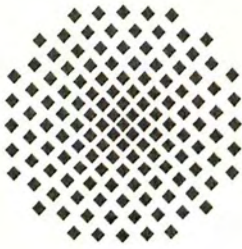


# Universität Stuttgart
# Fakultät Informatik

Institut für Informatik
Breitwiesenstraße 20-22
D-7000 Stuttgart 80

**Verbesserung der Software-Qualität durch Verifikation der Korrektheit der Implementierung im Projekt PROSOFT**

**Ein Vorschlag zur Formalisierung des Modells**

Daltro José Nunes

Report Nr. 1993/2

SABi

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

# Verbesserung der Software-Qualität durch Verifikation der Korrektheit der Implementierung im Projekt PROSOFT: Ein Vorschlag zur Formalisierung des Modells

Daltro Jose Nunes
Universidade Federal do Rio Grande do Sul
Departamento de Informatica Aplicada
Instituto de Informatica
Caixa Postal 15064
91501-970 Porto Alegre- RS
Brasilien

Inhalt:

1. Einführung
2. Ziele des Projekts PROSOFT
3. Neues PROSOFT
4. Konkurrierendes PROSOFT
5. Verteiltes PROSOFT
6. Das Problem des graphischen Darstellung
7. Schlußfolgerungen
8. Danksagung
9. Literatur

Diese Forschungsaktivität wurde durch folgende Forschungsgesellschaften gefördet: Alexander von Humboldt Stiftung, Conselho Nacional de Desenvolvimento Cientifico e Tecnologico-CNPq, Gesellschaft für Matematik und Datenverarbeitung-GMD und Coordenacao de Aperfeicoamento de Pessoal de Nivel Superior-CAPES.

## 1.Einführung

Das Forschungsvorhaben PROSOFT wurde im Jahre 1987 an der Bundesuniversität Rio Grade do Sul in Porto Alegre, Brasilien, am dortigen Institut für Informatik begonnen. Es setzt inhaltlich die Stuttgarter Dissertation von Daltro J. Nunes [Nun85] fort. PROSOFT stellt eine prototypische Software-Entwicklungsumgebung dar, die auf der "data driven strategy" beruht. Diese Strategie ist besonders dann geeignet, wenn das zu entwickelnde Softwareprodukt komplexe Objektstrukturen bearbeitet. Diese Vorgehensweise unterscheidet sich daher von einer eher traditionellen "process driven strategy".

Die im Jahre 1987 festgelegten Ziele des Projekts wurden zwischenzeitlich erfolgreich erreicht. Beschreibungen der bisher erzielten Ergebnisse finden sich im zahlreichen Publikationen. Viele Forschungsstipendiaten und Studierende der Bundesuniversität in Porto Alegre hatten Gelegenheit, am Vorhaben PROSOFT mitzuarbeiten und ihre Tätigkeit mit einer Diplomarbeit und/oder einer M.Sc.-Arbeit erfolgreich abzuschließen.

Die Arbeiten des Vorhabens wurden seit Projektbeginn durch das Institut für Informatik der Universität Stuttgart unterstützt, und zwar durch die Abteilung Dialogsysteme (Leiter: Prof. Dr. R. Gunzenhäuser) und die Abteilung Betriebssoftware (Leiter: Prof. Dr. K. Lagally).

Im Rahmen dieser Zusammenarbeit, die von der Alexander von Humboldt-Stiftung, vom Conselho National de Desenvolvimento Cientifico e Tecnologico-CNPq und von der Gesellschaft für Mathematik und Datenverarbeitung-GMD gefördert wurde, fanden mehrere Arbeitsaufenthalte statt, u.a. von Herrn Heribert Schlebbe in Porto Alegre während dreier Monate im Jahre 1990 und von Herrn Paulo Schmitt do Carmo in Stuttgart von August 1991 bis Februar 1992.

Professor Daltro J. Nunes, der Leiter des PROSOFT-Projekts, konnte für einen Forschungsaufenthalt von Dezember 1992 bis Februar 1993 an der Universität Stuttgart gewonnen werden. Dieser Aufenthalt diente dazu, um über die Ergebnisse der bisherigen gemeinsamen Forschungsarbeiten zu berichten und über die weitere inhaltliche Entwicklung der Projektarbeiten zu forschen.

Es war insbesondere erforderlich, neue wissenschaftliche Kontakte zu knüpfen. Dazu dienten Vorträge von Professor Nunes an mehreren deutschen Universitäten und bei der GMD sowie zahlreiche Fachdiskussionen mit deutschen Professoren. Auf Grund dieser Kontakte wurden neue Kooperationen konzipiert.

Ein wesentliches Resultat dieses Forschungaufenthalts besteht darin, daß im folgenden neue Ideen zum weiteren Ausbau des Projekts dargestellt werden können. Daraus werden neue Konzepte hergeleitet, um Wege zur Umsetzung und Integration dieser Ideen im Projekt PROSOFT zu finden.

## 2. Ziele des Projekts PROSOFT

Die PROSOFT Entwicklungsumgebung enthält eine Sammlung von Werkzeugen, die sowohl die Herstellung von Anwenderprogrammen unterstützen als auch die Entwicklung von neuen Werkzeugen für die Software-Umgebung PROSOFT; beide Vorgehensweisen basieren auf derselben Strategie. Eine gute Beschreibung des Projekts PROSOFT befindet sich in [Nun92].

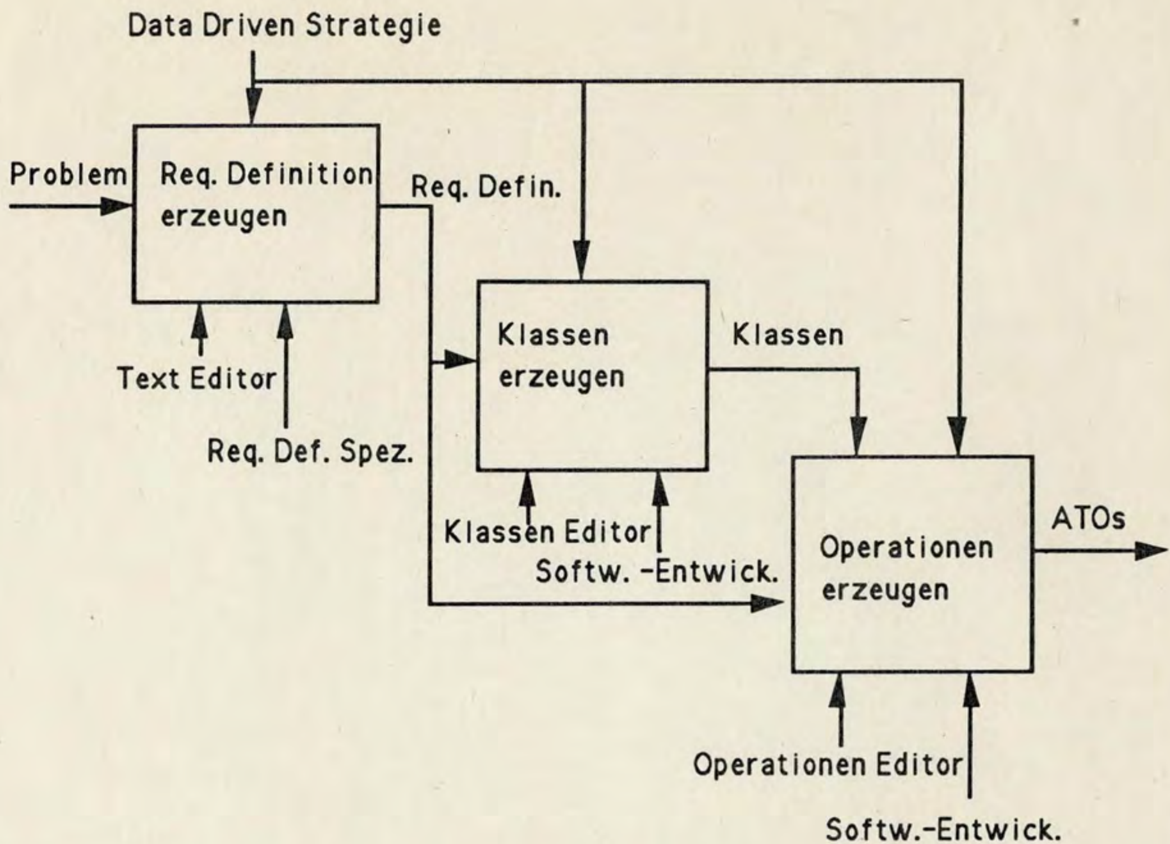
Als erster Schritt werden hierbei die Datenstrukturen - hier Klassen genannt - definiert, als zweiter Schritt die Operationen, die auf diesen Datenstrukturen wirken. In der PROSOFT-Terminologie werden diese "Schnittstellenoperationen" genannt. Die Klassen und die Menge ihrer Schnittstellenoperationen heißen dann ATO (portugiesisches Akronym für Objekt-Manipulations-Umgebung).

PROSOFT enthält mehrere Werkzeuge, die schon jetzt die Erzeugung von Klassen und von ATOs unterstützen. Ein Beispiel dafür ist ein Hypertext-Werkzeug, das die Wiederverwendbarkeit von Klassen und von Operationen unterstützt.

In PROSOFT wurde ferner ein Struktureditor entwickelt, der auf der "data driven strategy" beruht und eine semi-formale Anforderungsdefinition erzeugt, die auch auf dieser Strategie beruht. Über die detaillierte Gestaltung dieser Anforderungsdefinition wird noch gearbeitet.

Die Anforderungsdefinition wird von einem Spezialisten erstellt, der die Aufgabenstellung sehr gut versteht und sehr genaue Vorstellungen davon hat, "was" ein Computersystem tun soll, der jedoch nicht genau weiß, "wie" der Computer dies tut. Durch seine Anforderungsdefinition ist dieser Experte in der Lage, die Lösung des Problems so gut zu beschreiben, daß ein Informatiker dann die Lösung prototypisch entwickeln und implementieren kann.

Bei der Erzeugung der Anforderungsdefinition läßt sich beobachten, daß ein Experte für Anforderungsdefinitionen die Klassen und Operationen semi-formal definieren wird. Die semi-formalen Klassen werden nicht notwendigerweise aus den primitiven oder parametrisierten Klassen konstruiert. Der Softwareentwickler hat also doppelte Arbeit zu leisten: Erstens soll er mit Unterstützung eines ATO-Editors die Anforderungsdefinition in einen formalen Ausdruck (ATO) umsetzen und zweitens soll er aus der Menge der ihm zur Verfügung stehenden primitiven und parametrisierten Klassen diejenigen auswählen, mit deren Hilfe er die formale Klasse definiert. Die Korrektheit einer solchen "Implementierung" läßt sich nicht beweisen, weil die Anforderungsdefinition ein semi-formaler Ausdruck ist. Das folgende Bild zeigt auf abstrakter Ebene, wie die Software entwickelt wird.



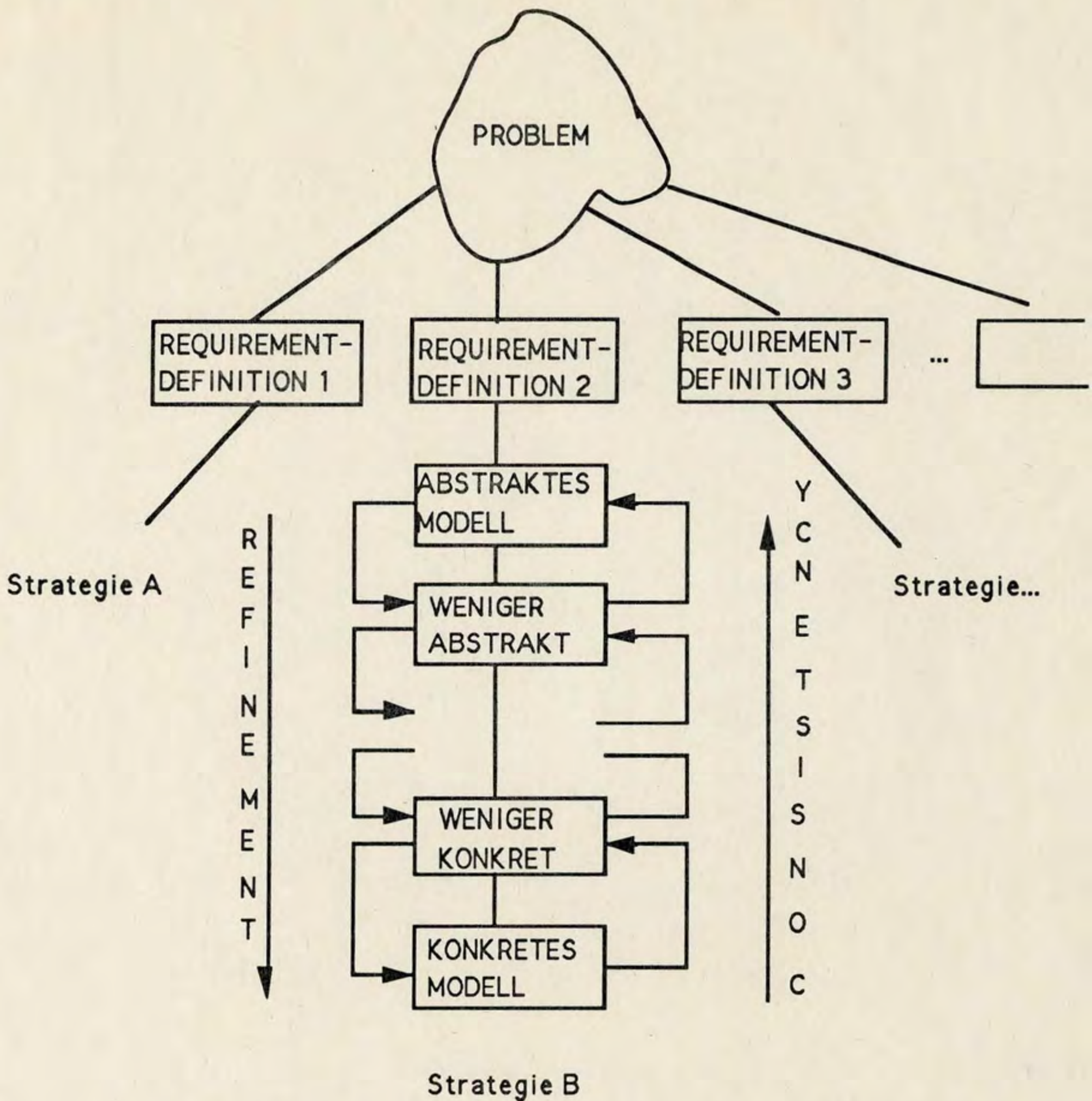

Unsere Umgebung zeigt, daß sich durch die Wiederverwendung von Klassen und Operationen sowie durch die Anwendung von mächtigen Datentypen (map, set, list usw.), die in imperativen Programmiersprachen wie Pascal, Ada usw. nicht vorhanden sind, **eine große Produktivität** erreichen läßt. Über die Qualität des Produktes hingegen, insbesondere dessen Korrektheit, lassen sich aber keine Aussagen machen.

## 3. Neues PROSOFT

Nach Meinung mancher Wissenschaftler muß die Lösung, also die semi-formale Anforderungsdefinition [Coa86], [Jon80], ein abstraktes Modell haben. Dieses sollte dann schrittweise in ein Programm umgewandelt werden. Ein solches Verfahren hat den Vorzug, daß das Programm seine Spezifikation stets erfüllt. Zur Überprüfung der Konsistenz (siehe Abbildung) benötigt man dann einen Theoremprüfer, wenn das Verfahren in der Praxis zur Anwendung kommen soll.



UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA




Das auf der "data driven" Strategie basierendes PROSOFT

(i) hilft zwar bei der Problemlösung und steigert die Produktivität erheblich, es verlangt aber gleichzeitig auch vom Softwareentwickler eine gute Kenntnis derjenigen Programmiersprache, die zur Implementierung der Operationen eingesetzt wird, und

(ii) erlaubt keine automatische Überprüfung der Korrektheit des ATOs.

Das Problem liegt darin, daß sich nicht untersuchen läßt, ob die ATOs auch mit der Spezifikation zusammenpassen. Zur Lösung dieses Problems postulieren wir, daß jedes ATO, ausgehend von seiner informellen Anforderungsdefinition, zunächst formal spezifiziert werden muß. Wir entscheiden uns für die

**algebraische Methode**, da eine strukturelle Analogie zwischen ATOs und algebraischen Spezifikationen besteht.

Im PROSOFT-Projekt werden die informellen Schnittstellenoperationen in Pascal implementiert. Auf dem Gebiet des Ingenieurwesens aber ist es üblich, Objekte durch graphische Symbole darzustellen. Bei näherer Untersuchung stellt man fest, daß die Objekte ein "Verhalten" (behaviour) repräsentieren sollen. Ein solches Verhalten läßt sich formal beschreiben: Wir sagen, daß die Objekte eine "Antwort" auf einen gewissen "Stimulus" geben. Dieser Stimulus wird durch ein Objekt hervorgerufen, und die Antwort ist dann Stimulus für ein anderes Objekt. So kommunizieren Objekte miteinander. Programme einer imperativen Programmiersprache, wie z.B. Pascal, Ada, C, lassen sich zwar formal beschreiben, es wird aber nicht gezeigt, wie sie miteinander kommunizieren. Da Programme in der Regel textuell dargestellt werden, ist ihr Verhalten oft nur sehr schwer zu verstehen. Noch schlimmer ist, daß eine formale Beschreibung des Verhaltens eines Programms in der Praxis kaum etwas taugt. Hier muß Abhilfe geschaffen werden. Ein Vorschlag dafür wäre, daß Programme

**graphisch dargestellt werden,**
**miteinandert kommunizieren können und**
**ein bekanntes Verhalten besitzen.**

***Die Implementierung*** *dieser Programme muß dann die entsprechende algebraische Spezifikation erfüllen.*

Die CCS-Methode erfüllt die obigen Anforderungen gut und hat zudem noch den Vorzug, auf einer soliden mathematischen Grundlage zu stehen. So lassen sich die primitiven und die parametrisierten Operationen zunächst als "Agenten" realisieren.

Beispiel: Die Operation

$$a+b$$

läßt sich als Agent

$$\text{Summe} = \pi a.\delta b.\overline{\gamma}(a+b).\ \text{Summe}$$

implementieren, d.h. der Agent Summe empfängt den Wert a durch die Türe (Port) $\pi$, b durch die Türe $\delta$ und liefert a+b durch die Türe $\gamma$. Türen mit positivem Vorzeichen verbrauchen Werte und Türen mit negativen Vorzeichen liefern Werte. Ein anderes Beispiel:

| | |
|---|---|
| (i) CreateSet | $cm = \pi.\overline{\delta}\Phi.cm$ |
| (ii) Header einer Liste | $p = \pi l.\overline{\delta}header(l).p$ |
| (iii) Zerlegung eines Records, wobei N1 und N2 Feldnamen sind | $z = \pi(N1.a,N2.b).\overline{\delta}(N1.a).\overline{\gamma}(N2.b).z$ |



UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

(iv) Der Wert eines Feldes eines Records $s = \pi N2.\delta(N1.a,N2.b).\overline{\gamma}b.s$

(v) CreateRecord $r = \pi N1.\delta a.\gamma N2.\overline{\beta}(N1.a,N2.b).r$

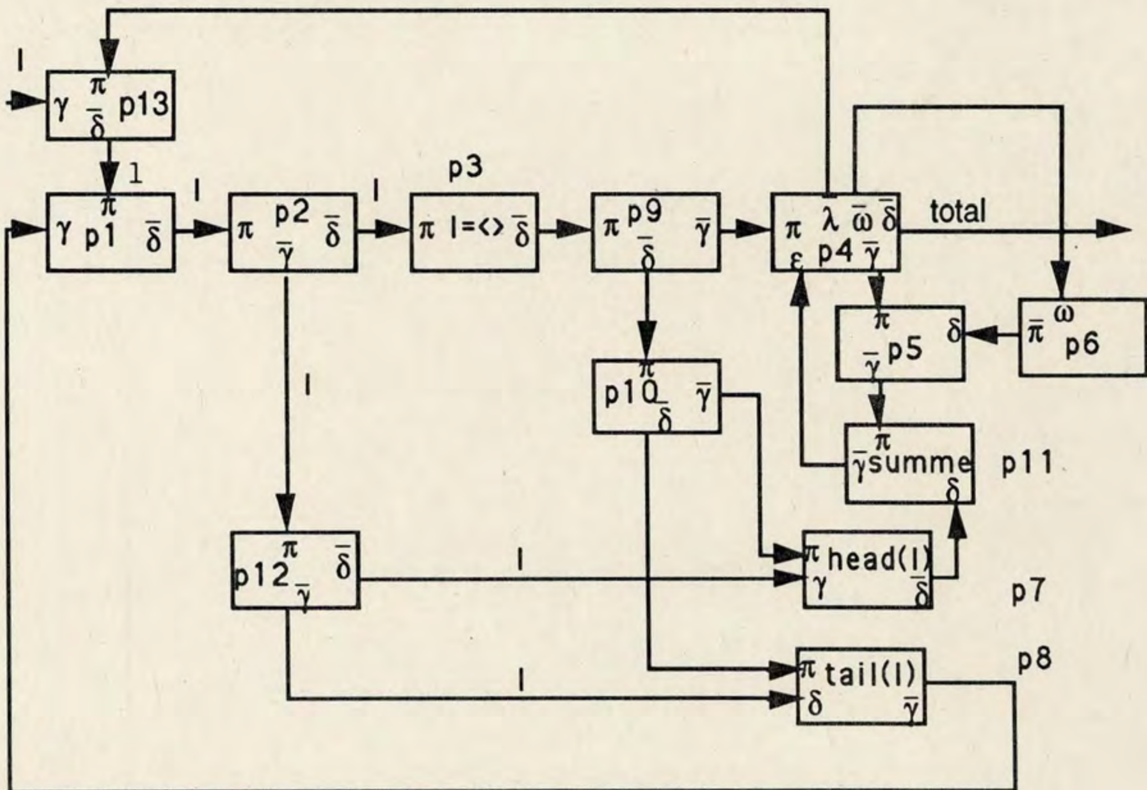
Um eine Schnittstellenoperation durch die CCS-Methode [Mil89] erzeugen zu können, ist es erforderlich, zunächst eine geeignete Programmiersprache zu definieren. Diese Programmiersprache soll graphisch sein, ihre Programme sind Datenfluß-Diagramme. Die Agenten werden als Rechtecke symbolisiert, positive und negative Türen werden durch Pfeile miteinander verbunden. Diese Programmiersprache nennen wir *Prozeßkommunikationssprache.*

Das folgende Flußdiagramm berechnet die Summe der Elemente einer Liste, deren algebraische Spezifikation lautet:

summe: Liste-Nat -> Nat

summe(l) = if l=<> then 0 else head(l) + summe(tail(l))

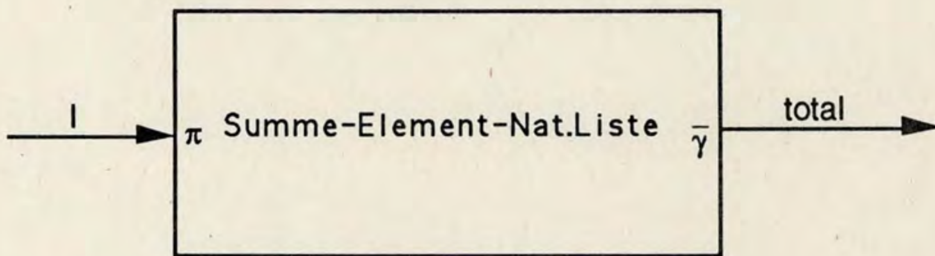
Eine mögliche Implementierung wäre dann:



Die erforderlichen Agenten für diese Aufgabe sind:

p1= $\pi$l.$\bar{\delta}$l.p1+$\gamma$l.$\bar{\delta}$l.p1
p2=$\pi$l.$\bar{\delta}$l.$\bar{\gamma}$l.p2
p3=$\pi$l.(if l=<> then $\bar{\delta}$ true else $\bar{\delta}$ false).p3
p4=$\pi$b.$\varepsilon$total.(if b then $\bar{\delta}$total.$\lambda$.$\omega$ else $\bar{\gamma}$ total).p4
p5 =$\pi$total.$\bar{\gamma}$total.p5+$\delta$zero.$\bar{\gamma}$zero.p5
p6 = $\bar{\pi}$0.$\omega$.p6
p7= $\pi$b. (if not (b) then $\bar{\delta}$ head(l) else $\bar{\delta}$0).p7
p8 = $\pi$b. $\delta$l. (if not(b) then $\bar{\gamma}$ tail(l).p8 else p8)
p9= $\pi$b.$\bar{\gamma}$b.$\bar{\delta}$b.p9
p10=p9
p11= $\pi$ total. $\delta$h.$\bar{\gamma}$ (total+h).p11
p12=p2
p13= $\gamma$l.$\bar{\delta}$l.$\pi$.p13

Das Flußdiagramm kann als Agent abstrahiert (encapsulated) und später wiederverwendet werden.




Wie oben schon erwähnt, soll der Softwareentwickler, nachdem er die Anforderungsdefinition erhalten hat, als ersten Schritt die Klassen erzeugen und dann die Operationen. Die Operationen sollen algebraisch spezifiziert werden. Ein ATO wird also zunächst algebraisch spezifiziert und seine Operationen dann durch Agenten implementiert. PROSOFT hat schon einen Text-Editor für algebraische Spezifikationen. Es fehlt noch die Algorithmen der Vollständigkeit, der Konfluenz und der Konsistenz zu implementieren, um die Korrektheit der algebraischen Spezifikation sicherzustellen.
Wenn f,

$$f: K_1 K_2 ... K_m ---> K_n$$

eine Operation ist, die implementiert werden soll, müssen die Klassen $K_i, 1=<i<=n$, schon existieren.

*Ein nicht einfach zu lösendes Problem ist es, zu beweisen, daß die Implementierung die Spezifikation erfüllt.*

Um einen Interpreter für die Ausführung der Operationen, also der Agenten, zu erzeugen, muß man die Semantik der Prozeßkommunikationssprache klar beschreiben. Ein geeigneter Vorschlag dafür wäre, den Plotkin-Formalismus [Plo81] zu verwenden, um eine operationale Semantik zu formulieren.

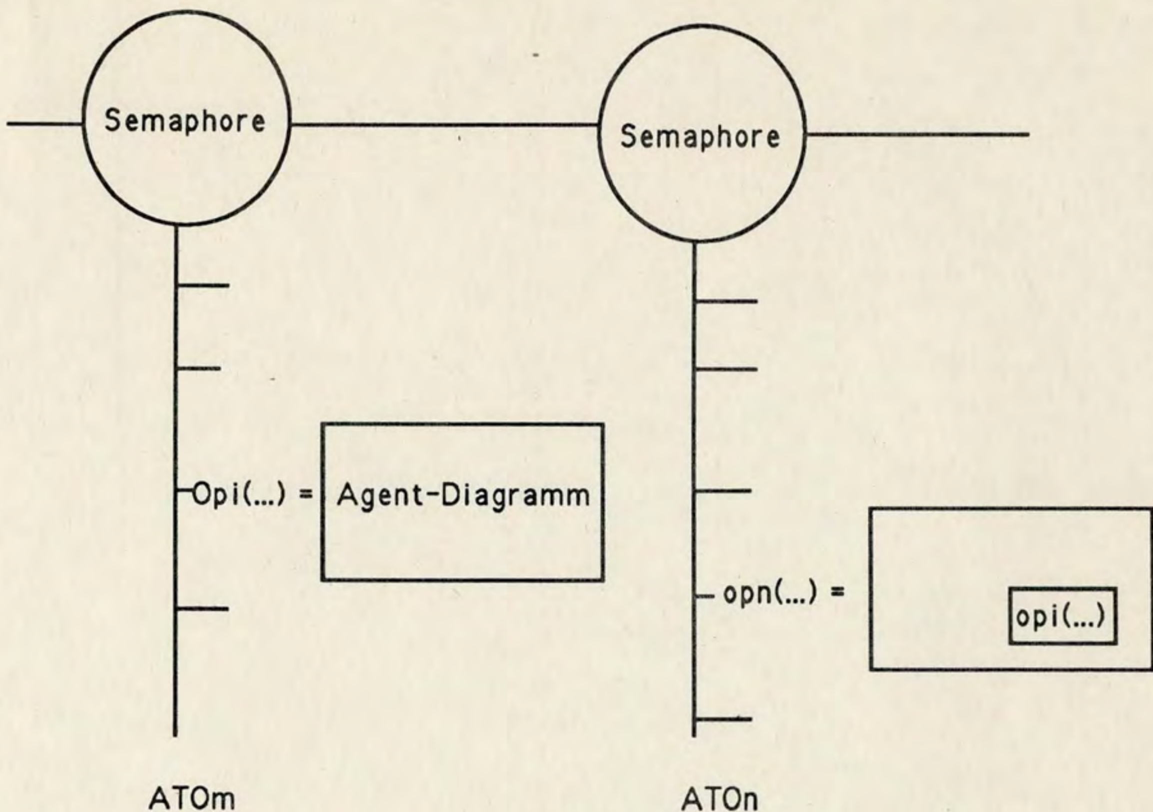
## 4. Konkurrierendes PROSOFT

Man kann die primitiven und die parametrisierten Operationen als "physikalische" Objekte betrachten, die in unbegrenzter Anzahl zur Verfügung stehen. Das ist aber bei den Schnittstellenoperationen nicht der Fall.

Wenn

- alle Schnittstellenoperation gleichzeitig laufen sollen,
- nur eine Exemplar von jeder Schnittstellenoperation vorkommt, und
- verschiedene Operationen eine bestimmte Operation gleichzeitig verwenden wollen,

dann muß man über einen Kontrollmechanismus verfügen, der jederzeit darüber entscheidet, wer die Operation benutzen darf, wenn sie gerade nicht benutzt wird. Dazu benötigt man *ein neues Modell* für die ATOs. Jedes ATO soll seinen eigenen Kontrollmechanismus besitzen. Dieser Kontrollmechanismus bekommt einen "request" von einem anderen ATO, der signalisiert, daß eine seiner Operationen angefordert wird, und bekommt auch einen request von seinen eigenen Operationen, wenn sie bei der Ausführung von Operationen anderer ATOs benötigt werden. Ein erster Vorschlag wäre es, das Konzept der Semaphore einzusetzen, wie unten gezeigt wird.




Die Abbildung zeigt, daß die Operation $op_n(...)$ des $ATO_n$ die Operation $op_i(...)$ des $ATO_n$ benötigt. Die Operation $op_i(...)$ von $ATO_n$ stellt de facto ein Aufrufmechanismus (AM) dar.

(i) Der AM wird aktiviert, wenn irgendein Parameter, der die Operation $op_i$ braucht, zur Verfügung gestellt wird.

(ii) Der AM sendet ein Signal zum Semaphor von $ATO_n$ mit der Botschaft, daß $op_i$ benötigt wird.

(iii) Der Semaphor von $ATO_m$ untersucht, ob $op_i$ frei ist.

(iii.i) Wenn $op_i$ frei ist, dann markiert der Semaphor des $ATO_m$ die Operation $op_i$ als belegt und setzt den AM mit $op_i$ in Verbindung. $op_i$ wird dann unter Kontrolle des AM ausgeführt. Wenn die Operation $op_i$ ihre Aufgabe zum Ende gebracht hat, signalisiert der AM dem Semaphor des $ATO_m$, daß $op_i$ nun frei ist.

(iii.ii) Wenn $op_i$ belegt wird, dann muß der AM warten.

Sowohl der AM als auch die Semaphore müssen in der Prozeßkommunikationssprache implementiert werden.

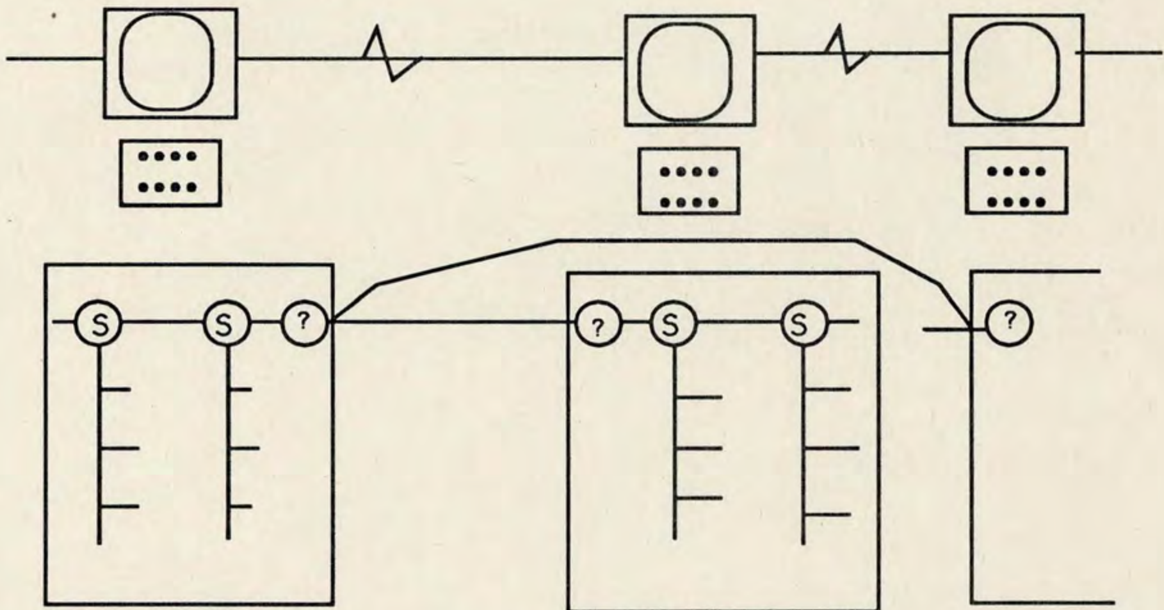
## 5. Verteiltes PROSOFT

Diese Eigenschaft bedeutet, daß PROSOFT auf verschiedenen Maschinen gleichzeitig laufen kann. Dies bietet den Vorteil,

(i) daß die ATOs auf mehrere Maschinen verteilt sind, also eine Optimierung des Verfahrens, und

(ii) daß die Benutzer miteinander kommunizieren können als Voraussetzung für kooperative Arbeit.

Wie das Kommunikationsprotokoll im einzelnen aussieht, ist eine Frage der Forschung. Das Kommunikationsprotokoll wird daher in der unten stehenden Figur mit ? gekennzeichnet und sollte in der Prozeßkommunikationssprache implementiert sein.




## 6. Das Problem der graphischen Darstellung

Ein ATO ist eine Sammlung von Operationen

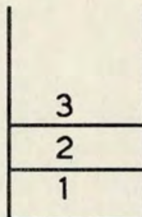
$$f_i: K_1\ K_2 \ldots K_m \text{->} K_n$$

wobei $K_i$ Klassen sind. Die Operationen erzeugen/ändern Elemente oder Werte der Klasse des ATOs. Elemente der Klassen werden in der PROSOFT-Terminologie "Objekte" genannt. Sind die Operationen algebraisch spezifiziert,

so erzeugen/ändern sie den "Term" der Klasse oder den einer Sub-Klasse $K_n$. Diese Terme müssen eine graphische Darstellung besitzen.
Beispiel:

push(push(push(newstack,1),2),3)

ist ein Term der Klasse (Sort) Stack. Wenn das das Resultat der Anwendung einer Operation ist, wäre es wünschenswert, wenn dieser Term eine andere Darstellung hätte.
Beispiel:




Um das zu erreichen, muß man die Spezifikation des Typs Stack, wie unten ansatzweise gezeigt wird, algebraisch implementieren:

```
PUNKT
Sort: Real, Punkt, Boolean
Operationen:
punkt: Reale Reale -> Punkt
X-Axis: Punkt        -> Reale
Y-Axis: Punkt        -> Reale
>: Punkt Punkt -> Boolean
Axioms:
Variable: x,y:Reale
X-Axis(punkt(x,y)) = x
Y-Axis(punkt(x,y)) = y
>(punkt(x1,y1),punkt(x2,y2))= if x1>x2
                                 then true
                                 else if y1>y2
                                         ^x1=x2
                                         then true
                                         else false
End PUNKT
```



UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

SEGMENT
Sort: Punkt, Segment
Operation:
segment: Punkt Punkt -> Segment
zweiterpunkt: Segment -> Punkt
ersterpunkt: Segment -> Punkt
Axioms:
Variable: p1,p2:Punkt
zweiterpunkt(segment(p1,p2)) = p2
ersterpunkt(segment(p1,p2)) = p1
End SEGMENT

LISTEVONSEGMENTEN
Sort: List_seg, Segment
Operation:
newlist: -> List_seg
Append: List_seg Segment -> List_seg
.......

End LISTEVONSEGMENTEN

newstack':=
append(newlist,segment(ersterpunkt(segment(punkt(x2,y2),zweiterpunkt(
segment(punkt(x1,y1),punkt(x1,y2))))),punkt(x2,y1)))

wobei newstack'

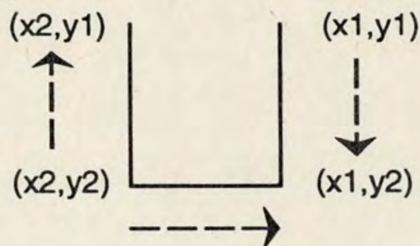
newstack': -> List_seg

ist die Implementierung der Operation newstack

newstack: -> Stack.

Wie man sehen kann, ist Stack als ein List_seg implementiert worden. Ob die Implementierung die Spezifikation erfüllt, muß man beweisen [Ber91],[Hor89].

Wenn die Operationen segment, ersterpunkt, zweiterpunkt und punkt in einer Programmiersprache implementiert worden wären, dann würde newstack' auf dem Bildschirm etwa so aussehen:



Es ist klar, daß die Reduzierung des äußersten Terms

```
segment(ersterpunkt(segment(punkt(x2,y2),zweiterpunkt(
                    segment(punkt(x1,y1),punkt(x1,y2))))),punkt(x2,y1)))
```

beispielsweise durch ein regelbasiertes Ersetzungssystem, ergibt:

```
segment(punkt(x2,y2),punkt(x2,y1)).
```

Dieses Resultat ist aber inkorrekt, da die anderen Segmente nur als Zwischenergebnisse auftreten, während in Wirklichkeit alle drei Segmente **zusammen** das Resultat bilden.

Noch komplizierter ist die Implementierung der Operation push. Wenn der Term

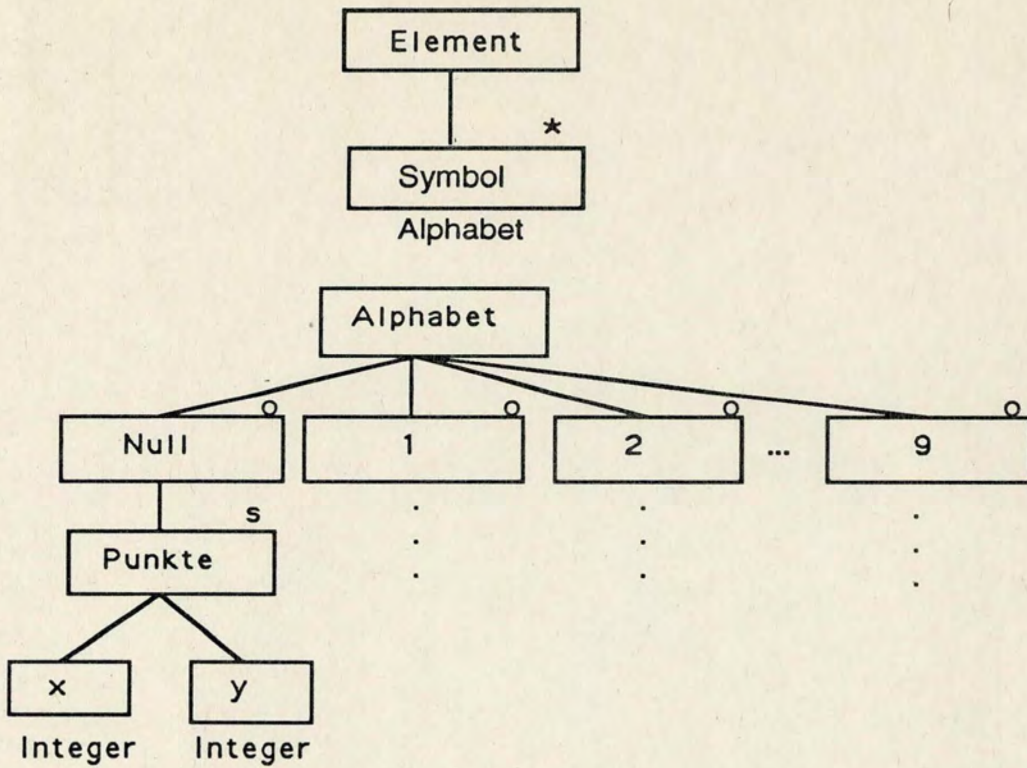
```
push(push(push(newstack,1),2),3)
```

auch eine graphische Darstellung hätte, wie oben gezeigt, dann müßten nicht nur die Segmente gezeichnet werden, sondern auch deren Symbole 1,2 und 3.

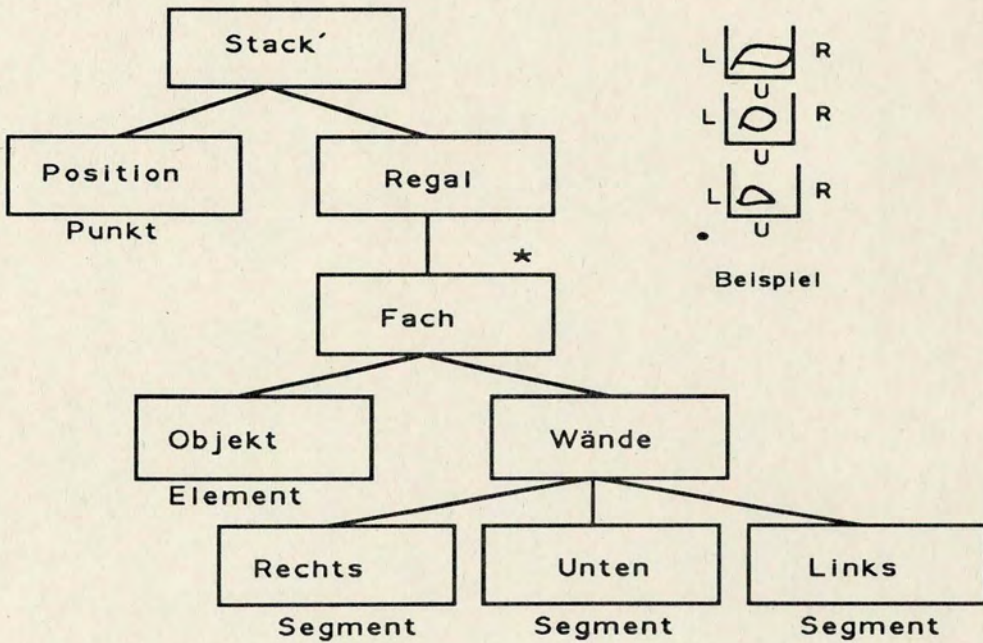
Stellen wir uns vor, die Symbole 1,2,...9 seien durch eine Menge von Punkten charakterisiert:

```
zero       -> append(append...punkt(.)),punkt(.))
succ(zero) -> append(append...punkt(.)),punkt(.)) ...
```

Ferner sei Alphabet eine Menge von Symbolen, und Element sei eine Liste von Symbolen. Diese Strukturen lassen sich in PROSOFT wie folgt als Klassen darstellen:




Stack wird als ein Register (Stack´) implementiert. Diese Klasse wird wie folgt dargestellt:



Es sei vorausgesetzt, daß die Klasse Segment schon existiere.

Die Implementierung von newstack

newstack: -> Stack

ist newstack':

newstack': Punkt -> Stack'

newstack':= make(punkt(x1,y1),newlist)

wobei make eine Operation zur Erzeugung eines Records ist.

Die Operation push

push: Stack Nat -> Stack,

wird durch push´

push': Stack' Element -> Stack',

implementiert.

Die Spezifikation von push´ ist deswegen schwierig, weil man die graphische Repräsentation ganz genau beschreiben müßte.

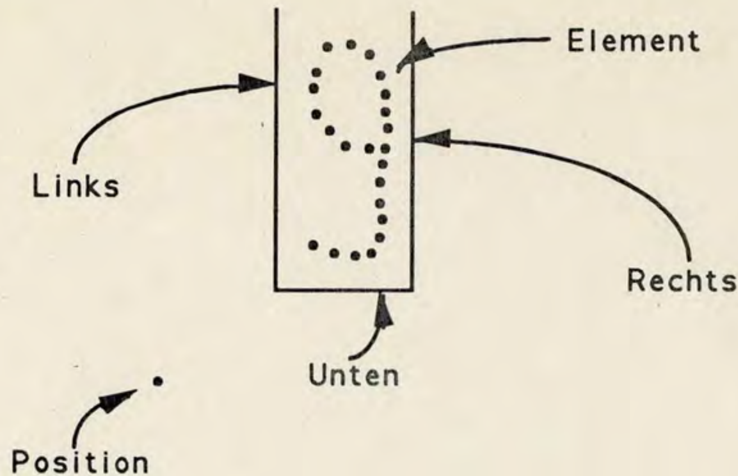
Die Reduktion des Terms

push'(st',el) ,

wobei st' leer ist, soll liefern:

```
make(punkt(x1,y1),
        append(make(newlist,
        make(el´,make(segment(...),segment(...),segment(...))))) ,
```

wobei el' die Anpassung des Elements el an die Position punkt(x1,y1) ist, und segment(...) die drei Segmente darstellt, die wir durch die Operation push' erzeugen müssen. Wenn el 9 ist, und st´ newstack´ ist, dann wird stack´ wie unten dargestellt:




## 7. Schlußfolgerungen

Um eine bessere Software-Qualität zu erreichen, sollten die nachstehenden Punkte erfüllt sein:

(i) Die ATOs müssen algebraisch spezifiziert werden. Ein algebraischer Text-Editor dafür existiert bereits, benötigt wird noch ein Werkzeug, das die Konzistenz der Spezifikation überprüft.

(ii) Wenn die Spezifikation eine algebraische Implementierung haben soll, so muß die Implementierung anhand der Spezifikation überprüfbar sein. Es ist also noch ein Werkzeug erforderlich, das auch diese Konzistenz überprüft.

(iii) Die Spezifikation selbst oder ihre algebraische Implementierung muß in der Prozeßkommunikationssprache implementiert werden. Benötigt wird eine Grammatik für diese Sprache, ein Editor, eine Operationale Semantik und ein Interpreter. Dazu ist ein Werkzeug nötig, das überprüft, ob die Implementierung die Spezifikation erfüllt.

(iv) Ferner muß ein "Loader" verfügbar sein, der ein ATO mit Semaphoren und Aufrufmechanismus (AM) produziert, sowie ein Linkeditor, der das neue ATO mit den anderen bindet.

## 8. Danksagung

Prof. Gunzenhäuser bin ich zu besonderem Dank verpflichtet dafür, daß er die wissenschaftliche Tätigkeit in Rahmen dieses Forschungsaufenthaltes ermöglicht hat. Mein Dank gilt ferner Herrn Schlebbe für fachliche Diskussionen und private Betreuung, Prof. Lagally für die Unterstützung seiner Abteilung sowie Prof. Ludewig für die Bereitstellung von Arbeitsgerät. Bei den Professoren Ehrig (TU-Berlin), Jähnichen (GMD Berlin), Prof. Neuhold (GMD Darmstadtt), Prof. Lauber

(TU-Stuttgart) und Prof. Rembold (TU-Karlsruhe) möchte ich mich für die Vortrageinladung und für ihre Anregungen zum Projekt PROSOFT bedanken.

**9. Literatur**


[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall. 1989

[Plo81] G.D.Plotkin. *A Structural Approach to Operational Semantics.* Computer Science Department. Aarhus University, Denmark. 1981

[Ber91] J.A. Bergstra. *Algebraic Methods II. Theory, tools and aplications.* Berlin: Springer. 1991.

[Hor89] I. Horebeek. *Algebraic Specifications in Software Engineering.* Berlin: Springer. 1989

[Nun 92] D.J.Nunes. *Estrategia Data-Driven no Desenvolvimento de software.* VI Simposio Brasileiro de Engenharia de Software. Sociedade Brasileira de Computacao. Proceedings 1992

[Coh86] B. Cohen et. al. *The specification of complex Systems* . Wohingham Addison-Wesley, 1986.

[Jon80] Clifford B. Jones. *Software developmente: a rigorous aproach.* Englewood Cliffs, N. F., Prentice Hall International, 1980.

[Nun85] D. J. Nunes. *Ein Verfahren zur rechnerunterstützten Programmkonstruktion.* Dissertation. Universität Stuttgart. 1985