellte sich jedoch heraus, daß sich auch große Vorteile durch die Anwendung der Methode auf die Beschreibung von Parallel-Arbeit und Problemdaten, sowie bei der Niederschrift von Programm-Spezifikationen ergaben.
Zur Abgrenzung gegenüber anderen Spezifikations- und Entwurfsverfahren /3/ ist zu sagen, daß diese Methode ein Werkzeug für die Programm-Entwicklung ab der Feinspezifikation darstellt, und daß kompilierbare Programme automatisch aus den Entwürfen erzeugt werden. Das dafür benötigte Hilfsmittel ist leicht zu implementieren (der Prototyp des Preprozessors wurde an einem Tage geschrieben). Sie reicht insofern bis in die Test- und Wartungsphase, als nur für Testzwecke benötigte Programmteile durch sehr einfache Maßnahmen aus den Programmen entfernt, bzw. wieder hineingenommen werden können. Die Methode bedient sich keiner speziellen Entwurfs-Sprache, sondern setzt nur PEARL-Kenntnisse voraus. (Anmerkung: da die Methode an sich nicht sprachgebunden ist, wurde sie auch schon für die Entwicklung von Mikroprozessor-Assemblerprogrammen benutzt.)

Aufgaben und Lösungen

Vor die Aufgabe gestellt, ein Programm zu entwerfen und zu schreiben, steht der Entwickler in folgender Situation: er hat eine mehr oder weniger ungenaue Beschreibung dessen, was das Programm leisten soll, und eine ungefähre Vorstellung, wie die Aufgabe zu lösen sein könnte.
Die beste Methode, hier zu einer Lösung zu kommen, besteht darin, die Aufgabe grob in Teilaufgaben zu zerlegen; es sollten nicht mehr als 10-15 sein, damit man den Überblick behält. Die einzelnen Teilaufgaben werden ihrerseits wieder zerlegt; mit der Zerlegung von Teilaufgaben in Teilaufgaben wird so lange fortgefahren, bis alle Teilaufgaben so klein geworden sind, daß sie einzeln gelöst werden können. Wenn dies gelingt, ist damit die ganze Aufgabe gelöst.

Es kann jedoch geschehen, daß man bei dem Versuch, eine Aufgabe durch schrittweise Verfeinerung zu lösen, in eine Sackgasse gerät: für irgendeine Teilaufgabe ist weder eine Lösung noch eine weitere Zerlegung zu finden. Falls die Aufgabe überhaupt lösbar war, wird dies in der Regel dadurch verursacht, daß irgendeine frühere Zerlegung falsch durchgeführt worden ist. Wenn man solche Fehler finden will, ohne ganz von vorn beginnen zu müssen, muß man alle Verfeinerungen genau notiert haben.

Traditionsgemäß werden Programmabläufe graphisch, z.B. mit Ablaufplänen oder mit Struktoprogrammen dargestellt. Wenn man das auch bei der Aufgabenlösung durch schrittweise Verfeinerung tut, muß man entweder viel zeichnen oder Zwischenschritte weglassen. Beides ist schlecht: wenn man viel zeichnet, vergißt man die guten Ideen, welche man über den Fortgang der Verfeinerung hatte; wenn man Zwischenschritte nicht mitzeichnet, findet man die Stellen nicht, wo man in die Irre gegangen ist. Einzige Abhilfe: nicht zeichnen,

sondern aufschreiben, wie man sich den Programmablauf denkt. Am besten tut man das mit Befehlen, die aus ganzen Sätzen - auch mehreren - bestehen. Wenn solche Befehle mehrmals ausgeführt oder mal das eine, mal das andere getan werden muß, verwendet man Konstruktionen wie bei der Programmierung in einer höheren Sprache: Wiederhole .... Ende, Falls .... dann .... sonst. Mit anderen Worten: man sollte mit Pseudo-Code programmieren. Da man bei der schrittweisen Verfeinerung jeden Pseudo-Code-Befehl durch eine Folge von Pseudo-Code-Befehlen genauer erklären muß, verliert man aber ohne besondere Vorkehrungen sehr schnell die Übersicht, wie Pseudo-Code-Befehle und ihre erklärenden Pseude-Code-Befehlsfolgen zusammengehören. In den Programmiersprachen META-S und ELAN wird dieses Problem dadurch gelöst, daß einfach der Pseude-Code-Befehl noch einmal vor seiner Verfeinerung geschrieben wird. Das ist nicht gut: erstens muß man sehr viel zweimal schreiben, und zweitens ist es mühsam, beim Lesen eines längeren Programmes die Stelle zu suchen, wo genau der gleiche Satz noch einmal vorkommt.

Besser ist es, wenn man jeden Pseudo-Code-Befehl und seine verfeinernde Erklärung mit derselben Kennziffer versieht. Die Kennziffern wählt man so, daß man ihnen ansieht, zu welchem Teil des Gesamtproblems sie gehören: zum Befehl mit der Kennziffer 218F gehören in der Verfeinerung die Befehle mit Kennziffern 218F1, 218F2 usw.. Die Kennziffer schreibt man am besten links vor die Pseudo-Code-Befehle, damit beide dicht zusammen in einer Zeile stehen. Wenn man außerdem die Kennziffern vor den Befehlen etwas einrückt und die vor den zugehörigen Verfeinerungen direkt an den Rand schreibt, lassen sich zusammengehörende Kennziffern auch in längeren Programm-Entwürfen erstaunlich schnell finden, zumal sie ohnehin nicht völlig unsortiert stehen.

Nachdem man das Problem an allen Stellen durch Verfeinerung so präzisiert hat, daß man die Lösung genau kennt, könnte man das Programm in einer Programmiersprache niederschreiben. Das wäre wieder schlecht: Erstens wäre es dann schwierig, Programm und Pseudo-Code zu vergleichen, weil der Pseudo-Code in der Reihenfolge steht, wie man gedacht hat, während die Befehle des Programmes in der Reihenfolge stehen müssen, in der sie ausgeführt werden sollen. Zweitens wird man beim Programmtest feststellen, daß man an einigen Stellen doch falsch gedacht hat; dann ist die Versuchung sehr groß, nur das Programm zu ändern, und der Programm-Entwurf wird als Teil der Programm-Dokumentation unbrauchbar.

Besser ist es, wenn man die Programmiersprach-Zeilen und -Abschnitte als letzte Stufe der Verfeinerung zwischen den Pseudo-Code schreibt. Dann braucht man letzteren nur noch als Kommentar zu kennzeichnen und die Entwurfs-Zeilen in eine andere Reihenfolge zu bringen, um ein kompilierbares Programm zu erhalten, in dem der Pseudo-Code als Kommentar steht. Diese Arbeit ist so einfach, daß dafür ein Preprozessor geschrieben worden ist. Um dem seine Arbeit zu erleichtern, werden die Kennziffern mit einem führenden Sonderzeichen geschrieben.

Dieser Preprozessor kann gleichzeitig nach Fehlern suchen, die man beim Schreiben des Programm-Entwurfs gemacht haben könnte: Kennziffern müssen jeweils mindestens zweimal vorkommen und als Kommentar gekennzeichnet sein, Kommentar-Kennzeichen können vergessen oder zuviel notiert worden sein. Bei sehr umfangreichen Programmen erzeugt der Preprozessor auf Wunsch eine Kopie des Programm-Entwurfs, bei der Pseudocodes und Verfeinerungen durch Zeilennummer-Verweise auf den jeweils anderen Partner ergänzt sind.

Ein Beispiel

B i l d 1 zeigt den Anfang des Preprozessor-Programmentwurfs. Es ist ziemlich selbst-erklärend. Bemerkenswert ist, daß auf allen Ebenen der Verfeinerung bereits PEARL-Zeilen in den Pseudo-Code eingestreut sein können.

B i l d 2 zeigt einen Teil des daraus erzeugten PEARL-Programmes, dem in B i l d 1 die Zeile 21 entspricht. Die Zeilen-Numerierung hat sich gründlich geändert, weil der Preprozessor wegen der Reihenfolge der Kennziffern B i l d 1, Zeilen 7, 8, 12, 13, 14 Systemteil, Spezifikationen, Deklarationen und Prozeduren vor der Task eingeordnet hat. Er liest nämlich eine Verfeinerung nur solange monoton ein und gibt sie aus, solange sie aus PEARL-Code besteht. Eine Pseudocode-Anweisung wird ebenfalls eingelesen und ausgegeben; dann aber geht der Preprozessor zur Ausgabe der zugehörigen Verfeinerung über, wobei er darin enthaltene Pseudocode-Anweisungen genauso behandelt; am Ende jeder Verfeinerung wird in der jeweils darüber liegenden Ebene fortgefahren. Da das MODEND am Ende der obersten Verfeinerungsebene steht, kommt es automatisch an den Schluß des Programmes.

```
 1. /* ENTWURF EINES PEARL-PROGRAMMES ZUR UMWANDLUNG EINES NACH TOP-DOWN-
 2. METHODEN GESCHRIEBENEN PEARL-PROGRAMMENTWURFS IN EINE KOMPILIERBARES
 3. PROGRAMM
 4. VERSION 15.6/21.5.81
 5. AUTOR:FREVERT
 6.              */ MODULE PRPRO; /*
 7.    #1        SYSTEMTEIL
 8.    #2        PROBLEMTEIL
 9.              */ MODEND; /*
10. ==========================================================================
11. #2            */ PROBLEM; /*
12.    #21       PASSIVE OBJEKTE
13.    #22       UNTERPROGRAMME
14.    #23       BEARBEITUNGSTASK
15. --------------------------------------------------------------------------
16. #23           */ MAIN:TASK GLOBAL; /*
17.    #231      ERDEFFNE DATIONS UND FUEHRE STEUERDIALOG
18.    #232      LIES ENTWURF AUS STREAM-DATION UND BAUE ADRESSBUCH AUF.
19.                MELDE DABEI FEHLER
20.    #233      GIB AUF WUNSCH REFERENZEN-LISTE AUS
21.    #234      LIES ENTWURF IN RICHTIGER REIHENFOLGE DER ZEILEN UND GIB
22.                KOMPILIERBARES PROGRAMM AUS (AUF WUNSCH UND WENN DER ENTWURF
23.                 KEINE GROBEN FEHLER ENTHAELT)
24.    #235      SCHLIESSE DATEIEN UND MELDE FEHLERZAHL
25.              */ END; /*
26. #231          */
27.                OPEN TASTEN;
28.                OPEN SCHIRM;
29.                 PUT 'WELCHE DATEI SOLL BEARBEITET WERDEN?' TO SCHIRM;
30.                GET ENTWURFSNAME FROM TASTEN;
31.                ENTWURFSDATEINAME:=ENTWURFSNAME)(' ')('-TX';
32.                ZIELDATEINAME:=ENTWURFSNAME)(' -P ';
33.                CALL BEFEHLSLESEN('VORUEBERSETZEN? :',OUTPUT);
34.                CALL BEFEHLSLESEN('REFERENZLISTING? :',  REFERENZLISTING);
35.                IF REFERENZLISTING THEN
36.                OPEN DRUCKER;
37.                    PUT 'REFERENZLISTE VON ',ENTWURFSDATEINAME TO DRUCKER BY
38.                      SKIP,A,A,SKIP;
39.                FIN;
40.                OPEN ENTWURF BY IDF(ENTWURFSDATEINAME),OLD; /*
41. #232
42.    #232A        BAUE SORTIERTES ADRESSBUCH AUF
43.    #232B        UNTERSUCHE ADRESSBUCH AUF WIDERSPRUECHE ODER UNVOLLSTAENDIGKEITEN
44.               UND DRUCKE FEHLER AUS
45. #232A
46.    #232A1       BEREITE AUFBAUEN VOR
47.            */ REPEAT /*
48.    #232A2       UNTERSUCHE ZEILE FUER ZEILE UND BAUE ADRESSBUCH AUF
49.    #232A3       BIS DATEIENDE ERREICHT
50.    #232A4       BIS UEBERLAUFFEHLER PASSIERT
51.    #232A5       REPEATEND
52. #232A1
53.            */
54.                LETZTEREINTRAG:=0;
55.                SCHLUESSEL(1):=' ';
56.                FOR I TO ADRESSBUCHLAENGE  REPEAT
57.                  ZEILENNOTIZ(I,1):=0;
58.                  ZEILENNOTIZ(I,2):=0;
59.                END;
60.                         UEBERLAUFENDZEIGER:=0; /*
61. #232A2
62.    #232A21      ERHOEHE ZEILENZAEHLER, LIES ZEILE EIN
63.    #232A22      UNTERSUCHE ZEILE AUF UEBERSCHRIFTANFANG ODER KAPITELANFANG
64.            %TA24 DRUCKE ZWISCHENERGEBNISSE
65.    #232A23      IF UEBERSCHRIFTANFANG ODER KAPITELANFANG
66.    #232A24      THEN MACHE VERMERK IM ADRESSBUCH; FIN
67.    #232A25      MELDE FEHLER,FALLS ZUVIEL ODER ZUWENIG KOMMENTARANFAENGE
68.
```

Bild 1. Anfang des Programm-Entwurfs für den Preprozessor. Pseudo-Code- und PEARL-Anweisungen sind auf allen Verfeinerungs-Stufen gemischt; letztere stehen zwischen "umgekehrten" Kommentarzeichen. Die links vor den Pseudo-Code-Anweisungen stehenden Kennziffern assoziieren jene mit den zugehörigen Verfeinerungen; dort sind die Kennziffern nicht eingerückt.

```
559. #23          */ MAIN:TASK GLOBAL; /*
560.   #231       EROEFFNE DATIONS UND FUEHRE STEUERDIALOG
561. #231         */
562.               OPEN TASTEN;
563.               OPEN SCHIRM;
564.                PUT 'WELCHE DATEI SOLL BEARBEITET WERDEN?' TO SCHIRM;
565.               GET ENTWURFSNAME FROM TASTEN;
566.               ENTWURFSDATEINAME:=ENTWURFSNAME)(' ')('-TX';
567.               ZIELDATEINAME:=ENTWURFSNAME)(' -P ';
568.               CALL BEFEHLSLESEN('VORUEBERSETZEN? :',OUTPUT);
569.               CALL BEFEHLSLESEN('REFERENZLISTING? :',  REFERENZLISTING);
570.               IF REFERENZLISTING THEN
571.               OPEN DRUCKER;
572.                  PUT 'REFERENZLISTE VON ',ENTWURFSDATEINAME TO DRUCKER BY
573.                    SKIP,A,A,SKIP;
574.               FIN;
575.               OPEN ENTWURF BY IDF(ENTWURFSDATEINAME),OLD; /*
576.   #232       LIES ENTWURF AUS STREAM-DATION UND BAUE ADRESSBUCH AUF.
577.                MELDE DABEI FEHLER
578. #232
579.   #232A         BAUE SORTIERTES ADRESSBUCH AUF
580. #232A
581.   #232A1        BEREITE AUFBAUEN VOR
582. #232A1
583.           */
584.               LETZTEREINTRAG:=0;
585.               SCHLUESSEL(1):=' ';
586.               FOR I TO ADRESSBUCHLAENGE  REPEAT
587.                 ZEILENNOTIZ(I,1):=0;
588.                 ZEILENNOTIZ(I,2):=0;
589.               END;
590.                          UEBERLAUFENDZEIGER:=0; /*
591.           */ REPEAT /*
592.   #232A2        UNTERSUCHE ZEILE FUER ZEILE UND BAUE ADRESSBUCH AUF
593. #232A2
594.   #232A21       ERHOEHE ZEILENZAEHLER, LIES ZEILE EIN
595. #232A21         */
596.                 CALL EINLESEN; /*
597.
598.   #232A22       UNTERSUCHE ZEILE AUF UEBERSCHRIFTANFANG ODER KAPITELANFANG
599.           %TA24 DRUCKE ZWISCHENERGEBNISSE
600. #232A22         */ CALL UNTERSUCHUNG ; /*
601.
602.   #232A23       IF UEBERSCHRIFTANFANG ODER KAPITELANFANG
603. #232A23          */ IF UEBERSCHRIFTANFANG OR KAPITELANFANG /*
604.
605.   #232A24       THEN MACHE VERMERK IM ADRESSBUCH; FIN
606. #232A24         */ THEN /*
607.   #232A240      DIE KOMMENTARZAEHLUNG MUSSTE VOR EINLESEN DER ZEILE 0 GEWESEN
608.             SEIN. DANACH MUSS SIE 0 ODER 1 SEIN.
609. #232A240        */ IF KOMMENTARZAEHLUNG/=1 THEN
610.                  IF KOMMENTARZAEHLUNG<1 THEN
611.                      CALL FEHLERMELDUNG('ZUWENIG KOMMENTARANFAENGE VOR ZEILE',
612.                                                 ZEILENR);
613.                    ELSE
614.                      CALL FEHLERMELDUNG('ZUVIELE KOMMENTARANFAENGE VOR ZEILE',
615.                                             ZEILENR);
616.                  FIN;
617.                KOMMENTARZAEHLUNG:=1;
618.             FIN;
619.             /*
620.
621.   #232A241      IF UEBERSCHRIFTANFANG
622. #232A241        */ IF UEBERSCHRIFTANFANG /*
623.
624.   #232A242        THEN BEHANDLE UEBERSCHRIFT;FIN
625. #232A242
626.             */ THEN /*
```

Bild 2. Teil des aus dem Programm-Entwurf erzeugten Programmes. Pseudo-Code und Kennziffern bilden die Kommentare. Zeile 599 bezieht sich auf ein nur zum Test benötigtes Programmstück, das jetzt vom Preprozessor hinter dem MODEND eingeordnet worden ist; Rückänderung des % in # würde es in das Programm zurückholen.

In der Verfeinerungs-Hierarchie der Task erfolgt der Abstieg über die Kennziffern 23-231, dann die Rückkehr nach 232, erneuter Abstieg 232-232A-232A1, Rückkehr nach 232A2, Abstieg 232A2-232A21, Rückkehr nach 232A22, usw.. Einen besonderen Hinweis verdient Zeile 599 aus B i l d 2. Dort ist ein Programmstück aus der Testfassung des Programmes dadurch entfernt worden, daß in der Kennziffer das führende # in ein % geändert wurde. Würde diese Änderung im Entwurf rückgängig gemacht, würde der Preprozessor die Zeile als Pseudocode-Anweisung erkennen und die zugehörige Verfeinerung hinter ihr einordnen, während sie jetzt hinter dem MODEND stehen bleibt.

Erfahrungen

Bisher wurden der Preprozesser (ca. 1500 Zeilen) und ein Programm zur Steuerung einer Modelleisenbahn (10 Moduln mit insgesamt 20 Tasks und ca. 3000 Zeilen) mit dieser Methode entwickelt, die bei dem letzten Projekt auch für die Abfassung der Spezifikationen verwendet wurde. Dabei wurden folgende Erfahrungen gemacht.

* Die Programm-Entwicklung geht erstaunlich rasch; bei schwierigen Programmstücken und bei Änderungen in der Wartungsphase dürfte die Zeitersparnis gegenüber Entwerfen mit Struktogrammen über 50 % betragen.

* Unterbrechungen bei der Arbeit wirken sich bei weitem nicht so störend aus, wie sonst beim Programmieren, weil der Gedankenfluß bis zum Zeitpunkt der Unterbrechung lückenlos dokumentiert ist.

* Schwere Programmfehler sind fast nur da, wo Programmteile nicht mit schrittweiser Verfeinerung, sondern sofort in der Programmiersprache entwickelt wurden.

* Die Programm-Entwürfe dokumentieren die Denkprozesse des Entwicklers. Das hilft außerordentlich bei der Suche nach logischen Fehlern. Sie werden häufig bei nochmaligem Durchlesen gefunden.

* Man kann alle Teile des Programmes - Parallelarbeit, Daten - übersichtlich und gut gegliedert beschreiben; B i l d 3 zeigt einen Ausschnitt aus der Datenbeschreibung des Preprozessors, B i l d 4 einen Teil der Tasking-Beschreibung der Bahnsteuerung.

* Die Programme sind auch nach längerer Zeit noch leicht zu ändern, da sie sofort wieder verständlich sind.

```
377. #21
378.    #211      DATIONS
379.    #212      PROBLEMDATEN
380.    #213      HILFSDATEN
381.
382. #212
383.    #2121     KONSTANTEN
384.    #2122     ADRESSBUCH
385.    #2123     STACK
386.
387.
388. #2122
389.    #21221       SCHLUESSEL-NOTIERUNGEN
390.    #21222       NOTIERUNG DER PROGRAMMZEILEN, IN DENEN SCHLUESSEL VORKOMMEN
391.                 (JEWEILS GLEICHER INDEX WIE 21221)
392.    #21223       VERKETTUNG DER SCHLUESSEL IN ALPHABETISCH AUFSTEIGENDER REIHEN-
393.                  FOLGE
394.    #21224       GEMEINSAME ZEIGER, HILFSDATEN
395.    #21225       NOTIERUNG DER ZEILEN-EIGENSCHAFTEN, UM 2. DURCHGANG ZU
396.                 BESCHLEUNIGEN
397.
398.
399. #21221       */ DCL SCHLUESSEL (300) CHAR(10),
400.                     SCHLUESSELLAENGE INV FIXED INIT(10)# /*
401.
402. #21222
403.    #212221   JEDER SCHLUESSEL KOMMT I. A. JE EINMAL VOR EINER UEBERSCHRIFT
404.              UND VOR EINEM KAPITELANFANG VOR. EINGETRAGEN WERDEN DIE JEWEILIGEN
405.              ZEILENNUMMERN.
406.    #212222   UEBERLAUFTEIL FUER EINTRAG MEHR ALS 2-MAL VORKOMMENDER SCHLUESSEL;
407.              IN DIESEM FALL WIRD IN 212221 EIN VERWEIS AUF DEN UEBERLAUFTEIL
408.               EINGETRAGEN (NEGATIVES VORZEICHEN). DER UEBERLAUTEIL ENTHAELT
409.              U.U. VERKETTUNGEN ZU WEITEREN EINTRAGUNGEN. KETTENENDE IST 0
```

Bild 3. Ausschnitt aus dem Entwurf des Preprozessors: Beschreibung des Adressbuches.

```
181. #U3      FUER JEDEN FAHRWEG GIBT ES JE ZWEI TASKS, DIE DAFUER SORGEN, DASS
182.          DER FAHRWEG FORTGESETZT WIRD, BZW., DASS ER TEILWEISE WIEDER
183.          FREIGEGEBEN WIRD.      EINE TASK, DIE IHREN FAHRWEG NICHT FORTSETZEN
184.          KANN, WARTET AUF EINEM REQUEST IN DER PROZEDUR AUFVERSUCHWARTEN
185.               (RELEASE IN PROZEDUR VERSUCHMACHENLASSEN, TEIL DER FREIGABE).
186.          FUER JEDEN SENSOR EXISTIERT EINE TASK, IN DER AUF DIE SENSORMELDUNG
187.          GEWARTET UND DIE MELDUNG BEARBEITET WIRD.
188.
189.          DA DIE BEARBEITUNG EINER SENSORMELDUNG UND DIE FAHRWEGFORTSETZUNG
190.          BEIDE DEN ZUGORT DES FAHRWEGES NEU EINTRAGEN, WERDEN SIE VORSICHTS-
191.          HALBER DURCH DEN SEMAPHOR KRITISCH1 BZW. KRITISCH2 SEQUENTIALISIERT.
192.
193.          AUSSERDEM EXISITERT FUER JEDEN FAHRWEG EINE UEBERWACHUNGSTASK,
194.          DIE EINE FEHLERMELDUNG VERURSACHT, WENN LAENGER ALS 30 SEC
195.          AUF EINE FAELLIGE SENSORMELDUNG GEWARTET WORDEN IST.
196.
197.          ALTERNATIV KOENNTE FUER JEDE DER FUNKTIONEN ANFORDERN, WEICHENSTELLEN
198.          USW. EINE EIGENE TASK GESCHRIEBEN WERDEN. DIESE TASKING-STRUKTUR
199.          WUERDE JEDOCH FUER AUF- UND ABBAU DER FAHRWEGE INSGESAMT 6 TASKS
200.          ERFORDERN.
201.
202.    #U31  FORTSETZUNGSTASKS FUER BEIDE FAHRWEGE
203.    #U32  FREIGABETASKS FUER BEIDE FAHRWEGE
204.    #U33  TASKS ZUR INTERRUPT-BEARBEITUNG
205.    #U34  UEBERWACHUNGSTASKS FUER BEIDE FAHRWEGE
206.
207. #U31
208.             */  WEG1:TASK PRIO 2;
209.                 CALL FORTSETZUNGSARBEIT(1);
210.         /*
211.    %UT61    TEST-SIMULATION
212.         */
213.               END;
214.
215.               WEG2:TASK PRIO 2;
216.                 CALL FORTSETZUNGSARBEIT(2);
217.         /*
218.    %UT62    TEST-SIMULATION
219.         */
220.               END; /*
221.    #U311 GEMEINSAME BEARBEITUNGSPROZEDUR FORTSETZUNGARBEIT
```

Bild 4. Ausschnitt aus dem Programmentwurf für die Steuerung einer Modellbahn: Beschreibung des Tasking.

- Nur für den Test benötigte Programmteile lassen sich fast vollautomatisch aus den Programmen entfernen und später bei Wartungsarbeiten wieder in die Programme einordnen.

- Der Rückwärts-Bezug vom kompilierbaren Programm zum Programm-Entwurf ist wegen der Kennziffern sehr leicht.

- Die Entwürfe konnten so geschrieben werden, daß kritische Abschnitte aus einer einzigen Pseudocode-Anweisung bestanden. Dadurch ist es möglich, Operationen auf denselben Semaphor eng benachbart niederzuschreiben. Mögliche Verklemmungen werden so leicht erkannt.

- Das Top-down-Verweis-System ist zur übersichtlichen Gliederung aller verbalen Beschreibungen, z.B. auch für Grob- und Fein-Spezifikationen, gut geeignet; der Preprozessor ermöglicht deren Prüfung auf formale Vollständigkeit.

- Fein-Spezifikationen und Programm-Entwurf lassen sich zu einem einzigen Dokument integrieren; dadurch läßt sich das Nicht-Übereinstimmen nach dem Programm-Test leichter vermeiden.

- Last not least: der Lernaufwand für die Formalien der Methode ist so gering, daß sich ihre Einführung auf freiwilliger Basis durchführen läßt.

Diesen Vorteilen stehen nur zwei Nachteile gegenüber:

- Nach Programmänderungen ist zusätzlich zur Kompilation usw. ein Preprozessor-Lauf erforderlich.

- Möglichkeiten zu lokaler Optimierung sind im Entwurfs-Dokument schwer zu erkennen, weil nacheinander auszuführende PEARL-Anweisungen in diesem oft weit voneinander entfernt stehen.

Schlußbemerkungen

Es dürfte wesentlich zur Schnelligkeit und zum Erfolg kreativer Arbeit beitragen, wenn kein Zwang zur Ausarbeitung unwichtiger Details oder zu umständlichen Darstellungsmethoden besteht. Insofern ist es nicht

überraschend, daß eine Papier- und -Bleistift-Methode, bei der zunächst nur das Wichtigste möglichst schnell notiert zu werden braucht, gute Ergebnisse liefert.

In den Programmentwürfen sind dieselben Sachverhalte mehrfach beschrieben: in verschiedenen Verfeinerungsstufen des Pseudocodes, zuletzt in der Programmiersprache. Angesichts dieser Redundanzen ist es wiederum kein Wunder, daß die Programme leichter verstanden und Fehler leichter gefunden werden.

Anhang: Technische Daten der Preprozessor-Implementation

Der Preprozessor wurde unter Verwendung von Bais-PEARL entwickelt. Auf der Krupp-Atlas EPR-1300 belegt er insgesamt ca. 8 K 16-Bit-Worte, von denen knapp 4 K für ein Adreßbuch benötigt werden, das durch folgende (leicht änderbare) technische Daten bestimmt wird:

Maximale Zeilenlänge des Inputs:	80 Zeichen
Maximale Zeilenzahl des Inputs:	3000 Zeilen
Maximale Kennzifferlänge:	10 Zeichen
Maximale Anzahl d. Kennziffern:	300
Maximale Verfeinerungstiefe:	20

Er benötigte auf der EPR-1300 zur Erzeugung eines kompilierbaren Programmes aus seinem eigenen Entwurf ca. 6 Minuten. Diese Zeit konnte durch Übergang von Einzelzeichen-Verarbeitung auf Substrings auf die Hälfte gesenkt werden. Die Bearbeitungszeit hängt wesentlich vom E/A-System und vom verwendeten Massenspeicher ab, sie dürfte auch auf anderen Systemen etwa in der Größenordnung der für eine Kompilation benötigten Zeit liegen.
Von den 1500 Zeilen des Preprozessors enthalten ca. 300 die hierarchisch gegliederte Spezifikation, ca. 180 Zeilen Pseudocode und ca. 600 Zeilen PEARL-Code. Eine für einen Bootstrap zusammengestrichene PEARL-Version würde ca. 400 Zeilen lang sein.
Komplette Listings von Preprozessor-Entwurf und kompilierbarem Programm werden auf Wunsch zugeschickt.

Literatur

/1/ META Sprachbeschreibung: Krupp-Atlas-Elektronik, Bremen

/2/ Rainer Hahn, Peter Stock: ELAN Handbuch; Akademische Verlagsgesellschaft Wiesbaden (1979)

/3/ G.Hommel (Hrsg.): Vergleich verschiedener Spezifikationsverfahren am Beispiel einer Paketverteilanlage; Kernforschungszentrum Karlsruhe GmbH, KfK-PDV 186 (1980)

Kurzmitteilungen

NEUE VERTRIEBSSTRATEGIE FÜR DEC-PEARL

DIGITAL EQUIPMENT hat seit 1977 die Entwicklung von PEARL-Compiler- und Laufzeitsystemen mit großem Aufwand betrieben. Als erstes Ergebnis dieser Aktivitäten wurde ein PEARL-Compiliersystem für die PDP-11 Rechnerfamilie Mitte 1980 auf den Markt gebracht und aktiv vertrieben. Dieses System ist mittlerweile auch bei zahlreichen Anwendern im Einsatz.

Nahtlos daran anschließend wurde die Entwicklung eines PEARL-Systems für die 32 Bit VAX-11 Rechnerfamilie begonnen. Diese Arbeiten sind mittlerweile abgeschlossen, VAX-PEARL hat den Feldtest erfolgreich bestanden und wird in Kürze offiziell angekündigt und zum Vertrieb freigegeben werden.

Mit Wirkung vom 9.Juni 1981 hat die DIGITAL EQUIPMENT GmbH nun ihre Vertriebsstrategie für PEARL geändert. Während bisher PEARL-Software mit eingebündelten Dienstleistungen angeboten wurde (A-Lizenz), werden PEARL-Software-Lizenzen künftig in der generellen Preisliste ohne dieses Dienstleistungspaket angeboten (C-Lizenz). Bestehende Gewährleistungsverpflichtungen werden davon nicht berührt.

Der Grund für diese Änderungen in der Vertriebspolitik ist die Tatsache,daß die von DIGITAL EQUIPMENT veranschlagten Verkaufszahlen für PEARL-Compiler bisher nicht erreicht wurden.

Welche Konsequenzen ergeben sich nun für die Anwender von PEARL aus dieser Änderung der Vertriebsstrategie?

Zunächst gilt, daß PEARL von DIGITAL EQUIPMENT auch weiterhin für die Rechnerfamilien PDP-11 und VAX-11 angeboten wird. Dienstleistungen können auch weiterhin bezogen werden, falls dies vom Anwender im Einzelfall mit DIGITAL EQUIPMENT vereinbart wird. Nähere Auskünfte dazu erteilt DIGITAL EQUIPMENT München (Herr E. Mezera), Tel. 089/9250-2290.

Längerfristig kann eine Steigerung der Nachfrage am Markt DIGITAL EQUIPMENT veranlassen, ihre Vertriebsstrategie anzupassen und A-Lizenzen wieder generell in der Preisliste anzubieten.

INTERNE PRÄSENTATION DES DORNIER PEARL SYSTEMS

Am 9.7.1981 fand vor einem internen Anwenderkreis eine Präsentation des DORNIER-PEARL-Systems mit großem Erfolg statt. Damit steht das System auch externen Anwendern zur Verfügung.

Es besteht aus:

- Compileroberteil
- Codegenerator
- Assembler
- Modulares Betriebssystem
- Pre-Linker
- Linking-Loader
- Testhilfen

Das PEARL-System ist ablauffähig auf einer PDP 11/70 als Gastrechner. Die Zielsysteme sind INTEL 8086 (DORNIER-MUDAS 426-Prozessor) und DORNIER-MUDAS 432-Prozessor. Weitere Einzelheiten können auf Anfrage mitgeteilt werden.

DORNIER SYSTEM GMBH, Abt. VAP
Postfach 13 60, 7990 Friedrichshafen

NACHTRAG ZUR PEARL RUNDSCHAU HEFT 2 BAND 2

Der Beitrag "Höhere Programmiersprachen im Programmentwicklungsprozeß" von P. Elzer ist ein Nachdruck aus der Dornier Post 3/80

PEARL-Kurse

21. - 25.9.1981 Systematisches Programmieren mit PEARL

Einführungskurs für Programmierer mit praktischen Übungen am Rechner (die Kurse sind bereits vom IRT München ausgebucht).

Leitung: Prof. Dr. Pflügel, Furtwangen

Veranstalter: PEARL-Verein e.V. in Zusammenarbeit mit DORNIER-System GmbH.

Ort: München

14. - 18.9.1981 Systematisches Programmieren mit PEARL

Der Kurs gibt an Hand praxisnaher Beispiele eine Einführung in das Programmieren mit PEARL. Für praktische Übungen steht ein PDP-11 PEARL-System zur Verfügung.

Leitung: Prof.Dr. Welfonder Stuttgart

Veranstalter: PEARL-Verein e.V. in Zusammenarbeit mit dem VDI-Bildungswerk

Ort: Stuttgart

19.-30.10.1981 und 1.-12.3.1982 PEARL

Der Kurs bietet eine Einführung in PEARL für Siemens Rechner.
Der Kurs wird bei Bedarf auch in Englisch angeboten.

Veranstalter: Siemens AG
Schule für Prozeßrechnertechnik
Herr Heim
Östl.Rheinbrückenstr.50
7500 Karlsruhe 21
Tel. 0721/5954125

Ort: Karlsruhe

Veranstaltungen und Termine

2.-3.9.1981	Informationstagung "Messen, Prüfen und Steuern in Walzwerken" Luxemburg Kommission der EG, Rue de la Loi 200, B-1049 Brüssel
15.-17.9.1981	Kongress "Kraftwerke 1981" Den Haag, Holland VGB, Klinkestr. 27-31 D-4300 Essen
28.-30.9.1981	PEARL-Informationsveranstaltung Cernobbio/Mailand - Italien
1.-2.10.1981	"PEARL in Ausbildung, Lehre und Schulung" Erlangen PEARL-Verein, Düsseldorf
6.-7.10.1981	BIAS '81 - 17 th International Conference on Control of Industrial Processes Mailand, Italien FAST, Piazza Rodolfo Morandi 2, I-2012 Milano
7.-8.10.1981	Konferenz "Leittechnik in Wärmekraftwerken" Essen
6.-8.10.1981	3 rd Data Processing Conference Zürich UNIPEDE, 39, avenue de Friedland F-75008 Paris
19.-23.10.1981	GI-Jahrestagung München darin: Fachgespräche "Systematischer Entwurf von PDV-Systemen"
19.-23.10.1981	SYSTEMS '81 Internationaler Kongress und Internationale Fachmesse München
16.-17.11.1981	Automatisierte Materialfluß-Systeme München VDI-Gesellschaft Materialfluß und Fördertechnik
7.-8.12.1981	PEARL-Tagung '81 "PEARL-Praxis und Zukunft" Düsseldorf
7.12.1981	Mitgliederversammlung des PEARL-Vereins Düsseldorf
9.-10.3.1982	Programmiersprachen und Programmentwicklung GI-Fachtagung München
24.5.-28.5.1982	IMEKO 82 Berlin